

Osijek, 2022.

BACKEND

PROGRAMER

Priručnik

Sva prava pridržana. Niti jedan dio ove knjige ne smije se reproducirati ili prenositi u bilo kojem obliku, niti na koji način. Zabranjeno je svako kopiranje, citiranje te upotreba knjige u javnim i privatnim edukacijskim organizacijama u svrhu organiziranih školovanja, a bez pisanog odobrenja autorskih prava.

Copyright© RCK ELPROS

BACKEND PROGRAMER

Priručnik

Algebra d.o.o.

Osijek, 2022.

„Sadržaj publikacije/emitiranog materijala isključiva je odgovornost RCK ELPROS“



Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.

Sadržaj:

1. POGLAVLJE: Pregled tehnologije i osnovni pojmovi	8
1.1. Relevantne tehnologije i terminologija	8
1.1.1. Osnovni pojmovi	9
1.1.2. Programi i programski jezici	10
1.1.3. Protokoli	12
1.1.4. Licence i autorska prava	14
1.2. Organizacija klijent – poslužitelj arhitekture	15
1.2.1. Tehnologija na strani klijenta: frontend	16
1.2.2. Tehnologija na strani poslužitelja: backend	17
1.3. Servisi i serveri za komunikaciju između klijenta i servera	17
1.3.1. Web-servisi	17
1.3.2. Web-serveri	18
1.4. Alati za razvoj	19
1.5. Verzioniranje koda	20
1.5.1. GIT	21
1.5.2. Lokalni repozitorij	22
1.5.3. Distribuirani repozitorij	23
1.5.4. Servisi za verzioniranje koda	23
1.5.5. SVN	24
2. POGLAVLJE: Baze podataka – MS SQL	25
2.1. Uvod	25
2.2. Baze podataka	26
2.2.1. Direktne datoteke	26
2.2.2. Sustav za upravljanje bazom podataka	27
2.3. Dizajn baze podataka	29
2.4. Radna okolina	30
2.4.1. MS SQL server	30
2.5. Modeliranje podataka	35
2.5.1. Entiteti i veze	35
2.5.2. ER dijagram	36
2.6. Relacijski model	37
2.6.1. Primarni ključ	38
2.6.2. Pretvorba entiteta i veza u relacije	39
2.6.3. Normalizacija	41
2.7. SQL jezik	43
2.7.1. Ključne riječi	45
2.7.2. Sintaksni dijagrami	45
2.8. Tipovi podataka	47
2.8.3. Datum i vrijeme	50
2.8.2. Približni brojevi	50
2.8.4. Znakovi i nizovi znakova	51
2.8.5. Binarni nizovi znakova	52
2.8.6. Ostali podatkovni tipovi	53
2.9. Upravljanje strukturom baze	54
2.9.1. Stvaranje objekata CREATE naredbom	54
2.9.2. Izmjena objekata ALTER naredbom	55
2.9.3. Brisanje objekata DROP naredbom	56

2.9.4. Sistemske baze podataka SQL servera	57
2.9.5. Ograničenja	58
2.10. Manipulacija podacima	62
2.10.1. Ubacivanje	62
2.10.2. Izmjena	64
2.10.3. Brisanje	65
2.11. Dohvat podataka iz baze	68
2.11.1. Filter upita	69
2.11.2. Operatori	70
2.11.3. Grupiranje rezultata upita	75
2.11.4. Odabir 'naj' rezultata	76
2.11.5. Sortiranje rezultata upita	76
2.11.6. Upiti nad više tablica	77
2.11.7. Izrada tablica kao rezultat upita	81
2.12. Ostale naredbe	82
2.12.1. Naredbe pristupa	82
2.12.2. Transakcije	84
2.13. Sigurnosne kopije	86
2.14. Pogledi	89
2.15. Funkcije	91
2.15.1. Ugrađene funkcije	91
2.16. Procedure	98
2.17. Okidači	100

3. POGLAVLJE: Programski jezik C#	102
3.1. Uvod	102
3.2. Razvojno programsko okruženje	105
3.2.1. Visual Studio	106
3.2.2. Instalacija Visual Studio alata	108
3.2.3. Prilagodba sučelja Visual Studio	110
3.2.4. Konzolna aplikacija	111
3.2.5. Osnovna pravila pisanja koda	120
3.3. Varijable i tipovi podataka	123
3.3.1. Tipovi podataka	123
3.3.2. Varijable	128
3.3.3. Konverzije podataka	132
3.4. Operatori	133
3.5. Grafički prikaz algoritma	138
3.6. Kontrola toka programa grananjem	139
3.6.1. Naredba if	140
3.6.2. Naredba switch	144
3.7. Kontrola toka programa petljama	145
3.7.1. Programska petlja for	146
3.7.2. Programska petlja while	148
3.7.3. Programska petlja do ... while	149
3.7.4. Ugnježđivanje petlji	150
3.7.5. Naredbe kontrole	151
3.7.6. Naredba goto	153
3.8. Polja	154
3.8.1. Jednodimenzionalna polja ili nizovi	154

3.8.2. Petlja foreach	158
3.8.3. Sortiranje	159
3.8.4. Višedimenzionalna polja	166
3.9. Funkcije	167
3.9.1. Prosljeđivanje parametara	172
3.9.2. Rekurzivne funkcije	175
3.10. Objektno orijentirano programiranje	177
3.10.1. Klase i objekti	178
3.10.2. Enkapsulacija	182
3.10.3. Događaji	183
3.10.4. Nasljeđivanje	185
3.10.5. Polimorfizam	188
3.10.6. Otvorena rekurzija	192
4. POGLAVLJE: .NET razvojni okvir	193
4.1. Razvojni okvir	193
4.2. Rad s iznimkama	198
4.2.1. Upravljanje iznimaka	200
4.2.2. Korištenje vlastitih iznimaka	206
4.3. Rad sa stringovima	208
4.3.1. String literali	210
4.3.2. String izlazne sekvence	214
4.3.3. Format stringova	214
4.3.4. Pristup individualnim znakovima	216
4.3.5. Null stringovi i prazni stringovi	217
4.3.6. stringBuilder klasa	218
4.3.7. Operatori i stringovi	219
4.3.8. Metode i svojstva string klase	220
4.4. Rad s datotekama	223
4.4.1. Klase za čitanje i pisanje	224
4.4.2. Klase za rad s datotečnim sustavom	229
4.5. Strukture podataka	234
4.5.1. Kolekcije objekata	235
4.5.2. Generičke kolekcije	238
4.6. Pretvorbe podataka	253
4.6.2. JSON	263
4.7. Interakcija s korisnikom	273
4.7.1. Aplikacije s tekstualnim sučeljem	274
4.7.2. Desktop sučelja	275
4.7.3. Web-sučelja	290
4.8. LINQ	313
4.8.1. From blok	314
4.8.2. Where blok	315
4.8.3. Select blok	315
4.8.4. Povrat zapisa	315
4.8.5. Sortiranje podataka	316
4.8.6. Dodatne mogućnosti	316
4.9. Datum i vrijeme	317
4.9.1. TimeSpan	321
4.10. Enumeracije	323

4.11. Sučelja	324
5. POGLAVLJE: C# i .NET napredno programiranje	327
5.1. Integracija dodatnih biblioteka	327
5.1.1. NuGet	328
5.1.2. PdfSharp	331
5.1.3. Newtonsoft.Json	333
5.1.4. GtkSharp	334
5.2. ADO.NET i Entity Framework	335
5.2.1. ORM	335
5.2.2. ADO.NET	335
5.2.3. Entity Framework	344
5.3. Višedretveno programiranje	356
5.3.1. Sinkrono programiranje	356
5.3.2. Asinkrono programiranje	364
5.4. Parcijalne klase	370
5.4.1. Nesiguran kod	373
5.5. Web-servisi	377
5.5.1. WCF	377
5.5.2. WEB API	385
5.6. Validacija	385
5.6.1. Validacija na strani poslužitelja	386
5.6.2. Klijentska validacija	387
5.7. Sigurnost korisničkih podataka	387
5.7.1. Autentikacija ili autentifikacija	388
5.7.2. Autorizacija	389
5.8. Isporuka aplikacija	389
5.8.1. Debug i release koncepti	389
5.8.2. Distribucija razvijenog programskog rješenja	391
6. POGLAVLJE: Predlošci izrade programskih rješenja	394
6.1. Predlošci kreiranja objekta	395
6.1.1. Abstract Factory	395
6.1.2. Builder	401
6.1.3. Factory Method	405
6.1.4. Object pool	409
6.1.5. Prototype	413
6.1.6. Singleton	416
6.2. Predlošci struktura	421
6.2.1. Adapter	421
6.2.2. Bridge	423
6.2.3. Composite	426
6.2.4. Decorator	431
6.2.5. Facade	435
6.2.6. Flyweight	438
6.2.7. Private class data	443
6.2.8. Proxy	443
6.3. Predlošci ponašanja	447
6.3.1. Chain of responsibility	447
6.3.2. Command	452

6.3.3. Interpreter	456
6.3.4. Iterator	456
6.3.5. Mediator	461
6.3.6. Memento	465
6.3.7. Null Object	471
6.3.8. Observer	471
6.3.9. State	476
6.3.10. Strategy	479
6.3.11. Template method	483
6.3.12. Visitor	487
6.4. Predlošci dizajna programskih rješenja	487
6.4.1. MVC	487
6.4.2. MVP	489
6.4.3. MVVM	491
6.4.4. MVW	492
6.4.5. MVI	492
6.4.6. MHVC	493
6.4.7. MMV	494
6.4.8. MVA	494
7.POGLAVLJE: WEB API	495
7.1. REST	496
7.2. Serijalizacija i deserijalizacija	497
7.2.1. XML i JSON	498
7.3. WEB API	498
7.3.1. WEB API projekt	499
7.3.2. Kontroler	502
7.3.3. Akcije	504
7.4. Mapiranje složenih podataka	507
7.4.1. Mapiranje HTTP metoda	508
7.4.2. Čitanje i pisanje u bazu podataka	512
7.5. Povezivanje s aplikacijom	515
8.POGLAVLJE: Automatizirano testiranje	519
8.1. Testiranje koda	519
8.2. Unit test u MVC aplikacijama	530
8.2.1. Testiranje kontrolera	531
8.2.2. Testiranje podataka	535
8.2.3. Testiranje akcija	536
8.3. Mjerenje uspjeha i stvaranje izvješća	538
9.POGLAVLJE: Zaštita na radu	540
9.1. Osnove zaštite na radu	540
9.2. Opasnosti i način zaštite na radnom mjestu	542
9.3. Ergonomija i mjere prevencije pri radu s računalom	546

1. POGLAVLJE:

Pregled tehnologije i osnovni pojmovi

U OVOM POGLAVLJU NAUČIT ĆETE:

- pojmove vezane uz poslužitelje
- organizaciju klijent – poslužitelj arhitekture
- razlikovati servise i servere
- upoznati se s osnovnim alatima razvojnih okruženja
- upoznati se s alatima za verzioniranje koda.

1.1. Relevantne tehnologije i terminologija

Računalo je u svojoj osnovi najobičniji kalkulator koji jako brzo i automatizirano uglavnom računa. Osobno računalo u osnovi je multifunkcijsko računalo, koje je zbog svoje veličine i cijene dostupno za pojedinačnu upotrebu. Ovakva definicija računala rudimentarna je i možda se čini zastarjelom te danas manje-više gubi značaj, jer je nastala krajem 1970-ih i početkom 1980-ih.

Zašto počinjemo s time? Jer u svakom dobu imamo drukčije shvaćanje računala. Uz današnje tehnologije i pristupačnost pojam računala postao je fluidan u društ-

vu. Većina korisnika ni ne shvaća da su današnji pametni satovi, IoT uređaji i slično sami po sebi računala. Danas nije kao prije problem smanjiti računalo na neku manju razinu, ali postoji problem komunikacije računala, odnosno načina na koji pojedina računala komuniciraju s korisnicima.

1.1.1. Osnovni pojmovi

Osobno računalo u svojoj je suštini stroj koji se sastoji od sklopovlja (*hardware*) i programa (*software*). Ako gledamo tako, sklopovlje bi bilo fizički dio računala, onaj koji možemo opipati i direktno vidjeti. Iako zbog kompleksnosti određenih komponenti sigurno ne možemo vidjeti sve dijelove samo tako, no svakako fizički postoje. Programi su suprotno od toga, oni su točno određen skup instrukcija koje tvore naredbe, koje opisuju korake izvršavanja nekog posla. Programe fizički ne možemo vidjeti, oni su skup podataka zapisan negdje u memoriju.

Napravit ćemo samo kratki pregled često korištenih pojmova:

Web-stranica je *web*-dokument pogodan za prikaz u internetskom pregledniku. *Web*-stranica u biti je ono što vidimo u *web*-pregledniku, a često je napravljena kombiniranjem nekih od tehnologija za *web*-dizajn (HTML, CSS, JavaScript...).

Web-sjedište predstavlja polazišnu točku prema drugim odredištima i aktivnostima na internetu. *Web*-sjedište predstavlja skup *web*-stranica u vlasništvu nekog subjekta. Danas često možemo reći da je *web*-sjedište s programerskog gledišta *web*-aplikacija koja se nalazi na nekom poslužitelju.

Poveznica ili link jedinstvena je adresa nekog resursa na internetu. Prilikom pristupa bilo kojoj stranici u adresnoj traci preglednika imamo adresu ili link, što predstavlja adresu traženog resursa ili stranice.

Web-preglednik program je koji nam služi za pregledavanje *web*-stranica. Preglednici su programi poput Microsoft Edge, Google Chrome, Mozilla Firefox, Opera, Safari, Netscape i drugi.

Web-pretraživači programi su ili sustavi koji nam služe za pretraživanja pojmova na internetu. Kao rezultat pretrage daju nam popis stranica koje sadrže traženi pojam. Pretraživači su: Google, Bing, Yahoo i drugi.

URL ili Uniform Resource Locator adresa je preko koje pristupamo nekom resursu ili ga dobivamo. Ovisno o tome kako ćemo napisati url on može biti lokalni url za pristup resursima na lokalnom računalu ili u obliku poveznice na neku *web*-stranicu ili *web*-servis.

Cookies ili kolačići podaci su koje pojedina stranica pohranjuje na računalo klijenta koji pristupa stranici. Mogu biti kolačići koji su u vlasništvu stranice kojoj trenutno pristupamo i kolačići koje rade dodaci *web*-stranici (naprimjer Google ukoliko koristite mape ili analitiku).

Hosting je zakup poslužiteljskog prostora preko kojeg poslužujemo našu aplikaciju na internetu.

1.1.2. Programi i programski jezici

Program je, najkraće rečeno, povezan slijedan skup naredbi koje se izvršavaju na računalu. Kraće rečeno – sve što zovemo *software* jest program ili skup programa.

Programe možemo podijeliti s obzirom na nivo izvršavanja na:

- *firmware*
- operativni sustav
- systemske programe
- korisničke programe.

Firmware je najniži oblik programa, često zvan i sklopovski program. Primjer naprednog *firmwarea* je BIOS, a u mikroupravljačima i IoT uređajima za program koji radimo za njih isto možemo reći da je *firmware* program. *Firmware* programi izvršavaju se automatski prilikom uključivanja uređaja i to je najosnovniji program.

Operativni sustavi programi su koji nama korisnicima pružaju mogućnost rada, često pojednostavljenu, s osnovnim ulaznim i izlaznim operacijama. Preko operativnog sustava možemo otvarati datoteke i manipulirati njima, ali također i pokretati druge programe. Primjeri operativnih sustava: Microsoft Windows, Linux, MacOS, Solaris, UNIX...

Sistemske programe programi su koji se automatski pokreću u operativnom sustavu, najčešće bez korisnikova znanja. U ovu kategoriju programa možemo ubrojiti *drive* (pogonske programe) i razne servise operativnog sustava.

Korisnički programi najraširenija su skupina programa. Ovdje praktički spada svaki program koji možemo pokrenuti i ima neko sučelje. U korisničke programe možemo nabrojati *web*-preglednike, *office* alate, videoigre, programe za rad poput raznih uređivača sadržaja i poslovnih aplikacija itd.

Programe rade programeri korištenjem programskog jezika. **Programski jezik** skup je pravila i simbola kojima se piše neki program. Pravila pisanja ovisit će o samom programskom jeziku koji se koristi i o vrsti programa koji radimo. Postoje tisuće programskih jezika, a svake godine nastane još nekoliko novih.

Podjela programskih jezika nije lagana, no ovo je najčešća:

- niži programski jezici:
 - strojni kod
 - *assembleri*
- viši programski jezici:
 - direktno prevedeni jezici
 - skriptni jezici.

Niži programski jezici najstariji su i oni se izvode skoro direktno na instrukcijskoj razini procesora. Strojni kod najbolji je primjer toga, jer to je program koji je prilagođen točno određenom instrukcijskom setu procesora. Asemblerski jezici oni su koje treba prevesti u strojni kod da bi se program mogao pokrenuti. Niži programski jezici najbrži su i najoptimiziraniji što se tiče izvođenja, no najkompliciraniji i

najsporiji što se tiče razvoja novih programa.

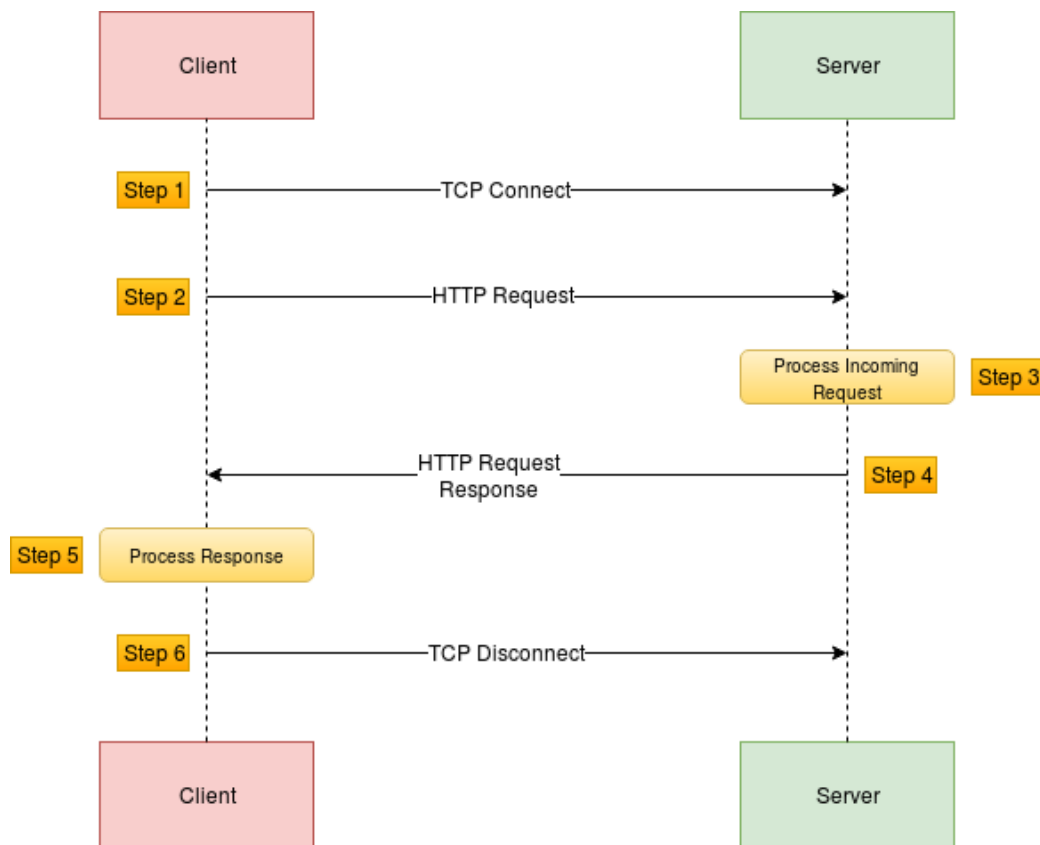
Viši programski jezici danas se uglavnom koriste jer pojednostavljaju proces pisanja programa. No budući da u toj skupini ima više programskih jezika nego u nižim, razlike među pojedinim programskim jezicima ogromne su. Te razlike postoje u pisanju, izvođenju, namjeni i potrebnim resursima. U skupinu viših programskih jezika spadaju C, C++, Java, C#, Basic, Cobol, JavaScript, Python, F#, R, Objective-C itd.

Podjela koju smo napravili kod viših programskih jezika jest na one koji se prevode u drugi programski jezik ili u strojni jezik za optimizirano izvođenje prije nego se pokrenu i skriptne programske jezike gdje se prevođenje i izvršavanje događa istovremeno tijekom samog izvršavanja programa. Generalno gledano, skriptni programski jezici sporiji su i uzimaju više resursa, a ovisno o potrebama ne govorimo o nekim sitnim postocima, nego i o nekoliko stotina puta lošijim karakteristikama.

1.1.3. Protokoli

Protokol je standardizirani način komunikacije između dva subjekta. Kod računala protokol je način na koji će komunicirati dva sustava međusobno. Imamo više različitih protokola, čije korištenje ovisi o upotrebi i o zahtjevima sustava.

Najčešće korišteni protokol jest *HTTP* protokol (*HyperText Transfer Protocol*). Služi za prijenos podataka na temelju zahtjev – odgovor metodike na klijent – poslužitelj modelu. HTTP protokol jest protokol aplikacijskog sloja povrh TCP protokola (*Transmission Control Protocol*). Bez stanja je na aplikacijskoj razini i treba pouzdan mrežni transportni protokol da bi se vršila razmjena podataka između poslužitelja i klijenta. U http implementaciji TCP/IP veza koristi se korištenjem poznatog porta (80 ako veza nije enkriptirana, a 443 ako je veza enkriptirana). Klijent se inicijalno pokušava povezati na poslužitelj uspostavom veze. Poslužitelj sluša unaprijed dogovoreni port čekajući poruku klijenta. Klijent šalje zahtjev poslužitelju (*HTTP Request*), koji kad primi zahtjev pošalje nazad http odgovor (*HTTP Response*), koji se sastoji od zaglavlja i tijela. Tijelo poruke tipično je odgovor na zahtjev (tražena stranica ili podatak), ali može se poslati i poruka u slučaju greške, ili neka druga korisna informacija.



Slika 1. Prikaz rada HTTP protokola

HTTPS je nadogradnja HTTP protokola, gdje S znači *secure* ili sigurno. Sigurnost se dobiva enkripcijom koristeći TLS ili SSL enkripciju. Zato se prvotno zvao HTTP over TLS ili HTTP over SSL. Motiv za uvođenje HTTPS-a je sigurnost, pogotovo kad se putem *web*-stranica prenose osobni podaci i podaci osjetljive prirode.

Za **SSL** i **TLS** već smo spomenuli da su kriptografski protokoli. Dizajnirani su da pruže dodatnu sigurnost pri prijenosu podataka u računalnim mrežama. Sigurnost u smislu da spriječe mogućnost prisluškivanja, odnosno presretanja paketa prijenosa podataka u svrhu njihova direktnog čitanja i manipulacije. TLS je je unaprijeđena verzija SSL-a.

DNS (Domain Name System) je registar domena. Da bismo komunicirali s računalima, trebamo znati njihovu IP adresu. No IP adrese brojevi su koji nisu često pamtljivi. DNS je nastao da bismo umjesto brojčane adrese računala mogli koristiti imensku adresu računala u svrhu lakšeg i bržeg pristupa. Npr. prilikom pristupa stranici na adresi 142.250.201.206 jednostavno upišemo google.com za adresu stranice.

1.1.4. Licence i autorska prava

Programska licenca legalni je instrument koji govori o pravima korištenja i distribucije programa. Programe možemo zaštititi u obliku zaštite izvornog koda i finalnog proizvoda. Autori programa program i kod mogu donirati da bude javni i tad nije podložan nikakvoj vrsti licenciranja.

Tipična programska licenca daje korisniku dozvolu da koristi jednu ili više kopija programa na način koji je propisan u uvjetima korištenja programa.

Ovaj dio više pripada domeni pravnika nego programera, no ono čega moramo biti svjesni kao programeri i kao korisnici jest to da programe i kod koji su zaštićeni nekom licencom moramo koristiti u okviru te licence – inače kršimo zakon.

Licence možemo podijeliti generalno na dvije kategorije:

- vlasničku
- besplatnu ili open source.

Ovo se odnosi na prava pružanja mogućnosti korištenja, modifikacije i distribucije programskog rješenja, ali i izvornog koda.

Open source ili otvoreni kod znači da licence praktično nema, odnosno kompletan program i kod možemo koristiti bez ikakvih pravnih prepreka.

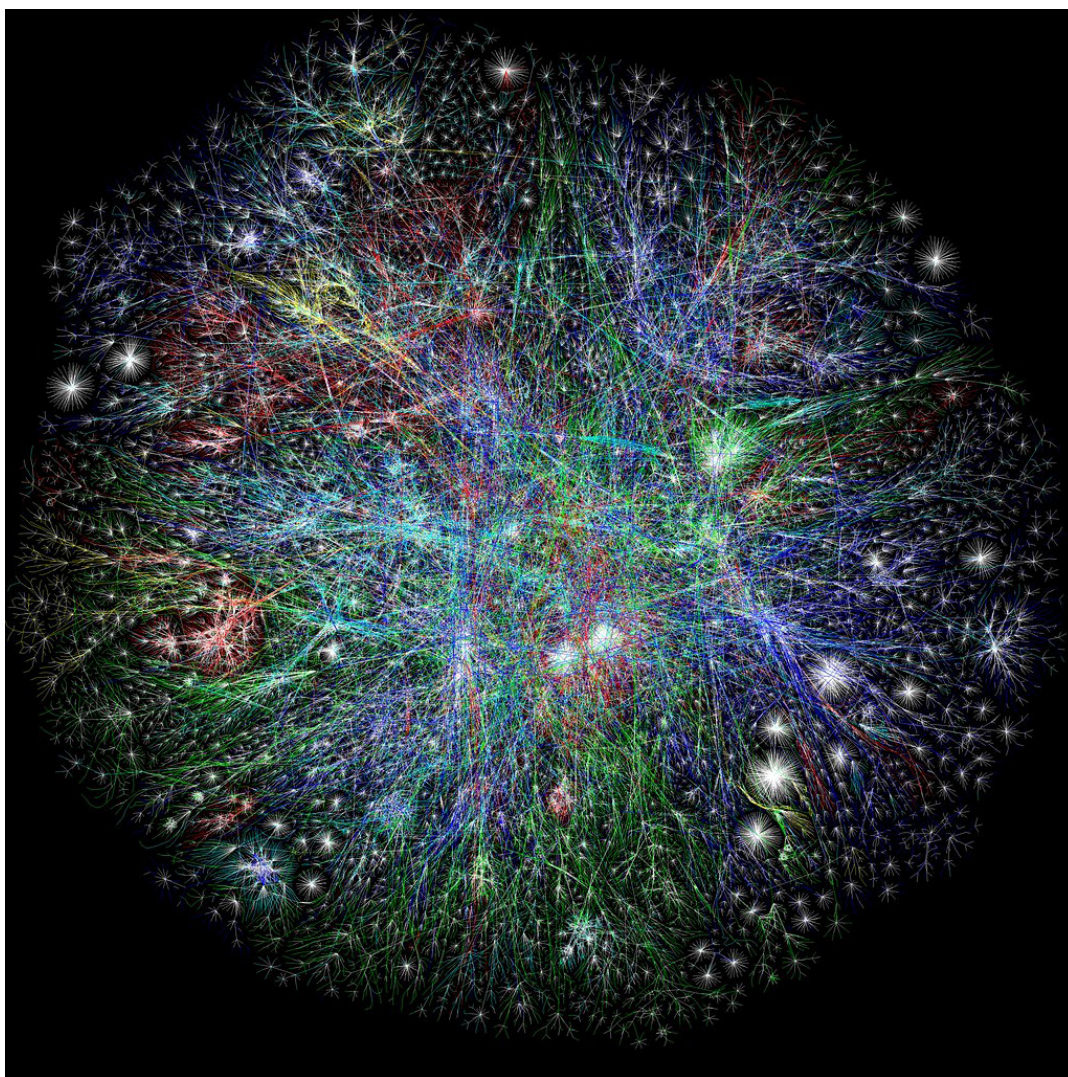
Ukoliko program ili izvorni kod imaju neku licencu, to je malo kompliciranije, gdje prava korištenja moramo tražiti preko proizvođača i uklopiti ih prema njihovim specifikacijama načina korištenja.

Generalno gledano, program je autorsko djelo programera, jednako kao što je i napisana pjesma autorsko djelo skladatelja ili tekstopisca. Time prava korištenja programa i samog koda ovise o programeru, odnosno o tvrtki u kojoj radi, a gdje je sve definirano nekom vrstom ugovora.

1.2. Organizacija klijent – poslužitelj arhitekture

Danas je većina interneta bazirana na klijent-poslužiteljskoj arhitekturi. Već smo spomenuli protokole, gdje se klijent povezuje s poslužiteljem i šalje zahtjev te na temelju zahtjeva poslužitelj šalje odgovor. To je ukratko način na koji komuniciraju poslužitelj i klijent.

U stvarnosti ovaj opis, iako potpuno istinit, shvaća se prejednostavno. Internet možemo gledati kao mrežu svih mreža gdje je sve povezano. Zato internet danas gledamo kao isprepletenu mrežu jako velikog broja poslužitelja i još većeg broja klijenata.



Slika 2. Vizualizacija interneta

Slika predstavlja vizualiziranu povezanost računala na internetu, gdje je svaka linija veza između računala, a točke koje ne možemo ni izbrojati pojedina su računala. Krajnje točke koje se ne vežu ni na što dalje su klijenti.

Poslužitelj i klijent iako komuniciraju neprestano preko određenih protokola, a tehnike izrade programa koji se izvode s klijentske i poslužiteljske strane različite su.

Zato ovisno o tome s koje je strane predviđeno izvršavanje aplikacija, aplikacije dijelimo na serverske i klijentske. Kako ime kaže, serverske aplikacije izvode se na serveru (poslužitelju), dok se klijentske izvršavaju s klijentske strane.

Češće podjelu radimo na:

- **backend aplikacijski dio**
- **frontend aplikacijski dio.**

Backend aplikacijski dio često zovemo i poslovna logika. To je dio koji korisnik nikad neće vidjeti kako radi i ni ne treba vidjeti kako radi. Kod server – klijent arhitekture to je najčešće dio aplikacije koji se izvršava isključivo na serveru.

Frontend aplikacijski dio često zovemo i prezentacijski dio. To je dio koji korisnik vidi, najčešće u obliku nekog korisničkog sučelja. U slučaju server – klijent aplikacije to je web-stranica, odnosno html koji klijent dobije da bi pregledao samu stranicu.

1.2.1. Tehnologija na strani klijenta: frontend

Na klijentskoj strani imamo mnogo tehnologija. Svaka tehnologija ovisi o vrsti aplikacije koju radimo, no ove tehnologije svode se na izradu korisničkih sučelja i prezentiranje podataka putem korisničkih sučelja.

Sa strane *web*-stranica imamo: HTML, CSS, JavaScript itd.

U desktop aplikacijama: radimo preko nekog *markupa* (poput XAML) ili direktno s programskim jezicima pomoću kojih radimo aplikacije (C#, C++, Java...).

1.2.2. Tehnologija na strani poslužitelja: backend

Sa strane izrade poslovne logike ili pozadinskog dijela aplikacije, kao i kod *frontenda* to ovisi o samoj tehnologiji izrade, odnosno o aplikaciji koju radimo. S *backend* strane ima više mogućnosti nego kod frontenda.

Za izradu poslužiteljskih dijelova *web*-stranica i servisa imamo: C# (web-forme, MVC, Core...), PHP (s svojim raznim frameworkima), C++, Python...

Za izradu ostalih aplikacija sve ovisi o tome koju vrstu aplikacija radimo. Ovdje zato doslovno možemo početi nabrajati sve moguće programske jezike pomoću kojih možemo napraviti neku vrstu aplikacije.

S *backend* strane uvijek imamo i neki sustav za upravljanje podacima, bilo da se radi o bazi podataka (SQL), radu s datotekama ili pristupu drugim servisima.

1.3. Servisi i serveri za komunikaciju između klijenta i servera

Kako smo rekli, komunikacija se odvija preko nekog komunikacijskog protokola između klijenta i poslužitelja.

1.3.1. Web-servisi

Web-servisi jesu servisi koje konzumiramo na internetu. Kad kažemo konzumiramo, mislimo na pristup servisu preko kojeg vršimo osnovne CRUD operacije (čitanje, pisanje, izmjena i brisanje). Takvih servisa ima nebrojeno u današnje vrijeme, s ve-

likim brojem protokola. Praktički skoro svaka *web*-aplikacija može biti neki servis, samo je pitanje interpretacije podataka. No kad govorimo o servisima, mislimo na pristup čistim podacima bez prezentacijske logike.

Primjer servisa bili bi RSS servisi, servisi za dohvat vijesti, servisi za čitanje tečaja, burzovnih lista, servisi koje koriste mobilne aplikacije za učitavanje podataka ili poruka itd. Ukratko, sve što danas imate na mobilnom uređaju, a treba vezu na internet, koristi *web*-servis.

1.3.2. Web-serveri

Web-serveri računala su koja su zadužena za posluživanje aplikacija klijentima. Praktično gledano, da bismo pristupili nekoj aplikaciji ili stranici, ona mora negdje biti objavljena na internetu i moramo imati adresu kako joj pristupiti.

Poslužitelje često nećemo mi sami „podizati” (osim ako ne radite u nekoj jako velikoj tvrtki kojoj treba vlastita infrastruktura), nego ćemo zakupiti neku vrstu serverskog paketa. Ako tako gledamo, imamo opcije:

- hosting paketa
- VPS-a
- dedikirani servera.

Hosting paket zakupljeni su prostor i resursi na nekom dijeljenom poslužitelju, koje ćemo koristiti za posluživanje naše *web*-aplikacije. Hosting paketi razlikuju se po prostoru za pohranu, broju domena koje možemo imati i tehnologiji na kojoj rade u smislu operativnog sustava i instaliranih biblioteka/dodataka. Odabir hosting paketa prilagođavamo tehnologiji izrade aplikacije, a moguće je i da, ako znamo tehnologije poslužitelja, u samom startu na temelju toga radimo aplikaciju.

Dedicirani server fizičko je računalo koje je spojeno na internet i konfigurirano da nam poslužuje aplikaciju. U ovom slučaju mi direktno upravljamo hardwareom samog servera ili serverskog klastera (više servera spojeno u vezu gdje su prepozna-

ti kao jedno jedinstveno računalo).

VPS ili Virtual Private Server (virtualni privatni server) način je na koji mi možemo imati virtualni server, koji će se vidjeti kako pravi server, samo vrlo vjerojatno na tom serverskom klasteru ima više virtualnih servera i aplikacija koje se izvode.

1.4. Alati za razvoj

Razvojno programsko okruženje program je koji ćemo koristiti radi lakšeg programiranja. Praktično programirati možemo pomoću bilo kojeg tekstualnog editora, samo pisati kod, a onda preko drugog programa ili naredbene trake kompajlirati ili prevesti naš kod u izvršnu datoteku ili program nekog drugog programskog jezika.

Glavne značajke razvojnog okruženja možemo podijeliti na:

- tekstualni editor
- kompajler i interpreter
- build integrator
- debugger
- označivač sintakse
- grafičko korisničko sučelje
- ostale mnogobrojne mogućnosti (podrška za odabrani jezik, uređivanje tekstualnog editora, *unit testing*, dijagrami, dodaci i proširenja, pretraga koda, završavanje koda, dijagnostički alati itd.).

Naravno, ne postoji samo jedno razvojno okruženje za sve. Razvojna okruženja razlikuju se u dosta stvari, od podržanih programskih jezika, pa sve do razine implementacije pojedinih značajki i njihove uključenosti. Od nekih popularnih razvojnih okruženja svakako bismo spomenuli:

- Visual Studio
- Visual Studio Code
- IntelliJ IDEA

- AptanaStudio
- PyCharm
- PhpStorm
- NetBeans
- Eclipse
- Komodo IDE
- XCode
- ...

Ukratko, ima mnogo razvojnih okruženja. Ovisno o korištenom programskom jeziku, vrsti aplikacije koju radite i potrebama koje imate (bilo operativnog sustava, značajki razvojnog okruženja itd.) odabrat ćete odgovarajuće razvojno okruženje. Mi ćemo koristiti Visual Studio, a o razlozima odabira ćemo kasnije, kad budemo detaljnije opisivali navedeno razvojno okruženje.

Razvojna okruženja za rad s bazama podataka:

- SQL Server Management Studio (ujedno i naš odabir kad budemo radili SQL)
- MySQL Workbench
- MariaDB
- Squirrel
- HeidiSQL
- DBeaver
- ...

1.5. Verzioniranje koda

Sustavi za verzioniranje koda jesu sustavi koji bilježe promjene na datotekama izvornog koda aplikacija kroz vrijeme razvoja kako bismo se u bilo kojem trenutku mogli vratiti na specifičnu verziju koda. Ovo nam je izuzetno korisno kad radimo na kompleksnijim projektima, pogotovo u timskom okruženju. Korištenje alata za verzioniranje koda dobra je odluka i preporučljivo je jer omogućuje da određene datoteke ili

čak cijele projekte vratimo u neko prethodno stanje prije nego što je nastao određeni problem – ukoliko nastane. Ovi sustavi također bilježe ne samo promjene, nego i tko je promjene napravio i kada.

Sustave za verzioniranje koda možemo podijeliti u 3 skupine:

- lokalne repozitorije – danas se jako rijetko koriste, repozitorij se sprema samo na računalo na kojem se radi program
- centralizirane repozitorije – repozitorij je centralno spremljen na nekom poslužitelju koji je dostupan svim korisnicima. Primjeri su SVN sustavi, Team Foundatiuon server i drugi
- distribuirane repozitorije – danas najpopularnija vrsta, bazirani su na konceptu u kojem svaki klijent sadrži kompletnu kopiju repozitorija sa svim verzijama koda. Između klijenata radi se sinkronizacija repozitorija. Primjer ovog sustava je GIT.

1.5.1. GIT

GIT je distribuirani sustav za verzioniranje koda koji promjene sprema u obliku snimaka stanja svaki put kad napravimo akciju *commit*.

U GIT repozitoriju imamo 3 osnovna stanja u kojima datoteka može postojati:

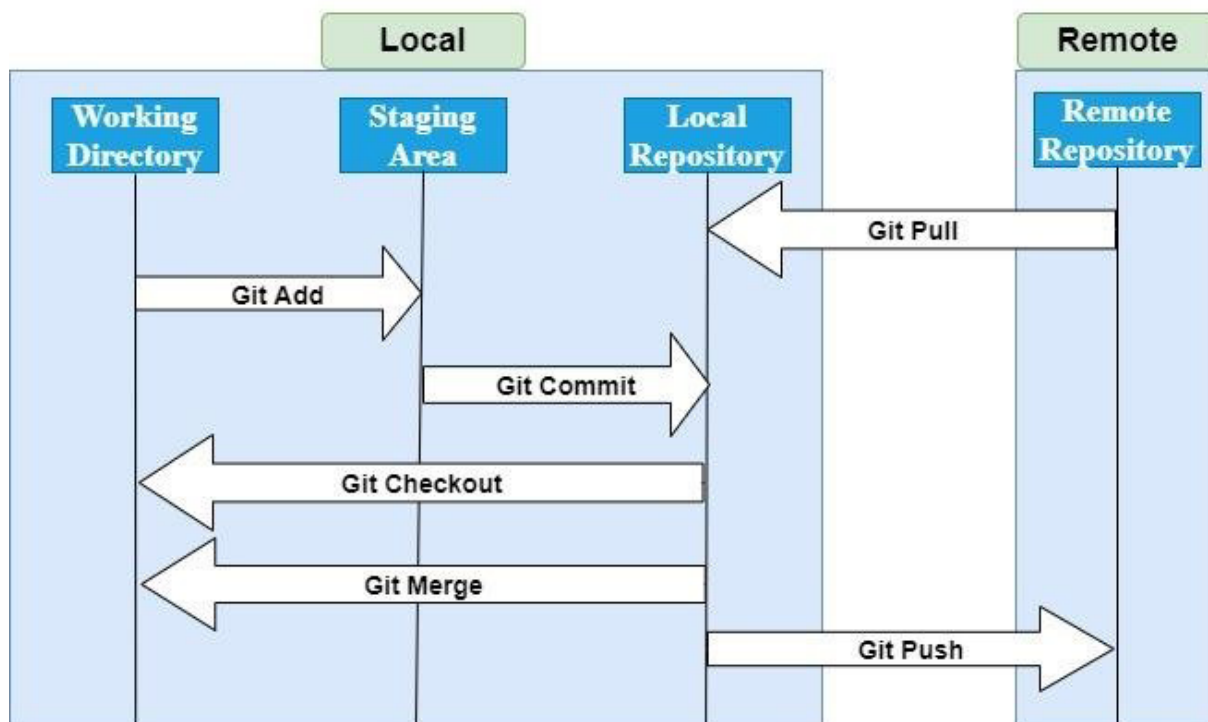
- committed – trenutna verzija datoteke uredno je spremljena u bazu
- modified – datoteka je izmijenjena, ali promjene nisu spremljene u bazi
- staged – datoteka je označena za odlazak u bazu s naredbom *commit*.

U procesu rada s GIT repozitorijem razlikujemo 3 sekcije:

- Git Directory – mjesto gdje GIT sprema metapodatke i bazu podataka projekta na kojem radimo. Ovo je centralni dio GIT-a i taj se dio kopira na druga računala
- Working Directory – lokano raspakirana verzija repozitorija, odnosno radna verzija repozitorija spremna za uređivanje i modificiranje datoteka
- Staging Area – datoteka koja sadrži informacije o tome što sve ulazi u sljedeći *commit*.

GIT je podržan i danas često integriran u većinu alata za razvoj aplikacija.

Hodogram rada s GIT repozitorijem prikazan je na slici 3.



Slika 3. Hodogram GIT-a

Preporuka je da imate otvoren račun na nekom od GIT servisa poput github (<https://github.com/>), koji mogu služiti kao backup projekta u oblaku. U današnje vrijeme, u životopisu je postalo normalno priložiti profil s nekog od GIT servisa, da potencijalni poslodavac vidi na kojim ste projektima radili i procjeni koliko znate raditi na projektima koji su njima potrebni.

1.5.2. Lokalni repozitorij

Lokalni repozitoriji nisu više toliko česti. Podrazumijevaju da je na radnom računalu cijeli kod, zajedno s povijesti rada koda i verzijama. U najosnovnijem obliku ovo može biti periodična snimka projekta spremljena na drugoj lokaciji, čak se ne mora koristiti nikakav sustav. U današnje vrijeme ukoliko sami radimo možemo imati samo lokalni

GIT repozitorij, no dobro je da preko GIT-a možemo raditi sigurnosnu kopiju našeg lokalnog projekta u slučaju da nešto krene krivo.

1.5.3. Distribuirani repozitorij

Kako smo već rekli, GIT sustav najčešći je primjer. Distribuirani sustavi jesu sustavi gdje svaki stroj koji sudjeluje u razvoju aplikacije ima svoju lokalnu verziju repozitorija. Iako je distribuirano, svakako imamo centralno mjesto koje je zaslužno za povlačenje repozitorija sa svih lokalnih računala te radi na sjedinjavanju promjena i ponovnoj distribuciji svih promjena na sva računala. Ovo je dobar pristup jer ako jedno računalo ima katastrofalnu grešku od koje se ne može oporaviti, iako je vratiti radnu verziju aplikacije povlačenjem radne verzije s bilo kojeg računala na kojem je program u razvoju, zajedno s njegovim lokalnim repozitorijem.

1.5.4. Servisi za verzioniranje koda

Postoji nekoliko servisa koji nam nude mogućnost sinkronizacije koda među programerima, ali imaju na svojim servisima i kopiju repozitorija za slučaj da zatreba.

Nabrojat ćemo samo najčešće servise bez detaljnog ulaska u njih:

- GitHub
- GitLab
- Bitbucket
- Codebase
- Microsoft Azure DevOps
- Amazon AWS CodeCommit
- Beanstalk
- SourceForge
- ...

Ukratko, ima dosta servisa, a na vama je da procijenite koji je najbolji za vaše potrebe.

1.5.5. SVN

SVN ćemo spomenuti kao centralizirani sustav za verzioniranje koda. SVN je besplatan *open source* sustav za verzioniranje koda. Kod ovog sustava repozitorij je centralno smješten na neki poslužitelj kojem developeri pristupaju.

Primjeri sustava jesu: TotroiseSVN, NextGen SVN i drugi.

Princip rada je preko komandne linije, odnosno preko terminala upisujući tekst komande.

Danas se SVN slabo adaptira s novim programima, a možemo reći da je ostao relevantan zbog inercije, jer ljudi koji su navikli koristiti SVN ostaju na njemu.

Prednosti SVN-a bile bi:

- verzioniraju se mape i individualne datoteke, što ga čini bržim od GIT-a
- postoji norma za manipulaciju datotekama
- programeri promjene u dizajnu i modifikacije mogu pratiti iz minute u minutu jer nema baze podataka i sinkronizacija je na jednom mjestu
- jednostavniji je za izvođenje u mrežnom okruženju
- može se koristiti s više programskih jezika i razvojnih okruženja, neovisan je sustav.

2. POGLAVLJE:

Baze podataka – MS SQL

U OVOM POGLAVLJU NAUČIT ĆETE:

- SQL jezik
- modelirati podatke – relacijski model
- raditi s podacima u bazi podataka
- izmijeniti podatke u bazi podataka
- raditi s upitima u SQL Serveru
- raditi s pogledima
- raditi s funkcijama i procedurama
- raditi s okidačima.

2.1. Uvod

Baze podataka jedan su od važnijih objekata koji se koriste u svakodnevnom radu. Svi podaci koje koristimo u svakodnevnom radu moraju biti negdje zapisani, odnosno pohranjeni. Prije su se podaci pohranjivali pismeno u knjigama, a dolaskom računala podaci su se počeli spremati u datoteke na računalima na diskete, trake, diskove itd.

Za početak trebali bismo razjasniti razliku između podatka i informacije. Podatak je bilo koji znak ili niz znakova. U računalu podatak gledamo kao bit ili niz bitova. Podatak sam po sebi nema značenje. Davanjem značenja podacima dobivamo informaciju. Zato kažemo da nam je informacija podatak s kontekstom ili značenjem. S gledišta programa neka datoteka samo je skup podataka, a kad je otvorimo u konkretnom programu, program služi kao sredstvo koje podacima daje kontekst, odnosno prezentira korisniku programa podatak na način na koji je korisniku čitljiv i time podatak postaje informacija.

Baze podataka nastale su radi potrebe za organizacijom podataka i lakšim dohvatom. Uz organizaciju i dohvat postepeno je dolazila i obrada podataka na samoj bazi podataka. Svaka baza podataka pohranjena je na računalo u nekom digitalnom obliku u nativnom (izvornom) datotečnom sustavu operativnog sustava koji posluhuje bazu podataka. Radi lakšeg i standardiziranog načina komunikacije s bazama podataka izmišljen je jezik za komunikaciju s bazom podataka. Iako danas postoji nekolicina takvih jezika, najčešći je SQL i on će biti jezik koji ćemo koristiti u radu s bazama podataka.

2.2. Baze podataka

2.2.1. Direktne datoteke

U uvodu je spomenuto da su datoteke u početku bile standard spremanja podataka. Datoteku možemo promatrati kao kolekciju, gdje se svaki zapis sastoji od polja, gdje svako polje pripada nekom tipu podataka. To se zove linearan ili direktan zapis datoteke (engl. *flat file*) gdje nema nikakve organizacije zapisa. Primjer direktnog zapisa u datoteci bi bio sljedeći:

Marko;Marulić;12345678901;lvo;lvić;11111111111;lvan;Horvat;22222222222;lvan;;01234...

Problemi takvog zapisa višestruki su, no glavni je problem organizacija datoteka, gdje zapisivanje ovisi o fizičkom mediju, odnosno gdje programer koji izrađuje aplikaciju mora uzeti u obzir sve aspekte takvog rada i zapisa. To otežava izradu aplikacije i

uvodi nekolicinu problema od kojih možemo reći da su najveći sljedeći:

- **konkurentnost** – podacima u jednom trenu može pristupiti samo jedan program
- **integritet** – ako podatke koristi više programa, može doći do korupcije odnosno neželjene promjene podataka (engl. data corruption) i potencijalnih problema. Nema kontrole integriteta ni standarda zapisa, osim onoga koji svaki program ima u sebi
- **ponovna iskoristivost** – direktna datoteka ne može se iskoristiti na nekom drugom programu
- **sigurnost** – nema univerzalnog načina osiguravanja pristupa podacima. Podaci su podložni neovlaštenom čitanju, osim ako programer ne napravi neku vrstu enkripcije.

2.2.2. Sustav za upravljanje bazom podataka

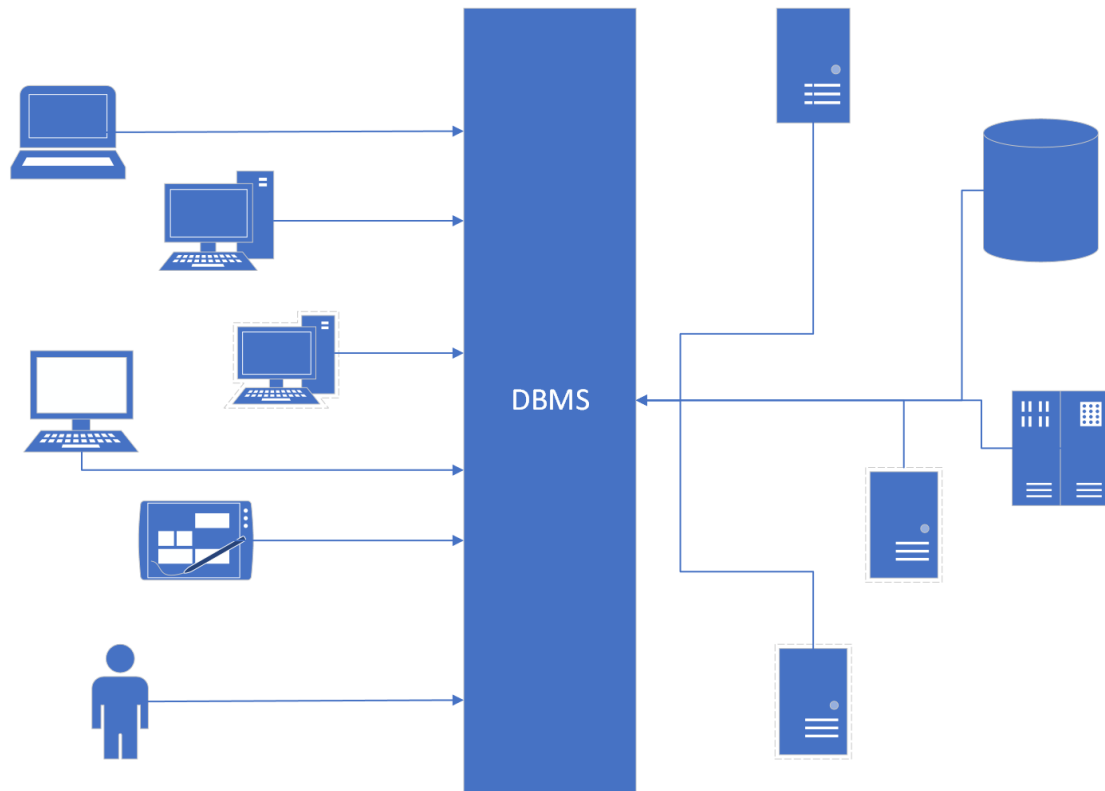
Engleski *Database management system (DBMS)* ili na hrvatskom **sustav za upravljanje bazom podataka (SUBP)** jest sustav koji je zadužen za fizičku pohranu podataka, ali i kontrolu dostupnosti raznim korisnicima i programima. Podaci se svakako nalaze u nekoj fizičkoj datoteci, no za razliku od klasičnog datotečnog pristupa, programer ima unificirani pristup podacima, gdje ne mora brinuti o najmanjim sitnicama pohrane i pristupa podacima.

DBMS u osnovi funkcionira kao poslužitelj baze podataka koja se nalazi između korisnika/aplikacije i same baze podataka. Sam DBMS u ime korisnika obavlja sve operacije s podacima (pristup, izmjena, obrada...).

Rad s bazom podataka ne predstavlja samo dohvat podataka, nego ih treba dodavati, mijenjati, brisati, čuvati im integritet, čuvati sigurnosne kopije, provjere korisnika itd. Zbog svih potrebnih radnji vidljivo je zašto nam treba DBMS.

Sam DBMS mora biti nezavisan u fizičkom i logičkom smislu. Fizička nezavisnost znači da baza može promijeniti medij na kojem zapisuje svoje podatke, čak i biti dis-

tribuirana na više računala dok korisnik uvijek vidi to kao jednu cjelinu. Logička nezavisnost znači da se mogu raditi promjene logičke razine (promjene u podacima i organizaciji podataka) bez da korisnik išta mijenja u svom načinu pristupa bazi.



Slika 4. Vizualizacija DBMS-a

Podaci u bazi podataka organizirani su u skladu s modelom koji predstavlja skup pravila koja određuju logičku strukturu baze. DBMS-ovi obično podržavaju neki od već poznatih modela kao što su:

- relacijski model
- mrežni model
- hijerarhijski model
- objektni model.

Danas je najrasprostranjeniji relacijski model, ali u današnje vrijeme neke se aplikacije pomiču na objektni model, iako to zbog svoje kompliciranosti još nije prevladao. Relacijski je zbog svoje logike i jednostavnosti ipak još uvijek najrasprostranjeniji i zato će on ovdje biti u fokusu.

2.3. Dizajn baze podataka

Kao i sve, tako i baze podataka imaju nekoliko faza dizajna. Ukoliko se radi o nekim manjim bazama, neće biti potrebe da se radi neki veliki inženjering, pa se dobar dio toga može raditi napamet. No većinom ćemo raditi s većim bazama, pa treba uložiti i veći napor prilikom dizajna i izvedbe. Generalno svaki projekt, tako i dizajn baze, možemo podijeliti u nekoliko faza:

- analiza potreba
- modeliranje podataka
- implementiranje
- testiranje
- održavanje.

Analiza potreba odnosi se na prikupljanje informacija o sustavu koji trebamo implementirati. Često je, ako osoba nije upoznata s poslovnim procesom, potrebno konzultirati se sa stručnjacima za to područje. Rezultat analize potrebna je početna dokumentacija sustava, koja će služiti za daljnje korake, odnosno njega možemo nazvati specifikacijom potreba ili korisničkim zahtjevom. S dokumentacijom potreba možemo krenuti na konceptualni dizajn, odnosno raščlanjivanje podataka i izradu *ER dijagrama*.

U koraku **modeliranja podataka** identificiramo kako usustaviti podatke (stvoriti entitete, čime ćemo se upoznati u poglavlju 2.5) koji čine sustav i njihovu međusobnu povezanost. Kad smo to napravili, sljedeći je korak izvođenje modela za implementaciju. Ukratko rečeno, iz ER dijagrama stvaramo stvarnu bazu podataka.

Faza implementacije korak je u kojem dobivamo praktički gotovu bazu podataka. Na temelju shema i pomoću odabranog DBMS-a fizički se kreira baza podataka. Određujemo detalje tehničke izvedbe kako bismo osigurali efikasan rad, implementiramo entitete, određujemo prava pristupa itd.

Faza testiranja ključna je da se testira rad korisnika i da provjerimo zadovoljava li sve tehničke zahtjeve. U ovoj fazi možemo otkriti eventualne greške nastale u ranijim fazama razvoja. Ključno je otkriti imamo li negdje propusta od prije, pogotovo u zaht-

jevima sustava, iako je takve greške idealno otkriti prije implementacije. Testiranje se danas može jednim dijelom automatizirati, no svakako se prije prave implementacije razvijaju približni prototipovi baza podataka koji se pokazuju korisnicima u svrhu ranijeg otkrivanja propusta.

Zadnja je faza **održavanje**, koja se odvija nakon što baza uđe u redovnu upotrebu. Sastoji se od eventualnih popravljivanja grešaka koje nisu bile ranije otkrivene, uvođenja promjena zbog novih zahtjeva korisnika i podešavanja DBMS-a. Svakako je savjet da nikad ne radite na „živim” bazama podataka, nego promjene uvijek isprobajte na testnom stroju ili radite sigurnosnu kopiju neposredno prije ubacivanja izmjena.

2.4. Radna okolina

Da bismo uspješno radili, moramo se upoznati i s radnom okolinom. Iako smo ih spomenuli u pregledu tehnologija, malo ćemo detaljnije ući u alate koje ćemo koristiti. Bitna su nam dva programa za rad, a to su:

- MS SQL server
- SQL Server Management studio.

2.4.1. MS SQL server

Microsoft SQL server sustav je za upravljanje relacijskim bazama podataka koji je razvila tvrtka Microsoft. Primarna funkcija identična mu je kao i svakom DBMS-u, a to je spremanje i dohvat podataka koje zahtijevaju ostali programi, koji ne moraju nužno biti na istom računalu.

SQL server ima dugu povijest, započeo je 1989. kao projekt portiranja Sybase SQL Server na OS/2 platformu u kolaboraciji sa Sybase, Ashton-Tate i samog Microsofta. To se može nazvati SQL Server 1.0, a napravljen je da bude kompatibilan s 16 bitnim serverima kompatibilnima s OS/2 platformom.

Neki od bitnijih verzija i velike promjene sljedeće su:

- 4.2 za Windows NT izašao je 1993.
- 6.0 je izdan 1995., gdje je prestala suradnja sa Sybaseom (koji je nastavio razvijati neovisno svoju verziju)
- 7.0 je izašao 1998., gdje je napravljen velik iskorak konverzija izvornog koda od C u C++
- SQL Server 2005 je izašao 2005., gdje je kompletno izbačen Sybaseov kod i Microsoft u potpunosti održava sav kod za SQL server. Također, vidljivo je promijenjena nomenklatura, gdje se za veće verzije sad uz standardno numeričku nomenklaturu koriste godine izlaska
- SQL Server 2012 dodaje kolumnarni in-memory storage (xVelocity)
- SQL Server 2017 ima podršku za Linux operativne sustave na Red Hat Enterprise, SUSE Linux, Ubuntu, Docker i još nekoliko.

Aktualna verzija u trenutku pisanja je SQL Server 2019.

Instalaciju MS SQL servera moguće je preuzeti na sljedećoj poveznici:

<https://www.microsoft.com/en-us/sql-server/sql-server-downloads>

Također, Microsoft ima višestruke verzije SQL Servera ovisno o različitim setovima dodataka i ciljanoj skupini korisnika. Napraviti ćemo usporedbu 3 najčešće verzije, Express, Standard i Enterprise.

Cijela usporedba može se pronaći na sljedećoj poveznici, no proći ćemo samo ukratko:

<https://docs.microsoft.com/en-us/sql/sql-server/editions-and-components-of-sql-server-2019?view=sql-server-ver15>

	Express	Standard	Enterprise
Maks. broj jezgri koje može koristiti	4	24	Maksimum podržan od strane OS-a
Memorija po instanci (RAM)	1.5 GB	128 GB	Maksimum podržan od strane OS-a
Maksimalna veličina baze podataka	10 GB	524 PB	524 PB
Cijena*	Besplatno	Ovisno o potrebama, od 209 \$, pa sve do 3586 \$	13 748 \$

* <https://www.microsoft.com/en-us/sql-server/sql-server-2019-pricing>

Osnovna je razlika u limitu resursa koji se mogu koristiti u obradi i spremanju podataka, kao i cijeni koja se čini velikom, no ovisno o potrebama sustava može biti samo kap u moru. Naravno, ima još mnogo toga što bismo mogli proći, no većina razlika trenutno nije u domeni razmatranja.

Ono što je svakako u domeni razmatranja jest da svaka od verzija ima podršku za integraciju s Visual Studiom, Management studiom, integraciju s višestrukim programskim jezicima, nativnu podršku za rad s XML-om, podršku za FILESTREAM, UTF-8 podršku, prostorno indeksiranje, prostorne biblioteke, planarne i geografske biblioteke itd.

Ukratko, potreba za većom verzijom SQL Servera morat će se napraviti na temelju potreba programa i sustava s kojim će se program koristiti.

2.4.2. SQL Server Management Studio

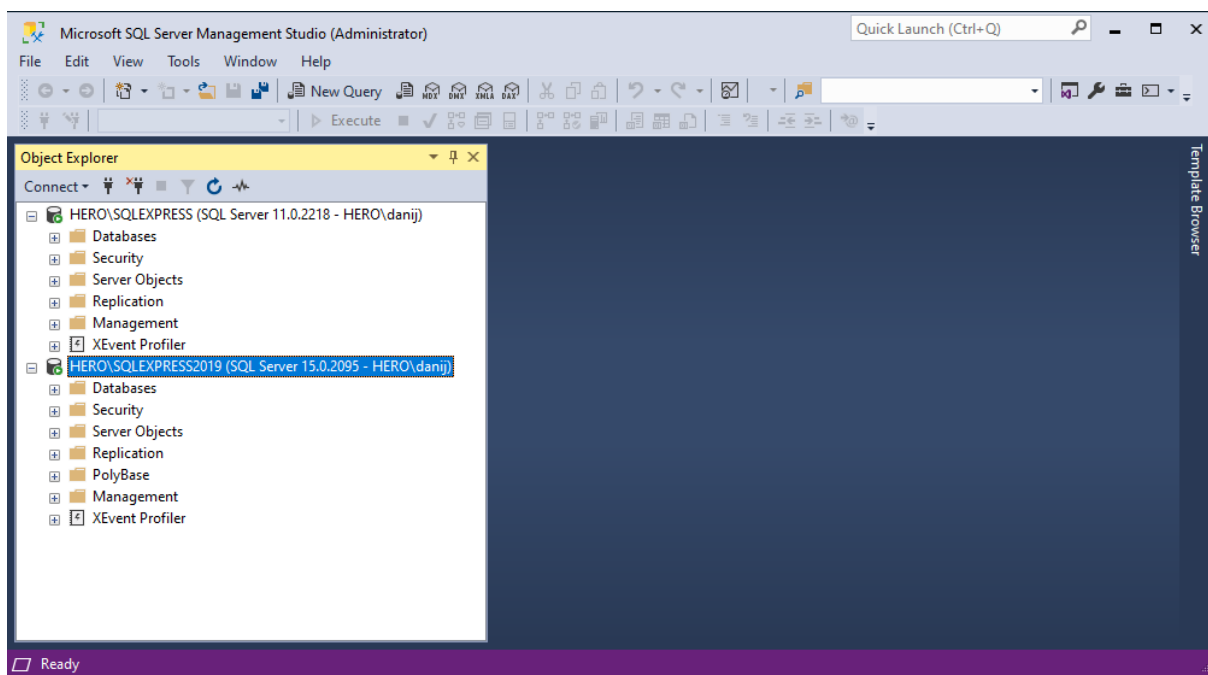
SQL Server Management Studio (SSMS) aplikacija je koja se koristi za konfiguraciju, upravljanje i administraciju svih komponenti MS SQL servera. Inicijalno je lansiran sa SQL Serverom 2005., no nasljednik je Enterprise managera iz SQL Servera 2000. i

prije njega. Uz već navedeno koristi se i za dizajn upita, upravljanje bazama podataka i upravljanja skladišta podataka putem računala.

Moguće je istovremeno spajanje na više različitih inačica i instanci SQL servera, kao i njihovo upravljanje.

U trenutku pisanja aktualna je verzija 18, a može se preuzeti na sljedećoj poveznici:
<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver15>

Centralna funkcionalnost SSMS-a jest Object Explorer (preglednik objekata), koji omogućuje korisnicima da pregledavaju, odabiru i rade s objektima unutar servera.

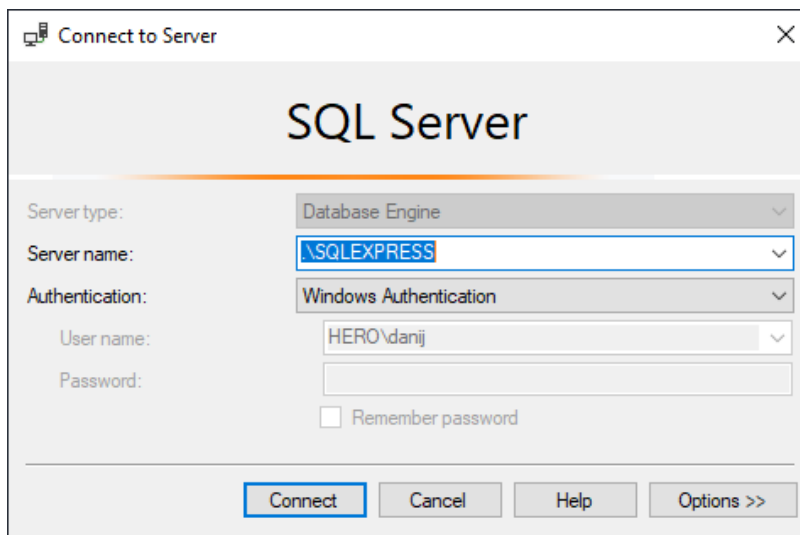


Slika 5. Sučelje SQL Server Management Studia

U prijašnjim verzijama postojalo je više različitih inačica SSMS-a. Zadnje inačice SSMS-a nemaju takvu raznolikost, već su u potpunosti opremljene svim funkcionalnostima. Naravno, potrebna funkcionalnost ovisi o verziji i inačici SQL Servera. Novije verzije SSMS-a u potpunosti su kompatibilne s prijašnjim verzijama SQL Servera.

SSMS je od 2015. godine zaseban program, neovisan o SQL Serveru, iako je prije dolazio kao dio SQL Servera u obliku alata za upravljanje.

Nakon otvaranja SSMS-a ili kasnijim klikom na Connect, po odabiru „database engine” dobit ćete sljedeći prozor kao na slici 6.



Slika 6. Prozor za odabir DBMS-a

Ovaj prozor nam služi da bismo se spojili na DBMS. *Server type* najčešće će biti one-mogućen prilikom pokretanja SSMS-a, a kasnije klikom na Connect možete odabrati tip servera na koji se želite spojiti, odnosno automatski će se ispuniti „Server type”. U kućicu „Server name” možete sami unijeti ime servera ukoliko ga znate ili možete odabrati „Browse for more” ako želite da proces SQL Server Browser pokuša pronaći vidljive instance servera na lokalnom računalu ili na mreži. Opcija „Authentication” ima više sigurnosnih opcija autentifikacije koje provjeravaju smijemo li se spojiti na server. Dvije opcije koristit ćemo najčešće: to su „Windows Authentication” i „SQL Server Authentication”. Kod Windows Authentication za pravo pristupa koristi se lokalni račun kojim ste se prijavili na operativni sustav. Informacije o računima spremljene su na lokalnom računalu ili na Active User Directory servisu na poslužitelju. SQL Server Authentication koristi korisničko ime i lozinku koji se spremaju u DBMS-u.

2.5. Modeliranje podataka

2.5.1. Entiteti i veze

Entitet je objekt, pojava ili događaj čije podatke želimo spremati u bazu. Entitet može i ne mora postojati, ali mora se moći identificirati, npr. proizvod, proizvođač, osoba itd. Obično je iskazan imenicom u jednini.

Svaki entitet ima svoje karakteristike koje ga jedinstveno opisuju, a nazivaju se atributima. Ako uzmemo na primjer da želimo napraviti entitet Osoba, onda bi nam atributi bili ime, prezime, datum rođenja, OIB itd.

Za svaki entitet stvara se tablica ili relacija. Ako pokušamo jednostavno to opisati, pojedinačni entitet (instanca entiteta) bio bi red u tablici, a atributi bi bili stupci.

Veza je odnos između entiteta (sam sa sobom, između dva ili više) i iskazuje se glagolom, odnosno to možemo smatrati vezom između tablica. Imamo nekoliko vrsta veza ovisno o funkcionalnosti:

- **1 na 1 veza** – jedan zapis iz jedne tablice odgovara samo jednom zapisu iz druge tablice. Npr. jedna država ima jednog predsjednika ili jedna osoba ima jednu vozačku dozvolu
- **1 na n veza** – jednom zapisu iz jedne tablice može odgovarati više zapisa iz druge tablice. Npr. jedna država ima više ministara ili jedna osoba ima više vozila
- **n na m veza** – jednom zapisu iz jedne tablice može biti pridruženo više zapisa iz druge tablice, dok istovremeno jednom zapisu iz druge tablice može biti pridruženo više zapisa iz prve tablice. Ova veza direktno nije moguća, nego se u praksi izvodi preko međutablica. Npr. više filmova je režiralo više redatelja ili više osoba je posudilo više knjiga iz knjižnice.

U modeliranju podataka rezultat ovog koraka popisivanje je svih veza, entiteta i njihovih atributa. U modeliranju baze podataka svakako trebamo znati što sve treba zapisati, odnosno spremati da bismo u kasnijim koracima imali što manje vraćanja na

početke. U ovom koraku nekad je više bolje jer je svakako bolje i ne koristiti, odnosno u konačnosti ne spremati neke podatke, nego morati dodavati nove attribute na entitete.

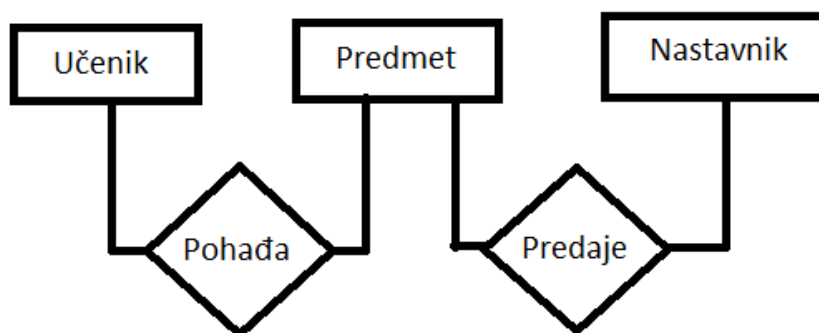
2.5.2. ER dijagram

ER (Entity Relationship) dijagram jest dijagram koji pokazuje relacije (odnose) između entiteta u sustavu. Sam dijagram mora biti jednostavno prikazan da bi ga ljudi koji čak nisu ni upoznati s bazama podataka mogli pročitati i razumjeti, odnosno ljudi različitih struka. ER shema služi prvenstveno za komunikaciju projektanta baze podataka s ljudima različitih struka, odnosno s krajnjim korisnicima u ranijim fazama razvoja aplikacije.

Nijedan DBMS na temelju ER dijagrama ne može implementirati direktno bazu, nego se na temelju ER sheme rade detaljnije razrade i izgrađuje se baza podataka.

ER dijagram podsjeća jednim dijelom na dijagram toka u programiranju, no simboli imaju drugo značenje. Entiteti se prikazuju pravokutnicima, a veze među njima rombovima koji su povezani s entitetima. U dijagram unosimo također imena entiteta i veza, te brojkama označavamo koji tip veze imamo, odnosno koja je funkcionalnost veze.

Primjer ER dijagrama u modeliranju odnosa škole:



Entiteti: Učenik, Nastavnik, Predmet.

Veze: Pohađa, Predaje.

U praksi, prilikom modeliranja baze podataka često ćemo imati nekoliko verzija ER dijagrama. Broj tih probnih verzija s iskustvom će padati, odnosno ranije ćemo moći imati potpuni dijagram na temelju kojeg ćemo pristupiti izradi baze podataka. Baza podataka se iz ER dijagrama dobiva pretvorbom dijagrama u relacijski model (tablice i njihove veze).

2.6. Relacijski model

Relacijski model koncipirao je 1970-ih dr. Edgar F. Codd objavom rada u kojem govori o relacijskom modelu podataka za velike distribuirane banke podataka. Na temelju tog rada imamo ono što danas zovemo relacijski model baze podataka.

Relacijski model baze podataka zasnovan je na ideji da se cjelokupni skup podataka koji želimo spremiti razdvoji u niz tablica povezanih relacijama. Svaka relacija treba imati svoje ime po kojem je razlikujemo od ostalih u bazi.

U osnovi svaka se tablica sastoji od entiteta koji ima svoje atribute. Svaki atribut ima svoje ime, te zato kažemo da je svaki red unutar tablice instance entiteta, a stupci su atributi. Podatkovni tip atributa definira i opseg vrijednosti koji atribut može imati, ali istovremeno ga nije obavezno unijeti.

Tablica u relacijskom modelu skup je zapisa u bazi podataka. Sadrži jedan ili više stupaca, iako su primjeri gdje imamo tablice sa samo jednim stupcem, odnosno atributom, stvarno ekstremni. Svaki stupac ima jedinstveni naziv unutar tablice, ne može se dogoditi da imamo više stupaca istog naziva. Kako smo ranije rekli, svaki red tablice zapis je instance entiteta. Vrijednost atributa nekog entiteta dobijemo kao presjek ćelije, gdje red predstavlja entitet, a stupac atribut koji želimo pročitati.

Osoba			
Šifra	Ime	Prezime	OIB
1	Marko	Marulić	12345678901
2	Ivo	Ivić	11111111111
3	Ana	Anić	22334455669

Danas se relacijski model koristi u velikoj većini komercijalnih sustava iz više razloga. Najvažniji je to što se pokazalo da su mrežni i hijerarhijski modeli usko vezani uz implementaciju baze, a relacijski model nije, pa je relacijski model popularniji izbor. Možda se ne čini baš logičnim na prvu, no kada imamo nešto usko vezano uz implementaciju, veliki problemi nastaju prilikom naknadne izmjene u strukturi, što jako otežava održavanje i proširivanje baze podataka. Mrežni i hijerarhijski modeli uspostavljaju vezu između zapisa pomoću odgovarajućih struktura, a relacijski uspostavljaju vezu na temelju vrijednosti koje ti zapisi sadrže.

Glavna prednost relacijskog modela u tome je što vrlo efikasno rješava dva najveća problema koja se javljaju kod pohrane podataka, a to su redundancija i nekonzistentnost podataka, koje rješavamo postupkom normalizacije baze podataka.

2.6.1. Primarni ključ

Primarni ključ (Primary key) atribut je koji jedinstveno definira zapis. U svakoj tablici vrlo je važno definirati taj atribut. Ako gledamo po primarnom ključu, on je uvijek različit u svakom retku i jedinstveno određuje zapis.

Ako gledamo u praksi, npr. u popisu artikala u trgovini, svaki artikl ima svoju šifru koja je jedinstvena za svaki artikl. Tako kad dođemo u trgovinu i kažemo trgovcu da trebamo artikl pod određenom šifrom, brzo se može pogledati i naći da samo jedna vrsta artikla odgovara toj šifri. Također, ako gledamo u gornjem primjeru s tablicom Osoba, ako kažemo da tražimo osobu pod šifrom 2, znamo da je to Ivo Ivić.

Primarni ključevi u praksi mogu biti različiti. Primjeri su JMBG, OIB, ISBN knjige, poštanski broj, bar-kod proizvoda, pozivni broj itd.

Preporuka je da prilikom dizajna baze podataka probate uvijek definirati svoj primarni ključ ukoliko moguće, na primjer IDOsoba, koji će poprimiti cjelobrojne vrijednosti krenuvši od 1 pa dalje po redu. U praksi se pokazalo nezgodnim vanjske vrijednosti staviti za primarne ključeve jer su moguće izmjene vrijednosti. Npr. ako za primarni

ključ stavite OIB osobe, ukoliko je netko pogrešno unio ili nije mogao odmah unijeti OIB, problem može nastati promjenom primarnog ključa. Primarni se ključ jednom kad se dodijeli ne bi smio mijenjati, odnosno ukoliko se mijenja, moramo paziti da se automatski promijeni i na mjestima koja se referenciraju na vrijednost tog entiteta u drugim tablicama.

Primarni ključ obavezno je definirati u relacijskom modelu, a preporuka je da se naknadno ne mijenja. Može sadržavati jedno ili više polja. Ukoliko sadrži više polja, to se zove kompozitni primarni ključ i u tom je slučaju moguće ponavljanje vrijednosti unutar atributa, no kombinacija atributa korištenih za primarni ključ mora biti jedinstvena.

2.6.2. Pretvorba entiteta i veza u relacije

Pretvorba entiteta vrši se tako da svaki atribut ima svoj stupac u tablici, a u redove upisujemo realne vrijednosti entiteta, odnosno radimo zapise unutar tablice koji predstavljaju konkretni entitet. Drugim riječima, naziv entiteta postaje naziv tablice, atributi postaju naziv stupca, a u redove zapisujemo instance entiteta, odnosno konkretne vrijednosti.

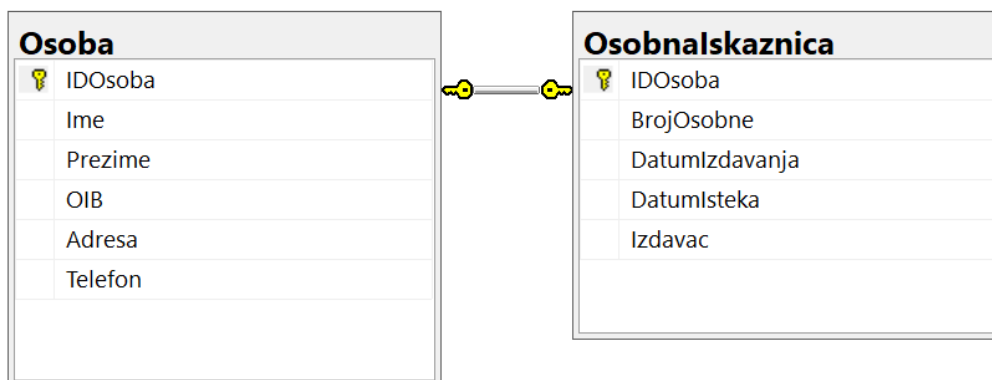
Pretvorba veza kompleksnija je stvar nego sam entitet i ovisi o vrsti veze koju imamo. Ovdje ćemo se upoznati s pojmom *strani ključ*. Strani ključ atribut je koji je vezan za primarni ključ druge tablice, odnosno njega možemo gledati kao referencu (vezu) na drugu tablicu gdje je jednoznačno određen.

Ovisno o funkcionalnosti veze koju trebamo definirati ćemo relacije stranih ključeva.

1 na 1 veza:

Ovo je tipična veza primarnih ključeva. U 1 na 1 vezi primarni ključ jedne tablice ujedno je i strani ključ. Primjer imamo na slici – jedna osoba može imati samo jednu osobnu iskaznicu i jedna osobna iskaznica je povezana na samo jednu osobu. Pazite samo prilikom odabira primarne tablice. U ovom slučaju Osoba je primarna tablica i

logički gledano, ali i realno gledano, zapis osobe mora postojati, ali osoba ne mora imati osobnu iskaznicu, dok suprotno nije istina – osobna iskaznica ne može postojati bez osobe.

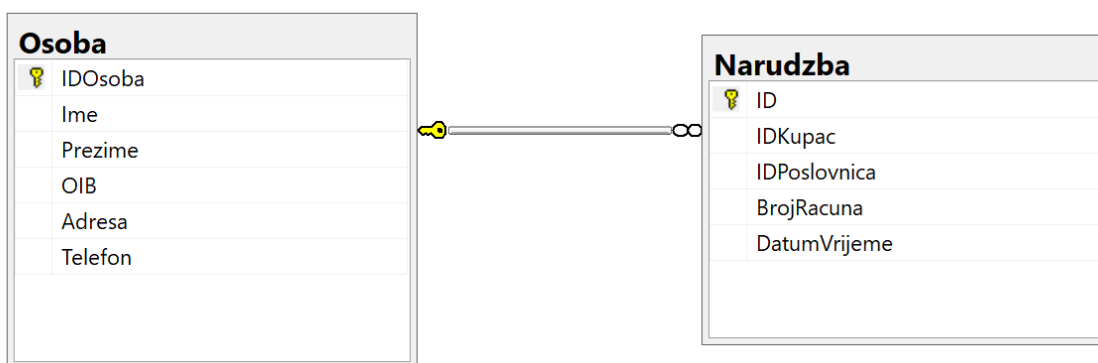


Također treba napomenuti da je ova vrsta veze česta kad se rade proširenja aplikacija. Nije preporučljivo raditi kasnija proširenja tablica, jer je ovisno o veličini projekta teško predvidjeti gdje bi moglo biti problema ukoliko se mijenja definicija tablica. Lakše je napraviti prema novom modulu i napraviti vezu 1 na 1.

1 na n veza:

Ovo će nam biti jedna od najčešćih veza. U ovoj se vezi jedan zapis iz prve tablice može vezati na n zapisa druge tablice. U prvoj je tablici zapis na koji se referira druga tablica primarni ključ, dok je u drugoj tablici strani ključ atribut koji se referira na prvu tablicu.

To je lakše pokazati u sljedećem primjeru, gdje imamo Osobu koja je vezana za narudžbu. IDOsoba je primarni ključ u prvoj tablici, a u drugoj tablici strani ključ je IDKupac koji referencira primarni ključ IDOsoba u tablici Osoba.

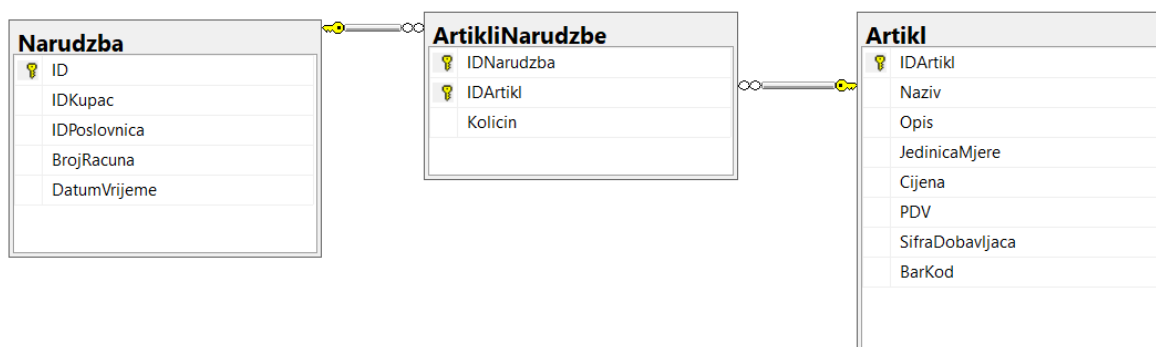


Usput i mala napomena, strani i primarni ključ ne moraju biti istog imena, često i nisu. Iako se nekad lakše snaći kad su istog imena, ovisno o realnom slučaju koji analiziramo uvijek će ime ovisiti o kontekstu u koji stavljamo atribut.

m na n veza:

Veza m na n je jedina koju nije moguće direktno stvoriti. N elemenata jedne tablice ne može se direktno referencirati na m elemenata druge, a da imamo konkretan primarni ključ. Zato se ova veza uvijek se ostvaruje u obliku nove tablice razbijanjem na dvije 1 na n veze.

Prikazat ćemo na najčešćem primjeru, proširivanjem tablice narudžbi. U realnom svijetu znamo da jedna narudžba može imati više artikala, a u isto vrijeme jedan artikl može biti u više narudžbi.



Novu tablicu imamo da ne bi dolazilo do ponavljanja podataka, odnosno tako eliminiramo redundanciju podataka. Da nemamo novu tablicu, u barem bi jednoj tablici bilo ponavljanja podataka ovisno o potrebama druge tablice. Ovako u primjeru definirana tablica ArtikliNarudzbe sadrži 1 na n vezu prema tablici Narudžba i 1 na m vezu prema tablici Artikl.

2.6.3. Normalizacija

Normalizacija je proces organizacije podataka s ciljem smanjenja dupliciranja podataka radi dobivanja efikasne, pouzdane i fleksibilne baze podataka. Sam postupak nor-

malizacije temelji se na dokazano ispravnom matematičkom modelu i ako ga pratimo on sigurno rezultira dobrom bazom podataka. No kao i sve u životu, normalizacija se može i loše napraviti. Ukoliko je postupak normalizacije loše izveden, baza podataka može sadržavati neke greške, no ako se držimo osnovnih pravila normalizacije, to će se jako rijetko dogoditi.

Pravila kojima dolazimo do konzistentnih podataka nazivaju se normalne forme.

Postoji 6 normalnih formi:

1. Prva normalna forma (1NF)
2. Druga normalna forma (2NF)
3. Treća normalna forma (3NF)
4. Boyce-Coddova normalna forma (BCNF)
5. Četvrta normalna forma (4NF)
6. Peta normalna forma (5NF).

1NF zahtijeva da svaka vrijednost u stupcu koja se ponavlja bude atomarna i da svi redci imaju isti broj polja. U biti ako gledamo logično, 1NF nam govori da se jedna vrijednost ne smije pojavljivati kao skup podataka. Primjer toga dan nam je u tablici Osoba, gdje nećemo u isti atribut stavljati ime i prezime, nego ćemo posebno odvojiti ime, a posebno prezime. Da ih imamo u istom atributu, teško bi bilo raditi sortiranja po prezimenu ili imenu jer ne bismo znali koje je koje u kompozitnom atributu. Rijetko kad imat ćemo tablicu koja nije u 1NF jer se nekako iskustveno i zdravom logikom dobivaju razdijeljeni podaci, no svakako se svodi na analizu problema.

2NF kao preduvjet zahtijeva da bude zadovoljen 1NF. Nadalje, zahtjev je da svi redovi budu jednoznačno određeni i ovisni o primarnom ključu. Primjer je vidljiv u tablici Osoba, gdje je primarni ključ IDOsoba ili generalno to bi bila šifra po kojoj dolazimo do samo jednog zapisa u tablici. Ukratko, svaki entitet u svim tablicama mora imati svoj primarni ključ.

3NF kao preduvjet zahtijeva da bude zadovoljen 2NF i da ne postoje tranzitivne ovisnosti o primarnom ključu. Svi stupci koji nisu primarni ključ moraju biti neovisni jedan o drugome, što znači da ne smijemo imati ovisnost jednog podatka o drugome. Ovo se

pravilo može prekršiti jednostavno, ako npr. u nekoj tablici računa imamo attribute za jediničnu cijenu, količinu i ukupno. Budući da atribut ukupno dobivamo jednostavno množenjem jedinične cijene i količine, on ovisi o dotičnim i nema ga smisla zapisivati kad možemo izračunati. Eliminacijom tranzitivnih veza eliminiramo nekonzistentnosti i moguće netočnosti podataka u bazi podataka. Razmislite ovako, kad bi ostali jedinična cijena, količina i ukupno, a netko promijeni jedan podatak bez ažuriranja drugih: u tom slučaju, koji bi podatak bio ispravan?

BCNF zahtijeva da ako postoji stupac o kojem je neki drugi ovisan, da po automatizmu on mora biti ključ tablice. Nije spomenuto da je preduvjet BCNF-a 3NF jer ako je relacija u BCNF, onda je automatski u 3NF, ali postoje situacije gdje relacija koja je u 3NF nije u BCNF.

4NF i **5NF** u praksi se ne koriste jako često, odnosno često znaju komplicirati stvari, laički rečeno. U suštini se svodi na „razbijanje” baze na mnogo malih tablica, što može usporavati čitanje i pisanje te može biti lošije rješenje od nenormalizirane tablice. Često osoba koja razvija bazu podataka odlučuje do koje mjere treba normalizirati i iskustvom procjenjuje što ima smisla.

Samo kao primjer, 4NF kao preduvjet ima 3NF i BCNF, ali zahtijeva i da tablice nemaju viševrijednosne zavisnosti. Viševrijednosne zavisnosti stvaraju nepotrebnu redundanciju i mogu dovesti do nekonzistentnosti podataka. Na primjer, da bismo tablicu osoba stavili u 4NF, atribut ime i prezime trebali bi razdvojiti u posebne tablice. Da ne bi došlo do više vrijednosti s istim zapisom, trebali bismo napraviti tablicu imena i svako individualno ime da ima svoj ključ, te u tablici osoba umjesto imena imati strane ključeve koji se referiraju na tablicu imena. Kako smo već gore naveli, to bi bilo nepotrebno kompliciranje u većini slučajeva.

2.7. SQL jezik

SQL je programski jezik posebno dizajniran za upravljanje podacima u relacijskim bazama podataka. Nastao je kasnih 1970-ih od strane IBM-ovog inženjera Donald D. Chamberlina, koji je o relacijskom modelu baze podataka učio od Edgar F. Codd ranih 1970-ih. Bilo je mnogo različitih implementacija, no i danas najpoznatija je DB2

razvijena 1983. godine. Do 1986. godine ANSI i ISO standardizacijske grupe usvojile su „Database Language SQL” standard, koji je kasnije doživio svoja unapređenja. Zadnja i aktualna verzija samog standarda publicirana je 2019. godine.

Svaka tvrtka ili organizacija koja je napravila svoju verziju DBMS-a utemeljenog na SQL standardu također je dodala svoja proširenja. Sva ta proširenja imaju svoje zasebno ime, ali u osnovi se maksimalno pridržavaju standarda na temelju kojeg su napravljeni. Tako primjerice Microsoft ima Transact-SQL (T-SQL), Oracle ima PL/SQL itd. No događa se da neke implementacije ne podržavaju neka zadana svojstva standardom. Najčešći je primjer gdje varijacije imaju vlastitu implementaciju zapisa vremena i datuma. Zbog različitih implementacija upiti napisani za jedan DBMS ne mogu se često direktno prenijeti na drugi DBMS bez promjena koda. Mnogo je razloga zbog kojih postoje razlike u implementaciji:

- standard ne specificira ponašanje baza podataka u ključnim područjima rada, prepuštajući odluke implementacije standarda proizvođaču
- semantika nije standardizirana, pa postoje dvosmislenosti
- proizvođač DBMS-a ne mora implementirati sve dijelove standarda, nego može implementirati samo dijelove za koje procjeni da su potrebni u radu njegovih korisnika (MySQL i MariaDB, koji je predviđen da bude minimum za rad web-stranica)
- proizvođači mogu proširiti standard svojim implementacijama adekvatno ciljanoj skupini korisnika i specifičnim primjenama.

Danas ima mnogo implementacija SQL standarda, odnosno DBMS-ova, pa da navedemo par, bez nekog redoslijeda:

Proizvođač	Proizvod	Napomena
Microsoft	MS SQL Server	
Oracle	Oracle Database	
IBM Corporation	DB2	
Sybase Inc.	Sybase SQL Server	
Hewlet Packard Co.	Allbase/SQL	
MySQL AB	MySQL	open source
Microsoft	MS Access	T-SQL
IBM Corporation	Informix	

2.7.1. Ključne riječi

SQL jezik sastoji se od nekoliko grupa ključnih riječi, koje same po sebi možemo ponovno nazvati jezikom:

- DCL : Data Control Language ili jezik za kontrolu podataka
- DML : Data Manipulation Language ili jezik za manipulaciju podataka
- DDL : Data Definition Language ili jezik za definiranje podataka.

DDL sadrži osnovne naredbe koje se koriste za definiranje podataka. Tipične ključne riječi su **CREATE** (stvoriti), **DROP** (obrisati ili ispustiti) i **ALTER** (izmijeniti). Koristimo ih na razini servera da bismo radili osnovne radnje s bazama podataka (stvaranje, brisanje i izmjena baze podataka na serveru) i na razini same baze, gdje pomoću tih naredbi stvaramo, brišemo i mijenjamo objekte poput tablica, okidača, indeksa itd.

DCL se koristi za autorizaciju, odnosno pomoću tih naredbi dopuštamo korisnicima baze podataka pristup objektima i rad s objektima. Tipične naredbe DCL-a jesu **GRANT** (daj dopuštenje) i **REVOKE** (makni dopuštenje) naredbe.

DML nam je ovdje najveća skupina, ali i najviše korištena. Pomoću naredbi DML-a manipuliramo podacima u smislu dohvata, dodavanja, brisanja i izmjena. Naredbe DML-a bile bi **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**, **MERGE**, **START TRANSACTION**, **END TRANSACTION**, **COMMIT**, **ROLLBACK**, **FROM**, **WHERE**, **GROUP BY**, **HAVING**, **ORDER BY** i još druge. U ovoj skupini svakako imamo uvelike najkorišteniju ključnu riječ u SQL-u, a to je **SELECT**.

Svakako napomena, ovdje nisu spomenute sve ključne riječi SQL-a, niti ih ima smisla nabrajati. Nekih ćemo se dotaknuti kako ih budemo koristili, a neke nećemo ni spomenuti jer se toliko rijetko koriste i spominju da nema smisla ulaziti u tolike detalje.

2.7.2. Sintaksni dijagrami

Kod SQL-a, a pogotovo u čitanju naredbi SQL-a, bitno je znati kako čitati sintaksne dijagrame. Najbolje je to pokazati na primjeru **SELECT** naredbe:

```
-- Syntax for SQL Server and Azure SQL Database

<SELECT statement> ::=
    [ WITH { [ XMLNAMESPACES , ] [ <common_table_expression> [,...n] ] } ]
    <query_expression>
    [ ORDER BY <order_by_expression> ]
    [ <FOR Clause> ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
<query_expression> ::=
    { <query_specification> | ( <query_expression> ) }
    [ { UNION [ ALL ] | EXCEPT | INTERSECT }
      <query_specification> | ( <query_expression> ) [...n] ]
<query_specification> ::=
SELECT [ ALL | DISTINCT ]
    [ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
    < select_list >
    [ INTO new_table ]
    [ FROM { <table_source> } [ ,...n ] ]
    [ WHERE <search_condition> ]
    [ <GROUP BY> ]
    [ HAVING < search_condition > ]
```

Primjer je preuzet iz službene dokumentacije:

<https://docs.microsoft.com/en-US/sql/t-sql/queries/select-transact-sql?view=sql-server-ver15>

Prvo što treba napomenuti, SQL ne prepoznaje razliku u velikim i malim slovima, tako da kad pišete upite nije bitno pišete li velikim ili malim slovima. Čak možete i kombinirati velika i mala slova unutar iste riječi nasumično (npr. sEleCt). Prilikom pisanja koda nepisano je pravilo da se ključne riječi pišu velikim slovima, a ostalo malim slovima.

U sintaksnim dijagramima:

- riječi napisane velikim slovima rezervirane se riječi SQL-a koje treba napisati točno kako je navedeno
- riječi napisane između < (manje) i > (veće) predstavljaju varijable koje treba zamijeniti nazivima objekata ili konkretnim vrijednostima
- okomita crta | razdvaja ponuđene opcije i možemo odabrati samo jednu od ponuđenih opcija
- uglate zagrade [] govore nam koje stvari nije obavezno navesti
- vitičaste zagrade { } govore nam što smo obavezni napisati
- u uglatim zagradama zapis [,...n] govori nam da je ovo stavka koja se može ponavljati jednom ili više puta, no stavke moramo razdvojiti zarezima
- -- (dva minusa ili povlake) označuju komentar do kraja trenutne linije, dok

se višelinijski komentar obilježava isto kao i u C# programskom jeziku, počinje kosom crtom i zvjezdicom (*), a završava sa zvjezdicom i kosom crtom (*).

2.8. Tipovi podataka

Tip podataka jest skup vrijednosti koje atribut može poprimiti i definira koje operacije možemo koristiti nad tim skupom.

Malo suhoparna definicija, pogotovo onima koji se prvi put susreću s ovime. Za primjer, možemo imati prirodne brojeve {0, 1, 2, 3, ... } te na njima dobro nam poznate operacije zbrajanja, oduzimanja, množenja, dijeljenja i ostalih. No ako imamo tekst, neke operacije nemaju smisla ili nisu izvedive. Na tekstu primjerice ne možemo raditi operacije oduzimanja i dijeljenja, kao i neke kompleksnije matematičke operacije poput korjenovanja.

Tipovi podataka bitni su nam kako bismo znali što možemo prikazati s pojedinim tipom, naravno i uz operacije koje koristimo. Danas je zbog brzine računala i količine memorije koju imamo na raspolaganju jednostavno zaboraviti koncept štednje resursa, ali još gore dosta programera zaboravilo je koncept optimizacije. Uzmite u obzir da mali podatkovni tipovi imaju mali raspon vrijednosti, ali i kraći zapis broja ima manje znamenki, što znači da je rad s osnovnim operacijama brži. Baze podataka nećemo često koristiti samo za jednog korisnika, nego će često posluživati nekoliko korisnika/aplikacija, pa sve do posluživanja web-stranica za globalne korisnike (do potencijalno preko nekoliko milijuna).

Svakako jedna napomena – kako smo rekli, svaka implementacija SQL standarda nije baš potpuno točna i postoje odstupanja. Koncentrirat ćemo se na Microsoftovu implementaciju koja je prisutna u MS SQL Serveru, ali i MS Access programima. No nemojte misliti da je ovo ograničavajuće, kad naučite ovo, jednostavno ćete se moći prebaciti na bilo koju implementaciju SQL standarda.

Nećemo spomenuti sve moguće podatkovne tipove, jednostavno ih je previše i nema potrebe opterećivati se time. Neke ćemo jako često koristiti, neke možda nikada nećemo koristiti.

Za početak svakako moramo spomenuti „vrijednost” koja se koristi za odsutnost podatka u zapisu. Naziv za odsutnost podatka je **NULL**. Primjerice kod brojeva, reći da je vrijednost 0 ili NULL nije isto. 0 označava da imamo konkretan zapis, vrijednost 0, a NULL označava da podatak ne postoji, da nije definiran. Zamislite to ovako, to vam je kao račun u banci, ako imate račun u banci i nemate novaca na njemu, bit će vam 0 na računu, a NULL bi bilo da uopće nemate račun u banci, kao da niste klijent banke. Svaki podatkovni tip ima posebno definiranu NULL vrijednost.

Podatkovne tipove možemo podijeliti na nekoliko podatkovnih skupina, pa idemo redom.

Naziv	Pohrana (byte)	Raspon
tinyint	1	0 do 255
smallint	2	Od -2^{15} do $2^{15}-1$ Od -32.768 do 32.767
int	4	Od -2^{31} do $2^{31}-1$ Od -2.147.483.648 do 2.147.483.647
bigint	8	Od -2^{63} do $2^{63}-1$ Od -9.223.372.036.854.775.808 do 9.223.372.036.854.775.807
bit	1	0 ili 1
decimal, numeric	5 do 17	Broj fiksne preciznosti od $-10^{38}+1$ do $10^{38}-1$ (maks. preciznost)
money	8	Valuta s preciznošću na 4 decimale -922,337,203,685,477.5808 do 922,337,203,685,477.5807
smallmoney	4	Valuta s preciznošću na 4 decimale od -214.768,3648 do 214.768,3647

U prikazu brojeva u računalu pravilo je da broj koji možemo prikazati ovisi o broju bitova koje imamo na raspolaganju. Ako imamo n bitova, onda je najveći broj koji možemo prikazati $(2^n - 1)$. Inače s istim brojem bitova možemo prikazati 2^n kombinacija, ali budući da je i 0 broj koji prikazujemo, zato ide -1 u maksimalnom broju, gdje se podrazumijeva da su sve jedinice zapisane. Ako trebamo zapisivati i negativne brojeve, prvi bit često nam govori radi li se o zapisu pozitivnog ili negativnog broja, no to nam ne smanjuje broj kombinacija koje možemo koristiti za brojeve, nego samo pomičemo raspon u kojem brojeve možemo pisati. Efektivno upola smanjimo najveći broj koji možemo zapisati da bismo imali točan zapis i negativnih brojeva. Zato kod `small int`, `int` i `bigint` imamo ovakav raspon brojeva.

Iako bit ima samo dvije vrijednosti, za spremanje trebamo cijeli bajt jer je nemoguće pohraniti manju jedinicu memorije. Pomoću bit podatkovnog tipa spremamo jedinicu ili nulu, ali možemo spremiti i `NULL` vrijednost. Koristimo je uglavnom umjesto boolean podatkovnog tipa.

`Decimal` nam je varijabilan što se tiče toga koliko će podataka spremiti jer najčešće moramo definirati preciznost i skalu. Kad deklariramo decimalni podatkovni tip, napominjemo u zagradama dva broja. Definiramo kao `decimal(p,s)`, gdje nam p predstavlja preciznost broja, a s skalu broja. Preciznost ide od 1 do 38, gdje je standardna preciznost 18. Skala je u rasponu $0 \leq s \leq p$, a govori nam o tome koliko će se spremati decimalnih mjesta nakon decimalne točke. Primjer `decimal (7,2)`, znači da je preciznost 7 znamenki, s dva decimalna mjesta.

`Money` i `smallmoney` podatkovni su tipovi preciznosti jedne deset tisućinke monetarne jedinice. Ovo je primjer proširenja SQL standarda, jer sam SQL standard nema ovaj podatkovni tip. `Money` podatkovni tip doslovno je `bigint`, samo su posljednje 4 znamenke umjesto cijelog broja decimalni brojevi. `Smallmoney` slično tako, samo `int` umjesto `bigint`.

2.8.2. Približni brojevi

Možda malo čudno zvuči približni brojevi umjesto decimalnih ili realnih brojeva, no ovako je preciznije rečeno, a i na engleskom se kaže „approximate numeric”. Razlog tome je jednostavan – decimalne brojeve često ne možemo točno prikazati kao binarni decimalni broj s konačnim brojem decimalnih mjesta, pa zato kažemo da spremamo približne vrijednosti.

Naziv	Pohrana (byte)	Raspon
float	4 ili 8	- 1.79E+308 do -2.23E-308, 0 i 2.23E-308 do 1.79E+308
real	4	3.40E + 38 to -1.18E - 38, 0 i 1.18E - 38 do 3.40E + 38

Inače kad se definira float, definira se kao float(n), gdje n predstavlja broj bitova mantise (mantisa predstavlja preciznost zapisa broja) kako je definirano ISO standardom za zapis realnog broja. Ukoliko je n između 1 i 24, onda je zapis kao standardni 32 bitni realni broj (4B memorije), a ukoliko je između 25 i 53, onda je zapis kao standardni 64 bitni realni broj (8B memorije).

Real podatkovni tip je gledano po ISO standardu zapis realnog broja u 32 bita.

2.8.3. Datum i vrijeme

SQL jezik se sastoji od nekoliko grupa ključnih riječi, koje same po sebi možemo ponovno nazvati jezikom:

Naziv	Pohrana (byte)	Raspon
date	3	Prikaz datuma od 1.1.0001. do 31.12.9999. s preciznošću od jednog dana
datetime	8	Prikaz datuma i vremena od 1.1.1753. 00:00:00 do 31.12.9999 23:59:997 s preciznošću od 0.003 sekunde
datetime2	8	Prikaz datuma i vremena od 1.1.0001. 00:00:00 do 31.12.9999 23:59:59.9999999 s preciznošću od 100 nanosekunde

smalldatetime	4	Prikaz datuma i vremena od 1.1.1900. 00:00 do 31.12.2079 23:59 s preciznošću od jedne sekunde
time	5	Prikaz vremena od 00:00:000000000 do 23:59:59.99999999 s preciznošću od 100 nanosekundi
datetimeoffset	10	Prikaz datuma i vremena od 1.1.0001. 00:00:00 do 31.12.9999 23:59:59.9999999 s preciznošću od 100 nanosekundi

Naizgled neki imaju slične ili čak iste raspone pohrane vrijednosti, no ono po čemu se razlikuju su standardni format zapisa podataka i preciznost rada.

2.8.4. Znakovi i nizovi znakova

Jedna od najbitnijih skupina za pohranu podataka upravo je ova. Služi nam za pohranu znakova i niza znakova (teksta).

Naziv	Pohrana (byte)	Raspon
char	1 do 8000	Standardni ASCII znakovi fiksne duljine Najveća duljina niza je 8000 znakova
varchar	1 do 8000 ili 2GB	Standardni ASCII znakovi varijabilne duljine, gdje je najveća duljina niza 8000 znakova ili max
nchar	2 do 8000	Za spremanje Unicode znakova fiksne duljine. Maksimalno 4000 znakova
nvarchar	2 do 8000 ili 2GB	Za spremanje unicode znakova varijabilne duljine, gdje je najveća duljina niza 4000 znakova ili max
text	do 2GB	Ne unicode znakovi varijabilne duljine s maksimalnom $2^{31}-1$ znakova
ntext	do 2GB	Unicode znakovi varijabilne duljine s maksimalnom $2^{30}-1$ znakova. U ISO standardu poznato kao national text

Char i **varchar** u osnovi spremaju jednake znakove. Razlika je samo u memoriji koju zauzimaju. Prilikom deklaracije podatkovnog tipa pored imena stavlja se u zagradu n (npr. char(n)). Parametar n ima ulogu kazati koliko će maksimalno bajtova zapis niza zauzimati. Izuzetno je bitno spomenuti broj bajtova, a ne broj znakova. Ukoliko se koriste single byte enkodirani znakovi, to znači i broj znakova koji će biti u nizu, no ukoliko je to multy byte encoding, znači da će broj znakova biti manji od definiranog n. Na podatkovni tip s var i bez var u svom imenu trebamo paziti. Bez var znači da je za definirani n uvijek rezervirano točno toliko memorije koliko smo napomenuli. Ukoliko koristimo var, količina memorije koja se stvarno koristi može varirati od toga koliko je potrebno za trenutni zapis pa do n bajtova koliko je rečeno.

Nchar i **nvarchar** koriste se za spremanje unicode znakova. Ovdje je svaki znak spremljen u 2 bajta, što je i vidljivo prema omjeru maksimalno potrebne memorije za pohranu podataka i broja znakova koje možemo pohraniti. U ovom slučaju, kad definiramo duljinu, govorimo o tome koliko ćemo znakova spremiti, a ne o veličini potrebne memorije.

Text i **ntext** služe za pohranu većih količina teksta. Ni kod jednog ne navodimo veličinu, nego se svakako varijabilno spremaju u memoriji do njihova maksimuma znakova koji je za text otprilike dvije milijarde znakova, a za ntext otprilike milijardu znakova. Nije pogreška, svaki atribut tako definiran može imati podataka u rasponu oko milijarde. Razlika je što text sprema ne unicode znakove, a ntext unicode znakove.

- Svakako jako bitna napomena, najavljeno je da će se u budućim verzijama ukloniti text i ntext podatkovni tipovi, pa je umjesto njih svakako preporučljivo koristiti varchar ili nvarchar.

2.8.5. Binarni nizovi znakova

Binarni nizovi služe nam za spremanje onoga što zovemo „sirovi” binarni podaci. Često se koriste za spremanje datoteka ili kodiranih podataka u sustavu.

Naziv	Pohrana (byte)	Raspon
binary	1 do 8000	Binarni podatak fiksne dužine bajtova od 1 do 8000
varbinary	1 do 8000 ili 2GB	Binarni podatak varijabilne dužine bajtova od 1 do 8000, ali može biti i max
image	0 do 2GB	Binarni podaci varijabilne dužine do 2GB

- Svakako jako bitna napomena, najavljeno je da će se u budućim verzijama ukloniti image podatkovni tipovi, pa je umjesto njih svakako preporučljivo koristiti varbinary.

2.8.6. Ostali podatkovni tipovi

Naravno, nismo sve podatkovne tipove prošli, za to treba puno više prostora, a i o svakome bi se moglo pričati uvelike. Još ćemo samo spomenuti nekoliko češćih uz kratak opis za referencu.

Naziv	Pohrana (byte)	Raspon
cursor		Podatkovni tip za varijable ili za izlazne parametre spremljenih procedura koje sadrže referencu na cursor. Ne može se koristiti kao podatkovni tip atributa unutar tablice!
table		Specijalni podatkovni tip za spremanje rezultata za buduću obradu. Može se koristiti samo kao lokalna varijabla, ne može biti atribut unutar tablice!
sql_variant	Do 8kB	Tip podataka za pohranu različitih vrijednosti, ali i različitih tipova podataka. Mogu se koristiti i korisnički tipovi podataka. Mogu se koristiti u tablicama, zajedno s podatkom prema se i njegov tip, što omogućuje u jednom atributu spremanje različitih podatkovnih tipova.
xml	Do 2GB	Za spremanje XML podataka, može se koristiti kao atribut i kao lokalna varijabla u programu.

geometry		Za spremanje planarnih prostornih podataka. Ovaj tip predstavlja podatke u Euklidovom koordinatnom sustavu. Tolerancija pogreške je 10 ⁻⁷ između dvije točke. Dio je OGC standarda (Open Geospatial Consortium) i set je Microsoftovih ekstenzija tom standardu. Također imamo i mnogo funkcija za rad s ovim podatkovnim tipom.
geography		Za spremanje geografskih prostornih koordinata. Sprema elipsoidne podatke poput GPS koordinata. Izuzetno korisno u GIS sustavima. Isto je dio OGC standarda. Korisno za rad s mapama, rutama i drugim podacima na kartama.

2.9. Upravljanje strukturom baze

Da bismo radili s tablicama, prvo što svakako moramo napraviti jest kreirati bazu podataka. S time ćemo raditi u programu kojim ćemo pristupiti DBMS-u. Budući da mi radimo s MS SQL Server-om, koristit ćemo ranije spomenuti SQL Server Management Studio.

2.9.1. Stvaranje objekata CREATE naredbom

Da bismo napravili bazu podataka, potrebno je koristiti naredbu **CREATE DATABASE** i dati ime baze na sljedeći način:

```
CREATE DATABASE Trgovina
GO
```

Ovom naredbom napraviti ćemo novu bazu podataka. Naredba **GO** nije dio SQL-a, nego jednostavno na taj način Management Studio razlikuje blokove naredbi, kojih može biti više.

Skoro svaka naredba ne mora se napisati, nego se može i preko korisničkog sučelja napraviti klikanjem. Desnim klikom na objekt u Object Explorer dijelu otvaraju se moguće opcije. Tako bazu možemo napraviti i desnim klikom na „Databases” pa „New database”.

Prilikom kreiranja SQL stvara dvije datoteke za rad s bazom podataka. Obje datoteke imaju ime baze podataka koju smo naveli, ali razlikuju se po ekstenzijama. Glavni dio baze podataka spremljen je u mdf datoteku, dok druga datoteka ima ekstenziju ldf. Obje datoteke neophodne su za rad baze podataka u DBMS-u!

Sad kad imamo kreiranu bazu podataka, vrijeme je za kreiranje tablice u koju ćemo spremati podatke. Za to koristimo naredbu **CREATE TABLE**, koja izgleda ovako:

```
CREATE TABLE <ime_tablice>
(
    <stupac> <podatkovni tip> <nulabilnost>
    ...
)
GO
```

Ako u našoj bazi želimo napraviti tablicu Osoba, to bi izgledalo ovako:

USE Trgovina

```
CREATE TABLE Osoba(
    IDOsoba    int,
    Ime        nvarchar(100),
    Prezime    nvarchar(100),
    OIB        char(11),
    Adresa     nvarchar(100),
    Telefon    nvarchar(20)
)
GO
```

Naredba **USE** kaže Management Studiu da se sljedeće naredbe vrše unutar navedene baze podataka. Tijekom kreiranja tablice navodimo za svaki atribut ime atributa i podatkovni tip atributa.

2.9.2. Izmjena objekata **ALTER** naredbom

Bilo kakve promjene na objektima vršimo pomoću **ALTER** naredbe. Tom naredbom možemo mijenjati bazu podataka u smislu pomicanja – gdje će se pohranjivati i druge opcije. Također, pomoću ove naredbe možemo mijenjati i druge objekte unutar baze podataka.

Na tablici možemo mijenjati definiciju tablice u smislu dodavanja, brisanja i izmjene stupca. Za primjer ćemo u tablici Osoba dodati e-mail i promijeniti podatkovni tip telefona.

```
USE Trgovina
```

```
ALTER TABLE Osoba
    ADD email nvarchar(50)
GO
ALTER TABLE Osoba
    ALTER COLUMN Telefon char(20)
GO
```

2.9.3. Brisanje objekata DROP naredbom

Naredba **DROP** služi nam za brisanje objekata u bazi podataka, ali i za brisanje cijele baze podataka.

- Svako brisanje nepovratna je operacija, tako da treba biti oprezan!

Brisanje stupca telefon izvršili bismo ovako:

```
USE Trgovina

ALTER TABLE Osoba
    DROP COLUMN Telefon
GO
```

Brisanje tablice:

```
USE Trgovina

DROP TABLE Osoba
GO
```

Brisanje cijele baze:

```
USE master

DROP DATABASE Trgovina
GO
```

Kod brisanja baze podataka moramo koristiti neku drugu bazu podataka, tako da ona koju brišemo ne bude zauzeta. Ukoliko neki program ili drugi korisnik trenutno koriste bazu podataka, odnosno imaju otvorenu konekciju prema bazi podataka, neće biti moguće obrisati bazu podataka dokle god sve konekcije ne budu zatvorene. Inače se to može i prisilno napraviti.

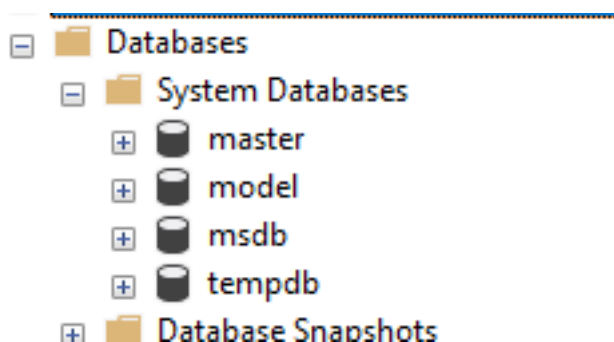
Baza podataka master jedna je od specijalnih sistemskih baza podataka.

- Sve operacije koje radimo naredbama možemo raditi i grafički pomoću Object Explorera u SQL Server Management Studio alatu. Iako se grafički nekada čini lakše i vidimo opcije rada, ipak radimo preko koda jer taj isti kod koristimo za rad s bazom podataka iz naše aplikacije.

2.9.4. Sistemske baze podataka SQL servera

Svaki DBMS ima nekoliko sistemskih baza podataka koje moraju sadržavati informacije koje su bitne za server i koje će server koristiti u svom radu. Ukoliko proširite u Object Exploreru pod Databases, pa System Databases, vidjet ćete da server sadrži 4 baze:

- master
- model
- msdb
- tempdb



Master baza podataka sadrži sve sistemske informacije SQL Servera.

Model baza podataka koristi se kao predložak na temelju kojeg se rade sve nove baze podataka na serveru. Informacije o novokreiranim bazama podataka zapisuju se u master bazu, a sve ostalo zapisuje se kao najmanje dvije datoteke na sustavu za pohranu u obliku primarne (mdf) datoteke i log (ldf) datoteke. Ukoliko drukčije ne nave-

demo, nove baze podataka kreiraju se u instalacijskoj mapi pod mapom Data. Možete slobodno provjeriti u mapi „C:\Program Files\Microsoft SQL Server\MSSQL15.SQL-EXPRESS\MSSQL\DATA” ako prilikom instalacije nije promijenjena putanja instalacije.

Baza msdb služi za rad SQL server agent procesa, koji je zadužen za izvršavanje poslova koje treba periodički obaviti, npr. arhiviranje baza podataka i slično.

Tempdb baza služi nam za spremanje privremenih objekata koji su potrebni u radu SQL Servera.

2.9.5. Ograničenja

Za stvaranje baze podataka važan je dio definiranje ograničenja, koja nam služe za osiguravanje integriteta baze podataka, ali i samih podataka. Ograničenja su pravila koja možemo napraviti prilikom kreiranja baze ili tablica, ali i kasnije naredbom **ALTER**. Ograničenjima uvodimo pravila po kojima se podaci ponašaju unutar nekog stupca. U SQL Serveru imamo nekoliko kategorija ograničenja: **NOT NULL**, **CHECK**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** i **DEFAULT**.

NOT NULL:

NOT NULL ograničenje koristimo kada želimo spriječiti unošenje prazne (null) vrijednosti u stupac, odnosno kada je unos obavezan. Recimo da nema smisla dodati u zapis osobu koja nema svoj ID ili šifru po kojoj bismo pamtili nekoga. Ovo ograničenje možemo dodati prilikom stvaranja tablice ili kasnije izmjenom tablice:

```
USE Trgovina
```

```
CREATE TABLE Osoba(  
    IDOsoba          int NOT NULL,  
    Ime              nvarchar(100) NOT NULL,  
    Prezime          nvarchar(100) NOT NULL,  
    Mjesto_stanovanja nvarchar(100)  
)  
GO
```

Sad kad ovako definiramo tablicu, svaki stupac koji ima NOT NULL ograničenje, ima zabranu unosa NULL vrijednosti, odnosno mora se unijeti taj podatak. Ako to želimo napraviti i kod Mjesto_stanovanja, možemo napraviti izmjenom na sljedeći način:

```
USE Trgovina
```

```
ALTER TABLE Osoba
    ALTER COLUMN Mjesto_stanovanja nvarchar(100) NOT NULL
GO
```

Ukoliko ne navedemo NOT NULL, podrazumijeva se da stupac može primiti NULL vrijednost.

CHECK:

Ovo ograničenje koristimo kad želimo ograničiti raspon vrijednosti koji se može unijeti u neki stupac. Npr. ako želimo ograničiti IDOsobe da bude između 100 i 1000, to bismo rekli ovako:

```
USE Trgovina
```

```
CREATE TABLE Osoba(
    IDOsoba                int NOT NULL,
    Ime                    nvarchar(100) NOT NULL,
    Prezime                 nvarchar(100) NOT NULL,
    Mjesto_stanovanja       nvarchar(100)
    CONSTRAINT chk_IDOsoba CHECK (IDOsoba BETWEEN 100 and 1000)
)
GO
```

Ograničenje smo mogli uvesti i na drugi način, da ne napravimo imenovanu instancu kao u gornjem primjeru, nego bez imenovane instance ovako:

```
USE Trgovina
```

```
CREATE TABLE Osoba(
    IDOsoba                int NOT NULL
                           CHECK (IDOsoba BETWEEN 100 and 1000),
    Ime                    nvarchar(100) NOT NULL,
    Prezime                 nvarchar(100) NOT NULL,
    Mjesto_stanovanja       nvarchar(100)
)
GO
```

Jedina je razlika što imenovanu instancu kasnije po potrebi možemo referencirati i time lakše ukloniti ako postoji potreba.

UNIQUE:

Ograničenje **UNIQUE** koristimo kada želimo da u navedenom stupcu ne želimo ponavljanje vrijednosti. Oprezno samo s NULL vrijednostima: budući da se to u zapisu smatra vrijednošću, ako je stupac deklariran kao unique, NULL vrijednost može se pojaviti samo jedanput.

PRIMARY KEY:

PRIMARY KEY ograničenje koristimo kada želimo da navedeni stupac bude primarni ključ. Da se podsjetimo, primarni ključ vrijednost je koja jedinstveno određuje neki redak unutar tablice. Svaki primarni ključ ujedno je i unique ograničenje i ne može biti null vrijednost.

Svaka tablica trebala bi imati primarni ključ. Preporuka je da primarni ključ bude što kraći, što u praksi dovodi do toga da često bude neka brojčana vrijednost (najčešće int ili bigint ako trebamo veći raspon). Primarni ključ definiramo na sljedeći način prilikom kreiranja tablice:

USE Trgovina

```
CREATE TABLE Osoba(  
    IDOsoba          int PRIMARY KEY,  
    Ime              nvarchar(100) NOT NULL,  
    Prezime          nvarchar(100) NOT NULL,  
    Mjesto_stanovanja nvarchar(100)  
)  
GO
```

Kod primarnog ključa dobro je dodati opciju „Identity” s korakom inkrementa. Na taj način SQL ne daje da se unosi primarni ključ, nego ga dodjeljuje na temelju posljednje dodanog. Ovako sprečavamo pokušaje dodavanja duplog ključa i računanje koji je sljedeći u seriji. Identity bi dodali na sljedeći način:

```
IDOsoba          int IDENTITY(1,1) PRIMARY KEY
```

FOREIGN KEY:

FOREIGN KEY ograničenje također smo ranije spominjali. Strani ključ označava vezu (relaciju) između tablica. Strani ključ referenca nam je na primarni ključ iz druge tablice. Ukoliko je neki stupac definiran kao strani ključ, u redak na tom mjestu ne možemo ubaciti vrijednost koja se ne pojavljuje kao primarni ključ u drugoj tablici na koju se referira. Jedini izuzetak ovom pravilu je null vrijednost, koju možemo unijeti u strani ključ, naravno ukoliko je to dopušteno u definiciji tablice. Primjer gdje u tablici Narudžba možemo dodati strani ključ:

```
USE Trgovina
GO
```

```
CREATE TABLE Narudzba(
    ID          int PRIMARY KEY,
    IDKupac     int
                FOREIGN KEY
                REFERENCES Osoba (IDOsoba),
    IDPoslovnica int,
    BrojRacuna  nvarchar(100),
    DatumVrijeme datetime
)
GO
```

DEFAULT:

DEFAULT ograničenje koristimo da postavimo neku početnu vrijednost u stupcu ukoliko prilikom unosa podatka dotični podatak nije eksplicitno definiran. Default ograničenje dodjeljuje zadanu vrijednost umjesto da ostavi null zapis. Primjer u tablici narudžba u prijašnjem primjeru, za default vrijednost postaviti ćemo da ako nema nove vrijednosti, da ubaci trenutni datum i vrijeme:

```
USE Trgovina
GO
```

```
CREATE TABLE Narudzba(
    ID          int PRIMARY KEY,
    IDKupac     int
                FOREIGN KEY
                REFERENCES Osoba (IDOsoba),
    IDPoslovnica int,
    BrojRacuna  nvarchar(100),
    DatumVrijeme datetime DEFAULT getdate()
)
GO
```

2.10. Manipulacija podacima

2.10.1. Ubacivanje

Nakon što smo napravili bazu podataka i kreirali tablice, vrijeme je da popunimo bazu tablice podacima. Nove vrijednosti u tablicu unosimo naredbom **INSERT INTO**. Prilikom unosa bitno je vidjeti da ako unosom podataka prekršimo neko od prije navedenih ograničenja, DBMS će odbiti izvršiti naredbu koja krši ograničenja.

Insert naredba ima sljedeću sintaksu:

```
INSERT
{
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO ]
    { <object> | rowset_function_limited
      [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
    }
    {
        [ ( column_list ) ]
        [ <OUTPUT Clause> ]
        { VALUES ( { DEFAULT | NULL | expression } [ ,...n ] ) [ ,...n
    ]
        | derived_table
        | execute_statement
        | <dml_table_source>
        | DEFAULT VALUES
    }
}
[;]
```

Ukratko, najčešći oblik je:

```
INSERT INTO <ime tablice> [ ( popis_stupaca ) ]
VALUES ( popis_vrijednosti )
```

Popis stupaca nije obavezan. Ukoliko ne navedemo popis stupaca, odnosno atributa koje želimo ubaciti, onda se u **VALUES** očekuje da unesemo sve attribute koje možemo unijeti točno onim redoslijedom kako su napisani prilikom kreiranja tablice.

Ako upisujemo novu osobu u tablicu, upisali bismo je na sljedeći način:

```
USE Trgovina
GO
INSERT INTO Osoba VALUES(1, 'Josip', 'Kozarac', 'Vinkovci')
GO
```

Unos ne mora biti redoslijedom kako su stupci navedeni u tablici, no ako želimo promijeniti redoslijed ili uopće ne unijeti neke podatke, moramo koristiti popis stupaca. Praktično gledano, možemo unijeti samo prazan zapis, ali zbog referentnog integriteta moramo paziti da primarni ključ uvijek bude definiran. Tako možemo raditi sljedeće zapise:

```
USE Trgovina
GO
INSERT INTO Osoba(IDOsoba) VALUES(2);
INSERT INTO Osoba(Prezime, IDOsoba) VALUES('Runjanin', 3);
GO
```

Ukoliko nismo naveli podatke u redu koji želimo upisati, oni će se automatski postaviti na NULL vrijednosti. Oprezno s time, ukoliko je navedeno not null ograničenje, nije moguće ne navesti stupac, odnosno obavezno je spremiti neku vrijednost. NULL možemo eksplicitno navesti ukoliko želimo da podatak ostane da nije unesen kao u sljedećem primjeru:

```
USE Trgovina
GO
INSERT INTO Osoba VALUES(4, 'Ivan', 'Mažuranić', NULL)
GO
```

U tablicu nije moguće unositi podatke ako krše referentni integritet, odnosno kada kršimo neko od ograničenja. Ukoliko pokušamo unijeti tako zapis, javit će nam se greška i pokušaj upisivanja bit će neuspješan.

2.10.2. Izmjena

Izmjena podataka koje imamo u tablici česta je, bilo da se radi o ažuriranju, odnosno ispravicima netočno unesenih podataka ili ako prilikom unosa nismo imali sve podatke, pa sad moramo unijeti u već definirani red dodatak zapisima. Bilo kakve izmjene na zapisu radimo **UPDATE** naredbom, koja ima sljedeću sintaksu:

```
UPDATE
[ TOP ( expression ) [ PERCENT ] ]
{ { table_alias | <object> | rowset_function_limited
  [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
  }
  | @table_variable
}
SET
{ column_name = { expression | DEFAULT | NULL }
  | { udt_column_name. { { property_name = expression
                        | field_name = expression }
                        | method_name ( argument [ ,...n ] )
                      }
  }
  | column_name { .WRITE ( expression , @Offset , @Length ) }
  | @variable = expression
  | @variable = column = expression
  | column_name { += | -= | *= | /= | %= | &= | ^= | |= }
expression
  | @variable { += | -= | *= | /= | %= | &= | ^= | |= } expression
expression
  | @variable = column { += | -= | *= | /= | %= | &= | ^= | |= }
expression
  } [ ,...n ]

[ <OUTPUT Clause> ]
[ FROM { <table_source> } [ ,...n ] ]
[ WHERE { <search_condition>
  | { [ CURRENT OF
      { { [ GLOBAL ] cursor_name }
        | cursor_variable_name
      }
    }
  ]
}
]
[ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]
```

Ukratko, najčešći oblik je:

```
UPDATE <ime tablice> [ ( popis_stupaca ) ]
SET ime_stupca = nova_vrijednost
[, ime_stupca = nova_vrijednost ...]
[WHERE uvjet]
```

Za primjer, ako želimo promijeniti mjesto stanovanja u zapisu koji ima IDOsoba 4, onda ćemo to napraviti na sljedeći način:

```
USE Trgovina
UPDATE Osoba
SET Mjesto_stanovanja='Zagreb'
WHERE IDOsoba=4
GO
```

S WHERE ćemo se detaljnije pozabaviti u sljedećem poglavlju dohvata iz baze podataka. No svakako ga moramo spomenuti i sad. Izmjena će se napraviti u zapisima koji odgovaraju uvjetu upita, koji je naveden u WHERE dijelu. Svi zapisi koji zadovoljavaju taj uvjet imat će izmjene na podacima koje smo napomenuli u SET dijelu.

- Ukoliko nemamo naveden uvjet, izmjene će se dogoditi na svim podacima koji su u navedenoj tablici!

To je vrlo važno napomenuti, jer jednom kad se naredba izvrši, nema vraćanja podataka osim ako ne uzimamo podatke iz neke prijašnje sigurnosne kopije. Zato nikada ne radite na produkcijskim bazama podataka, da se ne dovedete u nezgodne situacije.

2.10.3. Brisanje

Brisati se može gotovo sve u bazama podataka, odnosno svi objekti koji postoje. Brisati možemo podatke, cijele tablice, baze podataka...

Svakako, prije nego što krenemo s brisanjem, opet napomena, jednom kada obrišete nešto iz baze podataka, nema vraćanja! Vraćanje opet možemo vršiti iz neke sigurnosne kopije ili ponovnim kreiranjem obrisanih objekata. Znači oprez prilikom brisanja obavezan je.

Za brisanje podataka iz tablice koristimo naredbu **DELETE** sa sljedećom sintaksom:

```
[ WITH <common_table_expression> [ ,...n ] ]
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM ]
    { { table_alias
      | <object>
      | rowset_function_limited
      [ WITH ( table_hint_limited [ ...n ] ) ] }
      | @table_variable
    }
    [ <OUTPUT Clause> ]
    [ FROM table_source [ ,...n ] ]
    [ WHERE { <search_condition>
              | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                    | cursor_variable_name
                  }
                }
            ]
          }
    ]
    [ OPTION ( <Query Hint> [ ,...n ] ) ]
[; ]
```

Odnosno skraćeni (najčešći) oblik je sljedeći:

```
DELETE FROM <ime tablice> [WHERE uvjet]
```

Ukoliko želimo obrisati osobu s IDOsoba 4, napravili bismo to na sljedeći način:

```
USE Trgovina
DELETE FROM Osoba WHERE IDOsoba=4
GO
```

Za where vrijedi slično kao i za update, time ćemo se detaljnije pozabaviti u dijelu s dohvaćanjem iz baze podataka, ali važnije je napomenuti sljedeće:

- Ako nemamo naveden uvjet, obrisat će se svi podaci koji su u navedenoj tablici!

Ima i slučajeva gdje se zapisi neće obrisati iako su navedeni, a to je ako se povrijedi referentni integritet. U tom slučaju, slično kao i kod unosa podataka, izbacit će se

greška prilikom pokušaja izmjena. Povreda se može dogoditi često i zbog foreign key constrainta u drugoj tablici. Ukoliko u drugoj tablici imamo strani ključ koji se referencira na primarni ključ koji pokušavamo obrisati, neće se moći izvršiti brisanje (osim ako nemamo definirane okidače koji će napraviti kaskadne izmjene ili brisanje, ali o tome kasnije).

Naredba DELETE briše samo podatke, no nije uvijek efikasna za brisanje cijele tablice. Ključno je reći da briše samo podatke, ali briše zapis po zapis unutar tablice i za svaki obrisani zapis u dnevnik transakcija upisuje jednu stavku. Za brisanje, odnosno čišćenje cijele tablice efikasnija nam je **TRUNCATE** naredba jer oslobađa sve zapise povezane s tablicom (npr. interni *seed* za auto inkrement primarnog ključa itd.).

Sintaksa naredbe sljedeća je:

```
TRUNCATE TABLE
    { database_name.schema_name.table_name | schema_name.table_name |
table_name }
    [ WITH ( PARTITIONS ( { <partition_number_expression> | <range> }
    [ , ...n ] ) ) ]
[ ; ]
```

Odnosno skraćeno ćemo koristiti:

```
TRUNCATE TABLE <ime tablice>
```

Ako želimo obrisati sve podatke iz tablice Osoba, napravili bismo to na sljedeći način:

```
USE Trgovina
TRUNCATE TABLE Osoba
GO
```

Brisanje tablice nećemo raditi naredbom delete, nego naredbom drop koju smo već naveli ranije. Također, naredbom drop brišemo i baze podataka.

2.11. Dohvat podataka iz baze

Dosad smo naučili kako ubacivati podatke u bazu podataka, kako ih mijenjati i brisati, a sada je vrijeme da prođemo jednu od najbitnijih naredbi (jer se najviše koristi), a to je naredba **SELECT**, koja ima sljedeću sintaksu:

```
SELECT
  [ TOP n ]
  [ ALL | DISTINCT ]
  { * | popis_stupaca }
  [ INTO nova_tablica ]
FROM { ime_tablica } [ ,...n ]
[ WHERE <uvjet za traženje> ]
[ GROUP BY <uvjet grupiranja> ]
[ HAVING <uvjet traženja> ]
[ ORDER BY <izraz za sortiranje> ]
```

Malo pojednostavljeno:

```
SELECT
  [ TOP n ]
  [ ALL | DISTINCT ]
  { * | popis_stupaca }
  [ INTO nova_tablica ]
FROM { ime_tablica } [ ,...n ]
[ WHERE <uvjet za traženje> ]
[ GROUP BY <uvjet grupiranja> ]
[ HAVING <uvjet traženja> ]
[ ORDER BY <izraz za sortiranje> ]
```

Najosnovnija verzija upita na primjeru upita na tablici osoba sljedeća je:

```
USE Trgovina
SELECT * FROM Osoba
GO
```

Na ovakav način dobivamo sve podatke iz tablice Osoba. Sa * označavamo da želimo sve stupce iz upita. Na ovakav način izlistat će nam se stupci kako su napravljeni u bazi podataka. Ukoliko želimo drugi raspored stupaca ili nam ne trebaju svi stupci, moramo ih nabrajati kao u sljedećem primjeru:

```
USE Trgovina
SELECT Ime, Prezime, Mjesto_stanovanja FROM Osoba
GO
```

Naravno, u prikazu rezultata možemo promijeniti imena stupaca pomoću ključne riječi AS u nabranjanju stupca kao u sljedećem primjeru:

```
USE Trgovina
SELECT IDOsoba AS 'Šifra',
       Ime AS 'Ime osobe',
       Prezime AS 'Prezime osobe',
       Mjesto_stanovanja AS 'Mjesto stanovanja'
FROM Osoba
GO
```

Promjena stupaca nikako ne znači mijenjanje imena stupca u samoj bazi, to se odnosi samo na imena stupaca u prikazu rezultata. Iznimno je korisno kada želimo ubaciti neki izraz koji će se računati kao u sljedećem primjeru:

```
USE Trgovina
SELECT IDOsoba AS 'Šifra',
       IDOsoba*10 AS 'Šifra puta 10',
       Ime+' '+Prezime AS 'Ime i prezime',
       Mjesto_stanovanja AS 'Mjesto stanovanja'
FROM Osoba
GO
```

Sjetimo se da u bazi ne bi smjelo biti nikakvih tranzitivnih ovisnosti, pa prema tome nećemo spremati stupce koji su međusobno ovisni. Primjerice, ovo nam je česta stvar u računima, gdje nikad ne spremamo ukupnu vrijednost računa, nego svaki put računamo ukupnu vrijednost na temelju spremljenih pojedinačnih podataka.

2.11.1. Filter upita

Često ne želimo imati sve podatke koji se nalaze u nekoj tablici ili tablicama. I zašto bismo ih imali? U suštini, uglavnom će nam se upiti svoditi na traženje nekog specifičnog podatka. Da bismo filtrirali niz podataka, koristimo ključnu riječ **WHERE**. Spominjali smo je i u prijašnjim dijelovima kod update i delete, a sad je vrijeme da malo detaljnije uđemo u to.

Na temelju istinitosti uvjeta unutar WHERE dijela upita izabiremo koji red uopće želimo kao rezultat. Ukoliko je uvjet evaluiran kao istinit, red će biti prikazan u rezultatu, a ukoliko je evaluiran kao laž, rezultat neće biti prikazan. Recimo da želimo popis osoba koje žive u Osijeku. Taj bi nam upit izgledao ovako:

```
USE Trgovina
SELECT * FROM Osoba
WHERE Mjesto_stanovanja='Osijek'
GO
```

Prilikom izgradnje uvjeta možemo koristiti sve stupce iz tablice, kombinirati s raznim stupcima unutar istog upita, koristiti funkcije, operatore spajanja itd. Unutar where uvjeta možemo koristiti i kompliciranije podupite kojima možemo tražiti pripadnost u skupovima i još mnogo toga. Rad s uvjetima definitivno je jedan od kompliciranijih dijelova i ovdje dolazi do izražaja logika rada baze podataka.

2.11.2. Operatori

Operatore bi, da bismo ih lakše razumjeli, trebali grupirati. Operatora postoji mnogo, pa onda moramo posebno paziti na njihove prioritete i slijed izvođenja. Operatori uvelike ovise o podatkovnim tipovima s lijeve i desne strane. Svakako rezultati mogu varirati ovisno o operatoru i podatkovnim tipovima koje imamo. U nekim slučajevima zbog različitih podatkovnih tipova ili nedefiniranosti određene operacije na njima neće biti moguće koristiti sve operatore. Zato ćemo operatore po grupama malo pregledati.

Važna napomena, neke operatore možemo koristiti prilikom obrade podataka u prikazu kod nabiranja stupaca u select dijelu.

Aritmetički operatori:

Ovo su nam svakako najbolje poznati operatori. U ovoj skupini nalaze se operatori zbrajanja (+), oduzimanja (-), množenja (*), dijeljenja (/) i ostatka prilikom dijeljenja (%), koji je poznat kao modulo.

Rezultat ovih operatora neće uvijek biti onaj koji očekujemo te će uvelike ovisiti o

podatkovnom tipu. Uzmimo primjer samo brojeva, nije isto raditi operacije dijeljenja s cijelim brojevima ili realnim (decimalnim) brojevima. Ukoliko radimo dijeljenje s cijelim brojevima, npr. $3/2$, rezultat će biti cijeli broj, u ovom slučaju 1 (nije greška, samo 1, bez decimala), jer je operator dijeljenja definiran da ako dijelimo dva cijela broja, za rezultat dobivamo cijeli broj. No ukoliko dijelimo dva decimalna broja, možemo očekivati da će rezultat biti decimalni broj, pa u tom slučaju $3.0/2.0$ daje rezultat 1.5, jer je tako definiran. Ako uzmemo iste brojeve u obzir, u skupu cijelih brojeva možemo koristiti operator ostatka prilikom dijeljenja, dok u skupu realnih brojeva taj operator nema smisla.

Primjer korištenja aritmetičkih operatora:

```
USE Trgovina
SELECT IDOsoba*10 AS 'Šifra puta 10'
      Ime+ ' ' +Prezime AS 'Ime i prezime'
FROM Osoba
WHERE IDOsoba%2=0
GO
```

Relacijski operatori:

Relacijski operatori služe nam za usporedbu te se zato često zovu i operatori uspoređivanja. U ovoj grupi imamo operatore manje (<), veće (>), manje ili jednako (<=), veće ili jednako (>=), jednako (=), različito (<> ili !=).

U slučajevima logičkih operacija rezultat nam je uvijek logički podatkovni tip, odnosno rezultat može biti samo istina ili laž. Također se uglavnom u svim izračunima prihvaća da je 0 laž, a bilo što različito od 0 istina. Svakako uzmite jednu stvar u obzir, NULL nije 0, ali nije ni laž.

Logički operatori:

Logičke operatore koristimo, kako naslov kaže, kada spajamo više izraza logičkog tipa u jedan jedinstveni izraz. Često će se dogoditi da imamo više uvjeta koji moraju biti istiniti, odnosno moramo na temelju relacije između više uvjeta donijeti odluku o istinitosti konačnog uvjeta. Npr. kad trebamo iz tablice osoba uzeti popis svih osoba koji stanuju u Osijeku ili Vinkovcima, to bismo napravili na sljedeći način:

```

USE Trgovina
SELECT *
    FROM Osoba
    WHERE Mjesto_stanovanja='Osijek' or Mjesto_stanovanja='Vinkovci'
GO

```

Operatori koje ćemo koristiti jesu operatori logičke algebre. Osnovni su operatori logičke algebre I (and), ILI (or) i NE (not). Prisjetimo se malo logičke algebre sa sljedećom tablicom:

A	not A (NE)	B	A and B (I)	A or B (ILI)
false	true	false	false	false
false	true	true	false	true
true	false	false	false	true
true	false	true	true	true

Naravno, sve logičke funkcije možemo dobiti pomoću ovih triju osnovnih.

Ostali operatori:

Ove prve tri skupine nisu svi operatori, ali možemo reći da su najčešće korišteni.

Od ostalih operatora mogli bismo navesti nekoliko posebnih operatora kao što su **IN**, **LIKE**, **BETWEEN** i **EXISTS**.

Operator **IN** koristimo kad želimo provjeriti je li jedna vrijednost sadržana u nekom drugom skupu. Primjerice, možemo provjeriti nalazi li se neki podatak u popisu podataka, kao na sljedeći način:

```

USE Trgovina
SELECT *
    FROM Osoba
    WHERE Mjesto_stanovanja IN ('Osijek', 'Vinkovci', 'Našice')
GO

```

Naravno, budući da je rezultat upita je li nešto u skupu ili nije, rezultat je istina ili laž, te s time često kombiniramo logičke operatore da npr. provjerimo je li nešto ili nije u skupu. Npr. ako želimo popis osoba koji ne žive u navedenim gradovima, možemo to napraviti na sljedeći način:

```
USE Trgovina
SELECT *
      FROM Osoba
     WHERE Mjesto_stanovanja NOT IN ('Osijek', 'Vinkovci', 'Našice')
GO
```

Najveća moć IN operatora jest to što skup može biti neki drugi upit, odnosno druga SELECT naredba. U sljedećem primjeru uzet ćemo popis osoba koje nisu u trgovini ništa kupile:

```
USE Trgovina
SELECT *
      FROM Osoba
     WHERE IDOsoba NOT IN (SELECT IDKupac
                          FROM Narudzba)
GO
```

Samo moramo biti oprezni, rezultat podupita mora davati popis samo po jednom atributu, ne možemo navesti sve stupce tablice koju dohvaćamo.

Operator **LIKE** koristi nam za uspoređivanje niza znakova (stringova). Koristimo ga tako da zadamo određeni uzorak prema kojem se uspoređuju dani nizovi. Rezultat LIKE operatora može biti samo istina ili laž. Operator LIKE nema puno smisla koristiti u prikazu rezultata, ali često se koristi u WHERE dijelu tijekom rješavanja uvjeta.

Sintaksa LIKE operatora sljedeća je:

izraz [NOT] LIKE uzorak

Tijekom rada s uzorkom koristit ćemo sljedeće znakove za označavanje u uzorku:

Znak	Opis	Primjer
%	Bilo koji niz prazan ili više znakova	Mjesto_stanovanja LIKE '%grad%' Ovako bismo odabrali sva mjesta stanovanja koja u sebi sadrže grad.
_	Podcrtava (underscore) označava bilo koji jedan znak (samo jedan!)	Prezime LIKE '_o%' Ovako bismo odabrali sva prezimena kojima je drugi znak o.
[]	Bilo koji jedan znak u rasponu ([a-h] ili skupu ([abcdef])	Ime LIKE '[PV]%' Ovako bismo odabrali sva imena koja počinju s P ili V. Ukoliko napišemo '[P-V]%', minus između označava raspon od P do V.
[^]	Bilo koji znak koji nije u rasponu ili skupu. Slično kao prethodno, samo negacija.	Prezime LIKE '[^BO]%' Ovako bismo odabrali sva prezimena koja ne počinju s B ili O.

Nekoliko primjera:

U tablici Artikl naći sve artikle koji u nazivu imaju riječ procesor:

```
USE Trgovina
SELECT * FROM Artikl where Naziv LIKE '%procesor%'
GO
```

Svakako treba napomenuti da kako na razini baze, tako i na razini uspoređivanja stringova DBMS ne razlikuje velika i mala slova.

U tablici Osoba naći sve osobe čije prezime završava slovom ć ili š:

```
USE Trgovina
SELECT * FROM Osoba where Prezime LIKE '%[ćš]'
GO
```

Operator **BETWEEN** koristimo da bismo provjerili je li atribut u određenom rasponu ili nije. Primjer ako želimo osobe koje imaju IDOsoba između 13 i 25:

```
USE Trgovina
SELECT * FROM Osoba where IDOsoba BETWEEN 13 AND 25
GO
```

Dodatno možemo koristiti i operator **NOT** da bismo rekli da vrijednost atributa nije u tom rasponu.

Za kraj, svakako moramo paziti na prioritete operatora, odnosno kojim će se redosljedom prilikom kompleksnog izraza razrješavati operatori. Postoji strogo definiran redosljed izvršavanja operacija, ali ako niste sigurni, svakako možete koristiti zagrade. Ako imamo operatore istog prioriteta izvršavanja, vrijedi matematičko pravilo da se rješava s lijeva na desno. Prioriteti operatora: prvo se rješavaju aritmetički operatori (kao u matematici, množenje i dijeljenje imaju prednost ispred zbrajanja i oduzimanja), zatim relacijski, zatim logički operatori (gdje prednost ima I ispred ILL), pa onda ostali operatori.

2.11.3. Grupiranje rezultata upita

Ukoliko želimo grupirati rezultate, to je moguće na više načina. Možemo koristiti ključnu riječ **DISTINCT** ili ključnu riječ **GROUP BY**. Recimo da želimo doći do popisa mjesta stanovanja svih osoba iz tablice Osoba:

```
USE Trgovina
SELECT DISTINCT Mjesto_stanovanja FROM Osoba
GO

--ili
SELECT Mjesto_stanovanja FROM Osoba
GROUP BY Mjesto_stanovanja
GO
```

U prvom slučaju s ključnom riječi **DISTINCT** rekli smo da želimo da se odaberu različiti redci. Ovako DBMS odabire sva mjesta stanovanja pa prikaže samo ona koja su različita. Svakako treba napomenuti da je NULL vrijednost posebna vrijednost i ako ih ima više, prikazat će se samo jedanput. U drugom slučaju s **GROUP BY** odabrali smo sva mjesta stanovanja, pa onda kreirali grupe prema mjestu stanovanja.

Iako naizgled rade istu stvar, mehanizmi rada ipak su različiti. U prvom slučaju čak bismo mogli koristiti i * za odabir, dobili bismo tako sve retke jer DISTINCT gleda da bude jedinstven rezultat upita, a budući da bismo imali i primarni ključ u izboru, svaki redak bio bi jedinstven. S GROUP BY ne možemo koristiti * jer se rezultatni podaci

grupiraju te se vraća samo jedan podatak iz tako dobivenog skupa. To nam omogućuje dobivanje dodatnih informacija o tim grupama, odnosno omogućuje nam da svaku pojedinu grupu posebno analiziramo pomoću agregatnih funkcija ili drugih funkcija, no o tome ćemo kasnije kad budemo spominjali funkcije.

2.11.4. Odabir 'naj' rezultata

Where svakako utječe na rezultat koji ćemo dobiti, no ne zanimaju nas uvijek svi rezultati, nego samo možda prvih nekoliko. Za dobivanje prvih nekoliko rezultata ne koristimo where dio select naredbe, nego koristimo ključnu riječ **TOP** odmah nakon SELECT dijela. Ključnu riječ TOP možemo koristiti na nekoliko načina ovisno o tome želimo li točan broj ili postotak.

```
USE Trgovina
--Prvih 5 zapisa
SELECT TOP 5 * FROM Osoba
GO
--Prvih 10% zapisa
SELECT TOP 10 PERCENT * FROM Osoba
GO
```

Sljedeće naredbe dohvaćaju kako je u komentaru rečeno. Ako kažemo TOP pa N, to znači N broj zapisa. Ukoliko želimo da nam to bude postotak, samo kažemo s **PERCENT** da je riječ o postotku, jer znak % ima drugo značenje, pa ga ne smijemo ovdje koristiti!

Ključnu riječ TOP dobro je koristiti s ORDER BY naredbom jer nam omogućuje sortiranje podataka već unaprijed.

2.11.5. Sortiranje rezultata upita

U slučajevima kad želimo sortirati podatke, u SELECT naredbi dodamo ORDER BY naredbu. Za primjer, ako želimo osobe posložiti po prezimenu, napravili bismo to na sljedeći način:

```
USE Trgovina
SELECT * FROM Osoba ORDER BY Prezime
GO
```

Sortiranje možemo napraviti u dva smjera, silazno ili uzlazno. Ako sortiramo silazno, dodajemo ključnu riječ **DESC** (*descending*), a ako sortiramo uzlazno, dodajemo ključnu riječ **ASC** (*ascending*).

U GROUP BY dijelu možemo uvijek sortirati po više kriterija. Recimo, za primjer, da želimo sortirati po prezimenu silazno, a ukoliko imaju isto prezime, onda po imenu uzlazno:

```
USE Trgovina
SELECT * FROM Osoba ORDER BY Prezime DESC, Ime ASC
GO
```

Ukoliko ne navedemo redoslijed sortiranja, uzima se da je redoslijed rastući, pa prema tome ASC ne moramo navoditi.

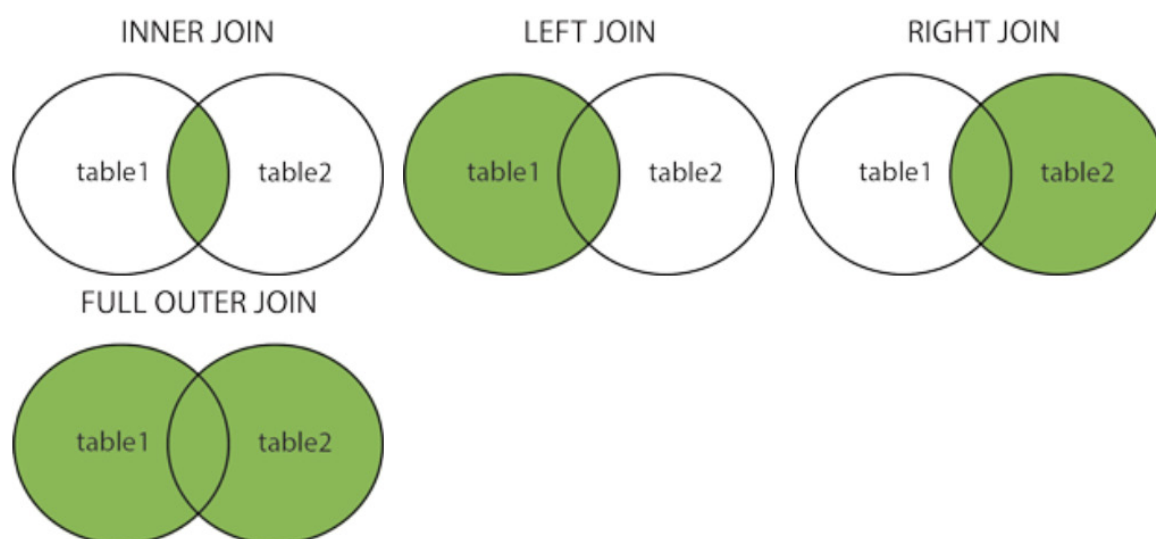
2.11.6. Upiti nad više tablica

U dosta slučajeva trebat ćemo pogledati podatke iz više tablica. Za te slučajeve koristit ćemo spajanja tablica, odnosno upite nad više tablica.

Spajanja tablica imamo nekoliko vrsta:

- cross-join
- unutarnje spajanje (inner join)
- vanjsko spajanje, lijevo i desno (left outer join i right outer join)
- puno spajanje (full join)
- samospajanje (self-join).

Ilustracija spajanja vidljiva je na slici 7.:



Slika 7. Ilustracija različitih spajanja

Najjednostavnija vrsta spajanja takozvani je cross-join. Njega možemo gledati kao čisto kombinaciju tablica, gdje svaki redak iz jedne tablice spajamo sa svakim retkom iz druge tablice.

Ako imamo tablice Osoba i Mjesto, cross-join bismo napravili na sljedeći način:

```
USE Trgovina
SELECT * FROM Osoba,Mjesto
GO
```

Ili možemo napisati ovako:

```
USE Trgovina
SELECT * FROM Osoba cross join Mjesto
GO
```

Problemi nastaju u prikazu rezultata. Ako tablica osoba ima 140 zapisa, a tablica mjesto ima 92 zapisa, rezultirajuća tablica ima $140 * 92 = 12880$ redaka. Za rezultat možemo reći da je skoro beskoristan jer smo za svaku osobu dobili po 92 retka. Ovo spajanje zove se još i Kartezijev produkt. Da bismo dobili korisne podatke, svakako trebamo navesti WHERE uvjet, gdje bi nam pod uvjetom bio naveden ključ po kojem ćemo spajati tablice. Ima slučajeva gdje cross-join dobro dođe, jer njime dobivamo sve moguće kombinacije iz obje tablice. Primjer da u jednoj tablici imamo veličine majica, a u drugoj tablici boje majica. S cross-join ćemo dobiti sve moguće kombinacije

veličina i boja majica.

Svakako je puno korisnije koristiti vanjska i unutarnja spajanja.

Unutarnje spajanje:

Unutarnje spajanje ili inner join dio je SELECT izraza koji dohvaća zapise iz više tablica spajajući ih preko jednog stupca. Pri spajanju stupaca često se može dogoditi da imamo stupce istog imena; uobičajeno je prilikom nabrajanja stupaca imenovati i tablice iz kojih se čita podatak. Prvo se navodi ime tablice, pa stavi točka, pa ime stupca (npr. Osoba.Ime). Zahvaljujući toj nomenklaturi jednostavno možemo odrediti koji točno stupac mislimo referencirati prilikom izvršavanja upita.

Primjer ćemo napraviti s tablicom Osoba, gdje ćemo ispisati sva imena, prezimena i mjesta stanovanja. Budući da u tablici osoba imamo poštanske brojeve, a ne mjesta, poštanske brojeve koristit ćemo kao stani ključ u tablici osoba koji se referencira na primarni ključ u tablici Mjesto.

```
USE Trgovina
SELECT Ime, Prezime, Mjesto.Naziv AS 'Mjesto stanovanja' FROM Osoba
INNER JOIN Mjesto ON Osoba.pbrStanovanja=Mjesto.pbr
GO
```

Prvo što se u ovom upitu napravi jest uzimanje prve tablice iz FROM dijela (u ovom slučaju Osoba) i svih redaka. Nakon toga u drugoj se tablici (Mjesto) traži redak koji odgovara na odgovarajućem mjestu uvjetu navedenom u ON dijelu upita. Tako spojeni redci stavljaju se u privremenu tablicu iz koje se konačno odabiru stupci koje smo naveli u SELECT dijelu upita.

Inače, u ovom primjeru možemo navesti Ime i Prezime stupaca jer ne postoji u drugoj tablici stupac s tim imenom. Ukoliko DBMS vidi da u obje tablice postoji jednako imenovani stupac, a nije navedeno iz koje tablice uzimamo, odbit će se izvršavanje upita.

Ako se dogodi da u jednom spajanju moramo često pisati puni naziv tablice, možemo uvesti alias umjesto punog imena, korištenjem ključne riječi AS kod imena tablice. Primjer prijašnjeg upita s aliasima:

```
USE Trgovina
SELECT o.Ime, o.Prezime, m.Naziv AS 'Mjesto stanovanja' FROM Osoba AS o
INNER JOIN Mjesto AS m ON o.pbrStanovanja=m.pbr
GO
```

Jednom kada uvedemo alias, ne smijemo više koristiti prijašnje ime, nego moramo upotrijebiti alias.

Vanjsko spajanje:

Naspram unutarnjeg spajanja koje uspoređuje vrijednosti u stupcima i rezultira vraćanjem gdje može usporediti, vanjsko spajanje kao rezultat vraća sve retke tablice na koju spajamo drugu tablicu. U slučaju lijevog vanjskog spajanja, uzimaju se svi redci prve tablice i spajaju odgovarajući redci druge tablice koji zadovoljavaju uvjet spajanja. Ukoliko nemamo retke druge tablice koji se mogu spojiti na prvu, rezultati će biti prikazani kao NULL vrijednosti za drugu tablicu. Desno vanjsko spajanje radi isto, samo se uzimaju svi redci druge tablice i na nju se spajaju odgovarajući redci prve tablice.

Uvijek je moguće da u tablici A na koju se nadovezuje tablica B ima više odgovarajućih redova. Zato kažemo da iz tablice A u rezultatu dobivamo minimalno jedan zapis (maka ne imali redova iz tablice koja se spaja na nju i u tom slučaju imamo NULL zapis), a moguće je da imamo više redova iz tablice B na koje se spajaju i toliko puta ponavljaju zapisi tablice A.

Zašto lijevo i desno vanjsko spajanje grupiramo pod isto? Jer doslovno ovdje ovisi samo redoslijed navođenja tablica. Mi uvijek možemo napraviti lijevo vanjsko da bude kao desno vanjsko, samo moramo zamijeniti redoslijed navođenja tablica.

Idemo napraviti primjer. Ispisat ćemo imena i prezimena osoba zajedno s njihovim narudžbama, poredano po datumu.

```
USE Trgovina
SELECT Ime, Prezime, BrojRacuna, DatumVrijeme FROM Osoba AS o
      LEFT OUTER JOIN Narudzba AS n ON o.IDOsoba=n.IDKupac
      ORDER BY DatumVrijeme
GO
```

Isto možemo dobiti putem desnog vanjskog, samo malo promijenimo redoslijed:

```
USE Trgovina
SELECT Ime, Prezime, BrojRacuna, DatumVrijeme FROM Narudzba AS n
      RIGHT OUTER JOIN Osoba AS o ON o.IDOsoba=n.IDKupac
      ORDER BY DatumVrijeme
GO
```

Kada koristiti jedno naspram drugog doslovno je na programeru, rezultat je svakako isti.

Ostala spajanja:

Potpuno vanjsko spajanje ili *full outer join* vraća sve retke iz prve tablice i sve retke iz druge tablice tako da spoji one koji odgovaraju prema kriteriju spajanja, a one koji ne odgovaraju nadopunjava NULL vrijednostima. Rijetko ćemo kad imati potrebu za ovakvom vrstom spajanja, no može se dogoditi.

Samospajanje ili *self join* jest slučaj gdje se tablica spaja sama sa sobom. Naspram full joina ovo ćemo trebati češće. Događa se kad želimo napraviti zavisnost tablice unutar same sebe, npr. kada radimo hijerarhiju gdje i slično. Primjer možemo dati ako npr. tablicu osoba proširimo tako da sadržava strani ključ na roditelja koji je isto zapisan u tablici Osoba:

```
USE Trgovina
SELECT o1.Ime+ ' '+o1.Prezime AS 'Osoba'
      ,o2.Ime+ ' '+o2.Prezime AS 'Roditelj'
FROM Osoba AS o1 LEFT OUTER JOIN Osoba AS o2
ON o1.IDRoditelj=o2.IDOsoba
GO
```

Kod postavljanja uvjeta bitno je tko je prvi, a tko drugi!

2.11.7. zrada tablica kao rezultat upita

Upit možemo koristiti i da bismo kreirali novu tablicu i ujedno je popunili. Korištenjem **SELECT INTO** ključnih riječi možemo eliminirati stvaranje nove tablice korištenjem naredbi **CREATE TABLE**, a zatim unosom podataka pomoću **INSERT INTO** naredbe.

Sjetite se SELECT naredbe s početka sintakse. Naredba **SELECT INTO** uzima rezultate SELECT upita te na temelju toga kreira novu tablicu. Moguće je napraviti privremenu tablicu korištenjem znaka # (*hash*) ispred imena tablice. Privremene tablice koriste se samo tijekom rada trenutne sesije (npr. unutar procedure) te kad završimo s njom automatski se brišu.

Tablice koje nastaju ovom naredbom imaju nekoliko velikih nedostataka: nemaju indekse, primarne ključeve i strane ključeve, zadane vrijednosti, naknadno uneseni podaci u originalnoj tablici neće biti vidljivi u tablici koja je nastala naredbom select

into itd. S tako kreiranom tablicom, da bismo održali referentni integritet, odnosno da bismo je normalizirali, moramo koristiti ALTER TABLE naredbu ili u samom startu napraviti CREATE TABLE naredbu, a zatim popuniti podatke pomoću INSERT naredbe. Unutar INSERT naredbe umjesto VALUES možemo navesti SELECT naredbu.

Za primjer ćemo kreirati tablicu Vinkovcani pomoću tablice Osoba:

```
USE Trgovina
SELECT * INTO Vinkovcani FROM Osoba
      WHERE Mjesto_stanovanja='Vinkovci'
GO
```

Za primjer možemo kreirati privremenu tablicu Osijecani na sljedeći način:

```
USE Trgovina
SELECT * INTO #Osijecani FROM Osoba
      WHERE Mjesto_stanovanja='Osijek'
GO
```

2.12. Ostale naredbe

Naravno, teško je proći sve naredbe koje SQL ima, pa ćemo pokušati proći još nekoliko bitnih.

2.12.1. Naredbe pristupa

Naredbe pristupa sigurnosne su naredbe kojima dajemo mogućnost upravljanja i/ili korištenja objekata na razini DBMS-a. S naredbama pristupa najčešće radi administrator DBMS-a, u smislu da će pojedinim korisnicima davati ili oduzimati mogućnosti rada na bazi podataka ili općenito na sustavu.

Sintaksa **GRANT** naredbe sljedeća je:

```
-- Simplified syntax for GRANT
GRANT { ALL [ PRIVILEGES ] }
      [ permission [ ( column [ ,...n ] ) ] [ ,...n ]
      [ ON [ class :: ] securable ] TO principal [ ,...n ]
      [ WITH GRANT OPTION ] [ AS principal ]
```

Pogledajmo na primjeru kako bismo korisniku „Pero” dali da radi upite na tablici Osoba i mogućnost izvršavanja procedure zabrani_kupca:

```
USE Trgovina;  
GRANT SELECT ON dbo.Osoba TO Pero;  
GRANT EXECUTE ON zabrani_kupca TO Pero;  
GO
```

Ako nemate korisnika, na SQL ga dodajete tako da na razini Servera, pod mapom „Security”, pa na mapu „Logins” desno kliknete i odaberete „New Login...”. U ovom primjeru koristili smo i dbo (*data base object*) pomoću kojeg pristupamo samo objektima baze podataka navedenima u USE naredbi.

Naredba **DENY** zabranjuje korisniku određena prava na razini DBMS-a. Onemogućuje korisniku da naslijedi pravo pristupa temeljem grupe korisnika ili uloge na razini sustava. Ne primjenjuje se ako je korisnik vlasnik objekta na razini DBMS-a ili ako je član skupine sistemskih administratora.

Sintaksa **DENY** naredbe sljedeća je:

```
-- Simplified syntax for DENY  
DENY { ALL [ PRIVILEGES ] }  
    [ <permission> [ ( column [ ,...n ] ) ] [ ,...n ]  
    [ ON [ <class> :: ] securable ]  
    TO principal [ ,...n ]  
    [ CASCADE ] [ AS principal ]  
[ ; ]
```

Nekoliko opaski s **DENY** naredbom:

- zabrana pristupa tablici ne može imati prioritet ispred dozvole pristupa stupcu
- zabrana kontrole pristupa bazi podataka implicitno zabranjuje spajanje na bazu podataka.

Naredba **REVOKE** uklanja prijašnju dozvolu pristupa danu naredbom GRANT ili zabranu napravljenu DENY naredbom.

Sintaksa **REVOKE** naredbe sljedeća je:


```
-- Simplified syntax for REVOKE
REVOKE [ GRANT OPTION FOR ]
{
    [ ALL [ PRIVILEGES ] ]
    |
    permission [ ( column [ ,...n ] ) ] [ ,...n ]
}
[ ON [ class :: ] securable ]
{ TO | FROM } principal [ ,...n ]
[ CASCADE ] [ AS principal ]
```

Primjer kako korisniku 'Pero' oduzeti dozvolu:

```
USE Trgovina;
REVOKE SELECT ON dbo.Osoba TO Pero;
REVOKE EXECUTE ON zabrani_kupca TO Pero;
GO
```

2.12..2. Transakcije

Transakcija je cjelina rada na bazi podataka koja se može sastojati od jedne ili više naredbi. Dosad ih nismo spominjali, odnosno svaku naredbu gledali smo kao pojedinačnu transakciju. Ukoliko je transakcija uspješna, sve promjene podataka koje su napravljene unutar transakcije bit će trajno spremljene u bazu podataka. Ukoliko transakcija ima nekih nepravilnosti ili grešaka, ona se otkazuje, odnosno bilo kakve izmjene na bazi podataka ne spremaju se i brišu se.

To nam je korisno u naprednijem programiranju na razini baze podataka. Ukoliko ne navodimo da se skup naredbi izvršava u transakciji, svaka naredba transakcija je za sebe. No u nekim slučajevima moramo izvršavati više naredbi i ako one ne budu uspješno izvršene, trebali bismo otkazati bilo kakve promjene na bazi podataka. Uzmimo u primjer našu bazu s narudžbama. Ukoliko plaćanje jedne narudžbe ne prođe provjeru, narudžba ne bi trebala biti proslijeđena kupcu, nego stornirana, odnosno promjene na bazi podataka ne bi se trebale spremati.

Naredbe za upravljanje transakcijama jesu sljedeće:

- BEGIN TRANSACTION
- COMMIT
- ROLLBACK.

Naredbom **BEGIN TRANSACTION** započinjemo blok SQL naredbi kojim želimo upravljati. Sve naredbe bit će dio tog bloka dokle god ne dođe jedna od preostale dvije naredbe.

Naredbu **COMMIT** koristimo kad smo sigurni da se sve uspješno izvršilo. Kad se izvrši **COMMIT** naredba, znači da će sve promjene nastale na bazi podataka biti trajno spremljene u bazu podataka.

Ukoliko je došlo do greške i ne želimo spremiti promjene koje su nastale nakon **BEGIN TRANSACTION** naredbe, onda koristimo **ROLLBACK** naredbu.

Primjer rada s transakcijama, gdje izvodimo više naredbi:

```
USE Trgovina
BEGIN TRANSACTION
INSERT INTO Osoba VALUES (3, 'Marko', 'Marulić', 'Zagreb');
INSERT INTO Osoba VALUES (3, 'Ana', 'Anić', 'Zagreb');
COMMIT;
GO
```

Ukoliko dođe do greške i naredba ne uspije doći do **COMMIT**, automatski se napravi **ROLLBACK**. Transakcije se najčešće koriste unutar procedura i funkcija kada radimo naprednije programiranje na bazi podataka, u kombinaciji s iznimkama na bazi podataka.

Primjer gdje ne dajemo spremanje u bazu podataka:

```
USE Trgovina
BEGIN TRANSACTION
INSERT INTO Osoba VALUES (3, 'Marko', 'Marulić', 'Zagreb');
ROLLBACK;
GO
```

2.13. Sigurnosne kopije

Sigurnosne kopije jedna su od onih stvari koje je bolje imati, a ne trebati, nego trebati, a nemati. Još jedna dobra izreka jest da postoje dvije vrste ljudi, oni koji rade sigurnosne kopije i oni koji će raditi sigurnosne kopije.

Generalna sintaksa backup naredbe sljedeća je:

```
--Backing Up a Whole Database
BACKUP DATABASE { database name | @database_name_var }
  TO <backup device> [ ,...n ]
  [ <MIRROR TO clause> ] [ next-mirror-to ]
  [ WITH { DIFFERENTIAL
          | <general_WITH_options> [ ,...n ] } ]
```

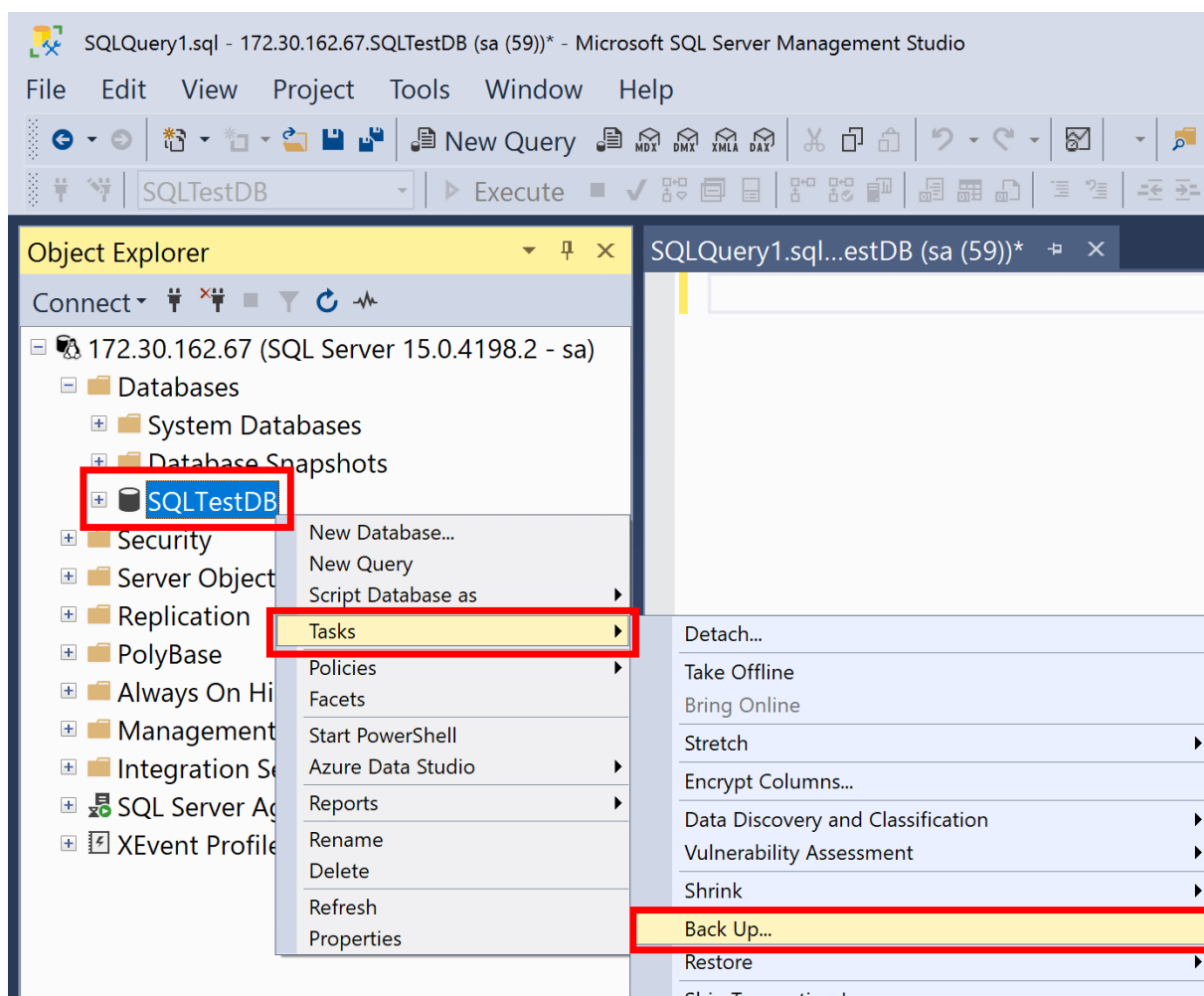
```
[;]
```

Nažalost praksu izrade sigurnosnih kopija najviše cijenimo tek kad jednom ostanemo bez svih podataka, a to je uvijek kasno. Zato pomoću SQL Servera imamo nekoliko mehanizama izrade sigurnosnih kopija, a najjednostavnije je izraditi sigurnosnu kopiju korištenjem naredbe backup database:

```
USE master;
GO
BACKUP DATABASE Trgovina
TO DISK = N'C:\trgovina.bak'
WITH NOFORMAT, NOINIT,
NAME = N'TrgovinaDB-Full Database Backup', SKIP, NOREWIND, NOUNLOAD, STATS
= 10;
GO
```

U ovom primjeru radimo backup baze Trgovina u datoteku na putanju koja je rečena u DISK opciji. Svakako, da bismo napravili backup, najbolje je koristiti master bazu podataka. Za rad backupa moramo imati nekoliko stvari na umu: moramo imati dozvolu za izradu backup baze podataka i moramo imati pravo pristupa mediju na koji želimo napraviti pohranu. Tako umjesto opcije DISK možemo navesti opciju URL i dati url udaljenog mjesta na koje želimo spremiti sigurnosnu kopiju. Dodavanjem zarezova možemo navoditi više mjesta za spremanje sigurnosnih kopija, tako da istovremeno možemo na više medija ili mjesta spremati kopije baze podataka.

Iako isto tako možemo napraviti backup pomoću grafičkog sučelja, desnim klikom na bazu podataka koju trebamo pohraniti odaberemo opciju „Tasks” i u tom podizborniku opciju „Back Up”. Nakon toga otvorit će nam se prozor s opcijama za spremanje sigurnosnih kopija, gdje svim potrebnim opcijama možemo grafički pristupiti.



Slika 8. Izrada sigurnosne kopije baze

Svaka opcija pohrane sigurnosne kopije ima mogućnost spremanja na lokalni medij ili po potrebi na neki udaljeni medij.

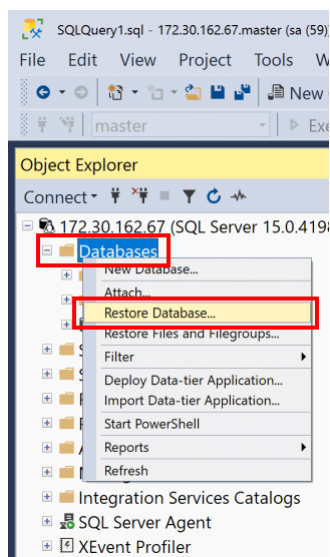
Ukoliko trebamo vratiti neku prijašnju verziju baze podataka ili na novom serveru postaviti bazu podataka iz sigurnosne kopije, možemo to napraviti na dva načina. Naredbom **RESTORE DATABASE** ili isto tako grafički.

Sintaksa RESTORE DATABASE naredbe sljedeća je:

```
--To Restore an Entire Database from a Full database backup (a Complete Restore):
RESTORE DATABASE { database name | @database_name_var }
[ FROM <backup_device> [ ,...n ] ]
[ WITH
{
[ RECOVERY | NORECOVERY | STANDBY =
{standby_file_name | @standby_file_name_var }
]
, <general WITH options> [ ,...n ]
, <replication WITH option>
, <change data capture WITH_option>
, <FILESTREAM WITH option>
, <service broker WITH options>
, \<point_in_time_WITH_options-RESTORE_DATABASE>
} [ ,...n ]
]
[;]
```

Ukoliko želimo na SQL Server vratiti bazu podataka iz naše prijašnje sigurnosne kopije, to bismo najjednostavnije napravili na sljedeći način:

```
USE master
RESTORE DATABASE Trgovina
FROM DISK = 'C:\Trgovina.bak'
GO
```



Slika 9. Vraćanje baze podataka iz sigurnosne kopije

Svakako da napomenem, umjesto opcije DISK možemo navesti opciju URL, gdje možemo dati link na bazu podataka koju treba vratiti, ukoliko smo radili sigurnosne kopije na udaljenom mjestu. Za razliku od backup opcije, nije moguće istovremeno navoditi više FROM opcija, jednostavno bi došlo do konflikta ako pokušamo vratiti bazu iz više kopija.

Također, možemo grafički vratiti bazu podataka iz sigurnosne kopije tako da kliknemo desnom tipkom miša na „Databases” pod SQL Serverom u Object exploreru pa odaberemo opciju „Restore Database...”.

Nakon ovoga otvorit će nam se više opcija rada, odnosno sve inače tekstualne opcije imat ćemo ovdje u izbornicima.

Opet za kraj napominjemo, imati sigurnosne kopije jako je dobra praksa. Preporuka je da svakako periodički izrađujete sigurnosne kopije, ali i prilikom izmjena baze podataka svakako napravite kopiju za slučaj da dođe do nepredviđenih grešaka da biste mogli vratiti posljednju poznatu stabilnu verziju baze podataka.

2.14. Pogledi

Pogledi su virtualna tablica koja predstavlja podatke iz jedne ili više tablica. Pogled nam predstavlja maskirani upit, u smislu da pogled možemo definirati kao neki kompleksni upit na jednu ili više tablica, a s njim radimo kao da radimo s jednom jednostavnom tablicom.

Poglede ćemo koristiti u nekoliko slučajeva:

- kad želimo ograničiti rad na tablici na samo neke retke ili stupce
- kad spajamo više tablica i/ili stupaca pa želimo to koristiti kao jednu tablicu
- kad želimo grupirati podatke i prikazati ih skupno umjesto pojedinačno detaljno.

S pogledima ćemo raditi na sljedeći način:

```
CREATE [ OR ALTER ] VIEW [ schema_name . ] view_name [ (column [ ,...n ] )  
]  
[ WITH <view attribute> [ ,...n ] ]  
AS select statement  
[ WITH CHECK OPTION ]  
[ ; ]
```

Pojednostavljeno, oblik koji ćemo najčešće koristiti:

```
CREATE VIEW ime_pogleda AS  
SELECT izraz
```

Primjer gdje ćemo kreirati pogled Procesori upitom na tablicu Artikli:

```
USE Trgovina  
CREATE VIEW ViewProcesori AS  
SELECT Naziv, Opic, Cijena,SifraDobavljacka FROM Artikl where Naziv LIKE  
'%procesor%'  
GO
```

Na ovaj način kreiramo novi objekt u bazi podataka, koji možemo koristiti kao tablicu. Gdje god smo koristili tablicu Artikl, sad možemo koristiti i pogled Procesori uz uvjet da nas zanimaju samo podaci o procesorima. Sad isti rezultat kao i upit koji je unutar pogleda možemo dobiti i jednostavnim upitom na pogled:

```
USE Trgovina
SELECT * FROM ViewProcesori
GO
```

Svakako dobivamo skraćenu verziju originalne tablice ovisno o tome što smo definirali prilikom kreiranja pogleda. Pogled će nam u ovom slučaju sadržavati samo procesore, tj. zahvaljujući pogledu skriveni su nam ostali redci i stupci koji nisu navedeni u pogledu. Pogledi nam tako mogu služiti kao filteri podataka. Prilikom rada upita na pogledu mi nismo ni svjesni za postojanje još nekih podataka jer prilikom rada sa SELECT ne možemo razlikovati pogled i tablicu. U praksi, da bismo razlikovali tablicu od pogleda, često se na prvo mjesto stavlja ključna riječ View (npr. ViewProcesori) da bismo tijekom rada razlikovali tablicu od pogleda.

Dok radimo na pogledu moramo biti oprezni jer preko pogleda možemo raditi modifikacije i na podacima u tablicama na koje se pogled referencira. No svakako imamo neka ograničenja:

- bilo koje modifikacije moraju referencirati stupce samo jedne tablice ukoliko pogled ima više tablica u upitu
- stupci koje mijenjamo moraju direktno referencirati podatke, ne mogu biti obrade nekih podataka
- stupci koje modificiramo ne smiju biti dio grupiranja preko GROUP BY, HAVING ili DISTINCT ključnih riječi.

Prilikom stvaranja pogleda potrebno je imati odgovarajuće ovlasti na bazi podataka. Moramo imati ovlasti za stvaranje objekata, ali i ovlasti pristupa tablicama/objektima koje definiramo unutar pogleda.

Pogled je dobro kreirati kad god postoji vjerojatnost da će nam informacija koju imamo u pogledu često i zatrebati. S time na umu treba paziti da ne pretjeramo, ne valja ih koristiti za svaku sitnicu, nego samo za često korištene informacije.

2.15. Funkcije

Funkcije su dijelovi koda koji služe za obavljanje nekog zadatka. Funkcije možemo gledati kao male ugrađene programe koje možemo pozivati da nam obave neki zadatak.

Dijelit ćemo funkcije u nekoliko kategorija. Najviše ćemo koristiti skalarne funkcije, ali vidjet ćemo i kako definirati vlastite funkcije.

Svaka funkcija prima od nijednog do više parametara i kao rezultat vraća neku izračunatu vrijednost.

2.15.1. Ugrađene funkcije

Ugrađene funkcije jesu funkcije koje su dio SQL Servera i možemo ih proizvoljno koristiti.

Tipično su to generalne funkcije koje opet možemo podijeliti u skupine ovisno o njihovoj funkcionalnosti. Najčešće ćemo koristiti ove skupine funkcija:

- matematičke funkcije
- funkcije za rad s tekстом
- funkcije za rad s datumom i vremenom
- funkcije za konverziju podataka
- ...

Funkcije unutar toga možemo još podijeliti na skalarne funkcije (funkcije koje primaju jednu vrijednost), agregatne funkcije (funkcije koje primaju grupu vrijednosti grupiranu pomoću GROUP BY, HAVING i sličnih naredbi), funkcije rangiranja, funkcije setova redova, determinističke funkcije... Nadalje, unutar tih podjela SQL Server ima još mnogo različitih skupina funkcija, ovisno o tome što trebamo (poput geografskih izračuna, geometrije...)

Matematičke funkcije koriste nam za operacije gdje nam operatori jednostavno nisu dovoljni za rad. Nabrojat ćemo neke funkcije:

- **ABS(broj)** – vraća apsolutnu vrijednost broja
- **ACOS(broj)** – vraća arkus kosinus broja
- **ASIN(broj)** – vraća arkus sinus broja
- **ATAN(broj)** – vraća arkus tangens broja
- **CEILING(broj)** – vraća realni broj zaokružen na prvi veći cijeli broj
- **FLOOR(broj)** – vraća realni broj zaokružen na prvi manji cijeli broj
- **COS(broj)** – vraća kosinus broja
- **DEGREES(broj)** – vraća stupnjeve za proslijeđene radijane
- **POWER(broj, potencija)** – vraća rezultat računanja broja na proslijeđenu potenciju
- **ROUND (broj,broj_znamenki)** – zaokružuje proslijeđeni realni broj na zadani broj znamenki
- **SQRT(broj)** – vraća korijen proslijeđenog broja
- **SIGN(broj)** – vraća predznak broja
- **PI()** – vraća broj PI (3.14...)
- **RAND()** – vraća nasumičan realni broj u rasponu od 0 do 1

Kod matematičkih funkcija zgodno je to što se mogu pozivati direktno u SELECT bez navođenja tablica:

```
SELECT PI(), POWER(2,10)  
GO
```

Rezultat:

```
-----  
3,14159265358979      1024
```

Mogu se koristiti u upitima zajedno s tablicama:

```
USE Trgovina  
SELECT PI()*IDoSoba, POWER(IDoSoba,2) FROM Osoba  
GO
```

Agregatne funkcije

Agregatne funkcije koristimo da bismo računali set vrijednosti i kao rezultat vratili jednu jedinstvenu vrijednost. Set podataka koji koristimo za ovo dobivamo tijekom grupiranja pomoću GROUP BY, HAVING i sličnih naredbi unutar SELECT naredbe.

Primjeri agregatnih funkcija:

- **COUNT(stupac)** – vraća broj redaka u grupi
- **AVG(stupac)** – vraća prosječnu vrijednost stupca unutar grupe
- **MAX(stupac)** – vraća najveću vrijednost unutar grupe
- **MIN(stupac)** – vraća najmanju vrijednost unutar grupe
- **SUM(stupac)** – vraća sumu elemenata unutar grupe

Primjer korištenja:

```
USE Trgovina
SELECT COUNT(*) FROM Osoba;
SELECT AVG(Cijena) FROM Artikl WHERE Naziv LIKE '%RTX%';
SELECT MIN(Cijena) FROM Artikl WHERE Naziv LIKE '%super%';
SELECT MAX(Cijena) FROM Artikl WHERE Naziv LIKE '%Procesor%';
GO
```

Funkcije za rad s tekstom

Ove funkcije koristimo za rad s tekstom. Funkcije u ovoj kategoriji vraćaju odgovarajući niz znakova ili neku brojčanu vrijednost ovisno o tome što tražimo. Pokazat ćemo samo dio funkcija, jer ih definitivno ima mnogo. Ukoliko parametar koji prosljeđujemo ovim funkcijama nije niz znakova, parametar će implicitno biti konvertiran.

- **DIFFERENCE(niz1,niz2)** – vraća brojčanu vrijednost razlike između dva niza znakova
- **LEN(niz)** – vraća broj znakova unutar niza
- **LEFT(niz, broj)** – vraća prvih broj znakova niza od početka (s lijeve strane)
- **LOWER(niz)** – vraća isti niz znakova, samo velike znakove pretvara u male
- **LTRIM(niz)** – vraća isti niz znakova bez razmaka na početku
- **REPLACE(niz1, niz2, niz3)** – vraća niz gdje u niz1 zamjenjuje sva ponavljanja niz2 s niz3
- **REVERSE(niz)** – okreće niz znakova

- **RIGHT(niz, broj)** – vraća prvih broj znakova niza od kraja (s desne strane)
- **RTRIM(niz)** – vraća isti niz znakova bez razmaka na kraju
- **SPACE(broj)** – vraća niz od broj razmaka
- **STR(broj,duljina,decimala)** – vraća niz znakova konvertiran iz brojeva
- **SUBSTRING(niz,start,duljina)** – vraća dio niza znakova niz, počevši od znaka start, duljine duljina
- **UPPER(niz)** – vraća isti niz znakova, samo male znakove pretvara u velike

Svakako treba napomenuti, iako imamo funkciju za konkatenciju nizova znakova (CONCAT i CONCAT_WS), najčešće koristimo operator zbrajanja (+). Već smo na nekoliko mjesta to prošli u zbrajanju imena i prezimena s razmakom između.

Funkcije za rad s datumom i vremenom

Dolazimo do nekih od najvećih problema tijekom rada većine informatičkih sustava, a to je problematika formata zapisa datuma i vremena. Svaki DBMS ima neku svoju varijaciju zapisa, ali i svaka kultura ima drukčiji način zapisa datuma i vremena (mi npr. koristimo zapis dan.mjesec.godina, dok se u SAD-u koristi format mjesec-dan-godina).

Problem najčešće nastaje tijekom utipkavanja datuma, odnosno kako interpretirati unos datuma? Ako utipkamo 3.4.2022., misli li se na 3. travnja ili na 4. ožujka? Kod SQL Servera postavljeni jezik uvijek je engleski (us_english) ako drukčije ne navedemo. Promjenu jezika možemo izvršiti jednostavnom naredbom **SET LANGUAGE** na sljedeći način:

```
SET LANGUAGE 'Hrvatski'
```

Naravno možemo i vratiti s naredbom:

```
SET LANGUAGE 'English'
```

Problem kod ovog načina jest to što ne funkcionira uvijek što se tiče formata datuma i vremena, pa onda moramo ići na direktniju metodu pomoću naredbe SET DATEFORMAT.

```
SET DATEFORMAT format
```

Ukratko, funkcije za rad s datumom i vremenom, uz napomenu da ovo nisu sve, jesu:

- **GETDATE()** – vraća trenutno sistemsko vrijeme i datum u datetime formatu
- **GETUTCDATE()** – vraća datetime format vremena u UTC formatu
- **DATEADD(dio, broj, datum)** – dodaje broj tipa dio na datum. Primjer:
DATEADD(day, 7,datum) dodaje 7 dana datumu datum. Moguće vrijednosti za dio zapisane su nakon popisa funkcija
- **DATEPART(dio, datum)** – vraća cijeli broj koji predstavlja dio datuma.
Primjer: DATEPART(m,'28.09.1984.')
- **DATEDIFF(dio, pocetni_datum,krajnji_datum)** – vraća koliko je tipa „dio“ i između početnog i krajnjeg datuma. Ne funkcionira uvijek za mjesec i godinu jer oboje mogu varirati.
Primjer: DATEDIFF(d,'24.04.1996.','29.04.1996') zbog d definiranog dijela, vraća 5 jer je 5 dana razlika između ta dva datuma.
- **DAY(datum)** – jednako DATEPART(day,datum)
- **MONTH(datum)** – jednako DATEPART(month,datum)
- **YEAR(datum)** – jednako DATEPART(year,datum)

Moguće vrijednosti za dio:

- year, yy, yyyy, y – godina
- quarter, qq, q – kvartal godine
- month, mm, m – mjesec
- day, dd, d – dan
- week, wk, ww – tjedan
- hour, hh – sat
- minute, mi, n – minuta
- second, ss, s – sekunda
- millisecond, ms – milisekunda.

Funkcije za konverziju podataka

Funkcije za konverziju podatkovnih tipova koristit ćemo u slučajevima kada nam zadani tip podataka ne odgovara u nekom izrazu. Za konverziju imamo nekoliko funkcija:

- CAST
- CONVERT
- PARSE.

U slučaju CAST i CONVERT sintaksa je sljedeća:

```
-- CAST Syntax:  
CAST ( expression AS data_type [ ( length ) ] )  
  
-- CONVERT Syntax:  
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

CAST funkcija pretvara izraz (expression) u ciljani tip podataka (data_type) zadane duljine. Npr. CAST(4 as nvarchar(4)) pretvorio bi cijeli broj 4 u tekst.

CONVERT pretvara izraz iz tipa u kojem jest u ciljani tip (data_type) tako da uzme u obzir duljinu ako se radi o tipu kod kojeg to ima smisla te poštuje stil (style) koji je također ovisan o tipu. Najčešće ćemo za stil navesti kod stila i najčešće se koristi kada treba pretvoriti datum u tekst ili kad treba formatirati realan broj.

CONVERT(nvarchar,datum,104) kao rezultat daje datum u formatu koji mi koristimo u Hrvatskoj.

Za kraj samo jedna napomena, nemaju sve pretvorbe smisla i treba paziti kad možemo napraviti pretvorbu, a kad ne. Primjer kad ne možemo napraviti pretvorbu jest da ne možemo svaki tekst pretvoriti u brojčanu vrijednost.

Korisničke funkcije

Kako možemo koristiti ugrađene funkcije, tako možemo i definirati vlastite. Prilikom izrade vlastitih funkcija rezultat funkcije može biti skalar ili tablica. Funkcije su nam jako korisni objekti unutar baze jer nam omogućuje da funkciju koristimo gdje god nam je potrebno.

Sintaksa za stvaranje funkcije sljedeća je:

```
CREATE [ OR ALTER ] FUNCTION [ schema name. ] function name
( [ { @parameter name [ AS ] [ type_schema_name. ] parameter_data_type
  [ = default ] [ READONLY ] }
  [ ,...n ]
]
)
RETURNS return_data_type
  [ WITH <function_option> [ ,...n ] ]
  [ AS ]
BEGIN
    function body
    RETURN scalar_expression
END
[ ; ]
```

Tijekom rada najčešće ćemo koristiti skraćeni oblik:

```
CREATE FUNCTION ime_funkcije
( [ { @ime_parametra podatkovni_tip
  [ = default ] [ READONLY ] }
  [ ,...n ]
]
)
RETURNS povratni_podatkovni_tip
BEGIN
    naredbe_funkcije
    RETURN povratna_vrijednost
END
[ ; ]
```

Primjer za stvaranje funkcije koja zbraja dva proslijeđena broja:

```
CREATE FUNCTION zbroji
(
    @prvi int,
    @drugi int
)
RETURNS int
BEGIN
    Declare @rez int;
    Set @rez=@prvi+@drugi;
    RETURN @rez
END
```

Korisničke funkcije možemo pozivati unutar upita ili mimo upita kao i sve druge funkcije, gdje to ima smisla. Dobra je praksa da kad vidimo da nam se jedan dio koda ponavlja učestalo u upitima, da taj izračun „upakiramo” u funkcije.

Ovako napravljene funkcije spremaju se na razini baze podataka, što znači da te funkcije možemo koristiti na bazi podataka, odnosno bez posebnih dozvola nije moguće koristiti funkcije jedne baze na drugoj bazi. Funkcije ne smiju sadržavati upite (select, update, delete, drop...).

2.16. Procedure

Procedure su slične funkcijama, samo uz jednu bitnu razliku, unutar procedura smije se koristiti upite nad bazom podataka. Procedure su nam slične kao u ostalim programskim jezicima. U njih spremamo naredbe koje možemo naknadno koristiti jednostavnim pozivom procedure na izvođenje. Naspram funkcija razlike postoje, pogotovo u načinu na koji ih pozivamo, dok je način na koji definiramo procedure sličan kao kod funkcija.

Procedure mogu:

- primati ulazne parametre i vraćati višestruke vrijednosti u obliku izlaznih parametara
- sadržavati naredbe i upite koji rade određene operacije na bazi podataka, uključujući i zvanje drugih procedura ili čak u nekim slučajevima i same sebe (rekurzivna procedura)
- vraćati statusnu vrijednost proceduri koja ju je pozvala da iskaže uspjeh ili neuspjeh naredbi unutar procedure.

Procedure se vežu za bazu podataka na kojoj su napravljene, što definitivno ima više smisla nego funkcije, s obzirom na to da procedure sadrže upite na bazu podataka na kojoj se nalaze.

Sintaksa pisanja procedure sljedeća je:

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [schema_name.] procedure_name [ ; number ]
    [ { @parameter_name [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ] [ ,...n ]
    [ WITH <procedure_option> [ ,...n ] ]
    [ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]
```

Skraćeno, najčešći oblik koji ćemo koristiti za stvaranje:

```
CREATE PROCEDURE ime_procedure(  
    [ { @ime parametra podatkovni_tip }  
    [ ,...n ]  
)  
AS BEGIN  
    sql_naredbe [;] [ ...n ]  
END
```

Prilikom imenovanja procedura treba izbjegavati započeti ime sa „sp_” jer su tako imenovane sistemske procedure SQL Servera.

Primjer procedure za dohvat osoba koje su iz prosljeđenog grada:

```
CREATE PROCEDURE dohvati_sve_osobe(  
    @stanuje nvarchar(200)  
)  
AS BEGIN  
    SELECT * FROM Osoba WHERE Mjesto_stanovanja LIKE '%' + @stanuje + '%'  
END
```

Naspram funkcija, procedure pozivamo zasebnom naredbom EXEC. Prethodnu proceduru pozvali bismo na sljedeći način:

```
EXEC dohvati_sve_osobe 'Osijek'
```

Procedure nemaju predefiniranu maksimalnu veličinu, što znači da je tijekom rada procedura ograničavajući faktor veličina baze podataka. Unutar procedura možemo raditi kompleksne programe, koristiti korisnički definirane podatkovne tipove i sistemske podatkovne tipove.

Procedure su nam efikasniji način rada nego upiti. Tijekom prvog izvođenja procedure ona se kompajlira da bi se odredio optimalni plan izvođenja i pristupa podacima. Sva-ko sljedeće izvođenje neće se više kompajlirati, nego će koristiti već određeni plan izvođenja. Time je rad procedura na bazi podataka optimiziran i preporuka je da se većina upita, pogotovo kompleksnih, napravi unutar procedura.

2.17. Okidači

Okidači su programi, odnosno naredbe koje će se automatski pokrenuti ukoliko se dogodio događaj koji ih pokreće. Oni su specijalne procedure jer ih se ne može pozvati kao ostale procedure, nego se automatski pozivaju s promjenama na tablicama. Također, naspram procedura ne mogu primiti nikakve parametre i unutar okidača ne možemo napraviti commit ili rollback transakcije.

Sintaksa izrade okidača sljedeća je:

```
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] >
}
```

Okidači su specifični zbog načina definicije, jer jedna tablica može imati više okidača koji će se pokretati u različitim uvjetima. Okidači ovise o događaju koji se dogodio i o trenutku u kojem će se pokrenuti. U sintaksnom dijagramu vidimo da okidače možemo definirati za naredbe unosa (INSERT), izmjene (UPDATE) i brisanja (DELETE), no bitno je i da za svaki od tih događaja možemo birati u kojem će se trenutku pokrenuti. Okidač može istovremeno pokrivati sve navedene naredbe. Trenutak pokretanja može biti u trenutku izmjene (FOR), nakon izvršene izmjene (AFTER) te možemo preusmjeriti izmjenu da se pokrene neki niz naredbi umjesto konkretne izmjene (INSTEAD OF).

U tijelu, kad pišemo okidač, možemo pristupiti dvjema tablicama: „deleted” i „inserted”. Tablica deleted sadrži stare vrijednosti, odnosno vrijednosti koje treba izbrisati, dok inserted tablica sadrži nove vrijednosti, odnosno vrijednosti koje su unesene ili koje je potrebno unijeti.

Primjer pisanja okidača koji će izbaciti grešku ukoliko netko pokuša unijeti parni broj za IDOsoba:

```

CREATE TRIGGER neparan_samo
ON Trgovina.Osoba
FOR INSERT, UPDATE
AS BEGIN
    declare @noviID int;
    set @noviID=(SELECT TOP 1 IDOsoba from inserted);
    if @noviID%2=0
    BEGIN
        RAISERROR ('Nedozvoljen unos ili promjena!', 16, 10);
        ROLLBACK;
    end
END;
GO

```

Kako smo već rekli, okidače ne možemo pokrenuti mi, nego se oni automatski pokreću prilikom događaja za koji su definirani. U ovom slučaju ako netko pokuša unijeti parni broj ili promijeniti IDOsoba da bude parni broj, izbacit će se pogreška i promjena neće moći biti spremljena.

Prednosti okidača:

- omogućuju nam veću kontrolu nad bazom podataka i velika su pomoć prilikom očuvanja integriteta baze podataka
- pomažu u validaciji podataka prilikom izmjena ili unosa
- pomažu pri praćenju podataka
- povećavaju efikasnost baze podataka jer se kao i procedure kompajliraju tijekom prvog izvođenja
- smanjuju kod na klijentskoj strani programa i time štede vrijeme
- lagano se održavaju.

Loše strane okidača:

- omogućuju produžene validacije, što može dovesti do usporenja sustava
- automatski se pokreću i njihovo izvršavanje nevidljivo je korisniku, pa time nije jednostavno otkrivati greške ukoliko se dogode na razini baze
- povećavaju korištenje resursa sa strane poslužitelja
- možemo definirati višestruke okidače za istu stvar
- možemo stvarati okidače samo na trenutnoj bazi, ali oni mogu referencirati objekte koji se ne nalaze u bazi.

3. POGLAVLJE:

Programski jezik C#

U OVOM POGLAVLJU NAUČIT ĆETE:

- razvojno programsko okruženje
- C# programski jezik
- varijable i podatkovne tipove
- operatore
- algoritme
- kontrolirati tok programa s grananjem i petljama
- polja
- funkcije
- osnove objektno orijentiranog programiranja.

3.1. Uvod

C# programski jezik je generalne namjene. Kad kažemo generalne namjene, misli se da je to jezik s kojim možemo razvijati aplikacije skoro svih mogućih tipova i primjena poput web-aplikacija, desktop aplikacija, mobilnih aplikacija itd. S različitim mogućnostima primjena imamo i različite pristupe rada aplikacija, koje je sve moguće napraviti u C# programskom jeziku.

Sam C# programski jezik dizajnirao je Anders Hejlsberh, koji je radio u Microsoftu 2000., i kasnije je odobren kao međunarodni standard 2002. – kao ECMA-334 (European Computer Manufacturers Association) standard, a 2003. kao ISO/IEC 23270. Microsoft je C# predstavio kao dio .NET Frameworka i Visual Studio programskog alata još prije standardizacije. Kasnije u 2004. godini C# je krenuo rutom otvorenog koda (*open source*) u sklopu Mono projekta, koji je prvi ponudio C# kao *cross platform* (podršku za više operativnih sustava) kompajler i okruženje za izvođenje programa. Desetak godina kasnije Microsoft je objavio Visual Studio Code, Roslyn kompajler i unificiranu .NET platformu koja podržava C# u obliku besplatnog, *open source* i *cross platform frameworka*. Mono je također pripojen Microsoftu, no nije u potpunosti spojen u .NET.

U trenutku pisanja aktualna verzija C# standarda jest verzija 10.0, a aktualna verzija .NET Frameworka je 6.0 i Visual Studio 2022 (17.0).

Za C# u početku se smatralo da je kopija Java programskog jezika zbog određenih principa rada. Slično kao Java programski jezik, C# je napisan sa sintaksom sličnom C/C++ programskom jeziku, za koji se može reći da je bio baza za Javu. Nećemo prolaziti razlike i sličnosti, nema smisla ako se ne znaju detalji svih programskih jezika u usporedbi.

Od verzije 2.0 iz 2005. godine C# i Java evoluirali su u različitim smjerovima te postali dva na očigled različita jezika. Dodaje se podrška za *generics* podatkovne tipove, *code generation* itd.

S verzijom 3.0 iz 2007. dolaze velike promjene u smislu dodavanja više bitnih stvari poput LINQ-a, lambda izraza, parcijalnih metoda, anonimnih tipova – kao nekih od važnijih – ali i mnogih drugih.

Verzija 4.0 izašla je zajedno s Visual Studiom 2010, a neki od važnijih dodataka su dinamičko vezanje, opcionalni argumenti i drugi. Najviše se spominjalo ubacivanje ključne riječi *dynamic* jer uvodi dosta prostora za potencijalne greške, ali rješava dosta drugih problema.

S Visual Studiom 2012 izašla je verzija 5.0 C# programskog jezika, koja dovodi jedan revolucionaran koncept, a to je asinkrono programiranje. Ovaj koncept nam rješava dosta problema sinkronizacije više različitih dretvi (*threads*), gdje ne znamo koliko će trajati izvođenje ili imamo operacije koje dugo traju.

Verzija 6.0 koja je izašla s Visual Studiom 2015 nije imala nikakve velike dodatke, nego možemo reći da je niz manjih funkcionalnosti koje su nam omogućile veću produktivnost s C# programskim jezikom. Primjeri dodataka bili bi: statični uvozi, filteri iznimaka, null propagatori, interpolacije nizova znakova, operator „nameof”, indeks inicijalizatori, await u catch/finally blokovima itd.

C# verzija 7.0 izašla je s Visual Studiom 17 i ima nekoliko revolucionarnih stvari kao što su out varijable, tuple, logičke funkcije, ref ključna riječ za rad s referencama itd. One omogućuju nove mogućnosti rada s varijablama, gdje je moguće i višestruko vraćanje vrijednosti preko tuplea.

S verzijom 7.1 krenulo je takozvano verzioniranje nakon točke. Do verzije 7 nije bilo npr. verzije 5.1 i slično. Neki od važnijih dodataka verziji 7.1 bili su asinkrona main metoda i prepoznavanje uzoraka na generičkim tipovima.

Verzija 7.2 dala je nekoliko manjih dodataka poput inicijalizatora na stackalloc poljima, korištenje fiksiranih izraza s bilo kojim tipom koji podržava uzorke, deklaraciju strukturnih tipova samo za čitanje, „in” modifikator na parametrima, referentne strukture itd.

Verzija 8.0 prva je „velika” verzija koja specifično cilja .NET Core. Neke mogućnosti oslanjaju se na nove CLR implementacije, druge na tipove u bibliotekama koje su dodane samo u .NET Core. Neke od bitnih mogućnosti u ovoj verziji bile su: članovi samo za čitanje, poboljšanja na prepoznavanju uzoraka, korištenje deklaracija, statične lokalne funkcije, nullabilne reference, asinkroni streamovi itd.

Verzija 9.0 izdana je s .NET 5 sa sljedećim dodacima: „records”, Init only setters, support for code generators itd. Ova verzija nastavlja s tri teme prijašnjih verzija, a to su:

micanje ceremonija, odvajanje podataka od algoritama i davanje više uzoraka na više mjesta.

Zadnja i aktualna verzija C#-a jest verzija 10, izašla u studenom 2021. godine. Neki od dodataka u novijoj verziji bili su: record strukture, poboljšanja strukturnih tipova, globalne using direktive, proširena svojstva uzoraka, omogućivanje konstantnih interpoliranih nizova znakova, omogućivanje pridruživanja i deklaracije u istoj dekonstrukciji, omogućivanje AsyncMethodBuilder atributa na metodama itd. Tema je nastavak rada na micanju ceremonija, odvajanje podataka od algoritama i poboljšanju performansi .NET Runtimea.

3.2. Razvojno programsko okruženje

Razvojno programsko okruženje program je koji ćemo koristiti radi lakšeg programiranja. Praktično programirati možemo pomoću bilo kojeg tekstualnog editora, samo pisati kod, a onda preko drugog programa ili naredbene linije kompajlirati ili prevesti naš kod u izvršnu datoteku ili program nekog drugog programskog jezika.

Glavne značajke razvojnog okruženja možemo podijeliti na:

- tekstualni editor
- kompajler i interpreter
- build integrator
- debugger
- označivač sintakse
- grafičko korisničko sučelje
- ostale mnogobrojne mogućnosti (podrška za odabrani jezik, uređivanje tekstualnog editora, *unit testing*, dijagrami, dodaci i proširenja, pretraga koda, završavanje koda, dijagnostički alati...).

Naravno, ne postoji samo jedno razvojno okruženje za sve. Razvojna okruženja razlikuju se u dosta stvari, od podržanih programskih jezika pa sve do razine implementacije pojedinih značajki i njihove uključenosti.

Od nekih popularnih razvojnih okruženja svakako bismo spomenuli:

- Visual Studio
- Visual Studio Code
- IntelliJ IDEA
- Aptana Studio
- PyCharm
- PhpStorm
- NetBeans
- Eclipse
- Komodo IDE
- XCode
- ...

Ukratko, ima mnogo razvojnih okruženja. Ovisno o korištenom programskom jeziku, vrsti aplikacije koju radite i potrebama koje imate (bilo operativnog sustava, značajki razvojnog okruženja itd.) odabrat ćete odgovarajuće razvojno okruženje.

Mi ćemo se ovdje koncentrirati na Visual Studio kao jedno od najkompletnijih rješenja i danas najčešće korištenih razvojnih okruženja.

3.2.1. Visual Studio

Trenutno aktualna verzija u trenutku pisanja jest Visual Studio 2022 (17.0). Iako za većinu slučajeva korištenja, kad se uči, nećete uopće primijetiti razliku između VS2019 i VS2022. Ukratko, razlike su relativno male. U novijoj verziji malo je optimizirano korištenje memorije, odnosno VS2022 koristi manje memorije u radu, direktno je integrirano nekoliko mogućnosti koje su se prije rješavale s dodacima preko ekstenzija i nekoliko unaprijedjenih dodataka. Ukratko, za početnike neće biti skoro nikakve razlike. Veće razlike vjerojatno ćete pronaći u pojedinim distribucijama unutar iste verzije Visual Studia. Isto kako, prije smo za SQL Server pokazali da ima više mogućih distribucija, ovdje moramo reći da Visual Studio dolazi isto tako.

Otvorite službenu Microsoftovu stranicu Visual Studio na sljedećoj poveznici:

<https://visualstudio.microsoft.com/>

Prvo što ćete vidjeti jest da Microsoft dijeli Visual studio na:

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code.

Mi ćemo se koncentrirati na prvi s popisa, Visual Studio. No mislim da ne treba posebno opisivati ostalo, Visual Studio for Mac upravo je to što piše, verzija Visual Studio koja je namijenjena korisnicima Apple Mac računala s OS X operativnim sustavom. Visual Studio Code samo je *editor source* koda, koji se može pokretati na više operativnih sustava (Windows, macOS i Linux), a svu ostalu funkcionalnost dobiva instalacijama odgovarajućih ekstenzija.

Odabirom Visual Studio alata vidjet ćete da imate još dodatne tri verzije Visual Studio alata:

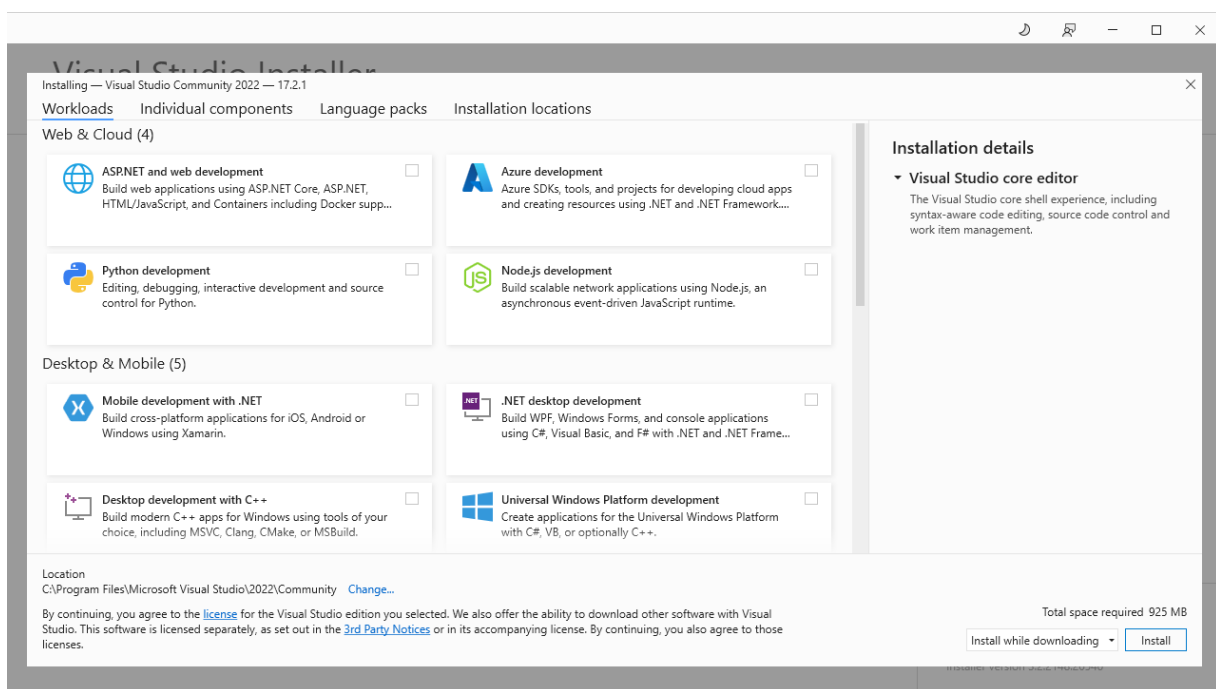
- Community
- Professional
- Enterprise.

Na poveznici <https://visualstudio.microsoft.com/vs/compare/> možete detaljnije pogledati koje su razlike, no pokušat ćemo ih u par rečenica sumirati. Prva razlika koju najviše ljudi primijeti jest cijena. Community verzija besplatna je, dok se Profesional i Enterprise plaćaju. Naravno, na što veće verzije idete, imate neke dodatne pogodnosti koje nemate u manjim verzijama, no stvarno se svodi na to da li vam to treba. Community verzija može se koristiti samo u organizacijama do pet korisnika, dok ako ste dio komercijalnog razvojnog tima s preko pet programera, onda morate ići na Professional verziju ili Enterprise.

3.2.2. Instalacija Visual Studio alata

Instalacija je relativno jednostavna. Svakako na navedenoj poveznici (<https://visualstudio.microsoft.com/vs/compare/>) možete kliknuti free download na Visual studio Community ili na sljedećem linku <https://visualstudio.microsoft.com/vs/> odabirom odgovarajuće verzije klikom na „Download Visual Studio”. Odabirom će se skinuti izvršna datoteka – pokrenite je čim preuzimanje završi. Program će tražiti da skine nekolicinu dodatnih datoteka, pa mu slobodno dopustite da preuzme.

Ukoliko nemate već instaliranu verziju, otvorit će vam se dijaloški okvir za odabir komponenti koje se trebaju instalirati, što je vidljivo na slici 10.



Slika 10. Dijaloški okvir instalacije Visual Studio 2022

Tijekom instalacije vidjet ćete da je kategorizirano u opcijama u gornjoj traci.

„Workloads” je opcija gdje možete odabrati kompletne pakete za razvoj aplikacija, odnosno za te tipove aplikacija instalirat će se odgovarajuće komponente.

U kategoriji „Individual components” možete instalirati dodatne komponente ili samo odabrane komponente koje vam trebaju. Ovdje svakako odabirete i dodatne komponente koje neće biti uključene u pojedinim kategorijama Workloads (poput Git for Windows, LINQ to SQL i drugih po potrebi).

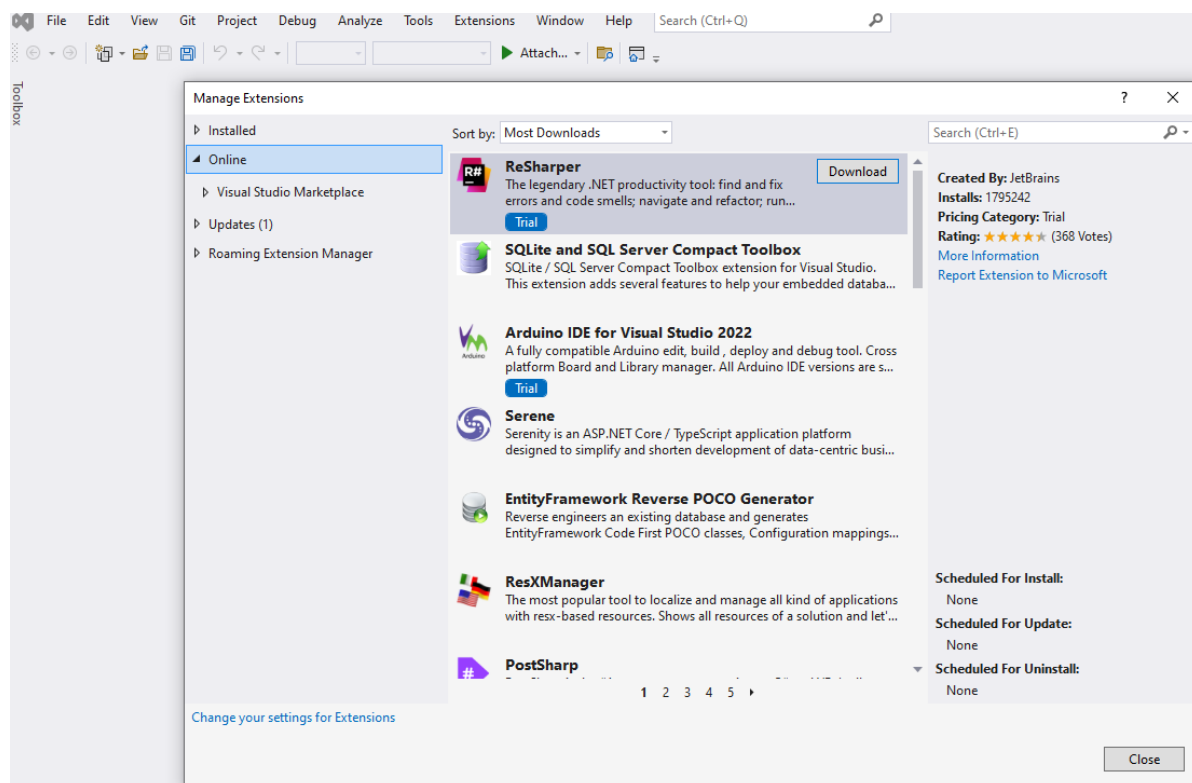
Kategorija „Language packs” sadrži jezične pakete za Visual Studio. Odustanite u samom startu od hrvatskog jezika jer nije među podržanim jezicima, bar ne u trenutku pisanja.

Zadnja kategorija, „installation locations”, koristi se za instalaciju programa i njegovih paketa, ali i za smještaj privremenih datoteka.

Instalacijski proces vam unaprijed kaže u donjem desnom kutu koliko će memorije biti potrebno za instalaciju, no svakako to će varirati ovisno o tome što ste točno odabrali (moguće i preko 80 GB). Klikom na „Install” započinje se instalacija, koja će trajati ovisno o brzini interneta i brzini računala.

Ukoliko se sjetite da neku komponentu niste odabrali, odnosno niste instalirali, uvijek možete pokrenuti „Visual Studio Installer”, koji će sad trajno biti na vašem računalu, te modificirati instalaciju tako da obrišete komponente koje vam nisu potrebne ili samo dodate komponente koje vam trebaju.

Također, dodatne komponente možemo instalirati putem dodataka. Dodatke instaliramo tako da na glavnom prozoru, u izborniku odaberemo „Extensions”, a zatim u podizborniku „Manage extensions”.



3.2.3. Prilagodba sučelja Visual Studio

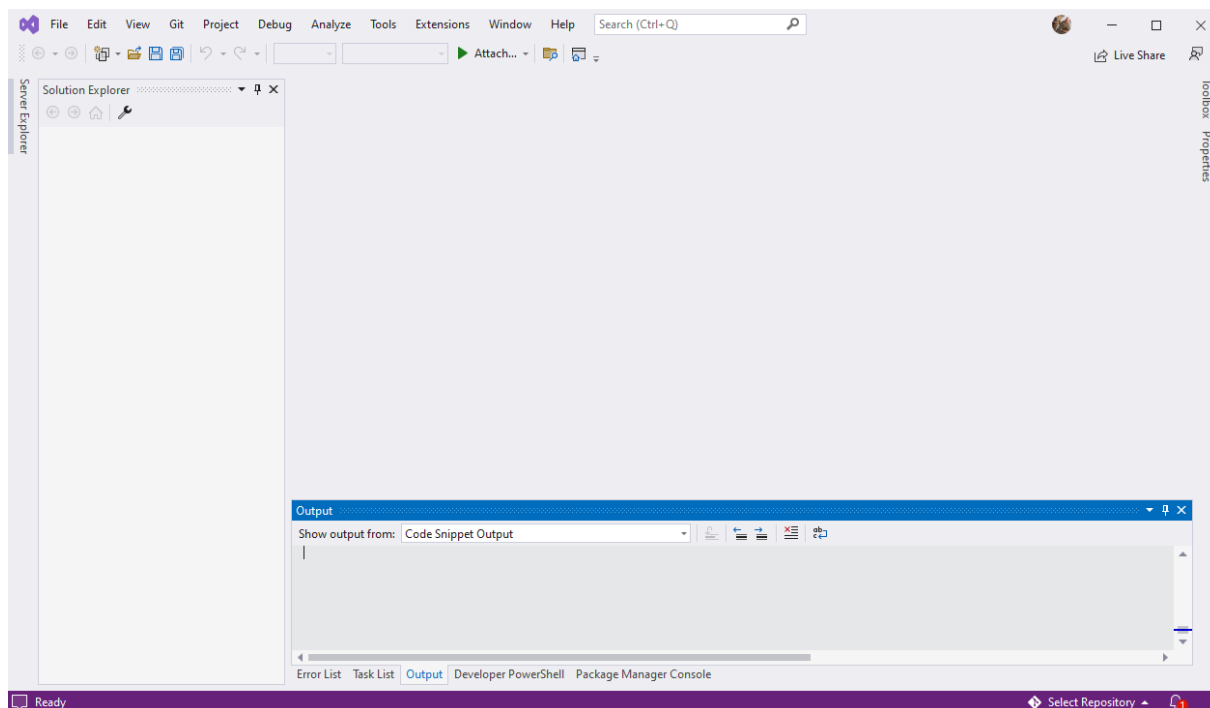
Prilagoditi sučelje svojim potrebama jedno je od najsubjektivnijih dijelova bilo kojeg programa. Nema univerzalnog recepta što je bolje, a što gore. Ovdje sve doslovno ovisi o navikama pojedinca i potrebama za pristupačnost određenih alata. Zato svakako napomena, proći ćemo kako se to radi, ali slike koje ćete vidjeti prilagođene su navici i iskustvima autora, te vi sebi možete prilagoditi kako god želite.

Svakako, prije nego što krenete u rad, jedna napomena. Iako je Community verzija besplatna, trebate imati Microsoftov račun, koji morate povezati na Visual Studio. Prilikom prvog pokretanja tražit će vas da se logirate. Ukoliko se ne želite logirati, možete 30 dana koristiti probnu verziju, no nakon 30 dana nećete više moći koristiti Visual Studio. Registracija je besplatna i neobvezujuća, a ukoliko već imate Microsoftov račun, trajat će svega koju minutu.

Treba spomenuti nekoliko dijaloških okvira koje ćete vjerojatno često koristiti:

- Solution Explorer – za prikaz projekata i datoteka unutar projekta
- Toolbox – prikaz alata koje možete uključivati u program koji radite
- Error list – popis grešaka i upozorenja nakon „izgradnje” projekta
- Output – izlaz radnji koje radite, pokazuje sve detalje radnji koje pokrenete
- PowerShell – za pokretanje PowerShell naredbi
- Properties – pregled svojstava odabranih objekata
- Task List – Iako ne jedan od obaveznih, jako koristan alat za direktno traženje „TODO:” komentara u kodu i pozicioniranje na njih.

Ovo nije popis svih dijaloških okvira, ima ih mnogo, ali većinu njih nećemo često koristiti. Naravno, raspored da će se razlikovati o odnosu na sliku kako sam ja osobno poslagao kako meni odgovara. Raspored mijenjate jednostavnih povlačenjem okvira na drugo mjesto. Također možete postaviti da dijaloški okviri budu prikazani ili da se automatski skrivaju.



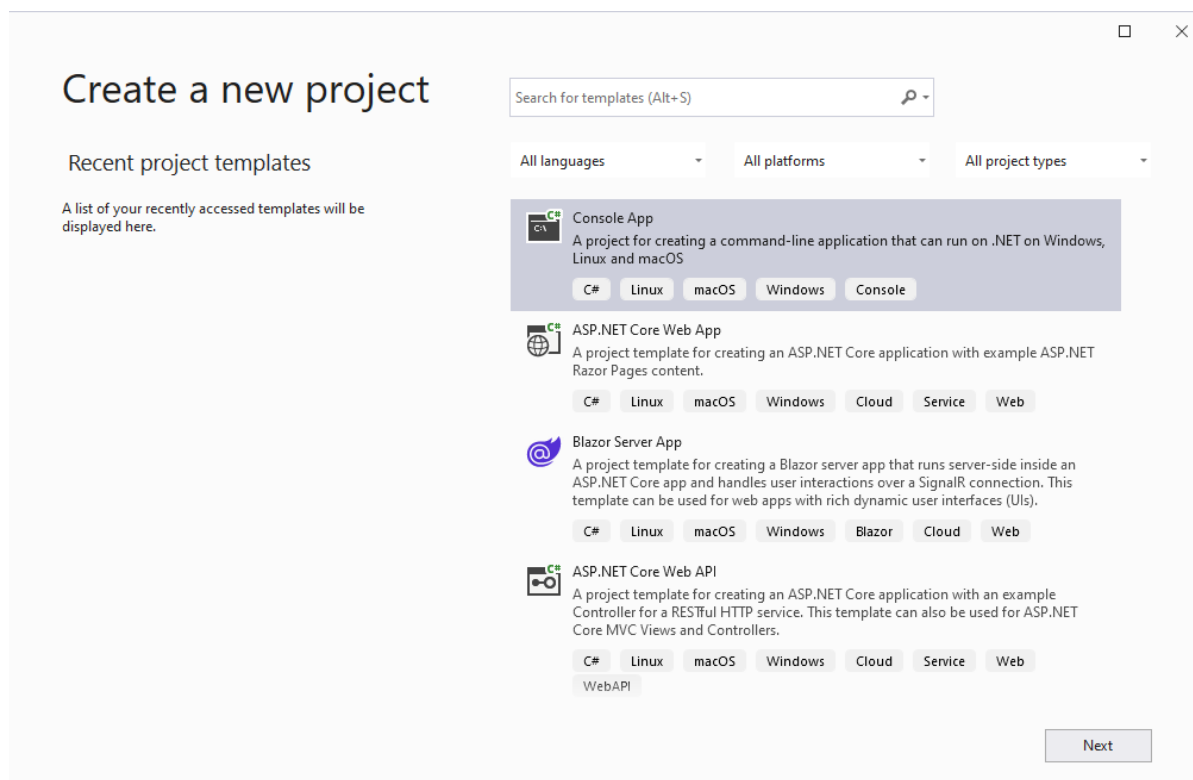
Slika 12. Izgled Visual Studio razvojnog okruženja

Klikom na View u gornjem dijelu možete odabirati koje dijaloške okvire želite prikazati, ukoliko isključite neki od okvira.

3.2.4. Konzolna aplikacija

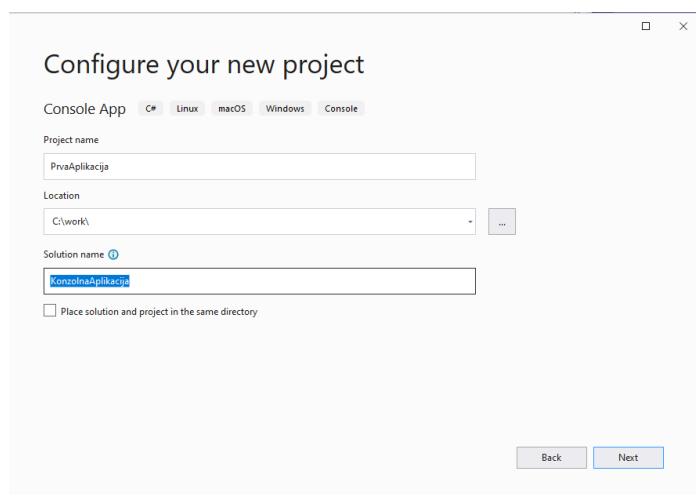
Neke dijelove Visual Studia lakše je pokazati na primjeru, pa ćemo zato napraviti prvu konzolnu aplikaciju i na njoj prikazati nekoliko tipičnih stvari i malo detaljnije objasniti dijelove razvojnog okruženja.

Da bismo napravili konzolnu aplikaciju, moramo kreirati novi projekt. Da bismo kreirali novi projekt, treba kliknuti „File”, zatim „New” pa „Project”. To će nam otvoriti dijaloški okvir kao na slici nakon ovog paragrafa. Svakako treba napomenuti da će popis mogućih projekata s desne strane ovisiti o instaliranim komponentama prilikom instalacije. Vidjet ćete padajuće izbornike u kojima možemo odabirati jezik koji ćemo koristiti, platformu za koju razvijamo aplikaciju i tip projekta. Također, vidjet ćete u opisu projekta koji možemo odabrati koji je jezik kojim možete razvijati aplikaciju, operativni sustav na kojem se može izvršavati i vrstu aplikacije u naizgled kvadratnim tagovima.



Slika 13. Odabiri predloška za stvaranje novog projekta

Odabrat ćemo „Console App”, no ukoliko vam nije ponuđeno prvo, možete taj naziv utipkati u okvir za pretragu na vrhu popisa ili odabrati za jezik C#, za platformu Windows i za tip projekta odaberete Console. Imat ćete dvije vrste konzolne aplikacije „Console App” i „Console App (.NET Framework)”. Razlika je, kako vidite po tagovima, u tome što se druga izvršava samo na Windows operativnom sustavu, a prva je kompatibilna s više operativnih sustava. Odabirom „Console App” i klikom na next pojavit će vam se okvir sa sljedećim opcijama kao na slici 14.

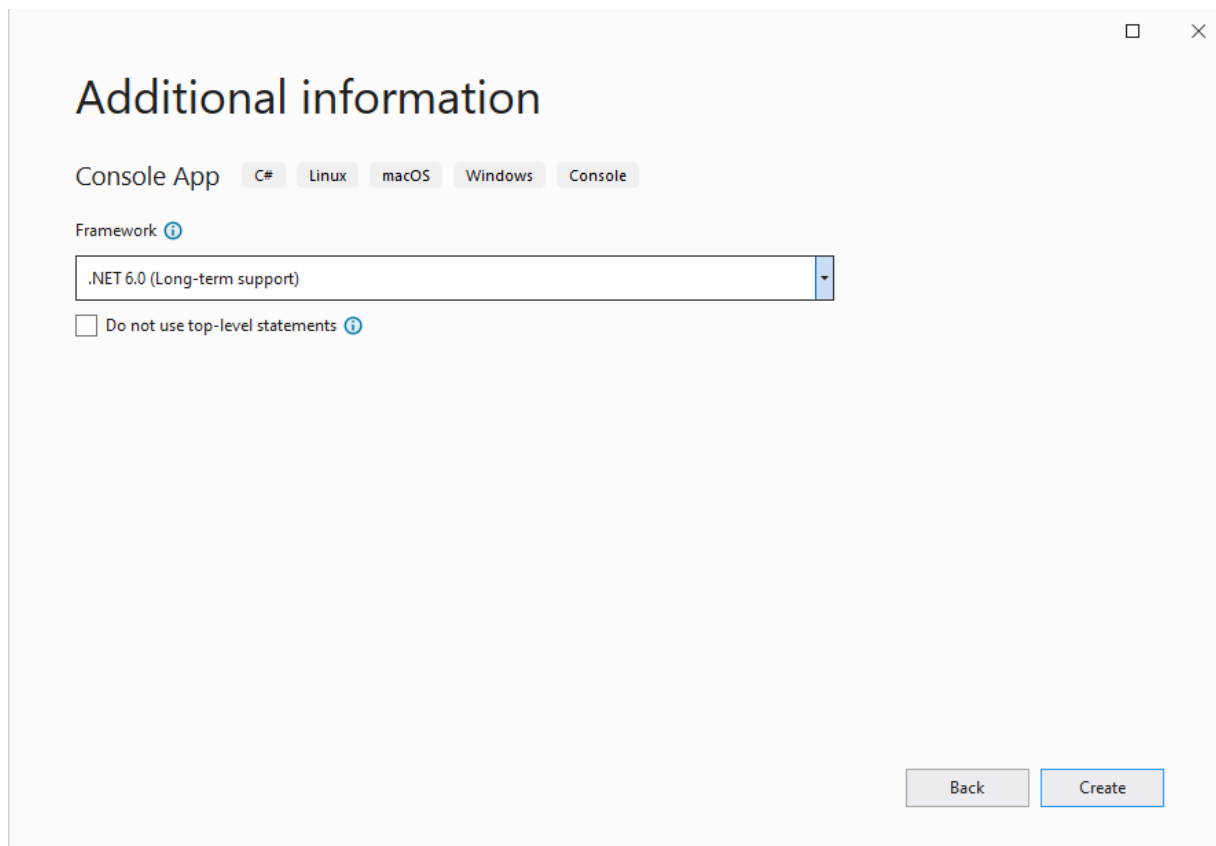


Ovdje ćete odabrati ime projekta (u ovom slučaju „PrvaAplikacija”), lokaciju na disku gdje da se projekt spremi (dobro je odabrati neku lokaciju kojoj možete lako pristupiti, ja sam navikao na C disk napraviti mapu „work” za moje aktivne projekte).

Slika 14. Podešavanje novog projekta

Solution, ili ako prevedemo kao *rješenje* s engleskog, organizacijski je element prikazan u dijaloškom okviru Solution Explorer. Organizacija je takva da u VS-u morate raditi u nekom Solutionu, koji može sadržavati od nijednog projekta do koliko god želite projekata. Projekt nam je konkretan program ili dio programa koji radimo. Ukoliko već nemate Solution otvoren, VS automatski stvara osnovni Solution, koji se može zvati kao i prvi projekt ili ga možete nazvati drukčije. Ovo je čisto organizacijski.

Klikom na „Next” ponudit će vam se framework koji će biti korišten, odnosno ciljani.



Slika 15. Odabir ciljanog frameworka

Napokon, klikom na Create VS će napraviti projekt sa svim potrebnim poveznicama na biblioteke i otvorit će vam jednu kreiranu datoteku s kodom, a to je „Program.cs” sljedećeg sadržaja:

// See <https://aka.ms/new-console-template> for more information

Console.WriteLine("Hello, World!");

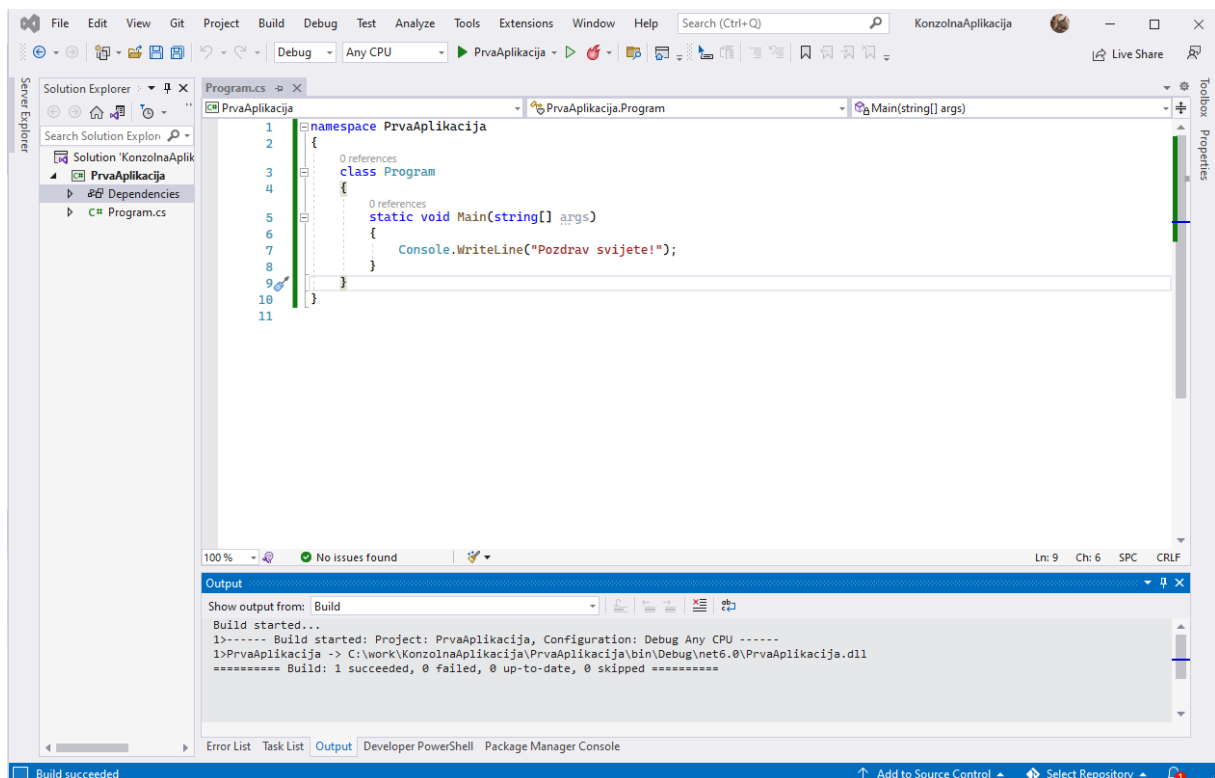
Sad ćemo taj kod zamijeniti sljedećim:

```
namespace PrvaAplikacija
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Pozdrav svijete!");
        }
    }
}
```

Na ovakav ćemo način definirati klasu „Program” sa samo jednom metodom „Main”, koja prima parametar kao argument. O tome će biti puno više priče kasnije, no zasad uzmite ovo zdravo za gotovo. **Main** nam je glavna ulazna točka aplikacije, odnosno ta metoda automatski se poziva prilikom pokretanja aplikacije. U zagradama args polje koristimo za unos argumenata, odnosno, ako se sjećate od ranije rada tekstualnih operativnih sustava ili ako ste ikad koristili komandnu liniju, nakon imena programa zadavali smo argumente. Te argumente u programu bismo ovako učitali i onda koristili unutar aplikacije.

Da bismo aplikaciju mogli pokrenuti, prvo je treba izgraditi. Izgradnju aplikacije možemo napraviti najčešće u dvije konfiguracije, debug i release, odabirom konfiguracije točno ispod „Build” opcije u glavnom izborniku. Debug konfiguracija znači da će unutar ciljane datoteke biti ubačeni dijelovi koda koji nam omogućuju praćenje rada datoteke prilikom debugiranja, da možemo izvršavati liniju po liniju i pratiti resurse koje će koristiti program. Release znači da će se finalni program pokušati optimizirati za brzo i efikasno izvođenje, ali nećemo imati mogućnost praćenja izvođenja koda. Release verzija treba više vremena da se izgradi, ali zbog optimiziranosti ćemo tako izgraditi ciljani dio za korištenje u produkcijskim uvjetima.

Build radnju možemo obaviti na dva načina. Klikom na Build, nakon čega odaberemo Build Solution da izgradimo sve projekte unutar Solutiona. Isto možemo postići jednostavnim pritiskom na funkcijsku tipku F6 na tipkovnici.



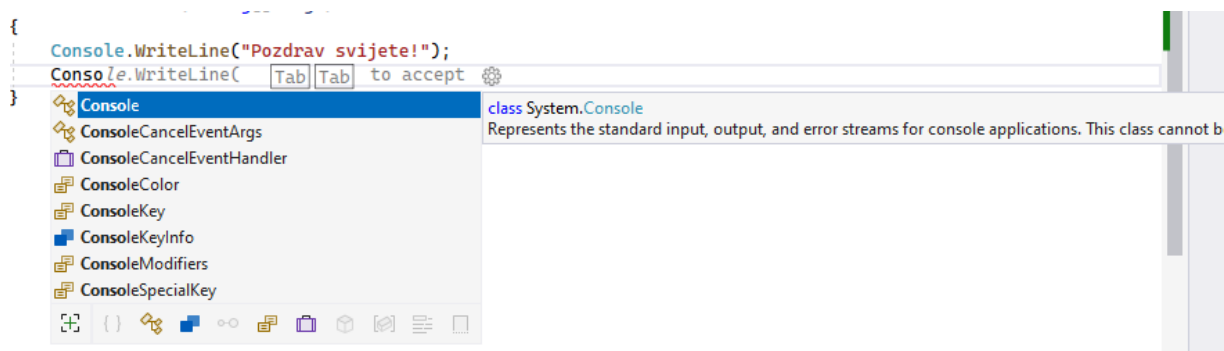
Slika 16. Izgled VS prozora i izgradnja aplikacije

Sad možemo napraviti malo analizu sučelja i što koji okvir prikazuje. Prvo što možemo vidjeti jest da u sredini imamo uređivač teksta, odnosno *code editor*.

Code editor uvelike nam pomaže prilikom pisanja koda na nekoliko načina:

- isticanjem sintakse, gdje vidimo različito obojane različite elemente koda. Ovo se može urediti, odnosno modificirati stiliziranjem
- prikazom broja linija, da se lakše snađemo gdje se nalazimo u kodu, kao i mogućnosti kolapsiranja i širenja blokova koda
- isticanjem referenci, odnosno da li se negdje referiramo na metode i kod unutar projekta
- **IntelliSense** sustavom, za koji slobodno mogu tvrditi da je jedna od najvažnijih i najboljih stvari prilikom pisanja koda, koji je u novoj verziji bolji i kompletniji nego ikada prije.

Na *IntelliSense* sustavu ovdje bih se najviše zadržao. Ovo je sustav koji nam tijekom pisanja nudi postojeće mogućnosti. Pogledajmo na slici 17.



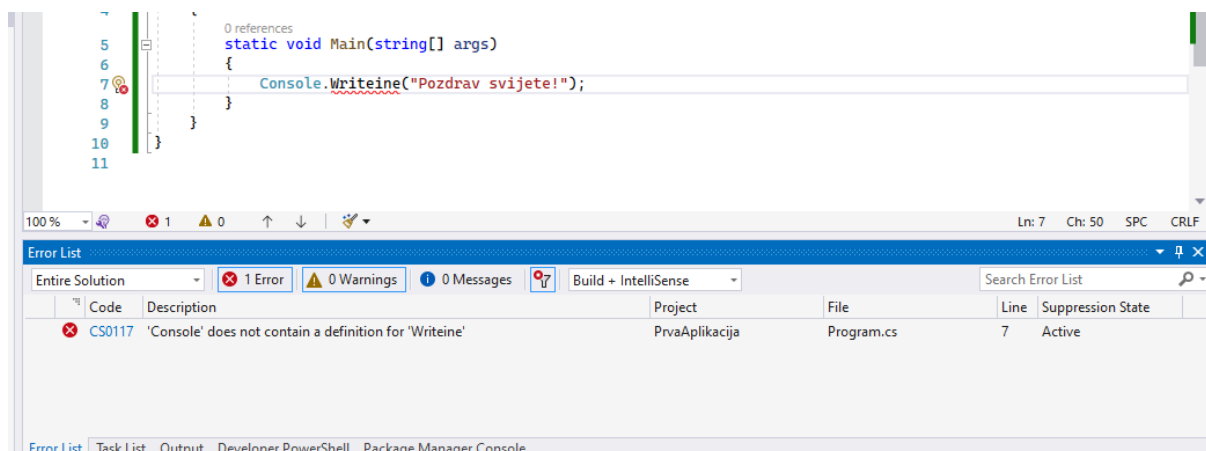
Slika 17. IntelliSense sustav u radu

IntelliSense je sustav koji će nam pomoći prilikom pisanja koda u smislu da imamo popis imena metoda, varijabli, klasa i ostaloga što odgovara onome što smo krenuli tipkati. Jednostavnim pritiskom na enter, popunit će ime onime što je odabrano i tijekom toga paziti da bude pravilno napisano. U novijim verzijama pokazat će nam što će se dogoditi pritiskom na „Tab” na tipkovnici, gdje popunjava kod na temelju pretpostavki prema onome što je prije toga napisano. Ukoliko nismo krenuli pisati, istovremenim držanjem tipke „Ctrl” i pritiskom na razmaknicu automatski će se otvoriti ovaj okvir i omogućiti nam odabir onoga što želimo napisati. Unutar IntelliSense okvira možemo i odabirati unaprijed definirane dijelove koda, takozvane *Code snippets*, koji će nam automatski napraviti okvir kompliciranijih naredbi koje možemo samo popuniti. No da ne dužimo o svim dobrobitima, kako budete radili, tako će vam se ovaj sustav sve više i više sviđati i nakon ovoga, iskreno rečeno, teško je prijeći na neko drugo razvojno okruženje, pogotovo kad vam ovo donese veliku brzinu u pisanju koda i razvoju aplikacija.

U ovom primjeru vidimo kako u dijaloškom prozoru „Solution Explorer” vidimo hijerarhiju programa. Prvo i najbitnije jest Solution s njegovim imenom. Unutar Solutiona imamo aplikaciju, a unutar aplikacije imamo poveznice i datoteke s kodom. I te datoteke s kodom mogu se dodatno proširiti pa na taj način iz Solution Explorera možemo vidjeti unutar koje nam se datoteke nalaze pojedini dijelovi koda, odnosno objekti poput klasa i njihovih dijelova, enumeracija itd.

Dijaloški okvir *Output*, kako vidimo, daje nam informacije vezane za izgradnju programa. Ovdje ćemo vidjeti rezultate izgradnje programa, gdje su izvršne datoteke spremljene i ostale bitne informacije.

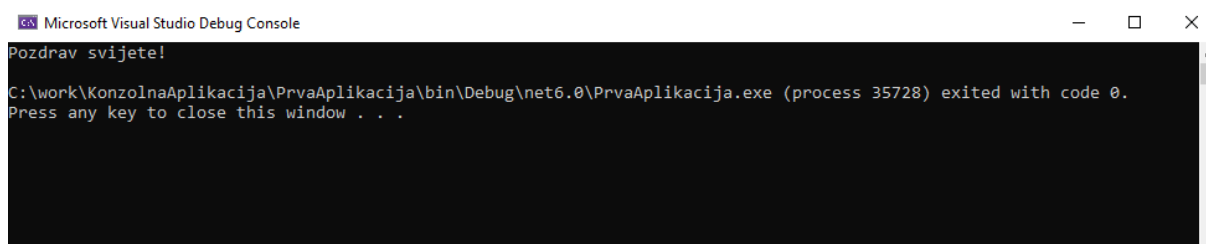
Da vidimo kako funkcioniše dijaloški prozor „Error List”, namjerno ću u sljedećem primjeru napraviti grešku u kodu.



Slika 18. Greška u kodu

U primjeru sam jednostavno izbrisao slovo L. Dijaloški okvir Error List prikazat će nam popis prepoznatih grešaka, s popisom kojem se tipu greške radi (ovdje jako dobro pomaže znanje engleskog jezika u točnom prepoznavanju grešaka), projekt u kojem je greška, datoteku gdje se greška nalazi i točnu liniju gdje je greška. Ovo je izuzetno korisno kad radite velike aplikacije s velikim brojem datoteka jer nam omogućuje točno praćenje i eliminaciju grešaka. Još ako pogledamo u kombinaciji s editorom koda, podcrtat će nam gdje je točno greška, a IntelliSense će nam dati klikom na ikonicu žarulje i potencijalno rješenje.

Pokretanje programa možemo obaviti na više načina iz VS-a, odnosno u više modova. To je u debug modu i bez debug moda. Pokretanje bez debug moda opet možemo napraviti na dva načina, a to je iz Visual Studia klikom na „Debug”, pa onda „Start Without Debugging” ili bez klikanja kraticom Ctrl + F5. Tom radnjom otvorit će se konzolni prozor koji će nam ispisati gornju poruku kao na slici 19.

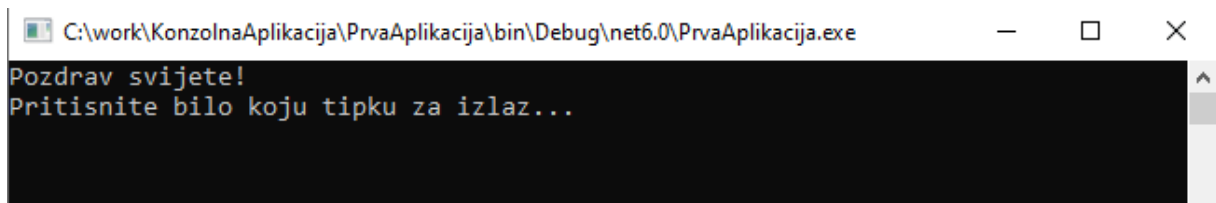


Slika 19. Prikaz rezultata izvršavanja programa

Drugi je način da odemo direktno mimo Visual Studio na lokaciju na disku koja je prikazana u gornjem prozoru i dvaput kliknemo na datoteku „PrvaAplikacija.exe”. Ipak, prilikom ove radnje primijetit ćete da samo „zatitra” prozor i nema više ništa. Razlog za to jednostavan je, ali treba prvo imati nekoliko stvari na umu: „Računalo i njegovi programi su samo glupi stroj koji radi točno što mu kažete.” I to će uvijek biti tako. U našem slučaju program kaže samo da se ispiše poruka i to je to. Čim ispiše poruku, prozor se odmah zatvori. Pokretanjem iz VS-a, VS automatski, kao što vidimo, stavlja pauzu na zatvaranje prozora dokle god ne pritisnemo bilo koju tipku za zatvaranje prozora. Idemo dodati par linija koda nakon `Console.WriteLine` da bismo mu rekli da čeka u programu da pritisnemo neku tipku. To ćemo učiniti na sljedeći način:

```
Console.WriteLine("Pozdrav svijete!");  
Console.WriteLine("Pritisnite bilo koju tipku za izlaz...");  
Console.ReadKey(true);
```

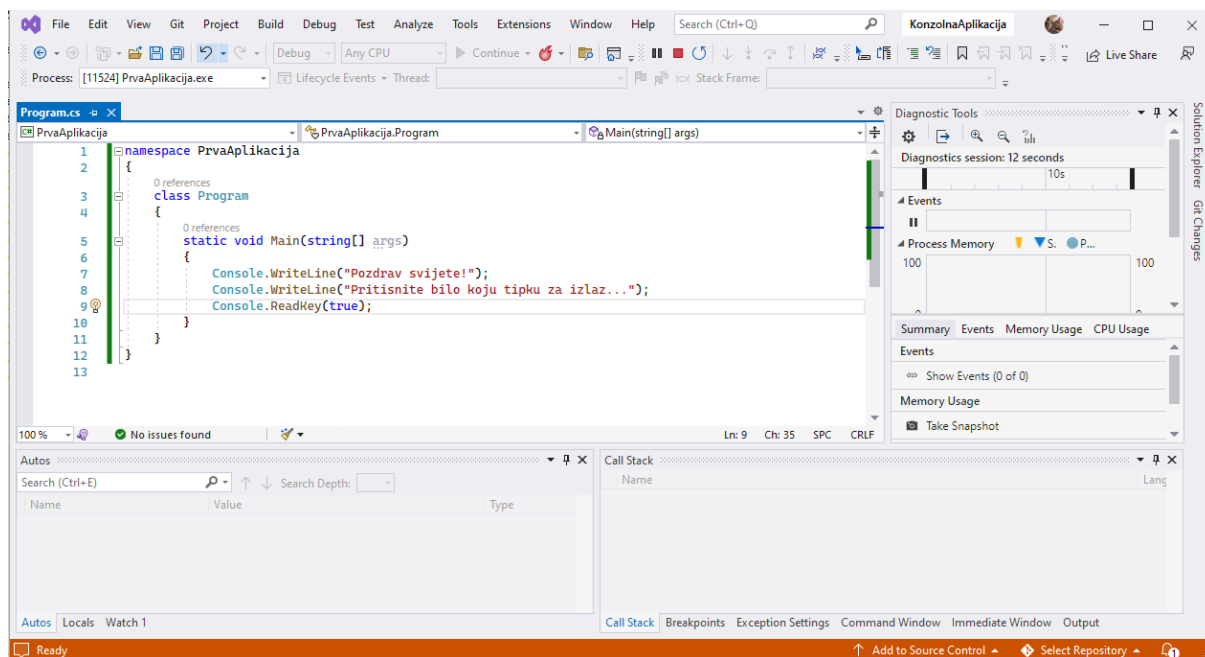
Naravno, prije pokretanja morate napraviti Build akciju da se ponovno izgradi izlazna datoteka. Nakon pokretanja datoteke dvoklikom dobivamo rezultat kao na slici 20.



Slika 20. Rezultat prilikom čekanja

Sad se program ne zatvara dokle god mi ne pritisnemo bilo koju tipku na tipkovnici.

Drugi način pokretanja aplikacije u VS-u jest da pokrenemo aplikaciju u takozvanom Debug modu. To radimo jednostavnim pritiskom na „Debug” pa „Start Debugging” ili pritiskom na tipkovnici tipke F5 ili klikom na tamnozeleni trokut u izborniku. Sada će VS ući u Debug mod kao na slici 21.



Slika 21. Debug mod Visual Studio alata

Iako malo drukčiji, ovo je opet Visual Studio. U ovom modu mi možemo namjestiti da staje na određenim naredbama, ali i pratiti vrijednosti u varijablama, koje se metode i dijelovi programa pozivaju, zauzeće resursa u računalu (memorija, procesor itd.)... Naravno, ovaj mod nam je jako dobar ukoliko dođe do grešaka u radu programa, jer će nam odmah pokazati gdje je došlo do greške, koja je greška, pauzirati izvođenje programa da vidimo koje su vrijednosti u varijablama itd.

3.2.5. Osnovna pravila pisanja koda

Već smo napisali nekoliko linija koda, no prije nego što krenemo dalje u učenje programa, idemo spomenuti nekoliko osnovnih pravila, odnosno smjernica u pisanju koda.

Osnovni program malo ćemo proširiti zbog sadržavanja svih dijelova na sljedeći način:

```
using System;
```

```
namespace PrvaAplikacija
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Pozdrav svijete!");  
        }  
    }  
}
```

Prvo i najvažnije, vidimo da svaka naredba u C# programskom jeziku završava točka zarez (;) znakom. To nam omogućuje da, ako želimo više naredbi napisati u istom redu, možemo, samo svaka mora završiti znakom točka zarez. Praznih linija možemo imati koliko god želimo, one nam koriste da bismo lakše čitali kod, odnosno da razdijelimo kod na neke logičke blokove.

Program se sastoji od zaglavlja u kojem ćemo najčešće upisati „using” naredbe, da kažemo koje ćemo dijelove, odnosno iz kojeg ćemo imenskog prostora koristiti klase. Imenski prostor dizajniran je tako da pruži način da držimo jedan set imena odvojen od drugog seta imena. Na ovaj način možemo u različitim imenskim prostorima jednako imenovati objekte, ali zahvaljujući imenskom prostoru neće međusobno ulaziti u konflikte.

Ključnu riječ „**namespace**” koristimo da bismo organizirali svoj kod, odnosno tako kažemo u kojem će imenskom prostoru naš kod biti.

Vitičastim zagradama { i } označavamo početak i kraj bloka naredbi. Kod zagrada općenito obavezno je pratiti koliko imamo otvorenih zagrada – toliko moramo imati i zatvorenih zagrada.

Ključnu riječ „**class**” kasnije ćemo detaljnije proći, no zasad ukratko, klasa sadrži podatke i metode koje donose funkcionalnost programa. U C# kodu svaka linija koda koju želimo pokrenuti mora biti unutar neke klase. U našem primjeru klasu smo imenovali „Program”. Nije nužno, ali često je nepisano pravilo da datoteku imenujemo po klasi koju pišemo u toj datoteci, jer se na takav način lakše snalazimo u kodu i hijerarhiji datoteka.

Svaki C# program mora negdje sadržavati **Main** metodu. Zasad nećemo ulaziti u detalje što piše prije i poslije Main, to ćemo proći kasnije kod funkcija i metoda, ali i kod objektno orijentiranog programiranja.

U našem programu Console.WriteLine(); je naredba, odnosno metoda unutar Console klase koju u našem slučaju koristimo za ispis teksta unutar zagrada na konzolnom prozoru.

Što se tiče C# jezika, moramo jako dobro paziti na velika i mala slova. C# naspram SQL-a razlikuje pisanje malih i velikih slova, pa nije isto napišemo li Broj ili broj ili BRoj ili bRoj itd.

Naredbe nećemo nabrajati ni prolaziti ovdje, no svakako organizacijski u pisanju dobro je znati kako lakše pisati kod radi lakšeg čitanja. U svom dosad napisanom kodu možete vidjeti kako smo radili uvlake, odnosno da VS automatski radi uvlake ovisno o hijerarhiji koda. Svaki put kad pišemo podblok koda, dobro je povećati broj uvlaka da vidimo koji dio koda pripada kojem bloku, odnosno kakva je hijerarhija. Taj dio pogotovo će biti vidljiv u kasnijim dijelovima kako budemo radili. Većinom kad steknete dovoljno iskustva u pisanju koda, ovo će vam biti podsvjesna dobra navika.

Govoreći o dobrim navikama, jedna od boljih jest ubacivanje komentara unutar koda. Komentari su tekst koji se build akcijom neće uzimati kao kod koji se može izvršiti, nego je to točno ono što ime kaže, tekst koji koristimo kao opis radi lakšeg snalaženja u kodu. Komentare možemo podijeliti na jednolinijske i višelinijске. Jednolinijske komentare označavamo s dvije kose crte (//) i one nam označavaju da je sve iza toga komentar. Često ih koristimo ako želimo neku liniju koda izbaciti iz build akcije, odnosno ako ne želimo da bude dio izlaza. Višelinijске komentare otvaramo znakom kosa crta i zvijezda (/*), a zatvaramo sa zvijezda i kosa crta (*). Kod višelinijških komentara sve između tih tagova smatra se komentarom. Visual Studio komentare najčešće označava zelenom bojom. Primjeri komentara:

//linijski komentar

```
Console.WriteLine("Pozdrav svijete!"); //linijski komentar na kraju linije
```

```
/* Komentar koji je u više linija
```

```
često koristimo kao opisni dio
```

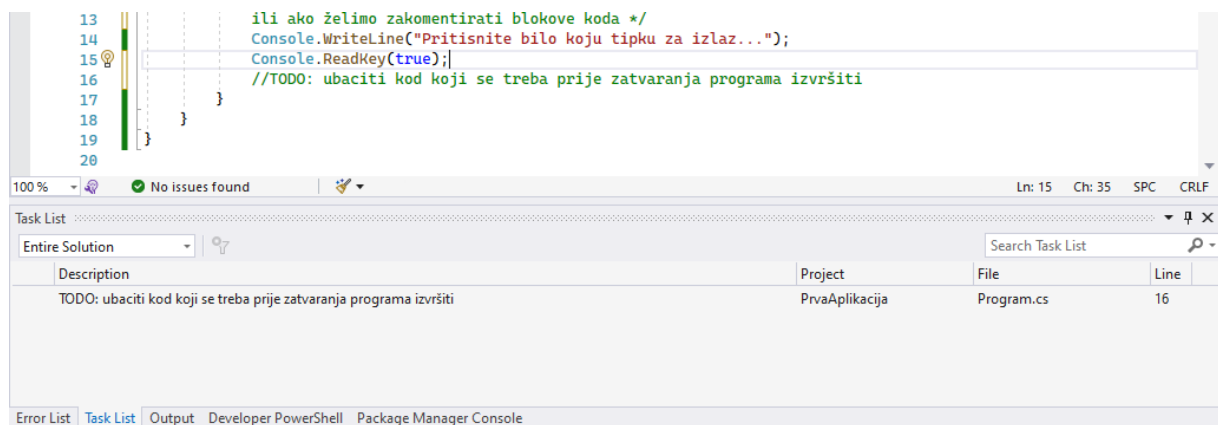
```
ili ako želimo komentirati blokove koda */
```

```
Console.WriteLine("Pritisnite bilo koju tipku za izlaz...");
```

```
//čekanje pritiska bilo koje tipke s tipkovnice
```

```
Console.ReadKey(true);
```

Komentari su izrazito korisni i u još nekoliko slučajeva. Osobno često koristim takozvane **TODO** komentare. Ostavljanjem **TODO** u komentaru, kasnije se putem Task List dijaloga možemo vratiti na dijelove programa u različitim datotekama ako treba nešto doraditi. Primjer takvog rada s komentarima na slici je 22.



Slika 22. Rad s TODO podsjetnicima

Ako pogledamo, automatski će nam biti prikazano gdje se taj komentar nalazi te tako možemo brzo navigirati do njega.

Sljedeći odličan način korištenja komentara jest u dokumentacijske svrhe. To dobivamo tako da prije klase ili metode napišemo tri kose crte (`///`) i VS će nam automatski generirati dio teksta u koji možemo ubaciti opise koji će biti korišteni tijekom korištenja tih objekata, odnosno tijekom generiranja dokumentacije.

```
namespace PrvaAplikacija
{
    /// <summary>
    ///
    /// </summary>
    class Program
    {
        /// <summary>
        ///
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            //linijski komentar
        }
    }
}
```

3.3. Varijable i tipovi podataka

Da bi program radio, moramo imati način za rad s podacima unutar računala, odnosno moramo znati kako se podaci u računalu prikazuju i kako možemo manipulirati i raditi operacije s podacima. C# kao programski jezik takozvani je strong type, odnosno jako tipiziran, što znači da jednom kada odredimo podatkovni tip varijabli, ta varijabla ne može mijenjati svoj podatkovni tip. Podatkovni tip određuje nam ponašanje zapisa, odnosno kako ćemo zapis u memoriji interpretirati u programu.

3.3.1. Tipovi podataka

Tipovi podataka određuju način na koji će se ponašati zapis podataka, kako će biti

registriran u memoriji, skup mogućih vrijednosti i koje operacije možemo raditi nad pojedinim podacima.

Podatkovne tipove inače možemo podijeliti na osnovne i kompleksne ili složene. No ova podjela u čisto objektno orijentiranim jezicima poput C#-a zna se malo zamagliti. Budući da u OOP-u sve gledamo kao objekte, ovdje su osnovni podatkovni tipovi implementirani preko sistemskih struktura, no ipak u osnovi su osnovni podatkovni tipovi.

Prvo ćemo spomenuti složene podatkovne tipove, za koje možemo reći da se sastoje od jednog ili više osnovnih podatkovnih tipova i/ili njihovih složenijih zapisa. Tipično složeni podatkovni tipovi jesu string (niz znakova), polja, klase itd.

Osnovne podatkovne tipove možemo podijeliti na cjelobrojne podatkovne tipove, decimalne podatkovne tipove, logički podatkovni tip i znakovni podatkovni tip.

Cjelobrojni podatkovni tipovi predstavljaju se u binarnom sustavu određenim brojem bitova. Ako gledamo broj bitova i označimo s N , brojevi koje možemo prikazati su od 0 do $2^N - 1$, no ukoliko želimo prikazati i negativne brojeve, jedan bit moramo koristiti kao predznak. Korištenjem jednog bita kao predznak, ne smanjujemo broj brojeva koji možemo prikazati, nego samo pomičemo raspon brojeva koje možemo prikazati i „presiječemo” maksimalnu vrijednost koju možemo zapisati za pola. Time dobivamo izraz da možemo zapisati brojeve od -2^{N-1} do $2^{N-1} - 1$. U sljedećoj tablici vidimo kako imenujemo i u kojem su rasponu vrijednosti:

Tip	Veličina	Raspon vrijednosti	.NET tip
sbyte	8 bita (1 bajt)	-128 do 127	System.SByte
byte	8 bita (1 bajt)	0 do 255	System.Byte
short	16 bita (2 bajta)	-32 768 do 32 767	System.Int16
ushort	16 bita (2 bajta)	0 do 65 535	System.UInt16
int	32 bita (4 bajta)	-2 147 483 648 do 2 147 483 647	System.Int32
uint	32 bita (4 bajta)	0 do 4 294 967 295	System.UInt32
long	64 bita (8 bajtova)	-9223372036854775808 do 223372036854775807	System.Int64
ulong	64 bita (8 bajtova)	0 do 18446744073709551615	System.UInt64

U varijable možemo zapisivati vrijednosti kao klasične dekadске brojeve, kao heksadekadске brojeve i binarnim zapisom. Tijekom pisanja brojeva samo moramo imati odgovarajuće prefikse, odnosno bez prefiksa za dekadski, 0x za heksadekadski i 0b za binarni zapis. Primjer zapisa cijelih brojeva na sva 3 načina:

```
var dekadski = 42;  
var heksadekadski = 0x2A;  
var binarni = 0b_0010_1010;
```

Za prikaz decimalnih brojeva koristimo tri podatkovna tipa: float, double i decimal. Oni se razlikuju po skupu dopuštenih vrijednosti i po broju značajnih decimala. Za decimalne brojeve često ćemo koristiti i pojam približna vrijednost jer ćemo se često dovesti u situaciju da ne možemo sve decimalne brojeve prikazati s konačnim brojem binarnih znamenki.

U sljedećoj tablici vidimo dopuštene vrijednosti, preciznost i veličinu zapisa:

Tip	Dopuštene vrijednosti	Preciznost	Veličina	.NET tip
float	$\pm 1.5 \times 10^{-45}$ do $\pm 3.4 \times 10^{38}$	~6-9 znamenki	4 bajta (32 bita)	System.Single
double	$\pm 5.0 \times 10^{-324}$ do $\pm 1.7 \times 10^{308}$	~15-17 znamenki	8 bajtova (64 bita)	System.Double
decimal	$\pm 1.0 \times 10^{-28}$ do $\pm 7.9228 \times 10^{28}$	28-29 znamenki	16 bajtova (128 bita)	System.Decimal

Za zapis konstanti, odnosno pri dodjeljivanju vrijednosti ovdje imamo više različitih načina nego kod cijelih brojeva. Tijekom pisanja brojeva možemo pomoću sufikasa odrediti koji je od navedenih podatkovnih tipova vrijednost zapisana. Za float koristimo f ili F sufiks, za double koristimo d ili D sufiks, a za decimal koristimo m ili M sufiks kao u sljedećim primjerima:

```
double d = 3D;  
d = 4d;  
d = 3.934_001;
```

```
float f = 3_000.5F;  
f = 5.4f;
```

```
decimal dec = 3_000.5m;  
dec = 400.75M;
```

Koristi se `_` kao separator znamenki, a može se koristiti i u cijelim brojevima umjesto razmaka, točke ili zareza. Na ovakav način možemo razdvajati znamenke u kodu. Ne smijemo napisati 3,000.00, pa umjesto zareza pišemo 3_000.00 radi separacije znamenki, no svakako možemo napisati i 3000.00 bez upotrebe separatora.

Decimalne brojeve možemo zapisivati i u znanstvenom obliku korištenjem eksponenta kao na sljedeći način:

```
double d = 0.42e2;  
Console.WriteLine(d); // izlaz: 42
```

```
float f = 134.45E-2f;  
Console.WriteLine(f); // izlaz: 1.3445
```

```
decimal m = 1.5E6m; // slovo E može biti veliko i malo slovo e  
Console.WriteLine(m); // izlaz: 1500000
```

Još su nam ostala dva osnovna podatkovna tipa:

Tip	Veličina	Raspon vrijednosti	.NET tip
bool	8 bita (1 bajt)	true, false	System.Bool
char	16 bita (2 bajta)	0 do 65535	System.Char

Bool ima samo dvije moguće vrijednosti, istina i laž, kao što i je u logičkoj algebri. Logičke vrijednosti često ćemo koristiti prilikom izračuna uvjeta kasnije kada budemo radili kontrolu toka programa.

Podatkovni tip **char** koristimo za spremanje i prikaz znakova, točnije UTF-16 znakova. U pozadini, char je implementiran kao 16 bitni cijeli broj (*ushort*) za prikaz vrijednosti. Time char možemo gledati kao cijeli broj, ali i kao znak. Vrijednost možemo postaviti na više načina, kao literal znakovni, kao unikod izlaznu sekvencu, kao heksadecimalnu sekvencu ili kao brojčanu vrijednost. Char konstante pišemo unutar jednostrukih navodnika. Sljedeći primjer pokazuje sve navedene načine:

```
var znakovi = new[]
{
    'j',
    '\u006A',
    '\x006A',
    106,
};
Console.WriteLine(string.Join(" ", znakovi)); // izlaz: j j j j
```

Vrijednost spremljena u char podatkovni tip predstavlja znak kao u ASCII tablici. U računalu sve znakove moramo spremiti u binarnom obliku, čak i one koji nemaju grafičku interpretaciju poput novog reda, tabulatora, strelice itd. Takve znakove zapisujemo pomoću određenih specijalnih oznaka zvanim izlazne sekvence (*escape sequence*). Svaki takav znak započinje obrnutom kosom crtom (*backslash*). Nekoliko često korištenih sekvenci:

Escape sekvenca	Znak
\'	Jednostruki navodnik
\“	Dvostruki navodnik
\\	Predstavlja jedan backslash
\n	Novi red
\t	Tab
\b	<i>backspace</i>

Tijekom korištenja podatkovnih tipova jednako je koristi li se ime tipa ili odgovarajuća struktura kako je navedena u .NET-u (.NET tip).

Posebno mjesto među podatkovnim tipovima ima i string tip, no on je niz znakova implementiran kao polje znakova (char array). No o njemu ćemo definitivno kasnije jer je jedan od najčešće korištenih podatkovnih tipova.

3.3.2. Varijable

Varijabla je identifikator u memoriji gdje je pohranjena vrijednost. Varijabla ima svoje ime, vrijednost i memorijsku lokaciju. Varijable koristimo kao privremene spremnike podataka unutar našeg programa, a koji nam služe za računanje. Da bismo koristili varijablu, potrebno je napraviti *deklaraciju varijable*. Deklaracija varijable objava je u programu koje ćemo ime koristiti za pristup podacima i kojeg će biti podatkovnog tipa. Generalni oblik sljedeći je:

{podatkovni_tip} {ime_varijable};

Za primjer, ako želimo koristiti varijablu za računanje cijelim brojevima, možemo to napraviti na sljedeći način:

int broj;

Ovdje smo objavili da ćemo koristiti varijablu „broj” koja je tipa int, odnosno cijeli broj, te se u memoriji rezervirao prostor od 4 bajta za čuvanje vrijednosti koju ćemo spremiti u tu varijablu.

O podatkovnim tipovima ćemo uskoro, no sada se idemo koncentrirati na varijable, odnosno imenovanje varijabli. Često se kaže da je pravilno imenovanje varijabli ključno kao dio dokumentacije, odnosno za lakoću čitanja koda. Preporuka za imenovanje varijabli jest da bi one trebale imati smisla, odnosno da trebaju imati smisla unutar programa koji radimo. U gornjem primjeru nećemo koristiti ime varijable „tekst” ako želimo u nju spremiti cijeli broj. U istom redu možemo napraviti višestruku deklaraciju, odnosno odmah najaviti više varijabli koje će biti istog podatkovnog tipa, kao u sljedećem primjeru:

int a,b,c;

Za imenovanje varijabli nemamo neka stroga pravila, ali imamo stroga pravila o tome što ne smijemo raditi prilikom imenovanja varijabli.

Prilikom imenovanja varijabli NE smijemo:

- koristiti razmake i druge specijalne znakove (+ - . ; * zagrade, točka, zarezi, znakove koji se mogu smatrati operatorima itd.)

- početi ime varijable znamenkom
- ime varijable biti ključna riječ u programu
- koristiti nestandardne znakove (poput „naših” znakova č, ć, š, ž...).

U dosta slučajeva imamo određena nepisana pravila što napraviti u slučajevima kad trebamo nešto što bi se kosilo s gore navedenim. Npr. umjesto razmaka često stavljamo podcrtu () ili ako trebamo početi brojem, stavimo i podcrtu ispred broja. Nekoliko primjera:

```
int ime varijable;    //nije dozvoljeno, razmak
int 3musketira;      //nije dozvoljeno, znamenka na početku
int _3musketira;     //dozvoljeno
float pi3.14;        //nije dozvoljeno, točka u imenu
float +broj;         //nije dozvoljeno, operator u imenu
char znak+slovo;     //nije dozvoljeno, operator u imenu
double znakovi(;     //nije dozvoljeno, zagrada
int double;          //nije dozvoljeno, ključna riječ
bool kočijaš;        //nije dozvoljeno, č i ć
```

Deklaracijom varijable ne dodjeljuje se vrijednost, nego ju je potrebno **inicijalizirati**. *Inicijaliziranje* varijable možemo gledati kao dodjeljivanje početne vrijednosti. To najčešće radimo operatorom pridruživanja (=) na sljedeći način:

```
int broj;
broj = 7; //varijabla broj poprima vrijednost 7, a ne broj jednako 7
```

Također, inicijaliziranje možemo napraviti i prilikom deklaracije u istom redu, na sljedeći način:

```
int broj = 7;
```

Tip vrijednosti koji pridružujemo mora odgovarati podatkovnom tipu varijable kojoj pridružujemo vrijednost. Zato inicijaliziranje varijable možemo napraviti i pomoću druge varijable ili izraza kao u sljedećem primjeru:

```
int a, b;
b = 3;
a = b + 2;
```

Kad radimo s objektima, oni se ne inicijaliziraju, nego ih, budući da su oni referentni tipovi, moramo instancirati. Instanciranje radimo s operatorom `new` na sljedeći način:

```
imeKlase ime_varijable=new imeKlase(listaParametara);
```

Ovdje se u memoriji odvaja prostor za pohranu objekta klase `imeKlase`, a operator `new` vraća referencu ili adresu prostora koji se onda može dodijeliti referentnoj varijabli. Prilikom korištenja operatora `new` na klasi poziva se konstruktor klase, no o ovome ćemo više govoriti u poglavlju o objektno orijentiranom programiranju.

C# je strog kad se radi o varijablama i ukoliko varijabla nije deklarirana i inicijalizirana, neće dopustiti njeno korištenje.

S varijablama obavezno je još voditi računa o doseg varijabli, odnosno hoće li nam određene varijable biti vidljive u nekim dijelovima koda. U pravilu varijablu koju definiramo u jednom bloku koda, ona je vidljiva u tom bloku i u svim podblokovima tog bloka. Recimo da imamo sljedeći primjer:

```
int a = 7;    // varijabla a
{
    //novi blok unutar prethodnog
int b = 6;    //lokalna varijabla b
    b = a;
}
b = a;        //neispravno, varijabla b nije deklarirana u bloku
```

Iako ukoliko napravimo grešku prilikom dosega varijabli, Visual Studio uvijek upozori tijekom kodiranja, a u najgorem slučaju ćemo tijekom *builda* dobiti grešku koju svakako treba ispraviti.

Ima slučajeva gdje određeni podatkovni tipovi nemaju mogućnosti navesti null vrijednosti. U takvim slučajevima koristimo posebno operator `?` uz podatkovni tip da bismo naznačili da varijabli može biti dodijeljena null vrijednost.

```
int a = 0;
int? b = null;
```

Osnovni podatkovni tipovi ne mogu imati null vrijednost, pa na ovakav način moramo naznačiti da mogu biti null i da ne moraju biti inicijalizirani. Ukoliko radimo s referentnim podatkovnim tipovima, oni su null automatski do instanciranja.

U C# jeziku možemo koristiti i implicitni tip podataka s ključnom riječi `var`. Korištenjem ključne riječi `var` umjesto podatkovnog tipa prilikom deklaracije znači samo da će kompajler odrediti koji je tip podataka varijabla utemeljen na prvoj inicijaliziranoj vrijednosti. U sljedećem primjeru obje su deklaracije ekvivalentne:

```
int i = 30;  
var i = 30;
```

Svakako treba napomenuti da jednom kada kompajler odredi podatkovni tip varijabli, taj podatkovni tip ne može se mijenjati! Podsjetimo se da je to jedno od osnovnih pravila svakog jako tipiziranog programskog jezika. Korištenje ključne riječi `var` svakako će nam skratiti deklaracije tijekom pisanja u slučajevima kada za podatkovni tip imamo duga imena. Druga tipična upotreba je da ne moramo koristiti isto ime kad dođemo do objektno orijentiranog programiranja, gdje zahvaljujući `var` ključnoj riječi ne moramo ponavljati ime u deklaraciji i instanciranju objekata.

Svi podatkovni tipovi deklarirani preko `var` ključne riječi su nulabilni. Nulabilni podatkovni tipovi znači da varijable za vrijednost mogu imati null.

Varijablama prilikom deklaracije možemo dodati ispred podatkovnog tipa još ključnu riječ „**const**”. Ukoliko dodamo `const`, označavamo da će varijabla biti konstantna, odnosno nakon prvog dodjeljivanja vrijednosti bilo koji sljedeći pokušaj mijenjanja vrijednosti varijable uzrokuje grešku.

```
const int pi = 3.14;  
pi = 4; //greška!
```


3.3.3. Konverzije podataka

U programiranju je čest slučaj da prilikom obrade podataka jedan podatkovni tip moramo pretvoriti u drugi, odnosno podatke iz jedne varijable prebaciti u drugu, iako su različitih podatkovnih tipova. U C# programskom jeziku konverzija se može raditi automatski (*implicitno*) ili pod kontrolom programera (*eksplicitno*).

Implicitna konverzija automatska je, gdje se sve automatski odvija bez naše intervencije. Često će nam to biti slučaj, pogotovo kod brojeva kad računamo i spremamo u neku varijablu. Primjer:

```
int broj = 2147483647;  
long velikibroj = broj;
```

Implicitna konverzija nije uvijek moguća, iako nam to izgleda logično s ljudskog stajališta. Ukoliko pokušamo postaviti implicitnu konverziju gdje to nije moguće, prilikom kompajliranja dobit ćemo grešku „cannot implicitly convert type 'prvi' to 'drugi'”. Kompajler može izbaciti tu grešku čak i ako može napraviti konverziju, ali prilikom konverzije postoji opasnost od gubitka podataka. Na primjer, ako želimo pretvoriti broj iz decimalnog u cijeli, gubimo sve iza decimalne točke, a ako je decimalni broj manji od 1, gubimo sve podatke. U tom slučaju moramo raditi eksplicitnu konverziju koju ćemo kontrolirati.

Kontrolirana konverzija zove se eksplicitna konverzija ili *cast*. U eksplicitnoj konverziji mi informiramo kompajler da ćemo napraviti konverziju i da smo svjesni da postoji mogućnost gubitka podataka ili da konverzija neće uspjeti tijekom izvođenja programa. Takvu konverziju radimo tako da u obliku zagradama ispred varijable napišemo u koji podatkovni tip želimo pretvoriti trenutni podatak. Primjer pretvorbe iz decimalnog broja u cijeli eksplicitno:

```
double pi = 3.14;  
int a = (int)pi;
```

Eksplicitnu konverziju ćemo često koristiti kod objektno orijentiranog programiranja tijekom konverzije iz baznog tipa u derivirani tip, no više o tome kad dođemo do toga.

Samo da napomenemo da tijekom izvođenja, s obzirom da se radi o referentnim tipovima, konverzija varijable ne mijenja tip objekta kako je spremljen, nego mijenja vrijednost reference koja se koristi za taj objekt. No više o tome kad dođemo do polimorfizma.

Konverzije ćemo često raditi u našim primjerima tijekom čitanja podataka iz konzole. Za čitanje podataka koristit ćemo `Console.ReadLine()`, koji će čitati sve što je uneseno nakon zadnjeg zapisa na konzolu, a unos će prestati pritiskom na tipku enter. Primjer za čitanje linije teksta:

```
string linija=Console.ReadLine();
```

Povratna vrijednost ove metode uvijek je niz znakova. Zato taj niz moramo pretvoriti u odgovarajući niz podataka. Konverziju možemo napraviti metodama unutar struktura osnovnih podatkovnih tipova ili nekim drugim klasama, odnosno metodama. Primjer korištenja `Parse` metode za pretvorbu prethodnog unosa u broj:

```
int a = int.Parse(linija);
```

No budite oprezni, nije moguće svaki niz parsirati, pa ukoliko pokušate unijeti nešto što nije broj, program će se srušiti s prikazom greške koja je nastala. U ovom slučaju greške možemo izbjeći korištenjem `TryParse` metode (o tome kad budemo radili grananje) ili hvatanjem greške kao iznimke (no o tome kod iznimki).

3.4. Operatori

Operator definira kako će se jedna varijabla ili podatak ponašati s obzirom na drugi. Operatore generalno možemo podijeliti na nekoliko načina, ovisno kako djeluju. Najosnovnija je podjela s obzirom na parnost: na binarne i unarne operatore. Postoje još i terarni operatori, ali izrazito su rijetki i često se svode na dva binarna operatora. Unarni operatori jesu operatori koji djeluju samo s jednom varijablom ili vrijednosti. Binarni operatori zahtijevaju dvije varijable ili vrijednosti da bi dali rješenje.

Operatore možemo dijeliti još s obzirom na svoju funkciju, pa ćemo sad napraviti pre-

gled operatora s obzirom na svoju funkciju djelovanja.

Aritmetički	Relacijski	Logički	Ostali
*	<	&&	=
/	<=		{{podatkovni tip}}
%	>	!	Bitovni operatori
+	>=		
-	!= ili <>		
++	==		
--			

Operator = jedan je od najbitnijih jer služi za dodjeljivanje neke vrijednosti varijabli.

Varijabla koju koristimo ne može mijenjati vrijednost bez operatora =, ali i to ima iznimku (inkrement, dekrement...). Služenje `s =` jednostavno je, s lijeve strane uvijek ide varijabla u koju spremamo vrijednost s desne strane. Ako s desne strane imamo komplicirani izraz, prvo se taj izraz mora izračunati.

Pridruživanje je destruktivna operacija. Pod tim mislimo da jednom kad spremimo neku vrijednost u varijablu, mi ne možemo doći do prijašnje vrijednosti. Nema *undo* kad se spremaju vrijednosti varijabli. Za pamćenje prijašnjih vrijednosti morate koristiti drugu varijablu i u nju spremati vrijednost koju želite zapamtiti iz prve varijable.

S desne strane pridruživanja moguće je da bude varijabla u koju spremamo rezultat. Bitno je reći, prvo će se odraditi sve operacije s desne strane, a na kraju se radi pridruživanje konačne vrijednosti u varijablu s lijeve strane.

```
int a, b;  
a = 17;      //pridodjeljivanje konstantne vrijednosti  
a = b;       //pridodjeljivanje vrijednosti druge varijable  
a = a + 4;   //pridodjeljivanje rezultata s desne strane. Program uzima  
vrijednost varijable a, zbraja joj 4 i rezultat sprema u varijablu a  
a = b + b - 12; //pridodjeljivanje kompleksnijeg izračuna
```

Aritmetički operatori predstavljaju osnovne računske operacije. Mislim da nije potrebno ulaziti u detalje opisa što koji radi, već samo ukratko podsjetiti se i spomenuti nekoliko napomena.

Svakako napomena, ovdje vrijede skoro sva pravila matematike, a pogotovo jako dobro treba paziti na prioritete i zagrade. Ukoliko niste sigurni s prioritetima, definitivno koristite zagrade.

Zvjezdica (*) je množenje, kosa crta (/) je dijeljenje, a postotak (%) je modul operator, odnosno ostatak prilikom dijeljenja. Ovi operatori imaju najveći prioritet prilikom računanja (osnove matematike). Kod dijeljenja treba paziti. Ukoliko radimo s cjelobrojnim varijablama, rezultat je cijeli broj. Da bismo dobili točnije dijeljenje, odnosno decimalni broj, moramo napraviti pretvorbu djelitelja u decimalni podatkovni tip.

+ i - služe za zbrajanje i oduzimanje ako se koriste kao binarni operatori, no mogu biti i unarni operatori koji predstavljaju predznak.

Aritmetičke operatore možemo kombinirati s operatorom pridruživanja. Tako dobijemo kompleksne operatore, gdje aritmetičku operaciju izvršavamo s operandom s lijeve strane i rezultat spremimo u nju samu.

Primjeri:

Jednostavni operator	Kombinirani operator
a =a+b;	a+=b;
a=a-b;	a-=b;
a=a*b;	a*=b;
a=a/b;	a/=b;
a=a%b;	a%=b;

Vidimo da možemo malo skratiti pisanje, što je u slučaju drugog imena varijable dobro.

Operator ++ unarni je operator inkrementa, a operator -- unarni je operator dekrementa. Oni su specifični zato što rezultat direktno spremaju u varijablu nad kojom vrše povećanje ili smanjenje. Tako u sljedećem primjeru ove dvije linije rade isti zadatak:

```
a++;  
a = a + 1;
```

Kod inkrementa i dekrementa bitno je i s koje strane varijable stavljamo operator.

Lijevo (prefiks)	Desno (postfiks)
<pre>int a = 4; int c;</pre>	
<pre>c=++a;</pre>	<pre>c=a++;</pre>
Rezultat u c je 5, kao i u a	Rezultat u c je 4, no u a 5!

Razlika *postfiksa* i *prefiksa* je u rezultatu. Kod prefiksa inkrement se izvrši prije dodjeljivanja, tj. prilikom čitanja vrijednosti varijable vrši se njen inkrement. Kod postfiksa se prvo vrši čitanje vrijednosti varijable i s tom očitanom vrijednosti se računa. Tek nakon čitanja računa se i inkrementira vrijednost varijable a.

Kod aritmetičkih operatora još jako dobro pazite na operande s desne i lijeve strane te varijablu rješenja. Čak i ako imate sve operande i rješenje istog podatkovnog tipa, ne mora uvijek biti točna operacija. Primjer imamo kod cijelih brojeva, gdje zbrajanje cijelih brojeva uvijek za rezultat daje cijeli broj, no dijeljenje dvaju cijelih brojeva ne mora dati točan rezultat u skupu cijelih brojeva. U ovim slučajevima morat ćemo raditi pretvorbe operanada, ali i paziti u koji podatkovni tip spremamo rezultat.

Relacijski i logički operatori daju nam samo jedan podatkovni tip rezultata, a to je bool. Samo rješenje može biti istina ili laž (true/false).

Kod relacijskih operatora moramo paziti da se operandi s lijeve i desne strane mogu uspoređivati. Zato oni najčešće moraju biti istog podatkovnog tipa, no to nije uvijek moguće, pogotovo kad budemo radili s kompleksnim podatkovnim tipovima.

Operator	Izraz	Rezultat
manje	$4 < 5$	istina (true)
manje	$4 < 4$	laž (false)
veće	$4 > 5$	laž (false)
manje ili jednako	$4 \leq 5$	istina (true)
manje ili jednako	$4 \leq 4$	istina (true)
veće ili jednako	$4 \geq 5$	laž (false)
veće ili jednako	$4 \geq 4$	istina (true)
različito	$4 \neq 4$ ili $4 <> 4$	laž (false)
različito	$4 \neq 5$ ili $4 <> 5$	istina (true)
jednako	$4 == 4$	istina (true)
jednako	$4 == 5$	laž (false)

Relacijske operatore još zovemo i operatori usporedbe.

Logički operatori su logičke algebre. Rad logičkih operatora prolazili smo kad smo radili Booleovu algebru. Bitno je reći da s lijeve i desne strane logičkog operatora moramo imati *bool* tip podatka, kao što će biti i rezultat. Važnost ovih operatora bit će nam najvidljivija kad budemo radili grananja i petlje.

A	B	A && B (I)	A B (II)
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Od operatora koje još trebamo spomenuti tu je *cast* operator, no njega smo već prošli prilikom eksplicitne konverzije podatkovnih tipova.

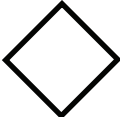
Operatore u koje nećemo ulaziti su bitovni operatori i *shift* operator. Samo ćemo napomenuti da su to operatori koji rade operacije direktno na registrima, odnosno binarnim zapisima podataka unutar varijable.

3.5. Grafički prikaz algoritma

Algoritam je konačan niz koraka koji nas dovode do rješenja ili spoznaje da rješenje ne postoji. Grafički prikaz algoritma zovemo još dijagram toka.

Dijagrama toka koristi se da bismo dizajnirali i dokumentirali jednostavne procese unutar programa. Naspram drugih tipova dijagrama, dijagram toka pomaže nam da vizualiziramo proces rada programa. Glavne prednosti dijagrama toka su to što će nam pomoći da prepoznamo očite greške i uska grla rada programa.

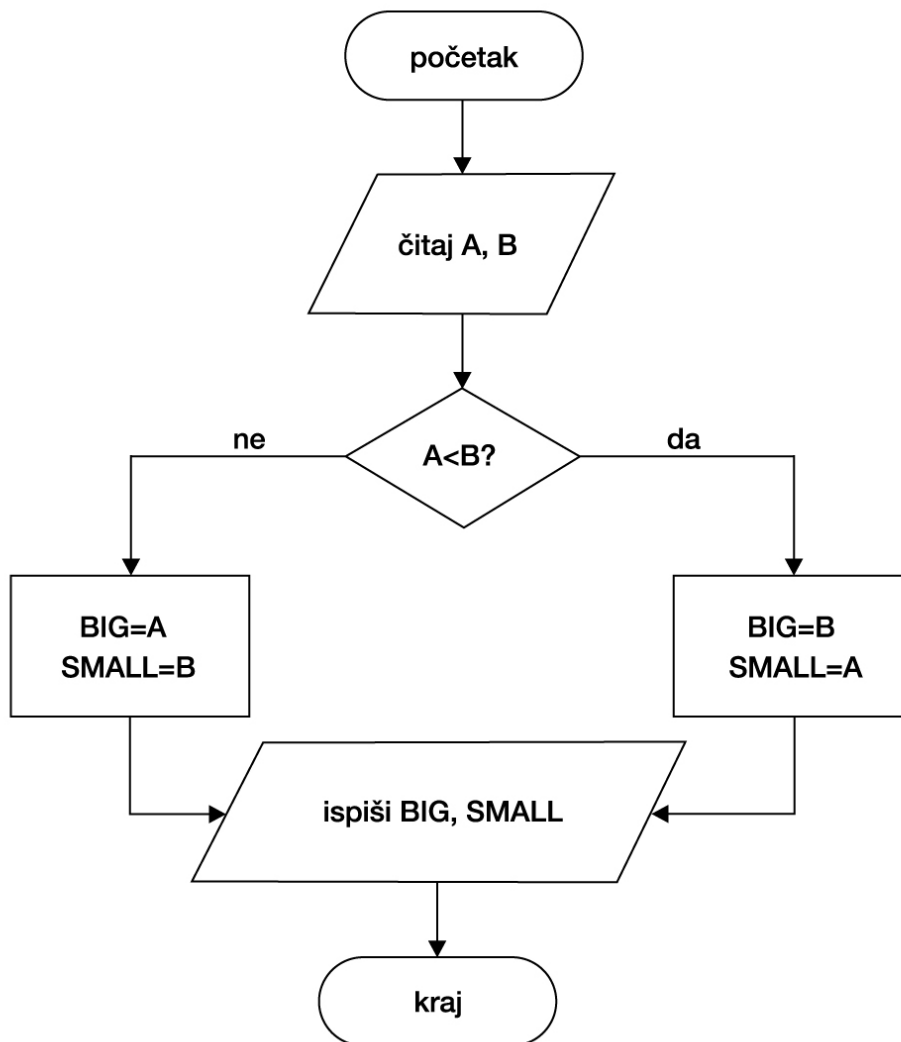
Tipično dijagrami toka koriste sljedeće simbole:

	elipsa označava početak i kraj programa ili funkcije ili odsječka koji prikazujemo
	pravokutnik označava naredbu ili blok naredbi
	romb označava točku odluke
	trapez označava ulaz/izlaz
	strelice označavaju tok programa

Ima još dodatnih simbola mimo osnovnih. Za, primjerice, označavanje baze podataka, pristupa dokumentima, inicijalizaciju itd. Dijagrami toka mogu se referencirati na druge, radi jednostavnosti.

Dijagram toka nikada neće imati implementaciju, nego samo generalni tok programa prikazan na što jednostavniji način.

Da bismo pokazali korištenje dijagrama toka, napraviti ćemo primjer gdje za dva učitana broja određujemo koji je manji, a koji veći i ispisujemo ih.



3.6. Kontrola toka programa grananjem

Grananje u programiranju proces je gdje se odlučuje koji će se dio koda izvršavati ovisno o uvjetu grananja. Drukčije rečeno, na temelju određenog uvjeta donosimo odluku hoćemo li neki dio koda izvršiti ili nećemo. U C# programskom jeziku imamo nekoliko načina kako možemo izvršiti grananje, odnosno nekoliko naredbi.

3.6.1. Naredba if

Naredba **if** najjednostavniji je oblik grananja. Ona ima sljedeću sintaksu:

```
if (uvjet) naredba;  
if (uvjet)  
{  
    //blok naredbi  
}
```

Naredbu **if** možemo gledati na jednostavan način: ako je uvjet istinit, izvrši naredbu ili blok naredbi.

Uvjet u zagradama uvijek mora biti evaluiran kao *bool* podatkovni tip, što znači da mora na kraju biti evaluiran kao istina ili laž (*true* ili *false*). Zato za uvjet često koristimo relacijske i logičke operatore.

Primjer programa gdje za uneseni broj ispiše je li paran ili neparan:

```
static void Main(string[] args)  
{  
    int broj;  
    Console.Write("Unesite broj : ");  
    broj=int.Parse(Console.ReadLine());  
    if (broj % 2 == 0) Console.Write("Broj je paran");  
    if (broj % 2 == 1) Console.Write("Broj je neparan");  
}
```

Kao što vidite u ovom primjeru, imamo složeni uvjet, odnosno program kad dođe do **if** naredbe, prvo mora izračunati uvjet korištenjem operatora, pa zatim na temelju rješenja radi odluku treba li ispisati ili ne niz znakova „Broj je paran”. Uvjet smo mogli izračunati i prije toga pa uvrstiti izračunatu varijablu u uvjetno grananje na sljedeći način:

```
bool uvjet = broj % 2 == 0;  
if (uvjet) Console.Write("Broj je paran");
```

Način na koji želite to raditi je na vama, no svakako iz gledišta optimizacije programa, preporučamo da nemate previše varijabli koje koristite samo jednom.

Grananje često koristimo dvostrano. Kako imamo blok koji će se izvršavati ukoliko je uvjet istinit, možemo navesti naredbu ili blok naredbi koji se trebaju izvršavati ukoliko uvjet nije istinit, odnosno uvjet je laž. Takav kod izgleda ovako:

```
if (uvjet)
{
    naredbe1; //naredbe za izvršavanje ukoliko je uvjet istinit
}
else
{
    naredbe2; //naredbe koje se izvršavaju ukoliko uvjet nije istinit
}
```

Naredba **else** opcionalna je, ona ne mora uvijek biti navedena. No svakako ne može biti navedena bez odgovarajuće if naredbe!

Sad možemo promijeniti primjer provjere je li broj paran ili neparan:

```
static void Main(string[] args)
{
    int broj;
    Console.Write("Unesite broj : ");
    broj=int.Parse(Console.ReadLine());
    if (broj % 2 == 0) Console.Write("Broj je paran");
    else Console.Write("Broj je neparan");
}
```

Zašto je bolje na ovakav način? Odgovor je – jednostavno zbog broja računanja. U ovom primjeru ne moramo dvaput raditi istu operaciju (broj%2 i usporedbu s brojem).

Budući da grananje podrazumijeva naredbu ili blok naredbi, možemo staviti grananje unutar grananja.

Stavljanje jednog niza naredbi ili strukture unutar druge zove se ugnježđivanje naredbi. To možemo prikazati na ovakav način:

```
if (uvjet)
{
    if (uvjet2)
    {
        naredbe;
    }
    else
    {
        naredbe;
    }
}
else
{
    naredbe;
    if (uvjet3) naredba;
}
```

Ukratko, mi ovisno o potrebi možemo, ako treba, bez ikakvog ograničenja ugnježđivati naredbe.

Često kad trebamo pobrojati uvjete određenog programskog koda, koristit ćemo takozvani **prošireni oblik if** naredbe. To izgleda ovako:

```
if (uvjet)
{
    naredbe;
}
else if (uvjet2)
{
    naredbe2;
}
else if (uvjet3)
{
    naredbe3;
}
...
```

```

else if (uvjetN)
{
    naredbeN;
}
else
{
    naredbeM;
}

```

Zašto je ovaj oblik dobar? Jer možemo redom ispitivati uvjet po uvjet. Prvo ispitujemo uvjet, koji ako je evaluiran kao true, odradit će prvi blok naredbi i dalje neće ni ići. No ako uvjet nije istinit, naravno ide na else blok u kojem se odmah nalazi if naredba s uvjet2. Nakon prve if naredbe, svaka sljedeća provjerava se samo ako je prijašnji uvjet bio false. Izvršavanje naredbi događa se kada prvi uvjet od nabrojanih bude true, a ako su ipak svi false, onda se izvršavaju naredbe samo kod zadnjeg else bloka.

Bitnost ovog proširenog oblika u brzini je izvođenja i broju ispitivanja uvjeta. Kad bismo to gledali kao niz klasičnih if naredbi s ispitivanjem uvjeta, onda bi program išao redom od jednog do drugog ispitivanja uvjeta neovisno o uspješnosti prijašnjeg. Kod proširenog oblika, čim dođe da je jedan od uvjeta istinit, nema više ispitivanja. Nadalje, dozvoljava nam da možemo raditi jednostavnije uvjete ispitivanja kad već znamo koje smo uvjete prije toga prošli.

Za usporedbu, napraviti ćemo primjere ispitivanje koliko znamenki ima uneseni cijeli broj. Koristit ćemo samo kod koji bi se nalazio unutar Main funkcije:

Normalan oblik	Prošireni oblik:
<pre> int broj; Console.WriteLine("Unesite cijeli broj : "); broj = int.Parse(Console.ReadLine()); if (broj < 10) Console.WriteLine("jednoznamenkast"); if (broj >= 10 && broj < 100) Console.WriteLine("dvoznamenkast"); if (broj >= 100 && broj < 1000) Console.WriteLine("troznamenkast"); if (broj >= 1000 && broj < 10000) Console.WriteLine("četveroznamenkast"); if (broj >= 10000 && broj < 100000) Console.WriteLine("peteroznamenkast"); if (broj >= 100000 && broj < 1000000) Console.WriteLine("sesteroznamenkast"); if (broj >= 1000000) Console.WriteLine("Vise od sest znamenki"); </pre>	<pre> int broj; Console.WriteLine("Unesite cijeli broj : "); broj = int.Parse(Console.ReadLine()); if (broj < 10) Console.WriteLine("jednoznamenkast"); else if (broj < 100) Console.WriteLine("dvoznamenkast"); else if (broj < 1000) Console.WriteLine("troznamenkast"); else if (broj < 10000) Console.WriteLine("četveroznamenkast"); else if (broj < 100000) Console.WriteLine("peteroznamenkast"); else if (broj < 1000000) Console.WriteLine("sesteroznamenkast"); else Console.WriteLine("Vise od sest znamenki "); </pre>

Kao što vidimo u ovom primjeru na oba načina, ako unesemo jednoznačenasti broj, u normalnom obliku program će ispitati svaku if naredbu, odnosno svaki uvjet mora biti izračunat. U proširenom obliku idemo malo bržom logikom. Provjeravamo je li jednoznačenasti (manji od 10), a ako nije, tek onda provjerava je li manji od 100. Ne moramo ubaciti prijašnji uvjet jer već znamo da prijašnji uvjet nije istinit, pa time nema smisla provjeravati je li njegova suprotnost istinita.

Prošireni oblik odličan je za optimizaciju, gdje možemo u određenim slučajevima staviti prvo uvjete koji imaju veću vjerojatnost ispunjavanja, pa je time manja vjerojatnost provjeravanja većeg broja uvjeta.

3.6.2. Naredba switch

Switch naredba služi za provjeravanje jednakosti zadane varijable ili izraza. Izgleda ovako:

```
switch ({varijabla ili izraz}) {  
    case vrijednost1:  
        naredbe1;  
        break;  
    case vrijednost2:  
        naredbe2;  
        break;  
    case vrijednost3:  
        naredbe3;  
        break;  
default:  
    naredbe;  
    break;  
}
```

Ključna riječ **case** označava da u slučaju da su varijabla ili izraz jednaki vrijednosti uz case, treba izvršavati te naredbe. Možemo reći da je switch naredba slična proširenom obliku if naredbe, uz nekoliko ključnih razlika. Uz case nikada ne smijemo imati varijablu, nego konstantnu vrijednost.

Kad dođe do toga da je jedan slučaj (case) istinit, nastavljaju se izvršavati sve naredbe do kraja bloka. Zato unutar case dijela, na kraju naredbe ako ne želimo da se nastavljaju izvršavati naredbe iz sljedećeg slučaja, moramo staviti ključnu riječ break. Inače, u većini programskih jezika break je opcionalan, no u C# obavezno moramo na kraju naredbi case navesti break. Ukoliko želimo da više različitih slučajeva ima iste naredbe, možemo case pisati jedan iza drugoga za one za koje želimo da izvršavaju istu naredbu ili naredbe. Primjer:

switch (ocjena)

```
{
    case 1:
        Console.WriteLine("Nije prolaz");
        break;
    case 2:
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Prolaz!");
        break;
    default:
        Console.WriteLine("Nije ocjenjeno");
        break;
}
```

Ključna riječ `break` i njen blok opcionalni su, nije ih obavezno navesti. Naredbe default bloka izvršit će se samo ukoliko nijedan od prijašnjih slučajeva nije istinit.

3.7. Kontrola toka programa petljama

Petlje su kontrolni mehanizmi koji nam omogućuju da neki kod ponovimo više puta. Kad govorimo o petljama u C#-u i njemu sličnim programskim jezicima, imamo 3 osnovne petlje. Te petlje su `for`, `while` i `do while`. Također, svaku petlju možemo prikazati pomoću druge petlje, pa će na kraju izbor kako koju petlju koristiti ovisiti o programeru. Svaka petlja ima nekoliko uobičajenih načina korištenja, a ostatak ćete sami zaključiti.

Još samo napomenuti da postoji i `foreach` petlja, koja nam služi za prolazak kroz iterabilnu strukturu, ali tu petlju proći ćemo u sljedećem poglavlju.

3.7.1. Programska petlja for

For petlju najčešće koristimo kad znamo broj ponavljanja petlje. Ne znam kako ovdje naglasiti najčešće, jer ne mora biti tako. Koristimo je često kad moramo prolaziti kroz polja i određene kompleksne podatkovne strukture.

Sintaksa for petlje sljedeća je:

```
for ({ inicijalizacija };{ uvjet};{ korak})  
{  
  naredbe;  
}
```

Moramo odmah napomenuti, svako polje opcionalno je, ne mora biti navedeno, no moramo svakako staviti točke zarez da se zna koje polje gdje ide.

Prvo polje *inicijalizacija* takozvana je inicijalizacijska naredba. Najčešće se koristi za dvije stvari. Prva je da postavimo početnu vrijednost varijable koju ćemo najčešće koristiti za prolaz kroz petlju. Drugo za što možemo koristiti ovo polje jest da definiramo lokalnu varijablu koja će postojati samo na razini petlje. Naredba koju ovdje definiramo pokreće se samo jednom kad program dođe do petlje.

Polje *uvjeta* radi točno to što mu ime govori. Ovdje definiramo uvjet ponovnog ulaska u petlju, tj. uvjet koji ovdje postavimo govori nam da će se petlja ponavljati dokle god je uvjet istinit. Podatkovni tip rezultata uvjeta mora biti evaluiran kao boolean (logički) podatkovni tip. Ispitivanje istinitosti uvjeta vrši se svaki put prilikom novog ulaska u petlju. Ukoliko ne napišemo uvjet u ovo polje, prevoditelj automatski smatra da je istina (true) navedena u ovom polju. Zato moramo biti jako oprezni, jer možemo dobiti beskonačnu petlju.

Polje *korak* sadrži naredbu koja se uvijek izvrši nakon što su naredbe unutar tijela petlje izvršene.

Primjer programa za zbrajanje prvih 100 prirodnih brojeva:

```
static void Main(string[] args)
{
    int sum = 0;
    for (int i = 1; i <= 100; i++)
    {
        sum += i;
    }

    Console.WriteLine("Suma prvih 100 prirodnih brojeva je {0} ",sum);
}
```

Tijekom rada u for petlji moramo biti pažljivi oko naredbi koje obavljamo unutar same petlje. Moguće su manipulacije koracima i/ili uvjetima tako da dobijemo beskonačnu petlju ili da preskačemo korake koje smo zadali u tijelu same petlje.

```
for (int i = 0; i < 100; i++)
{
    Console.WriteLine(i);
    i--;
}
```

U ovom primjeru imamo beskonačnu petlju. Iako je tijelo petlje zadano da se petlja ponavlja 100 puta, budući da imamo naredbu `i--`, sama varijabla se svakim prolaskom kroz petlju dekrementira točno prije nego se inkrementira. Zato će ovaj programski odsječak beskonačno ispisivati 0 dokle god ne prekinemo program.

U polje korak možemo upisivati izraz kompleksnijeg tipa. Ne mora biti jednostavan inkrement ili dekrement. U kompliciranijim podatkovnim strukturama korak često koristimo da bismo iterator postavili da pokazuje na sljedeći element strukture.

Primjer ispisa svih brojeva djeljivih s 4 od 100 do 0:

```
static void Main(string[] args)
{
    for (int i = 100; i > 0; i = i - 4) Console.WriteLine(i);
}
```

Ispis ovog programa bit će jedan ispod drugog redom brojevi 100, 96, 92, ... ,12,8 i 4.

3.7.2. Programska petlja while

Petlja **while** nam je druga vrsta petlje koju koristimo. Logika korištenja jednostavna je, naredbe unutar while petlje ponavljat će se dokle god je uvjet istinit.

while (uvjet)

```
{  
    naredbe;  
}
```

Petlja izgleda slično kao grananje if, samo nema nikakve dodatne opcionalne dijelove.

Uvjet isto kao kod grananja, mora biti evaluiran kao istina ili laž.

Primjer programa za brojanje od 20 do 30:

```
static void Main(string[] args)  
{  
    int broj = 20;  
    while (broj <= 30)  
    {  
        Console.WriteLine(broj);  
        broj++;  
    }  
}
```

Kao što vidimo u primjeru, naspram for petlje, mi osiguravamo ranije postojanost varijable koju koristimo u uvjetu petlje.

Zbog provjere uvijek istog uvjeta, trebali bismo osigurati da se varijabla koju koristimo u uvjetu mijenja tijekom izvođenja petlje, inače dobivamo beskonačnu petlju.

While petlja najviše se koristi kad nismo sigurni koliko ponavljanja petlje bismo mogli imati. To ćemo pokazati u sljedećem primjeru, gdje ćemo zbrojiti znamenke unesenog cijelog broja:

```

static void Main(string[] args)
{
    int broj;
    Console.Write("Unesite cijeli broj : ");
    broj=int.Parse(Console.ReadLine());
    int sum = 0;
    while (broj > 0)
    {
        //petlja će se ponavljati dokle god imamo znamenki
        sum += broj % 10; //suma je zbroj prijašnjih i zadnje znamenke
        broj /= 10;      //odbacujemo zadnju iskorištenu znamenku
    }
    Console.WriteLine("Suma njegovih znamenki je {0}", sum);
}

```

Kao što vidimo, mi ne možemo biti sigurni koliko će znamenkasti broj korisnik unijeti u program. U programu koristimo varijablu sum za spremanje sume znamenki, te je prije prve upotrebe postavljamo na neutralan element za zbrajanje. Svakim prolaskom kroz petlju, uzimamo zadnju znamenku broja i zbrajamo s prijašnjim zbrojem znamenki u varijabli sum. Nakon što iskoristimo zadnju znamenku za sumu, moramo odbaciti tu znamenku iz broja, budući da smo je ionako iskoristili.

3.7.3. Programska petlja do ... while

Petlju **do while** možemo jednostavno shvatiti na sljedeći način: Radi naredbe dokle god je uvjet istinit.

```

do
{
    naredbe;
} while (uvjet);

```

Kao što vidite iz koda, u ovom slučaju se naredbe prvo izvrše, a zatim se ispituje uvjet i provjerava treba li opet izvršiti blok naredbi. To nam je glavna i možemo reći jedina razlika naspram while petlje – mjesto gdje ispituje uvjet.

Dok se while petlja zbog uvjeta ne mora uopće izvršiti nijednom naredbe, do while petlja minimalno će jednom izvršiti naredbe koje su u tijelu petlje. To je primarna i jedina razlika naspram while petlje.

Do while petlja često nam je korisna prilikom čitanja podataka. Primjer programa gdje program čita podatke i zbraja dokle god se ne unese 0:

```
static void Main(string[] args)
{
    int broj, zbroj = 0;
    do
    { //blok naredbi se izvršava
        Console.Write("Unesite broj : ");
        broj=int.Parse(Console.ReadLine());
        zbroj += broj;
    } while (broj != 0); //provjeravamo je li unesena 0, ako jest ponavljamo
                           blok naredbi
    Console.WriteLine("Zbroj unesenih brojeva je {0}", zbroj);
}
```

3.7.4. Ugnježđivanje petlji

Slično kao što smo kod grananja imali ugnježđivanje, tako imamo i kod petlji. Budući da svaka petlja može izvršavati naredbu ili blok naredbi, unutar tih blokova mogu biti druge petlje i grananja. Često ćemo imati problema kod kojih ćemo morati koristiti petlju unutar petlje unutar petlje i tako dalje ovisno o složenosti problema.

Primjer ugnježđenih petlji program je za ispis prostih brojeva do 100:

```
static void Main(string[] args)
{
    Console.WriteLine("Prosti brojevi do 100:");
    for (int i = 1; i < 100; i++) //prvu petlju koristimo da cikliramo 100 brojeva
    {
        bool prost = true;      //krećemo od pretpostavke da je broj prost
        for (int j = 2; j < i; j++) //provjeravamo je li djeljiv s bilo kojim brojem
        {
            if (i % j == 0)
            {
                prost = false;
                break;
            }
        }
        if (prost)
            Console.Write(i + " ");
    }
    Console.WriteLine();
}
```

```

        većim od 1, a manji od njega samog
    {
        if (i % j == 0) prost = false; //ako nema ostatak prilikom dijeljenja, znači
        da nije prost i postavljamo varijablu prost na false
    }
    if (prost) Console.WriteLine(i); //ispiši i ako je prost
}
}

```

3.7.5. Naredbe kontrole

Naredbe kontrole petlji služe da zaustavimo ili promijenimo normalni tok petlje. Od kontrolnih naredbi spomenut ćemo dvije, a to su break i continue.

Break smo ranije spomenuli kod switcha. Naredba break služi kako bismo potpuno prekinuli blok naredbi u kojem se nalazimo. Kad su petlje u pitanju, to znači da će se petlja prekinuti, a izvođenje programa nastavlja se prvom naredbom nakon petlje. Najčešća upotreba break naredbe jest prekidanje beskonačne petlje.

Primjer korištenja break naredbe možemo vidjeti u sljedećem programu, koji zbraja sve unesene brojeve dok se ne unese 0:

```

static void Main(string[] args)
{
    int broj, zbroj = 0;
    while (true)
    { //beskonačna petlja
        Console.Write("Unesite broj : ");
        broj=int.Parse(Console.ReadLine());
        if (broj == 0) break; //prekid petlje
        zbroj += broj;
    }
    Console.WriteLine("Zbroj unesenih brojeva je {0}", zbroj);
}

```

Znači vidljivo je da `break` prekida potpuno izvođenje petlje zaobilazeći ispitivanje uvjeta u `while` petlji.

Continue nam je drugi način kontrole toka. Radi slično kao `break`, s jedinom i izuzetno bitnom razlikom, a to je da preskače samo ostatak bloka naredbi.

Ako gledamo s obzirom na vrstu petlji, `continue` kod `while` i `do while` petlje preskače sve naredbe i skače na dio provjere uvjeta petlje, dok kod `for` petlje preskače naredbe i skače na dio koraka, odnosno dio gdje se izvršava naredba korak. Budući da korak nije dio bloka naredbi, izvršavanjem naredbe `continue` ne utječemo na izvršavanje naredbe ili naredbi koraka, koje se svakako izvršavaju.

Kad pogledamo u kodu, razlika između naredbi `break` i `continue` svodi se na jednu jedinu liniju razlike.

for	while	do while
<pre>for ({ inicijalizacija; { uvjet}; { korak}) { naredbe; } Nastavak;</pre>	<pre>while (uvjet) { naredbe; } Nastavak;</pre>	<pre>do { naredbe; } while (uvjet); Nastavak;</pre>

Ako pogledamo prototipove naredbi, `break` nam skače van petlje na naredbu „Nastavak”, dok `continue` skače poslije naredbi na zatvorenu vitičastu zagradu. Skokom na vitičastu zagradu, kod `while` i `do while` znači da je sljedeća naredba provjera uvjeta ide li opet izvršavati naredbe petlje, a kod `for` petlje to znači da se prvo izvršava naredba koraka, a zatim provjerava može li opet u petlju.

3.7.6. Naredba goto

Naredbe kontrole toka ne možemo završiti, a da ne kažemo i **goto** naredbu. To je naredba bezuvjetnog skoka na labelu ubačenu u kod. Izgleda na sljedeći način:

```
Naredbe;    //naredbe programa
LABELA:    //labela
Naredbe;    //naredbe programa
goto LABELA; //naredba skoka na labelu
Naredbe;    //naredbe programa
```

Kao što se vidi, naredba bezuvjetnog skoka ne izgleda baš elegantno, no jako je učinkovita. Labela može bilo kako izgledati, odnosno to je običan tekst koji na kraju ima dvotočku. Sama goto naredba sastoji se od **goto** ključne riječi i imena labele na koju treba skočiti.

Iako dosta programera govore da goto treba izbjegavati, osobno se ne bih složio. Iako smo napomenuli da nije baš elegantno za čitanje, ima dosta slučajeva gdje efikasno korištenje goto naredbe uvelike ubrzava program.

Jedan od tipičnih slučajeva korištenja jest izlaz iz višestruko ugniježđenih petlji:

```
while (uvjet1)
{
    ...
    while (uvjet2)
    {
        ...
        while (uvjet3)
        {
            ...
            goto IZLAZ;
        }
        ...
    }
    ...
}
IZLAZ:
Naredbe_van_petlji;
```

Tipičan primjer korištenja gdje ne moramo raditi višestruke provjere jesu li zadovoljeni uvjeti za break nakon svake ugniježdene petlje.

3.8. Polja

Polja koristimo za pohranu više podataka istog tipa, međusobno logički povezanih, koji se razlikuju po indeksu. Polja su podatkovna struktura koja može spremiti slijednu kolekciju elemenata istog podatkovnog tipa. Zato što se elementi spremaju slijedno u nizu, možemo takvu strukturu nazvati i niz u slučaju jednodimenzionalnog polja.

3.8.1. Jednodimenzionalna polja ili nizovi

Koristimo polja kako bismo umjesto da deklariramo veliki broj različitih varijabli istog podatkovnog tipa, deklarirali jednu varijablu koja može spremiti veliki broj individualnih vrijednosti istog podatkovnog tipa.

Niz grafički možemo predložiti na sljedeći način:

vrijednost[0]	vrijednost[1]	vrijednost[2]	...	vrijednost[n-2]	vrijednost[n-1]

Sva polja se sastoje od slijednih varijabli. Za razliku od C i C++, u C#-u polja su objekti, a ne kontinuirano adresirana memorija. Pristupamo elementima polja preko broja u uglatim zagradama koje odgovaraju poziciji unutar polja. Taj broj naziva se indeks. Prvi član polja kreće s indeksom 0. Za polje koje ima n članova zadnjem članu pristupamo preko indeksa n-1. Budući da je u memoriji polje slijedni niz, pokušaj pristupanja elementu koji se nalazi izvan polja uzrokovat će iznimku tijekom izvođenja.

Deklariramo polje na sljedeći način:

```
podatkovni_tip[] ime_polja;
```

Polje jednom kad ga definiramo moramo instancirati, budući da je ono referentni tip. Da bismo ga instancirali moramo koristiti operator `new` te pomoću njegova podatkovnog tipa instanciramo polje željene veličine. Svakako možemo na više načina to instancirati, pa ćemo u sljedećem primjeru pokazati kako na više načina instancirati polje od 5 članova:

```
// Deklaracija polja od 5 cijelih brojeva
```

```
int[] polje1 = new int[5];
```

```
// Deklaracija i postavljanje vrijednosti
```

```
int[] polje2 = new int[] { 1, 3, 5, 7, 9 };
```

```
// Alternativni način
```

```
int[] polje3 = { 1, 2, 3, 4, 5, 6 };
```

Za početne vrijednosti svi su elementi polja inicirani s početnom vrijednosti na principu 0-bit uzorka, što znači da ako radimo s cijelim brojevima, sve su vrijednosti 0.

Ukoliko mijenjate polje, odnosno dajete već definiranoj varijabli polja novo polje kao u primjeru 3, morate koristiti **new** operator. Ne možete jednom polje instancirati, a zatim pokušati dodijeliti novo polje, nebitno jesu li iste ili različite duljine! Primjer što ne smijete raditi:

```
int[] polje;
```

```
polje = new int[] { 1, 3, 5, 7, 9 }; // OK
```

```
polje = {1, 3, 5, 7, 9}; // Error!!!!
```

Pisanje i čitanje elemenata polja postižemo preko imena polja i indeksa, odnosno pozicije unutar polja. Indeksi počinju s brojem 0, a ne jedan. Tako da pristup n-tom elementu unutar polja dolazimo tako da računamo indeks (n-1) prilikom pristupa. Slijede primjeri postavljanja i čitanja vrijednosti elemenata polja:


```

// Deklaracija polja od 10 cijelih brojeva
int[] polje = new int[10];
polje[0] = 5;           //dodjeljivanje vrijednosti 5 prvom elementu
polje[3] = 43 + 121;    //dodjeljivanje vrijednosti neke operacije
for (int i = 0; i < 10; i++)
{
    //petlja za postavljanje svih vrijednosti polja na 2
    polje[i] = 2;
}
polje[8] = polje[7];    //dodjeljivanje nove vrijednosti čitanjem stare

```

Polje jednom kad je instancirano, ne može promijeniti svoju veličinu. Ukoliko želimo ubaciti element više nego što je to veličina polja, moramo kreirati novo polje za jedan veće, kopirati prijašnje polje u novo i zatim ubaciti novi element. Idemo to prikazati primjerom, gdje ubacujemo novi element na mjesto pos:

```

static void Main(string[] args)
{
    int n = 10;
    int[] polje = new int[n];
    int i;

    // inicijalizacija polja od 10 elemenata
    for (i = 0; i < n; i++) polje[i] = i + 1;

    // ispis originalnog polja
    for (i = 0; i < n; i++) Console.Write(polje[i] + " ");
    Console.WriteLine();

    // element za unos
    int x = 50;

    // pozicija na koju ubaciti element
    int pos = 5;

    // create a new array of size n+1
    int[] novopolje = new int[n + 1];
}

```

```

// kopiranje starog polja i unos na poziciju pos novog elementa
for (i = 0; i < n + 1; i++)
{
    if (i < pos - 1) novopolje[i] = polje[i];
    else if (i == pos - 1) novopolje[i] = x;
    else novopolje[i] = polje[i - 1];
}

// ispis novog polja
for (i = 0; i < n + 1; i++) Console.Write(novopolje[i] + " ");
Console.WriteLine();
}

```

Isto tako, ukoliko želimo smanjiti veličinu polja, potrebno je napraviti novo manje polje te elemente koje želimo iz starog polja kopirati u novo.

Budući da je cijelo polje isto tako objekt, detaljima možemo pristupiti preko već definiranih svojstava. Iako ćemo više o tome u objektno orijentiranom programiranju, spomenut ćemo neka tipična svojstva polja. Kako i s detaljima, tako i sam objekt ima metode pomoću kojih s poljem možemo raditi.

Nekoliko primjera ugrađenih svojstava i funkcija:

```

int n = 10;
int[] polje = new int[n];
int duljina = polje.Length;
int prvi = polje.First();
int zadnji = polje.Last();
int najveći = polje.Max();
int najmanji = polje.Min();

```

Ovo je samo dio, no ima i funkcija za pretragu, spajanja, računanja itd.

Podatkovni tip polja može biti bilo koji podatkovni tip ili klasa, pa time možemo imati i polja od kompleksnih podatkovnih tipova poput stringova ili drugih referentnih tipova.

```

string[] dani = { "Ponedjeljak", "Utorak", "Srijeda", "Četvrtak", "Petak", "Subota", "Nedjelja" };

```

3.8.2. Petlja foreach

Petlje smo već spominjali, ali i spomenimo još jednu petlju koju nismo odmah pogledali. Sada je prilika pogledati kako **foreach** petlja funkcionira. Petlju foreach koristimo da bismo prošli sve elemente nekog polja ili druge iterabilne kolekcije. Petlja će izvršiti definirani blok naredbi za svaki element u instanci kolekcije koju prolazimo. Foreach petlja izgleda ovako:

```
foreach (pod_tip element in kolekcija<pod_tip>)
{
    //naredbe
}
```

Foreach petlju možemo gledati doslovno kao specijalni primjer for petlje. Idemo pogledati primjer kako bismo kroz polje brojeva prošli for petljom, a kako foreach petljom:

```
int[] brojevi = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
//for petlja:
for (int i = 0; i < brojevi.Length; i++)
{
    Console.WriteLine(brojevi[i]);
}
//foreach petlja:
foreach (int broj in brojevi)
{
    Console.WriteLine(broj);
}
```

U definiciji listanja kroz petlju moramo paziti da podatkovni tip elementa koji koristimo u petlji odgovara podatkovnom tipu elementa kolekcije. Ukoliko koristimo ključnu riječ **var** umjesto podatkovnog tipa, bit će nam čak i lakše jer će kompajler biti taj koji će automatski na temelju elemenata kolekcije to odrediti. Kroz brojeve kao u prošlom primjeru možemo proći na sljedeći način:

```
foreach (var broj in brojevi)
{
    Console.WriteLine(broj);
}
```

Kao i u svim petljama ovdje možemo koristiti `break` i `continue`, koji rade što smo i prije rekli. Jedina je razlika što `continue` ovdje prelazi na sljedeći element.

Tijekom korištenja `foreach` petlje nema povratka na prethodni element. Tijekom korištenja `for` petlje samo bismo osigurali da korak petlje ode na prethodni element, no tijekom korištenja `foreach` petlje nema načina prelaska na prethodni element.

Elementu pristupamo preko referentnog tipa, odnosno promjene na varijabli elementa vidljive su i u kolekciji koju iteriramo. Samo budite oprezni, ne možemo izbacivati elemente iz kolekcije ni ubacivati elemente u kolekciju koju iteriramo.

3.8.3. Sortiranje

Pomoću računala rješavamo mnogo problema u našoj svakodnevici. Računala danas skupljaju ogromnu količinu podataka na svakodnevnoj bazi, ali moramo imati neku vrstu sistematizacije i slaganja podataka da bismo mogli podatke obrađivati i razumjeti, odnosno iz podatka dobiti adekvatnu informaciju. Tipičan primjer bio bi da imamo rezultate školskog natjecanja, pa trebamo napraviti rang-listu učenika. U tom slučaju podatke svih učenika koje imamo trebamo sortirati (posložiti) po nekom kriteriju, odnosno broju bodova. Za mali broj rezultata to se relativno brzo napravi, no što ako imate nekoliko stotina ili možda tisuća rezultata? Tipičan primjer toga jest rang lista učenika na državnoj maturi.

Postoji nekoliko vrsta algoritama za sortiranje lista. Neki su jednostavni i intuitivni, no postoje i izuzetno kompleksni algoritmi. Bit algoritma je učinkovitost. Učinkovitost algoritma možemo gledati vremenski i memorijski. Tipičan primjer memorijske učinkovitosti jest ako neki algoritam treba pomoćnu listu, svakako će koristiti puno više radne memorije od algoritma koji treba samo jednu varijablu istog podatkovnog tipa kao polje koje koristimo. Svaki od tih algoritama možemo analizirati na više načina, ali u konačnosti sve se svodi na programera da odabere najbolji algoritam za dotični problem. Primjer odabira: nećete programirajući mikroupravljač koristiti memorijski intenzivan algoritam ako znate da nemate dovoljno radne memorije.

Nekoliko algoritama ćemo objasniti na način da slažemo elemente od najmanjeg do najvećeg.

Selection sort:

Najjednostavniji algoritam za sortiranje jest odabirno sortiranje ili sortiranje zamjenom ili selection sort. Algoritam je relativno jednostavan, no dosta neučinkovit. Svodi se na ono što najčešće radimo, a to je da redom tražimo najmanji element u nesortiranoj listi i zamijenimo ga tako da ga stavimo na prvo mjesto. Nakon toga ostatak polja opet istom metodom pretražujemo dok ne nađemo najmanji od ostatka i tako dalje dok ne ostanemo bez elemenata.

Primjer:

Imamo listu od 5 elemenata (13, 15, 9, 90, 62)

Prvim prolazom kroz listu tražili bismo koji je najmanji element, u ovom slučaju 9, te ga zamijenili prvim elementom. Dobili bismo listu koja izgleda ovako (9, 15, 13, 90, 62).

Kad imamo prvi element, nema potrebe više provjeravati njega, zato ćemo sad gledati ostatak liste, od 2. elementa (15), pa do kraja liste. Tražimo sljedeći najmanji element, odnosno 13 i mijenjamo elementom na drugom mjestu. Lista će nakon tog prolaza i zamjene izgledati ovako (9, 13, 15, 90, 62).

Tako nastavljamo dalje, znajući da je na 2. mjestu najmanji element ostatka liste. Sad opet gledamo ostatak liste, od 3. elementa nadalje, koji je najmanji. Ako je kao u ovom slučaju na 3. mjestu odgovarajući element, nema zamjene. Lista ostaje (9, 13, 15, 90, 62).

Ovako nastavljamo dalje sve dok ne dođemo do zadnjeg elementa, koji ostaje najveći broj iz liste. Sortirana lista nam je (9, 13, 15, 62, 90).

Na temelju ove logike rada možemo napraviti sljedeći kod za unos i sortiranje polja od 50 cijelih brojeva:

```

static void Main(string[] args)
{
    const int dimenzija = 50;
    int[] polje=new int[dimenzija];
    //unos elemenata
    for (int i = 0; i < dimenzija; i++)
    {
        Console.Write("Unesite {0}. element: ", i + 1);
        polje[i]=int.Parse(Console.ReadLine());
    }

    //sortiranje
    for (int i = 0; i < dimenzija; i++)
    {
        //traženje najmanjeg u ostatku polja
        int najmanji_index = i;
        for (int j = i; j < dimenzija; j++)
        {
            if (polje[j] < polje[najmanji_index]) najmanji_index = j;
        }
        //zamjena vrijednosti
        int pom = polje[i];
        polje[i] = polje[najmanji_index];
        [najmanji_index] = pom;
    }

    //ispis:
    Console.WriteLine("Sortirano polje je:");
    for (int i = 0; i < dimenzija; i++)
    {
        Console.Write("%d ", polje[i]);
    }
}

```

Insertion sort:

Sortiranje umetanjem ili *insertion sort* algoritam je sortiranja koji radi tako da jedan po jedan element s liste umeće na odgovarajuće mjesto u prijašnje sortirani dio liste. Svaka iteracija liste ili polja trenutni element pomiče unutar liste dokle god ne dođe do sortirane liste.

Ako konceptualno gledamo, možemo polje koje sortiramo podijeliti na dva dijela, na dio gdje imamo sortirane elemente i na preostale elemente. Sa svakim korakom sortiranja povećavamo dio koji nam sadrži sortirane elemente, a ujedno smanjujemo nesortirani dio. Primjer sortiranja liste:

Početna lista:	20	5	17	4	3	35	22	37	39
Nakon 1. prolaska:	5	20	17	4	3	35	22	37	39
Nakon 2. prolaska:	5	17	20	4	3	35	22	37	39
Nakon 3. prolaska:	4	5	17	20	3	35	22	37	39
Nakon 4. prolaska:	3	4	5	17	20	35	22	37	39
Nakon 5. prolaska:	3	4	5	17	20	22	35	37	39
Nakon 6. prolaska:	3	4	5	17	20	22	35	37	39
Nakon 7. prolaska:	3	4	5	17	20	22	35	37	39

Žutom bojom označavamo element koji trebamo umetnuti u sortirani dio polja, odnosno on označava početak nesortiranog dijela polja.

U sljedećem primjeru napraviti ćemo sortiranje umetanjem za polje od 50 cijelih brojeva:

```
static void Main(string[] args)
{
    const int dimenzija = 50;
    int[] polje = new int[dimenzija];
```

```

//unos elemenata
for (int i = 0; i < dimenzija; i++)
{
    Console.Write("Unesite {0}. element: ", i + 1);
    polje[i] = int.Parse(Console.ReadLine());
}

//sortiranje
for (int i = 1; i < dimenzija; i++)
{
    int j = i;
    while (polje[j] < polje[j - 1] && j > 0)
    {
        int pom = polje[j];
        polje[j] = polje[j - 1];
        polje[j - 1] = pom;

        j--;
    }
}

//ispis:
Console.WriteLine("Sortirano polje je:");
for (int i = 0; i < dimenzija; i++)
{
    Console.Write("%d ", polje[i]);
}
}

```

Bubble sort:

Sortiranje mjehurićem ili *bubble sort* relativno je jednostavan algoritam sortiranja. Radi tako da ponavlja zamjenu susjednih elemenata ako su u krivom redoslijedu. Bubble sort nije pogodan za velike podatkovne skupove jer mu je prosječna kompleksnost i najgori slučaj vremenski dosta skup (traje dugo).

Demonstrirat ćemo primjerom sortiranja. Ako imamo polje koje nije sortirano, definiramo na sljedeći način:

`int[] polje = {5,1,4,2,8};`

U prvom prolazu bubble sort krene s prva dva elementa, uspoređujući ih da provjeri koji je veći.

`{5 1 4 2 8} -> {1 5 4 2 8}`, algoritam uspoređuje prva dva elementa i zamjenjuje ih jer $5 > 1$

`{1 5 4 2 8} -> {1 4 5 2 8}`, zamjena se vrši jer $5 > 4$

`{1 4 5 2 8} -> {1 4 2 5 8}`, zamjena se vrši jer $5 > 2$

`{1 4 2 5 8} -> {1 4 2 5 8}`, algoritam ne zamjenjuje jer su već u dobrom redoslijedu

Drugi prolaz trebao bi izgledati ovako:

`{1 4 2 5 8} -> {1 4 2 5 8}`

`{1 4 2 5 8} -> {1 2 4 5 8}`, zamjena se vrši jer $4 > 2$

`{1 2 4 5 8} -> {1 2 4 5 8}`

`{1 2 4 5 8} -> {1 2 4 5 8}`

Treći prolaz nije baš potreban, jer polje je već sortirano, ali algoritam ne zna je li gotov. Algoritam treba jedan cijeli prolaz kroz polje bez ikakvih zamjena da bi znao da je sortiran:

`{1 2 4 5 8} -> {1 2 4 5 8}`

`{1 2 4 5 8} -> {1 2 4 5 8}`

`{1 2 4 5 8} -> {1 2 4 5 8}`

`{1 2 4 5 8} -> {1 2 4 5 8}`

Kod bi izgledao ovako:

```
static void Main(string[] args)
{
    int[] polje = { 64, 34, 25, 12, 22, 11, 90 };
    int n = polje.Length; //duljina polja
    while (true)
    { //beskonačna petlja koja će se vrtjeti dokle god je ne prekinemo
        bool ima_zamjena = false;
        for(int i = 0; i < n-1; i++)
```

```

        { //prolazak kroz polje
            if(polje[i] > polje[i + 1])
            { //ako je i element veći od i+1 elementa, zamijeni
                ima_zamjena = true;
                int pom = polje[i];
                polje[i] = polje[i + 1];
                polje[i+1] = pom;
            }
        }
        if (!ima_zamjena) break; //ako nije bilo zamjena, prekini while petlju
    }

    //ispis
    Console.WriteLine("Sortirano polje je:");
    for (int i = 0; i < n; i++)
    {
        Console.Write("%d ", polje[i]);
    }
    Console.ReadLine();
}

```

Algoritama sortiranja ima mnogo, svaki sa svojim prednostima i manama. Od ostalih algoritama mogli bismo spomenuti merge sort ili sortiranje spajanjem, shellsort, quicksort itd.

Iako možemo koristiti vlastite algoritme za sortiranje, najčešće ćemo koristiti već dostupne metode sortiranja unutar .NET Frameworka. Tako Array klasa ima statičnu metodu Sort. Poziv izgleda na sljedeći način:

```
Array.Sort(polje);
```

Samo oprezno kad radite s kompleksnim podatkovnim tipovima. Sort funkcija funkcionirat će samo ako elementi polja imaju implementaciju IComparable sučelja. O sučeljima ćemo govoriti u poglavlju 4.11.

3.8.4. Višedimenzionalna polja

Polja mogu biti višedimenzionalna. Iako se u konačnosti svako polje, ma koliko mi rekli da ima dimenzija, sprema kao niz podataka u memoriju. Lakše nam je predložiti si polje kao višedimenzionalno zbog koncepta snalaženja u više dimenzija. Višedimenzionalno polje deklariramo na sličan način kao i jednodimenzionalno, samo u uglatim zagradama kažemo koliko imamo dimenzija. Na sljedeće bismo načine deklarirali i instancirali dvodimenzionalno polje, odnosno matricu ili tablicu:

```
int[,] matrica = new int[2, 3];  
int[,] matrica2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Ovo možemo gledati kao polje čiji su elementi polja, ili u drugoj terminologiji kao kolekciju čiji su članovi druge kolekcije.

Elementima pristupamo tako da indekse razdvajamo zarezom. Ako u matrica2 želimo pristupiti zadnjem elementu (broj 6, koji se nalazi u drugom redu i trećem stupcu) i promijeniti ga u broj 9, napravili bismo to na sljedeći način:

```
matrica2[1,2] = 9;
```

Cijeli program za unos elemenata i ispis napravili bismo na sljedeći način:

```
static void Main(string[] args)  
{  
    int[,] matrica = new int[3, 3];  
    //unos:  
    for(int i = 0; i < matrica.GetLength(0); i++)  
    {  
        for(int j = 0; j < matrica.GetLength(1); j++)  
        {  
            Console.Write("matrica[{0},{1}]= ", i, j);  
            matrica[i, j] = int.Parse(Console.ReadLine());  
        }  
    }  
}
```

```

    }

    //ispis
    Console.WriteLine("Uneseno polje:");
    for (int i = 0; i < matrica.GetLength(0); i++)
    {
        for (int j = 0; j < matrica.GetLength(1); j++)
        {
            Console.Write(matrica[i,j]);
            Console.Write(" ");
        }
        Console.WriteLine();
    }
    Console.ReadLine();
}

```

3.9. Funkcije

Prilikom programiranja često ćemo doći u situaciju gdje program treba ponoviti isti programski kod. Zato te dijelove programa organiziramo u posebno imenovane blokove naredbi koji se nazivaju funkcijama. Iako je C# objektno orijentirani programski jezik, svaka funkcija metoda je neke klase, no o tome ćemo u sljedećem poglavlju.

Osnovna funkcija koju smo i dosad koristili jest Main funkcija (koja je ulazna funkcija svakog programa), dok ostale možemo po volji dodavati. Budući da je svaka funkcija ujedno i metoda, s obzirom na to da je dio klase, definirat ćemo ih na sličan način kao i Main:

```

static povratni_tip imeFunkcije([popis_parametara])
{
    //tijelo funkcije s naredbama
}

```

Ključna riječ **static** u definiciji služi da kaže da metodu možemo pozvati direktno iz

same klase bez da instanciramo novi objekt klase u kojoj se nalazi. Poziv metode se vrši tako da navedemo njeno ime, a zatim u zagradama navodimo potrebne parametre ukoliko su definirani. Svaka proslijeđena vrijednost mora biti podatkovnog tipa kako je definirano u popisu parametara.

Povratni tip funkcije govori nam kojeg će povratnog tipa biti rezultat koji funkcija vraća. Funkcije koje nemaju povratnu vrijednost, tj. ne vraćaju vrijednost, označene su ključnom riječi `void`. Ukoliko funkcija vraća neki povratni podatak, tijelo funkcije mora sadržavati naredbu `return` s odgovarajućim tipom podatka kako je definirano povratnim tipom.

Ime funkcije nam je točno kao to što pojam govori, ime koje ćemo koristiti da bismo pozivali funkciju. Zajedno s listom parametara ime tvori takozvani potpis funkcije. Za imenovanje funkcija vrijede ista pravila koja smo prolazili kad smo radili imenovanja varijabli.

Lista parametara služi nam da bismo znali koje vrijednosti zajedno s podatkovnim tipovima trebamo proslijediti funkciji. Taj parametar zajedno sa svojom proslijeđenom vrijednosti kao lokalna nam je varijabla unutar funkcije. Lista parametara može sadržavati proizvoljan broj parametara, a ne mora sadržavati nijedan. U listi parametara moramo svakako navesti njegov podatkovni tip s imenom parametra koje ćemo koristiti unutar funkcije. Pojedine parametre odvajamo zarezom.

Primjeri funkcija:

```
//funkcija za ispis pozdrava
static void pozdrav()
{
    Console.WriteLine("Pozdrav svijete");
}
```

//funkcija za zbrajanje dva cijela broja

static int suma(**int** a, **int** b)

```
{  
    return a + b;  
}
```

//funkcija za provjeru je li broj prost

static bool prost(**int** n)

```
{  
    for (int i = 2; i < n; i++)  
    {  
        if (n % i == 0) return false;  
    }  
    return true;  
}
```

//funkcija koja vraća veći broj od proslijeđena dva

static int max(**int** prvi, **int** drugi)

```
{  
    int rezultat;  
    if (prvi > drugi) rezultat = prvi;  
    else rezultat = drugi;  
    return rezultat;  
}
```

Funkcije pozivamo navođenjem imena funkcije i zagradama u kojima moramo navesti parametre kako su definirani. Ranije napisane funkcije i pozivali bismo na sljedeći način:

```
pozdrav();  
int s = suma(2, 4);  
bool p = prost(39);  
int m = max(s, 3);
```

Naravno, ovi pozivi podrazumijevaju da se nalazimo u klasi gdje su funkcije definirane. Ukoliko hoćemo pristupiti funkcijama koje se nalaze u nekoj drugoj klasi, to ćemo napraviti preko pisanja imena klase, zatim točkom i onda reći koju funkciju želimo pozvati. Tipičan primjer bilo bi nam koristiti klasu koja sadrži matematičke funkcije koje ne možemo dobiti operatorima poput trigonometrijskih funkcija, logaritamskih i drugih. Primjer s pristupom funkcijama:

```
double kor = Math.Sqrt(2);           //korijen od 2
double opseg = 2 * kor * Math.PI;     //izračun opsega ako je radijus kor i
                                      pristup konstanti pi u Math klasi
var p = Math.Pow(kor, 2) * Math.PI / 2; //izračun površine
var t = Math.Tan(73);                 //izračun tangensa od 73
```

Budući da su sve metode unutar Math klase statične, nema potrebe instancirati klasu, nego samo koristimo metode. Math klasa ni ne može se instancirati, nego je čisto statična klasa, no o tome u sljedećem poglavlju.

Funkciji moramo proslijediti jednak broj parametara i jednakog tipa kako su definirani prototipom funkcije. **Prototip funkcije** deklaracija je funkcije koja se sastoji od povratnog tipa, imena i popisa varijabli sa svojim podatkovnim tipovima. Ako funkcija nema nikakvih parametara koje prosljeđujemo, zagrade svakako moramo koristiti, samo unutar zagrada ostaje prazno (primjer s funkcijom pozdrav()).

U C# programskom jeziku ne moramo parametre prosljeđivati ni redoslijedom koji smo nabrojali. Parametre možemo proslijediti proizvoljnim redoslijedom, samo moramo napisati ono ime kako se parametar zove u funkciji, zatim napisati dvotočku i tek onda vrijednost koju želimo proslijediti. Za primjer ćemo u funkciju max, definiranu prije, proslijediti suprotnim redoslijedom parametre na ovakav način:

```
int m = max(drugi: 4, prvi: 6);
```

Parametre ne moramo uvijek ni proslijediti, no moramo paziti da u tom slučaju kod definicije funkcije postavimo početne vrijednosti za te parametre. Početne vrijednosti postavljamo tako da unutar liste parametara za parametar uz deklariranje radimo i inicijalizaciju. Na takav način inicijalizirana vrijednost bit će ta koja će se uzeti ukoliko

parametri nisu proslijeđeni. U listi kad jednom inicijaliziramo jedan parametar, ostatak liste mora biti inicijaliziran. Primjer:

```
static void PozdravOd(string pozdrav, bool prikazilme = false,string ime="Per  
                   o",string prezime = "Perić")  
{  
    Console.Write(pozdrav);  
    if (prikazilme)  
    {  
        Console.Write(" od " + ime + " " + prezime);  
    }  
    Console.WriteLine("!");  
}
```

Pozive ove funkcije možemo izvršiti na nekoliko načina i u više kombinacija kao u sljedećem primjeru:

```
PozdravOd("Pozdrav");  
PozdravOd("Pozdravlja vas", true);  
PozdravOd("Pozdravlja vas", true,prezime:"Marulić");
```

Više funkcija može imati isto ime, samo se moraju razlikovati po prototipu. Ako imaju isto ime, mogu se razlikovati samo po listi parametara. To se zove preopterećivanje (overloading) funkcija. Lista parametara ne mora se razlikovati samo po broju parametara, nego može i po tipu parametara. Znači, možemo imati više funkcija s jednakim brojem parametara koje se isto zovu, ali podatkovni tipovi parametra moraju se razlikovati u barem jednom parametru.

```
public abstract int max(int a, int b);  
public abstract float max(float a, float b);  
public abstract char max(char a, char b);  
public abstract double max(double a, double b);  
public abstract double max(double a, int b);  
public abstract double max(double a, double b, double c, double d);  
public abstract double max();
```


Zasad ignorirajte *public abstract* dio, idemo se koncentrirati na ostatak.

Gore definirani prototipovi funkcija redom su ispravni i prevoditelj će ih prevoditi bez ikakve greške. Kako što smo rekli, mogu imati isto ime, no lista parametara drukčija je kod svake funkcije.

Funkcije ne možemo preopterećivati po povratim tipovima.

```
public abstract int max(int a, int b);  
public abstract void max(int a, int b); //ne smijemo!
```

Pokušaj ovakvog definiranja funkcija dovest će do greške tijekom prevođenja programa. Budući da se tijekom izvođenja programa kreira izvršna tablica funkcija, funkcije se mogu razlikovati samo po imenu i parametrima. Tijekom izvođenja nije moguće napraviti razliku po povratnom tipu, stoga prevoditelj ovakve definicije odbija uopće prevesti.

3.9.1. Prosljeđivanje parametara

Parametre u funkciji možemo prosljeđivati na dva načina, po vrijednosti i po referenci. Dosad smo radili samo prosljeđivanje po vrijednosti. To znači da mi u funkciju prosljeđujemo samo vrijednosti određenih varijabli. Povratkom u dio koji je pozvao funkciju vrijednost varijable ne mijenja se u originalnom dijelu programa.

Pozivom funkcije po referenci moguće je raditi izmjene na varijablama u dijelu programa koji je pozvao funkciju.

U C#-u vrijednosti se automatski prosljeđuju po vrijednosti, osim ako se ne radi o referentnim tipovima (najčešće objektima), onda se prosljeđuje referenca na objekt. Za rad s prijenosom po referenci možemo koristiti dva načina, odnosno dvije ključne riječi: **ref** i **out**.

Ključnom riječi `ref` ispred podatkovnog tipa varijable prosljeđujemo varijablu po referenci. To znači da svaka izmjena varijable unutar funkcije utječe na njenu vrijednost i prilikom povratka u dio programa koji je pozvao tu funkciju. Kod poziva metode koja ima definirani parametar `ref` ispred imena varijable moramo napisati ključnu riječ **`ref`**.

Uzmimo sljedeći primjer za demonstraciju:

```
static void Main(string[] args)
{
    int x = 1;
    Console.WriteLine(x);
    povecaj1(x);
    Console.WriteLine(x);
    povecaj2(ref x);
    Console.WriteLine(x);
}

static void povecaj1(int a)
{
    a++;
}

static void povecaj2(ref int a)
{
    a++;
}
```

U ovom primjeru testiramo prosljeđivanje varijable `x` po vrijednosti pa po referenci. Na početku varijabla `x` ima vrijednost 1 inicirano, što se i ispisuje s prvom `Console.WriteLine` naredbom. Nakon toga se pozivom `povecaj1` funkcije unutar funkcije povećava vrijednost prosljeđene vrijednosti, no povratkom iz funkcije ispisuje se opet 1 kao i prije toga. Tomu je tako jer prosljeđivanjem po vrijednosti mi ne mijenjamo varijablu u dijelu programa koji je pozvao funkciju, odnosno prosljedili smo vrijednost iz varijable `x`. U slučaju funkcije `povecaj2` to je drukčije. Mi ne prosljeđujemo vrijednost iz varijable `x`, nego ključnom riječi `ref` prosljeđujemo referencu na varijablu `x`. Budući da funkcija `povecaj2` sad radi s referencom na `x` spremljenom u lokalnoj varijabli, bilo kakve izmjene nastale u funkciji bit će vidljive i izvan funkcije. Povratkom vidimo da se sad vrijednost povećala i program ispisuje 2 na ekranu.

Prosljeđivanje po referenci i vrijednosti možete gledati kao kopiju i original. Kad prosljeđujemo po vrijednosti, mi, u prenesenom značenju, funkciji šaljemo kopiju podatka te bilo kakvi zapisi na kopiji ostaju unutar funkcije. Slanjem po referenci je kao da dajemo original, pa će bilo kakve izmjene ili šaranja po originalu nama biti vidljive kad nam se original vrati.

Ključna riječ `out` malo je drukčija od `ref`. Kod `ref` podrazumijeva se da je varijabla inicijalizirana te time i da ima neku referencu sama na sebe. Ključnu riječ `out` koristit ćemo kada nismo sigurni je li varijabla uopće inicijalizirana. Napomenuti moramo samo da ukoliko pokušavamo koristiti neinicijaliziranu varijablu, kompajler javlja grešku.

Pomoću ključne riječi `out` prosljeđena varijabla, ako nije inicijalizirana, bit će inicijalizirana unutar funkcije.

```
static void Main(string[] args)
{
    int x;
    inicijaliziraj(out x);
    Console.Write(x);
}
private static void inicijaliziraj(out int a)
{
    a = 1;
}
```

Na ovakav način nema greške kompajlera ne javlja grešku jer koristimo neinicijaliziranu varijablu. Ovakav tip inicijalizacije varijable često ćemo koristiti prilikom pokušaja parsiranja teksta. Metoda **TryParse** sastavni je dio skoro svakog numeričkog tipa, a pokušava pretvoriti tekst u broj u jednom koraku. Ako je moguće, inicijalizira varijablu, a ako to nije moguće, varijabla će biti inicijalizirana na vrijednost 0. Metoda `TryParse` vraća i boolean tip koji nam govori je li parsiranje bilo uspješno. Primjer:

```
static void Main(string[] args)
{
    int x;
    if(int.TryParse(Console.ReadLine(), out x))
```

```

    {
        Console.WriteLine("Uspješno unesen broj.");
    }
    else
    {
        Console.WriteLine("Nije unesen broj!");
    }
}

```

3.9.2. Rekurzivne funkcije

Rekurzivne funkcije jesu funkcije koje pozivaju same sebe. U osnovi to znači da ćemo negdje unutar implementacije funkcije imati opet poziv funkcije koju implementiramo. Isprva se čini malo nelogično, ali kad se malo dublje uđe u programiranje, imat će uvelike smisla neke algoritme implementirati rekurzivnom funkcijom jer će to biti najlakše.

Primjer rekurzije:

```

void funkcija ([argumenti]) {
    //naredbe funkcije
    funkcija([argumenti_drugi]);
    // naredbe funkcije
}

```

Temelj rekurzivnih funkcija možemo pronaći u matematici, gdje također postoje rekurzivne definicije članova redova i nizova. U principu to se svodi na to da se sljedeći element niza definiran nekim prethodnim elementom ili elementima.

Kod rekurzivnih funkcija moramo paziti na to da ne uđemo u beskonačnu petlju, koja se lagano može dogoditi. Zato je prilikom definiranja rekurzije obavezno imati neki uvjet prekida rekurzije, prije nego što funkcija opet samu sebe pozove, te sa svakim pozivom rekurzivne funkcije moramo se približavati uvjetu zaustavljanja. To ćemo demonstrirati na primjeru, obratite pozornost na prvu liniju unutar funkcije fakt u sljedećem primjeru.

Prvi primjer na kojem ćemo demonstrirati jesu faktorijele. Faktorijele možemo matematički zapisati na sljedeći način:

$$1! = 1$$

$$2! = 2 * 1! = 2 * 1 = 2$$

$$3! = 3 * 2! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3! = 4 * 3 * 2 * 1 = 24$$

...

$$n! = n * (n - 1)! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$$

Ukratko, vidimo pravilo ponašanja, svaku sljedeću faktoriјelu možemo izračunati množenjem prethodno izračunate faktoriјele formulom $n! = n * (n - 1)!$. Funkciju „fakt” napravimo tako da odmah definiramo uvjet izlaza iz rekurzije, tako da ako je $n \leq 1$, da jednostavno vrati 1, a u suprotnom da vrati n pomnožen s izračunom prethodnih faktoriјela. Na temelju toga napraviti ćemo sljedeći kod za rekurzivno rješavanje faktoriјela i ispis u Main funkciji:

```
static void Main(string[] args)
{
    int a;
    Console.Write("Unesite broj: ");
    a = int.Parse(Console.ReadLine());
    Console.WriteLine("{0}!={1}", a, fakt(a));
}

static int fakt(int n)
{
    if (n <= 1) return 1;
    return n * fakt(n - 1);
}
```

3.10. Objektno orijentirano programiranje

Kod **objektno orijentiranog programiranja** (skraćeno OOP), programer polazi od objekta iz realnog svijeta i njegov je posao napraviti model tog objekta u programu. Taj model apstrakcija je stvarnog objekta, kojem se kontrolirano uključuju i isključuju neki detalji, koji su bitni ili nisu bitni za sam program. Objekti posjeduju statičke i dinamičke osobine. Ako uzmemo za primjer automobil, možemo reći da on ima boju, marku, motor sa svojim specifikacijama itd. Taj automobil može se kretati, stajati, okrenuti se, a za te osobine kažemo da su dinamičke. Akcija koju objekt napravi zapravo je sastavni dio tog objekta.

Objektno orijentirano programiranje vodi se ovim činjenicama, pa stoga objekti imaju svoja unutrašnja stanja koja nazivamo svojstvima, ali i skup akcija koje objekt može izvršiti, koje nazivamo metodama. Samo programiranje svodi se na sklapanje više objekata u jednu cjelinu.

U radu s računalom, osnovna misao vodilja automatizacija je poslova. Tako kod OOP-a objekte iste vrste tretiramo na isti način.

Klasa je apstrakcija konkretnih objekata iste vrste. Svaki objekt u programu objekt je određene klase, koja objekt određuje svojim sučeljem (interface), odnosno njegovim svojstvima, metodama, događajima, preko kojih možemo komunicirati s objektom u programu. Za objekt kažemo da je instanca klase, jer se na temelju konstruktora unutar klase on gradi.

Objekti su osnovna modularnost i strukturiranost objektno orijentiranog programiranja. Oni predstavljaju samostalne cjeline koje možemo sklapati s drugim objektima u skladu s našim potrebama i tako rješavati složene probleme. Svaki objekt može komunicirati s drugim objektima u smislu da im šalje poruke i prima od njih poruke.

Objektima se može pristupiti samo preko njegova sučelja koristeći javno dostupna svojstva i metode. Sakrivenost unutarnje funkcionalnosti rada objekta preko njegovih javno dostupnih metoda i sučelja zovemo enkapsulacija.

Nasljeđivanje je jedan od vrlo bitnih koncepata OOP-a, ono je posljedica generalizacije. Nasljeđivanje možemo smatrati kao proširivanje funkcionalnosti klase koju smo naslijedili, odnosno u nekim slučajevima kažemo da smo čak i specijalizirali klasu koju nasljeđujemo. Samim postupkom nasljeđivanja naslijedili smo sve attribute, metode, svojstva polazne odnosno bazne klase.

Ovisno o objektu ista metoda može djelovati različito u svojem izvođenju. Tu pojavu nazivamo **polimorfizam**. Apstrakcijom klasa, odnosno izradom generalne klase, mi možemo napraviti jedinstveno programsko sučelje za rad s objektima, a specijalizacijom te klase možemo dobiti konkretno klasu koja će nam moći izraditi te objekte. Polimorfizam možemo predložiti kao da imamo apstraktnu klasu vozilo. Klasa vozilo ne bi rezultirala nijednim objektom, no ako iz te klase napravimo klasu automobil, brod i slično, preko tih izvedenih klasa instancirali bismo objekt. No tijekom rada, da bismo imali unificirano sučelje prema svim klasama, gledali bismo ih preko sučelja klase vozilo.

3.10.1. Klase i objekti

Objekti su nam projekcije objekata iz stvarnog svijeta u našem programu. Osobine objekata preslikavaju se u svojstva (*properties*), a radnje koje objekt radi postaju nam metode u programskim objektima. Budući da stvarni objekti imaju određene podražaje izvana, tako će programski objekti imati događaje koji su reakcija na utjecaj drugih objekata.

Za svaki objekt možemo reći da je određene klase. Tako ćemo u programu raditi klase koje nam opisuju objekte te ćemo na temelju tih klasa instancirati, odnosno graditi objekte. Svaka klasa definira se na sljedeći način:

```
class ImeKlase
{
    //članovi (fields)
    //svojstva (properties)
    //metode (methods)
    //događaji (events)
    //delegati (delegates)
}
```

Klasa se definira navođenjem ključne riječi **class**, imenom klase i nakon toga vitičaste zagrade unutar kojih definiramo članove, svojstva, metode, događaje itd.

Najbolje je na primjeru klase definirati klasu Osoba sa svojim elementima na sljedeći način:

```
class Osoba
{
    //članovi (fields)
    public string Ime="";
    public string Prezime="";
    public DateTime DatumRodjenja;
    //konstruktori
    public Osoba() {}
    public Osoba(string ime, string prezime, DateTime datumRodjenja)
    {
        Ime = ime;
        Prezime = prezime;
        DatumRodjenja = datumRodjenja;
    }
    //metode (methods)
    public int Dob()
    {
        return (DateTime.Now - DatumRodjenja).Days / 365;
    }
}
```

Klasu Osoba definirali smo tako da smo naravno prvo naveli ključnu riječ **class**, zatim ime klase. Na početku klase definirali smo i inicijalizirali članove klase (class fields, koji su ime, prezime i datum rođenja). Članove ćemo definirati kao i varijable unutar programa, samo nam ovdje te varijable imaju doseg unutar svih elemenata klase. Napisano je pravilo da javne članove klase i javna svojstva u imenovanju započinju s velikim slovom.

Metode unutar klase ono su što smo prije nazivali funkcije. Ovdje ćete vidjeti da imamo metode koje se zovu jednako kao i klasa te nam predstavljaju metode koje grade

objekt, odnosno to su nam konstruktori klase. Konstruktor se automatski poziva prilikom instanciranja novog objekta klase i u njemu se obično inicijaliziraju vrijednosti člana i svojstva novokreiranog objekta. Naša klasa ima definirana dva konstruktora, što znači da imamo više načina na koje možemo instancirati novi objekt (polimorfizam). Konstruktor nije obavezno definirati, no u tom se slučaju svakako automatski generira konstruktor klase (prazan kao u našem primjeru i naziva se *default constructor*). Ukoliko definiramo konstruktor koji prima parametre, ako želimo konstruktor bez parametara, moramo ga zasebno definirati kao u gornjem primjeru.

Objekte klase Osoba možemo pomoću ova dva konstruktora instancirati na više načina:

```
Osoba o1 = new Osoba();
```

```
Osoba o2 = new Osoba("Marko", "Marulić", DateTime.Parse("18.8.1450"));
```

Prvo je instanciranje preko prvog konstruktora, no tako bi nam izgledalo instanciranje i da nismo naveli nijedan konstruktor. Drugo instanciranje događa se koristeći drugi konstruktor, gdje smo preko konstruktora prosljedili podatke na temelju kojih će se vršiti konstrukcija objekta.

U klasi, ali i općenito, moramo svakako spomenuti modifikatore pristupa. Već smo mnogo puta spominjali `public`, no sad je vrijeme da objasnimo što znače.

Svi tipovi i tipizirani članovi imaju svoju razinu pristupa. Razina pristupa kontrolira da li se taj član može koristiti iz nekog drugog dijela koda.

U C# imamo sljedeće modifikatore pristupa:

- **public** – pristupačno iz svih dijelova koda
- **private** – pristupačno samo kodu unutar iste klase ili strukture
- **protected** – pristupačno samo kodu u istoj klasi ili u deriviranoj (naslijeđenoj) klasi
- **internal** – pristupačno kodu unutar istog assemblya, odnosno iste kompilacije
- **protected internal** – pristupačno kodu unutar iste kompilacije ili iz derivirane klase
- **private protected** – pristupačno od strane tipova deriviranih iz klase koja je deklarirana unutar iste kompilacije.

Nisu svi modifikatori pristupa validni za sve vrste članova. U nekim slučajevima tip člana ograničen je zbog pristupačnosti člana kojeg sadrži.

Klase, zapisi i strukture deklarirane unutar istog imenskog prostora (*namespacea*) mogu biti public ili internal. Ukoliko nijedan nije specificiran, po automatizmu je internal.

Članovi strukture, uključujući ugniježdene klase i strukture, mogu biti deklarirani kao public, internal ili private. Članovi klase, uključujući ugniježdene klase i strukture, mogu biti deklarirani bilo kojim gore navedenim modifikatorom. Ukoliko nije ništa navedeno, automatski se smatra da je modifikator private, što znači da neće biti moguće pristupiti mu izvan objekta u kojem se nalazi.

Derivirane klase ne mogu imati višu razinu pristupačnosti nego njihova bazna klasa. Ne možemo deklarirati „public class B” koja je naslijedila „internal class A” jer bi u tom slučaju klasa A bila public, jer su svi njezini protected i internal članovi pristupačni deriviranoj klasi.

Nakon izgradnje objekta s definiranim razinama pristupa možemo pristupati članovima klase. Pristupamo članovima klase tako da jednostavno upišemo ime varijable kako smo je imenovali, pa točka i ime člana kojem želimo pristupiti. Recimo da želim na temelju prijašnjeg primjera izračunati dob za o2, onda bih to napravio na sljedeći način:

```
o2.Dob();
```

U ovom slučaju, da je metoda Dob ispred sebe imala klasifikator static, onda ne bih morao ni instancirati klasu, a mogao bih pristupiti metodi preko imena klase i imena metode. Statičnim članovima ne moramo prozvati samo metode, nego možemo static ključnu riječ navesti i za individualne članove, poput varijabli unutar klase.

3.10.2. Enkapsulacija

Enkapsulacija je proces u kojem vežemo podatke i metode u jednu jedinu jedinicu. Istinsku enkapsulaciju dobivamo izradom svojstava(properties) koja nam omogućuju pristup članovima klase preko strogo definiranog sučelja (getera i setera).

Svojstva definiramo na sljedeći način:

```
modifikator tip ImeSvojstva
{
  get { return vrijednost_tipa; }
  set { lokalna_varijabla=value; }
}
```

Svojstva mogu biti razine pristupa kao i svi ostali članovi klase. Svojstva možemo gledati kao funkcije kojima pristupamo kao da su varijable. Da bismo detaljnije pogledali, idemo u klasi Osoba modificirati da ime koristimo kao svojstva:

```
private string ime="";
public string Ime
{
    get { return ime; }
    set { ime = value; }
}
```

Svojstva mogu imati dva bloka naredbi:

Get blok omogućuje da objekt vrati vrijednost koja je podatkovnog tipa kao definirano svojstvo, odnosno koristimo ga za čitanje svojstva.

Set blok koristimo za proslijeđivanje vrijednosti svojstvu korištenjem operatora jednako (=). Kažemo da postavljamo vrijednost svojstva, bilo da direktno proslijeđujemo vrijednost u neku varijablu kao u našem primjeru ili u nekako naizgled potpuno nezvezano, da vrijednost uzmemo i obradimo, pa može utjecati na više članova klase ili uopće ne utjecati.

Ne moraju svaki put biti oba bloka definirana. Ukoliko nemamo get blok, svojstvo je samo za pisanje, a ukoliko nemamo set blok, kažemo da je svojstvo samo za čitanje.

Unutar ovih blokova programeri najčešće provode sve potrebne provjere da u programima koji će raditi s novim objektima ne dođe do nekakvih grešaka. Na ovakav način mi sakrivamo internu implementaciju klase, odnosno možemo preko svojstava prezentirati podatke na sasvim drukčiji način nego što su interno zapisani.

Svojstva bi praktično gledano mogli implementirati i preko metoda. Recimo da želimo napraviti svojstvo ili metodu `ImeIPrezime`. To možemo napraviti na sljedeće načine:

```
public string ImeIPrezime
{
    get { return ime+" "+prezime;}
}
public string ImeIPrezime()
{
    return ime + " " + prezime;
}
```

Kad gledamo, nema razlike hoćemo li ćemo koristiti svojstvo ili metodu, doslovno se svodi na stil programera. Jedina je razlika u korištenju, odnosno u pozivu. Svojstvo pozivamo kao da čitamo iz varijable, dok metodu moramo pozivati kao funkciju.

3.10.3. Događaji

Događaji su posebne metode koje se izvršavaju kad se dogodi neka promjena u programu, odnosno kada objekt signalizira da se dogodila promjena koja će signalizirati da se mora pokrenuti neki drugi dio koda. U konzolnim aplikacijama nisu toliko bitna stvar, no događaji su temelj za rad s korisničkim sučeljem koje se gotovo cijelo temelji na događajima (klik, ulazak miša, izlazak, pritisak tipke, kotačić...).

Da bismo demonstrirali događaj kako radi, napraviti ćemo događaj koji će se podići na promjenu prezimena osobe:

```
class Osoba
{
    //delegat
    public delegate void PromjenaPrezimanaDelegat(object sender, EventArgs e);
    //događaj
    public event PromjenaPrezimanaDelegat PromjenaPrezimana;
    private string prezime;
    public string Prezime
    {
        get { return prezime; }
        set
        {
            prezime = value;
            //Provjera je li događaj implementiran u glavnom programu
            if(PromjenaPrezimana != null)
            {
                PromjenaPrezimana(this, new EventArgs());
            }
        }
    }
}
```

Za implementaciju događaja moramo deklarirati događaj i napraviti njegov delegat. Delegat nam je pokazivač na preusmjeravanje izvođenja programa iz klase u neki kod koji se nalazi negdje drugdje. U suštini, delegat je taj koji pokreće izvršavanje koda koji se nalazi na nekom drugom mjestu, vjerojatno unutar neke druge klase. Provjera null vrijednost obavezna je jer ukoliko pokušamo pokrenuti nepostojeći dio programa, odnosno pozvati funkciju koja ne postoji, javit će nam se greška.

U glavnom dijelu programa kod će izgledati ovako:

class Program

```
{  
    static void Main(string[] args)  
    {  
        Osoba o = new Osoba();  
        //dodavanje metode koju treba izvršiti na promjenu prezimena  
        o.PromjenaPrezimana += PromjenaPrezimana;  
        o.Prezime = "Jelačić";  
  
        //micanje metode za izvršavanje na promjeni prezimena  
        o.PromjenaPrezimana -= PromjenaPrezimana;  
        o.Prezime = "Šokčević";  
    }  
    private static void PromjenaPrezimana(object sender, EventArgs e)  
    {  
        Console.WriteLine("Prezime promijenjeno!");  
        Console.WriteLine("Novo prezime je " + ((Osoba)sender).Prezime);  
    }  
}
```

Kao što vidimo, događaj koji se izvršava nije implementiran unutar klase, nego negdje drugdje, te se samo veže na delegat. Kako se povezuje operatorom +=, tako se naravno veza može i maknuti operatorom -=.

3.10.4. Nasljeđivanje

Nasljeđivanjem proširujemo funkcionalnost bazne klase s novom funkcionalnosti klase koju izvodimo. Nasljeđivanjem postizemo da sve članove koji su nam definirani u baznoj klasi ne moramo ponovno definirati u naslijeđenoj klasi te nam tako kod koji nasljeđujemo ostaje samo na jednom mjestu, a to je u baznoj klasi.

C# je u potpunosti objektno orijentiran programski jezik, tako da svaki element koji definiramo nasljeđuje takozvanu superklasu, odnosno baznu klasu svih objekata, a to je klasa „object”. Zbog toga svaka klasa ima metode poput ToString, Equals, GetType i druge.

Prilikom definiranja klase nakon imena klase stavljamo dvotočku i navodimo ime klase koju nasljeđujemo:

```
class Derivirana_klasa : Bazna_klasa
{
    ...
}
```

Prikažimo nasljeđivanje primjerom u kojem imamo baznu klasu Osoba sa svojim članovima, te deriviranu klasu Učenik.

Klasa Osoba može izgledati ovako:

```
public class Osoba
{
    //članovi:
    private string prezime;
    private string ime;
    private DateTime datumRodjenja;

    //Svojstva:
    public string Prezime
    {
        get { return prezime; }
        set { prezime = value; }
    }
    public string Ime
    {
        get { return ime; }
        set { ime = value; }
    }
    public DateTime DatumRodjenja
    {
        get { return datumRodjenja; }
        set { datumRodjenja = value; }
    }

    //Konstruktori:
    public Osoba(string prezime, string ime, DateTime datumRodjenja)
    {
        Prezime = prezime;
        Ime = ime;
        DatumRodjenja = datumRodjenja;
    }
    public Osoba() {}

    //Metode:
    public string Punolme()
    {
        return Ime + " " + Prezime;
    }
}
```

Klasa Učenik može izgledati ovako:

```
public class Ucenik : Osoba
{
    //Konstruktori:
    public Ucenik() { }
    public Ucenik(string ime,string prezime,DateTime datumRodjenja)
    :base(ime, prezime, datumRodjenja) { }

    //članovi:
    private int matematika;

    //Svojstva:
    //Novo Svojstvo u deriviranoj klasi
    public int Matematika
    {
        get { return matematika; }
        set { matematika = value; }
    }

    //Metode:
    //Premošćivanje metode naslijeđene iz klase objekt
    public override string ToString()
    {
        return Punolme();
    }
    public string Punolme()
    {
        return Ime + ", " + Prezime;
    }
}
```

Klasa Ucenik nasljeđuje se iz klase Osoba, što znači da će iz bazne klase Osoba dobiti svojstva, metode itd.

Konstruktori se ne nasljeđuju automatski, pa ih moramo implementirati unutar klase Ucenik. No možemo koristiti konstruktor iz bazne klase i praktički ga nadograditi. Elementima bazne klase možemo pristupiti ključnom riječi base, ali svakako moramo paziti na to imamo li odgovarajuće pravo pristupa elementu. U klasi Ucenik mi možemo pristupiti javnim članovima, no ne možemo privatnim.

U ovom primjeru treba spomenuti još jednu metodu koja se koristi prilikom nasljeđivanja, a to je premošćivanje. U našem primjeru mi smo uzeli naslijeđenu metodu ToString i ključnom riječi **override** premostili njenu baznu funkcionalnost. Često će biti slučaj da nam bazna funkcionalnost sa stajališta naše klase izgleda besmisleno, zato na ovakav način možemo prepraviti da metoda u naslijeđenoj klasi radi smisleniji posao.

U C# postoje ograničenja u nasljeđivanju. Naslijediti možemo samo jednu klasu. To ograničenje može biti dobro jer nam ne dozvoljava da imamo višestruka nasljeđivanja iste klase. Ujedno, nasljeđivanje samo jedne klase može biti gnjavaža, pogotovo u situaciji gdje trebamo višestruko nasljeđivanje.

Iako višestruko nasljeđivanje nije dozvoljeno kad su klase u pitanju, dozvoljeno je raditi višestruko nasljeđivanje sučelja (interface 4.11 poglavlje). No o tome ćemo u kasnijim poglavljima.

Nasljeđivanje je iznimno korisno kad nasljeđujemo već definirane klase. Recimo da želimo napraviti kolekciju, naslijedili bismo klasu System.Collection.CollectionBase i proširujući tu klasu napravili bismo svoju vlastitu kolekciju. Kod programiranja Windows aplikacija nasljeđujemo klasu Form, koja se nalazi u System.Windows.Forms, i na taj način već koristimo definirani prozor, pa samo proširujemo mogućnosti dodavanjem članova u našu klasu, poput gumbova i drugih elemenata korisničkog sučelja.

3.10.5. Polimorfizam

Polimorfizam je osobina po kojoj metoda ili funkcija ima parametre različitih tipova, ali se jednako se zove. Često se smatra da je treći stup OOP-a, nakon enkapsulacije i nasljeđivanja.

Ima dva vrlo bitna aspekta:

- tijekom izvođenja objekt derivirane klase može se tretirati kao objekt bazne klase u smislu metoda, parametara i kolekcija. Kad se ovo dogodi, deklarirani objektni tip nije identičan njegovu tipu tijekom izvođenja

- bazne klase mogu definirati i implementirati virtualne metode, a derivirane klase ih mogu premošćivati, što znači da derivirane klase pružaju svoju definiciju i implementaciju virtualnih metoda. Tijekom izvođenja, kad klijentski kod poziva metodu, CLR pregledava tip objekta u izvođenju i poziva tu premošćenu virtualnu metodu. U kodu se može pozivati metoda na baznoj klasi, no u konačnosti izvodi se metoda derivirane klase.

Virtualne metode omogućuju nam da radimo s grupama ovisnih objekata na uniformni način. Uzmimo za primjer da trebamo napraviti aplikaciju za crtanje gdje korisniku dajemo da radi razne oblike. Prilikom kompajliranja ne znamo s kojim će specifičnim oblikom korisnik raditi. No aplikacija mora pratiti sve različite tipove i oblike koji se stvaraju te ih ažurirati sukladno akcijama korisnika. Polimorfizam rješava ovaj problem u dva jednostavna koraka:

1. stvara hijerarhiju klasa u kojoj se svaki specifičan oblik derivira iz jedinstvene bazne klase
2. koristi virtualne metode za pozivanje odgovarajuće metode u bilo kojoj deriviranoj klasi preko jedinstvene bazne klase.

Da to demonstriramo, napraviti ćemo baznu klasu Oblik te iz nje derivirati klase Kvadrat, Krug i Trokut. Klasa Oblik imat će virtualnu metodu Nacrtaj, koja će biti premošćena u svakoj deriviranoj klasi jer treba nacrtati oblik koji predstavlja klasa. Primjer bi izgledao ovako:

```
public class Oblik
{
    // Nekoliko svojstava
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Visina { get; set; }
    public int Sirina { get; set; }
    // Virtualna metoda
    public virtual void Nacrtaj()
    {
        Console.WriteLine("Crtanje iz bazne klase");
    }
}
```

```

}
public class Krug : Oblik
{
    public override void Nacrtaj()
    {
        // Kod za crtanje kruga
        Console.WriteLine("Crtanje kruga");
        base.Nacrtaj();
    }
}
public class Kvadrat : Oblik
{
    public override void Nacrtaj()
    {
        // Kod za crtanje kvadrata
        Console.WriteLine("Crtanje kvadrata");
        base.Nacrtaj();
    }
}
public class Trokut : Oblik
{
    public override void Nacrtaj()
    {
        // kod za crtanje trokuta
        Console.WriteLine("Crtanje trokuta");
        base.Nacrtaj();
    }
}

```

Stvorit ćemo polje oblika `Oblik[]` i dodati krug, trokut i kvadrat. Prilikom ažuriranja nacrtanih objekata, koristeći `foreach` petlju iterirat ćemo kroz listu i pozivati metodu `nacrtaj` na svakom obliku. Iako je svaki objekt liste deklariran kao `Oblik`, tijekom izvođenja, budući da smo gradili objekte koristeći derivirane klase, pozivat će se metoda klase koja je gradila objekt.

```

Oblik[] oblici =
{
    new Trokut(),
    new Kvadrat(),
    new Krug()
};
foreach(var oblik in oblici)
{
    oblik.Nacrtaj();
}
// Ispis ovog dijela programa:
/*
Crtanje trokuta
Crtanje kvadrata
Crtanje kruga
*/

```

Virtualni članovi:

Kad derivirana klasa nasljeđuje iz bazne klase, dobiva sve metode, članove, polja, svojstva i događaje bazne klase. Dizajner derivirane klase ima različite izbore za ponašanje virtualnih metoda:

- derivirana klasa može premostiti virtualne članove u baznoj klasi, stvarajući tako novo ponašanje
- derivirana klasa može naslijediti najbližu metodu bazne klase bez da je premosti, sadržavajući postojano ponašanje, ali omogućujući daljnjim deriviranim klasama da premoste njeno ponašanje
- derivirana klasa može definirati nove nevirtualne implementacije koje sakrivaju implementaciju bazne klase.

Derivirana klasa može premostiti člana bazne klase samo ako je deklariran kao virtual ili abstract. Mora se koristiti ključna riječ override da se eksplicitno kaže da će se metoda premostiti.

Ne mogu svi elementi biti virtualni. Samo metode, svojstva, događaji i indekseri mogu biti virtualni. Virtualne metode i svojstva omogućuju deriviranoj klasi da proširi baznu klasu bez potreba da se koriste implementacije metoda bazne klase.

Sve što je deklarirano kao virtualno u baznoj klasi, moramo implementirati u klasi koju planiramo koristiti. Virtualne klase ne mogu se instancirati, zato bazne klase ako imaju virtualne članove ne instanciramo, nego samo njihove derivirane klase, a bazne klase koristimo kao sučelje za rad s više različitih deriviranih klasa.

3.10.6. Otvorena rekurzija

Otvorena rekurzija svojstvo je da jedna metoda u svojoj implementaciji može pozvati drugu metodu unutar istog objekta. To se u C#-u omogućuje upotrebom ključne riječi „this”, gdje this predstavlja referencu na objekt u kojem se trenutno nalazimo. Iako pojam otvorene rekurzije nije baš jednak pojmu rekurzije (funkcija zove samu sebe), značenje otvorene rekurzije jest da objekt može referencirati samog sebe tijekom rada. To svojstvo često ćemo koristiti u slučajevima kada imamo isto imenovane lokalne varijable metode kao i članove klase, da znamo da se referenciramo na člana klase.

Uz this nekoliko smo puta spomenuli i base ključnu riječ. Base nam služi slično kao this, samo korištenjem base ključne riječi pristupamo dijelovima klase koju smo naslijedili u našoj klasi.

Primjer korištenja this objekta:

```
public class Radnik
{
    private string alias;
    private string ime;
    public Radnik(string ime, string alias)
    {
        // Upotreba this da pristupamo članovima klase umjesto parametrima
        this.ime = ime;
        this.alias = alias;
    }
}
```

4. POGLAVLJE: .NET razvojni okvir

U OVOM POGLAVLJU NAUČIT ĆETE:

- raditi s iznimkama
- raditi naprednije sa stringovima
- čitati datoteke i pisati u datoteke
- raditi kompleksne strukture podataka
- raditi pretvorbe podatkovnih tipova
- raditi s datumom i vremenom
- LINQ.

4.1. Razvojni okvir

.NET razvojni okvir besplatan je, otvorenog koda, za izgradnju mnogo različitih aplikacija poput:

- web-aplikacija, WEB API-ja i mikroservisa
- bezserverske funkcije u oblaku
- native aplikacije u oblaku
- mobilne aplikacije
- desktop aplikacije
 - Windows WPF
 - Windows Forms
 - Universal Windows platform (UWP)

- igre
- IoT (Internet of Things)
- strojno učenje
- konzolne aplikacije
- Windows servisi
- ...

Dijeljena funkcionalnost između različitih aplikacija i tipova aplikacija dobiva se korištenjem klasnih biblioteka.

S .NET-om kod i projektne datoteke izgledaju kao da su iste neovisno o tome koju vrstu aplikacije radimo. Imamo pristup istom runtimeu, API-ju i jezičnim mogućnostima sa svakom aplikacijom.

S .NET-om možemo napraviti aplikacije za mnoge operativne sustave uključujući: Windows, macOS, Linux, Android, iOS, tvOS i watchOS.

Podržane su sljedeće procesorske arhitekture: x64, x86, ARM32 i ARM64.

.NET nam pruža pristup mogućnostima specifičnima za odabranu platformu, poput API-ja operativnog sustava. Primjer su Windows forme i WPF na Windowsu i nativno povezivanje na svaku mobilnu platformu korištenjem Xamarina.

Razvojni okvir .NET-a dolazi u nekoliko formalnih implementacija. .NET 5+ i .NET Core najnovije su implementacije i omogućuju izvođenje na bilo kojoj platformi. .NET framework originalna je implementacija .NET-a koja se izvodi samo na Windows operativnom sustavu. Mono se koristi kad nam je potreban mali runtime. UWP se koristi na modernim Windows aplikacijama (tablet, mobitel, desktop).

Svaka implementacija uključuje izvršno okruženje i biblioteku klasa. Može sadržavati i aplikacijski radni okvir i alate za razvoj.

.NET standard nije implementacija .NET-a, nego API specifikacija koja nam omo-

gućuje da razvijamo klasne biblioteke za višestruke implementacije .NET-a.

U potpunosti .NET podržava tri jezika: C#, F# i Visual Basic.

Iako možemo razvijati .NET aplikacije i s drugim jezicima (poput managed C++), nisu podržane sve funkcionalnosti kao s prethodno navedena tri jezika.

Razvojna okruženja za rad s .NET-om razvijena od strane Microsofta jesu:

- Visual Studio
- Visual Studio Code
- Visual Studio za Mac
- GitHub Codespace (online verzija Visual Studia, u trenutku pisanja u beta fazi).

Naglasili smo da su ovo razvojna okruženja razvijena od strane Microsofta. Drugi razvojni alati isto mogu pružiti mogućnost rada, odnosno razvoja, samo moguće sa smanjenim i/ili nepotpunim mogućnostima.

.NET SDK set je biblioteka i alata za razvoj i pokretanje .NET aplikacija. Kad skinete .NET, možete izabrati koju verziju SDK-a želite ili koju izvršnu verziju želite, poput .NET *runtimea* ili ASP.NET Core *runtimea*.

Instaliranjem *runtimea* radite izvršno okruženje za program na uređaju koji želite pripremiti za izvršavanje, odnosno rad vaših .NET aplikacija.

SDK instalirate na uređaj koji želite koristiti za razvoj .NET aplikacija. Kad skinete i instalirate SDK, po automatizmu instalira se i runtime okruženje. S njime instalirate sljedeće komponente:

- .NET CLI – alati za rad preko komandne linije
- „dotnet” pogonski program – CLI komanda koja pokreće aplikacije ovisno o *frameworku*
- Roslyn i F# kompajlere

- MSBuild pogonski sustav za izgradnju programa
- .NET *runtime*
- *runtime* biblioteke
- ASP.NET Core *runtime*
- desktop *runtime*

.NET aplikacija gradi se iz izvornog koda korištenjem MSBuild alata. Projektna datoteka specificira koje su ciljane biblioteke i pridruženi zadaci zaslužni za kompajliranje, pakiranje i objavu koda. Postoje identifikatori SDK-a koji referiraju standardne kolekcije ciljeva i zadataka. Korištenje tih identifikatora pomaže da nam projektne datoteke budu male i lagane za rad. Projektu datoteku možemo vidjeti tako da desno kliknemo u Solution Exploreru na projekt i kliknemo opciju „Edit Project file”. Naravno, grafički možemo sve dobiti ako kliknemo „Properties”.

Primjer za konzolnu aplikaciju:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Primjer za web-aplikaciju:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
</Project>
```

SDK atribut specificira koji set MSBuild ciljeva i zadataka trebamo da bismo izgradili projekt. TargetFramework element specificira koju ćemo verziju .NET-a koristiti. Također možemo dodavati dodatne zadatke projektu, no iskustveno, rijetko ćete kad to raditi.

MSBuild i .NET CLI mogu se koristiti i u kombinaciji s različitim alatima i okolinama za kontinuiranu integraciju poput GitHubActions, Azure DevOps, CAKE za C# itd. Ovo su neki od alata koji pomažu u slučaju DevOps specifičnih zadataka za kontinuiranu integraciju osoba koje programiraju, ali i za distribuciju završenih dijelova.

NuGet svakako moramo spomenuti kao upravitelj paketa otvorenog koda dizajniran za .NET. NuGet paket je .zip datoteka s .nupkg ekstenzijom koja sadrži kompajlirani kod u obliku biblioteke (.dll), raznih datoteka koje su u relaciji s kodom i opisni manifest koji uključuje informacije o samom paketu (poput verzije i slično). Programeri koji žele dijeliti svoj kod mogu ga tako pakirati i objaviti na nuget.org stranici ili nekom privatnom poslužitelju. Ukoliko programeri žele koristiti dijeljeni kod u svom projektu, dovoljno je napraviti API poziv prema paketu koji uključuje u svoj projekt.

.NET aplikacije izvođe „managed code” u runtime okruženju poznatom pod imenom Common Language Runtime ili skraćeno CLR. .NET CLR višeplatformski je *runtime* koji uključuje podršku za Windows, MacOS i Linux. CLR upravlja memorijom i njenom alokacijom. Također, CLR je virtualni stroj koji ne samo da izvršava aplikacije, nego generira i kompajlira kod koristeći *just-in-time* kompajler (JIT).

Programski jezici višeg nivoa poput C#-a kompajliraju se do skoro sklopovskog instrukcijskog seta, koji se zove „Intermediate Language” (IL). Kad se aplikacija pokrene, JIT kompajler prevodi IL u strojni jezik koji fizički procesor može razumjeti. JIT kompilacija događa se na istom stroju na kojem će se program izvršavati. Generalno, to je razlog zašto je C# sporiji jezik u usporedbi s C ili C++ programskim jezikom, no budući da se dobar dio kompajliranja već napravio prijevodom u IL, opet je C# uvelike brži i efikasniji od skriptnih jezika poput JS-a ili Pythona, koji se u potpunosti kompajliraju tokom izvođenja.

Budući da se JIT kompajliranje izvršava tijekom izvođenja aplikacije, vrijeme kom-

pajliranja dio je izvođenja. Time JIT kompajler mora balansirati vrijeme provedeno u optimizaciji koda naspram uštede vremena izvođenja kompajliranog koda. No za razliku od programera, JIT kompajler zna točan *hardware* na kojem će se program izvršavati, pa time oslobađa programera, koji ne mora raditi specifične implementacije za specifičnu platformu.

Jedna od najvažnijih stvari način je na koji se automatski upravlja memorijom. „Garbage collector” (GC) upravlja alokacijom i otpuštanjem memorije za aplikacije. Svaki put kada kod stvori novi objekt, CLR alocira memoriju za objekt na upravljanoj gomili (*managed heap*). Dokle god imamo na raspolaganju adresnog prostora u upravljanoj gomili, *runtime* će alocirati prostor za nove objekte. Kad ostanemo bez adresa koje možemo alocirati, GC provjerava koji se objekti više ne koriste od strane aplikacije i ako se ne koriste, otpušta tu memoriju. GC je jedan od servisa CLR-a koji pomažu u osiguravanju sigurne memorije. Program je memorijski siguran ako pristupa samo memoriji koja je njemu alocirana. Primjer sigurnosti memorije jest da program ne dozvoljava pristup memoriji koja je izvan dosega polja kad radimo s poljima.

Kad kod treba pristupiti ili referencirati neupravljane (*unmanaged*) resurse, koji nisu upravljani od strane .NET *runtimea*, objekt koji referencira taj resurs implementira `IDisposable` sučelje. Primjer je obična datoteka gdje je `FileStream` objekt upravljivi (*managed*) objekt, ali on referencira datoteku koja je neupravljivi objekt. Kad smo gotovi s korištenjem takvih objekata, moramo pozvati `Dispose()` metodu, koja je zadužena za oslobađanje neupravljivih resursa. U .NET-u imamo „using” naredbe kojima osiguravamo da se ta metoda uvijek na kraju pozove.

4.2. Rad s iznimkama

Tijekom programiranja imat ćemo slučajeve gdje se u kodu mogu dogoditi određene greške. Pod pojmom greške ne misli se na grešku u pisanju koda, takve vrste grešaka nam Visual Studio automatski prepoznaje na način da nam ne dozvoljava kompajlirati program. Naravno, VS nam ponekad nudi i način kako ispraviti te greške, a na programerima je da znaju što žele da kod radi, pa prema tome ispravljaju grešku.

Iznimku možemo definirati kao neplanirani događaj u programu. Pod neplaniranim događajem mislimo na nešto što poremećuje planirani tijek instrukcija koda, poput krivog korisničkog unosa, nemogućnosti da se neka vrijednost izračuna i slično. Iznimke mogu biti generirane od strane CLR-a i od strane drugog koda koji nije dio .NET-a.

„Nulla regula sine exceptione” ili „Nijedno pravilo bez izuzetaka” dobro naglašava kako da radimo s iznimkama. U prenesenom značenju, dok pišemo kod, pišemo pravila kako se program treba ponašati, a implementacijom rada s iznimkama u biti implementiramo što program treba napraviti za pojedine izuzetke od tih pravila.

Kad se iznimka dogodi, mi je možemo u kodu presresti i odrediti što napraviti u tim slučajevima. Za to koristimo ključne riječi **„try”**, **„catch”** i **„finally”**.

U mnogo slučajeva iznimka može biti bačena od strane metode koju naš kod nije direktno pozvao, nego od neke druge metode koju poziva kod koji smo mi pozvali. Kad se iznimka dogodi, CLR će provjeriti u pozivnom stogu koja se iznimka dogodila te sukladno tome tražiti „catch” blok za iznimku tog tipa pa izvršiti prvi blok koji odgovara tipu iznimke. Ukoliko se iznimka dogodi, a nema odgovarajućeg catch bloka za taj tip ili to uopće nije kod koji je generirao iznimku u try bloku, proces izvođenja programa terminira se i prikazuje se poruka korisniku da program ne može nastaviti s izvođenjem.

U C# programskom jeziku iznimke imaju sljedeća svojstva:

- iznimke su tipovi koji su u hijerarhiji nasljeđivanja derivirane od System.Exception klase
- koristimo „try” blok oko naredbi za koje mislimo da će uzrokovati iznimku
- kad se iznimka dogodi u try bloku, tok programa vodi nas do prvog rukovatelja iznimkama prisutnog unutar pozivnog stoga, koji definiramo preko „catch” ključne riječi
- ukoliko rukovatelj iznimke nije prisutan za iznimku koja se dogodila, program zaustavlja izvođenje s porukom greške korisniku
- iznimke ne vrijedi hvatati ukoliko se ne možemo oporaviti od njih i ostaviti aplikaciju u poznatom stanju. Ukoliko ne možemo obraditi iznimku, dobro

je proslijediti je dalje korištenjem „throw” ključne riječi

- ukoliko catch blok ima varijablu iznimke, preko iste možemo dobiti detalje iznimke
- u klasama možemo raditi vlastite iznimke korištenjem „throw” ključne riječi
- objekti iznimke sadrže detaljne informacije poput stanja pozivnog stoga i tekst opisa iznimke
- blok „finally” izvršit će se neovisno o tome je li se dogodila iznimka. Koristimo ga često za oslobađanje resursa otvorenih u try bloku
- upravljane iznimke (managed exceptions u C/C++) u .NET-u implementirane su na Win32 strukturama mehanizma za upravljanje iznimaka.

4.2.1. Upravljanje iznimaka

Kako smo već rekli, „**try**” blok koristi se unutar C# programskog jezika da bismo odijelili kod koji može biti pogođen nekom iznimkom. Uz try blok moramo asociirati jedan ili više „**catch**” blokova, koji se koriste da izvedu naredbe ako se određena iznimka dogodi. Blok „**finally**” opcionalan je i sadrži kod koji će se izvršiti neovisno o tome je li se iznimka dogodila.

Primjeri blokova:

```
try
{
    // Kod koji treba pokušati izvesti
}
catch (NekiException ex)
{
    // Kod za rukovanje iznimkom ide ovdje
    // Samo hvatajte iznimke od kojih se možete oporaviti
}
```

```

try
{
    // Kod koji treba pokušati izvesti
}
finally
{
    // Kod koji se izvršava nakon try bloka
}

```

```

try
{
    // Kod koji treba pokušati izvesti
}
catch (NekiException ex)
{
    // Kod za rukovanje iznimkom
}
finally
{
    // NekiException Code za izvođenje nakon try (i mogućeg catch) bloka
}

```

Try blok bez catch ili finally bloka uzrokuje grešku tijekom kompajliranja. Tijekom pisanja catch bloka opcionalno je imati zagrade s hvatanjem objekta iznimke, tako da možemo reći da je najjednostavniji oblik sljedeći:

```

try
{
    // Kod koji treba pokušati izvesti
}
catch
{
    // Kod za rukovanje iznimkom
}

```

Catch blok može točno specificirati tip iznimke koju želimo uhvatiti. Tip iznimke, kako smo već rekli, mora biti deriviran iz `System.Exception` klase (bilo direktno ili višestrukim nasljeđivanjem). Generalno, ne navodimo vrstu iznimke ako ne znamo kako se oporaviti od iznimke bačene unutar `try` bloka ili ako nismo izveli `throw` naredbu na kraju bloka.

Višestruki catch blokovi s različitim iznimkama mogu se ulančati tijekom pisanja koda. Catch blokovi evaluiraju se od prvog napisanog u kodu, ali samo jedan catch blok može biti izvršen. Izvršava se prvi blok koji specificira točan tip iznimke ili bazni tip iznimke. Ukoliko nema catch bloka koji specificira odgovarajuću iznimku, izvršava se catch blok koji nema nijedan tip iznimke ili ima baznu iznimku. Zbog toga nam je važno prvo navesti specifične iznimke, a tek onda generalne.

Primjer rada s višestrukim iznimkama gdje pokrивamo više slučajeva rada s datotekom:

```
public static void Main()
{
    try
    {
        using (var sw = new StreamWriter("./test.txt"))
        {
            sw.WriteLine("Pozdrav");
        }
    }
    // Prvo specifične iznimke
    catch (DirectoryNotFoundException ex)
    {
        Console.WriteLine(ex);
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine(ex);
    }
    // Najmanje specifične iznimke na kraju
    catch (IOException ex)
    {
        Console.WriteLine(ex);
    }
    Console.WriteLine("Gotovo");
}
```

Iznimke hvatamo kad je neki od uvjeta ispunjen.

- Kad razumijemo zašto se iznimka dogodila i kad možemo implementirati specifični oporavak od iznimke, poput govora korisniku da mora unijeti novo ime datoteke ako specificirana datoteka ne postoji (FileNotFoundException). Primjer:

```
string datoteka = Console.ReadLine();
try {
    StreamReader sr=new StreamReader(datoteka);
}
catch(FileNotFoundException ex)
{
    Console.WriteLine(ex.Message);
}
```

- Kad možemo stvoriti i baciti novu, specifičniju iznimku. Primjer kad pokušavamo čitati elemente izvan definiranog polja:

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parametar index je van dometa!", e);
    }
}
```

- Kad želimo djelomično obraditi iznimku prije nego što je proslijedimo za daljnju obradu. U sljedećem se primjeru catch blok koristi za dodavanje zapisa na errorLog prije proslijeđivanja iznimke:


```

try
{
    // Pokušaj pristupa resursima
}
catch (UnauthorizedAccessException e)
{
    // Poziv proceduri za logiranje greške
    LogError(e);
    // Ponovno bacanje iznimke
    throw;
}

```

Tijekom rada s iznimkama možemo dodavati i filtere iznimke, koji moraju biti evaluirani kao logički podatkovni tip, koji odgovaraju catch bloku. Filteri iznimaka služe nam slično kao if naredbe, no prilikom odabira treba li taj blok izvršiti ili ne, uz tip iznimke i uvjet mora biti istinit. Prethodni primjer proširit ćemo provjerom parametra:

```

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index ne može biti negativan!", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index ne može biti veći od veličine polja!", e);
    }
}

```

U praksi možemo koristiti filtere koji uvijek vraćaju false da presretnemo iznimke, ali ih ne obradimo. To tipično koristimo da bismo logirali iznimke kao u sljedećem primjeru:

```
public static void Main()
{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Iznimka se obradila");
}
private static bool LogException(Exception e)
{
    Console.WriteLine($"Unutar log metode. Uhvaćena {e.GetType()}");
    Console.WriteLine($"Poruka: {e.Message}");
    return false;
}
```

LogException metoda uvijek vraća false, nema catch bloka koji može odgovarati ovom filteru. Catch blok može biti generalnog tipa i kasniji blokovi mogu procesirati sa specifičnijom iznimkom iznimku koja se dogodila.

Blok finally omogućuje nam da počistimo sve akcije koje su se dogodile unutar try bloka. Sam finally blok opcionalan je i ako postoji, on se izvršava zadnji nakon try bloka i nakon catch bloka ako je nađen za odgovarajuću iznimku.

Blok finally može se koristiti za oslobađanje resursa poput datoteka, veza na bazu podataka i drugih resursa. U sljedećem primjeru koristit ćemo finally blok za zatvaranje datoteke koja je otvorena u try bloku. Prije zatvaranja svakako provjeravamo stanje objekta.

```

FileStream? datoteka = null;
FileInfo fileinfo = new System.IO.FileInfo("./datoteka.txt");
try
{
    datoteka = fileinfo.OpenWrite();
    datoteka.WriteByte(0xF);
}
finally
{
    // Provjeri null jer OpenWrite moguće nije uspio.
    datoteka?.Close();
}

```

4.2.2. Korištenje vlastitih iznimaka

Iznimke u C#-u derivirane su iz Exception klase. Ova klasa identificira tip iznimke i sadrži svojstva koja imaju detalje same iznimke. Bacanje iznimke uključuje stvaranje instance derivirane klase iznimke s opcionalnim parametrima te bacanje objekta korištenjem „throw” ključne riječi. Primjer rada:

```

class Mojalznimka : Exception
{
    public Mojalznimka(string poruka)
    {
    }
}
private static void TestThrow()
{
    throw new Mojalznimka("Testiranje iznimke u TestThrow()");
}

```

Programeri bi trebali bacati iznimke kad je jedan od sljedećih uvjeta istinit:

- metoda ne može izvršiti definiranu funkcionalnost, npr. ako metoda dobije neadekvatan parametar
- napravio se neprimjeren poziv objekta. Pod neprimjeren mislimo da s obzirom na stanje objekta, npr. kad pokušavamo pisati u datoteku koja je markirana da je samo za čitanje
- kad argument prilikom poziva metode uzrokuje iznimku. U ovom slučaju originalna iznimka trebala bi se uhvatiti i proslijediti. Primjer nam može biti kad pokušavamo pristupiti elementu izvan polja, gdje originalnu iznimku proslijeđujemo kao unutarnju iznimku:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index van dosega",
            nameof(index), ex);
    }
}
```

Iznimke imaju svojstvo „StackTree”, koje je string koji sadržava ime metode na trenutnom pozivnom stogu, zajedno s datotekom, imenom i brojem linije gdje se iznimka dogodila za svaku metodu. StackTree objekt stvara se automatski od strane CLR-a tijekom izvršavanja throw naredbe, tako da iznimka može biti bačena od točke gdje bi početak stabla stoga trebao biti.

Sve iznimke imaju svojstvo „Message”, odnosno poruka. Ovaj string trebao bi sadržavati opis razloga iznimke. Informacije koje su osjetljive prirode nikako se ne smiju stavljati u poruku. Dodatno uz „Message”, ArgumentException sadržava svojstvo „ParamName”, koje bi trebalo biti postavljeno na ime argumenta koji je uzrokovao iznimku.

Praktične stvari koje treba izbjegavati:

- nemojte koristiti iznimke za promjenu toka programa kao dio uobičajenog izvršavanja. Iznimke koristite samo za logiranje i obradu iznimki
- iznimke ne bi trebale biti korištene da vrate neke vrijednosti drugom dijelu programa
- nemojte bacati `System.Exception`, `System.SystemException`, `System.NullReferenceException` ili `System.IndexOutOfRangeException` iz svog koda, te će se iznimke često automatski bacati ako dođe do njih
- nemojte stvarati iznimke koje se mogu bacati u „debug” modu rada, ali ne i u „release” modu rada. Da biste našli greške tijekom razvoja programa, koristite „Debug Assert”.

4.3. Rad sa stringovima

String smo već mnogo puta spominjali, sad je napokon vrijeme da detaljno opišemo kako raditi s njim.

String je objekt tipa `string`, čija je vrijednost tekst. Interno, tekst je spremljen kao sekvenca objekata znakova (`char`) unutar kolekcije koja se može samo čitati. Nema takozvanog null karaktera na kraju C# stringa, tako da C# string može sadržavati bilo koji broj ugrađenih null karaktera (`'\0'`). Svojstvo „length” unutar stringa predstavlja broj znakova koje objekt sadrži.

Svaka varijabla tipa `string` posjeduje metode i svojstva klase `string` koji opisuju i obrađuju podatke tekstualnog tipa. Samu deklaraciju i inicijalizaciju stringa možemo napraviti na više načina, kao u sljedećim primjerima:

// Deklaracija bez inicijalizacije.

string poruka1;

// Inicijalizacija na null.

string poruka2 = **null**;

```

// Inicijalizacija kao prazan string
// Koristite Empty konstantu umjesto literala "".
string poruka3 = System.String.Empty;

// Inicijalizacija string literalom.
string staraPutanja = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Inicijalizacija s doslovnim string literalom.
string NovaPutanja = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Možemo koristiti System.String
System.String pozdrav = "Pozdrav svijete!";

// Unutar lokalne varijable implicitno
var temp = "Još uvijek jako tipizirani System.String!";

// Korištenjem const string da se poruka4 ne bi mogla mijenjati
const string poruka4 = "Ne možete me mijenjati!";

// Korištenje String konstruktora samo kad stvaramo
// string pomoću char*, char[] ili sbyte*
char[] slova = { 'A', 'B', 'C' };
string abeceda = new string(slova);

```

Operator `new` ne koristimo često za stvaranje objekta tipa `string`, osim kad inicijaliziramo `string` pomoću polja znakova. Inicijalizacija `stringa` pomoću `Empty` konstantne vrijednosti radimo da bismo stvorili novi `string` čija je duljina nula. Inicijalizacijom `stringa` kao praznog smanjujemo šansu da se dogodi `NullReferenceException` prilikom rada. Za provjeru `stringa` prilikom pristupa koristimo statičnu metodu `IsNullOrEmpty(String)` da provjerimo postojanje vrijednosti `stringa` prije nego što joj pristupamo.

Objekti tipa `string` su „immutable”, što znači da se ne mogu promijeniti nakon što su stvoreni. Sve metode `string` klase i C# operatori koji naizgled modificiraju `string` u stvarnosti vraćaju kao rezultat novi objekt tipa `string`. U sljedećem primjeru, kad konkatenuiramo (spajamo) sadržaj `s1` i `s2`, originalni su `stringovi` nepromijenjeni. Operator `+=` stvara novi `string` koji sadrži kombinirani sadržaj. Taj novi objekt dodjeljuje se varijabli

s1, a originalni objekt koji je bio dodijeljen s1 otpušta se da ga garbage collector pokupi i oslobodi, jer nema nijednog drugog objekta koji sadrži njegovu referencu.

```
string s1 = "String je više ";
string s2 = "nego suma njegovih znakova";

s1 += s2;

System.Console.WriteLine(s1);
// Izlaz: String je više nego suma njegovih znakova
```

Moramo biti jako oprezni prilikom stvaranja referenci na objekte tipa string. Ako stvorimo referencu na takav objekt i radimo modifikacije, originalna referenca ostaje i nastavlja pokazivati na originalni string, a ne na novonastali operacijama koje smo izvršili. Primjer koji to demonstrira:

```
string str1 = "Pozdrav";
string str2 = str1;
str1 += "Bok";
System.Console.WriteLine(str2);
//Izlaz: Pozdrav
```

4.3.1. String literali

Quoted string literals ili *citirani string literali* počinju i završavaju s duplim navodnicima unutar iste linije. Najbolje ih je koristiti za stringove koji stanu u jednu liniju i ne sadrže nikakve izlazne sekvence. Ukoliko moramo imati izlazne sekvence, moramo ih ugraditi u izlazne znakove kao u sljedećem primjeru:

```
string stupci = "Stupac 1\tStupac 2\tStupac 3";
//Izlaz: Stupac 1    Stupac 2    Stupac 3
string redovi = "Red 1\r\nRed 2\r\nRed 3";
/* Izlaz:
    Red 1
    Red 2
    Red 3
*/
```

Verbatim string literals ili *doslovan string literal* konvencionalniji je za rad stringova u više redova koji sadržavaju kose crte, duple navodnike itd. Kod njih dobijemo tekst točno kako vidimo (what you see is what you get), jedino moramo unutar stringa navesti dvaput dupli navodnik da bismo dobili duple navodnike. Sljedeći primjeri pokazuju doslovne string literale:

```
string putanja = @"C:\Users\Administrator\Documents\";  
//Output: C:\Users\Administrator\Documents\
```

```
string text = @"Neki tekst koji se može pisati  
u više redova, ali da prilikom ispisa  
formatiranje ostane kao u samom kodu";  
/* Izlaz:  
Neki tekst koji se može pisati  
u više redova, ali da prilikom ispisa  
formatiranje ostane kao u samom kodu  
*/
```

```
string quote = @"Njeno ime je ""Ana.""";  
//Output: Njeno ime je "Ana."
```

S C# 11 standardnom moguće je koristiti sirove string literale (raw string literals) da bismo jednostavnije stvorili stringove koji su više linijski ili koriste znakove koji imaju izlazne sekvence. Sirovi string literali odstranjuju potrebu da koristimo izlazne sekvence. Sirovi string literali imaju sljedeća svojstva:

- počinju i završavaju sekvencom od najmanje tri dvostruka navodnika ("""). Možemo staviti više od tri da bismo u kodu podržali string literale koji sa drže tri ili više uzastopnih navodnika
- jednolinijski sirovi string literali zahtijevaju otvarajuće i zatvarajuće znakove u istoj liniji
- višelinijski sirovi string literali zahtijevaju otvarajuće i zatvarajuće znakove na svojoj svojstvenoj liniji
- u višelinijskim literalima svaki razmak lijevo od zatvarajućih znakova uklanja se.

Sljedeći primjeri demonstriraju pravila:

```
string jednaLinija = ""Prijatelji kažu "Pozdrav" kad prolaze.""";
string viseLinija = ""
    "Pozdrav Svijete!" tipičan primjer prvog programa.
    "";
string embeddedXML = ""
    <element attr = "content">
        <body style="normal">
            Ovdje ide glavni tekst
        </body>
        <footer>
            Podnožje unesite ovdje
        </footer>
    </element >
    "";
// Linija "<element attr = "content">" počinje u prvom stupcu.
```

```
string rawStringLiteralDelimiter = """"
    Sirovi string literali su razdvojeni
    pomoću stringa od bar 3 dvostruka navodnika,
    poput ovih: """"
    """";
```

Sljedeći primjeri pokazuju greške tijekom prevođenja bazirane na prethodnim pravilima:

// CS8997: Unterminated raw string literal.

```
var multiLineStart = ""Ovo
    je početak stringa
    "";
```

// CS9000: Raw string literal delimiter must be on its own line.

```
var multiLineEnd = ""
    Ovo je početak stringa "";
```

```
// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
```

```
var noOutdenting = ""
```

```
    Linija teksta.
```

```
Pokušaj uvlake druge linije.
```

```
    "";
```

Prva dva primjera imaju grešku zbog potrebe da otvarajuća i zatvarajuća sekvenca budu u istom redu. Treći primjer je greška jer imamo uvlaku kod zatvarajuće sekvence.

Ovakva način pisanja stringova dobar je kad generiramo tekst koji sadrži znakove koji trebaju izlazne sekvence kad koristimo citirane stringove. Također, sirovi string literali lakši su za čitanje od ostalih jer su sličniji izlaznom tekstu. Primjer lakoće čitanja tekst je koji je formatiran u JSON formatu:

```
string jsonString = ""
```

```
{
```

```
    "Date": "2019-08-01T00:00:00-07:00",
```

```
    "TemperatureCelsius": 25,
```

```
    "Summary": "Hot",
```

```
    "DatesAvailable": [
```

```
        "2019-08-01T00:00:00-07:00",
```

```
        "2019-08-02T00:00:00-07:00"
```

```
    ],
```

```
    "TemperatureRanges": {
```

```
        "Cold": {
```

```
            "High": 20,
```

```
            "Low": -10
```

```
        },
```

```
        "Hot": {
```

```
            "High": 60,
```

```
            "Low": 20
```

```
        }
```

```
    },
```

```
    "SummaryWords": [
```

```
        "Cool",
```

```
        "Windy",
```

```
        "Humid"
```

```
    ]
```

```
}
```

```
    "";
```

4.3.2. String izlazne sekvence

Već smo nekoliko puta spominjali izlazne sekvence. To su znakovi koje ne možemo uvijek prikazati prilikom ispisa, no trebamo ih spremati u tekstualnom obliku. Sljedeća tablica pokazuje nam neke od češćih C# izlaznih sekvenci:

Izlazna sekvenca	Ime znaka	Unicode enkodiranje
\'	Jednostruki navodnik	0x0027
\“	Dvostruki navodnik	0x0022
\\	Backslash (obrnuta kosa crta)	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line (novi red)	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab (vodoravni tabulator)	0x0009
\v	Vertical tab (okomiti tabulator)	0x000B

Prilikom kompajliranja doslovan string svakako se pretvara u običan string, sa svim izlaznim sekvencama. Zato ćemo tijekom pregleda stringova u debuggeru vidjeti obične stringove s izlaznim sekvencama. Npr. doslovni string @"C:\datoteka.txt" prikazuje se kao "C\\:datoteka.txt".

4.3.3. Format stringova

Format stringova je string čiji se sadržaj određuje dinamički tijekom izvođenja programa. Formatirani stringovi stvaraju se ubacivanjem interpoliranih izraza ili držača mjesta (*placeholders*) unutar zagrada u stringu. Sve unutar zagrada {...} bit će razriješeno u vrijednost i izaći će kao formatirani string tijekom izvođenja programa. Postoje dvije metode za stvaranje formatiranog stringa: interpolacijsko formatiranje i kompozitno formatiranje.

Interpolacijsko formatiranje, uvedeno s C# verzijom 6, identificira se preko \$ specijalnog znaka i uključuje interpolirane izraze unutar vitičastih zagrada. S interpolacijom stringova poboljšava se čitkost i održavanje koda. Daje isti rezultat kao String.Format metoda, ali istovremeno olakšava jednostavnost korištenja i čistoću linija.

```
var jh = (ime: "Jupiter", prezime: "Hammon", godRodjenja: 1711, izdano: 1761);
Console.WriteLine($"jh.ime} {jh.prezime} bio je Afroamerički pjesnik rođen {jh.
godRodjenja}.");
Console.WriteLine($"Prvu publikaciju imao je {jh.izdano} u svojoj {jh.izdano - jh.
godRodjenja}. godini");
Console.WriteLine($"Bio bi preko {Math.Round((2022d - jh.godRodjenja) / 100d) *
100d} godina star danas.");

// Izlaz:
// Jupiter Hammon bio je Afroamerički pjesnik rođen 1711.
// Prvu publikaciju imao je 1761 u svojoj 50. godini
// Bio bi preko 300 godina star danas.
```

S verzijom C# 10 moglo se koristiti interpolacijsko formatiranje za inicijalizaciju konstantnog stringa kada su svi izrazi korišteni za držače mjesta također konstantni stringovi.

S verzijom C# 11 moglo se kombinirati sirove string literale sa string interpolacijama. Počeli bismo i završavali format stringa s tri ili više dupla navodnika, no ako string sadrži { ili } znak, može se koristiti ekstra \$ znak da bi se specificiralo koliko { i } znakova započinje i završava interpolaciju. Svaka sekvenca od manje { ili } znakova uključena je u izlazu. Sljedeći primjeri pokazuju kako koristiti ovu mogućnost za prikazivanje udaljenosti točaka od ishodišta i mjesta do točke unutar zagrada:

```
int X = 2;
int Y = 3;

var pointMessage = $$$""Točka {{X}}, {{Y}} udaljena je {{Math.Sqrt(X * X + Y * Y)}}
od ishodišta.""";
Console.WriteLine(pointMessage);

// Izlaz:
// Točka {2, 3} udaljena je 3.605551275463989 od ishodišta.
```

Možemo reći da je kompozitni format stringova klasičniji prikaz stringova. Koristi `String.Format` metodu, koja koristi držače mjesta da bi stvorila format. Sljedeći primjer prikazuje prethodni primjer korištenjem kompozitnog formatiranja:

```
var pw = (ime: "Jupiter", prezime: "Hammon", godRodjenja: 1711, izdano: 1761);
Console.WriteLine("{0} {1} bio je Afroamerički pjesnik rođen {2}.", pw.ime,
pw.prezime, pw.godRodjenja);
Console.WriteLine("Prvu publikaciju imao je {0} u svojoj {1}. godini.", pw.published, pw.izdano - pw.godRodjenja);
Console.WriteLine("Bio bi preko {0} godina star danas.", Math.Round((2022d - pw.godRodjenja) / 100d) * 100d);
// Izlaz:
// Jupiter Hammon je bio Afro Američki pjesnik rođen 1711.
// Prvu publikaciju je imao 1761 u svojoj 50. godini
// Bio bi preko 300 godina star danas.
```

4.3.4. Pristup individualnim znakovima

Unutar stringa možemo koristiti sintaksu kao kad radimo s poljima da bismo pristupili individualnim znakovima unutar stringa. Samo moramo biti svjesni da ne možemo mijenjati vrijednost nijednog znaka, jer kako smo rekli, svaki znak samo je za čitanje, nema pisanja! Slijedi primjer:

```
string s = " Pisanje unatrag";
for (int i = 0; i < s.Length; i++)
{
    System.Console.Write(s[s.Length - i - 1]);
}
// Izlaz: "gartanu ejnasiP"
```

Ukoliko string metode ne pružaju funkcionalnost modificiranja individualnih znakova, možemo koristiti `StringBuilder` objekt da modificiramo individualne znakove i onda stvorimo novi string za pohranu rezultata. U sljedećem primjeru modificiramo originalan string na određeni način i spremamo rezultat za buduće korištenje:

```
string pitanje = "KAKO ćemo RADITI s nepravILNIM koriŠTENJEM caps LOCK  
TIPke?";
```

```
System.Text.StringBuilder sb = new System.Text.StringBuilder(pitanje);
```

```
for (int j = 0; j < sb.Length; j++)
```

```
{
```

```
    sb[j] = System.Char.ToLower(sb[j]);
```

```
}
```

```
// Store the new string.
```

```
string ispravljeno = sb.ToString();
```

```
System.Console.WriteLine(ispravljeno);
```

```
// Output: kako ćemo raditi s nepravilnim korištenjem caps lock tipke?
```

4.3.5. Null stringovi i prazni stringovi

Prazni string instanca je `System.String` objekta koja sadrži nula znakova, odnosno ne sadrži nijedan znak. Prazni stringovi često se koriste da predstavljaju prazno polje za unos teksta, ali moraju biti inicijalizirani zbog mogućnosti poziva metoda. Prazni string možemo inicijalizirati na sljedeći način:

```
string s = String.Empty;
```

Za razliku od praznog stringa, null string nema nikakvu referencu na instancu `System.String` objekta te će bilo koji pokušaj poziva bilo koje metode ili svojstva uzrokovati `NullReferenceException`. Null stringove možemo koristiti prilikom konkatencije i uspoređivanja s ostalim stringovima. Sljedeći primjeri pokazuju neke slučajeve u kojima referenciranje null stringova radi i ne radi iznimke:

```
string str = "pozdrav";
```

```
string nullStr = null;
```

```
string prazanStr = String.Empty;
```

```
string tempStr = str + nullStr;
```

```
// Izlaz sljedeće linije: pozdrav
```

```
Console.WriteLine(tempStr);
```

```

bool b = (prazanStr == nullStr);
// Izlaz sljedeće linije: False
Console.WriteLine(b);

// Sljedeća linija stvara novi prazan string
string noviStr = prazanStr + nullStr;

// Null stringovi i prazni stringovi drukčije se ponašaju.
// Sljedeće dvije linije ispisuju 0
Console.WriteLine(prazanStr.Length);
Console.WriteLine(noviStr.Length);
// Sljedeća linija baca NullReferenceException.
//Console.WriteLine(nullStr.Length);

// null znakovi mogu se prikazati i pobrojati, kao i ostali znakovi.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Izlaz sljedeće linije: * abc*
Console.WriteLine(" *" + s1 + " *");
// Izlaz sljedeće linije: *abc *
Console.WriteLine(" *" + s2 + " *");
// Izlaz sljedeće linije: 4
Console.WriteLine(s2.Length);

```

4.3.6. stringBuilder klasa

Operacije sa stringovima u .NET-u visoko su optimizirane i u većini slučajeva ne stvaraju neki osjetni udar na performanse. No u nekim slučajevima, kad se string operacije moraju odvijati i nekoliko stotina tisuća puta u sekundi, mogu uzrokovati osjetni pad performansi.

Klasa `StringBuilder` kreira spremnik koji pruža bolje performanse tijekom rada programa sa stringovima. Također nam omogućuje da mijenjamo individualne znakove, koje nam ugrađena string klasa ne dopušta. Sljedeći primjer koda mijenja sadržaj stringa bez da kreira novi string:

```

System.Text.StringBuilder sb = new System.Text.StringBuilder("Zmija: idealni lju-
bimac");
sb[4] = 'e';
sb[sb.Length - 1] = 'i';
sb[sb.Length - 2] = 'c';
System.Console.WriteLine(sb.ToString());
// Izlaz Zmije: idealni ljubimci

```

U ovom primjeru `StringBuilder` koristimo da stvorimo objekt tipa `string` iz seta brojeva:

```

var sb = new StringBuilder();
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb); // prikazuje 0123456789

// kopiranje jednog znaka stringa (nije moguće sa System.String)
sb[0] = sb[9];
Console.WriteLine(sb); // prikazuje 9123456789

```

4.3.7. Operatori i stringovi

Prije smo spominjali operatore, no svakako ih moramo spomenuti opet. Nažalost, nad stringovima svi operatori nisu definirani, odnosno ne možemo sve poznate operacije koristiti nad `System.String` objektima.

Ako malo razmotrimo osnovne operatore, doći ćemo do nekoliko zaključaka. Logički operatori nemaju nikakvog smisla kad radimo sa stringovima. Bilo kakav pokušaj rada logičkog operatora sa stringovima dovest će do toga da nam kompajler izbacuje grešku (CS0023) da se operator ne može koristiti s navedenim podatkovnim tipom. Često će se pokušati napraviti implicitna pretvorba, ukoliko je moguća, no ovdje je nemoguće tekst pretvoriti u nešto što je logičkog podatkovnog tipa.

Kod relacijskih operatora nećemo imati sve operacije definirane. Ako malo logički razmislimo što ima smisla imati u radu s tekstom, doći ćemo do zaključka da samo dva operatora usporedbe možemo koristiti, a to su operator jednako (==) i operator različito (<> ili !=). Veće, manje, veće ili jednako i manje ili jednako nemaju smisla, jer kako usporediti? Da li po broju znakova, znakovima koju su navedeni ili nekom drugom kriteriju? No vratimo se na operator jednakosti. Dva stringa bit će jednaka ako su jednaka po duljini stringa i po znakovima koji su unutar stringova. Kad kažemo da znakovi moraju biti jednaki, pazite na to da C# razlikuje mala i velika slova. Primjer:

```
if ("zmija" == "zmija") Console.WriteLine("Obje su zmije!");  
if ("zmija" != "zmaj") Console.WriteLine("Nisu obje zmije!");  
if ("zmija" != "ZMIJA") Console.WriteLine("Obje su zmije, ali različite");
```

Najveća skupina operatora koje imamo, ali ovdje i jedna od kompliciranijih, skupina je aritmetičkih operatora. Jedan jednostavan razlog je tome jest to što nisu svi operatori definirani. Dosad smo najviše koristili u raznim primjerima + operator, koji se koristi za konkatenciju stringova, odnosno spajanje više stringova. Rezultat zbrajanja novi je objekt koji sadrži spojena oba stringa s lijeve i desne strane operatora. Ostali operatori jednostavno nisu definirani nad stringovima, ali kad bolje razmislimo, kako bismo ih definirali? Kako oduzeti dva stringa? Kako pomnožiti, podijeliti ili raditi ostatak prilikom dijeljenja dva stringa? Kako god pokušali, logički nije definirano, a samim time i pokušaj rada tih operacija rezultirat će greškom CS0019, odnosno porukom da operator koji koristimo ne može biti apliciran za te podatkovne tipove.

4.3.8. Metode i svojstva string klase

String klasa, kao što vidite, dosad je jedna od opširnijih klasa koje smo pokrili, no iz jednostavnog razloga što je i jedna od najkorištenijih u obradi podataka. Razlog je jednostavan, većina podataka s kojima radimo u tekstualnom su obliku, pa nam je rad s tekstom jedan od najvažnijih dijelova programiranja.

Kad spomenemo string klasu, spomenut ćemo samo da klasa sadrži devet različitih konstruktora, koje možemo koristiti ovisno kako želimo napraviti instanciranje objekta preko new operatora.

- `String(Char*)`
- `String(Char*, Int32, Int32)`
- `String(Char, Int32)`
- `String(Char[])`
- `String(Char[], Int32, Int32)`
- `String(ReadOnlySpan<Char>)`
- `String(SByte*)`
- `String(SByte*, Int32, Int32)`
- `String(SByte*, Int32, Int32, Encoding)`.

No kako smo već prije spomenuli, najčešće ne koristimo `new` operator da stvorimo `string` objekt. No opet nijedno pravilo nije bez izuzetaka, pa će nam tako ponekad biti nužno koristiti jedan od konstruktora.

Svojstva klasa `String` nema mnogo, ali ona koja postoje izrazito su nam korisna. Dva svojstva imamo u `string` klasi:

- `length` – daje nam duljinu stringa, odnosno koliko znakova sadrži
- `this[]` ili `Chars[]` – svojstvo za pristup pojedinim znakovima, definiran je samo `get` dio i zato možemo samo čitati, bez pisanja znakova u stringu.

Primjer za ispis duljine stringa i 5. znaka u stringu (4+1, jer brojanje kreće od 0):

```
string s = "Danas je lijep dan.";
Console.WriteLine(s.Length);
Console.WriteLine(s[4]);
```

```
// Izlaz:
// 19
// s
```

Ono što rad sa stringovima čini moćnim i nadasve korisnim brojne su metode unutar string klase. Uz standardne metode postoje i proširene. Previše bi prostora trebalo da ih sve nabrojimo, pa ćemo nabrojati samo neke najčešće:

- **ToLower** – vraća sva mala slova stringa
- **ToUpper** – vraća sva velika slova stringa
- **SubString** – izdvaja dio teksta iz string varijable
- **Split** – iz jedne string varijable vraća polje stringova tako da ih cijepa s obzirom na zadani separator
- **Clone** – vraća referencu na instancu varijable
- **Compare** – uspoređuje dva stringa s obzirom na zadane parametre
- **Concat** – konkatenira stringove, slično kao + operator, samo s dodatnim opcijama
- **Contains** – provjerava sadrži li string proslijeđeni znak ili string
- **Copy** – stvara novu instancu stringa koja sadrži identične znakove
- **EndsWith** – provjerava završava li string proslijeđenim znakom ili stringom
- **Format** – formatira string s obzirom na zadani format
- **IndexOf** – vraća indeks pojavljivanja znaka ili stringa s obzirom na proslijeđene parametre
- **Insert** – ubacuje novi string ili znak na poziciju proslijeđenu parametrom
- **IsNullOrEmpty** – provjerava je li objekt null ili prazan string
- **Join** – konkatenira stringove ili znakove s obzirom na proslijeđene parametre
- **PadLeft** – dodaje razmake s lijeve strane stringa
- **PadRight** – dodaje razmake s desne strane stringa
- **Remove** – uklanja znakove iz stringa ovisno o proslijeđenim parametrima
- **Replace** – mijenja jedan string ili znak drugim stringom ili znakom, ovisno o proslijeđenim parametrima
- **StartsWith** – provjerava počinje li string drugim stringom ili znakom
- **Trim** – uklanja na početku stringa znakove ovisno o proslijeđenim parametrima
- **TrimEnd** – uklanja na kraju stringa znakove ovisno o proslijeđenim parametrima
- ...

Primjeri rada nekih funkcija:

```
string s = "Danas je lijep dan.";
Console.WriteLine(s.ToLower()); //danas je lijep dan.
Console.WriteLine(s.ToUpper()); //DANAS JE LIJEP DAN.
Console.WriteLine(s.Substring(6,2)); //je
Console.WriteLine(s.Insert(9, "jako ")); //Danas je jako lijep dan.
Console.WriteLine(s.Remove(8)); //Danas je
Console.WriteLine(s.Replace("lijep", "prekrasan")); //Danas je prekrasan dan.
Console.WriteLine(s); //Danas je lijep dan
```

Svaka od navedenih metoda višestruko je preopterećena, tako da, slično kao s konstruktorima, imate više načina kako koristiti ove metode ovisno o vašim potrebama.

4.4. Rad s datotekama

Svaka aplikacija treba trajno spremiti neke podatke ili iz trajnog spremišta neke podatke pročitati. Ukoliko nema trajne pohrane ili čitanja podataka, svaki program uvijek bi se jednako pokretao i jednako radio, bez neke dugotrajne funkcionalnosti.

Prvo mali podsjetnik na pojmove.

Datoteka je skup podataka trajno pohranjen na neku jedinicu vanjske memorije. Kad kažemo vanjske memorije, podsjećamo da to može biti disk, USB stick, memorijska kartica, neki udaljeno mrežni disk itd. O kojem god načinu pohrane pričamo, svakako ćemo imati putanju prema mjestu pohrane.

Za potrebe rada s datotekama u .NET-u koristimo klase koje se nalaze u System.IO imenskom prostoru, koji uz klase sadrži još nekoliko često korištenih struktura, enumeracija i delegata.

4.4.1. Klase za čitanje i pisanje

Da bismo s datotekom mogli uopće raditi, moramo je otvoriti. Otvaranje datoteka radimo pomoću objekta `FileStream` na sljedeći način:

```
FileStream fs = new FileStream(putanja, FileMode);
```

Instanciramo objekt klase `file stream` koji otvara datoteku preko konstruktora sa sljedećim parametrima:

- **putanja** – putanja prema datoteci, podatkovnog tipa `string`
- **FileMode** – mod u kojem će raditi program s datotekom.

Enum tip koji može imati sljedeće opcije:

- **open** – datoteka će se koristiti za čitanje
- **create** – datoteka će se koristiti za pisanje. Ukoliko datoteka postoji, novi podaci pregazit će prijašnje
- **append** – datoteka će se koristiti za pisanje, s tim da se novi sadržaj nadopisuje iza prethodnog. Ako datoteka ne postoji, stvorit će se nova
- **truncate** – datoteka će se otvoriti za pisanje i sav sadržaj iz nje prije toga obrisat će se. Ako datoteka ne postoji, stvorit će se nova.

Svakako napomena, budite oprezni oko prava pristupa datotekama ili lokaciji gdje radite operacije čitanja, a pogotovo pisanja. Za te operacije vaš program mora imati dozvolu *UAC (User Account Control)* od strane operativnog sustava. Ukoliko nemate dozvolu, prilikom stvaranja objekta možete dobiti `UnauthorizedAccessException` ili `System.Security. SecurityException`. Zbog toga rad s datotekama uvijek stavljamo u `try` blokove jer su iznimke česta stvar kad radimo s datotekama.

Kad smo objekt klase `FileStream` instancirali, pomoću njega instanciramo objekte ovisno o radnji koju želimo raditi. Ukoliko želimo pisati, instancirat ćemo objekt klase `StreamWriter`, a ukoliko želimo čitati, instanciramo objekt klase `StreamReader`. To bismo napravili na sljedeći način:

```
StreamReader sr = new StreamReader(fs);  
StreamWriter sw = new StreamWriter(fs);
```

U kreiranom sw objektu klase StreamWriter za upis teksta u datoteku koristimo metode:

- Write
- WriteLine.

Obje se metode koriste kao metode unutar Console klase. Metoda write piše ono što proslijedimo u datoteku, u formatu koji navedemo. Jedina razlika između metoda Write i WriteLine jest u tome što metoda WriteLine na kraju zapisa uvijek dodaje novi red. Ovim metodama mi ne upisujemo podatke na disk, nego samo pripremamo za upis u međuspremniku. Pravi upis na disk radi se metodom flush na sljedeći način:

```
sw.Flush();
```

Prije poziva metode flush, da ste pokušali otvoriti datoteku koja je kreirana, vidjeli biste da je prazna, odnosno da je veličine 0 bajtova i nema ničega u njoj, koliko god mi puta pozivali Write ili WriteLine funkcije. Tek se pozivom Flush svi podaci koji se nalaze u privremenom spremniku spremaju na vanjsku memoriju, odnosno tom metodom signaliziramo operativnom sustavu da treba trajno spremati podatke.

Objekte klase StreamReader možemo čitati na nekoliko načina. Najjednostavnija i najdirektnija metoda jest metoda ReadToEnd, koja čita cijelu datoteku i njezin sadržaj vraća kao string objekt. To bismo radili na sljedeći način:

```
string podaci = sr.ReadToEnd();
```

Osim metode ReadToEnd u StreamReader klasi postoji nekoliko metoda za čitanje poput:

- Read – čita sljedeći znak u datoteci
- Readblock – čita blok podataka iz datoteke sukladno proslijeđenim parametrima
- ReadLine – čita liniju podataka do prvog znaka za novi red.

No tijekom čitanja na ove načine moramo paziti kad dođemo do kraja datoteke. Većina ovih metoda vraća nam null ako dođemo do kraja datoteke, pa samim time moramo provjeravati jesmo li došli do kraja datoteke. Koristeći ReadLine metodu, to možemo provjeriti na sljedeći način:

```
string? podaci;  
while ((podaci = sr.ReadLine()) != null)  
{  
    sw.WriteLine(podaci);  
}
```

Još jedan način na koji možemo provjeriti jesmo li došli do kraja datoteke jest korištenjem metode Peek, koja provjerava koji je sljedeći znak, ali ga neće vratiti, odnosno konzumirati, kako bi napravila metoda Read. Ukoliko Peek vrati -1 kao vrijednost, to znači da nema više znakova za čitanje iz datoteke te treba završiti s čitanjem.

Nakon što smo završili s korištenjem objekata FileStream, StreamWriter i StreamReader, te objekte trebamo zatvoriti. Zatvaranje radimo metodama Close na sljedeći način:

```
sw.Close();  
sr.Close();  
fs.Close();
```

Dobra je praksa da suprotnim redoslijedom od onog kako ste objekte otvarali objekte i zatvorite. Već smo spominjali da bi ovo bilo oslobađanje ranije zauzetih resursa, koje bismo često radili u finally bloku. Samo budite oprezni, ako se dogodila iznimka i neki od objekata nije uspio s instanciranjem, on će ostati na null vrijednosti. Zato prilikom zatvaranja u finally bloku provjerite je li vrijednost različita od null.

```
if(fs!=null) fs.Close();
```

Ako sumiramo, receptura za rad s datotekama jednostavna je i ide u nekoliko koraka:

- instancirajte FileStream objekt
- instancirajte StreamReader ili StreamWriter objekte
- radite sa StreamReader ili StreamWriter objektom
- zatvorite objekte (no pazite da napravite i flush prije zatvaranja ako radite sa StreamWriterom!).

Primjer za upisivanje teksta u datoteku i čitanja tog teksta iz datoteke:

```
using System;
using System.IO;

namespace PrvaAplikacija
{
    class Program
    {
        public static void Main()
        {
            Console.Write("Unesite ime datoteke: ");
            string datoteka = Console.ReadLine();

            //pisanje:
            FileStream fs;
            try
            {
                fs = new FileStream(datoteka, FileMode.Create);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Neuspjelo otvaranje datoteke!\n" + ex.Message);
                return;
            }

            StreamWriter sw = null;
            try
            {
                sw = new StreamWriter(fs);
                do
                {

```



```

        Console.WriteLine("Unesite rečenicu za upisati (upišite \"kraj\" za prekid): ");
        string pod = Console.ReadLine();
        if (pod == "kraj") break;
        sw.WriteLine(pod);
    } while (true);
    sw.Flush();
}
catch (Exception ex)
{
    Console.WriteLine("Greška prilikom pisanja u datoteku!\n" + ex.Message);
}
finally { if (sw != null) sw.Close(); }

if (fs != null) fs.Close();

//čitanje:
try
{
    fs = new FileStream(datoteka, FileMode.Open);
}
catch (Exception ex)
{
    Console.WriteLine("Neuspjelo otvaranje datoteke!\n" + ex.Message);
    return;
}
StreamReader sr = null;
try
{
    sr = new StreamReader(fs);
    string procitano = sr.ReadToEnd();
    Console.WriteLine("----- Sadržaj datoteke ----- ");
    Console.WriteLine(procitano);
}
catch (Exception ex)
{
    Console.WriteLine("Greška prilikom čitanja datoteke!\n" + ex.Message);
}
finally { if (sr != null) sr.Close(); }

if (fs != null) fs.Close();
}
}
}

```

U ovom primjeru koristili smo dobru praksu pisanja koda. Na početku tražimo da se preko tipkovnice unese putanja ili ime datoteke. Nakon što je korisnik unio ime datoteke, ta datoteka pokušava se otvoriti putem `fileStream` objekta. Ukoliko nastane neka vrsta greške, preko konzole piše se poruka greške, a nakon toga, budući da nema smisla dalje raditi, jednostavno se zatvara program. Greška može biti da datoteka ne postoji, nemamo dozvolu pristupa, greška prilikom rada s diskom itd. Ako je `fs` objekt instanciran, nastavljamo na instanciranje `StreamWriter` objekta. Slično kao i kod `fs` objekta, ovdje ćemo također instancirati unutar `try` bloka da ne bi bilo grešaka. Nakon instanciranja `sw` objekta preko petlje tražit ćemo da korisnik unosi rečenice dokle god ne unese riječ „kraj”. Nakon što smo sav korisnikov unos zapisali u međuspremnik, nakon gotovog unosa moramo pozvati `Flush` metodu da se sve zapiše na disk. Naravno, ako se dogodi greška, bit će uhvaćena i ispisana korisniku. Na kraju u bloku `finally` zatvaramo `sw` objekt ako je bio instanciran. Nakon zatvaranja `sw` objekta isto to radimo i s `fs` objektom.

Što se tiče čitanja, tu smo koristili istu recepturu. Instanciramo `fs` objekt, ovaj put u modu za čitanje. Instanciramo preko njega `StreamReader` objekt i u string varijablu čitamo sve što se nalazi u datoteci. Jednom pročitano, sav sadržaj ispisujemo na konzolu i redom radimo čišćenje. Zatvaramo `sr` objekt, zatvaramo `fs` objekt i gotovi smo s programom.

Klase koje smo ovdje spomenuli koristimo za čitanje i pisanje datoteka u tekstualnom obliku. Datoteke možemo čitati i pisati u njih u binarnom obliku, koristeći `BinaryRead` i `BinaryWriter` klase na sličan način. Također postoje `StringReader` i `StringWriter`, koje su slične klasama `StreamReader` i `StreamWriter`. Još treba napomenuti apstraktne klase `TextReader` i `TextWriter`, koje su bazne klase za `StreamReader`, `TextReader` i `StreamWriter`, `TextWriter`.

4.4.2. Klase za rad s datotečnim sustavom

Rukovanje datotekama i mapama inače je zadaća operativnog sustava. No imamo klase koje nam koriste da komuniciraju s operativnim sustavom, koje šalju naredbe što operativni sustav treba raditi s datotekama ili mapama. Ove klase korisne su jer

nam enkapsuliraju konkretnu implementaciju operativnog sustava i omogućuju da mi neovisno o operativnom sustavu na isti način koristimo datoteke i mape, neovisno o tome na kojem će se operativnom sustavu naš program izvršavati.

Najčešće klase koje ćemo koristiti za manipulaciju datotečnim sustavom klase su Path, File i Directory.

Klasa **Path** služi nam za rad s putanjama. Putanja ne mora nužno biti lokacija datoteke ili mape. Primjerice, putanja može mapirati lokaciju u memoriji ili neki drugi uređaj. Točan format putanje determiniran je trenutnom platformom na kojoj se program izvršava. Npr. na nekim sustavima putanja počinje slovom particije (Windows), dok na drugim taj element nije postojan (Linux). Na nekim sustavima putanje sadrže ekstenzije koje pokazuju tip informacija koje su pohranjene u datoteci. Format datotečnog imena također je specifičan za određenu platformu. Primjer limita je da su na FAT16 datotečnom sustavu ekstenzije limitirane na tri slova, a na drugim nisu. Također, trenutna platforma određuje koji znakovi odvajaju elemente putanje, koji se setovi znakova ne mogu koristiti itd. Zbog svih tih razlika polja unutar klase Path ponašaju se drukčije, ali i ponašanje klase ovisno je o točnoj platformi na kojoj se izvodi.

Putanja može sadržavati apsolutnu ili relativnu informaciju. Apsolutne putanje u potpunosti specificiraju lokaciju, odnosno datoteka i mapa mogu se unikatno identificirati neovisno o lokaciji s koje gledamo. Relativne putanje specificiraju lokaciju gdje je trenutna lokacija korištena kao startna točka prilikom traženja specificirane datoteke. Da bismo došli do trenutne mape, uvijek možemo pozvati metodu `Directory.GetCurrentDirectory`.

Većina članova Path klase ne interaktira s datotečnim sustavom i ne verificira postojanost datoteke specificirane preko string putanje. Članovi klase koji modificiraju string putanje, poput `ChangeExtension`, nemaju efekta na ime datoteke unutar datotečnog sustava.

Neke od metoda unutar Path klase koje ćemo koristiti:

- **Combine** – kombinira dva stringa u putanju
- **GetDirectoryName** – vraća informaciju mape za specificiranu putanju
- **GetFileName** – vraća informaciju datoteke za specificiranu putanju
- **GetExtension** – vraća string ekstenzije datoteke za specificiranu putanju
- **GetFileNameWithoutExtension** – vraća string imena datoteke bez ekstenzije za specificiranu putanju
- **GetFullPath** – vraća string pune putanje specificirane datoteke
- **GetRandomFileName** – vraća nasumično odabrano ime datoteke iz proslijeđene mape
- **GetTempPath** – vraća putanju prema privremenoj mapi trenutnog korisnika
- **HasExtension** – provjerava ima li datoteka na trenutnoj putanji ekstenziju
- **Join** – konkatenera više komponenti putanja u jedinstvenu putanju
- ...

Klasa **File** služi za tipične operacije nad datotekama poput kopiranja, micanja, preimenovanja, stvaranja, otvaranja, brisanja i dodavanja podataka. Klasa File također može koristiti i da pristupimo i mijenjamo attribute datoteke, poput informacija kad je datoteka stvorena, kad joj je zadnje pristupano, kad je pisano itd.

Sve metode unutar File klase statične su, odnosno ne možemo uopće instancirati objekt File klase. Sve metode File klase zahtijevaju putanju prema datoteci s kojom radimo, odnosno kojom pokušavamo manipulirati.

Neke od metoda File Klase koje ćemo koristiti:

- **Copy** – kopira postojeću datoteku na novu putanju
- **Create** – stvara novu datoteku
- **Decrypt** – dekriptira datoteku koja je enkriptirana encrypt metodom
- **Delete** – obriše datoteku na proslijeđenoj putanji
- **Encrypt** – enkriptira datoteku tako da je samo račun koji je korišten tijekom rada na računalu može dekriptirati
- **Exist** – provjerava da li datoteka na putanji postoji

- **GetCreationTime** – dohvaća kad je datoteka nastala
- **GetLastAccessTime** – dohvaća kad je datoteci zadnji put netko pristupio
- **Open** – otvara FileStream na specificiranu putanju
- **OpenRead** – otvara datoteku za čitanje
- **OpenWrite** – otvara datoteku za pisanje
- **ReadAllBytes** – otvara binarnu datoteku, čita sadržaj u byte polje i zatvara datoteku
- **ReadAllLines** – otvara datoteku, čita sav sadržaj i zatvara je
- **ReadAllText** – otvara datoteku, čita sav tekst i zatvara je
- **SetCreationTime** – postavlja vrijeme kad je datoteka stvorena
- **SetLastWriteTime** – postavlja vrijeme kad je zadnji put pisano u datoteku
- **SetLastAccessTime** – postavlja vrijeme kad je zadnji put netko pristupio datoteci
- **WriteAllLines** – piše sve linije u datoteku
- **WriteAllBytes** – piše sve podatke u binarnu datoteku
- **WriteAllText** – piše sav tekst u datoteku
- ...

Klasa **Directory** koristi se za tipične operacije nad mapama poput kopiranja, micanja, preimenovanja, stvaranja i brisanja mapa. Directory klasa statična je, što znači da su, isto kao u File klasi, sve metode statične, odnosno nije moguće instancirati klasu.

Neke od metoda Directory klase:

- **CreateDirectory** – stvara novu mapu na putanji, ukoliko već ne postoji
- **Delete** – briše mapu na specificiranoj putanji
- **EnumerateDirectories** – vraća popis pod mapa na specificiranoj putanji
- **EnumerateFiles** – vraća popis datoteka na specificiranoj putanji
- **Exists** – vraća postoji li mapa na disku koja odgovara putanji
- **GetCreationTime** – vraća kada je stvorena mapa specificirana putanjom
- **GetCurrentDirectory** – vraća punu putanju trenutne mape u kojoj se nalazi, odnosno što smatra trenutnom radnom mapom aplikacije

- **GetDirectories** – vraća imena pod mapa s njihovim putanjama u specificiranoj mapi
- **GetFiles** – vraća imena datoteka s njihovim putanjama u specificiranoj mapi
- **GetLogicalDrives** – vraća imena svih logičkih diskova na računalu
- **GetParent** – vraća roditeljsku mapu specificirane mape
- **Move** – premješta datoteku ili mapu i sve njene komponente na novu lokaciju
- ...

Još svakako treba spomenuti klasu **Environment**. Služi da dođemo do informacija o uređaju na kojem se naš program izvršava, ali i o argumentima komandne linije, da izađemo iz programa, pristupimo varijablama okoline operativnog sustava, o sadržaju pozivnog stoga, vremenu od zadnjeg podizanja operativnog sustava, verziji CLR-a i još mnogo toga.

Klasa **Environment** statična je klasa sa statičnim svojstvima i metodama. Neke od metoda i svojstva koje se nalaze unutar **Environment** klase:

- **CommandLine** – dohvaća komandnu liniju za trenutni proces
- **CurrentDirectory** – vraća punu putanju trenutne radne mape
- **ExitCode** – dohvaća ili postavlja kod za izlaz iz procesa
- **HasShutdownStarted** – dolazi do vrijednosti koja indicira da se trenutna aplikacija zatvara ili CLR da se zatvara
- **Is64BitOperatingSystem** – dolazimo do informacije da li je trenutni sustav 64 bitni
- **MachineName** – dolazimo do NetBIOS imena ovog lokalnog računala na kojem se aplikacija izvršava
- **OSVersion** – dohvaćamo identifikatora i verzije platforme, odnosno operativnog sustava
- **ProcessId** – dohvaćamo jedinstveni identifikator trenutnog procesa
- **ProcessPath** – dohvaćamo putanju u kojoj je krenulo izvršavanje trenutnog procesa

- **TickCount** – dohvaća broj otkucaja u milisekundama koji su prošli otkako je sistem pokrenut
- **UserName** – dohvaća ime korisnika koji je vlasnik trenutnog procesa izvođenja
- **Version** – dohvat pune trenutne verzije CLR-a
- **WorkingSet** – dohvat količine fizičke memorije koja je mapirana trenutnom procesu
- ...

4.5. Strukture podataka

Već smo kao osnovne kompleksne podatkovne tipove spominjali polja. No kod njih imamo velikih problema tijekom rada, a posebno ograničavajući faktor fiksna je veličina polja. Ta fiksna veličina polja radi velike probleme kad moramo mijenjati polje, odnosno dodavati i brisati elemente polja. Npr. ako već imamo maksimalno popunjeno polje, dodavanje novog elementa svodi se na to da moramo prvo napraviti novo veće polje, kopirati sve elemente koji idu prije ubačenog elementa, ubaciti element i onda kopirati sve elemente koji idu poslije ubačenog elementa. Kad smo napravili novo polje i popunili ga, onda brišemo staro polje. Ukratko, to je previše koraka koje bismo trebali raditi svaki put, ali još je gore to što je, kad se radi o velikim skupovima podataka, sporo i troši velike količine memorije.

Osim toga, polja nemaju ni ugrađene barem osnovne mogućnosti pretrage i sortiranja podataka, nego se svode na to da sami moramo implementirati algoritme za sortiranje (što smo već pokrili kad smo radili polja). Iz tih razloga, ali i još drugih, .NET ima dvije osnovne skupine kolekcija za pohranu i obradu podataka u obliku kolekcija:

- Type-unsafe kolekcije
- Type-safe kolekcije ili generičke kolekcije.

Prva i osnovna razlika između tih kolekcija jest kako će ih kompajler gledati. *Type safe* znači da kompajler jezika već unaprijed zna kojeg će podatkovnog tipa biti objekti spremljeni u kolekciju. Kod takvih kolekcija jednom kad definiramo podatkovni tip ele-

menata, možemo spremati samo objekte tog podatkovnog tipa u kolekcije. Kod *type unsafe* kolekcija nemamo takvog ograničenja, pa svaki element može biti drugog podatkovnog tipa, no prilikom izvođenja programa uvijek možemo ispitati koji je trenutni tip objekta koji se nalazi u kolekciji.

Generalno je prijedlog koristiti *type safe* kolekcije što više, kako zbog sigurnosti, tako i zbog performansi rada samih kolekcija.

Napomena – kroz sve kolekcije možemo prolaziti *foreach* petljom. Samo smatrajte *foreach* omotačem oko enumeratora, koji omogućuje samo čitanje iz kolekcije, ali ne i izmjene kolekcije. Kad kažemo kolekcije, znači da nema ubacivanja, brisanja, sortiranja i drugih operacije koje mijenjaju kolekciju, ali ostaje mogućnost promjene elementa koji je u trenutnoj iteraciji kroz *foreach* petlju.

4.5.1. Kolekcije objekata

Kad govorimo o *type unsafe*, odnosno o tipski „nesigurnim” kolekcijama, svodi se na to da se kolekcija sastoji od objekata koji su vršne klase koju nasljeđuje svaki objekt „*object*”. Kod ovih kolekcija ne moramo ni spominjati podatkovni tip jer se za njega podrazumijeva da je *object*.

Sve kolekcije ove vrste naći ćemo u imenskom prostoru `System.Collections`. Tako možemo instancirati klase kolekcija na sljedeći način:

```
ArrayList lista = new ArrayList();  
Hashtable hash_tablica = new Hashtable();  
Queue red = new Queue();  
SortedList sortirana_lista = new SortedList();  
Stack stog = new Stack();
```


Gore su pobrojane česte strukture podataka koje ćemo koristiti. Pa idemo redom spomenuti svaku:

- **ArrayList** je lista koja može imati proizvoljan broj elemenata
- **Hashtable** je kolekcija koja se sastoji od parova ključ/vrijednost koji su organizirani na bazi hash koda ključa
- **Queue** je red, koji prezentira takozvanu FIFO (first in first out), odnosno prvi koji ulazi prvi je koji izlazi strukturu podataka
- **SortedList** predstavlja kolekciju parova ključ/vrijednost koji su automatski sortirani po ključu i kojima se može pristupiti po ključu ili poziciji
- **Stack** je stog koji prezentira takozvanu LIFO (last in first out), odnosno zadnji koji ulazi prvi je koji izlazi, strukturu podataka.

Najčešće ćemo koristiti ArrayList kolekciju sa svojim metodama i svojstvima.

Neke od metoda ArrayList kolekcije:

- **Add** – dodaje novi element tipa object u kolekciju
- **Count** – vraća broj elemenata u kolekciji
- **Clear** – briše sve elemente iz kolekcije
- **Contains** – vraća true ako kolekcija sadrži proslijeđeni element, false ako ne sadrži
- **Insert** – slično kao Add metoda, no uz element moramo proslijediti i poziciju na koju želimo dodati novi element
- **Remove** – briše iz liste proslijeđeni element
- **RemoveAt** – briše iz liste element koji se nalazi na proslijeđenoj poziciji
- **Reverse** – obrće redoslijed elemenata u listi
- **Sort** – sortira listu s obzirom na proslijeđene parametre
- **ToArray** – kopira sve elemente u polje objekata. Prisjetimo se da je polje struktura fiksne veličine.

Idemo s nekoliko primjera:

Primjer programa koji traži unos brojeva dokle god se ne unese 0. Nakon toga ispisuju se uneseni elementi:

```
using System;
using System.Collections;

namespace Aplikacija
{
    class Program
    {
        public static void Main()
        {
            ArrayList lista = new ArrayList();

            int broj;
            while (true)
            {
                try
                {
                    Console.Write("Unesite broj (0 za kraj): ");
                    broj = int.Parse(Console.ReadLine());
                    if (broj == 0) break;
                    lista.Add(broj);
                }
                catch {}
            }

            Console.WriteLine("Uneseni brojevi: ");
            foreach (var item in lista)
            {
                Console.Write("{0} ", (int)item);
            }
        }
    }
}
```

Obratite pozornost na *foreach* petlju i prolazak kroz element. Budući da element `ArrayListe` može biti bilo kojeg podatkovnog tipa, tijekom korištenja elementa moramo raditi *cast* u odgovarajući podatkovni tip. Ukoliko niste sigurni u točan tip podatka, uvijek možete pozvati metodu u `Object` klasi `GetType()` da dođete do točnog podatkovnog tipa objekta tijekom izvođenja.

Uz *foreach* petlju ovdje možemo, slično kao i kod polja koristiti *for* petlju, no svakako ovisi o programeru koju će petlju koristiti u određenim situacijama. Za podsjetnik, *for* petlju bolje je koristiti ako je potrebno vraćati se natrag, a *foreach* petlju ako trebamo samo jednom proći kroz petlju.

Samo nekoliko savjeta za korištenje *foreach* petlje u dinamičkim kolekcijama:

- koristite za prolazak kroz kolekciju
- nemojte raditi izmjene na kolekciji u smislu broja elemenata ili rasporeda usred *foreach* petlje, radite samo s elementima
- objekti u kolekciji su referentnog tipa, što znači da korištenjem varijable za pristup elementu u kolekciji možemo mijenjati element kolekcije.

4.5.2. Generičke kolekcije

Type safe, odnosno kolekcije sa sigurnim tipovima, naći ćemo u `System.Collections.Generic` imenskom prostoru. Kako smo već spomenuli, kolekcije s „jakim tipovima” podataka imaju sve elemente kolekcije istog podatkovnog tipa, ali samim time naspram kolekcija objekata pružaju i bolje performanse u radu.

Prilikom instanciranja i pregleda generičkih kolekcija vidjet ćemo da imaju oznaku `<T>`, gdje `T` označava kojeg će podatkovnog tipa biti elementi kolekcije. Primjeri kolekcija koje koristimo:

```
List<int> list = new List<int>();
Dictionary<int, string> rjecnik = new Dictionary<int, string>();
HashSet<string> set= new HashSet<string>();
LinkedList<string> povezana_lista = new LinkedList<string>();
Queue<long> red = new Queue<long>();
SortedDictionary<string, string> sortiran_rjecnik = new SortedDictionary<string,
string>();
Stack<DateTime> stog = new Stack<DateTime>();
SortedSet<string> sortiran_set = new SortedSet<string>();
```

Kod ovih kolekcija moramo napomenuti kojeg će podatkovnog tipa biti elementi kolekcije. Podržani su doslovno svi tipovi podataka za ove kolekcije, što znači da čak i kolekcije mogu biti tip podataka koje koriste kolekcije. Svakako, jednom kad definiramo podatkovni tip koji će kolekcija koristiti, a definiramo ga prilikom instanciranja, nije moguće naknadno mijenjati tip podataka unutar kolekcije.

Iz generičkih kolekcija možemo praktički dobiti i negeneričke kolekcije, samo moramo navesti podatkovni tip da bude object. Realno gledano, sve generičke liste imaju svoje ekvivalentne negeneričke implementacije, odnosno kolekcije objekata.

No sad kad smo prošli osnove kolekcija, idemo prikazati kako se s njima radi na nekoliko primjera.

List<T>:

List je klasa koja nam predstavlja povezanu listu, gdje elementima možemo pristupiti preko indeksa pozicije. Također, klasa List pruža nam metode za pretragu, sortiranje i manipulaciju elementima liste. Glavna prednost korištenja liste u odnosu na polja jest u tome što nismo limitirani veličinom, jer se veličina liste dinamički mijenja ovisno o broju elemenata koji se nalaze u listi.

List<T> je generički ekvivalent ArrayList klasi i implementira IList<T> generičko sučelje korištenjem polja čija se veličina dinamički povećava po potrebi.

Dodavati elemente možemo preko Add ili AddRange metoda. List<T> koristi uspoređivač jednakosti i uspoređivač poretka. No to ne garantira da će lista biti sortirana. Moramo sortirati listu nekom metodom prije korištenja operacija koje zahtijevaju sortiranost liste. No umjesto da sami koristimo svoje načine sortiranja, klasa List<T> ima već metode za sortiranje – poput Sort metode.

Napomenuti svakako treba da List<T> prihvaća null kao valjanu vrijednost za referentne tipove i omogućuje nam da imamo ponavljanja elemenata u listi.

Nećemo prolaziti sve metode, ima ih previše da bismo pokrili sve. Svakako idemo s primjerom korištenja, pa ćemo napraviti pregled najčešćih svojstava i metoda List<T> klase:

```
List<string> dinosauri = new List<string>();

Console.WriteLine("Kapacitet: {0}", dinosauri.Capacity);

dinosauri.Add("Tyrannosaurus");
dinosauri.Add("Amargasaurus");
dinosauri.Add("Mamenchisaurus");
dinosauri.Add("Deinonychus");
dinosauri.Add("Compsognathus");
Console.WriteLine();
foreach (string dinosaur in dinosauri)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nKapacitet: {0}", dinosauri.Capacity);
Console.WriteLine("Broj: {0}", dinosauri.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosauri.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosauri.Insert(2, "Compsognathus");
Console.WriteLine();
foreach (string dinosaur in dinosauri)
{
    Console.WriteLine(dinosaur);
}
```

```
// Pristup elementu liste preko indeksa.
Console.WriteLine("\ndinosauri[3]: {0}", dinosauri[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosauri.Remove("Compsognathus");

Console.WriteLine();
foreach (string dinosaur in dinosauri)
{
    Console.WriteLine(dinosaur);
}

dinosauri.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("\nKapacitet: {0}", dinosauri.Capacity);
Console.WriteLine("Broj: {0}", dinosauri.Count);

dinosauri.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("\nKapacitet: {0}", dinosauri.Capacity);
Console.WriteLine("Broj: {0}", dinosauri.Count);

dinosauri.Sort();
foreach (string dinosaur in dinosauri)
{
    Console.WriteLine(dinosaur);
}

/* Ispis je sljedeći:

Kapacitet: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Kapacitet: 8
Broj: 5

Contains("Deinonychus"): True
```

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosauri[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Kapacitet: 5
Broj: 5

Sort()
Amargasaurus
Compsognathus
Deinonychus
Mamenchisaurus
Tyrannosaurus

Clear()
Kapacitet: 5
Broj: 0
***/**

Najčešće korištena svojstva List<T> klase:

- **Capacity** – dohvat ili postavljanje maksimalnog broja elemenata koje inter na struktura može imati bez promjene veličine
- **Count** – dohvat broja elemenata koji se nalaze u List<T> objektu.

Najčešće korištene metode List<T> klase:

- **Add** – dodavanje novog elementa na kraj liste
- **Clear** – uklanja sve elemente iz liste
- **Contains** – provjerava je li proslijeđeni element u listi
- **CopyTo** – kopira cijelu listu i polje
- **Exists** – provjerava je li element dio liste
- **Find** – pretražuje nalazi li se element u listi i, ako se nalazi, vraća poziciju prvog pojavljivanja
- **IndexOf** – vraća prvo pojavljivanje proslijeđenog objekta
- **Insert** – dodaje novi element na specificiranu poziciju
- **Remove** – uklanja prvo pojavljivanje proslijeđenog objekta
- **Sort** – sortiranje liste
- **ToArray** – kopira elemente liste u novo polje
- ...

Poveznica na List<T> klasu:

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0>

Dictionary<TKey,TValue>:

Dictionary (rječnik) klasa je koja nam omogućuje mapiranje iz seta ključeva u set vrijednosti. Svaki dodatak rječniku sadrži par koji se mora sastojati od ključa i vrijednosti koja se pridružuje ključu. Prednost ovoga je u tome što svaki podatak iz rječnika možemo dohvatiti preko njegova ključa. Prednost rječnika jest i to što je dohvat podataka jako brz, jer je Dictionary klasa interno implementirana kao hash tablica.

Bitno je reći da dokle god se objekt koristi kao ključ u Dictionary klasi, ne smije se mijenjati na način na koji bi moglo doći do promjene njegove *hash* vrijednosti. Svaki ključ u rječniku mora biti jedinstven prema rječnikovu komparatoru jednakosti. Ključ ne može biti null, ali vrijednost koja odgovara tom ključu može. Budući da komparator obavezno mora biti implementiran kad je ključ u pitanju, moramo biti oprezni oko toga koje klase koristimo za ključeve, odnosno da one imaju implementiran način usporedbe.

Prilikom prolaska kroz kolekciju preko foreach petlje za objekt elementa dobit ćemo objekt tipa **KeyValuePair<TKey,TValue>**, koji svakako sadrži ključ i vrijednost poput u primjeru:

```
foreach (KeyValuePair<string, string> item in rjecnik)
{
    Console.WriteLine("Ključ = {0}, Vrijednost = {1}", item.Key, item.Value);
}
```

Nećemo prolaziti sve metode, ima ih previše da bismo sve pokrili, no svakako idemo s primjerom korištenja, pa ćemo napraviti pregled najčešćih svojstava i metoda Dictionary<TKey,TValue> klase:

```
// Stvaranje novog rječnika sa string ključevima i string vrijednosti.
Dictionary<string, string> openWith = new Dictionary<string, string>();
```

```
// Dodavanje elemenata u rječnik.
// Nema duplikata ključeva, ali ima vrijednosti.
openWith.Add(„txt“, „notepad.exe“);
openWith.Add(„bmp“, „paint.exe“);
openWith.Add(„dib“, „paint.exe“);
openWith.Add(„rtf“, „wordpad.exe“);
```

```
// Add metoda baca iznimku za duplikat ključa.
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
```

```

{
    Console.WriteLine("Element s ključem = \"txt\" već postoji.");
}

// elementu rječnika ne mora se pristupati preko broja indeksa
// možemo umjesto broja indeksa koristiti ključ rječnika
Console.WriteLine("Za ključ = \"rtf\", vrijednost = {0}.",
    openWith["rtf"]);

// Indeks se može koristiti za promjenu vrijednosti.
openWith["rtf"] = "winword.exe";
Console.WriteLine("Za ključ = \"rtf\", vrijednost = {0}.",
    openWith["rtf"]);

// Ako ključ ne postoji, postavljanje novog indeksa dodaje novu vrijednost.
openWith["doc"] = "winword.exe";

// Indeks baca iznimku ako ključ ne postoji u rječniku.
try
{
    Console.WriteLine("Za ključ = \"tif\", vrijednost = {0}.",
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Ključ = \"tif\" nije pronađen.");
}

// Kad program treba pokušati raditi s ključem koji se ne nalazi u rječniku.
// TryGetValue efikasniji je način rada, odnosno dohvata vrijednosti.
string value = "";
if (openWith.TryGetValue("tif", out value))
{
    Console.WriteLine("Za ključ = \"tif\", vrijednost = {0}.", value);
}
else
{
    Console.WriteLine("Ključ = \"tif\" nije pronađen.");
}

// ContainsKey može se koristiti za testiranje postojanosti ključa prije unosa.

```

```

if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Vrijednost dodana za ključ = \"ht\": {0}",
        openWith["ht"]);
}

```

// Prolaz kroz rječnik foreach petljom.

```

Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in openWith)
{
    Console.WriteLine("Ključ = {0}, Vrijednost = {1}",
        kvp.Key, kvp.Value);
}

```

// Za dohvat samo vrijednosti koristimo Values svojstvo.

```

Dictionary<string, string>.ValueCollection valueColl =
    openWith.Values;

```

// elementi ValueCollection strogo su tipizirani

// tipom specificiranim u deklaraciji rječnika

```

Console.WriteLine();
foreach (string s in valueColl)
{
    Console.WriteLine("Vrijednost = {0}", s);
}

```

// Za dohvat samo ključeva koristimo Keys svojstvo.

```

Dictionary<string, string>.KeyCollection keyColl =
    openWith.Keys;

```

// elementi KeyCollection strogo su tipizirani

// tipom specificiranim u deklaraciji rječnika

```

Console.WriteLine();
foreach (string s in keyColl)
{
    Console.WriteLine("Ključ = {0}", s);
}

```

// Koristimo Remove metodu za brisanje para.

```

Console.WriteLine("\nRemove(\"doc\")");

```

```
openWith.Remove("doc");
```

```
if (!openWith.ContainsKey("doc"))  
{  
    Console.WriteLine("Ključ \"doc\" nije nađen.");  
}
```

/ Ispis je sljedeći:*

Element s ključem = "txt" već postoji.
Za ključ = "rtf", vrijednost = wordpad.exe.
Za ključ = "rtf", vrijednost = winword.exe.
Key = "tif" nije pronađen.
Key = "tif" nije pronađen.
Vrijednost dodana za ključ = "ht": hypertrm.exe

Ključ = txt, Vrijednost = notepad.exe
Ključ = bmp, Vrijednost = paint.exe
Ključ = dib, Vrijednost = paint.exe
Ključ = rtf, Vrijednost = winword.exe
Ključ = doc, Vrijednost = winword.exe
Ključ = ht, Vrijednost = hypertrm.exe

Vrijednost = notepad.exe
Vrijednost = paint.exe
Vrijednost = paint.exe
Vrijednost = winword.exe
Vrijednost = winword.exe
Vrijednost = hypertrm.exe

Ključ = txt
Ključ = bmp
Ključ = dib
Ključ = rtf
Ključ = doc
Ključ = ht

Remove("doc")
Ključ "doc" nije pronađen.

*

/

Najčešće korištena svojstva Dictionary<TKey,TValue> klase:

- **Count** – dohvat broja elemenata koji se nalaze u objektu
- **Keys** – dohvat kolekcije koja sadrži samo ključeve
- **Values** – dohvat kolekcije koja sadrži samo vrijednosti.

Najčešće korištene metode Dictionary<TKey,TValue> klase:

- **Add** – dodavanje novog elementa u rječnik
- **Clear** – uklanja sve elemente iz rječnika
- **Contains** – provjerava je li proslijeđeni element u rječniku
- **Remove** – uklanja vrijednost po specificiranom ključu iz kolekcije
- ...

Poveznica na Dictionary<TKey,TValue> klasu:

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-6.0>

Queue<T>:

Queue (red) klasa implementira generički klasu za rad s cirkularnim poljima. Objekti koji se ubacuju u red vade se iz reda redoslijedom kako su dodani. Ovaj pristup zovemo FIFO (*First In First Out*) struktura, gdje je prvi element koji je dodan u strukturu ujedno i prvi element koji se obrađuje iz strukture. Redovi, ali i stogovi koje ćemo poslije ovoga proći često su korisni kao privremeni spremnici informacija. Kad kažemo privremeni, to znači da se nakon iskorištavanja elementa iz reda on odbacuje, odnosno ne ostaje spremljen u redu.

Red ima tri osnovne operacije:

- **Enqueue** – dodavanje elementa na kraj reda (slično kao Add kod liste)
- **Dequeue** – uklanjanje najstarijeg elementa liste
- **Peek** – vraća najstariji element reda, no ne miče ga iz reda.

Nećemo prolaziti sve metode, ima ih previše da bismo ih sve pokrili, no svakako idemo s primjerom korištenja:

```
Queue<string> brojevi = new Queue<string>();
brojevi.Enqueue("jedan");
brojevi.Enqueue("dva");
brojevi.Enqueue("tri");
brojevi.Enqueue("četiri");
brojevi.Enqueue("pet");
```

// Kroz red možemo proći bez utjecaja na sadržaj.

```
foreach (string broj in brojevi)
{
    Console.WriteLine(broj);
}
```

```
Console.WriteLine("\nDequeuing '{0}'", brojevi.Dequeue());
Console.WriteLine("Kraj reda, sljedeći element koji možemo obraditi: {0}",
    brojevi.Peek());
Console.WriteLine("Dequeuing '{0}'", brojevi.Dequeue());
```

// Stvaranje kopije reda korištenjem ToArray metode.

```
Queue<string> Kopija_reda = new Queue<string>(brojevi.ToArray());
```

```
Console.WriteLine("\nSadržaj prve kopije:");
foreach (string broj in Kopija_reda)
{
    Console.WriteLine(broj);
}
```

// Stvaranje polja dvostruke veličine reda i kopiranje

// elemenata reda, počevši od sredine polja.

```
string[] polje = new string[brojevi.Count * 2];
brojevi.CopyTo(polje, brojevi.Count);
```

// Stvaranje drugog reda od polja.

```
Queue<string> Kopija_reda2 = new Queue<string>(polje);
```

```
Console.WriteLine("\nSadržaj druge kopije, s duplikatima i null vrijednostima:");
foreach (string broj in Kopija_reda2)
{
    Console.WriteLine(broj);
}
```

```
}
```

```
Console.WriteLine("\nKopija_reda.Contains(\"četiri\") = {0}",  
    Kopija_reda.Contains("četiri"));
```

```
Console.WriteLine("\nKopija_reda.Clear()");  
Kopija_reda.Clear();
```

```
Console.WriteLine("\nKopija_reda.Count = {0}", Kopija_reda.Count);
```

```
/* Ispis je sljedeći
```

```
jedan  
dva  
tri  
četiri  
pet
```

```
Dequeuing 'jedan'
```

```
Kraj reda, sljedeći element koji možemo obraditi: dva
```

```
Dequeuing 'dva'
```

```
Sadržaj prve kopije:
```

```
tri  
četiri  
pet
```

```
Sadržaj druge kopije, s duplikatima i null vrijednostima:
```

```
tri  
četiri  
pet
```

```
Kopija_reda.Contains("četiri") = True
```

```
Kopija_reda.Clear()
```

```
Kopija_reda.Count = 0
```

```
*/
```

Stack<T>:

Stack (stog) klasa je slična kao i klasa za red u većini stvari. Jedina je razlika to što sa stogom implementiramo takozvanu LIFO (*Last In First Out*) strukturu, gdje je zadnji element koji je ušao u stog prvi element koji obrađujemo. Tipična se stog upotrebljava za očuvanje stanja varijabli tijekom poziva procedura.

Glavne tri operacije za rad sa stogom i njegovim elementima jesu:

- **Push** – ubacuje element na vrh stoga
- **Pop** – uklanja element s vrha stoga
- **Peek** – vraća element na vrh stoga, ali ga ne uklanja sa stoga.

Nećemo prolaziti sve metode, ima ih previše da bismo ih sve pokrili, no svakako idemo s primjerom korištenja:

```
Stack<string> brojevi = new Stack<string>();
brojevi.Push("jedan");
brojevi.Push("dva");
brojevi.Push("tri");
brojevi.Push("četiri");
brojevi.Push("pet");
```

// Kroz stog možemo proći bez utjecaja na sadržaj.

```
foreach (string broj in brojevi)
{
    Console.WriteLine(broj);
}
```

```
Console.WriteLine("\nPopping '{0}'", brojevi.Pop());
Console.WriteLine("Vrh stoga, sljedeći element za obradu: {0}",
    brojevi.Peek());
Console.WriteLine("Popping '{0}'", brojevi.Pop());
```

// Stvaranje kopije stoga korištenjem ToArray metode.

```
Stack<string> kopija_stog = new Stack<string>(brojevi.ToArray());
```



```

Console.WriteLine("\nSadržaj prve kopije:");
foreach (string number in kopija_stog)
{
    Console.WriteLine(number);
}

// Stvaranje polja dvostruke veličine stoga i kopiranje
// elemenata reda, počevši od sredine polja.
string[] polje = new string[brojevi.Count * 2];
brojevi.CopyTo(polje, brojevi.Count);

// Stvaranje drugog stoga od polja.
Stack<string> stog = new Stack<string>(polje);

Console.WriteLine("\nSadržaj druge kopije, s duplikatima i null vrijednostima:");
foreach (string broj in stog)
{
    Console.WriteLine(broj);
}

Console.WriteLine("\nkopija_stog.Contains(\"four\") = {0}",
    kopija_stog.Contains("four"));

Console.WriteLine("\nkopija_stog.Clear()");
kopija_stog.Clear();
Console.WriteLine("\nkopija_stog.Count = {0}", kopija_stog.Count);

/* Ispis je sljedeći:

pet
četiri
tri
dva
jedan

Popping 'pet'
Vrh stoga, sljedeći element za obradu: četiri
Popping 'četiri'

```

Sadržaj prve kopije:

jedan

dva

tri

Sadržaj druge kopije, s duplikatima i null vrijednostima:

jedan

dva

tri

```
kopija_stog.Contains("četiri") = False
```

```
kopija_stog.Clear()
```

```
kopija_stog.Count = 0
```

```
*/
```

4.6. Pretvorbe podataka

Tijekom rada s podacima neizbježno je da ćemo često morati zapis podataka prilagoditi potrebama određenih sustava ili da ćemo iz drugih sustava morati obraditi zaprimljene podatke. Kad govorimo o prilagodbi podataka, često mislimo na pretvorbu podataka iz određenog formata zapisa ili u njega. Pretvorba u ovom slučaju nije tako jednostavna, odnosno nije kao što smo dosad radili samo „cast” jedne klase u drugu, odnosno jednog podatkovnog tipa u drugi.

U ovom poglavlju prikazat ćemo kako koristimo, odnosno kako čitamo i zapisujemo podatke s dva najčešća formata dokumenta, a to su **XML** i **JSON**.

4.6.1. XML

XML ili **Extensible Markup Language** jezik je i format datoteke za spremanje, prijenos i rekonstrukciju podataka. Definira set pravila za enkodiranje dokumenata u formatu koji je istovremeno čitljiv ljudima i strojevima. XML je otvoreni standard koji je

publiciran 1998. (započeo 1996.), s aktualnom verzijom standarda 1.1 izdanom 2006. godine.

Glavna svrha XML-a jest serijalizacija. To znači sljedeće: da bi dva različita sustava razmjenjivala informacije, moraju se dogovoriti oko toga koji će datotečni format koristiti. XML standardizira taj proces.

XML koristi oznake, kategorije i strukturalnu organizaciju podataka. XML tagovi predstavljaju podatkovnu strukturu i sadrže metapodatke. Unutar tih tagova nalaze se podaci, enkodirani na način koji je specificiran XML standardom. Dodatna XML shema (XDS) definira potrebne metapodatke za interpretaciju i validaciju XML-a. XML dokument koji odgovara osnovnim XML pravilima jest „dobro formatiran”, a onaj koji odgovara svojoj shemi „validan” je.

XML je postao standardno korišten za razmjenu podataka preko interneta. Stotine različitih formata dokumenata nastalo je na temelju XML-a, uključujući RSS, Atom, Open Office XML, OpenDocument, SVG, XHTML itd. XML također pruža baznu jezičnu potporu za komunikacijske protokole kao što su SOAP i XMPP. Također, XML je format za razmjenu poruka za asinkroni JavaScript i AJAX programsku tehniku.

Ključni termini jezika jesu:

- **znak** – XML dokument niz je znakova (string) i time se bilo koji unicode znak može koristiti u XML-u
- **procesor i aplikacija** – procesor analizira jezik i prosljeđuje strukturne informacije aplikaciji. U specifikaciji procesor je XML parser, a propisano je što mora i ne mora raditi. No aplikacija je izvan dosega specifikacije, a to je ono što mi kao programeri pišemo, odnosno u našoj aplikaciji mi određujemo kako i što ćemo raditi s informacijama
- **markup i sadržaj** – znakovi koji grade xml dokument podijeljeni su u markup i sadržaj, koji su čisto vidljivi uz apliciranje jednostavnih sintaksnih pravila. Generalno, stringovi koji grade markup počinju znakom <, a završavaju znakom > ili počinju sa znakom & i završavaju sa znakom ; . Nizovi znakova koji nisu markup su sadržaj

- **tag** – tag je markup konstrukcija koja počinje s < i završava s > . Postoje tri vrste tagova, a to su:
 - startni ili otvarajući tag poput <sekcija>
 - završni ili zatvarajući tag poput </sekcija>
 - prazni element tag, koji često zovemo samozatvarajući tag < sekcija />
- **element** – logička komponenta dokumenta koja počinje odgovarajućim tagom i završava odgovarajućim tagom ili sadrži samozatvarajući tag. Znakovi između otvarajućeg i zatvarajućeg taga, ako postoje, sadržaj su elementa, a može sadržavati sam sadržaj, ali i druge elemente
- **atribut** – atribut je markup konstrukcija koja sadrži parove ključ/vrijednost koji se nalaze unutar startnog taga ili samozatvarajućeg taga. Primjer:
- **XML deklaracija** – XML dokument može početi XML deklaracijom koja opisuje neke informacije o njima samima. Primjer je <?xml version="1.0" encoding="UTF-8" ?>.

Za rad u .NET-u imamo mnogo klasa za rad s XML dokumentima, a one se nalaze u System.Xml imenskom prostoru.

Glavna je klasa XmlDocument klasa, koja se može koristiti za učitavanje, validaciju, uređivanje, dodavanje i pozicioniranje unutar XML dokumenta.

Najbolje je prikazati osnovne radnje na primjerima. Započet ćemo s XML dokumentom koji sadrži nekoliko knjiga u kolekciji. Sadržaj je sljedeći u datoteci „booksData.xml”:

```
<?xml version="1.0" encoding="utf-8"?>
<books xmlns="http://www.contoso.com/books">
  <book genre="novel" ISBN="1-861001-57-8" publication
    date="1823-01-28">
    <title>Pride And Prejudice</title>
    <price>24.95</price>
  </book>
  <book genre="novel" ISBN="1-861002-30-1" publication
    date="1985-01-01">
    <title>The Handmaid's Tale</
    >
```

```

        <price>29.95</price>
    </book>
    <book genre="novel" ISBN="1-861001-45-3" publication
date="1811-01-01">
        <title>Sense and Sensibility</title>
        <price>19.95</price>
    </book>
</books>

```

Sljedeći primjer prikazuje kako da učitamo podatke iz datoteke. Ako datoteka ne postoji, jednostavno generiramo u kodu xml i učitamo te podatke:

```

XmlDocument doc = new XmlDocument();
doc.PreserveWhitespace = true;
try { doc.Load("booksData.xml"); }
catch (System.IO.FileNotFoundException)
{
    doc.LoadXml("<?xml version='1.0'?> \n" +
        "<books xmlns='http://www.contoso.com/books'> \n" +
        "  <book genre='novel' ISBN='1-861001-57-8' publication
            date='1823-01-28'> \n" +
        "    <title>Pride And Prejudice</title> \n" +
        "    <price>24.95</price> \n" +
        "  </book> \n" +
        "  <book genre='novel' ISBN='1-861002-30-1' publication
            date='1985-01-01'> \n" +
        "    <title>The Handmaid's Tale</title> \n" +
        "    <price>29.95</price> \n" +
        "  </book> \n" +
        "</books>");
}

```

Generalno postoje dvije metode za čitanje XML-a u C#-u, a to je pomoću XmlDocument klase i XmlReader klase. XmlDocument pročita cijeli XML sadržaj, sprema ga u memoriju i dozvoljava nam navigaciju u samom dokumentu kako god mi želimo, čak i da radimo upite korištenjem XPath tehnologije. XmlReader klasa brža je i koristi manje memorije, no dozvoljava prolazak kroz XML sadržaj samo element po element.

Idemo pokušati s primjerom, pročitat ćemo xml dokument koji sadrži tečajnu listu iz europske centralne banke. Iako je možemo skinuti i pročitati na lokalnom računalu, klase `XmlReader` i `XmlDocument` podržavaju čitanje XML-a s udaljenih adresa. Prednost čitanja s udaljene adrese jest to da uvijek imamo „svježe”, odnosno aktualne podatke. Kod je sljedeći:

```
XmlReader xmlReader = XmlReader.Create("http://www.ecb.int/stats/eurofxref/  
eurofxref-daily.xml");  
  
while (xmlReader.Read())  
{  
    if ((xmlReader.NodeType == XmlNodeType.Element) && (xmlReader.Name ==  
                                                    "Cube"))  
    {  
        if (xmlReader.HasAttributes)  
            Console.WriteLine(xmlReader.GetAttribute("currency") + ": " + xmlReader.  
                                                    GetAttribute("rate"));  
    }  
}
```

Počinjemo tako da stvorimo `XmlReader` instancu, korištenjem statične `Create()` metode. Unutar *while* petlje pozivamo *read* metodu, koja ide redom element po element i vraća istinu dokle god postoje elementi za čitanje. Unutar petlje možemo koristiti svojstva za pristup podacima trenutnog elementa.

U ovom primjeru provjeravamo `NodeType` da vidimo postoji li element i je li ime elementa „Cube”. Svaki je tečaj unutar elementa „Cube”, pa time i ono što mi tražimo; ostatak se ignorira. Inače, kopiranjem linka unutar *create* metode možete u pregledniku vidjeti točno kako izgleda xml koji učitavamo sa servera Europske centralne banke. Jednom kad imamo element koji nam treba, provjerimo ima li atribut i koristimo `GetAttribute` metodu za čitanje atributa „currency” i „rate”. Ispisujemo ih na konzolu i pomičemo se na sljedeći element. Rezultat će biti lista valuta i njihova vrijednost u usporedbi s eurom.

Ovaj pristup jednostavan je, ali samo zato što trebamo podatke istim redoslijedom kako se čitaju, bez ikakve obrade. U sljedećem primjeru isto to napraviti ćemo s `XmlDocument` klasom.

```

XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml");
foreach (XmlNode xmlNode in xmlDoc.DocumentElement.ChildNodes[2].ChildNodes[0].ChildNodes)
    Console.WriteLine(xmlNode.Attributes["currency"].Value + ": " + xmlNode.Attributes["rate"].Value);

```

Razlika naspram XmlReader klase jest to što cijeli sadržaj XML-a učitavamo u memoriju, a zatim podatke čitamo na hijerarhijski način, baš kao što je XML strukturiran. Pristup Cube čvoru ovdje radimo tako da idemo niz hijerarhiju. Od DocumentElement korijenskog čvora tražimo treće dijete, zatim unutar tog čvora tražimo prvi podčvor, pa zatim uzimamo cijelu podkolekciju čvorova. Ovo očito možemo napraviti kad znamo točnu strukturu XML dokumenta, ali ovakav način daleko je od fleksibilnog, lijepog ili laganog za kasniju izmjenu. No način navigacije u XML dokumentu uvelike ovisi o sustavu kojem pristupamo, odnosno načinu na koji nam pripremi podatke.

Da bismo radili s dokumentom, svakako trebamo moći raditi s čvorovima XML dokumenta, a s čvorovima ćemo raditi preko XmlNode klase. Generalno, cijeli XML je parsiran u XmlNode korijenski element, te zatim možemo pristupiti podčvorovima koristeći ChildNodes svojstvo. No XmlNode klasa daje nam pristup mnogim drugim informacijama poput imena taga, *attribute* taga, unutarnjeg teksta i samog xml-a.

Svojstvo Name daje nam ime čvora u kojem se nalazimo. Na primjer, sljedeći će nam kod za izlaz dati tekst „korisnik”:

```

XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<korisnik ime=\"Marko Marulić\">Čvor korisnika</korisnik>");
Console.WriteLine(xmlDoc.DocumentElement.Name);

```

Svojstva InnerText i InnerXml slična su, nod InnerText neće prikazati XML koji bi mogao biti unutar njega, dok ga InnerXml prikazuje. Sljedeći primjer ilustrira tu razliku:

```

XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<korisnici><korisnik>InnerText/InnerXml je ovdje</korisnik></korisnici>");
Console.WriteLine("InnerText: " + xmlDoc.DocumentElement.InnerText);
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);

```

Svojstvo OuterXml isto je kao InnerXml, ali će uključiti XML čvor u kojem se nalazi. Sljedeći primjer to ilustrira:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<korisnici><korisnik>InnerText/InnerXml je ovdje</korisnik></korisnici>");
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);
Console.WriteLine("OuterXml: " + xmlDoc.DocumentElement.OuterXml);
```

Navigaciju unutar dokumenta radimo preko nekoliko svojstava unutar XmlDocument klase, kako smo već prije rekli. No navigacija nije samo u jednom smjeru. Uvijek možemo pristupiti i nadređenom, odnosno roditeljskom čvoru putem svojstva ParentNode. Također, za listanje čvorova unutar određene strukture svojstva poput FirstChild, NextSibling, LastChild i PreviousSibling dobra su za prolazak kroz kolekciju čvorova unutar trenutnog čvora XML strukture.

Uz navigaciju, uvijek možemo raditi i upite nad XML-om u svrhu pretrage, da bismo našli čvor za koji bi nam bilo prekomplikirano ili dugotrajno tražiti ga prolaskom u kodu kroz njegovu strukturu. Tehnologija koja se koristi za pretrage čvorova unutar XML-a se zove XPath i održavana je od strane iste organizacije koja je stvorila XML standard. XPath je u biti zaseban jezik za upite nad XML-om, s mnogo mogućnosti, ali ovdje ćemo se koncentrirati na nekoliko osnovnih upita.

XmlDocument klasa ima nekoliko metoda koje uzimaju XPath upit kao parametar i vraćaju XmlNode ili više njih kao rezultat upita. Ovdje ćemo se koncentrirati na SelectSingleNode i SelectNodes metode, gdje prva vraća samo jedan čvor, a druga kolekciju čvorova.

Ovdje ćemo koristiti primjer parsiranja RSS obavijesti od strane CNN-a (novinske mreže) o vijestima iz cijelog svijeta.

U sljedećem primjeru koristit ćemo SelectSingle Node metodu da dođemo do naslova obavijesti, odnosno niza obavijesti. Ako pogledamo sam XML, vidjet ćemo da postoji <title> element kao podelement elementa <channel>, koji je podelement elementa

<rss>, koji je korijenski element. Taj se upit može opisati kao XPath na sljedeći način:
`//rss/channel/title`

Sam kod primjera bio bi ovaj:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("http://rss.cnn.com/rss/edition_world.rss");
XmlNode titleNode = xmlDoc.SelectSingleNode("//rss/channel/title");
if (titleNode != null) Console.WriteLine(titleNode.InnerText);
```

SelectSingleNode metoda uzima XPath kao string parametar, zatim provjerava je li neki rezultat vraćen uspoređujući imamo li null vrijednost za XmlNode objekt, odnosno je li različito od null ako je element nađen.

Sljedeći primjer demonstrira SelectNodes metodu, koja pronalazi sve elemente xml-a sa sadržajem unutar RSS-a i ispisuje informacije o njima:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("http://rss.cnn.com/rss/edition_world.rss");
XmlNodeList itemNodes = xmlDoc.SelectNodes("//rss/channel/item");
foreach (XmlNode itemNode in itemNodes)
{
    XmlNode titleNode = itemNode.SelectSingleNode("title");
    XmlNode dateNode = itemNode.SelectSingleNode("pubDate");
    if ((titleNode != null) && (dateNode != null))
        Console.WriteLine(dateNode.InnerText + ": " + titleNode.InnerText);
}
```

SelectNodes funkcionira na identičan način kao i SelectSingleNode u prethodnom primjeru, no budući da dobivamo kolekciju rezultata, možemo kroz nju iterirati *foreach* petljom.

Budući da je XML običan tekst, mogli bismo jednostavno početi pisati XML tagove u datoteku i dati joj .xml ekstenziju. No to nećemo raditi tako jer nam .NET daje nekoliko različitih klasa da to uvelike pojednostavimo i poluautomatiziramo. Naravno, slično kao s čitanjem, imamo pristup preko XMLWritera i XmlDocument klasa. Ovisno

o odabranoj klasi svest će se na slično kao i s čitanjem, na količinu memorije koju koriste. XmlWriter koristi daleko manje memorije od XmlDocument klase, što je problem samo ako imamo jako velike datoteke. No ovdje imamo još jednu jako bitnu razliku –XmlDocument klasom možemo čitati i pisati, a XmlWriter klasom možemo samo pisati, kao što XmlReader klasom možemo samo čitati.

Primjer zapisivanja XML-a korištenjem XmlWriter klase:

```
XmlWriter xmlWriter = XmlWriter.Create("test.xml");
```

```
xmlWriter.WriteStartDocument();  
xmlWriter.WriteStartElement("korisnici");
```

```
xmlWriter.WriteStartElement("korisnik");  
xmlWriter.WriteAttributeString("dob", "42");  
xmlWriter.WriteString("Marko Marulić");  
xmlWriter.WriteEndElement();
```

```
xmlWriter.WriteStartElement("korisnik");  
xmlWriter.WriteAttributeString("dob", "39");  
xmlWriter.WriteString("August Šenoa");  
xmlWriter.WriteEndElement();
```

```
xmlWriter.WriteEndElement();  
xmlWriter.WriteEndDocument();  
xmlWriter.Close();
```

Kod će stvoriti datoteku test.xml sljedećeg sadržaja:

```
<korisnici>  
  <korisnik dob="42">Marko Marulić</korisnik>  
  <korisnik dob="39">August Šenoa</korisnik>  
</korisnici>
```

Kod je sličan kao kad radimo s datotekama, pa ga idemo ukratko objasniti. Započinjemo stvaranjem instance objekta XmlWriter i statičnom metodom Create stvaramo datoteku na putanji gdje se datoteka treba napraviti. Prvo što napravimo jest sljedeće

– zovemo metodu `WriteStartDocument`, nakon čega počinjemo s elementom korisnici. `XmlWriter` to interpretira kao da treba krenuti s `<korisnici>` tagom. Prije nego što ga zatvori, rekli smo da započne drugi početni element „korisnik”, koji će postati podčvor od čvora „korisnici”. Zatim dodajemo atribut na čvor korištenjem `WriteAttributeString` i nakon toga pišemo unutarnji tekst elementa pozivom na metodu `WriteString`. Metodom `WriteEndElement` osiguravamo zatvaranje zadnje započetog elementa. Da bi na kraju `XmlWriter` zapisao XML na disk, moramo pozvati metodu `Close`.

Iako je jednostavno koristiti `XmlWriter`, još je jednostavnije, ali često i lakše raditi s `XmlDocument` klasom. To nam je očito pogotovo kad imamo situacija gdje postojeću XML datoteku treba otvoriti, ali i izmjene koje su rađene u hijerarhiji, bilo brisanjem, dodavanjem ili izmjenama čvorova, treba spremiti.

Sljedeći primjer koristit ćemo da bismo napravili isto što smo u prošlom primjeru napravili s `XmlWriter` klasom, samo ovaj put s `XmlDocument` klasom:

```
XmlDocument xmlDoc = new XmlDocument();
XmlNode rootNode = xmlDoc.CreateElement("korisnici");
xmlDoc.AppendChild(rootNode);
```

```
XmlNode userNode = xmlDoc.CreateElement("korisnik");
XmlAttribute attribute = xmlDoc.CreateAttribute("dob");
attribute.Value = "42";
userNode.Attributes.Append(attribute);
userNode.InnerText = "Marko Marulić";
rootNode.AppendChild(userNode);
```

```
userNode = xmlDoc.CreateElement("korisnik");
attribute = xmlDoc.CreateAttribute("dob");
attribute.Value = "39";
userNode.Attributes.Append(attribute);
userNode.InnerText = "August Šenoa";
rootNode.AppendChild(userNode);
```

```
xmlDoc.Save("test-doc.xml");
```

Pristup je više objektno orijentiran nego s XmlWriter klasom, ali zahtijeva i malo više koda. Nakon instanciranja XmlDocument objekta taj ćemo objekt koristiti da bismo kreirali novi element i atribut, korištenjem CreateElement i CreateAttribute metode. Svaki put kad to napravimo, element koji smo kreirali vežemo na dokument ili neki drugi element s Append ili AppendChild metodama, efektivno kreirajući hijerarhiju XML dokumenta. Cijeli dokument sprema se na disk pomoću Save metode.

Iako zahtijeva više koda za rad s XmlDocument klasom, omogućuje nam da radimo dosta brze izmjene na već postojećem XML dokumentu. Sljedeći primjer pokazuje kako bismo radili izmjene – povećavajući dob svake osobe u xml dokumentu.

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("test-doc.xml");
XmlNodeList korisniciNodes = xmlDoc.SelectNodes("//korisnici/korisnik");
foreach (XmlNode korisnikNode in korisniciNodes)
{
    int age = int.Parse(korisnikNode.Attributes["dob"].Value);
    korisnikNode.Attributes["dob"].Value = (age + 1).ToString();
}
xmlDoc.Save("test-doc.xml");
```

Kad bismo ovo htjeli raditi preko XmlReader i XmlWriter klasa, prvo bismo trebali čitati sve podatke, spremiti u neku kolekciju, zatim prolaziti kroz kolekciju i spremati sve članove kolekcije u datoteku. Na ovakav način, korištenjem XmlDocument klase, jednostavno učitamo cijelu hijerarhiju, napravimo izmjene koje trebamo i samo kažemo da spremi sve.

4.6.2. JSON

JSON (JavaScript Object Notation) otvoreni je standard formatiranja datoteke i format podatkovne razmjene koji koristi ljudima čitljivi tekst za pohranu i prijenos podatkovnih objekata sadržavajući atribut-vrijednost parove i polja ili njihove serijalizirane vrijednosti. Dosta je čest format, čija je upotreba široka u razmjeni podataka elek-

troničkim putem, a najviše u razmjeni podatka web-aplikacija s poslužiteljima (serverima).

JSON format neovisan je o programskom jeziku, iako je deriviran iz JavaScripta. Mnogi programski jezici uključuju kod za generiranje i parsiranje podataka u JSON formatu. Datoteke u kojima su podaci formatirani u ovom obliku koriste .json ekstenziju.

Sam format započet je 2001. godine, ali raspoloživ je kao internacionalna specifikacija od 2006. Prvi put je standardiziran 2013. godine, no 2017. publicirana je verzija koja se trenutno koristi.

Pravila JSON sintakse:

- svi podaci su u ime/vrijednost parovima
- podaci su razdvojeni zarezom
- vitičaste zagrade sadrže objekte
- uglate zagrade sadrže polja.

JSON podatak zapisan je u obliku para imena i podatka (vrijednosti), slično kao svojstvo u JavaScript objektu. Par se sastoji od polja ime u dvostrukim navodnicima, zatim dvotočke, pa vrijednosti. Primjer podatka:

```
"ime": "Marko"
```

Objekt u JSON-u piše se unutar vitičastih zagrada. Objekti mogu sadržavati više podataka, kao u sljedećem primjeru:

```
{ "ime": "Marko", "prezime": "Marulić" }
```

Ukoliko želimo više podataka, odnosno polje objekata, možemo ga definirati na sljedeći način:

```
"zaposlenici": [  
  {  
    "ime": "Marko",
```

```
    "prezime": "Marulić"  
  },  
  {  
    "ime": "Ivan",  
    "prezime": "Horvat"  
  },  
  {  
    "ime": "Ana",  
    "prezime": "Anić"  
  }  
]
```

Nakon imena objekta stavljamo dvotočku, a zatim unutar otvorene i zatvorene uglate zagrade nabrajamo objekte od kojih se polje sastoji.

S novijim verzijama postalo je lakše koristiti JSON s C# programskim jezikom jer ne zahtijeva više dodatne vanjske biblioteke za rad s JSON formatom, odnosno više ne trebamo vanjske biblioteke koje bismo preuzeli preko NuGeta. System.Text.Json imenski prostor sadrži svu potrebnu funkcionalnost za serijalizaciju i deserijalizaciju JSON-a.

Samo jedna opaska, System.Text.Json nije podržan u svim verzijama .NET-a, nego zahtijeva jedno od sljedećeg:

- .NET Core 3.0 ili noviju
- .NET 6 ili novije.

Za prijašnje verzije potrebno je instalirati System.Text.Json preko NuGet paketa, koji podržava sljedeće verzije:

- .NET Standard 2.0 i noviji
- .NET Framework 4.7.2 i noviji
- .NET Core 2.1 i noviji
- .NET 5 i noviji.

Ukratko, tijekom korištenja, da bismo napravili zapis u JSON formatu, sve što trebamo je pozvati JsonSerializer.Serialize metodu i dati joj odgovarajući objekt kao u sljedećem primjeru:

```
using System;
using System.Collections.Generic;
using System.Text.Json;

namespace Aplikacija
{
    class Zaposlenik
    {
        public string Ime { get; set; }
        public string Prezime { get; set; }
        public DateTime DatumRodjenja { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            var zaposlenik = new Zaposlenik
            {
                Ime = "Nikola",
                Prezime = "Tesla",
                DatumRodjenja = new DateTime(1856, 7, 10)
            };

            string json=JsonSerializer.Serialize(zaposlenik);

            Console.WriteLine(json);
        }
    }
}

// Ispis:
// {"Ime":"Nikola","Prezime":"Tesla","DatumRodjenja":"1856-07-10T00:00:00"}
```

Primjer pokazuje kako je „lagano” napraviti serijalizaciju objekta. No često ćemo to morati zapisati u neku datoteku, odnosno trebat ćemo datoteku spremi za slanje, a u C#-u to možemo lagano napraviti tako da iz gornjeg primjera napravljeni json string samo zapišemo u datoteku na sljedeći način:

```
string json=JsonSerializer.Serialize(zaposlenik);
string datoteka = "Zaposlenik.json";
File.WriteAllText(datoteka, json);
```

Serijalizaciju možemo i tipizirati, odnosno osigurati da zapisi objekata budu točno određenog podatkovnog tipa. Serijalizaciju bismo napravili na sljedeći način s tipiziranjem:

```
string json=JsonSerializer.Serialize<Zaposlenik>(zaposlenik);
```

Serijalizaciju možemo napraviti nad kolekcijama, ali i nad svojstvima koja su sama po sebi kolekcije. Slijedi primjer serijalizacije i spremanja liste zaposlenika:

```
using System;
using System.Collections.Generic;
using System.Text.Json;
```

```
namespace Aplikacija
```

```
{
```

```
    class Zaposlenik
```

```
    {
```

```
        public string Ime { get; set; }
```

```
        public string Prezime { get; set; }
```

```
        public DateTime DatumRodjenja { get; set; }
```

```
    }
```

```
    class Program
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        List<Zaposlenik> zaposlenici = new List<Zaposlenik>();
```

```
        var zaposlenik = new Zaposlenik
```



```

{
    Ime = "Nikola",
    Prezime = "Tesla",
    DatumRodjenja = new DateTime(1856, 7, 10)
};
zaposlenici.Add(zaposlenik);
zaposlenik = new Zaposlenik
{
    Ime = "Ruđer",
    Prezime = "Bošković",
    DatumRodjenja = new DateTime(1711, 5, 18)
};
zaposlenici.Add(zaposlenik);

string json=JsonSerializer.Serialize(zaposlenici);
string datoteka = "Zaposlenik.json";
File.WriteAllText(datoteka, json);

Console.WriteLine(json);
}
}
}

```

Nekoliko činjenica koje se odnose na ponašanje serijalizacije moramo znati:

- podrazumijeva se da su sva javna svojstva serijalizirana, iako se mogu navesti specifična svojstva koja se mogu ignorirati
- velika slova u JSON-u odgovaraju imenima u .NET-u
- cirkularne reference prepoznaju se i bacaju iznimku. Reference se mogu zadržati, no cirkularne reference moraju se posebno obraditi
- članovi koji nisu svojstva klase ignoriraju se, no mogu se uključiti.

Podržani tipovi za serijalizaciju jesu:

- .NET primitivi mapirani prema JavaScript primitivima poput numeričkih tipova, stringova i booleana
- korisnički definirani CLR objekti
- jednodimenzionalna polja

- kolekcije i rječnici iz sljedećih imenskih prostora:
 - System.Collections
 - System.Collections.Generic
 - System.Collections.Immutable
 - System.Collections.Concurrent
 - System.Collections.Specialized
 - System.Collections.ObjectModel.

Iako se uvijek mogu podržati i drugi podatkovni tipovi, ukoliko nisu u navedenom, potrebno je implementirati poseban konverter.

Deserijalizacija je slična serijalizaciji u načinu korištenja. Uobičajeni način deserijalizacije JSON zapisa jest da se prvo stvori klasa sa svojstvima polja koja predstavljaju jedno ili više JSON svojstava. Zatim, da bi se deserijalizirali string ili datoteka, poziva se JsonSerializer.Deserialize metoda. Za generičko premošćivanje metode prosljeđujemo tip klase koju smo kreirali kao generički parametar. Za negenerička premošćivanja prosljeđujemo tip klase koju smo kreirali kao parametar metode. JSON svojstva koja nisu u klasi ignoriraju se, odnosno neće biti u klasi.

Sljedeći kod pokazuje kako da deserijaliziramo JSON string:

```
using System;
using System.Collections.Generic;
using System.Text.Json;

namespace Aplikacija
{
    class Zaposlenik
    {
        public string Ime { get; set; }
        public string Prezime { get; set; }
        public DateTime DatumRodjenja { get; set; }
    }

    class Program
```

```

{
    public static void Main()
    {
        string json = @"{"Ime":"Nikola",
            "Prezime":"Tesla",
            "DatumRodjenja":"1856-07-10T00:00:00"}";

        Zaposlenik? zaposlenik=JsonSerializer.Deserialize<Zaposlenik>(json);

        Console.WriteLine("Ime: " +zaposlenik.Ime);
        Console.WriteLine("Prezime: " + zaposlenik.Prezime);
        Console.WriteLine("Datum Rođenja: " + zaposlenik.DatumRodjenja.ToShort
            DateString());
    }
}

```

Da bismo deserijalizirali iz datoteke, moramo samo pročitati cijeli string koji se nalazi u datoteci u jednu varijablu, a zatim to koristiti u deserijalizaciji. Slično kao u prijašnjem primjeru, svodi se na to da nemamo „hardkodiran” string u kodu, nego čitamo iz neke datoteke kao u sljedećem primjeru:

```

string datoteka = "Zaposlenik.json";
string json = File.ReadAllText(datoteka);

Zaposlenik? zaposlenik=JsonSerializer.Deserialize<Zaposlenik>(json);

```

Ponašanje deserijalizacije JSON-a:

- poistovjećivanje imena svojstava osjetljivo je na velika i mala slova
- ako JSON sadrži vrijednosti za svojstva koja se mogu samo čitati, vrijednost se ignorira i ne baca se iznimka
- ne javni (public) konstruktori ignorirani su od strane serijalizacije i deserijalizacije
- deserijalizacija na neiteratibilne objekte ili svojstva koji nemaju javni „set” pristupnik je podržana
- enumovi su podržani kao brojevi. Mogu se serijalizirati imena enumova kao stringovi

- članovi klase koji nisu svojstva ignoriraju se
- komentari u JSON-u bacaju iznimku, iako se mogu omogućiti
- maksimalna dubina deserijalizacije je 64.

Slično kao i kod serijalizacije, kod deserijalizacije mogu se implementirati zasebni konverteri, koji mogu pružiti funkcionalnost koja nije uključena u standardu.

Kod deserializacije uvijek bismo trebali navesti u koji objekt deserijaliziramo. Ako prilikom čitanja imamo neku kolekciju, moramo navesti u koju kolekciju čitamo datoteku:

```
string datoteka = "Zaposlenik.json";
string json = File.ReadAllText(datoteka);
zaposlenici = JsonSerializer.Deserialize<List<Zaposlenik>>(json);
```

Naravno najveća upotreba JSON-a rad je s podacima na mreži. Serijalizacija i deserijalizacija normalna je operacija za razmjenu podataka. Proširene metode na HttpClient i HttpContent omogućuju da izvodimo ove operacije u samo jednoj liniji koda.

Sljedeći primjer prikazuje kako da izvodimo ove operacije na testnom lažnom API-ju za testiranje i prototipiranje:

```
using System;
using System.Collections.Generic;
using System.Net.Http.Json;

namespace Aplikacija
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Username { get; set; }
        public string? Email { get; set; }
    }
}
```

class Program

```
{
    public static async Task Main()
    {
        using HttpClient client = new()
        {
            BaseAddress = new Uri("https://jsonplaceholder.typicode.com")
        };

        // Dohvat korisničkih informacija.
        User? user = await client.GetFromJsonAsync<User>("users/1");
        Console.WriteLine("Id: {user?.Id}");
        Console.WriteLine("Name: {user?.Name}");
        Console.WriteLine("Username: {user?.Username}");
        Console.WriteLine("Email: {user?.Email}");

        // Post novog korisnika.
        HttpResponseMessage response = await client.PostAsJsonAsync
            ("users", user);

        Console.WriteLine(
            "{(response.IsSuccessStatusCode ? "Success" : "Error")} - {response.
                StatusCode}");
    }
}
```

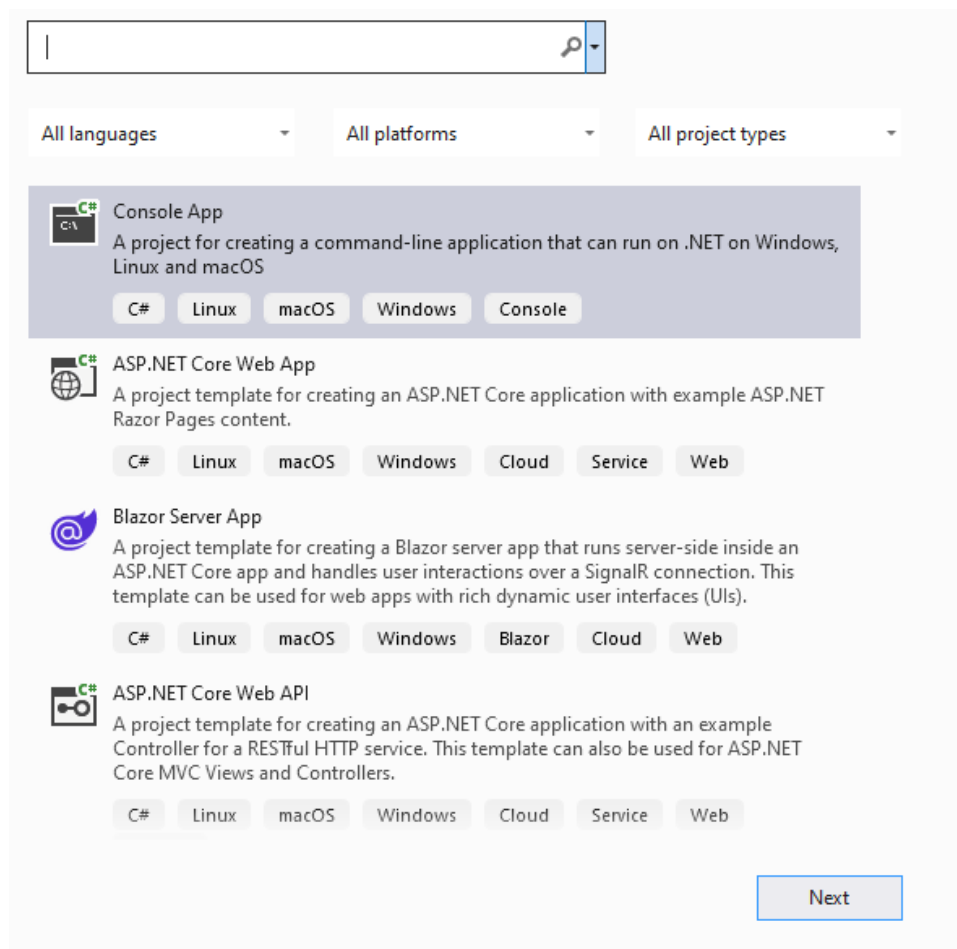
4.7. Interakcija s korisnikom

Pojam interakcije s korisnikom svodi se na to da korisnik koji koristi aplikaciju koju napišemo, mora na što lakši i intuitivniji način koristiti tu aplikaciju. No moramo uzeti i u obzir kojoj je grupi korisnika aplikacija namijenjena i na koji će se način aplikacija koristiti. Npr. nećemo aplikaciju koja treba biti kratka i jednostavna za programera ili sistemskog inženjera opteretiti kompleksnim korisničkim sučeljem, nego to može biti obična konzolna aplikacija kakve smo dosad uglavnom radili. No s druge strane, prosječan korisnik ne želi takvu aplikaciju, nego neku koja će mu biti jednostavna i intuitivna za rad, s estetski prihvatljivim i po mogućnosti što intuitivnijim korisničkim sučeljem, da što manje misli o programu, a što više na ono što treba raditi s programom koji koristi. Još kad moramo uzeti u obzir i za koju je upotrebu to namijenjeno (*web*, *desktop*, *mobitel* ili nešto sasvim drugo), stvari nam se dodatno kompliciraju. Često ćemo se dovesti u situaciju gdje kao backend programeri moramo napraviti neko sučelje, odnosno implementirati obrazac ponašanja aplikacije prema korisniku na temelju nečega što je napravio dizajner, što može biti i obična skica. Ovo je ono što *backend* programera dijeli od takozvanog *frontend* programera. *Backend* programer češće će ući u područje rada *frontend* programera (HTML, JavaScript, CSS...) radi generiranja sadržaja za korisnika s pozadinske strane, no *frontend* programer neće ulaziti u rad *backend* programera, jer *backend* programer obuhvaća potencijalno puno šire područje od *frontend* programera.

Visual Studio i sam .NET odlični su alati jer već sadrže veći broj predložaka na temelju kojih možemo raditi skoro svaku vrstu aplikacije koje poželimo, istim programskim jezikom.

Na temelju toga idemo napraviti pregled nekoliko vrsta aplikacija s obzirom na interakciju s korisnikom.

Svakako prvo napomena, preglede predložaka aplikacija dobivamo u Visual Studiu prilikom dodavanja novog projekta. Dodavanjem novog projekta s praznim filterima dobivamo popis svih mogućih vrsta aplikacija podržanih od strane trenutne instalacije, kao na slici 23.



Slika 23. Predlošci projekata u Visual Studio alatu

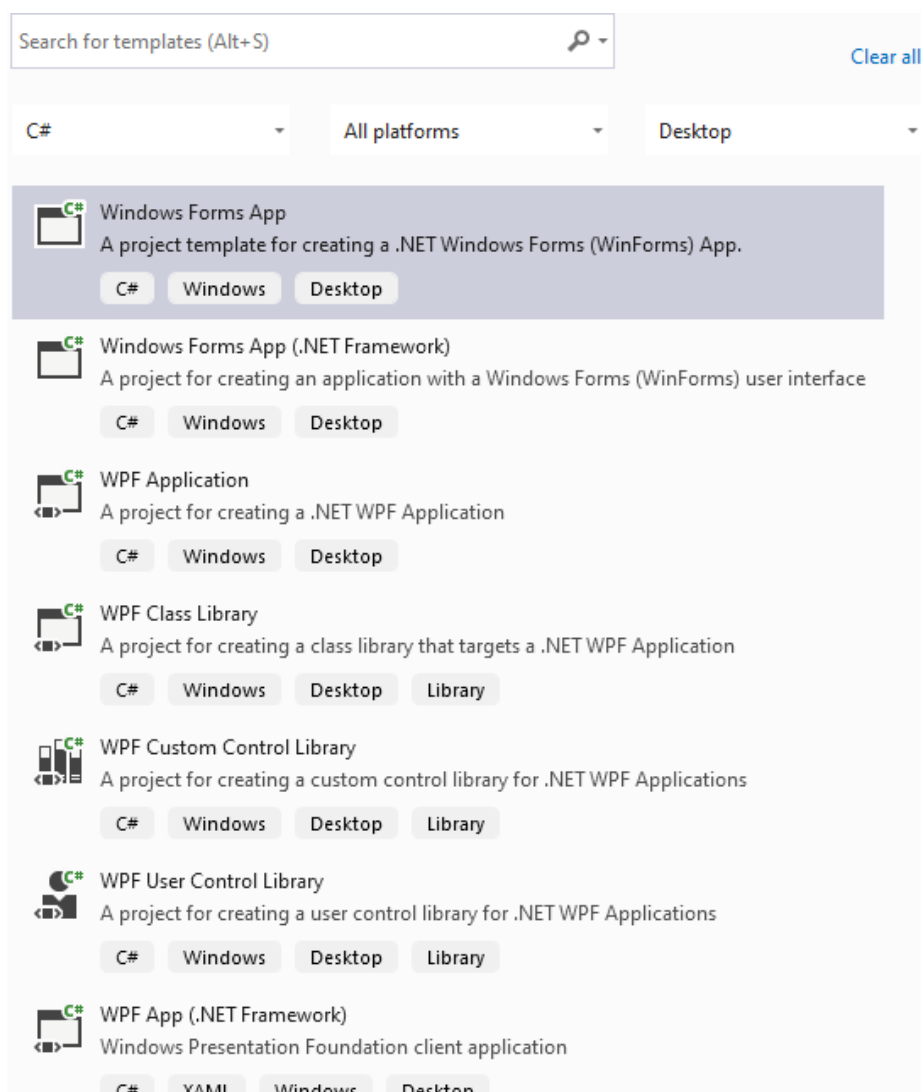
No omogućeno je i dodavanje novih vrsta projekata korištenjem dodataka za Visual Studio i instalacijom dodatnih biblioteka. Npr. instalacija Unityja omogućit će nam da napravimo projekt za izradu videoigara, slično kao da instaliramo Cocos, Ogre3D itd.

4.7.1. Aplikacije s tekstualnim sučeljem

Konzolne aplikacije, ili često nazvane aplikacije naredbenog retka ili aplikacije s tekstualnim sučeljem, nećemo posebno spominjati jer smo dosad radili uglavnom s njima, samo ćemo malo opisati. One se svode na to da kroz aplikaciju u potpunosti navigiramo preko tipkovnice u obliku teksta, no i da aplikacija sve što treba pokazati korisniku daje u obliku teksta. Ovaj tip aplikacija rijetko ćemo vidjeti kod krajnjeg korisnika, odnosno „običan” korisnik u današnje vrijeme aplikacije s tekstualnim sučeljem ni ne koristi.

4.7.2. Desktop sučelja

Aplikacije koje koriste grafičko sučelje u radu na računalu i koje rade na računalu gdje se pokreću zovemo desktop aplikacije. Danas je svaka aplikacija gdje korisnik može vidjeti elemente na prozoru na ekranu desktop aplikacija. Ako prilikom kreiranja projekta u Visual Studiu odaberemo odgovarajuće filtere (C# i desktop aplikacije), dobit ćemo danas najčešće korištene tehnike izrada desktop aplikacija kao na slici 24.

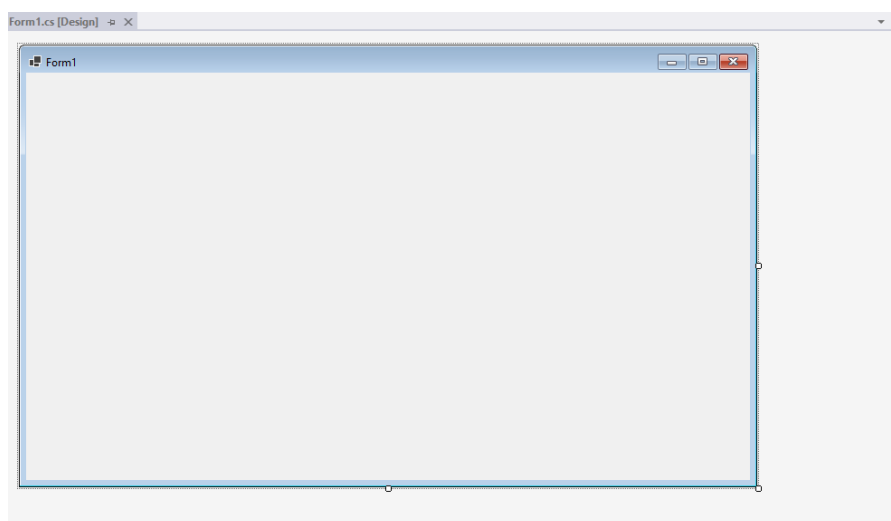


Slika 24. Predlošci izrade desktop aplikacija

Dva najčešća projekta tipa su nam Windows Forms App i WPF Application.

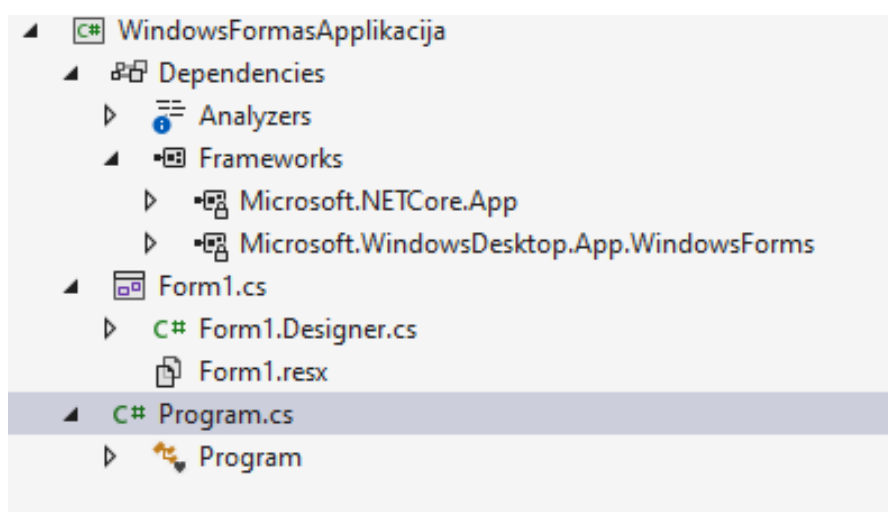
Windows Forms:

Odabirom na „Windows Forms App” otvorit će se još dvije opcije, slično kao i kad smo radili konzolnu aplikaciju. Prvo će nas pitati za ime projekta, a zatim, na prozoru nakon, da odaberemo koju verziju .NET-a želimo koristiti. Kad smo završili s odabirom opcija, novi projekt bit će kreiran, a za početak uvijek će se otvoriti prazan prozor za dodavanje elemenata kao na slici 25.



Slika 25. Windows Forms aplikacija

Unutar projekta dodane su dvije datoteke, kao na slici 26.



Slika 26. Prikaz datoteka novo definiranog Windows Forms projekta

Dependencies nećemo posebno prolaziti, to su nam samo pokazatelji referenci na koje će se biblioteke referirati.

Zanimljiviji je sadržaj Program.cs datoteke:

```
namespace WindowsFormasApplikacija
{
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            // To customize application configuration such as set high DPI settings
            // or default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new Form1());
        }
    }
}
```

Kako komentari na generiranom kodu kažu, ovo je ulazna točka aplikacije, gdje podešavamo inicijalnu konfiguraciju aplikacije. Ovaj dio koda skoro nikada nećemo mijenjati, osim ako ne trebamo da nam pokrene drugu formu umjesto ove automatski generirane (Form1).

No da pogledamo i Form1.Designer.cs :

```
namespace WindowsFormasApplikacija
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

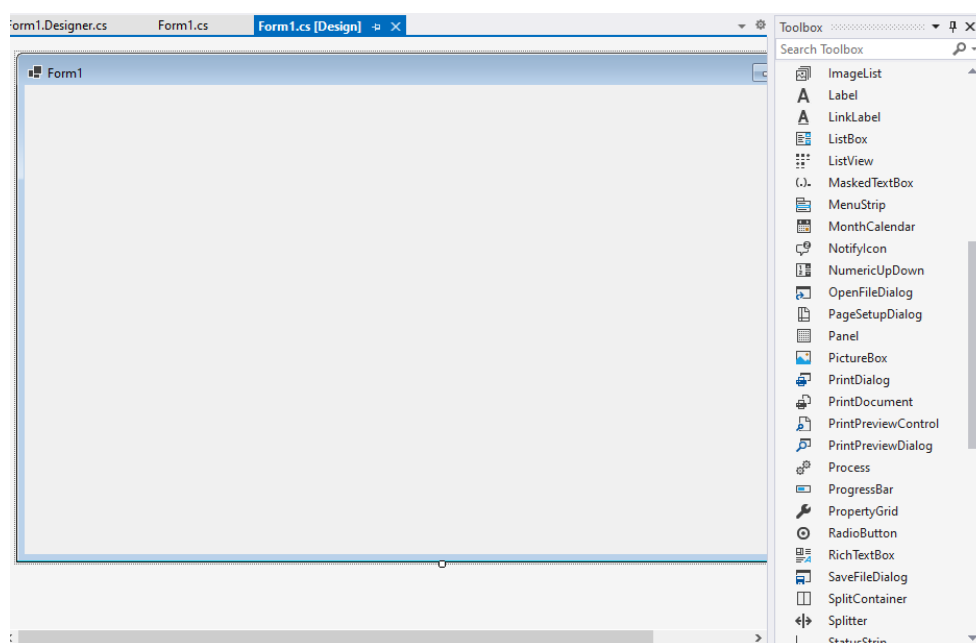
        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(800, 450);
            this.Text = "Form1";
        }

        #endregion
    }
}
```

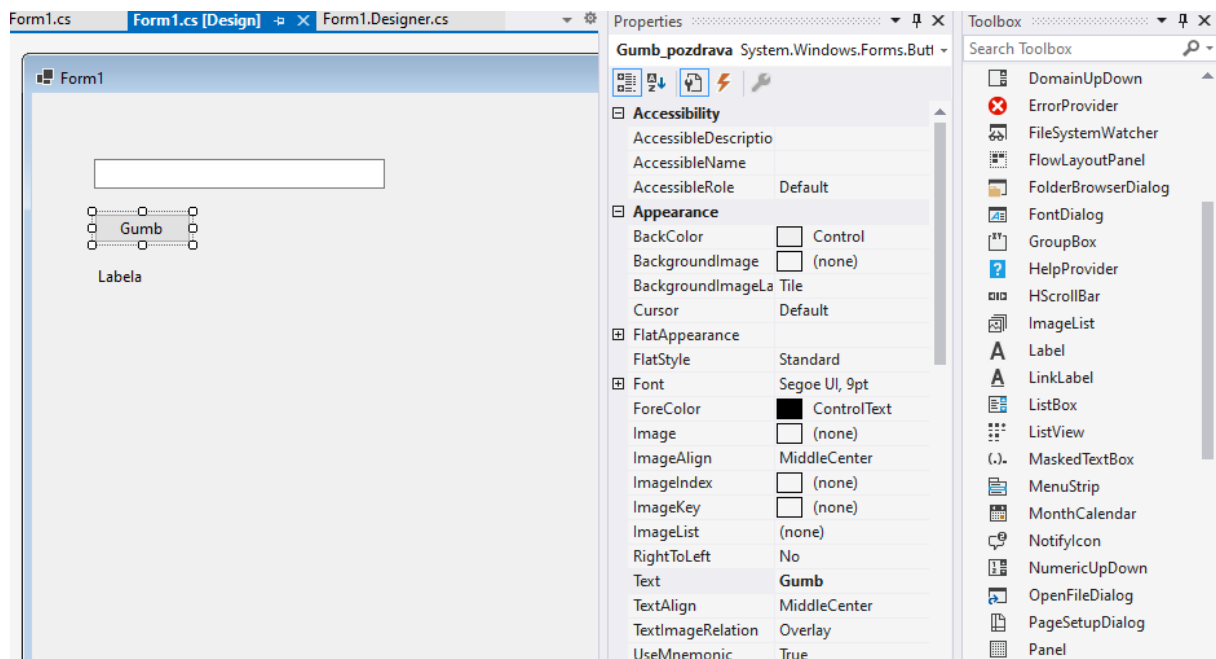
Kao što vidimo, ovo nam je sve automatski generirani C# kod, koji također možemo promijeniti po želji, od imena, teksta, veličine prozora, skaliranje s obzirom na postavke operativnog sustava itd.

Najmoćnija stvar kod Windows forms aplikacija jest to što ne moramo sve sami raditi, nego možemo koristiti „Toolbox” alat u VS-u i jednostavno povlačiti komponente koje nam trebaju na prozor, kao na slici 27.



Slika 27. Pregled Toolbox dijaloškog okvira s Windows Forms

Da demonstriramo rad, idemo napraviti samo osnovno sučelje, gdje ćemo povući kućicu za tekst, jedan gumb i labelu. Za svaki od dodanih elemenata možemo grafički podešavati svojstva. Klikom na element i gledajući prozor svojstava (*Properties*), možemo vidjeti koja sve svojstva imamo za kontrolu, ali uz svojstva i događaje koje možemo definirati nad njima. U ovom primjeru promijenio sam imena kontrola da bude lakše snalaziti se u kodu. Ukratko, ovo je dio koji bi inače dizajner radio, no budući da kod Windows Forms aplikacija korisničko sučelje nije odvojeno od koda koji se radi, teško da bi se netko tko radi dizajn mogao snaći u klasičnom kodu da ga mijenja. Također, zbog tih se limitacija danas ne koriste prečesto ovako klasične aplikacije za razvoj novih, osim u poslovnom sektoru, gdje je uvijek funkcionalnost daleko ispred estetike.



Slika 28. Izgrađeno Windows Forms sučelje

Nakon što smo definirali sve elemente, napravimo dvoklik na Gumb i Visual Studio automatski će nam generirati metodu unutar Form1.cs datoteke. Idemo pogledati taj sadržaj:

```
namespace WindowsFormasApplikacija
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Gumb_pozdrava_Click(object sender, EventArgs e)
        {
        }
    }
}
```

Sadržaj nam pokazuje dosta toga. Kao prvo, vidimo da je Form1 klasa derivirana iz Form klase, koja je sastavni dio .NET-a. U Form klasi imamo sve potrebno za rad s prozorom (okvir, sam prozor, gumbi za manipulaciju, metode za promjenu veličine...). Na nama je da praktički proširimo mogućnosti praznog prozora našim kodom. Dodana metoda jest metoda koja će se pokrenuti na događaj gumba, koji je kao i sve ostale kontrole ubačen u prozor.

Ako sad pogledamo sadržaj Form1.Designer.cs datoteke, vidjet ćemo veći broj promjena:

```
namespace WindowsFormasApplikacija
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed;
        otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
```

```

/// </summary>
private void InitializeComponent()
{
    this.kucica_teksta = new System.Windows.Forms.TextBox();
    this.Gumb_pozdrava = new System.Windows.Forms.Button();
    this.moja_labela = new System.Windows.Forms.Label();
    this.SuspendLayout();
    //
    // kucica_teksta
    //
    this.kucica_teksta.Location = new System.Drawing.Point(48, 51);
    this.kucica_teksta.Name = "kucica_teksta";
    this.kucica_teksta.Size = new System.Drawing.Size(225, 23);
    this.kucica_teksta.TabIndex = 0;
    //
    // Gumb_pozdrava
    //
    this.Gumb_pozdrava.Location = new System.Drawing.Point(48, 93);
    this.Gumb_pozdrava.Name = "Gumb_pozdrava";
    this.Gumb_pozdrava.Size = new System.Drawing.Size(75, 23);
    this.Gumb_pozdrava.TabIndex = 1;
    this.Gumb_pozdrava.Text = "Gumb";
    this.Gumb_pozdrava.UseVisualStyleBackColor = true;
    this.Gumb_pozdrava.Click += new System.EventHandler(this.Gumb_
    pozdrava_Click);
    //
    // moja_labela
    //
    this.moja_labela.AutoSize = true;
    this.moja_labela.Location = new System.Drawing.Point(48, 134);
    this.moja_labela.Name = "moja_labela";
    this.moja_labela.Size = new System.Drawing.Size(41, 15);
    this.moja_labela.TabIndex = 2;
    this.moja_labela.Text = "Labela";
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(7F, 15F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleModeMode.Font;

```

```

        this.ClientSize = new System.Drawing.Size(800, 450);
        this.Controls.Add(this.moja_labela);
        this.Controls.Add(this.Gumb_pozdrava);
        this.Controls.Add(this.kucica_teksta);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
        this.PerformLayout();

    }
    #endregion

    private TextBox kucica_teksta;
    private Button Gumb_pozdrava;
    private Label moja_labela;
}

```

Visual Studio automatski je generirao kod za korištenje kontrola koje smo „dovukli” u prozor. Naravno, ovdje možemo mijenjati svojstva u kodu, ali isto tako smo sve mogli mijenjati preko korisničkog sučelja. Obratite pozornost na Gumb_pozdrav, gdje je kreiran novi objekt tipa Button, sa svojim svojstvima, te je dodana metoda na njegov Click događaj.

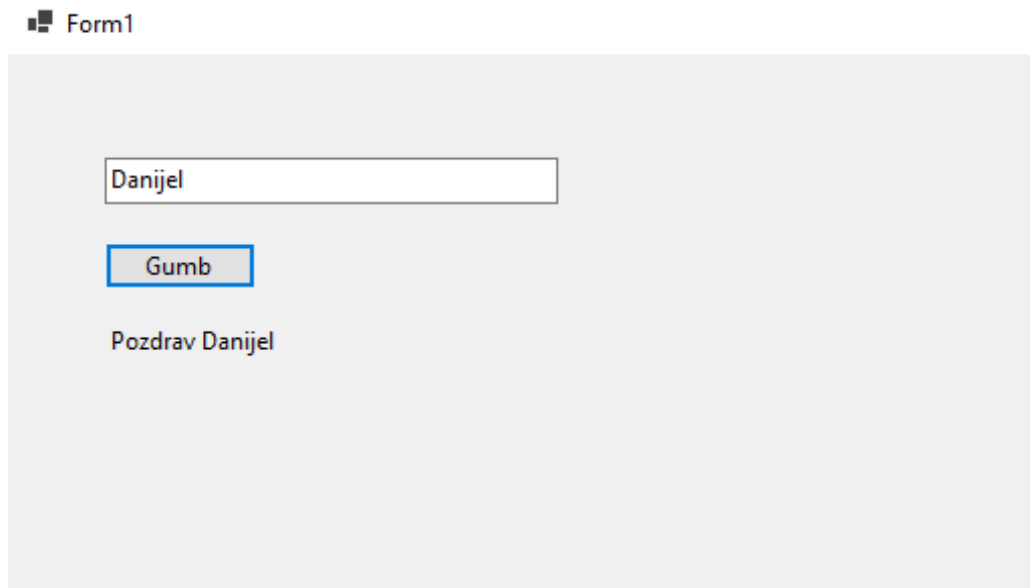
Vratimo se u Form1.cs datoteku i idemo napraviti da nešto radi na klik događaj. Ubacit ćemo jednostavan kod za čitanje iz kućice i u labelu upisivanje pozdrava:

```

private void Gumb_pozdrava_Click(object sender, EventArgs e)
{
    string ime = kucica_teksta.Text;
    moja_labela.Text = "Pozdrav " + ime;
}

```

Pokretanjem aplikacije i upisivanjem imena ispod gumba trebamo dobiti pozdrav s imenom iz kućice kao na slici 28.



Slika 29. Prozor napravljen s Windows Forms u akciji

Napomena samo, Windows Forms aplikacije same su po sebi kompleksna tema te bismo samo na temelju njih mogli raspisati nekoliko stotina stranica oko korištenja. Ovdje smo samo pokazali osnove osnova korištenja, u smislu kreiranja projekta i minimalne neke modifikacije.

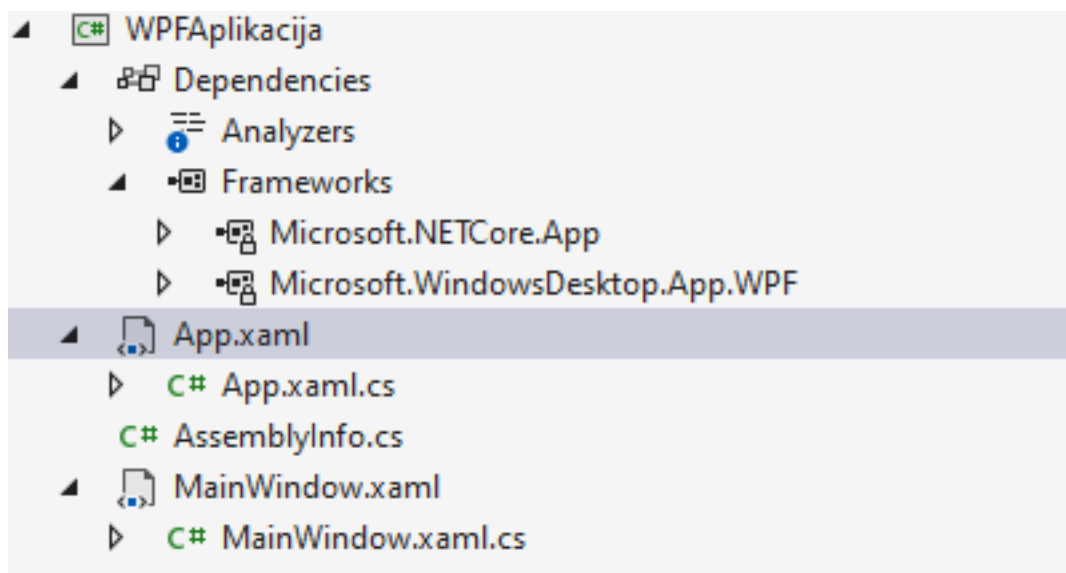
Za one koji žele naučiti više, možete to učiniti na sljedećoj poveznici:

<https://docs.microsoft.com/en-us/dotnet/desktop/winforms/?view=netdesktop-6.0>

WPF:

Odabirom WPF Application, te isto kao i kod formi odabirom imena i verzije .NET-a, generirat će nam se kod za rad s WPF aplikacijom. WPF ili Windows Presentation Foundation *framework* je korisničkog sučelja koji je praktički uvelike proširio mogućnosti rada s desktop aplikacijama. Rađen je da bude neovisan o rezoluciji na kojoj se radi i koristi vektorsko bazirani model iscrtavanja, napravljen da iskoristi moderan grafički *hardware*. Omogućuje veliki broj mogućnosti, što nije bilo moguće s Windows formama poput XAML-a – naprednih kontrola i sučelja, 2D i 3D grafike, animacija, stilova, uzoraka dizajna, dokumenata, multimedijских sadržaja, teksta i tipografije. WPF je dio .NET-a, što znači da možemo raditi aplikacije koje uključuju i druge elemente .NET API-ja.

Zgodna stvar kod WPF-a jest to što praktično odjeljuje korisničko sučelje preko mark-
upa od logike rada, što zovemo *code-behind* (kod iza). To nam je vidljivo u strukturi
koju dobijemo prilikom kreiranja projekta. MainWindow.xaml predstavlja markup, a
MainWindow.xaml.cs predstavlja kod iza. Tako generirana struktura izgleda kao na
slici 30.



slika 30. Struktura WPF aplikacije

Kao i u Windows formama, nećemo prolaziti Dependencies jer je identično kao i kod
formi, samo se druga biblioteka referencira. Zanimljive su nam datoteke App.xaml i
App.xaml.cs, u kojima se nalazi sljedeći sadržaj:

```
<Application x:Class="WPF Aplikacija.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WPF Aplikacija"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

```
using System;
using System.Collections.Generic;
```

```

using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace WPF Aplikacija
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}

```

Prvo što možemo primijetiti jest da je XAML markup praktično XML, ali napomena –uvelike specijaliziran. U App.xaml nalazit će se resursi i konfiguracije kojima možemo pristupiti na razini cijele aplikacije. App.xaml u slučaju WPF aplikacije početna je točka programa, odnosno ovdje kažemo koja će se forma pokrenuti kao početna. To možemo pročitati iz Application čvora, iz svojstva StartupUri.

App.xaml.cs tipičan je C# kod, odnosno ako trebamo, ovdje ćemo implementirati neka generalna svojstva i ponašanje aplikacije, no nećemo ići u previše detalja.

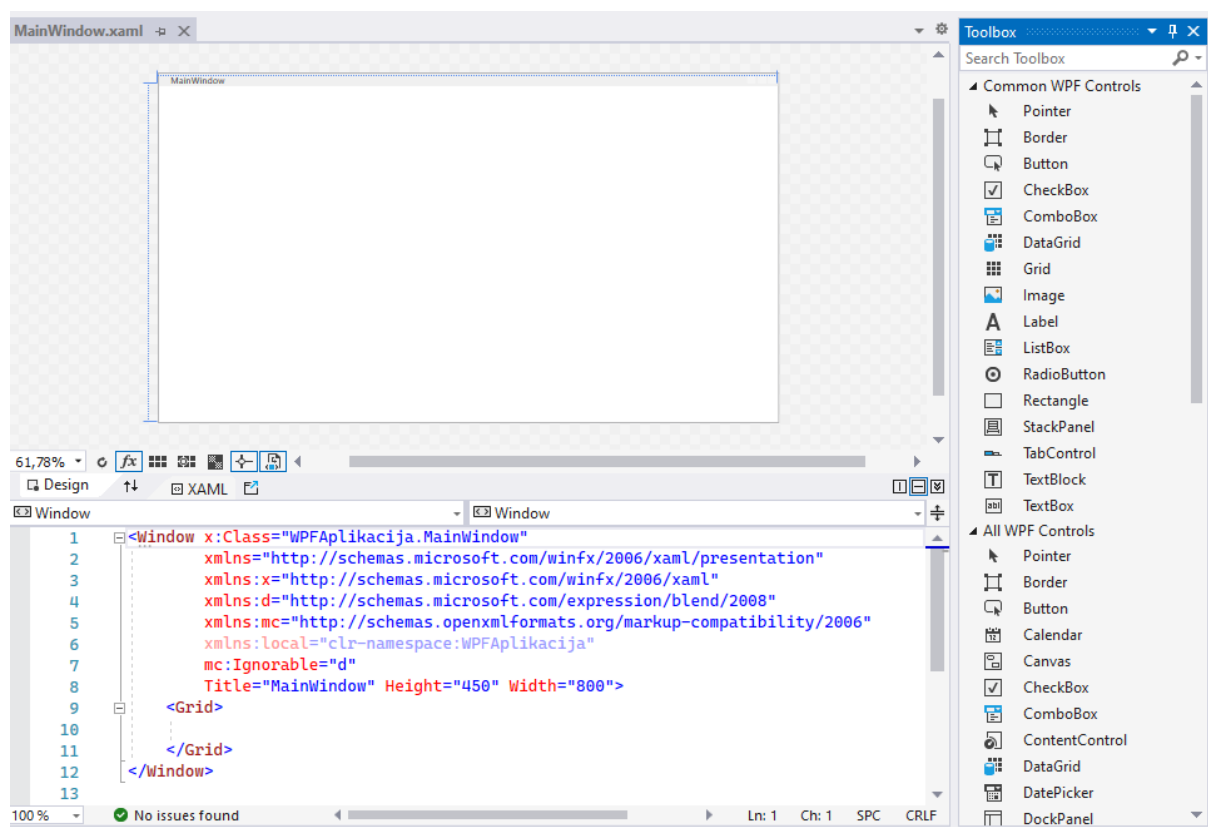
Opet napomena, slično kao na kraju Windows Forms dijela, WPF je ogroman s hrpom mogućnosti, te se samo na temelju toga može napisati nekoliko knjiga/skripti, tako da nećemo ulaziti u neke velike detalje.



Za one koji žele znati više, možete to učiniti na sljedećoj poveznici:

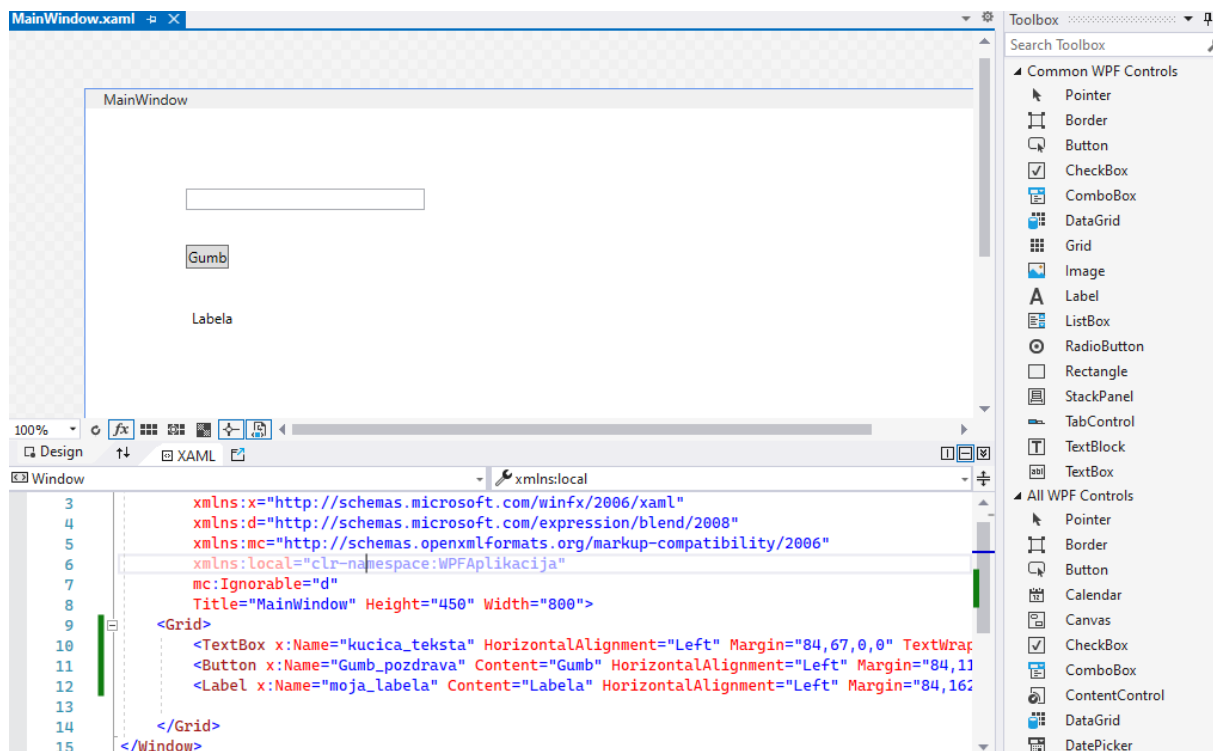
<https://docs.microsoft.com/en-us/dotnet/desktop/wpf/getting-started/?view=netframeworkdesktop-4.8>

MainWindow.xaml i MainWindow.xaml.cs pokazatelji su kako WPF sa svojim forma-
ma izgleda i kako se radi s njime. Kad smo definirali aplikaciju, ono što smo dobili ot-
voren je MainWindow.xaml prozor, koji je u ovom slučaju običan prazan prozor, slično
kao i kad smo definirali Windows Forms aplikaciju. No u samom tom prvom dijelu već
vidimo ogromnu razliku.



Slika 31. Izgled WPF prozora s paralelnim XAML markupom

U WPF-u imamo grafički prikaz i kodni prikaz prozora. Iako je moguće da se prikaže
samo jedno ili drugo, povlačenjem kontrola automatski možemo gledati kako se mi-
jenja i XAML koji je generiran. Preko XAML-a lakše je raditi i izmjene svojstava koja
trebamo. Zato ćemo kao u prijašnjem primjeru ubaciti kućicu za tekst, gumb i labelu.



Slika 32. Dodavanje kontrola u WPF aplikaciju

Ako pogledamo kod koji se generira, vidimo da imamo direktan pristup markupu, odnosno možemo direktno putem grafičkog i tekstualnog sučelja mijenjati pojedine postavke. Ovdje sam promijenio ime komponenti i možemo malo podesiti sadržaj koji se nalazi u pojedinoj kontroli. Dvoklikom na gumb Visual Studio automatski će dodati događaj, ukoliko korisnik klikne na kontrolu, i direktno nas prebaciti na kod u pozadini (MainWindow.xaml.cs), koji će izgledati ovako:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

```

```

using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPF Aplikacija
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Gumb_pozdrava_Click(object sender, RoutedEventArgs e)
        {
        }
    }
}

```

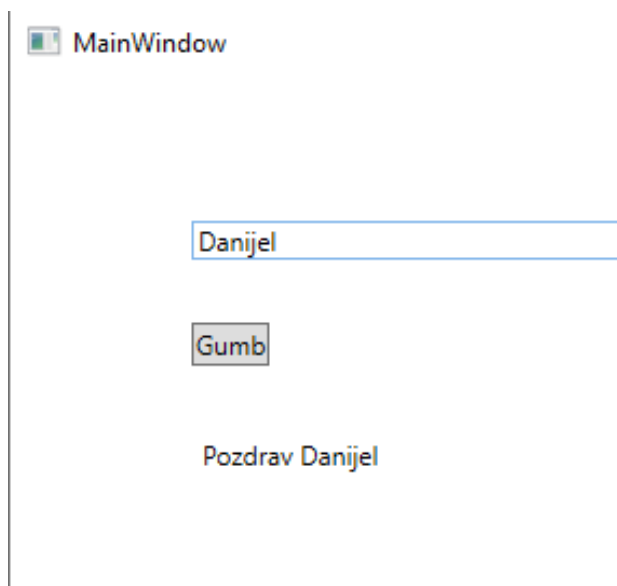
Ako pogledamo klasu, vidjet ćemo sličnosti s Windows formama, u smislu da je naš prozor isto derivirana klasa, ali ovaj put od Window klase. Već smo i prije rekli, ali dobro je ponoviti, dobra stvar ovakvog pristupa jest to što velikim dijelom možemo odvojiti dizajn korisničkog sučelja od pozadinskog koda. Što se tiče dizajna sučelja, čak postoje i alati za sam dizajn, koji na kraju generiraju čisti XAML kod koji možemo koristiti, a najpoznatiji je Microsoft Blend ili kraće samo Blend. No sad idemo nadopuniti kod u našoj metodi slično kao i u primjeru s Windows formama:

```

private void Gumb_pozdrava_Click(object sender, RoutedEventArgs e)
{
    string ime = kucica_teksta.Text;
    moja_labela.Content = "Pozdrav " + ime;
}

```

Rezultat izvođenja aplikacije praktički je isti kao kod forms aplikacije:



Slika 33. Prozor WPF aplikacije

Ako usporedimo s Windows formama, kod u pozadini skoro je identičan, što i jest bit rada s WPF-om. Danas većina novih aplikacija rađenih za Windows napravljene su WPF-om jer je i sam pristup UWP-u (*Universal Windows Platform*) rađen na temelju WPF-a, odnosno XAML markupa.

Ovdje isto vrijedi da nećemo ulaziti detaljno u rad WPF-a, jer to samo po sebi može biti razdvojeno u višestruke knjige, ovisno o tome što točno želite raditi WPF-om.

4.7.3. Web-sučelja

Kad govorimo o *web-sučeljima*, mislimo na rad aplikacija koje se mogu koristiti pomoću preglednika na računalu. Iako ova terminologija može biti malo zavaravajuća, pod pojmom *web-aplikacije* mislimo generalno na aplikaciju kojoj u potpunosti pristupamo preko mreže, bilo lokalne (intraneta) ili globalne (interneta). Svakako, da bismo pristupili aplikacijama s *web-sučeljem*, aplikacija ne može biti samo tako pokrenuta, nego mora biti postavljena na neki *hosting* servis ili na sam server. Lokalno gledano, svako računalo može biti server za hostanje aplikacije, samo je potrebno instalirati ili konfigurirati servis. Na svako računalo s Windows operativnim sustavom možemo

minimalno instalirati IIS, ali takvoj aplikaciji, koja je poslužena na IIS Expressu, možemo pristupiti samo s računala na kojem se nalazimo, pa onda to možemo smatrati lokalnom aplikacijom.

S Visual Studiom imamo nekoliko predložaka projekata za izradu aplikacije s *web*-sučeljem, što nam govori da izbor nije lagan. U .NET-u alati za rad s *web*-sučeljem stavljeni su pod dio koji se zove ASP.NET .

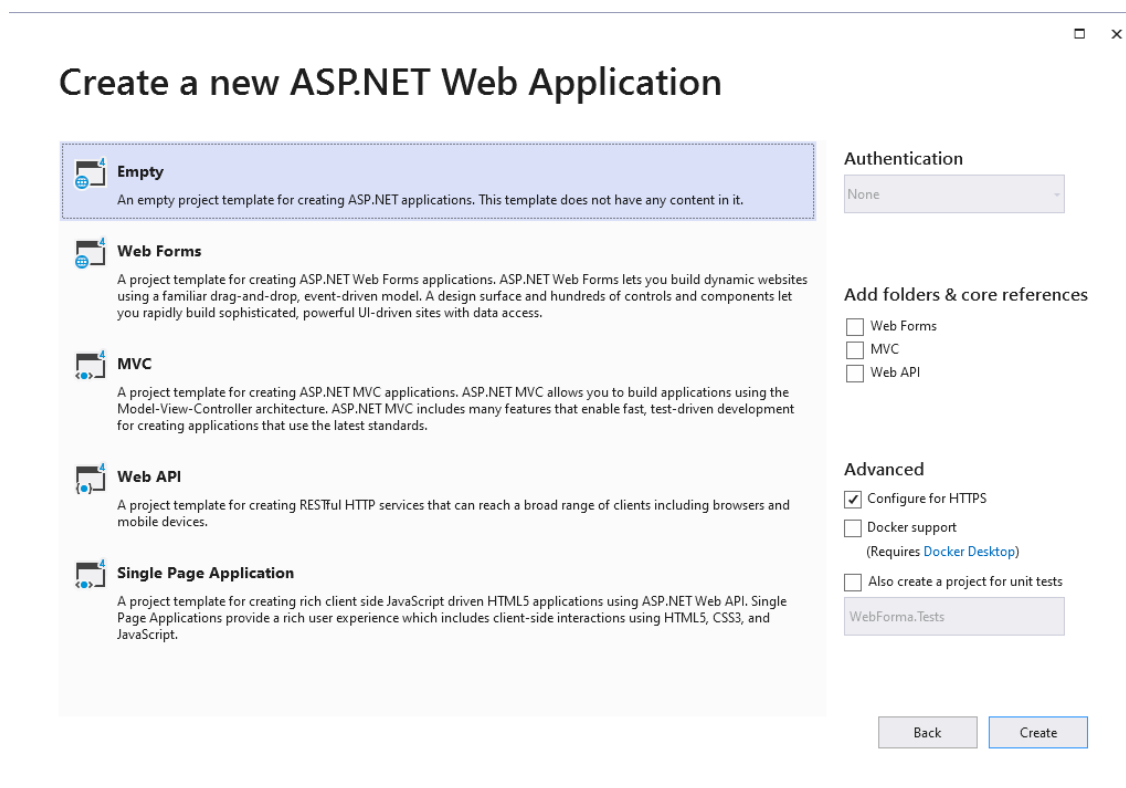
Sam ASP.NET sadrži višestruke predloške za izradu aplikacija s *web*-sučeljem, poput:

- *web*-Forme (Web Forms)
- MVC
- SPA (Single Page Application)
- Core
- Blazor.

Prva tri predloška dobivamo skoro preko istog sučelja, odnosno istog tipa projekta odabirom na ASP.NET Web Application (.NET Framework). Prikazano je na sljedećoj slici. Na slici ćete vidjeti nekoliko opcija, među kojima je najveća opcija za odabir tipa aplikacije, odnosno predloška koji želimo. Svaki predložak ima kratak opis što znači i kako je konfiguriran. Svakako, odabirom na odgovarajući predložak označit će se i pripremiti mape i reference potrebne za rad s predloškom, ali neki predlošci imaju već ugrađenu mogućnost autentifikacija, gdje po potrebi samo odaberemo koji tip autentifikacije želimo te napredne postavke koje su jako korisne – npr. konfiguriranje da se koristi https umjesto http, podrška za Docker za CI/CD operacije i automatsko stvaranje *unit test* projekta povezanog s ovim za testiranje aplikacije.

ASP.NET Core predložak relativno je jednostavan i, ukratko, skoro je identičan MVC predlošku prije spomenutom u radu i razvoju, uz nekoliko bitnih izmjena. Prvo i najbitnije jest to da, budući da se radi o .NET Coreu, podržano je da radi na Linux serverima kad treba objaviti aplikaciju. „Vidljiva” razlika u radu jest način na koji je napravljena konfiguracijska datoteka (`appsettings.json` naspram `web.config`, odnosno JSON naspram XML) i postojanje `wwwroot` mape za organizaciju *frontend* dijela. Ostatak rada praktično je identičan s kontrolerima, modelima i pogledima.

Blazor predložak radi Blazor aplikaciju, koja se izvodi s poslužiteljske strane unutar ASP.NET Core aplikacije. Interakcija se ovdje obavlja preko SignalR veze.

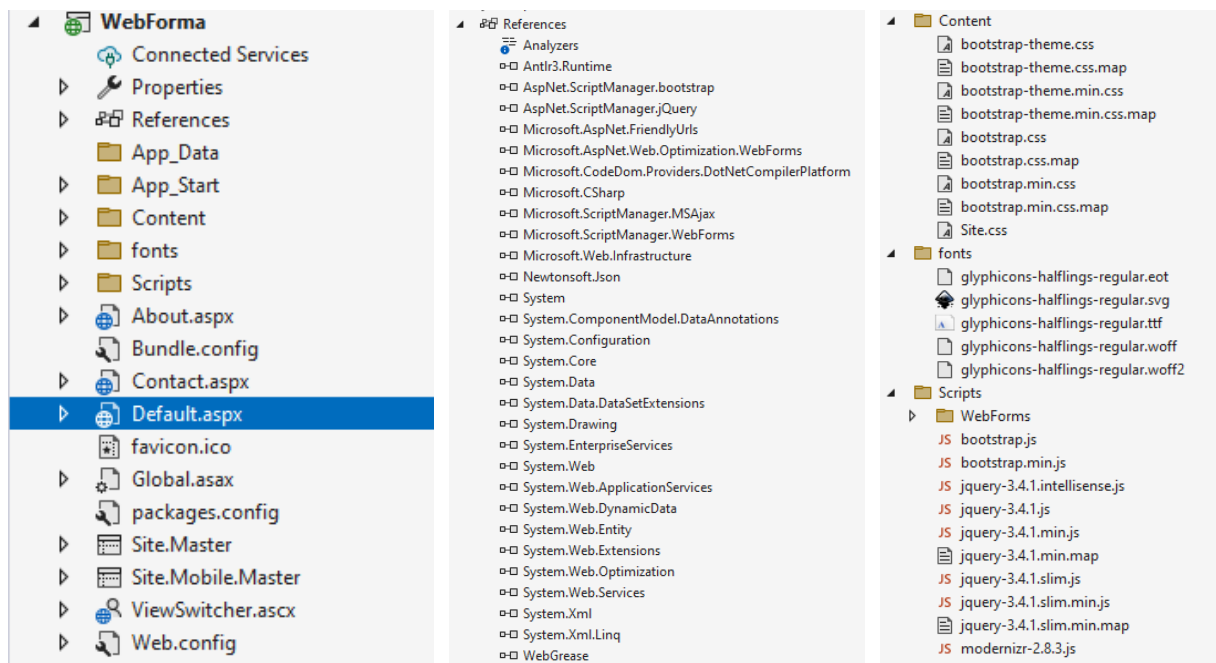


Slika 34. ASP.NET Web Application dijaloški okvir

Web Forme:

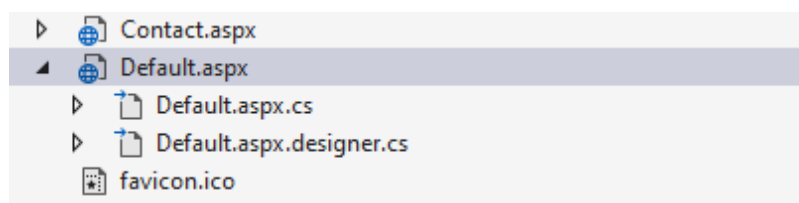
Odabirom Web Formsa s opcijama te nakon toga odabirom verzije **frameworka** Visual Studio pripremit će nam predložak za izradu web-formi.

Web-forme vuku tradiciju od prve verzije .NET-a, odnosno ASP.NET-a još od 2002. godine. Nastale su na temelju starije ASP tehnologije (*Active Server Page*). Koncept nastanka bio je takav da se pokušava prilagoditi već ranije treniranim programerima da na najlakši način prijeđu s razvoja desktop aplikacija na web-aplikacije. Taj je koncept razlog zašto web-forme imaju velik broj kontroli i zašto se sučelje može raditi jednostavnim povlačenjem kontrola iz *toolboxa*, a programiranje je zasnovano na događajima kao kod desktop aplikacija.



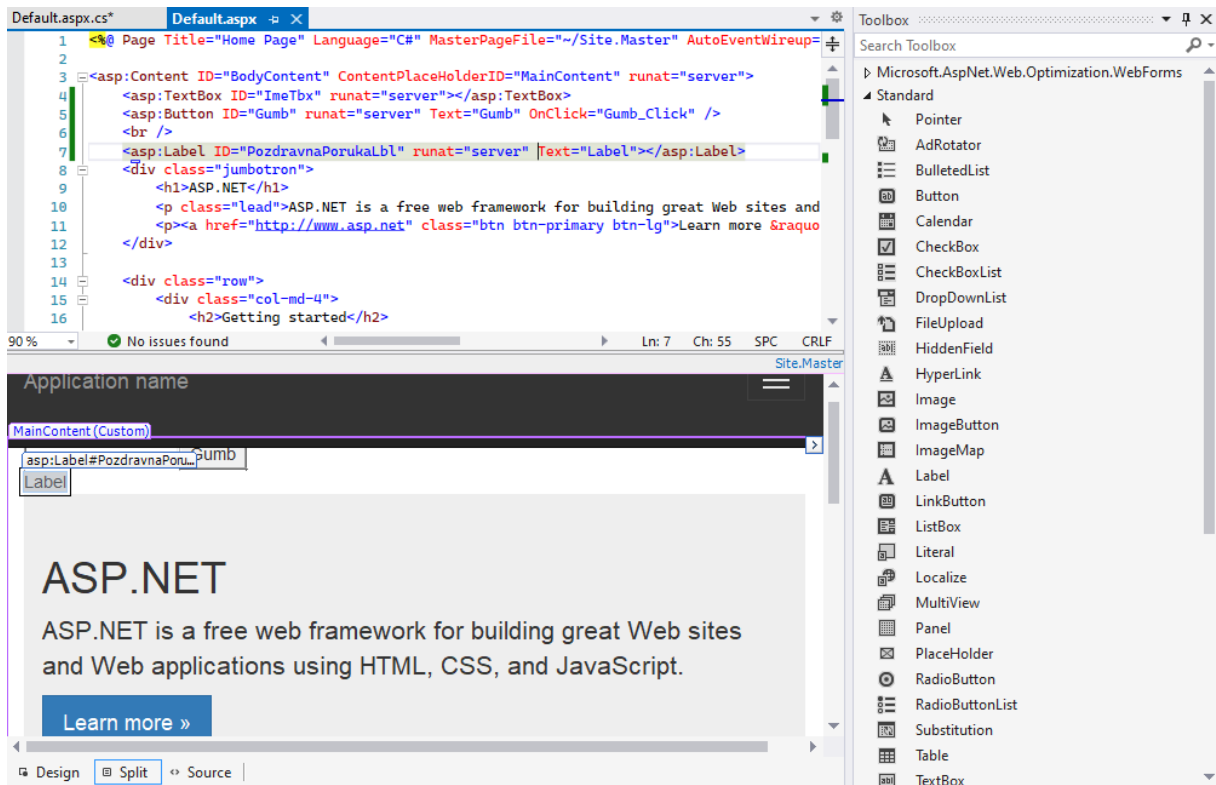
Slika 35. Organizacija Web Forms

Sam predložak napraviti će nekoliko datoteka, odnosno ako odaberemo već pripremljen predložak „Web Forms” umjesto „Empty” i odaberemo Web Forms reference, Visual Studio napraviti će osnovnu hijerarhiju stranica s nekoliko stranica, mape s takozvanim *frontend* dijelom i stiliziranjem.



Slika 36. Web Forms početne datoteke

Organizacija pojedinačnih datoteka sa sučeljem slična je kao i kod Windows forms aplikacija. Svaka stranica sastoji se od .aspx datoteke u kojoj je takozvani frontend dio zajedno s ASP kontrolama prema kojima su vezane još dvije datoteke. Datoteka s ekstenzijom aspx.designer.cs sadrži kodne poveznice na ubačene kontrole, odnosno kontrole s detaljima, slično kao designer.cs kod Windows formi. Datoteka aspx.cs sadrži kod koji će sadržavati što konkretna kontrola treba raditi, odnosno u toj datoteci nalazi se poslovna logika ili logika rada programa.



Slika 37. Izgled rada s Web formama

Ako otvorimo Default.aspx datoteku, vidimo da možemo doslovno samo povlačiti kontrole iz *toolboxa* i pozicionirati ih gdje želimo da budu. Naravno, možemo povlačenje napraviti u kod ili na *designer*. Kasnije ćemo spomenuti zašto pregled u novije vrijeme više ne radi, ali samo se ne dajte zbuniti što trenutno ne izgleda kao što će izgledati kad pokrenemo. Da ostanemo konzistentni s prijašnjim dijelom, ovdje ćemo ubaciti kućicu za tekst, gumb i labelu za ispis poruke. ID svojstva promijenjena su radi lakšeg snalaženja u kodu. Jednom kad dodamo događaj na gumb, implementirat ćemo u pozadini ponašanje na događaj klika. Pozadinski kod izgleda ovako:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

```
namespace WebForma
{
```

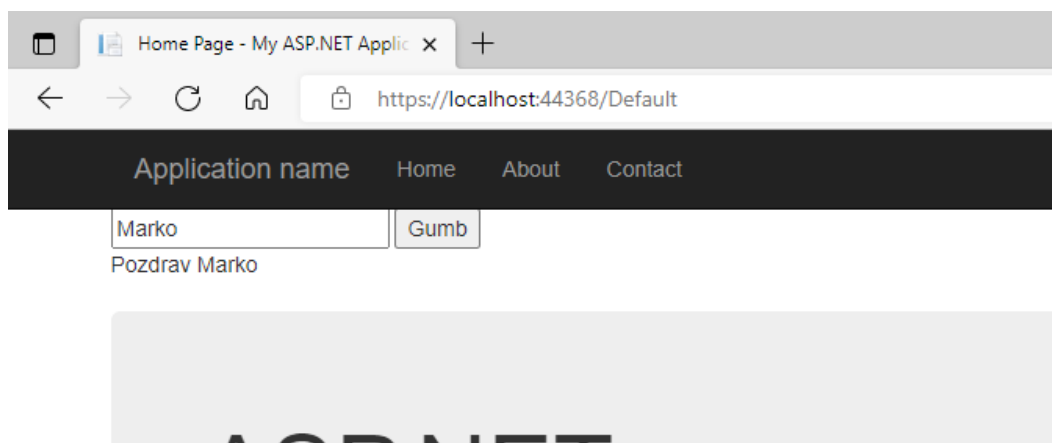
```

public partial class _Default : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

    protected void Gumb_Click(object sender, EventArgs e)
    {
        PozdravnaPorukaLbl.Text = "Pozdrav " + ImeTbx.Text;
    }
}

```

Prilikom pokretanja aplikacije dogodit će se isto što i sa svakom *web*-aplikacijom koju razvijamo, Visual Studio pokrenut će IIS Express, koji će nam djelovati kao privremeni poslužitelj za našu *web*-aplikaciju. Ovo nam omogućuje da u potpunosti testiramo kako će aplikacija raditi, a budući da je IIS Express lokalni servis, možemo testirati s više internetskih preglednika istovremeno. Rezultat izgleda kao na slici 38.



Slika 38. Rezultat izvođenja Web Forms aplikacije

Koncept rada za vrijeme nastanka bio je odličan, no s vremenom se uspostavilo da ima nekoliko problema u radu i konceptu. Svakako treba napomenuti da je Web Forms koncept odlično prihvaćen od strane programera, no nedostaci su sljedeći:

- budući da obrasci skrivaju html i http(https) gdje god je moguće, u specifičnim slučajevima to dovodi do nemogućnosti generiranja željenog html-a. Ovo nije predstavljalo veliki problem samim *backend* programerima, no za izradu sučelja bilo je loše
- velika količina podataka prenosila se preko mreže, pa ako imate mnogo korisnika, to je veliko opterećenje za poslužitelj. ASP kontrole izvode se isključivo na serveru, što je vidljivo po runat= "server" atributu same kontrole. Svaka promjena na stranici znači da se cijela stranica sa svim podacima mora poslati na server i server vrati novo stanje
- nije lagano podijeliti aplikaciju na poslovnu logiku i prezentacijski dio. Iako postoji podjela na prezentacijski dio (.aspx) i kod u pozadini (.aspx.cs), ona nije dovoljno prisutna. Pod ovim mislimo na to da *backend* programer mora dobar dio koda napraviti u prezentacijskom dijelu, a takozvani *frontend* programer ne može napraviti čisto prezentacijski dio, odnosno html, pogotovo uzimajući u obzir da kontrole ne mogu uvijek generirati željeni html
- konceptualno *web*-obraci nisu razvijeni da omoguće visok stupanj testiranja.

Svakako i za *web*-forme vrijedi što smo spominjali za druge vrste aplikacija, rad s *web*-formama možemo razdvojiti u višestruke knjige, ovisno što i kako želimo s njima raditi. Zbog gore navedenih nedostataka, danas se sve manje radi na *web*-formama u globalu, osim za intranetske aplikacije. No ove nedostatke rješava ASP.NET MVC.



Za one koji žele znati više o *web*-formama:

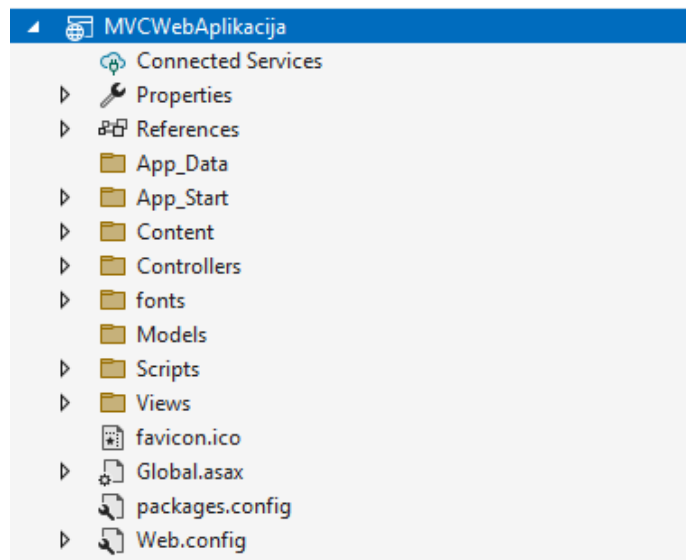
<https://docs.microsoft.com/en-us/aspnet/web-forms/>

MVC:

Prvo da napomenemo da je MVC predložak dizajna koji ćemo kasnije detaljnije objasniti. MVC je kratica od Model View Controller (model pogled upravljač) i predstavlja predložak dizajna za koji možemo reći da danas načinom rada dominira u razvoju kompleksnih *web*-aplikacija zbog svoje usklađenosti s osnovnim principima funkcioniranja *web*-aplikacija.

MVC razdvaja podatke, programsku logiku i korisničko sučelje, čime olakšava održavanje, razvoj i testiranje aplikacija.

No da prijedemo na konkretno, predložak za izradu MVC aplikacije dat će nam strukturu kao na slici 39.



Slika 39. Struktura ASP.NET MVC aplikacije

U strukturi vidimo razdvojenost o kojoj smo govorili po mapama. Model, Views, Controllers sadrži MVC dio, dok je Content, fonts i Script dio čisto vezan za *frontend* u obliku Javascripta, CSS-a i ostalog sadržaja potrebnog za korisničko sučelje.

Osnovni predložak imat će već definiran HomeController.cs, koji predstavlja upravljački dio aplikacije. Sadržaj je sljedeći:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVCWebAplikacija.Controllers
{
```

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult About()
    {
        ViewBag.Message = "Your application description page.";

        return View();
    }

    public ActionResult Contact()
    {
        ViewBag.Message = "Your contact page.";

        return View();
    }
}

```

Ukratko, u kontroleru svaki zahtjev gledamo kao akciju, gdje zahtjev obrađujemo ovisno o tipu i vrsti zahtjeva. Ovdje ćemo se ograničiti samo na to da malo modificiramo index, odnosno početnu stranicu aplikacije. Pogled (*view*) osnovne stranice naći ćemo u Views mapi, pa zatim u podmapi imena kontrolera (*Home*) i ovdje ćemo naći Index.cshtml datoteku. U njoj se nalazi sadržaj koji odgovara naslovnoj stranici. Primijetiti ćete jednu stvar, a to je da u MVC aplikacijama nemamo dizajnerski pogled i nećemo ga ni imati. Bit je ovdje da razdvojimo prezentacijski dio (html, css itd.) od ostatka.

Zbog tog razdvajanja rad s kontrolama nije tako jednostavan, odnosno, bolje reći, nije direktan kao kad radimo s *web-formama*. Kod *web-formi* kad ubacimo kontrole, http zahtjevi skriveni su, no ovdje u MVC-u mi ih moramo obraditi. Zato nećemo previše prolaziti kroz sam kod, nego ćemo ubaciti u Index.cshtml na početak nekoliko linija koda i obrisati automatski generirano, tako da će sadržaj izgledati ovako:

```
@{
    ViewBag.Title = "Home Page";
}

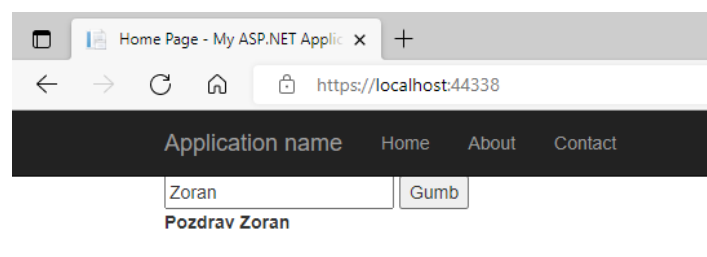
@using (Html.BeginForm("Index", "Home"))
{
    @Html.TextBox("Ime")
    <input id="Submit" type="submit" value="Gumb" />
}
@Html.Label("poruka",(string)ViewBag.Poruka)
```

Napomena za pogled, ne mora biti čisti html, iako može biti, u MVC aplikacijama možemo koristiti Razor sintaksu, koja je skoro identična C#-u, s nekoliko sitnih izmjena. Bitno kod Razer sintakse jest da kad koristimo takozvane htmlhelper metode (html pomoćnike), te se naredbe interpretiraju u čisti html.

Da bismo obradili što će se napraviti u kontroleru, trebamo samo definirati akciju koja će obraditi http zahtjev. To ćemo napraviti dodavanjem nove metode u kontroler klasu, no moramo metodi dati atribut – koju će vrstu zahtjeva odraditi. U kontroler dodajemo sljedeći kod, praktički samo preopterećujući Index metodu:

```
[HttpPost]
public ActionResult Index(string Ime)
{
    ViewBag.Poruka = "Pozdrav " + Ime;
    return View();
}
```

Nakon izvođenja rezultat će biti kao na slici 40.



Slika 40. Rezultat izvođenja MVC aplikacije

Ukratko, praktički identično rezultatu *web-formi*, što i jest bit jer korisnik na kraju dobiva čisti html sa svoje strane, koji se prezentira njemu u pregledniku.

Svakako napomena, po već nekoliko puta slična, ASP.NET MVC isto tako može biti raspisan na nekoliko knjiga, jer ovo je toliko kompleksna i detaljan tema, od rada s kontrolerima, pogledima, modelima, autentifikacija, autorizacija, testiranje, predlošcima dizajna sučelja, parcijalnim pogledima, pomoćnim metodama itd. Ukratko, samo za ASP.NET MVC ima mnogo materijala, pogotovo kad uzmemo u obzir koliko se često danas koristi, a upotreba i raste.



Za one koji žele znati više o MVC aplikacijama i kako ih razvijati:

<https://docs.microsoft.com/en-us/aspnet/mvc/>

SPA:

SPA je skraćeno *Single Page Application* ili *aplikacija jedne stranice*. Ovdje se nećemo raspisati. Ukratko rečeno, ovo je aplikacija koja se jednom učita na korisničkoj stranici, a zatim bilo kakva interakcija, odnosno komunikacija, ide preko nekog od Javascript *frameworka* prema pozadinskim servisima.

U Visual Studio predlošku ovo se svodi na to da se napravi MVC aplikacija s jednim kontrolerom i jednim pogledom. No odabirom predloška slože se i WEB API reference, a preko WEB API-ja slažemo pozadinski REST servis. Aplikacija JavaScriptom komunicira s nekim servisom u pozadini (bilo da ga mi napravimo ili pristupamo nekom vanjskom) i radi izmjene u html kodu na korisničkoj strani. Bitno je reći na korisničkoj strani, te je u tome glavna prednost SPA aplikacija – poslužitelj aplikacije radi minimalno, a većina se odrađuje s korisničke strane. Napomena je da to nekada i nije idealno, jer doživljaj korištenja aplikacije skoro je isključivo ovisan o pregledniku i uređaju na kojem korisnik pristupa aplikaciji.

Kod SPA aplikacije učitavanje aplikacije samo je tijekom prvog pristupa dugo, odnosno treba vremena da se s korisničke strane učita sav potreban JavaScript, odnosno svi potrebni dijelovi rada odjednom se prebacuju na korisnički uređaj.

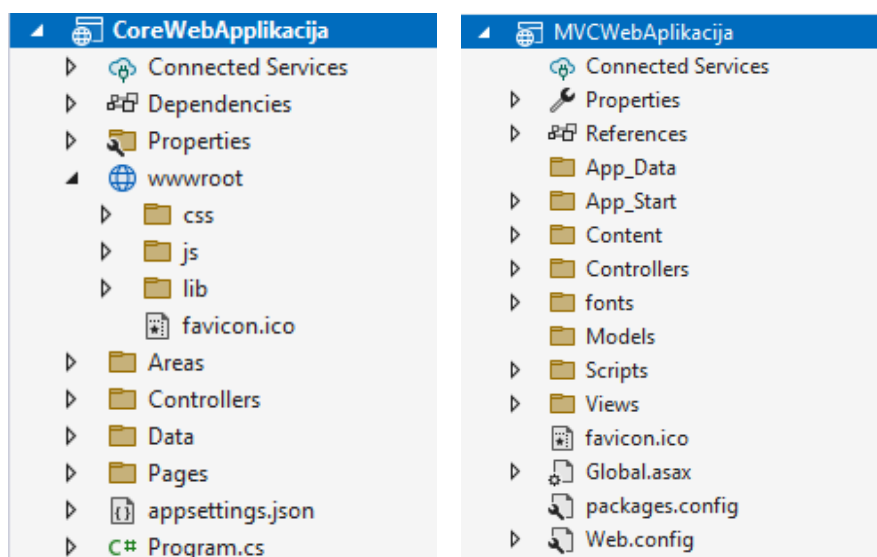
Primjer nećemo demonstrirati jer se svodi na prijašnji MVC, s tim da sa strane pogleda imamo JavaScript kod (koji je *frontend* dio) za upravljanje kontrolama u kombinaciji s *web-servisom*, a WEB API proći ćemo u zasebnom poglavlju.

Core:

ASP.NET Core Web App skoro je identičan ASP.NET MVC-u uz nekoliko sitnih razlika, koje ćemo ovdje samo brzinski preletjeti.

Prvo i najbitnije jest podržanost poslužitelja za posluživanje *web-aplikacije*. „Klasični” MVC izvršava se isključivo na Windows serveru, odnosno obavezno s .NET Frameworkom. ASP.NET Core fokusiran je na to da bude „cross platform”, odnosno da ne bude ovisan o tome da bude samo na Windows operativnom sustavu, nego se nativno može posluživati na Linux, Mac i drugim operativnim sustavima.

Manje bitne razlike u organizaciji su datoteka, što ćemo usporediti.



Slika 41. Usporedba ASP.NET Core i ASP.NET MVC aplikacije

Ako pogledamo strukturu, razlike su minimalne. U razvoju kontroleri nam ostaju u kontroler mapi, umjesto Model imamo Data mapu, umjesto Views imamo Pages mapu. Sljedeća razlika jest to što umjesto više mapa kod MVC aplikacije u Core aplikaciji sav frontend dio ubačen je u wwwroot mapu. Zatim umjesto Web.config konfiguracijske

datoteke u XML formatu imamo appsettings.json konfiguracijsku datoteku u JSON formatu. Ima još nekih sitnih razlika u načinu konfiguriranja projekta, no u ovim razmatranjima zanemarivo.

Prikaz koda i rada koda nećemo prolaziti jer je doslovno identičan klasičnom MVC-u u kodu koji se koristi. To je razlog zašto je lagano napraviti migraciju MVC aplikacije na Core aplikaciju ukoliko se prelazi na Linux server s Windows servera. Jedino tijekom kompajliranja koda vidjet će se koristi li aplikacija možda neke biblioteke iz .NET Frameworka koje nisu podržane na .NET Core dijelu.

Opet istu napomenu možemo reći kao i u svakom dijelu, ovo je prekomplikirana tema da bi se ovdje raspisalo kako raditi ASP.NET Core aplikacije; ukoliko vas zanima, za to ćete trebati čitati zasebne knjige ili materijale.



Za one koji žele znati više:

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-6.0&tabs=visual-studio>

Blazor:

Blazor nam dozvoljava da radimo interaktivne *web*-aplikacije korištenjem C#-a umjesto JavaScripta. Komponente se implementiraju korištenjem C#-a, HTML-a i CSS-a. Naspram prije navedenih projekata Blazorom klijentski i serverski kod pišemo C#-om, što nam omogućuje da dijelimo kod i biblioteke u svim dijelovima aplikacije.

U pozadini Blazor radi na temelju ASP.NET Core aplikacije i korisnička interakcija izvodi se preko SignalR veze. Sama aplikacija može izvoditi klijentski C# kod direktno u pregledniku koristeći WebAssembly. Time Blazore može izvršavati klijentsku logiku direktno na serveru. Klijentski ulazno-izlazni događaji šalju se na server korištenjem SignalR-a. Jednom kad se izvršavanje završi, potrebne promjene u korisničkom sučelju šalju se klijentu i sjedinjavaju se s dokumentom (stranicom) koji se već nalazi kod klijenta.

Blazor koristi otvorene *web*-standarde bez dodatnih dodataka i radi na svim modernim preglednicima, uključujući i mobilne. Kod koji se izvršava u pregledniku izvršava se jednako sigurno kao i JavaScript kod (*sandbox security*). Kod koji se izvršava na serverskoj strani ima fleksibilnost da radi sve što bi i inače serverski kod trebao izvršavati, poput pristupa datotečnom sustavu, pristupa bazama podataka itd.

Prednost Balazora u tome je što u jednom može raditi i pozive prema JavaScript API-ju i prema bibliotekama. To nam omogućuje da u određenim situacijama kombiniramo JavaScript kod koji već postoji i novi kod rađen u C#-u.

Sam predložak projekta koji Visual Studio ima dobro demonstrira kako koristiti i raditi s Blazorom. Predložak je praktički identičan u konfiguraciji ASP.NET Core aplikaciji, samo s malo promijenjenim sadržajem pogleda. U datoteci Counter.razor u mapi Pages možemo vidjeti kako se C# kod integrira uz html kod:

```
@page "/counter"
```

```
<PageTitle>Counter</PageTitle>
```

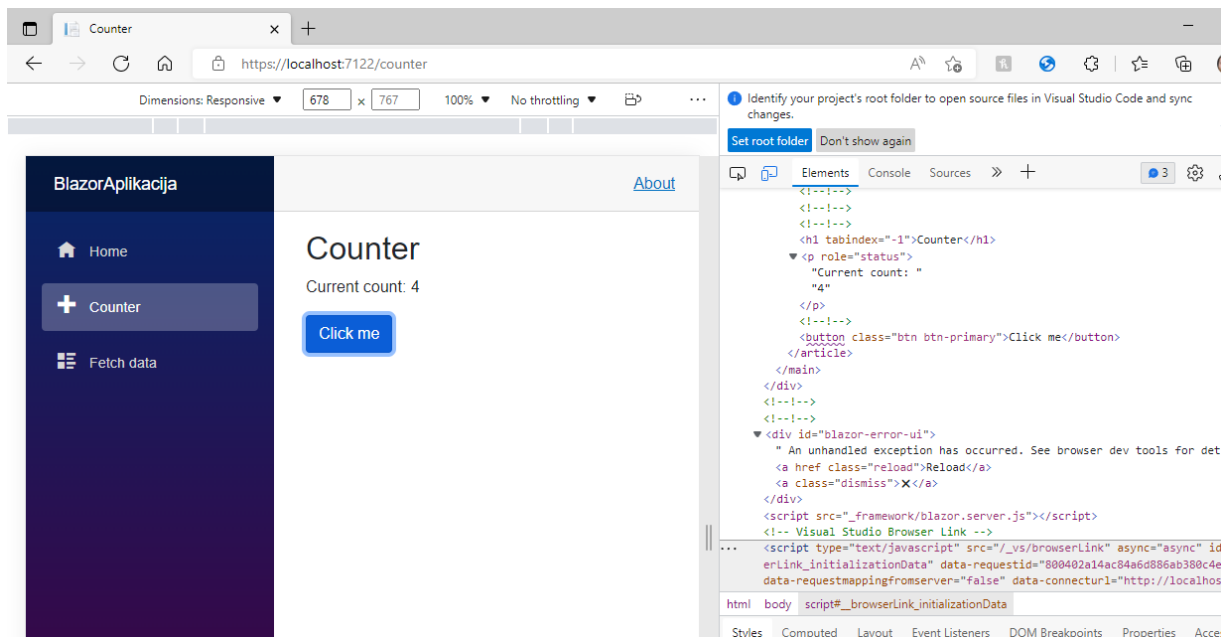
```
<h1>Counter</h1>
```

```
<p role="status">Current count: @currentCount</p>
```

```
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

```
@code {  
    private int currentCount = 0;  
  
    private void IncrementCount()  
    {  
        currentCount++;  
    }  
}
```

Kad pokrenemo i otvorimo counter, izgledat će kao na slici 42.



Slika 42. Blazor Counter primjer

Napomena, dev alate za kontrolu stranice, pregled HTML-a i ostalo možete dobiti jednostavnim pritiskom na F12 u skoro svakom pregledniku.

No da se vratimo na temu, ovdje vidimo kako izgleda kod, da se prikazuje čisti HTML, ali zahvaljujući SignalR-u, nemamo eksponirani klijentski kod. Pritiskom na „Click Me” možemo vidjeti kako se u realnom vremenu (u našem slučaju trenutno) akcija koju smo napravili integrira u dokument, bez ponovnog učitavanja cijelog dokumenta, odnosno ponovnog poziva na kompletnu stranicu.

Svakako za kraj opet ista napomena: Blazor, slično kao i ostalo prije ovoga, nije tako kratak, te bi se i o njemu moglo napisati jako puno materijala.



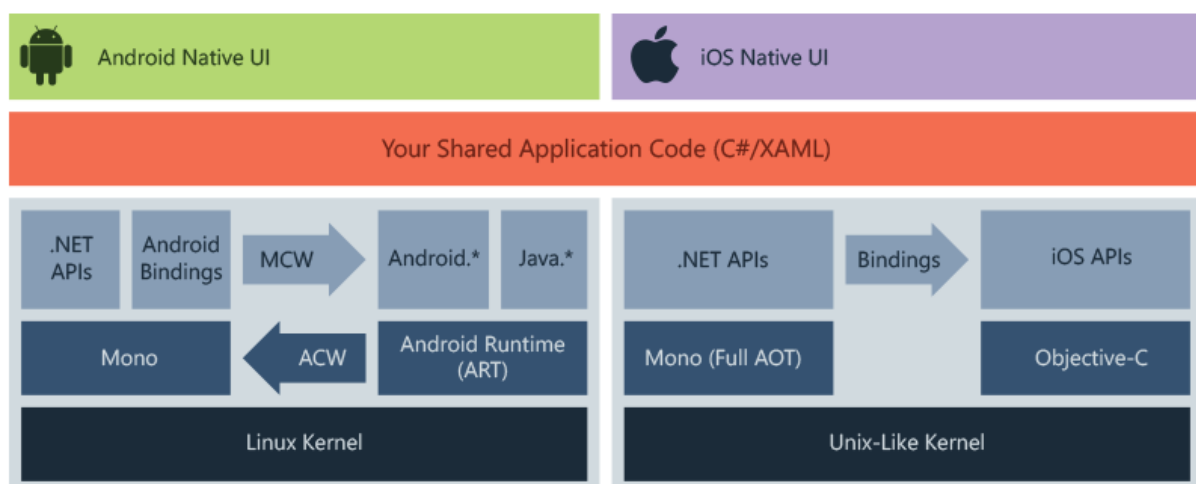
Za one koji žele znati više:

<https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0>

Xamarin:

Xamarin je *open source* platforma za izradu modernih aplikacija za iOS, Android i Windows s .NET-om. Xamarin je apstraktni sloj koji upravlja komunikacijom između dijeljenog koda i platforme na kojoj se program izvršava. Xamarin omogućuje programerima da dijele u prosjeku 90 % aplikacije na više platformi. Ovo omogućuje programerima da pišu logiku rada programa u jednom programskom jeziku, a imaju performanse, izgled i osjećaj kao da je pisano na nativnoj platformi.

Sljedeći dijagram pokazuje arhitekturu *cross-platform* Xamarin aplikacije. Xamarin nam omogućuje da stvorimo nativno korisničko sučelje za svaku platformu, a pišemo logiku rada programa u C# programskom jeziku. Logika rada dijeljena je na svim platformama.



Slika 43. Xamarin arhitektura

Xamarin kombinira mogućnosti nativnih platformi, ali dodaje i mogućnosti koje uključuju:

- potpuno vezanje za SDK koji je u pozadini svake platforme
- Objective-C, Java, C i C++ interoperabilnost
- moderne jezične konstrukcije
- robusna baza klasnih biblioteka
- moderni IDE
- podrška za *cross-platform* razvoj aplikacija.

Xamarin.Android aplikacije kompajliraju se iz C# koda u Intermediate Language (IL), koji je zatim Just-In-Time (JIT) kompajlira u nativni kod kad se aplikacija pokrene. Xamarin.Android aplikacije izvode se u Mono izvršnoj okolini uz Android Runtime (ART) virtualni stroj. Mono izvršno okruženje koristi Managed Collable Wrappers (MCW) i pruža Android Collable Wrappers (ACW) ART-u, što omogućuje obama okruženjima da pozovu kod jedno drugoga.

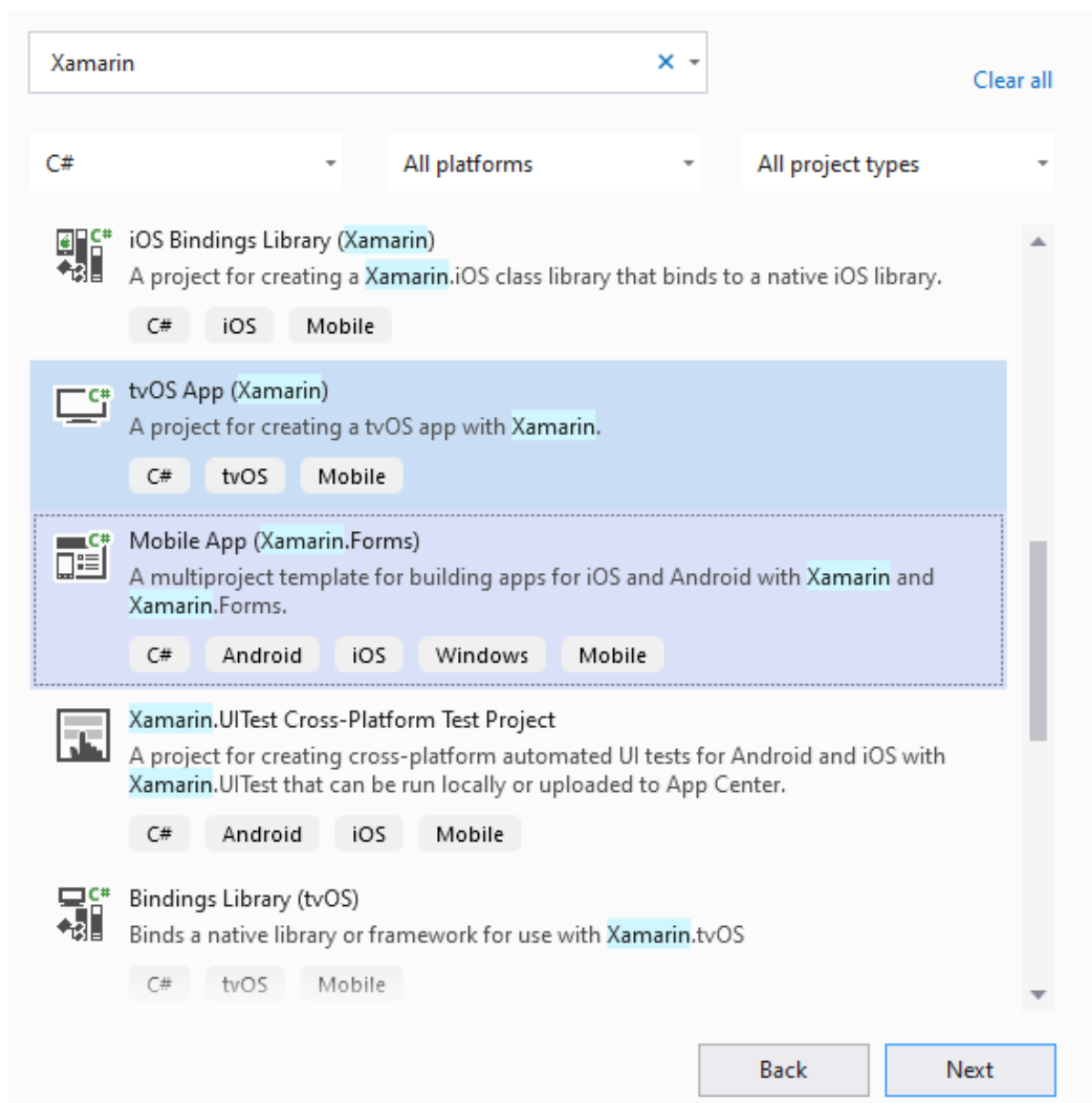
Xamarin.iOS aplikacije u potpunosti su Ahead-of-Time (AOT) kompajlirane iz C# koda u nativni ARM assembly kod. Xamarin koristi Selectors da eksponira Objective-C u upravljani C# i Registrars da eksponira upravljani C# kod u Objective-C kod. Selectors i Registrars skupno se zovi „bindings” i omogućuju da Objective-C i C# međusobno komuniciraju.

Xamarin se sastoji od dva dijela:

- Xamarin.Essentials – biblioteke koje pružaju *cross-platform* API-je za nativni pristup mogućnostima platforme. Neke od mogućnosti informacije su o uređaju, pristup datotečnom sustavu, akcelerometru, mogućnost rada s pozivima, zaključavanje i otključavanje zaslona, konverzija teksta u govor itd.
- Xamarin.Forms – *open source framework* za ulazno-izlazne radnje. Omogućuje da se izrađuju nativna korisnička sučelja za više platformi od istog koda. Dosta slično WPF-u, u smislu da se koristi XAML s C# kodom u pozadini.

Napomena, za rad s Xamarinom treba instalirati sve potrebne biblioteke, ali i dodatno, ovisno o platformi za koju radimo, treba instalirati i SDK te platforme. Nadalje, iOS aplikacije ne možemo simulirati ni izvršavati u Windows okruženju, nego za rad s iOS *buildovima* treba imati uređaj povezan s MAC OS-om (ili virtualni stroj, ali ne radi uvijek).

Ukoliko ste instalirali Xamarin, dobit ćete ga u popisu predložaka pretragom.



Slika 44. Pretraga Xamarin predloška među projektima

Iz predložka ćemo odabrati Xamarin.Forms jer, kako i opis kaže, možemo ciljati iOS i Android aplikacije istovremeno (no i UWP aplikacije). Nakon odabira lokacije i imena gdje želimo spremiti projekt, pokazat će nam se „New Mobile App” prozor, gdje odabiremo kakav generalni predložak početnog zaslona aplikacije želimo, kao na prikazanoj slici ispod. „Flyout” znači da će aplikacija imati izbornik koji neće biti vidljiv na malom zaslonu, ali se može povećati. „Tabbed” znači da će aplikacija koristiti tabove da bi navigirala između više različitih sekcija. „Blank” znači da će se stvoriti aplikacija s jednim inicijalnim ekranom, gdje bismo mi trebali umetnuti sve potrebne strukture. Radi jednostavnosti odabrat ćemo „Flyout” predložak.

New Mobile App

Select a template for your app

Flyout

An app with a side menu that can be collapsed on small screens.

Tabbed

An app that uses tabs to navigate between sections.

Blank

An empty app with a single, initial screen.

I plan to develop for:

☒ Android
 ☒ iOS
 ☐ Windows (UWP) i

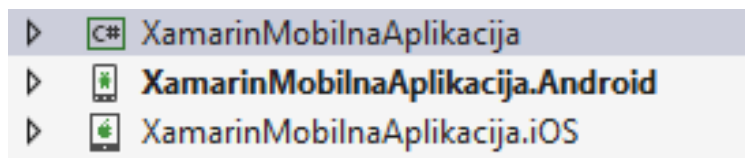
Back

Create

Slika 45. Odabir predloška dizajna glavne stranice u Xamarin Forms

Klikom na Create Visual Studio napravit će tri različita projekta:

- zajednički projekt
- projekt za Android
- projekt za iOS.



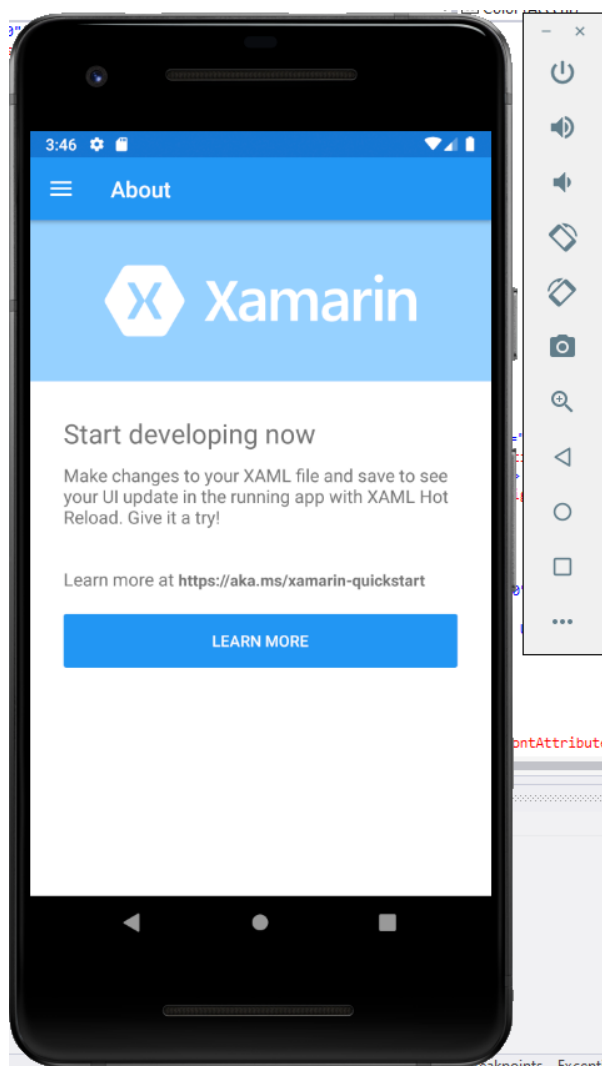
U zajedničkom projektu pisat ćemo kod koji nam je potreban za korištenje na svim platformama. Zbog gore navedenog načina kompajliranja nije moguće da sve bude u istom projektu. Zbog potrebe za izgradnjom i pokretanjem aplikacije imamo posebne projekte za svaku platformu, koji imaju reference na zajednički projekt. U projektima svake platforme možemo koristiti funkcije i pisati kod specifičan samo za tu platformu.

Kao početni projekt odabrana je Android aplikacija jer pokretanjem projekta/aplikacije želimo da se aplikacija pokrene na željenoj platformi gdje možemo testirati.

Standardno će se aplikacije pokrenuti pomoću virtualnog stroja (emulator mobilnog

uređaja) za koji trebate imati Hyper-V (instaliran, omogućen, ali i podržan od strane Windowsa i računala). Možete koristiti i mobilni uređaj, no za to morate osigurati da je instaliran USB *deploy driver* i da imate otključane developer opcije na svojem uređaju. Savjet je više koristiti konkretan mobilni uređaj jer je emulacija dosta „teška” po resurse računala i može jako usporiti aplikaciju, pa nećete imati dobru percepciju rada aplikacije.

Prvo pokretanje aplikacije može dugo trajati, pogotovo ako nisu instalirani svi potrebni dijelovi za izgradnju i izvođenje aplikacije, zato je potrebno mnogo strpljenja i čitajte ukoliko vam Visual Studio javi da postoji neka greška. Inicijalno pokretanja aplikacije u emulatoru izgledat će kao na slici 46.



Slika 46. Prikaz Xamarin projekta u emulatoru

Ako sad pogledamo kod i organizaciju u zajedničkom projektu, može se vidjeti da je organizacija projekta i rad sličan radu WPF aplikacija.

Xamarin aplikacija ima „XAML Hot Reload” mogućnosti, što znači da će promjene na XAML kodu unutar Visual Studia biti automatski ažurirane u realnom vremenu na aplikaciji. To je posebno dobro iz jednog razloga, a to je da nažalost nije uvijek moguće imati dizajnere za ove aplikacije, zbog previše različitih rezolucija mobilnih uređaja za koje se mogu razvijati aplikacije.

Nećemo prolaziti detalje organizacije koda, no kao što vidimo, u samom startu otvara se *AboutPage.xaml* i tu ćemo datoteku urediti, odnosno ubaciti nekoliko kontrola na sljedeći način:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamarinMobilnaAplikacija.Views.AboutPage"
    xmlns:vm="clr-namespace:XamarinMobilnaAplikacija.ViewModels"
    Title="{Binding Title}">

    <ContentPage.BindingContext>
        <vm:AboutViewModel />
    </ContentPage.BindingContext>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Color x:Key="Accent">#96d1ff</Color>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <StackLayout BackgroundColor="{
{StaticResource Accent}" VerticalOptions="FillAndExpand" HorizontalOptions="Fill">
            <StackLayout Orientation="Horizontal" HorizontalOptions="Center" VerticalOptions="Center">
                <ContentView Padding="0,40,0,40" VerticalOptions="FillAndExpand">
                    <Image Source="xamarin_logo.png" VerticalOptions="Center" HeightRequest="64" />
                </ContentView>
            </StackLayout>
        </StackLayout>
        <Grid Grid.Row="1">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Label Grid.Row="0" Text="Unesite ime:"></Label>

```

```

        <Entry Grid.Row="1" x:Name="KucicaUnosa" ></Entry>
        <Button Grid.Row="2" x:Name="GumbBtn" Text="Gumb" Clicked="GumbBtn_Clicked" ></Button>
        <Label Grid.Row="3" x:Name="porukaLbl"></Label>
    </Grid>
    <ScrollView Grid.Row="2">
        <StackLayout Orientation="Vertical" Padding="30,24,30,24" Spacing="10">
            <Label Text="Start developing now" FontSize="Title"/>
            <Label Text="Make changes to your XAML file and save to see your UI update in the running app with XAML Hot Reload. Give it a try!" FontSize="16" Padding="0,0,0,0"/>
            <Label FontSize="16" Padding="0,24,0,0">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Learn more at "/>
                            <Span Text="https://aka.ms/xamarin-quickstart" FontAttributes="Bold"/>
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Button Margin="0,10,0,0" Text="Learn more" Command="{Binding OpenWebCommand}" BackgroundColor="{StaticResource Primary}" TextColor="White" />
        </StackLayout>
    </ScrollView>
</Grid>

</ContentPage>

```

Naravno, treba dodati u pozadinski kod što da radi, pa ćemo kao i prije ubaciti da na pritisak gumba pokupimo tekst iz kontrole (*Entry*) i ispišemo pozdravnu poruku na labelu ispod gumba. Tako će onda *AboutPage.xaml.cs* izgledati ovako:

```

using System;
using System.ComponentModel;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

namespace XamarinMobilnaAplikacija.Views
{
    public partial class AboutPage : ContentPage
    {
        public AboutPage()
        {
            InitializeComponent();

            private void GumbBtn_Clicked(object sender, EventArgs e)
            {
                porukaLbl.Text = "Pozdrav " + KucicaUnosa.Text;
            }
        }
    }
}

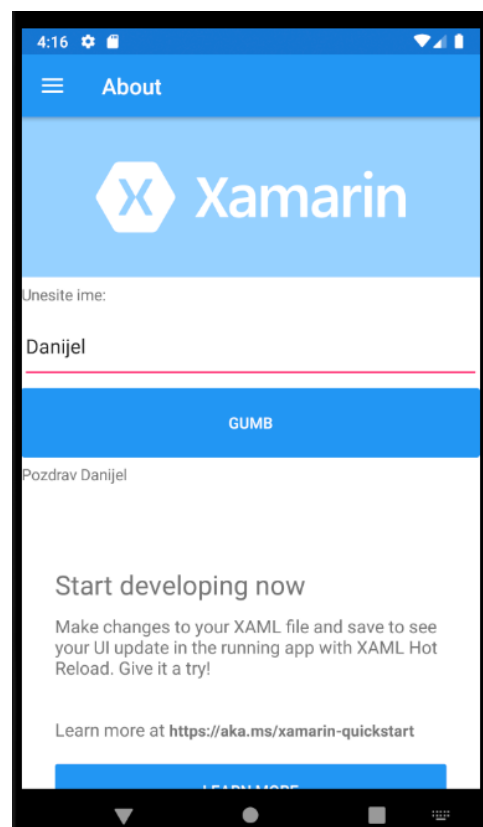
```

Kad radite izmjene na kodu u pozadini, obavezno će se morati ponovno buildati aplikacija, pa izmjene na kodu u pozadini nije tako lagano često raditi. Izmjene na XAML-u zahvaljujući „XAML Hot Reload” funkcionalnosti možemo raditi bez ponovnog buildanja aplikacije. Finalni izgled aplikacije bit će kao na slici 47.

Nećemo dalje ulaziti u Xamarin, no za one koje zanima, više možete saznati na sljedećoj poveznici:

<https://docs.microsoft.com/en-us/xamarin/>

Slika 47. Finalni izgled naše Xamarin aplikacije



4.8. LINQ

LINQ (Language Integrated Query) ime je seta tehnologija baziranih na integraciji upitnih karakteristika direktno u C# programski jezik. Tradicionalno su se upiti na podatke smatrali jednostavnim stringovima bez provjere tijekom kompajliranja, dodatno za svaki tip izvora podataka morao se naučiti zaseban jezik upita. S LINQ-om pišemo upite s jako tipiziranim kolekcijama objekta koristeći točnu jezičnu sintaksu i dobro poznate operatore. LINQ tehnologija pruža konzistentno iskustvo za objekte (LINQ to Objects), relacijske baze podataka (LINQ to SQL) i XML (LINQ to XML).

Za programera koji piše kod najbitniji i najvidljiviji je dio pisanje upita (koji često podsjećaju na SQL, koji smo već prošli). Upiti se pišu deklarativnom sintaksom, gdje možemo filtrirati, sortirati i grupirati podatke iz svojih izvora s minimalno truda.

LINQ upite možemo u C# kodu pisati za SQL server baze podataka, XML dokumente, ADO.NET Datasetove i bilo koje kolekcije objekata koje podržavaju *IEnumerable* sučelje ili generičko *IEnumerable <T>* sučelje. LINQ je također podržan s mnogo *web*-servisa i čak i drugih implementacija DBMS-a uz već spomenuti SQL Server.

Da demonstriramo kao LINQ radi, idemo napraviti jedan primjer:

```
using System;
using System.Collections.Generic;
using System.Net.Http.Json;

namespace Aplikacija
{
    public class Automobil
    {
        public int Id { get; set; }
        public string? Proizvodjac { get; set; }
        public string? Model { get; set; }
        public int? Godiste { get; set; }
    }
    class Program
```

```

{
    public static void Main()
    {
        List<Automobil> auti = new List<Automobil>();
        auti.Add(new Automobil { Proizvodjac = "Honda", Model = "Accord" });
        auti.Add(new Automobil { Proizvodjac = "VW", Model = "Passat" });
        auti.Add(new Automobil { Proizvodjac = "Ford", Model = "Mondeo" });

        var auto = (from a in auti
                    where a.Model == "Accord"
                    select a.Proizvodjac).SingleOrDefault();

        Console.WriteLine("Našao sam da je " + auto + " proizveo traženi
                           automobil");
    }
}

```

U primjeru vidimo da se LINQ upit sastoji od tri osnovna dijela, odnosno bloka.

Samo da jednu stvar napomenemo, LINQ upite možemo raditi samo s generičkim kolekcijama. Kolekcije objekata poput ArrayList i slično, odnosno kolekcije objekata bez strogo definiranog podatkovnog tipa nisu podržane.

4.8.1. From blok

Svaki LINQ upit počinje **from** ključnom riječi. Nakon toga dolazi proizvoljno imenovani objekt koji želimo izvući iz kolekcije, nakon čega slijedi ključna riječ **in**, nakon koje navodimo kolekciju koju želimo pretražiti. Logički gledano, ako bismo preveli upit na hrvatski, prijevod bi glasio „Iz objekta u kolekciji”.

Objektu koji koristimo u from bloku automatski će biti dodijeljen tip podatka, a ovisit će isključivo o podatkovnom tipu elementa kolekcije.

4.8.2. Where blok

Where blok koda dio je u kojem definiramo uvjet pretrage. Ovdje paralelu možemo povući sa SQL-om, samo pazite da ne koristite operatore isto kao u SQL-u. Operatore koristimo kako bismo ih koristili u C#-u, s mogućnostima da ovdje postavimo dosta komplicirane uvjete. Samo budite oprezni, kao i kod operatora, da podatkovni tip objekta koji koristite može raditi s operatorima koje imate. Također, kad se radi s objektima, moguće je koristiti metode klase tipa kojeg je objekt koji ovdje koristimo.

4.8.3. Select blok

Select definira koje nađene podatke vratiti upitom. Unutar from dijela definirali smo objekt koji ćemo ispitivati, no u select dijelu mi možemo još dodatno filtrirati što ćemo raditi s objektom koji smo dobili da će ići u rezultat, odnosno možemo dodatno obraditi što će se spremi i na koji način. Unutar select dijela možemo ostaviti i da vraćamo cijeli objekt, a ne samo dio. U našem primjeru mogli smo ostaviti cijeli objekt, pa da kasnije koristimo podatke u svim svojstvima po potrebi.

4.8.4. Povrat zapisa

Nakon upita u zagradama trebamo još definirati kako ćemo interpretirati podatke koje smo dobili upitom, odnosno na koji ćemo način raditi s njima nakon upita. Bez sekcije povrata zapisa LINQ upit vraća objekt koji odgovara sučelju `IEnumerable<T>`. Dobro je točno definirati koji podatkovni tip želimo za rješenje upita da bismo kasnije lakše radili s podacima.

Najčešće korištene metode povrata zapisa sljedeće su:

- **Single** – metoda vraća samo jedan rezultat upita. Ako ne postoji nijedan rezultat ili je više rezultata, diže se iznimka
- **SingleOrDefault** – radi slično kao Single, samo umjesto dizanja iznimke vraća null vrijednost

- **First** – vraća samo prvi rezultat. Ako nema nijednog rezultata, diže se iznimka
- **FirstOrDefault** – isto kao First, samo ako nema rezultata, vraća null
- **Last** – vraća zadnji rezultat. Ukoliko nema nijednog rezultata, diže se iznimka
- **LastOrDefault** – isto kao Last, samo ukoliko nema rezultata, vraća null
- **ToList** – vraća cijelu kolekciju objekata upita u listi
- **toDictionary** – vraća cijelu kolekciju objekata upita u rječniku
- ...

Ukratko, imamo mnogo metoda kao i sa svakom kolekcijom koja implementira `IEnumerable` sučelje.

4.8.5. Sortiranje podataka

S LINQ-om tijekom izrade upisa možemo jednostavno sortirati pronađene zapise. Ovo je opcionalan blok i koristimo **orderby** (bez razmaka) ključnu riječ. Sortiranje podataka možemo napraviti silazno i uzlazno.

Recimo da želimo sortirati sve modele koje je proizveo Audi, napravili bismo ovako:

```
var auti = (from a in auti
            where a.Proizvodjac == "Audi"
            orderby a.Model ascending
            select a.Proizvodjac).ToList();
```

Ključna riječ **ascending** znači da će se modeli poredati abecedno uzlaznim redom (od A do Z). Ukoliko želimo sortirati silaznim redom, treba navesti **descending** (od Z do A).

4.8.6. Dodatne mogućnosti

Naravno ovo navedeno nisu ni blizu sve LINQ mogućnosti. Postoji još mogućnost grupiranja (**groupby**), korištenje identifikatora (**let**), korištenje lambda izraza unutar upita itd.

Pokazali smo direktno korištenje LINQ na objektima, odnosno kolekciji objekata.

S EntityFrameworkom možemo koristiti LINQ to SQL, da bismo radili direktno s bazama podataka preko DataContext klase i Table kolekcija, održavajući relacije koje su definirane u bazi podataka.

S LINQ to XML iz XML datoteka možemo direktno raditi upite, samo XML datoteka mora biti na jedinstveni način formatirana.

4.9. Datum i vrijeme

Za rad s datumom i vremenom u C# programskom jeziku koristit ćemo strukturu DateTime.

DateTime sadrži podatke o datumu i točnom vremenu. U C#-u, bilo da želimo prikazati samo vrijeme ili samo datum, koristit ćemo DateTime strukturu.

DateTime struktura predstavlja datum i vrijeme s vrijednostima od 1.1.1. 00:00:00 do 31.12.9999. 23:59:59 po gregorijanskom kalendaru. Interno su vrijednosti koje se spremaju otkucaji, gdje je svaki otkucaj 100 nanosekundi. Ukratko, datum i vrijeme uvijek prikazujemo eksplicitno u kontekstu kalendara i vremena.

Inicijalizaciju DateTime objekta možemo napraviti na nekoliko načina:

- korištenjem konstruktora
- pozivajući metodu već postojećem DateTime objektu da vrati izračunatu novu vrijednost u obliku objekta
- parsiranjem iz stringa.

Primjeri inicijalizacije:

```
DateTime datum_neki = new DateTime(2022,4,24);  
DateTime datum_neki2 = DateTime.Parse("2/2/2022 21:32");  
DateTime datum_neki3 = datum_neki.AddDays(30);
```

Na objektima tipa `DateTime` možemo raditi usporedbe korištenjem relacijskih operatora. Samo napomena – prilikom usporedbe interno se usporedba radi na broju otkucaja (*ticks*) te su na temelju toga dva *datetime* objekta jednaka ako je njihov broj otkucaja evaluiran kao jednak. Znači ako se jedan objekt razlikuje od drugog u 100 nanosekundi, znači da su različiti datumi i vremena u pitanju.

Svojstva i metode možemo podijeliti na statične (za koje ne trebamo objekt) i dinamičke (za koje trebamo imati inicijalizirani objekt).

Važnija statička svojstva `DateTime` strukture:

- **Now** – vraća trenutno vrijeme lokalnog računala (čita sistemsko vrijeme iz BIOS-a)
- **Today** – vraća trenutni datum lokalnog računala
- **UtcNow** – isto kao Now, samo u UTC formatu.

Primjer različitog ispisa ovih triju svojstava:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

```
namespace DateTimePrimjeri  
{  
    internal class Program  
    {
```

```

static void Main(string[] args)
{
    Console.WriteLine(DateTime.Today);
    Console.WriteLine(DateTime.Now);
    Console.WriteLine(DateTime.UtcNow);
    Console.ReadLine();
}
}
}

```

// Ispis:

/*

18.6.2022. 0:00:00

18.6.2022. 16:52:29

18.6.2022. 14:52:29

*/

Važnije statične metode **DateTime** strukture:

- **Compare** – uspoređuje dva datuma
- **DaysInMonth** – za danu godinu i mjesec vraća koliko ima dana u tom mjesecu
- **Equals** – vraća jesu li dva **DateTime** objekta jednaka (isto kao `==` operator)
- **Parse** i **ParseExact** – za parsiranje iz teksta i vraća odgovarajući **DateTime** objekt
- **TryParse** i **TryParseExact** – isto kao i gore, no rade s out parametrom, tj. u slučaju greške ne bacaju iznimku nego samo vraćaju false vrijednosti da objekt nije instanciran.

Svojstva **DateTime** objekta:

- **Ticks** – broj otkucaja, kako se interno broji datum i vrijeme
- **Second** – komponenta sekundi u trenutnom zapisu
- **Date** – Vraća samo datum komponentu objekta
- **Month** – komponenta mjeseca u trenutnom zapisu

- **Minute** – komponenta minuta u trenutnom zapisu
- **Milisecond** – komponenta milisekunda u trenutnom zapisu
- **Hour** – komponenta sata u trenutnom zapisu
- **DayOfYear** – vraća koji je dan u godini u trenutnom zapisu
- **DayOfWeek** – vraća koji je dan u tjednu u trenutnom zapisu
- **Day** – komponenta dana u trenutnom zapisu
- **TimeOfDay** – vraća samo vremensku komponentu u trenutnom zapisu
- **Year** – komponenta godine u trenutnom zapisu.

Napomena – sva svojstva samo su za čitanje, ne možemo postavljati nijedno svojstvo.

Neke od metoda DateTime objekta:

- **DateTime** – konstruktora ima preko desetak ovisno o načinu kako ćemo raditi konstrukciju objekta upotrebom new operatora
- **Add** – dodavanje TimeSpan razlike u vremenu da se izračuna DateTime novog objekta
- **AddDays** – dodavanje dana na trenutni objekt
- **AddHours** – dodavanje sati na trenutni objekt
- **AddMilliseconds** – dodavanje milisekundi na trenutni objekt
- **AddMinutes** – dodavanje minuta na trenutni objekt
- **AddMonths** – dodavanje mjeseci na trenutni objekt
- **AddSeconds** – dodavanje sekundi na trenutni objekt
- **AddTicks** – dodavanje otkucaja na trenutni objekt
- **AddYears** – dodavanje godina na trenutni objekt
- **CompareTo** – usporedba s drugim objektom
- **Equals** – usporedba je li trenutni objekt jednak proslijeđenom
- **GetDateTimeFormats** – dohvat trenutnog formata zapisa
- **IsDaylightSavingsTime** – vraća je li trenutno pomaknut sat za štednju vremena
- **ToShortDateString** – vraća string formatiran prema sistemskom formatiranju za kratki zapis datuma

- **ToLongDateString** – vraća string formatiran prema sistemskom formatu ranju za dugi zapis datuma
- ...

4.9.1. TimeSpan

TimeSpan strukturu koristimo za računanje i prikaz točnog vremenskog intervala. Jedan od glavnih razloga korištenja jest preciznost prikaza. Interno gledano, **DateTime** sprema otkucaje sata i računa ostalo na temelju toga. Znači naći razliku između dva **DateTime** objekta svodi se na oduzimanje otkucaja jednog objekta od otkucaja drugog objekta. No rezultat, ako ga uzmemo kao **DateTime**, znači da će nam dati neki datum i vrijeme sukladno tomu od kada mjerimo datum i vrijeme. Problem nastaje prilikom prikaza mjeseca i većih vremenskih intervala koji nisu konstantni. Mjesec može sadržavati od 28 do 31 dana, godina može sadržavati 365 ili 366 dana. **TimeSpan** se zato ograničava na maksimalno prikazivanje dana, odnosno razlike u danima, jer tako možemo biti precizni i bez dvojbe definirati razliku.

Primjer dobivanja **TimeSpan** objekta:

TimeSpan vrijeme = DateTime.Now - new DateTime(2000, 1, 1);

Razlika vremena može biti negativna, kad oduzimamo manji broj od većeg. Negativni **TimeSpan** ne baca iznimku i ima logike to što imamo negativni vremenski interval. Korisno kad u knjigovodstvu prikazujemo da je prošao period za plaćanje.

Korisni statični članovi TimeSpan klase, sve samo za čitanje:

- **MaxValue** – daje maksimalnu vrijednost
- **MinValue** – daje minimalnu vrijednost
- **Zero** – daje nulu.

Statične metode TimeStamp strukture:

- **FromDay** – vraća TimeSpan objekt instanciran od proslijeđenog broja dana
- **FromHours** – vraća TimeSpan objekt instanciran od proslijeđenog broja sati
- **FromMilliseconds** – vraća TimeSpan objekt instanciran od proslijeđenog broja milisekundi
- **FromMinutes** – vraća TimeSpan objekt instanciran od proslijeđenog broja minuta
- **FromSeconds** – vraća TimeSpan objekt instanciran od proslijeđenog broja sekundi
- **FromTicks** – vraća TimeSpan objekt instanciran od proslijeđenog broja otkucaja
- **Parse** – vraća TimeSpan objekt instanciran od proslijeđenog stringa. Baca iznimku ako ne uspije parsirati string
- **TryParse** – vraća *true* ili *false* ovisno o tome je li uspio instancirati objekt *out* parametra za proslijeđeni string.

Članovi konstantni TimeSpan objekta, svi samo za čitanje:

- **TicksPerDay** – otkucaja po danu
- **TicksPerhour** – otkucaja po satu
- **TicksPerMillisecond** – otkucaja po milisekundi
- **TicksPerminute** – otkucaja po minuti
- **TicksPerSecond** – otkucaja po sekundi.

Svojstva TimeStamp objekta, sva samo za čitanje:

- **TotalMilliseconds** – vraća ukupan broj milisekundi vremenskog intervala
- **TotalHours** – vraća ukupan broj sati vremenskog intervala
- **TotalDays** – vraća ukupan broj dana vremenskog intervala
- **TotalMinutes** – vraća ukupan broj minuta vremenskog intervala
- **TotalSeconds** – vraća ukupan broj sekundi vremenskog intervala
- **Ticks** – vraća broj otkucaja spremljen u objektu
- **Seconds** – vraća komponentu sekundi vremenskog intervala

- **Minutes** – vraća komponentu minuta vremenskog intervala
- **Miliseconds** – vraća komponentu milisekundi vremenskog intervala
- **Hours** – vraća komponentu sata vremenskog intervala
- **Days** – vraća komponentu dana vremenskog intervala.

Nekoliko najčešćih metoda **Timestamp** strukture:

- **TimeSpan** – konstruktor klase za korištenje s new operatorom
- **Compare** – metoda za uspoređivanje trenutnog objekta s drugim objektom
- **Add** – dodavanje proslijeđenog TimeSpan objekta trenutnom
- **Divide** – dijeljenje vremenskog intervala proslijeđenim brojem
- **Multiply** – množi trenutni TimeSpan objekt s proslijeđenim brojem
- **Negate** – negira trenutni TimeSpan objekt
- **Subrtact** – isto kao operator oduzimanja
- ...

4.10. Enumeracije

Enumeracija je niz konstanti pri čemu svaki element niza ima svoje ime. Korisno je da si neke dijelove koda koje često koristimo, odnosno opcije, spremimo kao imenovane parametre da ne moramo stalno „pogađati” koju smo vrijednost mislili navesti. Enumeracije možemo definirati na razini imenskog prostora ili na razini klase. Primjeri enumeracije:

```
public enum Spol
{
    Muski,
    Zenski,
    Ostalo
}
public enum Operacija
{
    Citanje=1,
    Pisanje=2,
```



```
CitanjePisanje=12,  
NemaPristupa=0  
}  
Sučelja
```

4.11. Sučelja

Sučelje ili engleski **interface** referentni je tip koji sadrži samo apstraktne članove. Članovi sučelja mogu biti svojstva, metode i događaji, ali samo sa svojom deklaracijom, ne mogu biti implementirani. Klasa koja nasljeđuje sučelja jest ta koja ima njegovu implementaciju.

Sučelja su jako korisna u C# programskom jeziku jer rješavaju jedan veliki problem C#-a, a to je nemogućnost višestrukog nasljeđivanja. Kako smo već rekli, C# klasa može naslijediti samo jednu klasu, no da bismo omogućili nasljeđivanje više metoda i svojstava iz različitih izvora, klasa može naslijediti više sučelja.

Sučelja definiramo slično kao klase:

```
interface ImeSucelja  
{  
    //svojstva:  
    tip ImeSvojstva  
    {  
        get;  
        set;  
    }  
    ...  
  
    //Metode:  
    tip ImeMetode();  
    ...  
}
```

Članovi sučelja implementiraju se u klasi koja sučelje nasljeđuje. Prilikom nasljeđivanja sučelja, ukoliko nasljeđujemo i klasu, klasa mora biti navedena prva, a tek onda sučelje.

```
class Implementacija:DrugaKlasa,Sucelje1,Sucelje2..
{
    //Implementacija svih metoda i svojstava svih sučelja...
}
```

Tipično ime svakog sučelja, pogotovo sistemskih sučelja, počinje velikim slovom i (I). Korištenje sučelja puno ćemo više kasnije prolaziti u predlošcima dizajna, jer skoro se svaki predložak svodi na dobru implementaciju već definiranih sučelja. Primjer – ako želimo da nam klasa koju radimo bude kao da koristimo listu, implementirat ćemo IList sučelje.

Svaki dio .NET-a ima velik broj sučelja, koja ćemo prepoznati po početnom slovu I, pa zatim imenu funkcionalnosti sučelja. Implementacijom sučelja također našim klasama dajemo ponašanje koje može biti prepoznato da se koristi s već ugrađenim metodama. Npr. implementacijom IComparable sučelja u našim klasama, možemo koristiti Sort metodu za sortiranje kolekcije elemenata naše klase.

Neka od najčešće korištenih sučelja:

- **IEnumerable** – za korištenje s foreach petljom za prolazak kolekcije i LINQ
- **IDisposable** – za čišćenje resursa, koristi se s using blokom
- **IQueryable** – za izvršavanje upita prema podatkovnom izvoru
- **INotifyPropertyChanged** – za vezanje podataka i korisničkog sučelja
- **IComparable** i **IComparer** – za usporedbe, korisno za korištenje metoda sortiranja
- **IComparable** i **IEqualityComparer** – za usporedbe jednakosti
- **IList** i **ICollection** – za izradu vlastitih kolekcija
- **IDictionary** – za izradu kolekcija tipa rječnika
- **ISerializable** – za serijalizaciju objekta
- **ILogger** – za logiranje u programu.

Primjer implementacije IComparable sučelja u klasi Automobil za usporedbu automobila:

```
class Automobil : IComparable
{
    public int GodinaProizvodnje { get; set; }
    public string Marka { get; set; }
    public string Model { get; set; }
    public string SnagakW { get; set; }

    public int CompareTo(object obj)
    {
        if (obj == null) return 1;
        if (obj is Automobil)
        {
            var auto = obj as Automobil;
            if (this.GodinaProizvodnje.CompareTo(auto.GodinaProizvodnje) != 0) return this.GodinaProizvodnje.CompareTo(auto.GodinaProizvodnje);
            else if (this.SnagakW.CompareTo(auto.SnagakW) != 0) return this.SnagakW.CompareTo(auto.SnagakW);
            else if (this.Marka.CompareTo(auto.Marka) != 0) return this.Marka.CompareTo(auto.Marka);
            else return this.Model.CompareTo(auto.Model);
        }
        return 0;
    }
}
```

5. POGLAVLJE:

C# i .NET napredno programiranje

U OVOM POGLAVLJU NAUČIT ĆETE:

- kako integrirati dodatne biblioteke
- pristupati bazama podataka
- programirati za višedretveni rad
- koristiti parcijalne klase
- Windows communication foundation
- validaciju podataka
- sigurnost aplikacije
- načine isporuke aplikacija.

5.1. Integracija dodatnih biblioteka

Integracija dodatnih biblioteka omogućuje nam da proširimo funkcionalnost našeg programa s dijelovima programa i/ili klasama koje je netko drugi razvijao.

Željeni učinak integracije možemo postići na nekoliko načina. Možemo instalirati dodatne biblioteke preko nekog upravljača paketima (Nuget ili Extension managera u VS-u) ili možemo samo napraviti referencu na potrebnu biblioteku.

U svakom projektu možete dodavati nove biblioteke. Ovisno o tipu projekta koji imate, biblioteke možete dodavati preko „Dependencies” ili „References”. U konačnosti to

se svodi na slično, jer možete dodati:

- projektne reference – da biste mogli koristiti kod iz drugog projekta u *solutionu*
- COM reference – *Component Object Model* služi da bismo učitali biblioteke namijenjene drugim programskim jezicima. CLR automatski eksponira COM komponente preko *collable wrapera* (hrv. sklopivog omota) tijekom izvođenja programa
- Assemblies – komponente *.NET frameworka*
- zasebne biblioteke – biblioteke koje su učitane direktnim učitavanjem biblioteke, a nisu dio već gore navedenog.

Jednom kad učitamo biblioteku, funkcionalnost koju ona sadrži možemo pozivati iz našeg koda, ali i, u slučaju klasa, nasljeđivati ih i koristiti.

5.1.1. NuGet

Danas nužan alat za bilo koju modernu razvojnu platformu jest mehanizam preko kojeg programeri mogu razvijati, dijeliti i konzumirati koristan kod. Taj kod često grupiramo u pakete koji sadrže kompajlirani kod (najčešće u obliku dll-a) uz ostali sadržaj koji je potreban u projektu da se konzumira sadržaj paketa.

Za .NET, Microsoftov podržani mehanizam za dijeljenje koda jest **NuGet**, koji definira kako su paketi za .NET stvoreni, dijeljeni i konzumirani te pruža alate za pojedinu od navedenih uloga.

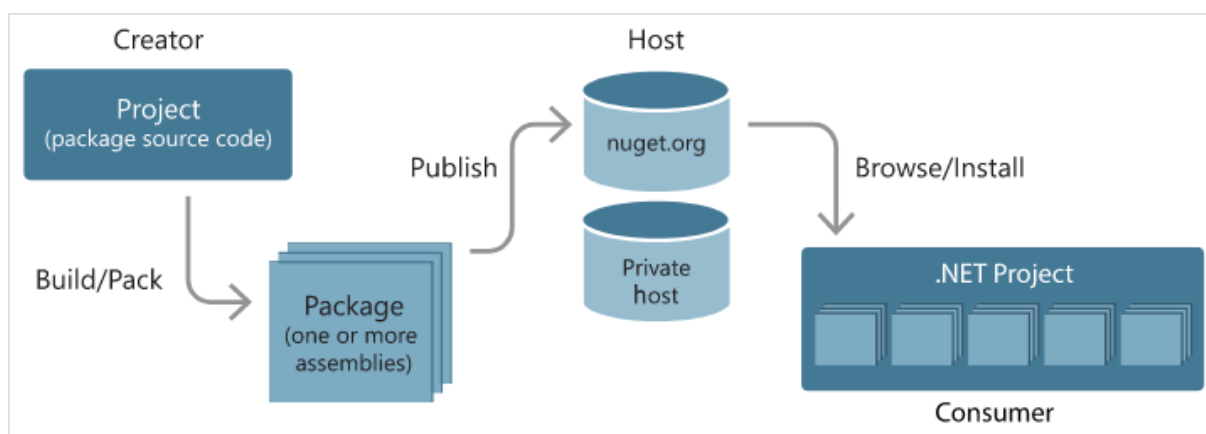
Jednostavno rečeno, NuGet paket jedna je zip datoteka s .nupkg ekstenzijom, koja sadrži pretkompajlirani kod (dll), ostale datoteke koje su srodne s tim kodom i opisni manifest koji sadrži informacije poput broja verzije paketa. Programeri koji žele dijeliti kod naprave paket i objavljuju ga na javnom ili privatnom poslužitelju. Programeri koji konzumiraju paket dobivaju te pakete od odgovarajućeg poslužitelja, dodaju ga svojim projektima, a zatim rade pozive na kod u tim paketima. Dobra je stvar to što NuGet odrađuje sve međudetalje, pa se programerima maksimalno olakšava rad.

U ulozi javne usluge NuGet održava centralni repozitorij od preko 100 000 (sto tisuća) jedinstvenih paketa na nuget.org, koje upotrebljava nekoliko milijuna programera na dnevnoj bazi. NuGet pakete moguće je posluživati privatno u oblaku (poput Azure DevOps), na privatnim mrežama ili je moguće imati pakete na lokalnom računalu. Preko konfiguracijskih opcija može se kontrolirati koji će točno hostovi biti pristupani s kojeg računala, time osiguravajući da se pakete može dobiti iz specifičnih izvora, a ne samo iz javnih repozitorija poput nuget.org-a.

Host se koristi kao veza između kreatora paketa i konzumenta paketa. Kreatori prave korisne NuGet pakete i objavljuju ih na hostu. Konzumeri traže sebi korisne pakete prisutne na hostu, skidaju ih i uključuju u svojim projektima. Jednom kad se instalira u projektu, paket API raspoloživ je ostatku koda projekta.

Slika 48. ilustrira grafički kako radi NuGet

(<https://docs.microsoft.com/en-us/nuget/what-is-nuget>).

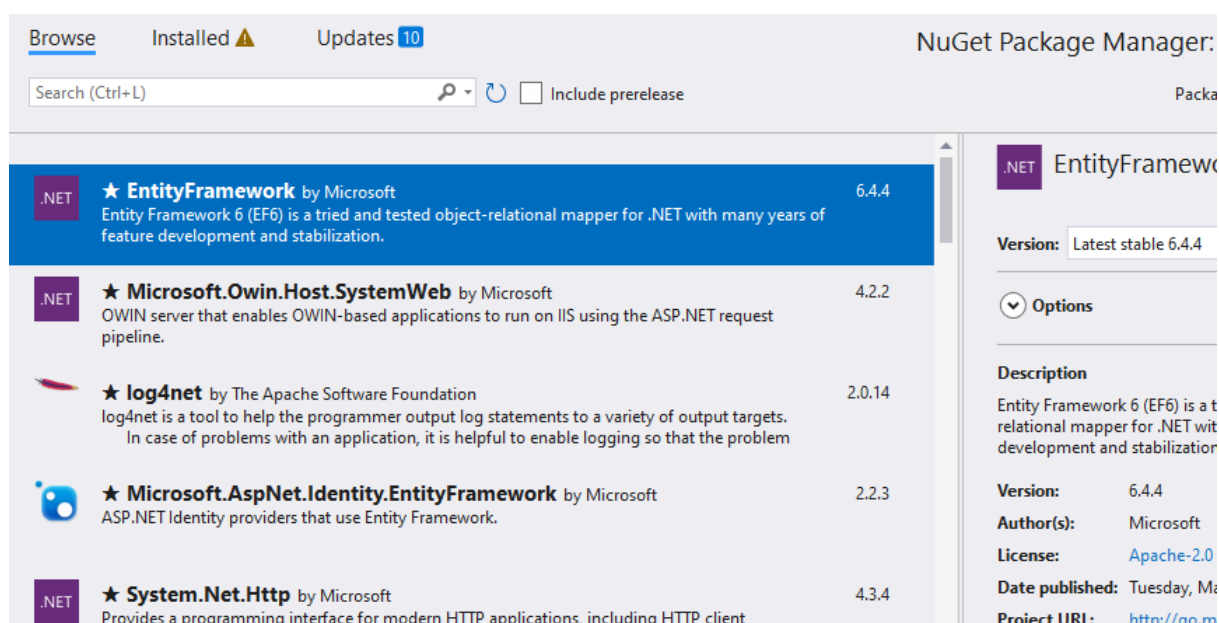


Slika 48. Grafički prikaz rada NuGet-a

Kompatibilni paket onaj je koji sadrži dijelove građene za bar jedan ciljani .NET framework i kompatibilan je s ciljanim projektnim frameworkom konzumerskog projekta. Programeri mogu stvoriti paket koji je specifičan za jedan framework, poput UWP kontrola, ali mogu podržavati i širi aspekt ciljeva. Da bi se maksimizirala kompatibilnost paketa, programeri ciljaju .NET standard, koji mogu konzumirati .NET i .NET Core projekti.

Alati NuGeta automatski se instaliraju s instalacijom Visual Studia (od 2017 na novije verzije), što znači da već imamo sve potrebno da dodajemo NuGet pakete u svoje projekte.

Za upravljanje NuGet paketima dovoljno je na projekt kliknuti desnim klikom miša i odabrati opciju „Manage NuGet Packages”. Otvorit će nam se prozor sličan prikazanom na slici 49.



Slika 49. Prikaz NuGet package managera

Prozor ima tri taba:

- Browse – za pretragu online repozitorija
- Installed – pokazuje koji su trenutno instalirani paketi
- Updates – pokazuje ima li trenutno instalirani paket ažuriranja.

Pretragom u traci za pretragu možemo pronaći točan paket koji tražimo, ali tražit će se samo u odabranom tabu.

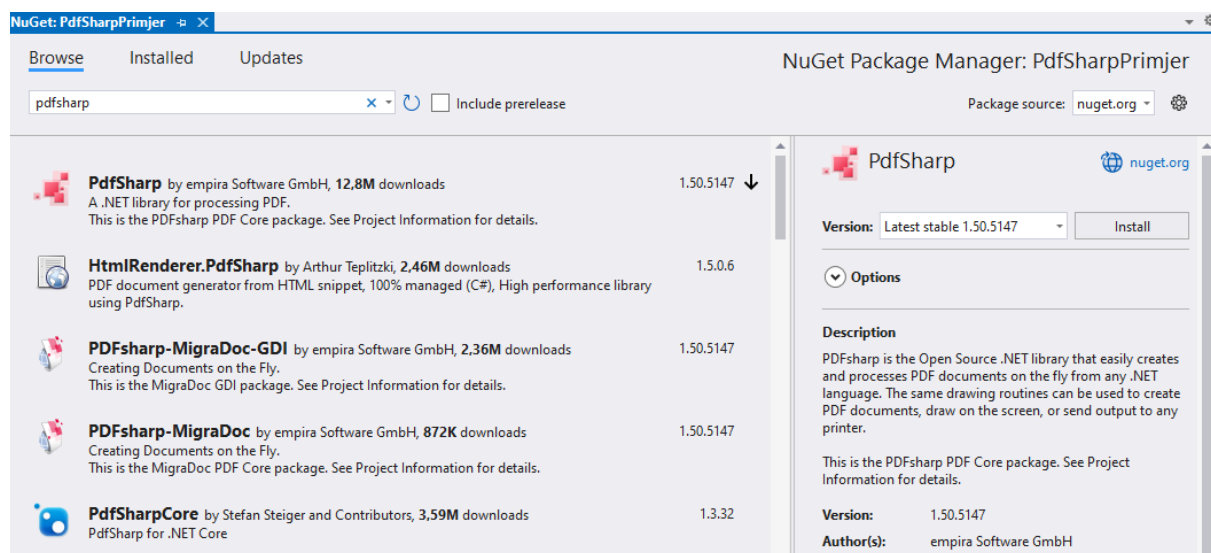
Klikom na paket u desnom prozoru možemo vidjeti koja je verzija paketa, a možemo odabrati i jednu od prijašnjih verzija. Nove verzije nekad ne moraju biti stabilne, pa ako znate za stabilnu verziju za koju ste sigurni, držite se stabilne verzije. Također, možete provjeriti još detalja paketa, poput opisa, verzije, autora, tipa licence, kad je izdan paket, URL projekta, ovisi li o drugim projektima itd.

5.1.2. PdfSharp

Primjer NuGet paketa je PdfSharp paket. PdfSharp je *open source* .NET biblioteka koja lagano stvara i obrađuje PDF dokumente na dinamički način iz bilo kojeg .NET programskog jezika. Iste metode crtanja mogu se koristiti za stvaranje PDF dokumenta, mogu se crtati na ekranu ili poslati izlaz na neki pisač.

PdfSharp ima mnogo funkcija, a one tipično vraćaju novo kreirane objekte. Jednom kad se nauče osnovni principi, lagano se radi s PdfSharom. Podržan je Intellisense, koji uvelike pomaže u pisanju koda. Također ima sustav pomoći integriran u Visual Studio i pomaže pružiti informacije o bilo kojoj klasi u PdfSharpu.

Za demonstraciju rada napraviti ćemo jedan primjer. Radi jednostavnosti napraviti ćemo opet konzolnu aplikaciju, no pazite da bude .NET Framework, a ne .NET Core (problem podrške za .NET Core PdfSharpa u trenutku pisanja). U aplikaciji ćemo preko NuGeta instalirati PdfSharp.



Slika 50. Instalacija PdfSharpa preko NuGeta

Naravno, preuzet ćemo zadnju verziju, iako će u našem jednostavnom primjeru i starija verzija poslužiti.

Program CS modificirat ćemo da izgleda ovako:

```
using System;
using System.Diagnostics;
using System.IO;
using PdfSharp;
using PdfSharp.Drawing;
using PdfSharp.Pdf;
using PdfSharp.Pdf.IO;

namespace PdfSharpPrimjer
{
    class Program
    {
        static void Main(string[] args)
        {
            // Stvaranje novog pdf dokumenta
            PdfDocument dokument = new PdfDocument();
            dokument.Info.Title = "Stvoreno s PDFsharp";

            // Stvaranje prazne stranice
            PdfPage stranica = dokument.AddPage();

            // Dohvat XGraphics objekta za crtanje
            XGraphics gfx = XGraphics.FromPdfPage(stranica);

            // stvaranje fonta
            XFont font = new XFont("Verdana", 20, XFontStyle.BoldItalic);

            // Draw the text
            gfx.DrawString("Pozdrav backendaši!", font, XBrushes.Black,
                new XRect(0, 0, stranica.Width, stranica.Height),
                XStringFormats.Center);

            // Pohrana dokumenta...
            const string imedatoteke = "Pozdrav.pdf";
            dokument.Save(imedatoteke);

            // ...otvaranje dokumenta.
```

```
        Process.Start(imedatoteke);
    }
}
}
```

Kao rezultat izvođenja otvorit će se .pdf datoteka i u mapi izvršne datoteke imat ćemo Pozdrav.pdf datoteku.

Naravno, ovo nisu sve mogućnosti rada s PdfSharp bibliotekom, no za sve dodatne funkcionalnosti slobodno pročitajte dokumentaciju koja je opširna s mnogo primjera i istražite sami ovisno o svojim potrebama.

Poveznica na PdfSharp: <http://www.pdfsharp.com/PDFsharp/>

5.1.3. Newtonsoft.Json

Newtonsoft.Json popularna je biblioteka za rad s JSON formatom zapisa. Newtonsoft.Json imenski prostor sadrži klase i metode koje se koriste da konvertiramo .NET tipove podataka u JSON tip. Iako, budući da je Microsoft napravio Text.Json, koji smo opisali kako radi u poglavlju 4.6.2, realno rečeno malo je razloga, ako uopće, za rad s Newtonsoft.Json pored Text.Json-a. Čitajući nekolicinu članaka i usporedbi, da ne ulazimo u previše detalja, jednostavno ćemo reći: za početnike jednostavnije je i bolje koristiti novi Text.Json za izradu serijalizacije i deserijalizacije. Za one koji već imaju veće iskustvo s Newtonsoft.Json ili koriste točno specifične metode rada s Newtonsoft.Json (a početnici to sigurno nisu), tada se isplati ostati na Newtonsoft.Json biblioteci.

Primjer korištenja JSON-a nećemo opet prolaziti, opet uputio bih na poglavlje 4.6.2 radi primjera rada s JSON-om.

5.1.4. GtkSharp

GtkSharp set je alata za izradu grafičkih korisničkih sučelja za mono i .NET. Projekt povezuje gtk+ set alata i GNOME biblioteku, što omogućuje razvijanje potpuno nativne grafičke Gnome aplikacije korištenjem Mono i .NET razvojnih alata.

GtkSharp jednostavno je omot (*wraper*) na gtk+, koji je *cross platform GUI framework*.

Ukratko, koristan je ako je netko već radio s gtk+ na drugoj platformi, pa se želi relativno brzo prebaciti na razvoj aplikacija s C#. Prvenstveno je GtkSharp namijenjen da radi s Mono *frameworkom*, s kojim je često i dolazio upakiran za razvoj aplikacija. Zbog toga je nastala zablude da treba Mono da funkcioniра, no dovoljan je bilo koji .NET kompatibilni *runtime*.

Ulaziti u detalje ovdje nema smisla, GtkSharp uglavnom je predviđen za rad s Monom te se kao i gtk+ korisničko sučelje u Visual Studiu radi u kodu, bez neke velike pomoći dizajnera.



Nama je prvenstveno fokus C# i .NET, a oni koji žele znati više, mogu potražiti sve informacije na poveznici:

<https://www.mono-project.com/docs/gui/gtksharp/>

Postoji još mnogo različitih biblioteka i alata, a definitivno nemamo prostora raspisati se o njima. Svakako potičemo da sami istražite ovisno o potrebama vaših budućih projekata.

5.2. ADO.NET i Entity Framework

U današnje vrijeme skoro svaka aplikacija koristi neki oblik skladištenja podataka. Najčešće korišten tip skladišta podataka neka je relacijska baza podataka (poput SQL Servera, koji smo prošli u poglavlju 2). Primarna uloga baze podataka trajna je pohrana podataka unesenih od strane korisnika aplikacije ili podataka uvezenih iz nekog drugog izvora.

5.2.1. ORM

ORM ili *Object-relational mapping* programska je tehnika za konverziju podataka između tipova koji se koriste u OOP jezicima. Ovom tehnikom stvaramo efektivnu virtualnu bazu objekata koje možemo koristiti unutar našeg programskog jezika. Mapiranje korištenjem Visual Studia, odnosno C#-a, uglavnom će nam se automatski konstruirati pomoću pogonskog sustava baze podataka.

Mnogi popularni sustavi za upravljanje bazom podataka nisu objektno orijentirani i mogu pohranjivati samo skalarne vrijednosti poput integera i stringova organiziranih u tablicama. Programer zatim mora pretvoriti objektnu vrijednost u grupu jednostavnijih vrijednosti za pohranu u bazi podataka ili koristiti jednostavnije skalarne vrijednosti u programu. ORM implementira pristup pretvaranja objekata u jednostavne vrijednosti za pohranu.

5.2.2. ADO.NET

ADO.NET objektni model nudi bogate mogućnosti za kreiranje distribuiranih aplikacija koje razmjenjuju gotovo svaku vrstu podataka. Omogućuje konzistentan pristup skladištima podataka kao što su MS SQL Server i XML datotekama. ADO.NET odvaja pristup podacima od manipulacije podacima na diskretne komponente koje se mogu koristiti zasebno ili u tandemu. ADO.NET uključuje pružatelje podataka .NET Framework za povezivanje s bazom podataka, izvršavanje naredbi i dohvaćanje rezultata. Ti se rezultati ili obrađuju izravno, stavljaju u ADO.NET DataSet objekt kako bi bili izloženi korisniku na *ad hoc* način, u kombinaciji s podacima iz više izvora, ili prosl-

jeđivani između razina. Objekt DataSet također se može koristiti neovisno o davatelju podataka .NET Framework za upravljanje podacima koji su lokalni za aplikaciju ili izvorni iz XML-a. ADO.NET ponekad se smatra evolucijom ActiveX Data Object (ADO) tehnologije, ali zbog velikih promjena koje su napravljene smatra se potpuno novim proizvodom.

ADO.NET konceptualno je podijeljen na potrošače i pružatelje podataka. Potrošači su aplikacije koje trebaju pristup podacima, a pružatelji su programske komponente koje implementiraju sučelje i pružaju podatke potrošačima.

Funkcijski postoje specijalizirane podklase DataSet klasa za specifičnu shemu baze podataka, koje pružaju jednostavan pristup svakom polju sheme preko jako tipiziranih svojstava. Ovo omogućuje da uhvatimo više grešaka u programiranju tijekom kompajliranja i omogućuje da efektivno koristimo Intellisense u izradi programa.

Pružatelj podataka programska je komponenta koja je u interakciji s podatkovnim izvorom. ADO.NET pružatelji podataka jesu ODBC driveri, JSBC driveri i OLE DB pružatelji.

ADO.NET pružatelji podataka mogu biti stvoreni da pristupaju jednostavnim podatkovnim skladištima poput tekstualnih datoteka i tablica, pa do kompleksnih baza podataka poput Oracle baze podataka, Microsoft SQL Servera, MySQL-a, PostgreSQL-a, SQLitea, IMB Db2-a itd. Mogu pružiti pristup i hijerarhijskim skladištima podataka poput e-mail sistema.

ADO.NET pružatelji podataka komponente su eksplicitno dizajnirane za manipulaciju podacima i brzo čitanje podataka. Receptura rada jednostavna je:

- Connection objekt koristimo za spajanje na bazu podataka
- Command objekt omogućuje pristup naredbama baze podataka (upiti, spremljene procedure, funkcije...)
- DataReader objekt omogućuje čitanje podataka iz baze podataka.

Ova osnovna tri koraka sve su što nam je potrebno za osnovni pristup bazi podataka.

Za rad s primjerima koristit ćemo bazu iz poglavlja 2 o SQL Serveru. Idemo raditi redom po prije navedenim koracima.

Prvi korak u radu s ADO.NET aplikacijama uspostavljanje je konekcije na bazu podataka. Za to nam služi `SqlConnection` klasa, koja se nalazi u `System.Data.SqlClient` imenskom prostoru.

`SqlConnection` objekt možemo instancirati na dva načina:

```
SqlConnection conn = new SqlConnection();  
SqlConnection conn = new SqlConnection(connString);
```

Tipičan primjer spajanja na bazu bio bi sljedeći:

```
string connString = "SERVER=.\SQLEXPRESS;DATABASE=MojaBaza;user=korisnik;password=lozinka";  
SqlConnection conn = new SqlConnection(connString);  
conn.Open();  
// rad na bazi podataka  
conn.Close();
```

Ključno za spajanje na bazu podataka vezni je string (*connection string*), odnosno `connString` varijabla. U toj varijabli specificirali smo ime servera na koji se spajamo, ime baze na serveru i korisničke podatke koje ćemo koristiti za spajanje na bazu podataka. U gornjem primjeru radi se spajanje ako je SQL Server autentifikacija omogućena i korisnik postoji s navedenim podacima. U *connection* stringu možemo još navesti dodatne podatke poput enkodiranja, integrirane sigurnosti i ostalo, no svako svaki podatak odvajamo točkom zarezom.

Idemo napraviti unutar konzolne aplikacija spajanje na bazu podataka i provjeru je li veza uspješna:

```

using System;
using System.Collections.Generic;
using System.Data.SqlClient;

namespace OsnoveSQL
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string connString = "SERVER=.\SQLEXPRESS;DATABASE=Skripta;inte-
grated security=true";
            SqlConnection conn = new SqlConnection(connString);
            try
            {
                conn.Open();
                Console.WriteLine("Veza na bazu uspješna");
            }
            catch (SqlException ex)
            {
                Console.WriteLine("Greška prilikom uspostave veze:\n" + ex.Message);
            }
            finally
            {
                conn.Close();
            }
            Console.ReadLine();
        }
    }
}

```

Ukoliko ste unutar `connString` napisali dobre podatke, dobit ćete kao rezultat poruku da je veza na bazu uspješna. U ovom primjeru koristimo Windows korisnički račun za pristup SQL Serveru (*integrated security = true*), što je moguće ako je trenutno logirani korisnik član grupe administratorskih ovlasti ili neke druge grupe koja daje prava korištenja SQL Servera. Ukoliko pogriješite nešto unutar *connection stringa*, `conn.Open()` bacit će `SqlException` iznimku i u `Message` svojstvu dobit ćete odgovarajuću

poruku. Npr. ako se kod SERVER parametra u stringu obriše zadnje slovo S, dobit će se sljedeća poruka:

Greska prilikom uspostave veze:

```
A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)
```

Ukratko, ne može naći instancu SQL servera, pa se ne može spojiti na nepostojeći server.

Nikako ne smijemo zaboraviti da vezu koju otvorimo trebamo i zatvoriti. Zato svaki poziv prema bazi podataka radimo između naredbi `conn.Open()` i `conn.Close()`.

Connection stringove moramo spomenuti. U njima su sve informacije koje trebamo da bismo uspostavili vezu na bazu podataka. Imena parametara ne moraju biti kako su navedena, nego možemo imati više validnih imena parametara za istu stvar. Na primjer, možemo koristiti ime „Server” ili „Data Source” ili „Address” ili „Addr” ili „Network Address”, a za bazu podataka možemo koristiti „Database” ili „Initial Catalog”.

Connection string u većini slučajeva nećemo imati hardkodiran u kodu, nego u nekoj konfiguracijskoj datoteci ili negdje spremljeno kao parametar koji možemo mijenjati bez da za svaku promjenu moramo kompajlirati program. Razlog je jednostavan, ako dođe npr. u produkcijskom okruženju do promjene poslužitelja, lakše je samo promijeniti jednu konfiguracijsku datoteku pa da se referenciramo na novi poslužitelj nego da za svaku sitnu promjenu ponovno kompajliramo cijelu aplikaciju.

Kako smo već napomenuli, kod za rad s bazom podataka bit će nam između `conn.Open()` i `conn.Close()` metoda. Alternativno, iz prijašnjeg primjera možemo na sljedeći

način napisati kod:

```
using(SqlConnection conn=new SqlConnection(connString))
{
    // Kod za rad s bazom podataka
} // Konekcija se automatski zatvara na kraju using bloka
```

Nakon uspostavljanja veze s bazom podataka koristimo SqlCommand objekt. Pomoću SqlCommand objekta možemo dohvaćati, upisivati, mijenjati i brisati podatke iz baze pomoću sql upita ili pozivom na procedure.

SqlCommand klasa predstavlja T-SQL naredbu ili proceduru. Objekt možemo instancirati na jedan od sljedećih načina:

```
SqlCommand cmd = new SqlCommand();
SqlCommand cmd1 = new SqlCommand(cmdText);
SqlCommand cmd2 = new SqlCommand(cmdText,conn);
SqlCommand cmd3 = new SqlCommand(cmdText,conn,transaction);
```

Parametar cmdText jest SQL naredba ili naziv procedure.

Parametar conn jest SqlConnection objekt preko kojeg je ostvarena veza na bazu podataka.

Parametar transaction objekt je u čijem se kontekstu odvija ova naredba, naravno samo ako je transakcija kreirana.

Kod SqlTransaction klase imamo nekoliko bitnih svojstava:

- **CommandText** – vraća ili postavlja SQL naredbu ili ime procedure
- **CommandTimeout** – vraća ili postavlja broj sekundi unutar kojih se naredba mora izvršiti
- **CommandType** – vraća ili postavlja kako commandText svojstvo treba biti interpretirano

- **Connection** – vraća ili postavlja SqlConnection objekt koji će se koristiti za izvršavanje naredbi
- **Parameters** – vraća kolekciju parametara povezanih s naredbom
- **Transaction** – vraća ili postavlja transakciju unutar koje se naredba treba izvršavati. Transakcija mora imati isti SqlConnection objekt kao i naredba.

Metode SqlTransaction klase koje ćemo koristiti:

- **Cancel** – služi da pokušamo otkazati izvršavanje naredbe
- **CreateParameter** – kreira novu instancu SqlParameter klase
- **ExecuteNonQuery** – izvršava naredbe koje ne vraćaju podatke (*insert*, *update* i *delete*)
- **ExecuteReader** – izvršava SQL upit i vraća DataReader objekt s podacima
- **ExecuteScalar** – izvršava upit i vraća prvi dohvaćeni podatak (kao i TOP 1), a ostali se ignoriraju
- **ExecuteXMLReader** – izvršava upit koji vraća XML podatke i kreira Xml Reader objekt
- **Prepare** – kreira pripremnu verziju naredbe u instanci SQL Servera.

Primjer koda s naredbama, odnosno ubacivanjem novog podatka u bazu i provjera broja zapisa:

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;

namespace OsnoveSQL
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string connString = "SERVER=\\.\\SQLEXPRES;DATABASE=Skripta;integrated security=true";
            using (SqlConnection conn = new SqlConnection(connString))
            {
```

```

conn.Open();
Console.WriteLine("Veza na bazu uspješna");
string cmdText = "INSERT INTO Osoba(Ime,Prezime)" +
    " Values('Alfred','Nobel)";
SqlCommand cmd = new SqlCommand(cmdText, conn);
cmd.ExecuteNonQuery();

cmd.CommandText = "SELECT COUNT(*) FROM Osoba";
int broj = (int)cmd.ExecuteScalar();
Console.WriteLine("U tablici imamo " + broj.ToString() + " zapisa");
}
Console.ReadLine();
}
}
}

```

Tijekom čitanja veće količine podataka koristit ćemo `SqlReader` objekt, koji je *read-only*, *forward-only* i *client-side* kursor koji može prihvatiti podatke vraćene pozivom na metode `ExecuteReader()` objekta tipa `SqlCommand`. Korištenje `SqlReader`a najbrži je način na koji možemo dohvatiti podatke iz baze podataka u aplikaciju.

Read-only kursor znači da se iz baze dohvaća jedan redak koji se može samo pročitati. *Forward-only* kursor znači da se tijekom čitanja ne možemo pomicati na redove koji su već pročitani, možemo raditi čitanje samo prema naprijed. *Client-side* kursor znači da se rezultat upita sprema u međuspremnik koji se nalazi na klijentskoj strani i dostupan je `SqlDataReader` objektu.

SqlDataReader objekt ima sljedeća korisna svojstva:

- **Depth** – daje nam dubinu ugniježđenosti trenutnog retka
- **FieldsCount** – vraća broj stupaca trenutnog retka
- **HasRows** – vraća broj redaka u `SqlDataReader` objektu
- **IsClosed** – vraća je li `SqlDataReader` objekt zatvoren
- **Item** – indexer svojstvo, vraća vrijednost stupca u izvornom formatu.

Važnije metode SqlDataReader objekta:

- **Close** – zatvara SqlDataReader objekt (samo sebe, ne zatvara SqlConnection)
- **GetFieldType** – vraća tip objekta za podatak u specifičnom stupcu
- **Getname** – vraća ime specificiranog stupca
- **GetOrdinal** – za proslijeđeno ime stupca vraća njegov redni broj
- **GetSchemaTable** – vraća DataTable objekt koji opisuje međupodatke za stupce upravljane SqlDataReader objektom
- **GetValue** – vraća vrijednost specificiranog stupca u svom izvornom formatu
- **GetValues** – kopira vrijednosti svih stupaca u proslijeđeni niz
- **IsNull** – provjerava sadržava li stupac null vrijednost
- **Read** – pomiče pokazivač unutar SqlDataReader objekta na sljedeći zapis ako postoji.

No idemo demonstrirati SqlDataReader za čitanje svih zapisa u tablici Osoba:

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;

namespace OsnoveSQL
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string connString = "SERVER=\\.\\SQLEXPRESS;DATABASE=Skripta;integrated security=true";
            using (SqlConnection conn = new SqlConnection(connString))
            {
                string cmdText = "SELECT * FROM Osoba";
                SqlCommand cmd = new SqlCommand(cmdText, conn);
                //Kreiranje SQLReader objekta izvršavanjem ExecuteReader metode
                SqlDataReader reader = cmd.ExecuteReader();
            }
        }
    }
}
```

```

//Provjera je li objekt kreiran
if (reader != null)
{
    //Provjera imamo li redaka za čitanje
    if (reader.HasRows)
    {
        //Dok reader ima podataka, čitamo ih i ispisujemo na konzolu
        while (reader.Read())
        {
            Console.WriteLine(reader["Prezime"].ToString() +
                " " + reader["Ime"]);
        }
    }
    reader.Close();
}
Console.ReadLine();
}
}

```

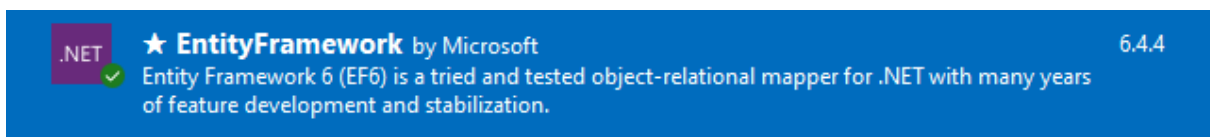
S ADO.NET-om možemo koristiti i DataSet objekte za eksplicitno dizajniranje za ujednačen pristup podacima bez obzira na vrstu izvora. DataSet može istovremeno pristupiti podacima SQL Server baze podataka i XML datoteci kroz unificirani podatkovni model. DataSet objekt omogućuje i privremenu lokalnu pohranu podataka.

5.2.3. Entity Framework

Entity Framework (EF) ORM je alat razvijen od strane Microsofta kako bi programerima omogućio rad s konceptualnim aplikacijskim modelom umjesto direktne komunikacije s bazama podataka. Primarno korištenjem Entity Frameworka eliminiramo potrebu za većinom koda potrebnog za pristup podacima. Kad govorimo o količini koda, s EF-om nema potrebe da svaki put navodimo konekcije, komande i zatvaranje istih. EF nam omogućuje kreiranje konceptualnog podatkovnog modela korištenjem C# koda ili ugrađenih alata za dizajniranje podatkovnog modela.

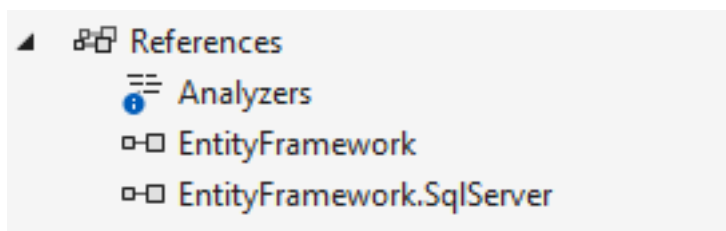
LINQ to SQL koristit ćemo da radimo upite prema kolekcijama objekata koje su mapirane pomoću Entity Frameworka na neki objekt u relacijskoj bazi podataka. Svaka tablica baze podataka predstavljena nam je zasebnom klasom unutar konceptualnog modela, a svaka komunikacija između klase i stvarne tablice odvija se u potpunosti automatizirano. LINQ to SQL u pozadini prevodi LINQ upit u SQL transakciju te tu transakciju šalje bazi podataka na izvršavanje. Kada baza podataka vrati rezultat, isti taj rezultat prevodi se u objekte aplikacije.

Entity Framework, ako se ne nalazi u projektu, instalirat ćemo preko NuGeta.



Slika 51. Entity Framework u NuGet package manageru

Jednom kad ste instalirali Entity Framework preko NuGet Package Managera, pod referencama pokazat će vam se dvije nove reference:



Slika 52. Reference na Entity Framework u projektu

Kad radimo mapiranje s Entity Frameworkom, imamo dva pristupa:

- Database First
- Code First.

Bazu ćemo koristiti na isti način, ali koncept pristupa EF-a dijametralno je suprotan.

Kod Database First pristupa podrazumijeva se da bazu podataka već imamo napravljenu, pa sve što trebamo jest napraviti odgovarajuće mapiranje prema bazi podataka.

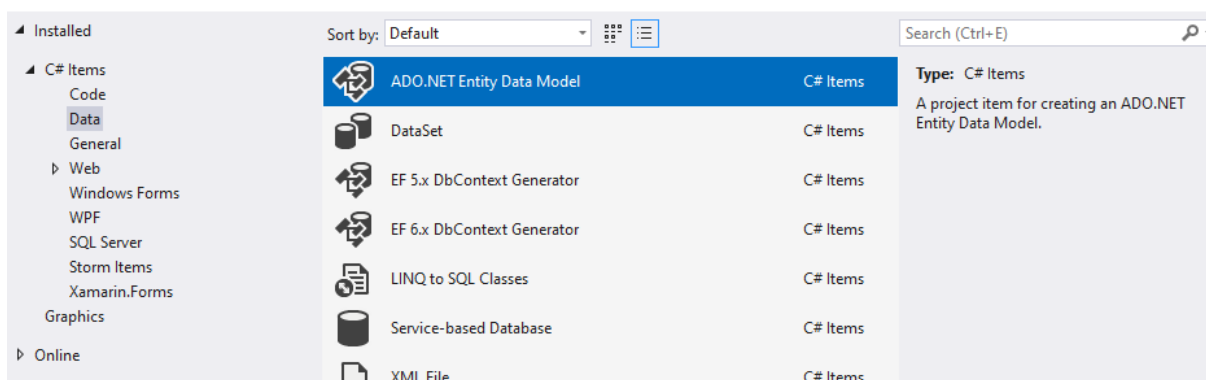
Kod Code First pristupa putem koda dizajniramo klase koje koristimo u programu, pa na temelju tih klasa izgrađujemo bazu podataka. Code First pristup, kad je došao, bio je potpuno nelogičan većini programera, jer generalno ovim pristupom možemo napraviti i koristiti bazu podataka bez da moramo ikada napisati ijednu liniju SQL koda.

Budući da nakon mapiranja koristimo EF na identičan način, idemo prvo proći kako mapirati podatke, pa ćemo demonstrirati kako ide korištenje EF-a.

Database First:

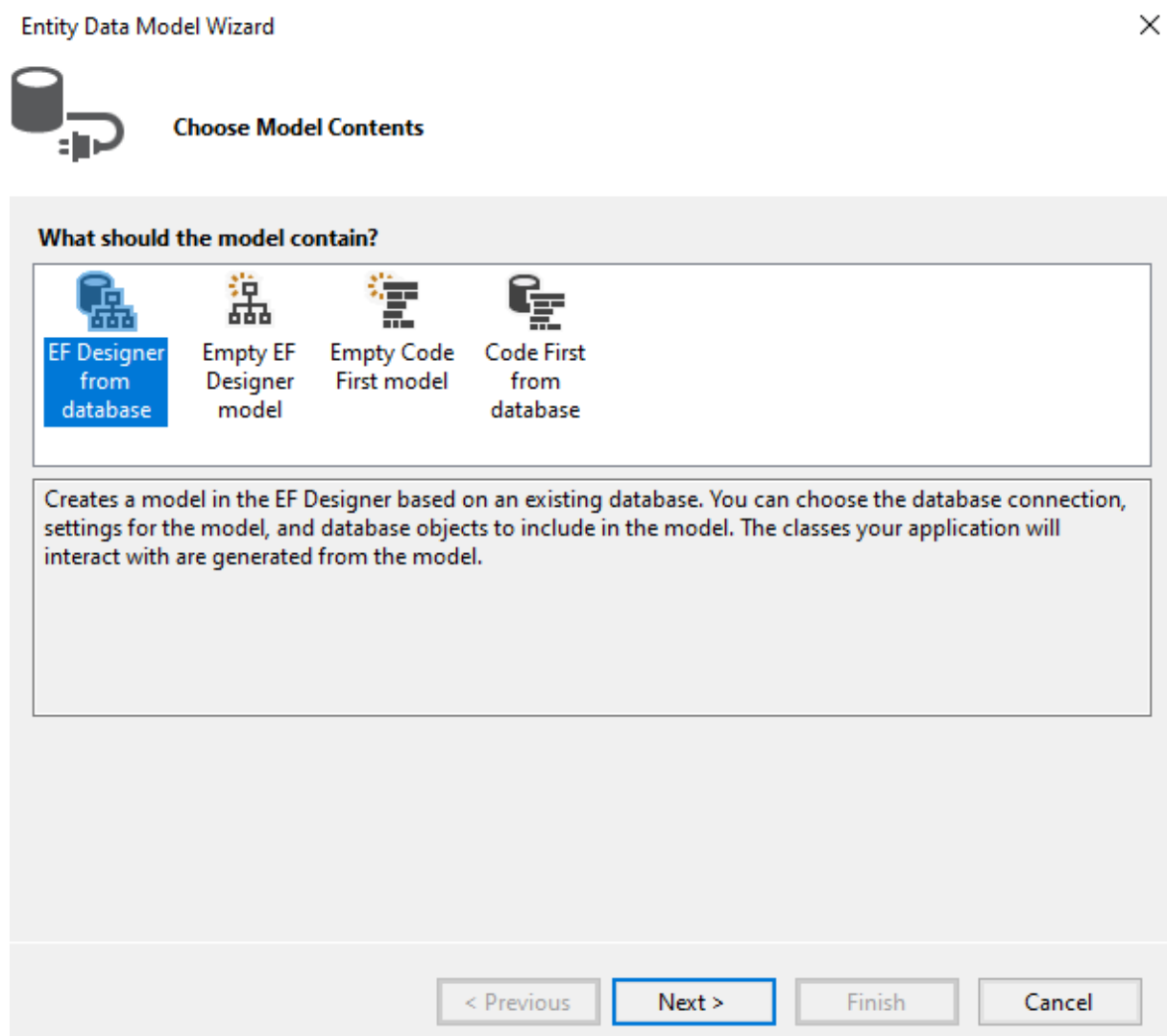
Kod **database first** pristupa, kako ime kaže, za početak imamo bazu podataka. U ovom pristupu mi već imamo bazu podataka, koju mapiramo prema objektima koje ćemo koristiti.

Mapiranje je jednostavno, prvo što moramo napraviti novi je ADO.NET Entity Data Model. Dodajemo ga tako da kliknemo desnom tipkom na projekt, zatim „Add”, pa „New Item”. U izborniku za dodavanje elemenata nađemo ADO.NET Entity Data Model pod Data dijelom pod C# Items izbornikom, kako je prikazano na slici 53.



Slika 53. Dodavanje ADO.NET Entity Data Modela

Nemojte zaboraviti promijeniti ime, ja ću imenovati ovaj element „DBFirstPristup”. Pojavit će se sljedeći prozor kao na slici 54.



Slika 54. Entity Data Model Wizard

Ovdje možete pročitati pojedinu opciju da vidite što znače. Mi ćemo odabrati prvu opciju, a to je „EF Designer from database” i kliknuti *Next*.

Na sljedećem prozoru tražit će vas vezu prema bazi podataka. Ukoliko već imate neki *connection string* u konfiguracijskoj datoteci, ovdje ga možete odabrati. No ako nemate, morat ćete ga konfigurirati klikom na *New Connection* vezu prema bazi podataka.



Choose Your Data Connection

Which data connection should your application use to connect to the database?

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Connection string:

☒ Save connection settings in App.Config as:

Slika 55. Određivanje veze na bazu podataka

Jednom kad u drugom prozoru konfigurirate vezu prema bazi (odaberete server, način autorizacije prema serveru i bazu podataka), automatski će vam se ispuniti ovaj prozor. Klikom na *Next* otvorit će vam se zadnji prozor, gdje odabirete objekte koje želite mapirati pomoću Entity Frameworka. Mapirati možete tablice, pogledе, funkcije i procedure, kao što je prikazano na sljedećoj slici. Mi ćemo jednostavno odabrati sve tablice koje smo prije kreirali u poglavlju 2, kad se radila baza podataka. No kasnije u primjerima nećemo koristiti sve tablice, ovo je samo da vidite kako je jednostavno mapiranje.



Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- ☒ Tables
 - ☒ dbo
 - ☐ Artiki
 - ☐ ArtikliNarudzbe
 - ☐ Narudzba
 - ☐ Osoba
 - ☐ Osobnalskaznica
 - ☐ sysdiagrams
 - ☐ Views
- ☒ Stored Procedures and Functions
 - ☒ dbo
 - ☐ fn_diagramobjects

☐ Pluralize or singularize generated object names

☒ Include foreign key columns in the model

☒ Import selected stored procedures and functions into the entity model

Model Namespace:

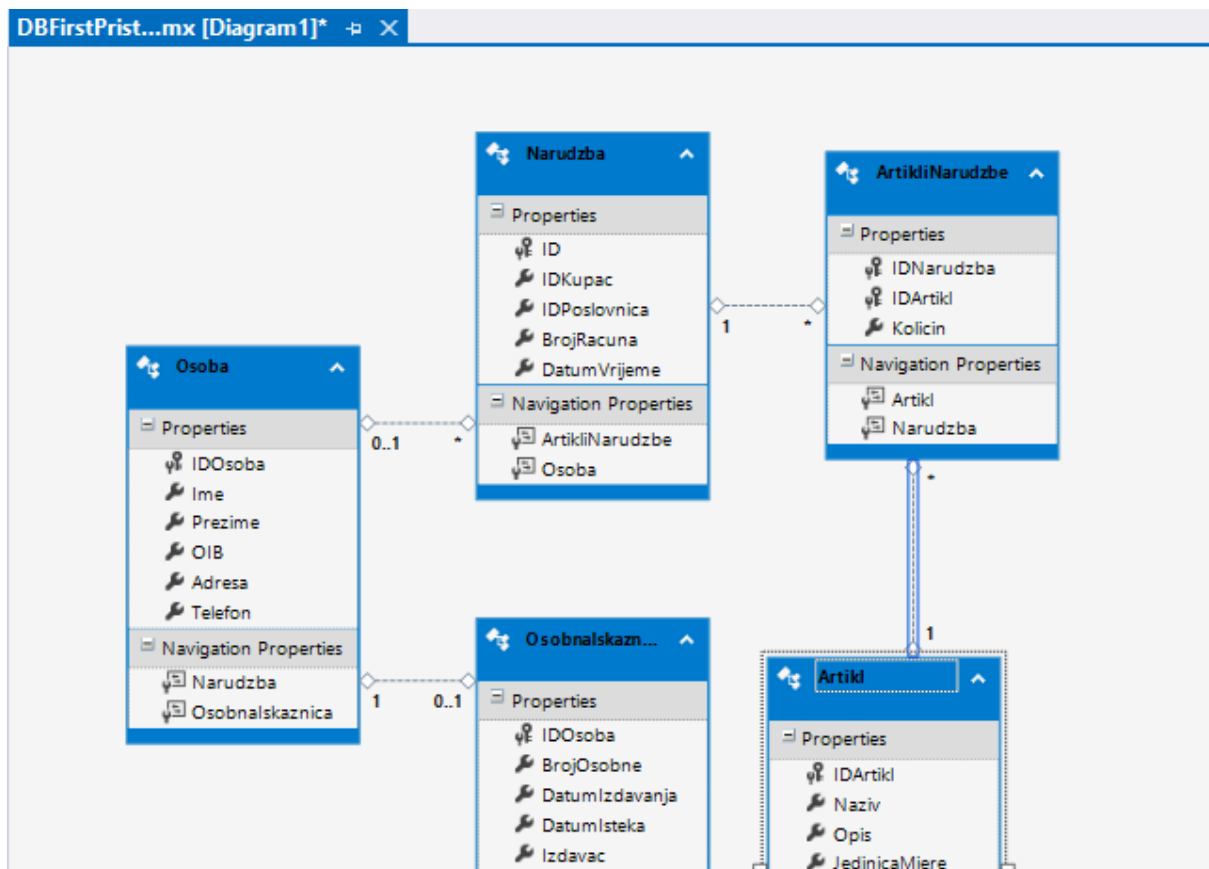
SkriptaModel

< Previous Next > **Finish** Cancel

Slika 56. Odabir objekata koje želimo mapirati

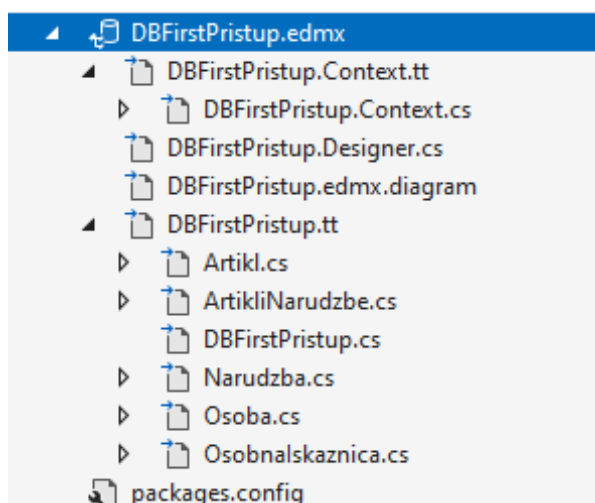
Kad odaberemo sve što trebamo, kliknemo na „Finish”. Samo da napomenemo, ne treba mapirati sistemske tablice (poput sysdiagrams) ili sistemske poglede, procedure i funkcije, jer nam generalno ne trebaju u radu.

Nakon nekoliko sigurnosnih upozorenja na koje odgovaramo s OK dobit ćemo grafički prikazano odabrane objekte kao na slici 57.



Slika 57. Mapirani objekti s relacijama

Raspored elemenata može varirati, ali i količina, ovisno o broju odabranih objekata koje želimo mapirati. Na ovom prozoru sačuvamo promjene (opet nekoliko eventualnih sigurnosnih poruka) i možemo ga zatvoriti. Sada, ako pogledamo u Solution Exploreru, vidjet ćemo da imamo u potpunosti mapirane odabrane objekte.



Slika 58. Pregled mapiranih objekata u Solution Exploreru

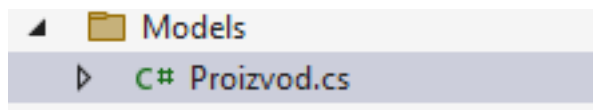
Entity framework automatski je napravio klase koje odgovaraju imenu tablica, a ako pogledamo unutar klase, svojstva odgovaraju imenu stupaca, ali i podatkovnom tipu pojedinog atributa iz baze podataka. Mapiranje je ovime gotovo u Database First pristupu.

Ukoliko kasnije imamo izmjena, odnosno ako je bilo izmjena na bazi podataka, sve što trebamo je otvoriti edmx datoteku, desno kliknuti i odabrati opciju *Update Model from Database* te pustiti da se osvježi mapiranje i sačuvati promjene.

Code First:

Kod Code First pristupa na temelju definiranih klasa konstruirat ćemo objekte u bazi podataka. Ovaj pristup zahtijeva malo više pisanja koda unutar projekta, no ako sve sami radimo, generalno se svodi na isto. Osobno mislim da ima dosta razloga zašto je bolji Database First pristup, koji će ionako prevladati ako radite kao dio razvojnog tima radi lakšeg odvajanja uloga.

U Code First pristupu radimo pristup podacima korištenjem modela. Zato ćemo u projekt dodati mapu „Models” i u tu mapu dodati novu klasu „Proizvod”, koju ćemo mapirati prema bazi podataka.



Unutar te klase dodat ćemo nekoliko svojstava, pa će klasa izgledati ovako:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace OsnoveSQL.Models
{
    public class Proizvod
    {
        public int ID { get; set; }
```

```

        public string Ime { get; set; }
        public string JedinicaMjere { get; set; }
    }
}

```

Sljedeće što moramo raditi jest dodati DbContext klasu koja upravlja entitet klasama ili modelom. To je središnji dio Entity Frameworka. DbContext klasa odgovorna je za spajanje na bazu podataka, izradu upita i ažuriranje baze podataka te sadržavanje informacija potrebnih za konfiguriranje baze podataka.

Mi ćemo napraviti vlastitu klasu koja će naslijediti DbContext klasu. U model mapi možemo napraviti EFContext klasu i napisati sljedeći kod u njoj:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;

namespace OsnoveSQL.Models
{
    public class EFContext:DbContext
    {
        private const string connString = "SERVER=.\SQLEXPRESS;DATABASE=Co-
deFirstBaza;integrated security=true";

        public EFContext():base(connString){ }

        public DbSet<Proizvod> Proizvodi { get; set; }
    }
}

```

Malo opisa koda. EFContext klasa nasljeđuje DbContext klasu. *Connection string* definirali smo kao privatnu varijablu, no inače to bismo čitali iz neke konfiguracijske datoteke, ali za ovaj primjer neka ovdje bude hardkodiran. Kod konstruktora praktički

zovemo konstruktor bazne klase i proslijeđujemo mu *connection string* na temelju kojeg ćemo se spajati na bazu podataka.

Stvaranje modela nije dovoljno za mapiranje na bazu podataka. Moramo napraviti DbSet svojstvo za svaki model unutar *context* klase. DbSet pruža metode poput dodavanja, uklanjanja i ostalih za svaki entitetski tip. *Context* klasa mapira te operacije u SQL naredbe i izvršava ih na bazi podataka.

Jednom kad je naš model spreman, moramo napraviti migraciju, odnosno definirane objekte stvoriti u bazi podataka. Ukoliko nemamo ni bazu podataka, ona će biti automatski napravljena.

Migraciju radimo preko „Package Manager Console” dodatka. Do Package manager konzole doći ćemo tako da kliknemo na Tools pa pod NuGetPackage Manager opcijom kliknemo na Package Manager konzolu.

Prvo upozorenje, sljedeće naredbe izvršavaju se samo jednom, nisu kao program da ih možemo više puta izvršavati.

Prvo moramo omogućiti migraciju, a to ćemo napraviti sljedećom naredbom:

```
PM> enable-migrations -ContextTypeName OsnoveSQL.Models.EFContext
```

Zatim moramo dodati migraciju, a to ćemo napraviti sljedećom naredbom:

```
PM> Add-Migration "CodeFirstBaza"
```

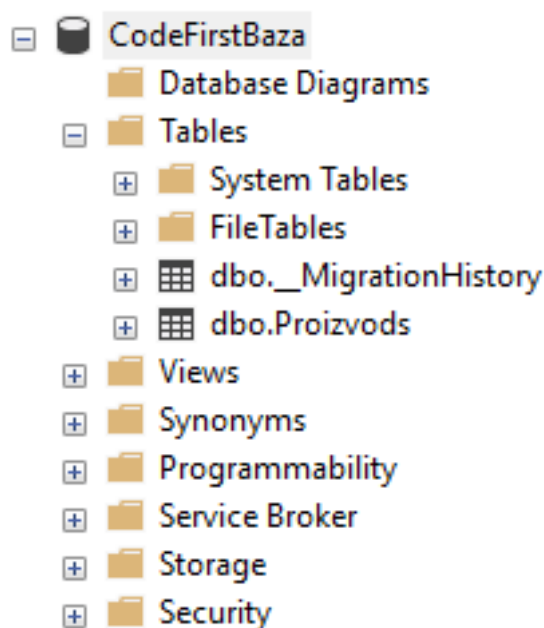
Add-Migration generira instrukcije da se generiraju SQL komande za ažuriranje baze podataka koja odgovara modelu.

Sljedeći je korak napraviti bazu podataka na temelju generiranih migracija. To ćemo napraviti sljedećom naredbom:

```
PM> update-database
```

Sad, ukoliko otvorimo SQL Server, vidjet ćemo da ako nismo imali bazu, baza je

napravljena s tablicama koje smo naveli u svojstvima s DbSet tipom:



Slika 59. Baza napravljena Code First pristupom

Ukoliko budemo imali dodatnih izmjena na klasama ili dodavanje novih klasa, sve što je potrebno jest dodati novi DbSet i izvršiti preko Package Manager konzole update-database naredbu.

Ovime je gotovo Code First mapiranje.

Rad s podacima preko Entiti Frameworka:

Raditi s podacima lagano je, možemo koristiti LINQ naredbe bez brige o konekcijama, readerima i slično. Sada s bazom radimo skoro kao sa svakom drugom kolekcijom objekata, budući da imamo pravilno mapirane tablice. Jedino oprez kad radite izmjene na bazi podataka, bilo da se radi o unosu, ažuriranju ili brisanju, mora se pozvati metoda SaveChanges.

Primjer pristupa podacima, gdje smo u istom projektu imali oba pristupa s bazama, koji pokazuje kako na jednostavan način možemo mapirati objekte prema više baza:

```

using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Linq;

namespace OsnoveSQL
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // konstrukcija db first Context objekta, ime dodijeljeno prema bazi
            var dbfirstdb = new SkriptaEntities();
            // konstrukcija code first Context objekta, poziv konstruktora Context
            // klase
            var codefirstdb = new Models.EFContext();

            //čitanje boja zapisa u bazi
            var osoba_u_bazi = (from o in dbfirstdb.Osoba
                               select o).ToList().Count;
            Console.WriteLine("Zapisa osoba u bazi: " + osoba_u_bazi);

            // dodavanje novog zapisa u tablicu:
            Osoba os = new Osoba { Ime = "Hrvoje", Prezime = "Horvat" };
            dbfirstdb.Osoba.Add(os);
            dbfirstdb.SaveChanges();

            //istovremeno preko codefirst varijable možemo raditi i s drugom bazom
            codefirstdb.Proizvodi.Add(new Models.Proizvod { Ime = "USB drive" });
            codefirstdb.SaveChanges();

            //ažuriranje po prezimenu Horvat, promjena u Hrvat:
            var osobe_azurirati = (from o in dbfirstdb.Osoba
                                   where o.Prezime == "Horvat"
                                   select o).ToList();
            foreach (var o in osobe_azurirati)
            {
                o.Prezime = "Hrvat"; // referentni podaci!
            }
        }
    }
}

```



```

        // moramo samo javiti contextu da je došlo do promjene na stanju
        // objekta
        dbfirstdb.Entry(o).State = System.Data.Entity.EntityState.Modified;
    }
    dbfirstdb.SaveChanges();
    Console.ReadLine();
}
}
}

```

Entity Framework ima još dosta dijelova koje nismo spomenuli, no zahvaljujući mapiranju mogu se očuvati relacije i održavati referentni integritet, kod modela u Code First pristupu mogu se dati atributi svojstvima tako da imamo 1 : 1 kontrolirano precrtavanje podatkovnih tipova i ograničenja...

5.3. Višedretveno programiranje

Multithreading ili **višedretvenost** sposobnost je procesora ili jedne jezgre unutar procesora da pruži izvršavanje više dretvi istovremeno. Ovaj pristup drukčiji je od višeprocenosti (više procesa koji se istovremeno izvršavaju, a često je jedan program vezan na jedan proces). U višedretvenim aplikacijama, dretve dijele resurse, dok kod procesa svaki dobije svoj resurs. Bit višedretvenog pristupa jest da istovremeno možemo paralelizirati izvođenje programa ukoliko ga možemo raščlaniti na nezavisne jedinice.

5.3.1. Sinkrono programiranje

Kod sinkronog programiranja govorimo o dretvama koje su ovisne o toku glavnog programa ili imaju neki mehanizam kontrole i sinkronizacije njihova rada.

Kad se proces pokrene, CLR automatski napravi jednu dretvu da izvršava aplikaciju.

Uz tu prvu dretvu, proces stvara jednu ili više dretvi da bi se izvršili dijelovi programa koji su asocirani s procesom. Te dretve mogu se izvršavati različitim prioritetima. Dodatno možemo koristiti ThreadPool klasu da bismo izvršavali kod na radnim dretvama koje su upravljane od strane CLR-a.

Životni ciklus dretve počinje kad objekt System.Threading.Thread klase bude kreiran, a završava kada se dretva prekine ili završi svoje izvođenje.

Dretva može biti u sljedećim stanjima:

- **nezapočeto stanje** – kad je dretva kreirana, no nije pokrenuta
- **spremno stanje** – dretva je spremna za izvršavanje i čeka procesorski ciklus
- **stanje izvršavanja** – dretva se izvršava
- **ne može se pokrenuti** – ne može se pokrenuti ako su pozvane metode spavanja ili čekanja ili je blokirana od strane neke operacije ulaza/izlaza
- **mrtvo** – dretva je završila izvođenje ili je izvođenje prekinuto.

U C#-u klasa System.Threading.Thread zadužena je za rad s dretvama. Omogućuje stvaranje i pristup individualnim dretvama u višedretvenoj aplikaciji. Prva dretva koja se izvršava glavna je dretva (main). Kad C# program krene s izvođenjem, ta glavna dretva automatski se napravi. Možemo joj pristupiti preko CurrentThread svojstva u Thread klasi.

Svojstva Thread klase koja se najčešće koriste:

- **CurrentContext** – dohvat trenutnog sadržaja u kojem se dretva izvršava
- **CurrentCulture** – dohvat ili postavljanje trenutne kulture trenutne dretve
- **CurrentPrinciple** – koristi se za sigurnost baziranu na ulogama
- **CurrentThread** – dohvat dretve koja se trenutno izvodi
- **ExecutionContext** – dohvaća informacije konteksta izvođenja trenutne dretve
- **IsAlive** – vraća vrijednost trenutnog statusa izvršavanja trenutne dretve
- **Isbackground** – povrat ili postavljanje vrijednosti koja govori je li dretva

pozadinska

- **IsThreadPoolThread** – dohvat vrijednosti koja pokazuje pripada li dretva upravljanoj bazenu dretvi
- **Name** – vraća ili postavlja ime dretve
- **Priority** – vraća ili postavlja vrijednost koja označava prioritet izvršavanja dretve
- **ThreadState** – vraća vrijednost koja sadrži stanje trenutne dretve.

Metode Thread klase koje se najčešće koriste:

- **Abort** – baca ThreadAbortException u dretvi na kojoj je pozvana da započne proces eliminacije dretve
- **BeginCriticalRegion** – obavještava hosta da izvršavanje ulazi u kritičan dio gdje prekidanje dretve može uzrokovati iznimku koja se neće obraditi ili može ugroziti druge zadatke u procesu
- **BeginThreadAffinity** – obavještava da treba izvršiti instrukcije koje ovise o trenutnoj dretvi operativnog sustava
- **EndCriticalRegion** – obavještava da završava kritični dio koda gdje prekidanje dretve može uzrokovati iznimku koja se neće obraditi ili može ugroziti druge zadatke u procesu
- **EndThreadAffinity** – obavještava da treba završiti izvršavanje instrukcija koje ovise o trenutnoj dretvi operativnog sustava
- **GetDomain** – vraća domenu na kojoj se dretva trenutno izvodi
- **Interrupt** – prekida dretvu da bude u stanju čekanja
- **Join** – čeka se da dretva završi izvođenje blokirajući daljnje izvršavanje dretve koja ju je pozvala
- **ResetAbort** – otkazuje zahtjev za prekid trenutne dretve
- **Start** – započinje dretvu
- **Sleep** – pauzira dretvu na neki vremenski period
- **SpinWait** – uzrokuje da dretva čeka određeni ciklus vremena
- **Yield** – trenutna dretva prepušta izvođenje drugoj, koja je spremna za pokretanje na trenutnom procesu. Selekcija procesa na čekanju ostavljena je operacijskom sustavu.

Dretve se stvaraju tako da se proširuju Thread klase. Proširene klase zatim zovu Start metode da bi započele izvođenje.

Primjer koncepta:

```
using System;
using System.Threading;

namespace ThreadingAplikacija
{
    internal class Program
    {
        public static void PozivPoddretvi()
        {
            Console.WriteLine("Pod dretva započela");
        }
        static void Main(string[] args)
        {
            ThreadStart pod = new ThreadStart(PozivPoddretvi);
            Console.WriteLine("U glavnoj: stvaram poddretve");
            Thread poddretva=new Thread(pod);
            poddretva.Start();
            Console.ReadLine();
        }
    }
}
// Ispis:
/*
U glavnoj: stvaram poddretve
Poddretva zapocela
*/
```

Pri upravljanju dretvama možemo koristiti mnogobrojne metode Thread klase. Sljedeći primjer pokazuje kako dretva (čak i glavna) može čekati određeni period:

```

using System;
using System.Threading;

namespace ThreadingAplikacija
{
    internal class Program
    {
        public static void PozivPoddretvi()
        {
            Console.WriteLine("Pod dretva započela");
            int spavaj = 5000; //milisekunde

            Console.WriteLine("Poddretva pauzirana {0} sekundi", spavaj / 1000);
            Thread.Sleep(spavaj);
            Console.WriteLine("Poddretva nastavlja izvođenje");
        }
        static void Main(string[] args)
        {
            ThreadStart pod = new ThreadStart(PozivPoddretvi);
            Console.WriteLine("U glavnoj: stvaram poddretve");
            Thread poddretva=new Thread(pod);
            poddretva.Start();
            Console.ReadLine();
        }
    }
}

// Ispis:
/*
U glavnoj: stvaram poddretve
Poddretva zapocela
Poddretva pauzirana 5 sekundi
Poddretva nastavlja izvođenje
*/

```

Potrebno je još spomenuti – metoda Abort koristi se da zaustavimo dretvu. Pri prekidu dretve metodom Abort dretva baca ThreadAbortException. Ova iznimka ne može se uhvatiti, kontrola se šalje u *finally* blok ako ga ima, ako pokušamo hvatati iznimke tijekom izvođenja dretvi. Primjer korištenja Abort i Join metoda:

```
using System;
using System.Threading;

namespace ThreadingAplikacija
{
    internal class Program
    {
        public static void Main()
        {
            Thread newThread = new Thread(new ThreadStart(TestMetoda));
            newThread.Start();
            Thread.Sleep(1000);

            // Abort newThread
            Console.WriteLine("Main prekida novu dretvu.");
            newThread.Abort("Informacije iz Main metode.");

            // Čekanje da dretva završi.
            newThread.Join();
            Console.WriteLine("Nova dretva prekinuta - Main gotov");
            Console.ReadLine();
        }

        static void TestMetoda()
        {
            try
            {
                while (true)
                {
                    Console.WriteLine("Nova dretva se izvodi.");
                    Thread.Sleep(1000);
                }
            }
        }
    }
}
```

```

        catch (ThreadAbortException abortException)
        {
            Console.WriteLine((string)abortException.ExceptionState);
        }
    }
}

```

// Ispis:

/*

Nova dretva se izvodi.

Main prekida novu dretvu.

Informacije iz Main metode.

Nova dretva prekinuta - Main gotov

***/**

U primjeru pokušavamo uhvatiti `ThreadAbortException`, no kako vidimo iz ispisa, ne možemo uhvatiti prekid izvođenja dretve.

Ukoliko želimo proslijediti metodu koja je parametrizirana, ne radimo konstrukciju `ThreadStart`, nego `ParametrizedThreadStart`, pomoću čije konstrukcije možemo proslijediti jedan objekt u metodu koja će biti pokrenuta unutar nove dretve.

Prosljeđivanje parametra radimo preko `Start` metode u dijelu programa gdje pokrećemo dretvu.

U sljedećem primjeru pokazat ćemo kako dretvama proslijediti parametre, no i kako svaka dretva može pristupiti zajedničkim podacima, u ovom slučaju statičnoj varijabli:

```

using System;

```

```

using System.Threading;

```

```

namespace ThreadingAplikacija

```

```

{

```

```

    internal class Program

```

```

    {

```

```

public static int broj = 0;

public static void Main()
{
    Thread t1 = new Thread(new ParameterizedThreadStart(TestMetoda));
    t1.Name = "Dretva 1";
    Thread t2 = new Thread(new ParameterizedThreadStart(TestMetoda));
    t2.Name = "Dretva 2";
    Thread t3 = new Thread(new ParameterizedThreadStart(TestMetoda));
    t3.Name = "Dretva 3";

    Console.WriteLine("Main, prije pokretanja dretvi broj="+broj.ToString());
    Console.WriteLine("Pokretanje svih dretvi!");
    t1.Start(50);
    t2.Start(60);
    t3.Start(80);

    Thread.Sleep(5000);
    Console.ReadLine();
}

static void TestMetoda(object cekaj)
{
    for (int i = 0; i < 5; i++) {
        broj++;
        Console.WriteLine("Dretva " + Thread.CurrentThread.Name + " provecala
broj na " + broj.ToString());

        Thread.Sleep((int)cekaj);
    }
}
}

// Ispis:
/*
Main, prije pokretanja dretvi broj=0
Pokretanje svih dretvi!

```



```
Dretva Dretva 1 provecala broj na 2
Dretva Dretva 2 provecala broj na 2
Dretva Dretva 3 provecala broj na 3
Dretva Dretva 1 provecala broj na 4
Dretva Dretva 2 provecala broj na 5
Dretva Dretva 3 provecala broj na 6
Dretva Dretva 1 provecala broj na 7
Dretva Dretva 2 provecala broj na 8
Dretva Dretva 2 provecala broj na 9
Dretva Dretva 3 provecala broj na 11
Dretva Dretva 1 provecala broj na 10
Dretva Dretva 2 provecala broj na 12
Dretva Dretva 1 provecala broj na 13
Dretva Dretva 3 provecala broj na 14
Dretva Dretva 3 provecala broj na 15 Main prekida novu dretvu.
Informacije iz Main metode.
Nova dretva prekinuta - Main gotov
*/
```

Višedretveno programiranje ima smisla u dosta slučajeva, pogotovo kad znate da možete paralelizirati izvođenje dijelova programa na svojevrsno neovisne blokove. Jedan od primjera jest da pustite da se korisničko sučelje učitava dok paralelno učitate podatke iz nekih drugih izvora ili kad imate velike setove podataka za koje znate da ih možete neovisno obrađivati. Danas računala imaju jako velik broj jezgri koje treba iskoristiti. Na programeru je da paralelizacijom programa optimizira izvođenje programa da bi bio fluidniji i brži.

5.3.2. Asinkrono programiranje

Asinkrono programiranje odnosi se na nastanak događaja neovisnog o glavnom toku programa. To mogu biti vanjski događaji poput dolaska signala ili akcija koje je pokrenuo neki dio programa da se dogodi bez čekanja rezultata, odnosno da programski dio koji pokreće taj kod ne čeka rezultat.

Standardni način rada s asinkronosti jest da u programskom sučelju napravimo pot-

program koji vraća budućnost ili obećanje koje predstavlja operaciju u tijeku i sinkronizacijsku operaciju koja će blokirati sve dok se budućnost ili obećanje ne završi.

Primjeri asinkronosti uključuju:

- asinkroni poziv procedura
- AJAX pozivi
- AMD (Asynchronous Method Dispatch).

U C# programskom jeziku asinkrono programiranje postićeno s `async` i `await`, korištenjem „zadatkovno asinkronog programskog modela”.

Task asynchronous programming model (*TAP – Task-based Asynchronous Parttern*) pruža apstrakciju asinkronog koda. U ovom modelu kod pišemo kako smo standardno i radili, ali kompajler napravi potrebne transformacije jer neke naredbe mogu početi neki posao i vratiti zadatak (*task*) koji predstavlja posao koji se izvodi.

Cilj ove sintakse omogućiti je kod koji izgleda kao i „običan”, ali izvodi se na mnogo kompliciranijim redoslijedom baziranim na vanjskim alokacijama resursa i kad su zadaci koje radi gotovi. Analogija je kako ljudi daju instrukcije za procese koji uključuju asinkrone zadatke.

Rezultat asinkronog programiranja mogli bismo postići i sa sinkronim, pomoću dretvi (*Threading*), no budući da često ne znamo koliko će trajati izvođenje (gdje bi onda trebalo pauzirati glavnu dretvu gdje se program izvodi) i s dretvama trebamo koristiti neki komplicirani oblik dohvata rezultata, lakše je koristiti asinkrono programiranje sa zadacima (*tasks*).

Kod asinkronog programiranja za svaku metodu za koju želimo da ima mogućnost asinkronog izvođenja moramo je deklarirati s ključnom riječi *async*. Idemo pogledati na primjeru koda:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AsinkronoProgramiranje
{
    internal class Program
    {
        static async Task Main(string[] args)
        {
            DugiProces();
            KratkiProces();
            Console.ReadLine();
        }

        static async void DugiProces()
        {
            Console.WriteLine("DugiProces počeo");
            await Task.Delay(4000); //nešto dugo radi
            Console.WriteLine("DugiProces završio");
        }

        static void KratkiProces()
        {
            Console.WriteLine("KratkiProces počeo");
            //nešto kratko radi...
            Console.WriteLine("KratkiProces završio");
        }
    }
}

// Ispis:
/*
DugiProces poceo
KratkiProces poceo
KratkiProces završio
DugiProces završio
*/

```

U ovom primjeru Main metoda označena je async ključnom riječi i povratni je tip Task. Zahvaljujući tome metoda je označena kao asinkrona. Sve metode koje su označene ključnom riječi async mogu se asinkrono izvesti. DugiProces metoda je označena kao asinkrona i unutar nje pokrećemo Task.Delay da zadrži proces 4 sekunde u izvođenju. Ključna riječ await označava da ćemo čekati da se metoda izvede prije nastavka izvođenja ostalog dijela naredbi. Za await možemo reći da je slično kao da u sinkronom programiranju čekamo da se Thread izvede Join metodom.

Kad se program počne izvršavati iz async Main metode unutar glavne aplikacijske dretve, on pokreće dvije metode. Metoda async DugiProces izvodi se u zasebnoj dretvi i glavna aplikacijska dretva nastavlja s izvođenjem sljedeće naredbe koja poziva KratkiProces metodu. DugiProces se ne čeka da se završi, nego jednostavno pušta da se izvršava.

Upotrebljavamo async uz await i Task ako asinkrona metoda vraća vrijednost dijelu programa koji je poziva. Ključna riječ await čeka asinkronu metodu da vrati neku vrijednost. Dio koda koji poziva na ovaj način metodu, pauzira svoje izvođenje dokle god kod ne dobije vrijednost koju čeka.

Klasa Task predstavlja asinkrone operacije i Task<TResult> generička klasa predstavlja operacije koje će vratiti neku vrijednost. U prijašnjem primjeru koristili smo await Task.Delay(4000), koja je počela asinkronu operaciju koja spava 4 sekunde, a zahvaljujući ključnoj riječi await čekamo da se zadatak izvrši prije nego što nastavimo dalje s izvršavanjem ostatka koda.

U sljedećem primjeru čekat ćemo povratnu vrijednost:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AsinkronoProgramiranje
{
```

```

internal class Program
{
    static async Task Main(string[] args)
    {
        Task<int> rezultat = DugiProces();
        KratkiProces();
        var val = await rezultat; // čekaj povratnu vrijednost
        Console.WriteLine("Rezultat: {0}", val);
        Console.ReadKey();
    }

    static async Task<int> DugiProces()
    {
        Console.WriteLine("DugiProces počeo");
        await Task.Delay(4000); //nešto dugo radi
        Console.WriteLine("DugiProces završio");
        return 10;
    }

    static void KratkiProces()
    {
        Console.WriteLine("KratkiProces počeo");
        //nešto kratko radi...
        Console.WriteLine("KratkiProces završio");
    }
}

// Ispis:
/*
DugiProces poceo
KratkiProces poceo
KratkiProces završio
DugiProces završio
Rezultat: 10
*/

```

U gornjem primjeru u DugiProces metodi Task<int> koristi se da povratni tip bude int. Async metoda treba vratiti void, Task ili Task<TResult>, gdje je TResult povratni tip async metode. Nevraćanje vrijednosti (void) inače se koristi za rukovatelje događajem. Ključna riječ async omogućuje nam da koristimo await ključnu riječ unutar metode tako da možemo čekati asinkronu metodu da se završi da možemo nastaviti izvoditi druge metode koje su ovisne o rezultatu.

Ako imamo više async metoda koje vraćaju vrijednost, možemo koristiti await za sve metode.

Asinkrono programiranje često ćemo koristiti za pristup web-servisima kad čitamo ili šaljemo podatke. No ne moraju to biti samo servisi; .NET u svojim novijim verzijama ima velik broj metoda koje su predviđene za to da rade asinkrono s await ključnom riječi. Prisjetimo se primjera s JSON-om kad pristupamo asinkrono podacima nekog web-servisa:

```
using System;
using System.Collections.Generic;
using System.Net.Http.Json;

namespace Aplikacija
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Username { get; set; }
        public string? Email { get; set; }
    }

    class Program
    {
        public static async Task Main()
        {
            using HttpClient client = new()
            {
                BaseAddress = new Uri("https://jsonplaceholder.typicode.com")
            }
        }
    }
}
```

```

};

// Dohvat korisničkih informacija.
User? user = await client.GetFromJsonAsync<User>("users/1");
Console.WriteLine($"Id: {user?.Id}");
Console.WriteLine($"Name: {user?.Name}");
Console.WriteLine($"Username: {user?.Username}");
Console.WriteLine($"Email: {user?.Email}");

// Post novog korisnika.
HttpResponseMessage response = await client.PostAsJsonAsync("users",
user);
Console.WriteLine(
    $"{{(response.IsSuccessStatusCode ? "Success" : "Error")}} - {{response.
StatusCode}}");
    }
}
}

```

Koju metodu rada odabrati? Sinkrono ili asinkrono programiranje? Izbor će doslovno ovisiti od slučaja do slučaja, odnosno koliku kontrolu želite imati ili koliko jednostavno želite raditi.

5.4. Parcijalne klase

Moguće je odvojiti definiciju klase, strukture, sučelja ili metode u dvije ili više datoteka. Svaka datoteka mora sadržavati sekciju tipa ili definiciju metode i svi dijelovi kombiniraju se kad se aplikacija kompajlira.

Postoji više situacija gdje je razdvajanje definicije klase poželjno:

- kad radimo na velikim projektima, podijeliti klasu na više datoteka omogućuje da više programera radi na njoj istovremeno
- kad radimo s automatski generiranim kodom, kod se može dodavati na klasu bez da moramo rekreirati izvorni kod. Visual Studio koristi ovaj

pristup kada stvara Windows forme, wrapper kod *web*-servisa itd. Na takav način stvaramo kod koji koristi ove klase bez da moramo modificirati datoteke koje nam Visual Studio stvara

- kad koristimo generatore koda, da generiramo dodatnu funkcionalnost u klasi.

Da bismo razdijelili definiciju klase, moramo koristiti ključnu riječ `partial` kao u sljedećem primjeru:

```
public partial class Radnik
{
    public void Radi() {}
}
```

```
public partial class Radnik
{
    public void Pauziraj() {}
}
```

Ključna riječ `partial` indikator je da se drugi dijelovi klase, strukture ili sučelja mogu definirati unutar istog imenskog prostora. Svi dijelovi moraju imati ključnu riječ `partial` i svi dijelovi moraju biti dostupni za vrijeme kompajliranja da bi formirali finalni tip. Svi dijelovi moraju imati istu razinu pristupa (`public`, `private` i slično).

Ako je bilo koji dio apstraktan, tada se cijeli tip smatra apstraktnim. Ako je bilo koji dio deklariran kao `sealed`, tada se cijeli tip smatra `sealed`. Ako bilo koji dio deklarira bazni tip, tada cijeli finalni tip nasljeđuje taj bazni tip.

Svi dijelovi koji specificiraju baznu klasu moraju se složiti s time. Dijelovi mogu specificirati različita bazna sučelja i finalni tip implementira sva sučelja svih parcijalnih deklaracija. Svaka klasa, struktura ili sučelje članova deklariranih u parcijalnoj definiciji na raspolaganju su svim dijelovima. Finalni tip kombinacija je svih dijelova prilikom kompajliranja!

Korištenje `partial` ključne riječi nije moguće na delegatima ili deklaracijama enumeratora.

Tijekom kompajliranja atributi parcijalnih klasa spajaju se. Npr. sljedeće deklaracije:

```
[SerializableAttribute]  
partial class Mjesec { }
```

```
[ObsoleteAttribute]  
partial class Mjesec { }
```

Ekvivalentne su sljedećoj deklaraciji:

```
[SerializableAttribute]  
[ObsoleteAttribute]  
class Mjesec { }
```

Sljedeće se sjedinjuje iz svih parcijalnih definicija:

- XML komentari
- sučelja
- atributi generičko tipiziranih parametara
- klasni atributi
- članovi.

Npr. sljedeća deklaracija:

```
partial class Earth : Planet, IRotacijaOkoOsi { }  
partial class Earth : IRotacijaOkoSunca { }
```

Ekvivalentna je sljedećoj:

```
class Earth : Planet, IRotacijaOkoOsi, IRotacijaOkoSunca { }
```

Postoji nekoliko pravila kojih se moramo pridržavati kad radimo s parcijalnim klasama:

- sve definicije parcijalnih tipova koje trebaju biti dio istog tipa moraju imati partial modifikator. Ne možemo napraviti sljedeće:

```
public partial class A { }
```

```
//public class A { } // Error, mora biti partial!!!
```

- modifikator partial može se pojaviti ispred ključne riječi class, struct ili interface
- ugniježđeni parcijalni tipovi dozvoljeni su u parcijalnim definicijama
- sve definicije parcijalnih tipova koje trebaju biti dio istog tipa moraju biti definirane unutar istog modula (exe ili dll datoteka). Parcijalne definicije ne mogu se razdijeliti na više modula!
- ime klase i generički tipiziranih parametara mora odgovarati na svim parcijalnim definicijama. Generički tipovi mogu biti parcijalni, gdje svaka parcijalna deklaracija mora koristiti isto imenovane parametre u istom redoslijedu
- ključne riječi na parcijalnim definicijama opcionalne su, ali ako su prisutne na jednoj parcijalnoj definiciji, ne mogu biti u konfliktu s ključnim riječima na drugoj parcijalnoj definiciji (public, private, protected, internal, abstract, sealed, base class).

5.4.1. Nesiguran kod

Nesiguran kod (*unsafe code*) mogućnost je da u C# programskom jeziku deklariramo i radimo s pokazivačima. Pokazivači su posebno deklarirane varijable koje sadrže memorijske adrese tipa na koji se referiraju, odnosno možemo reći da pokazuju gdje se u memoriji točno nalazi podatak koji je tipa kojeg je pokazivač. C# generalno ne

koristi pokazivače, nego umjesto njih ima reference i mogućnost kreiranja objekata koji su upravljani „garbage collectorom” (smetlarom). Zbog toga je C# sigurniji za korištenje nego C ili C++ programski jezik. U C# jeziku nije moguće imati neinicijaliziranu varijablu, koja bi se klasificirala kao „viseći pokazivač” ili izraz koji bi indeksirao polje izvan njegova dosega i time efikasno ulazio u memoriju koja je moguće rezervirana za drugi program. Pokazivači i rad s memorijom vjerojatni su krivac za mnogo grešaka u C i C++ programima, pogotovo kod početnika, koje su eliminirane u C# programskom jeziku. No pokazivače u nekim slučajevima moramo koristiti, npr. da bismo pristupili dijelovima operativnog sustava, pristupili nekom uređaju koji ima mapiranje u memoriji, implementirali neki algoritam koji je vremenski kritičan itd.

Nesigurne značajke C#-a moguće su samo u nesigurnom kontekstu, koji se defini-
ra korištenjem **unsafe** modifikatora podatkovnog tipa ili člana, ali i preko nesigurnih naredbi.

Deklaracija klase, strukture, sučelja ili delegata može sadržavati *unsafe* modifikator i u tom slučaju sav dio te deklaracije smatra se dijelom nesigurnog konteksta.

Deklaracija člana, metode, svojstva, događaja, indeksera, operatora, instance ili statičnog konstruktora može sadržavati *unsafe* modifikator i u tom slučaju sve što se nalazi u implementaciji smatra se dijelom nesigurnog konteksta.

Nesigurne naredbe omogućuju korištenje nesigurnog konteksta unutar bloka naredbi.

Nekoliko primjera:

```
// Cijela klasa postaje nesiguran kontekst
// Čime možemo deklarirati Lijevo i Desno kao pokazivače
public unsafe struct Cvor
{
    public int Vrijednost;
    public Cvor* Lijevo;
    public Cvor* Desno;
}
```

//unsafe modifikatori na deklaraciji člana

```
public struct Cvor  
{unsafe  
    public int Vrijednost;  
    public unsafe Cvor* Lijevo;  
    public Cvor* Desno;  
}
```

Modifikator `unsafe` samo stvara područje nesigurnog konteksta i time dozvoljava korištenje pokazivača, ali `unsafe` modifikator nema utjecaj na tip ili na samog člana.

```
public class A  
{  
    public unsafe virtual void F()  
    {  
        char* p;  
        ...  
    }  
}
```

```
public class B : A  
{  
    public override void F()  
    {  
        base.F();  
        ...  
    }  
}
```

Unsafe modifikator na F metodi u A klasi jednostavno napravi da cijelo tijelo F metode postane nesiguran kontekst u kojem se mogućnosti nesigurnog koda mogu koristiti. U premošćivanju F metode B metodom nema potrebe ponovno napomenuti `unsafe` modifikator, osim ako i u F metodi ne trebamo pristup nesigurnim mogućnostima.

Kad *unsafe* modifikator koristimo u parcijalnim deklaracijama tipova, samo za taj dio koda smatra se da je nesiguran kontekst.

Unutar nesigurnog konteksta tip može biti pokazivač kao i vrijednost, referentni tip ili tipski parametar. U nesigurnom kontekstu pokazivači mogu biti elementi tipa polja.

Pokazivači se definiraju tako da kažemo podatkovni tip na koji pokazuju, pa zatim uz podatkovni tip napišemo zvjezdicu i onda ime varijable, kao u sljedećem primjeru:

```
int* pi, pj; // NE kao u C/C++: int *pi, *pj;
```

Tip prije zvjezdice * referentni je tip pokazivačkog tipa. Ukazuje na koji će tip varijable vrijednost pokazivača pokazivati. Naspram referenci, pokazivači nisu praćeni garbage collectorom, odnosno on nema pojma o pokazivačima i o podacima na koje oni pokazuju. Zbog toga pokazivačima nije dozvoljeno da pokazuju na reference ili na strukture koje sadrže referencu.

Vrijednost pokazivača koji ima tip T* predstavlja adresu varijable koja je tipa T. Pomoću * možemo pristupiti ovoj varijabli. Ako želimo postaviti vrijednost na neku adresu, možemo to učiniti ovako:

```
*pi = 6;
```

Kao i referenca objekta, pokazivač isto može biti null. Pokazivač s null vrijednosti onaj je koji ima sve bitove postavljene na nulu.

Deklaracija void* predstavlja pokazivač koji pokazuje na nepoznati tip. Na pokazivaču na nepoznati tip ne mogu se izvršavati operacije nikakvog tipa, no on se može pretvoriti u neki drugi tip na kojem se mogu raditi operacije.

Pokazivački tipovi posebna su kategorija tipova jer naspram referentnih tipova i vrijednosnih tipova pokazivački tipovi ne nasljeđuju od object klase i ne postoji konverzija između pokazivačkih tipova i objekata.

Pokazivači su sami po sebi opsežni i velikom broju programera komplicirani. No većina C# programera prijeđe na C# programski jezik s C++ jezika jer ne žele raditi s pokazivačima.

Raditi općenito s nesigurnim kodom nije preporučljivo i treba ga izbjegavati ako je moguće, no za situacije gdje je to neophodno, dobro je znati da se može.

5.5. Web-servisi

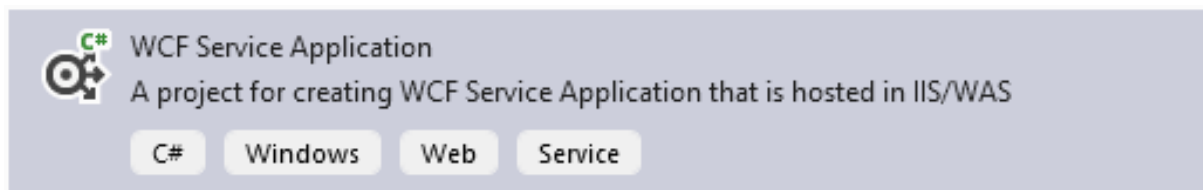
Web-servisi pojednostavljeno su jedinice funkcionalnosti čije je funkcije moguće konzumirati putem interneta. Naspram *web-stranica*, koje su napravljene da ljudima budu čitljive, *web-servisi* prvenstveno su napravljeni za čitanje od strane drugih programa. *Web-servise* najčešće konzumiraju druge *web-stranice* ili druge aplikacije koje se spajaju da zatraže podatke, koje zatim prikazuju korisniku.

Web-servisi u današnje se vrijeme nalaze u gotovo svim popularnim aplikacijama koje zahtijevaju pristup internetu. Pogodni su u situacijama kada želimo imati jedan pozadinski sustav koji bi usluživao različite aplikacije na različitim platformama. Tako na ovakav način imamo npr. Facebook aplikaciju, koja radi na svim mobilnim operativnim sustavima preko svoje aplikacije, preko weba, desktop aplikacija i ostalo. Ukratko, *web-servis* zaslužan je za primanje, slanje i obradu podataka, dok je aplikacija s druge strane namijenjena samo da korisniku prikaže te podatke.

5.5.1. WCF

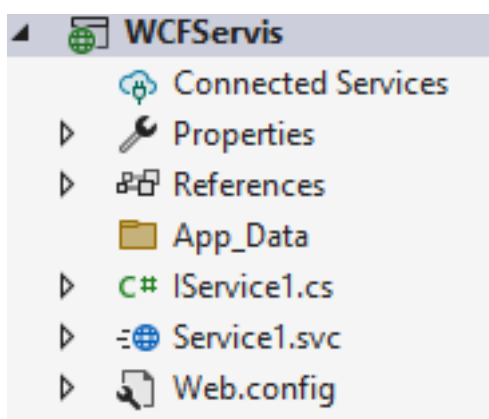
WCF (*Windows Communication Foundation*) niz je komponenti ugrađenih u Microsoft .NET framework namijenjen interoperabilnosti između različitih sustava. Dizajniran je u skladu s načelima servisno orijentirane arhitekture (SOA) za potporu distribuiranog računarstva gdje klijentske aplikacije koriste podatke drugih aplikacija ili drugih sustava. Funkcionalnost WCF-a izgrađena je oko servisa od kojih su danas najrašireniji *web-servisi* te su oni danas standard razmjene podataka između različitih sustava. Takvi servisi obično se koriste WSDL sučeljem (*Web Service Description Language*) kako bi korisnici servisa znali koje su i na koji način dostupne metode jednog *web-servisa*. WCF koristi mnoge napredne standarde protokola SOAP (*Simple Object Access Protocol*), kao što su WS-Addressing, WS-ReliableMessaging i WS-Security.

Rad s WCF-om najlakše je demonstrirati primjerom. Kreirat ćemo projekt iz predloška „WCF Service Application”.



Slika 60. WCF projekt predložak

Nakon odabira imena za projekt struktura projekta treba izgledati kao na slici 61.

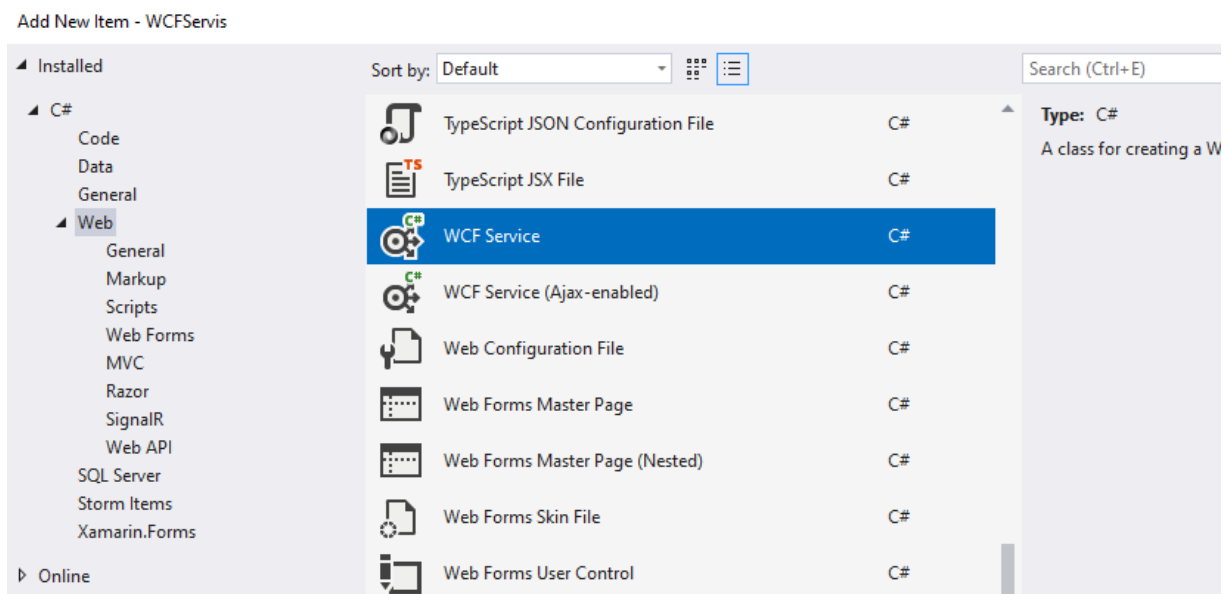


Slika 61. Struktura WCF projekta

U samom startu Visual Studio generirao nam je osnovne datoteke za izradu servisa, čak je generirao jedan servis koji se sastoji od IService1.cs datoteke i Service1.svc datoteke. Mi ćemo ove datoteke obrisati i napraviti novi servis od početka.

Kao i inače s dodavanjem nove komponente, desno ćemo kliknuti na projekt, zatim Add i New item.

Pod WEB kategorijom odabrat ćemo WCF Service kao na slici 62.



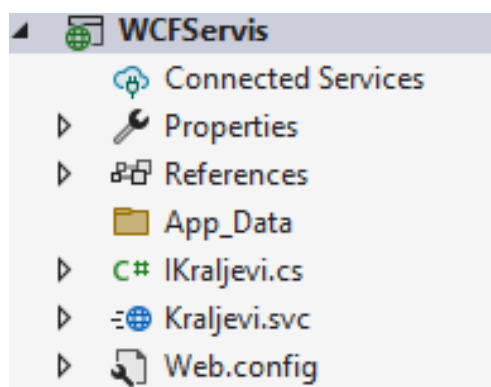
Slika 62. Odabir WCS service datoteke

Naravno, prije klika na Add dodijelit ćemo ime (ja ću dodijeliti ime Kraljevi).

Nakon klika na Add Visual Studio generirat će nam dvije datoteke:

- sučelje u kojem definiramo ugovor i metapodatke metoda servisa na koji će se klijent spajati
- datoteku koja se u biti sastoji od dvije datoteke, deklaracije web-servisa i implementacije sučelja deklariranog u prethodnoj datoteci.

Svaki WCF servis sastoji se od sučelja u kojem definiramo metode i implementacije koja se nalazi u svc datoteci. Projekt sad izgleda kao na slici 63.



Slika 63. Izgled WCF projekta nakon dodavanja nove datoteke

Za rad s podacima možemo preko NuGeta instalirati Entity Framework ako želimo raditi s bazom podataka, no radi jednostavnosti sad ćemo napraviti mali servis koji će samo dati popis kraljeva.

Prvo što trebamo napraviti jest definirati klasu Kralj, koja će izgledati ovako:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Runtime.Serialization;

namespace WCFServis
{
    [DataContract]
    public class Kralj
    {
        [DataMember]
        public string Ime { get; set; }
        [DataMember]
        public string Prezime { get; set; }
        [DataMember]
        public string Kraljevina { get; set; }
        [DataMember]
        public DateTime PocetakVladavne { get; set; }
        [DataMember]
        public DateTime KrajVladavine { get; set; }
    }
}
```

Atributi [DataMember] potrebni su za ispravan rad WCF servisa i nalaze se unutar imenskog prostora System.Runtime.Serialization. S DataMember atributom označavamo svojstva koja želimo da WCF servis izloži prema klijentskim aplikacijama. Svojstva koja nemaju DataMember atribut neće biti dostupna unutar klijentskih aplikacija koje se spajaju na servis. Također, ako želimo omogućiti da kompletna klasa bude izložena putem web-servisa, potrebno je dodati atribut DataContract.

Da bismo napravili WCF servis za izlaganje podataka, prvo moramo definirati ispravno sučelje. Zato ćemo sučelje definirati na sljedeći način:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
```

```
namespace WCFServis
{
    [ServiceContract]
    public interface IKraljevi
    {
        [OperationContract]
        List<Kralj> PopisKraljeva();
    }
}
```

I ovdje se moramo služiti atributima kako bi operacije WCF servisa bile dostupne vanjskim klijentima. ServiceContract atributom navodimo sučelje koje izlažemo putem WCF servisa. OperationContract atributom označavamo metode koje želimo izložiti putem WCF servisa.

Nakon definiranja ispravnog sučelja potrebno ga je implementirati. Taj dio posla radimo unutar svc datoteke (koja nam je praktično code behind). Editirat ćemo je na sljedeći način:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
```

```

namespace WCFServis
{
    public class Kraljevi : IKraljevi
    {
        public List<Kralj> PopisKraljeva()
        {
            List<Kralj> list = new List<Kralj>();
            list.Add(new Kralj
            {
                Ime = "Tomislav",
                PocetakVladavne = new DateTime(925, 1, 1),
                KrajVladavine = new DateTime(928,12,30)
            });
            list.Add(new Kralj
            {
                Ime = "Zvonimir",
                Prezime = "Dimitar",
                PocetakVladavne = new DateTime(1075, 1, 1),
                KrajVladavine = new DateTime(1089, 12, 30)
            });

            return list;
        }
    }
}

```

Ako je sve ispravno napravljeno, *buildamo* projekt. Desnim klikom na svc datoteku možemo odabrati opciju „View in Browser”, koja će nam prikazati HTML masku našeg web-servisa kao na slici 64.

Kraljevi Service

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using the svcutil.exe tool from the command line with the following syntax:

```
svcutil.exe http://localhost:51417/Kraljevi.svc?wsdl
```

You can also access the service description as a single file:

```
http://localhost:51417/Kraljevi.svc?singleWsdl
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your client application and use the generated client class to call the Service. For example:

C#

```
class Test
{
    static void Main()
    {
        KraljeviClient client = new KraljeviClient();

        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

Visual Basic

```
Class Test
    Shared Sub Main()
        Dim client As KraljeviClient = New KraljeviClient()
        ' Use the 'client' variable to call operations on the service.

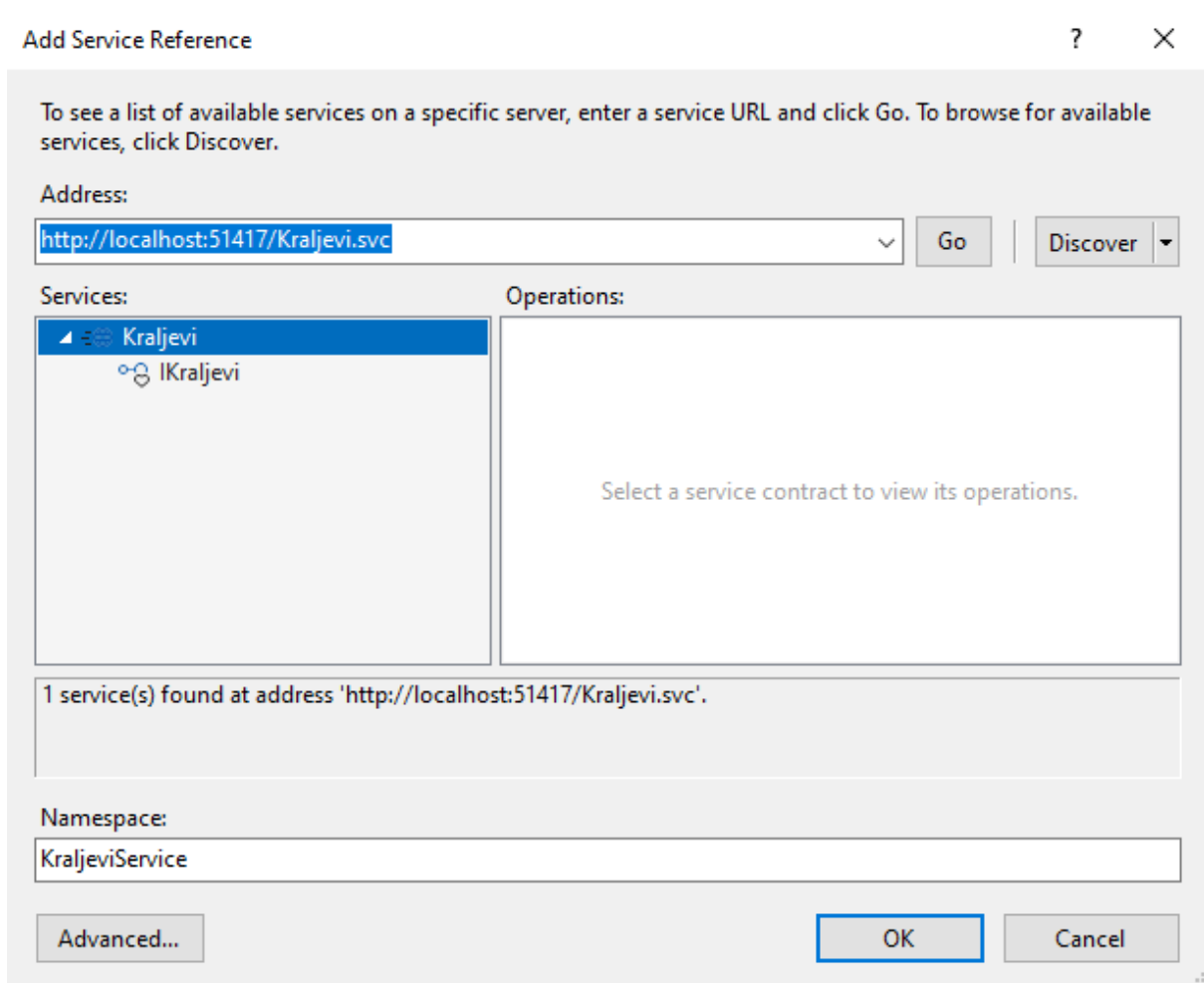
        ' Always close the client.
        client.Close()
    End Sub
End Class
```

Slika 64. Prikaz HTML maske servisa

Ako pogledamo u pregledniku adresu, vidjet ćemo da nam je Visual Studio napravio virtualni direktorij i stranicu kojoj je dinamički dodijelio TCP port. Preko te adrese servis je sada dostupan. Ako dodamo na kraju url-a (nadodamo nakon svc) „?singleWsdl”, dobit ćemo WSDL datoteku koju odašilje ovaj servis.

Budući da je web-servis dostupan preko HTTP protokola i izrađen u skladu sa SOAP standardima, funkcije su sad dostupne iz svih aplikacija i tehnologija koje podržavaju HTTP i SOAP standard. Mi ćemo sad, opet radi jednostavnosti, pokazati na konzolnoj aplikaciji kako konzumirati servis.

Konzumacija WCF servisa radi se dodavanjem servisne reference u projekt. To činimo tako da desno kliknemo na References i odaberemo opciju Add Service Reference.



Slika 65. Dodavanje servisne reference

Unutar okvira Add Service Reference možemo se spojiti na bilo koji *web*-servis ako znamo njegovu točnu adresu. Dodajmo Namespace ime kako bismo osigurali jedinstveni naziv za adresiranje klasa i metoda servisa unutar projekta. Klikom na OK Visual Studio unutar našeg projekta generirat će sve potrebne klase za rad s referencama *web*-servisa.

Kod s klijentske strane izgleda ovako:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

```
using KraljeviHrvatske.KraljeviService; //imenski prostor servisa
```

```
namespace KraljeviHrvatske
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //instanca klase servisa
            KraljeviClient klijent = new KraljeviClient();
            //poziv metode eksponirane u sučelju servisa
            List<Kralj> kraljevi = klijent.PopisKraljeva().ToList();
            foreach (Kralj k in kraljevi)
            {
                Console.WriteLine(k.Ime);
            }
            Console.ReadLine();
        }
    }
}

// Ispis:
/*
Tomislav
Zvonimir
*/
```

5.5.2. WEB API

WEB API još je jedan način izrade *web*-servisa, točnije RESTful servisa, koji je opisan u 7. poglavlju (WEB API) ovog priručnika.

5.6. Validacija

Validacija podataka osiguravanje je točnosti, preciznosti i konzistentnosti podataka. Validacija je često ovisna o platformi na kojoj se radi. Kod MVC aplikacija podaci su

opisani modelom koji se binda u metodama kontrolera, pa se validacija odnosi na validaciju modela.

Validacija podataka može obuhvaćati jednostavnu provjeru podataka kao što je ispravnost formata unesenog podatka, pa do složenih poslovnih pravila definiranih kroz više podataka koji su međusobno u nekoj relaciji. Kao jednostavnije primjere validacije možemo spomenuti tekst koji ne smije imati više od n znakova, datum koji ne smije biti u prošlosti, ime koje ne smije biti prazan string, OIB koji mora imati točno 11 brojevanih znamenki i slično.

5.6.1. Validacija na strani poslužitelja

Validacija podataka na strani poslužitelja odnosi se na provjeru točnosti, preciznosti i konzistentnosti podataka na strani poslužitelja.

Na aplikacijama ćemo sa strane poslužitelja najčešće raditi validaciju podataka unutar samog modela na temelju bindanja prema bazi (povezivanja s bazom podataka). Na temelju tog bindanja kodirat ćemo validacijska pravila. Uz pomoć atributa svojstva modela mogu definirati i validacije, ali i poruke ako dođe do kršenja pravila validacije podataka. Atributi validacije pomoći će nam i ako radimo *code first* pristup rada s Entity Frameworkom, jer na temelju atributa validacije mogu se generirati ograničenja na razini baze podataka. Atributi koje koristimo na modelu jesu:

- **Compare** – usporedba svojstva s navedenim drugim svojstvom
- **Range** – provjera je li vrijednost svojstva unutar nekog opsega
- **RegularExpression** – provjera zadovoljava li vrijednost svojstva regularni izraz
- **Required** – vrijednost svojstva obavezno je navesti
- **StringLength** – vrijednost stringa ne može imati više znakova od postavljene vrijednosti.

No ako se sjetimo Database First pristupa u Entity Frameworku, većinu ovih svojstava već ćemo imati definirano preko baze podataka. Zaključak koji možemo donijeti jest da spremanje podataka, odnosno rad s bazom podataka, također može biti svojevrs-

na validacija. Tako gledano, ako su podaci validni, bit će spremljeni, ako nisu, neće.

Kršenje validnosti podataka svodi se na kršenje referentnog integriteta baze podataka, što nam baca iznimku koju moramo obraditi.

Napraviti generalno recept izrade validacije skoro je nemoguće, jer uvelike ovisi o platformi na kojoj radimo i tipu aplikacije koji radimo.

5.6.2. Klijentska validacija

Validacija podataka na klijentskoj strani validacija je na klijentovu uređaju. Uvelike će ovisiti o vrsti klijentske aplikacije i tehnologije izrade same aplikacije. Ukoliko se radi o *web*-stranici, klijentska validacije odvija se s tehnologijama raspoloživima sa strane klijenta JavaScript i jQuery, a ukoliko se radi o nekoj mobilnoj aplikaciji, isto tako će ovisiti o tome koja je itd.

Govoriti o kompleksnosti implementacije klijentske validacije jako je širok pojam. Budući da ovisi o tehnologiji i platformi na kojoj se radi, teško je povući neku paralelu. Recimo, ukoliko koristimo WPF aplikaciju, unutar same kontrole koja je korisniku na raspolaganju unijeli bismo da mora odgovarati nekom regularnom izrazu ili bismo ograničili što korisnik uopće smije upisivati. Kućice za datum idealan su primjer, jer korisnika ograničavaju na to da može unijeti samo datum, a ako pokušava upisati nešto drugo, izbacit će grešku ili odbiti unos podataka koji su dani.

5.7. Sigurnost korisničkih podataka

Osiguravanje sigurnosti aplikacija složen je i zahtjevan posao za bilo kojeg programera. Samo postavljanje sigurnosnih postavki ovisi o vrsti aplikacije i zahtijeva pažljivo planiranje, a zahtijeva i dobro poznavanje mogućnosti i načina na koji se aplikacije mogu osigurati. Sigurnost podataka bitan je aspekt u svim aplikacijama, a još je više izražen kod aplikacija koje pristupaju internetu.

5.7.1. Autentikacija ili autentifikacija

Autentikacija je proces dokaza identiteta. U realnom životu svoj identitet dokazujemo osobnom iskaznicom ili nekim drugim dokumentom, a u slučaju aplikacija najčešće svoj identitet dokazujemo upisivanjem korisničkog imena i lozinke (bilo direktno ili putem nekog drugog servisa) ili korištenjem nekog kriptografskog certifikata.

U aplikacijama imamo nekoliko vrsta autentikacije.

- autentikacijska forma
- autentikacija preko vanjskog servisa (OpenID/OAuth)
- autentikacija preko operativnog sustava.

Autentikacijska forma obrazac je koji služi za upis korisničkih podataka. Nakon unosa podataka aplikacija samostalno i interno provjerava ispravnost unesenih podataka. Neki sustavi već unaprijed imaju ugrađene mehanizme za ove kompleksne provjere kako bi se olakšalo programerima rad da se mogu orijentirati na rad programa.

Autentikacija preko vanjskog servisa često se koristi u aplikacijama koje imaju pristup internetu (naravno, na kojima je neko registriran preko njihove autentikacijske forme). U ovom slučaju aplikacija ne mora validirati unesene korisničke oznake niti se brinuti o osjetljivosti podataka. Umjesto toga oslanja se na autentikacijske servise trećih strana za provjeru identiteta poput Googlea, Microsofta, Facebooka i drugih. Korištenjem ove vrste autentikacije podržava se i SSo proces (*single sign-in*), koji omogućuje da se korisnik prijavi u svoj korisnički račun na jednom servisu, a samim se time automatski prijavljuje i na aplikacije kojima je dozvoljen pristup. Korisnik u tom slučaju ne mora ponovno unositi svoje korisničke podatke, što povećava lakoću korištenja raznih aplikacija.

Autentikacija preko operativnog sustava namijenjena je najviše intranet aplikacijama i aplikacijama instaliranim na pojedinim uređajima. U slučaju Microsofta ovu vrstu autentikacije često zovemo i IntegratedAuth zbog mogućnosti single sign-ona unutar Windows okruženja. Drugi je primjer Google na Android uređajima. Ovaj način autentikacije nije prigodan za korištenje preko interneta jer bi to značilo da svaki korisnik

mora imati isti operativni sustav, što je nemoguće jer imamo razliku između operativnog sustava računala i mobilnih uređaja.

5.7.2. Autorizacija

Autorizacija korisnika proces je koji određuje što autenticirani korisnik smije raditi unutar aplikacije. Ako uzmemo primjer cloud servisa danas, treba osigurati da samo vlasnik datoteka bude u mogućnosti uređivati i pohranjivati datoteke, a onaj kome da pravo pristupa da samo može pregledati, a ne mijenjati.

Iako ne postoji neka prepreka da svaki programer razvije vlastiti mehanizam autorizacije korisnika aplikacije, u praksi se to pokazalo teškim i mukotrpnim poslom koji rezultira s jako puno redundantnog koda (višestruka dupliciranja koda, koji je onda teško pratiti i u potpunosti zamijeniti ukoliko dođe do promjena, a doći će). Svaki aplikativni sustav ima svoje implementirane sustave pristupa pojedinim dijelovima. Govorimo o specifičnim mehanizmima, klasama i metodama kako bi uvelike olakšali taj posao.

5.8. Isporuka aplikacija

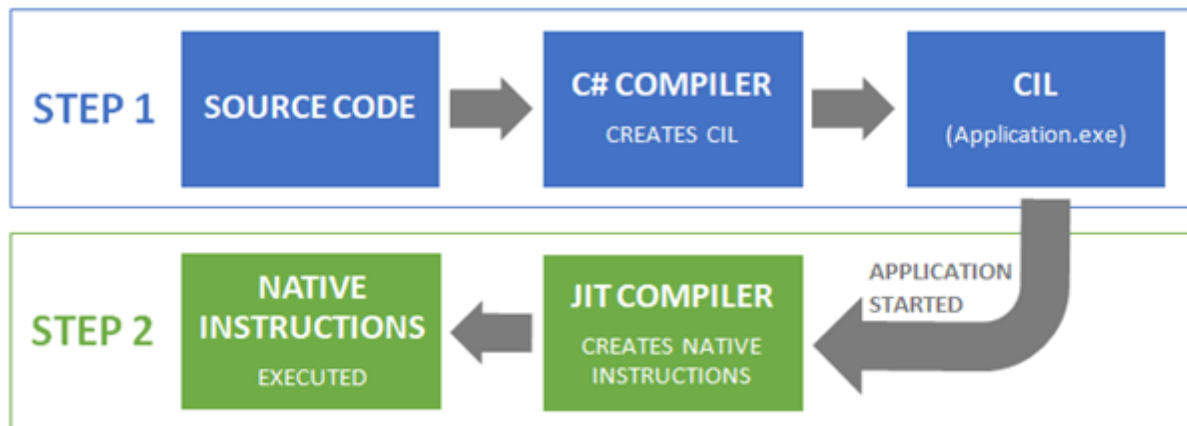
Isporuka aplikacije komplicirana je tema, koja uvelike ovisi o sustavu za koji se aplikacija radi i o vrsti aplikacije koja se radi. Svaka vrsta aplikacije ima svoje specifičnosti isporuke, platformu isporuke i ciljanu publiku kojoj je namijenjena.

5.8.1. Debug i release koncepti

Tijekom izgradnje aplikacije, odnosno projekta, Visual Studio imat će dva moda izgradnje:

- debug
- release.

C# kod prolazi kroz dva kompilacijska koraka da bi postao CPU instrukcije koje se mogu izvršavati.



Slika 66. Kompilacijski koraci C#-a

U koraku 1 aplikacija se izgrađuje pomoću C# kompajlera. Kod koji napišemo pretvara se u CIL (*Common Intermediate Language*), koji se može izvršavati u bilo kojoj okolini koja podržava CIL. CIL nije čitljiv tekst, nego metapodaci i bajt kod u obliku binarnih podataka.

U koraku 2, Just-in-time (JIT) kompajler pretvorit će CIL u instrukcije koje CPU može izvršavati. Ovaj dio neće se unaprijed raditi jer nema smisla kompajlirati u instrukcije procesora ako nismo sigurni na kojem će se računalu (CPU-u) program izvršavati.

JIT kompajler samo je jedan od gomile servisa koji tvore CLR (*Common Language Runtime*), što omogućuje izvođenje .NET koda. Većina optimizacije koda izvodit će se u drugom koraku.

Optimizacija je proces poboljšavanja faktora izvođenja poput brzine izvođenja, veličine koda, korištenja energije i u slučaju .NET-a, vremena koje je potrebno JIT-u da kompajlira kod.

To nas dovodi do ključne razlike između Debug i Release moda izgradnje, odnosno – razlika je u tome je li optimizacija omogućena ili nije.

JIT kompajler izvodi veliku dozu optimizacije, iako mu je glavni zadatak brzo izvođenje njegova posla tijekom izvršavanja programa. Nema mnogo informacija o internom radu JIT-a, to je strogo čuvana tajna od strane Microsofta i dosta varira u nativnom kodu koji proizvede, ovisno o mnogo faktora. Čak i dok se aplikacija izvodi, JIT profilira i ponovno kompajlira kod da poboljša njegovo izvođenje.

No vratimo se sad na modove izgradnje. Jednostavno rečeno, Release je potpuno optimizirani kod, a Debug nije uopće optimiziran. Ovo je bazno u slučaju debugiranja koda. No ovo je samo površni pogled.

Kod koji nije optimiziran lagano je pratiti tijekom izvođenja liniju po liniju. Pomoću ugrađenog Debuggera možemo pratiti izvođenje koda liniju po liniju i dodavati mogućnosti zaustavljanja izvođenja na pojedinim linijama, pratiti stanja varijabli i njihova sadržaja itd. To ga čini idealnim za traženje grešaka, praćenje koda u rekreiranju grešaka da bismo ih mogli otkloniti. Tijekom debugiranja debug mod nam omogućuje da, ako dođe do neuhvaćene iznimke, da nas direktno pozicionira na dio koda gdje je došlo do greške. Iz debug moda rezultat kompajliranja možemo direktno vratiti, odnosno možemo rekreirati izvorni kod.

Suprotno tome, Release mod optimizirani je kod. Release kod više ne možemo pratiti liniju po liniju, nemamo nikakve dodatne informacije o izvođenju koda. Nakon release kompajliranja koda ne možemo rezultat nazad pretvoriti u izvorni kod. Pri distribuciji aplikacije uvijek ćemo aplikaciju kompajlirati u release modu.

5.8.2. Distribucija razvijenog programskog rješenja

Distribucija ili objava aplikacije proces je u kojem aplikaciju činimo dostupnom korisniku. Inače u velikim timovima distribucijom upravljaju DevOps specijalisti.

Distribucija ovisi o mnogo faktora:

- potrebnoj infrastrukturi
- vrsti aplikacije

- platformi za koju je namijenjena
- korisnicima kojima je namijenjena.
-

Problem je to što, ovisno o vrsti programskog rješenja, imamo mnogo kombinacija gornjih faktora, a i pojedini faktori ovise jedan o drugom.

Prolaziti točne detalje određene distribucije nema smisla, pa ćemo napraviti mali pregled nekoliko najčešćih vrsta distribucija aplikacija:

- *web* i *cloud* aplikacije
- mobilne aplikacije
- standalone aplikacije.

Pod *web* i *cloud* aplikacijama mislimo na aplikacije ili dijelove aplikacije koji se izvode na nekom poslužitelju. Ovisno o vrsti tehnologije i vrsti aplikacije poslužitelj ćemo morati konfigurirati prema specifičnim potrebama aplikacija koje ćemo na njima imati ili odabrati odgovarajući servis koji će nam usluživati aplikaciju.

Mobilne aplikacije danas su jedne od najlakših aplikacija za distribuciju. Najčešće se distribuiraju preko takozvanih trgovina aplikacija (Google play, Microsoft store, Apple Store...). Aplikaciju moramo samo objaviti na željenoj trgovini i imamo sve alate za praćenje – kako instalacija aplikacije, tako i demografije korisnika i još mnogo korisnih podataka. Ovisno o operativnom sustavu mobilne aplikacije imamo mogućnost dijeliti aplikacije u obliku paketa mimo trgovina, no korisnicima se ne savjetuje da to koriste, pogotovo iz nepoznatih izvora, pa se vraćamo na to da je najsigurnije koristiti neke već poznate servise.

Standalone aplikacije jesu aplikacije koje se izvode na lokalnim računalima i ne trebaju ništa drugo da bi se izvodile (poput veze na internet). Ovakve aplikacije često ćemo instalirati na računala, nakon što do njih dođemo putem interneta (neki servis ili direktan *download* s *web-stranice*) ili iz nekih drugih izvora (CD, USB stick itd.). U ovoj domeni imamo možda čak i najveći spektar različitih aplikacija – od poslovnih aplikacija, aplikacija za zabavu itd. *Standalone* aplikacije često se rade i za korištenje neke specifične sklopovske komponente, a mogu biti podrška i radu drugih aplikacija. Tako

se primjerice driveri (upravljači programi) mogu smatrati *standalone* aplikacijama.

Iako ovako napravljena podjela nije baš uvijek istinita, postoji spektar gdje se pojedine aplikacije mogu međusobno nadopunjavati. Recimo, standalone aplikacija može biti i mobilna aplikacija (kalkulator sigurno ne treba internet za svoju funkcionalnost itd.). Svakako, neki internetski servis može nadopuniti bilo koju vrstu aplikacije i proširiti funkcionalnost.

6. POGLAVLJE:

Predlošci izrade programskih rješenja

U OVOM POGLAVLJU NAUČIT ĆETE:

- što su to predlošci dizajna
- pregled najčešće korištenih predložaka
- predloške kreiranja objekta
- predloške strukture
- predloške ponašanja
- predloške dizajna programskih rješenja.

Predlošci dizajna (*design patterns*) predstavljaju najčešće korištena praktična rješenja u izradi programa s jezikom baziranim na objektno orijentiranoj paradigmi. Oni su rješenja generalnih problema koji muče programera tijekom procesa razvoja programa. Ova rješenja nastala su procesom pokušaja i pogrešaka brojnih programskih inženjera kroz dugi niz godina.

Sam koncept predložaka dizajna pojavio se 1994. izdavanjem knjige „Design Patterns – Elements of reusable Object-Oriented Software” od strane četvorice autora kasnije prozvanih Gang of Four (GOF). Prema autorima uzorci dizajna primarno se baziraju na dva principa objektno orijentiranog dizajna: „Programiraj do sučelja, ne do implementacije” i „Vrednuj kompoziciju objekta prije nasljeđivanja”.

Uzorci dizajna imaju dvije glavne upotrebe u razvoju programa, a to su zajednička platforma za razvojne inženjere i najbolje prakse.

Uzorci dizajna pružaju nam standardnu terminologiju i specifični su prema određenom scenariju korištenja. Npr. singleton uzorak dizajna znači korist jednog objekta, pa svi koji su upoznati s ovim uzorkom dizajna mogu koristiti jedan objekt s njegovom jedinstvenom instancom i napraviti program koji koristi takav uzorak.

Tijekom vremena predlošci dizajna evoluirali su i sada za određene primjere korištenja imamo generalna rješenja kako napraviti program. Učenje ovih predložaka pomaže neiskusnim programerima da nauče kako dizajnirati program na lakši i brži način.

U originalnoj knjizi imamo tri predložka dizajna:

- predložak kreiranja
- predložak strukture
- predložak ponašanja.

Kasnije je dodano još predložaka dizajna programskih rješenja, koja su uglavnom ovisna o jezicima s kojima se radi, odnosno sami jezici već imaju predloške koji u samom startu pripremaju predložak za implementaciju (npr. ASP.NET MVC).

6.1. Predlošci kreiranja objekta

Predlošci kreiranja objekta pokazuju način kako stvoriti objekt dok istovremeno skrivamo logiku stvaranja umjesto da instanciramo objekt korištenjem `new` operatora. Napraviti ćemo pregled najčešće korištenih predložaka kreiranja objekta.

6.1.1. Abstract Factory

Abstract factory ili **tvornica sažetaka** predložak je koji rješava problem stvaranja produktne linije, odnosno produktne obitelji, bez specificiranja njihove konkretne klase.

U ovom uzorku dizajna definiramo sučelje za stvaranje svih pojedinačnih proizvoda, ali ostavljamo ključne dijelove stvaranja konkretnoj klasi. Svaka tvornica odgovara tom specifičnom proizvodu.

Klijentski kod poziva stvaralačku metodu tvornice objekata umjesto stvaranja proizvoda direktno pozivom na konstruktor. Budući da tvornica odgovara jednoj varijanti proizvoda, svi njeni proizvodi bit će kompatibilni. Klijentski kod radi s tvornicama i proizvodima preko apstraktnih sučelja. Ovo omogućuje klijentskom kodu da radi bez proizvodnih varijanti koje su stvorene preko tvorničkog objekta. Samo stvaramo novu konkretnu tvornicu i proslijedimo je klijentskom kodu.

Ovaj predložak relativno je čest u C#-u. Mnoge biblioteke koriste ga da pruže način proširenja i prilagodbe svojih standardnih komponenti. Predložak je lako prepoznatljiv preko metoda koje vraćaju factory objekt, koji se zatim koristi za stvaranje specifičnih komponenti.

Idemo pokazati na primjeru:

```
using System;
```

```
namespace AbstractFactory
```

```
{
```

```
    // Abstract Factory sučelje deklarira set metoda koje vraćaju  
    // različite apstraktne proizvode. Ovi proizvodi zovu se obitelj i oni su  
    // povezani preko neke teme ili koncepta. Proizvodi iste obitelji su  
    // obično u mogućnosti međusobno surađivati. Obitelj proizvoda može  
    // imati nekoliko varijanti, ali proizvodi jedne varijante nekompatibilni su  
    // s proizvodima druge varijante
```

```
    public interface IAbstractFactory
```

```
    {
```

```
        IAbstraktniProizvodA StvoriProizvodA();
```

```
        IAbstraktniProizvodB StvoriProizvodB();
```

```
    }
```

```
    // Konkretna tvornica proizvodi obitelj proizvoda koji pripadaju jednoj varijanti  
    // Tvornica garantira da su rezultatni proizvodi kompatibilni.
```

// Primijetimo da potpis konkretne tvornice vraća apstraktan proizvod
// dok se unutar metode proizvod stvara instanciranjem.

```
class KonkretnaTvornica1 : IAbstractFactory
{
    public IAbstraktniProizvodA StvoriProizvodA()
    {
        return new KonkretanProizvodA1();
    }

    public IAbstraktniProizvodB StvoriProizvodB()
    {
        return new KonkretanProizvodB1();
    }
}
```

// Svaka konkretna tvornica ima odgovarajuću proizvodnu varijantu.

```
class KonkretnaTvornica2 : IAbstractFactory
{
    public IAbstraktniProizvodA StvoriProizvodA()
    {
        return new KonkretanProizvodA2();
    }

    public IAbstraktniProizvodB StvoriProizvodB()
    {
        return new KonkretanProizvodB2();
    }
}
```

// Svaki pojedinačni proizvod proizvodne obitelji treba imati bazno sučelje.
// Sve varijante proizvoda moraju implementirati to sučelje.

```
public interface IAbstraktniProizvodA
{
    string KorisnaFunkcijaA();
}
```

// Konkretni proizvodi stvaraju se odgovarajućom konkretnom tvornicom.

```
class KonkretanProizvodA1 : IAbstraktniProizvodA
```

```

{
    public string KorisnaFunkcijaA()
    {
        return "Rezultat proizvoda A1.";
    }
}

class KonkretanProizvodA2 : IAbstraktniProizvodA
{
    public string KorisnaFunkcijaA()
    {
        return "Rezultat proizvoda A2.";
    }
}

// Ovdje je bazno sučelje drugog proizvoda. Svi proizvodi mogu
// interagirati međusobno, ali prava interakcija moguća je samo između
// proizvoda iste konkretne varijante.
public interface IAbstraktniProizvodB
{
    // Proizvod B može raditi nešto svoje...
    string KorisnaFunkcijaB();

    // ...ali može kolaborirati s Proizvod A.
    //
    // Apstraktna tvornica osigurava da svi proizvodi koje stvori budu
    // iste varijante i time kompatibilni.
    string JosJednaKorisnaFunkcijaB(IAbstraktniProizvodA collaborator);
}

// Konkretni proizvodi stvoreni su pomoću odgovarajućih konkretnih Tvornica.
class KonkretanProizvodB1 : IAbstraktniProizvodB
{
    public string KorisnaFunkcijaB()
    {
        return "Rezultat proizvoda B1.";
    }
}

```

```

// Varijanta proizvod B1 može raditi ispravno s varijantom proizvod A1
// Iako prima samo instancu proizvoda AbstraktniProizvodA kao argument.
public string JosJednaKorisnaFunkcijaB(IAbstraktniProizvodA collaborator)
{
    var result = collaborator.KorisnaFunkcijaA();

    return $"Rezultat B1 u kolaboraciji s ({result})";
}

class KonkretanProizvodB2 : IAbstraktniProizvodB
{
    public string KorisnaFunkcijaB()
    {
        return "Rezultat proizvoda B2.";
    }

    // Varijanta proizvod B2 može raditi ispravno s varijantom proizvod A1
    // Iako prima samo instancu proizvoda AbstraktniProizvodA kao argument.
    public string JosJednaKorisnaFunkcijaB(IAbstraktniProizvodA collaborator)
    {
        var result = collaborator.KorisnaFunkcijaA();

        return $"Rezultat B2 u kolaboraciji s ({result})";
    }
}

// Klijentski kod radi s tvornicama i proizvodima preko apstraktnih
// tipova: AbstractFactory and AbstraktniProizvod. Ovo omogućuje da
// prosljedimo bilo koju tvornicu ili proizvodnu podklasu klijentu bez da
// dođe do prekida koda.
class Client
{
    public void Main()
    {
        // Klijentski kod funkcionira s bilo kojom konkretnom klasom tvornice.
        Console.WriteLine("Klijent: Testiranje klijentskog koda s prvim tvorničkim
tipom...");
        ClientMethod(new KonkretnaTvornica1());
    }
}

```

```

        Console.WriteLine();

        Console.WriteLine("Klijent: Testiranje klijentskog koda s drugim tvorničkim
tipom...");
        ClientMethod(new KonkretnaTvornica2());
    }

    public void ClientMethod(IAbstractFactory factory)
    {
        var productA = factory.StvoriProizvodA();
        var productB = factory.StvoriProizvodB();

        Console.WriteLine(productB.KorisnaFunkcijaB());
        Console.WriteLine(productB.JosJednaKorisnaFunkcijaB(productA));
    }
}

class Program
{
    static void Main(string[] args)
    {
        new Client().Main();
    }
}

// Izgled izlaza:
// Klijent: Testiranje klijentskog koda s prvim tvorničkim tipom...
// Rezultat proizvoda B1.;
// Rezultat B1 u kolaboraciji s (Rezultat proizvoda A1.)
//
// Klijent: Testiranje klijentskog koda s drugim tvorničkim tipom...
// Rezultat proizvoda B2.;
// Rezultat B1 u kolaboraciji s (Rezultat proizvoda A2.)

```

6.1.2. Builder

Builder ili **graditelj** predložak omogućuje postepenu izgradnju kompleksnih objekata korak po korak. Jako je poznat uzorak u svijetu C#-a, pogotovo koristan kad trebamo stvoriti objekt s više mogućih opcija konfiguriranja.

Ovaj predložak može se prepoznati u klasi koja ima jednu metodu stvaranja i mnogo metoda konfiguriranja rezultirajućeg objekta. Metode ovog predloška često podržavaju ulančavanje.

U sljedećem primjeru prikazat ćemo strukturu graditelj predloška, koja treba odgovoriti na pitanja što klasa sadrži, koje uloge klasa ima i na koji su način elementi unutar predloška u odnosu:

```
using System;
using System.Collections.Generic;

namespace Builder
{
    // Builder sučelje specificira metode za stvaranje različitih
    // dijelova objekata Produkt
    public interface IBuilder
    {
        void BuildDioA();

        void BuildDioB();

        void BuildDioC();
    }
    // KonkretanBuilder klasa prati Builder sučelje i pruža
    // specifičnu implementaciju koraka izgradnje. Program može imati
    // višestruke varijante Buildera s različitim implementacijama.
    public class KonkretanBuilder : IBuilder
    {
        private Proizvod _Proizvod = new Proizvod();

        // Nova Builder instanca treba sadržavati prazan Proizvod objekt,
        // koji se koristi u budućim sastavljanjima.
    }
}
```

```

public KonkretanBuilder()
{
    this.Ponisti();
}

public void Ponisti()
{
    this._Proizvod = new Proizvod();
}

// Svi Proizvodni koraci rade s istom Proizvod instancom.
public void BuildDioA()
{
    this._Proizvod.Add("DioA1");
}

public void BuildDioB()
{
    this._Proizvod.Add("DioB1");
}

public void BuildDioC()
{
    this._Proizvod.Add("DioC1");
}

// Konkretni Builderi trebaju pružiti svoju metodu za
// dohvat rezultata. To je zato što različiti tipovi Buildera mogu
// stvoriti potpuno druge proizvode, koji nemaju isto sučelje, i time
// takve metode ne mogu biti deklarirane u baznom Builder sučelju
// (bar ne u statičnom programskom jeziku).
//
// Uobičajeno, nakon vraćanja krajnjeg rezultata klijentu od builder
// instance očekuje se da bude spremna proizvoditi sljedeći Proizvod.
// Zbog toga je uobičajena praksa pozivati poništi metodu na kraju
// `GetProizvod` metode. Iako, ovo ponašanje nije obavezno i može
// prisiliti buildere da čekaju eksplicitno poništavanja, pozvano

```

```
// sa strane klijentskog koda prije nego što se riješimo prijašnjeg rezultata.
public Proizvod GetProizvod()
{
    Proizvod result = this._Proizvod;

    this.Ponisti();

    return result;
}
}
```

```
// Ima smisla da koristimo Builder predložak samo kad su Proizvodi
// dosta kompleksni i zahtijevaju dosta podešavanja.
//
// Naspram drugih predložaka stvaranja različiti konkretni builderi mogu
// stvoriti neovisni proizvod. Drugim riječima, rezultat različitih buildera
// ne mora uvijek pratiti isto sučelje.
```

```
public class Proizvod
{
    private List<object> _Dijelovi = new List<object>();

    public void Add(string Dio)
    {
        this._Dijelovi.Add(Dio);
    }

    public string ListDijelovi()
    {
        string str = string.Empty;

        for (int i = 0; i < this._Dijelovi.Count; i++)
        {
            str += this._Dijelovi[i] + ", ";
        }

        str = str.Remove(str.Length - 2); // micanje zadnjeg ", "

        return "Proizvod Dijelovi: " + str + "\n";
    }
}
```



```
}  
}
```

```
// Director je samo zadužen za izvršavanje koraka izgradnje u slijedu  
// Dobro je proizvoditi proizvode u specifičnom slijedu ili konfiguraciji  
// Strogo govoreći, Director klasa opcionalna je  
// jer klijent može kontrolirati builder direktno.
```

```
public class Director
```

```
{
```

```
    private IBuilder _builder;
```

```
    public IBuilder Builder
```

```
    {
```

```
        set { _builder = value; }
```

```
    }
```

```
// The Director can construct several Proizvod variations using the same  
// building steps.
```

```
public void BuildMinimalViableProizvod()
```

```
{
```

```
    this._builder.BuildDioA();
```

```
}
```

```
public void BuildFullFeaturedProizvod()
```

```
{
```

```
    this._builder.BuildDioA();
```

```
    this._builder.BuildDioB();
```

```
    this._builder.BuildDioC();
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        // Klijentski kod stvara build objekt, prosljeđuje ga
```

```
        // direktoru i onda inicira proces konstrukcije.
```

```
        // Krajnji rezultat uzima se iz builder objekta.
```

```
        var director = new Director();
```

```

var builder = new KonkretanBuilder();
director.Builder = builder;

Console.WriteLine("Standardni bazni proizvod:");
director.BuildMinimalViableProizvod();
Console.WriteLine(builder.GetProizvod().ListDijelovi());

Console.WriteLine("Standardni potpuni proizvod:");
director.BuildFullFeaturedProizvod();
Console.WriteLine(builder.GetProizvod().ListDijelovi());

// Zapamtimo, buidler predložak može se koristiti bez direktora
Console.WriteLine("Bez direktora proizvod:");
builder.BuildDioA();
builder.BuildDioC();
Console.WriteLine(builder.GetProizvod().ListDijelovi());
    }
}
}

// Izgled izlaza:
// Standardni bazni proizvod:
// Proizvod Dijelovi: DioA1
//
// Standardni potpuni proizvod:
// Proizvod Dijelovi: DioA1, DioB1, DioC1
//
// Bez direktora proizvod:
// Proizvod Dijelovi: DioA1, DioC1

```

6.1.3. Factory Method

Factory method ili **tvornica metoda** predložak rješava problem stvaranja objekata proizvoda bez specificiranja njihove konkretne klase. Ovaj predložak definira metodu koja bi se trebala koristiti za stvaranje objekata umjesto direktnog poziva konstruktora

korištenjem new operatora. Podklase mogu premostiti ovu metodu tako da promijene klasu objekta koji će biti stvoren. Ovaj predložak vrlo je koristan kad trebamo pružiti visok nivo fleksibilnosti u kodu. Može se prepoznati preko metoda koje stvaraju objekte konkretnih klasa, no vraćaju ih kao objekte apstraktnih tipova ili sučelja.

U sljedećem kodu pokazat ćemo strukturu ovog predloška, koja se fokusira na to od kojih se klasa sastoji, koje uloge te klase imaju i na koji su način elementi u predlošku u relaciji:

```
using System;
```

```
namespace FactoryMethod
```

```
{  
    // Creator klasa deklarira tvorničku metodu koja treba vratiti  
    // objekt Proizvod klase. Creatorova podklasa uobičajeno pruža  
    // implementaciju ove metode  
    abstract class Creator  
    {  
        // Creator može pružiti neku uobičajenu implementaciju tvornice metoda  
        public abstract IProizvod FactoryMethod();  
  
        // Unatoč imenu Creatorova primarna dužnost nije stvarati proizvode.  
        // Uobičajeno sadrži neke osnovne elemente logike upravljanja koji su u  
        // relaciji s objektima proizvoda vraćeni preko tvornice metoda.  
        // Podklase mogu indirektno mijenjati tu logiku rada tako da  
        // premoste tvornicu metoda i vrate različiti tip proizvoda.  
        // Proizvod from it.  
        public string NekaOperacija()  
        {  
            // Call the factory method to create a Proizvod object.  
            var Proizvod = FactoryMethod();  
            // Now, use the Proizvod.  
            var rezultat = "Creator: Isti kod je radio s "  
                + Proizvod.Operation();  
  
            return rezultat;  
        }  
    }  
}
```

```

// KonkretanCreators premošćuje tvornicu metoda da bi promijenio tip
// rezultirajućeg proizvoda.
class KonkretanCreator1 : Creator
{
    // Potpis metode još uvijek koristi apstraktan tip Proizvod
    // iako je KonkretanProizvod u stvarnosti vraćen iz metode.
    // Ovako Creator može ostati nezavisan od klasa konkretnog proizvoda.
    public override IProizvod FactoryMethod()
    {
        return new KonkretanProizvod1();
    }
}

class KonkretanCreator2 : Creator
{
    public override IProizvod FactoryMethod()
    {
        return new KonkretanProizvod2();
    }
}

// Proizvod sučelje deklarira operacije koje svaki konkretan proizvod
// mora implementirati.
public interface IProizvod
{
    string Operation();
}

// KonkretanProizvod pruža razne implementacije Proizvod sučelja.
class KonkretanProizvod1 : IProizvod
{
    public string Operation()
    {
        return "{Rezultat od KonkretanProizvod1}";
    }
}

class KonkretanProizvod2 : IProizvod

```

```

{
    public string Operation()
    {
        return "{Rezultat od KonkretanProizvod2}";
    }
}

class Client
{
    public void Main()
    {
        Console.WriteLine("Aplikacija: Pokrenuta s KonkretanCreator1.");
        ClientCode(new KonkretanCreator1());

        Console.WriteLine("");

        Console.WriteLine("Aplikacija: Pokrenuta s KonkretanCreator2.");
        ClientCode(new KonkretanCreator2());
    }

    // Klijentski kod radi s instancom KonkretanCreator, ali preko njegova
    // baznog sučelja. Dokle god klijent nastavlja raditi s kreatorom preko
    // baznog sučelja, možemo mu prosljeđivati creatorovu podklasu.
    public void ClientCode(Creator creator)
    {
        // ...
        Console.WriteLine("Klijent: Nisam svjestan kreatorove klase," +
            "ali sve funkcionira.\n" + creator.NekaOperacija());
        // ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        new Client().Main();
    }
}

```

```
}  
}
```

// Izgled izlaza:

// Aplikacija: Pokrenuta s KonkretanCreator1.

// Klijent: Nisam svjestan kreatorove klase, ali sve funkcionira.

// Creator: Isti kod radio je s {Rezultat od KonkretanProizvod1}

//

// Aplikacija: Pokrenuta s KonkretanCreator2.

// Klijent: Nisam svjestan kreatorove klase, ali sve funkcionira.

// Creator: Isti kod radio je s {Rezultat od KonkretanProizvod2}

6.1.4. Object pool

Object pool ili **bazen objekata** predložak je i spremnik objekata koji sadrže listu drugih objekata. Jednom kad je objekt izvađen iz bazena, više nije raspoloživ u bazenu dokle god se ne vrati nazad. Na ovakav način možemo pratiti objekte koji se trenutno koriste. Objekti unutar bazena imaju životni ciklus koji se sastoji od stvaranja, validacije i uništavanja.

Kad gradimo .NET aplikaciju, možemo doći do objekata koji su skupi za stvaranje. U većini scenarija skupoća stvaranja novih objekata visoka je i ima utjecaj na performanse aplikacije. Bazen objekata dosta pomaže u ovakvim scenarijima, gdje u bazenu držimo instance objekata koje možemo ponovno koristiti po potrebi. Takvim načinom ponovnog korištenja ubrzavamo program. Kad program zahtijeva objekt na korištenje, ako je objekt raspoloživ u bazenu, vraća se iz bazena. Ako objekt nije raspoloživ u bazenu, stvara se nova instanca i ona se vraća dijelu programa koji ju je zahtijevao.

Da bismo pokazali kako funkcionira bazen objekata, koristit ćemo tvornica predložak. Stvorit ćemo tvornica metodu, koja će biti zadužena za stvaranje objekata. Pri zahtjevu za novim objektom tvornica metoda pogledat će prvo u bazen objekata koji je red objekata. Ako ima bilo koji raspoloživ objekt, vratit će taj objekt, a u suprotnom će novi objekt biti instanciran i on će biti vraćen: using System;

using System.Collections.Generic;

```

using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;

namespace ObjectPool
{
    class Program
    {
        static void Main(string[] args)
        {
            Tvornica tvornica = new Tvornica();
            Ucenik ucenik = tvornica.GetUcenik();
            Console.WriteLine("Prvi objekt");
            Ucenik ucenik1 = tvornica.GetUcenik();
            Console.WriteLine("Drugi objekt");
            Ucenik ucenik2 = tvornica.GetUcenik();
            Console.WriteLine("Treći objekt");
            Console.Read();
        }
    }

    class Tvornica
    {
        // Maksimalan broj objekata!
        private static int _PoolMaxSize = 3;
        // kolekcija koja će služiti za bazen objekata
        private static readonly Queue objPool = new Queue(_PoolMaxSize);
        public Ucenik GetUcenik()
        {
            Ucenik _Ucenik;
            // Provjera, ako postoji, vrati, ako ne, stvori novi
            if (Ucenik.ObjectCounter >= _PoolMaxSize &&
                objPool.Count > 0)
            {
                // Dohvati iz bazena
                _Ucenik = RetrieveFromPool();
            }
            else

```

```

    {
        _Ucenik = GetNewUcenik();
    }
    return _Ucenik;
}
private Ucenik GetNewUcenik()
{
    Ucenik ucenik = new Ucenik();
    objPool.Enqueue(ucenik);
    return ucenik;
}
protected Ucenik RetrieveFromPool()
{
    Ucenik ucenik;
    if (objPool.Count > 0)
    {
        ucenik = (Ucenik)objPool.Dequeue();
        Ucenik.ObjectCounter--;
    }
    else
    {
        // Return a new object
        ucenik = new Ucenik();
    }
    return ucenik;
}
}

```

```

class Ucenik
{
    public static int ObjectCounter = 0;
    public Ucenik()
    {
        ++ObjectCounter;
    }
    private string ime;
    private string prezime;
    private string razred;
}

```



```

public string Ime
{
    get {return ime;}
    set {ime = value;}
}

public string Prezime
{
    get { return prezime; }
    set {prezime = value; }
}

public string Razred
{
    get { return razred;}
    set { razred = value;}
}
}
}

```

Bazen objekata sličan je bazenu konekcija na bazu podataka. Minimalan i maksimalan broj objekata koji mogu biti dodani u bazenu varijabilan je, odnosno možemo ga postaviti. Prilikom dizajna možemo dodati i sljedeće slučajeve u bazen:

- povratak null vrijednosti ili bacanja iznimke ako program treba objekt iz bazena, ali je maksimalan broj objekata alociran
- blokirati poziv dokle god neki objekt ne bude raspoloživ
- povećati veličinu bazena da bismo mogli staviti više objekata.

Programer može modificirati bazen objekata da omogući da se minimalna i maksimalna veličina bazena pročita iz konfiguracijske datoteke. Kao dio inicijalizacije objekta bazena možemo osigurati da bazen sadrži minimalan broj objekata koji će se nalaziti u njemu u samome startu.

Bazeni objekata pomažu nam da smanjimo opterećenje programa u smislu stalnog

opetovanog instanciranja klasa i time povećavaju performanse programa.

6.1.5. Prototype

Prototype ili **prototip** predložak omogućuje kloniranje objekata bez vezanja na njihovu specifičnu klasu. Svi prototipovi klasa trebali bi imati jednako sučelje koje omogućuje kopiranje objekata čak i ako su njihove konkretne klase nepoznate. Prototip objekti mogu stvoriti pune kopije, budući da objekti iste klase mogu pristupiti privatnim poljima jedan drugog. Prototip predložak omogućen je `ICloneable` sučeljem u samom startu u C# programskom jeziku. Njegova implementacija u klasam može se odmah prepoznati po tome sadrži li metode `clone`, `copy` ili neku sličnu.

Primjer korištenja prototip predloška, no bez `ICloneable` klase, nego s vlastito implementiranom klasom:

```
using System;
```

```
namespace Prototype
```

```
{
```

```
    public class Osoba
```

```
    {
```

```
        public int Dob;
```

```
        public DateTime Datum_Rodjenja;
```

```
        public string Ime;
```

```
        public OIBInfo OIBInfo;
```

```
        public Osoba PlitkaKopija()
```

```
        {
```

```
            return (Osoba)this.MemberwiseClone();
```

```
        }
```

```
        public Osoba DubokaKopija()
```

```
        {
```

```
            Osoba clone = (Osoba)this.MemberwiseClone();
```

```

        clone.OIBInfo = new OIBInfo(OIBInfo.OIB);
        clone.Ime = String.Copy(Ime);
        return clone;
    }
}

public class OIBInfo
{
    public int OIB;
    public OIBInfo(int OIB)
    {
        this.OIB = OIB;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Osoba o1 = new Osoba();
        o1.Dob = 42;
        o1.Datum_Rodjenja = Convert.ToDateTime("1977-01-01");
        o1.Ime = "Jack Daniels";
        o1.OIBInfo = new OIBInfo(666);

        // Napravi plitku kopiju o1 i pridruži o2
        Osoba o2 = o1.PlitkaKopija();
        // Napravi duboku kopiju o1 i pridruži o3.
        Osoba o3 = o1.DubokaKopija();

        // Prikaz vrijednosti o1, o2 i o3.
        Console.WriteLine("Originalna vrijednost o1, o2, o3:");
        Console.WriteLine("  o1 vrijednosti: ");
        DisplayValues(o1);
        Console.WriteLine("  o2 vrijednosti:");
        DisplayValues(o2);
        Console.WriteLine("  o3 vrijednosti:");
        DisplayValues(o3);
        // Promijeni vrijednost o1 svojstva i prikaži vrijednosti o1,

```

```

// o2 and o3.
o1.Dob = 32;
o1.Datum_Rodjenja = Convert.ToDateTime("1900-01-01");
o1.Ime = "Frank";
o1.OIBInfo.OIB = 7878;
Console.WriteLine("\nVrijednosti o1, o2 i o3 nakon promjena o1:");
Console.WriteLine(" o1 vrijednosti: ");
DisplayValues(o1);
Console.WriteLine(" o2 vrijednosti (referentne vrijednosti promijenile su
se):");
DisplayValues(o2);
Console.WriteLine(" o3 vrijednosti (sve je ostalo isto):");
DisplayValues(o3);
}

public static void DisplayValues(Osoba p)
{
    Console.WriteLine(" Ime: {0:s}, Dob: {1:d}, Datum_Rodjenja: {2:MM/dd/
yy}",
        p.Ime, p.Dob, p.Datum_Rodjenja);
    Console.WriteLine(" OIB#: {0:d}", p.OIBInfo.OIB);
}
}
}

```

// Ispis programa:

/*

Originalna vrijednost o1, o2, o3:

o1 vrijednosti:

Ime: Jack Daniels, Dob: 42, Datum_Rodjenja: 01 / 01 / 77

ID#: 666

o2 vrijednosti:

Ime: Jack Daniels, Dob: 42, Datum_Rodjenja: 01 / 01 / 77

ID#: 666

o3 vrijednosti:

Ime: Jack Daniels, Dob: 42, Datum_Rodjenja: 01 / 01 / 77

ID#: 666

Originalna vrijednost o1, o2 i o3 nakon promjena o1:

o1 vrijednosti:

Ime: Frank, Dob: 32, Datum_Rodjenja: 01 / 01 / 00

ID#: 7878

o2 vrijednosti (referentne vrijednosti su se promijenile):

Ime: Jack Daniels, Dob: 42, Datum_Rodjenja: 01 / 01 / 77

ID#: 7878

o3 vrijednosti (sve je ostalo isto):

Ime: Jack Daniels, Dob: 42, Datum_Rodjenja: 01 / 01 / 77

ID#: 666

*/

6.1.6. Singleton

Singleton ili na hrvatskom **predložak** jednopojavaost jedan je od najpopularnijih predložaka u kojem klasa može imati samo jednu instancu unutar programa, koja pruža globalnu pristupnu točku programu. Drugim riječima, singleton je klasa koja omogućuje isključivo samo jednu instancu sebe same da bude stvorena i uobičajeno daje jednostavan pristup toj instanci.

Ima više načina na koje se može implementirati singleton predložak u C#-u. No kako god bilo, **singleton ima sljedeća svojstva:**

- konstruktor je privatan i bez parametara
- klasa je „zapečaćena” (sealed ključna riječ)
- statične varijable drže referencu na jednoj stvorenoj instanci
- mora imati javni i statični način dohvata reference na klasu (public static).

Prednosti singleton predloška jesu:

- može implementirati sučelja
- može biti lijeno-učitavanje (lazy-load) i ima statični inicijalizator
- pomaže u skrivanju ovisnosti
- pruža jednostruku točku pristupa određenoj instanci, pa je lagan za održavanje.

Nedostaci singleton predloška:

- unit testovi teški su jer predstavlja globalno stanje aplikacije
- smanjuje potencijal za paralelizaciju programa jer ga zaključava.

Razlike između singleton klase i statičnih klasa jesu:

- statična klasa ne može biti proširena, a singleton klasa može
- statična klasa ne može biti inicijalizirana, a singleton može (iako samo jednom)
- statična klasa učitana je automatski putem CLR-a kad se program koji sadrži klasu učitava.

Implementacije singleton predloška možemo napraviti na više načina:

- bez dretveno sigurnog singletona
- dretveno sigurni singleton
- dretveno sigurni singleton korištenjem dvostruko provjerenog ključanja
- dretveno sigurni bez zaključavanja
- korištenjem .NET-ovog Lazy<T> tipa.

Slijede primjeri implementacije singletona.

Bez dretveno sigurnog singletona specifično je to što se može slučajno dogoditi da različite dretve evaluiraju da singleton nije instanciran, pa ga mogu pokušati istovremeno instancirati. Iako je vjerojatnost da se to dogodi jako mala, no nikad nije ravna nuli:

```
public sealed class Singleton1
{
    private Singleton1() {}
    private static Singleton1 instance = null;
    public static Singleton1 Instance
    {
        get
        {
            if (instance == null)
            {
```

```

        instance = new Singleton1();
    }
    return instance;
}
}
}

```

Kod dretveno sigurnog singletona nema bojazni kao u prvom primjeru. U kodu dretva je zaključana na dijeljenom objektu i provjerava je li instanca stvorena ili ne. Osigurava se memorijska barijera i samo jedna dretva može stvoriti instancu. Također, samo jedna dretva može biti unutar tog koda. Najveći problem ovoga su performanse jer zbog mehanizma ključanja ključnog dijela koda samo jedna dretva može pristupiti samoj provjeri postojanja instance. Slijedi primjer:

public sealed class Singleton2

```

{
    Singleton2() {}
    private static readonly object lock = new object ();
    private static Singleton2 instance = null;
    public static Singleton2 Instance
    {
        get
        {
            lock (lock)
            {
                if (instance == null)
                {
                    instance = new Singleton2();
                }
                return instance;
            }
        }
    }
}

```

Dretveno sigurni singleton korištenjem dvostruko provjerenog ključanja zaključava dretvu na dijeljenom objektu i provjerava je li instanca stvorena ili nije dvostrukom provjerom:

```
public sealed class Singleton3
{
    Singleton3() {}
    private static readonly object lock = new object ();
    private static Singleton3 instance = null;
    public static Singleton3 Instance
    {
        get
        {
            if (instance == null)
            {
                lock (lock)
                {
                    if (instance == null)
                    {
                        instance = new Singleton3();
                    }
                }
            }
            return instance;
        }
    }
}
```

Dretveno sigurni singleton bez zaključavanja naizgled izgleda vrlo jednostavno. Ova implementacija ima statični konstruktor, pa se pokreće samo jednom po aplikacijskoj domeni:

```
public sealed class Singleton4
{
    private static readonly Singleton4 instance = new Singleton4();
    static Singleton4()
    {

```



```

    }
    private Singleton4()
    {
    }
    public static Singleton4 Instance
    {
        get
        {
            return instance;
        }
    }
}

```

Korištenjem .NET-ovog `Lazy<T>` tipa moguće je u verzijama 4 ili višim da bismo napravili jednostavan singleton. Možemo proslijediti delegat konstruktoru da pozove singleton konstruktor, što se jednostavno postiže lambda izrazom. To nam omogućuje da brzo provjerimo je li instanca stvorena `IsValueCreated` svojstvom:

```

public sealed class Singleton5
{
    private Singleton5()
    {
    }

    private static readonly Lazy<Singleton5> lazy = new Lazy<Singleton5>(() =>
new Singleton5());
    public static Singleton5 Instance
    {
        get
        {
            return lazy.Value;
        }
    }
}

```

6.2. Predlošci struktura

Ovi predlošci dizajna odnose se na klase i objektu kompoziciju. Koncept nasljeđivanja koristi se da se komponiraju sučelja i definiraju načini za kombiniranje objekata da dobiju nove funkcionalnosti.

6.2.1. Adapter

Adapter predložak omogućuje objektima s nekompatibilnim sučeljima da surađuju. Adapter se ponaša kao omotač između dva objekta, hvata pozive namijenjene jednom objektu i transformira ih u format i sučelje koje drugi objekt može prepoznati.

Često se koristi u sustavima baziranim na nekom prijašnjem kodu. U ovim slučajevima pomoću adaptera možemo napraviti klasičan kod da radi s modernim klasama.

Adapter se prepoznaje u konstruktoru koji uzima instancu drugog apstraktnog tipa ili sučelja. Kad adapter primi poziv na neku od svojih metoda, prevodi parametre u odgovarajući format i onda usmjerava poziv na jednu ili više metoda klase oko kojih je omotan.

Korištenje adaptera prikazat ćemo na sljedećem primjeru:

```
using System;
```

```
namespace Adapter
```

```
{
```

```
    // Target definira domenski specifično sučelje koje koristi klijentski kod
```

```
    public interface ITarget
```

```
    {
```

```
        string GetRequest();
```

```
    }
```

```
    // Adaptee sadrži korisno ponašanje, ali njegovo je sučelje
```

```

// nekompatibilno s postojećim klijentskim kodom. Adaptee treba
// prilagodbu prije nego što ga klijentski kod može koristiti
class Adaptee
{
    public string GetSpecificRequest()
    {
        return "Specifičan zahtjev.";
    }
}

// Adapter nam omogućuje da Adaptee sučelje bude kompatibilno s Target
// sučeljem
class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        this._adaptee = adaptee;
    }

    public string GetRequest()
    {
        return $"Ovo je '{this._adaptee.GetSpecificRequest()}';"
    }
}

class Program
{
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        ITarget target = new Adapter(adaptee);

        Console.WriteLine("Adaptee sučelje je nekompatibilno s klijentom.");
        Console.WriteLine("No adapter klijent može zvati njegovu metodu.");

        Console.WriteLine(target.GetRequest());
    }
}

```

```

    }
}

// Ispis programa:
/*
    Adaptee sučelje nekompatibilno je s klijentom.
    No adapter klijent može zvati njegovu metodu.
    Ovo je 'Specifičan zahtjev'.
*/

```

6.2.2. Bridge

Bridge ili **most** predložak dijeli poslovnu logiku ili velike klase u odvojene klasne hijerarhije koje mogu biti nezavisno razvijene. Jedna od tih hijerarhija (često zvana apstrakcija) dobit će referencu na objekt druge hijerarhije (implementacija). Apstrakcija će moći delegirati neke svoje pozive na implementaciju objekta. Budući da sve implementacije imaju zajedničko sučelje, mogu se mijenjati unutar apstrakcije.

Ovaj predložak posebno je koristan kad radimo s aplikacijama koje nisu vezane na jedan sustav (*cross-platform*), koje podržavaju višestruke tipove baza podataka ili rade s nekoliko API pružatelja specifičnih vrsta.

Predložak mosta može se prepoznati po čistoj odvojenosti između dijela koji kontrolira i nekoliko različitih platformi na koje se oslanja.

Korištenje predloška mosta pokazat ćemo na sljedećem primjeru:

```

using System;

namespace Bridge
{
    // Abstraction definira sučelje za kontrolni dio dvije hijerarhije klasa
    // Ona zadržava referencu na objekt implementacijske hijerarhije
    // i delegira sav stvarni rad na taj objekt.
    class Abstraction

```

```

{
    protected Implementation _implementation;

    public Abstraction(Implementation implementation)
    {
        this._implementation = implementation;
    }

    public virtual string Operation()
    {
        return "Abstract: Bazna operacija s:\n" +
            _implementation.OperationImplementation();
    }
}

// Možemo proširiti apstrakciju bez promjene implementacijskih klasa
class ExtendedAbstraction : Abstraction
{
    public ExtendedAbstraction(Implementation implementation) : base(imple-
mentation)
    {
    }

    public override string Operation()
    {
        return "ExtendedAbstraction: Proširena operacija s:\n" +
            base._implementation.OperationImplementation();
    }
}

// Implementacija definira sučelje za sve implementacijske klase.
// Ona ne mora odgovarati apstrakcijskom sučelju, u biti, dva sučelja
// mogu biti potpuno različita. Tipično implementacijsko sučelje pruža
// samo primitivne operacije, dok apstrakcija definira operacije višeg
// nivoa bazirane na primitivnim.
public interface Implementation
{
    string OperationImplementation();
}

```

```
// Svaka konkretna implementacija odgovara specifičnoj platformi i
// implementira implementacijsko sučelje koristeći API odgovarajuće
// platforme.
```

```
class ConcretImplementationA : Implementation
```

```
{
    public string OperationImplementation()
    {
        return "ConcretImplementationA: Rezultat u platformi A.\n";
    }
}
```

```
class ConcretImplementationB : Implementation
```

```
{
    public string OperationImplementation()
    {
        return "ConcretImplementationA: Rezultat u platformi B.\n";
    }
}
```

```
class Client
```

```
{
    // Osim za inicijalizacijsku fazu, gdje će objekt apstrakcije biti povezan
    // sa specifičnim implementacijskim objektom, klijentski kod trebao bi
    // ovisiti samo o apstrakcijskoj klasi. Ovako klijentski kod može
    // podržavati bilo koju apstrakcijsko-implementacijsku kombinaciju.
    public void ClientCode(Abstraction abstraction)
    {
        Console.WriteLine(abstraction.Operation());
    }
}
```

```
class Program
```

```
{
    static void Main(string[] args)
    {
        Client client = new Client();

        Abstraction abstraction;
        // Klijentski kod trebao bi biti u mogućnosti raditi s prije podešenom
```

```

// apstrakcijsko-implementacijskom kombinacijom.
abstraction = new Abstraction(new ConcretImplementationA());
client.ClientCode(abstraction);

Console.WriteLine();

abstraction = new ExtendedAbstraction(new ConcretImplementationB());
client.ClientCode(abstraction);
    }
}
}
// Ispis programa:
/*
    Abstract: Bazna operacija s:
    ConcretImplementationA: Rezultat u platformi A.

    ExtendedAbstraction: Proširena operacija s:
    ConcretImplementationA: Rezultat u platformi B.
*/

```

6.2.3. Composite

Composite ili **oblikovanje** kompozita predložak omogućuje slaganje objekata u strukturu nalik stablu i radi s tom kompozicijom kao da je jedan jedinstveni objekt. Ovaj predložak postao je popularno rješenje za većinu problema koji trebaju izgradnju stabla strukture, jer je njegova najveća prednost mogućnost izvođenja metoda rekurzivno kroz cijelo strukturno stablo i sumiranje rezultata.

Često se koristi za predstavljanje hijerarhije komponenti korisničkih sučelja ili koda koji radi s grafovima. Ukoliko imamo stablo objekata i svaki objekt stabla dio je iste klasne hijerarhije, onda se vjerojatno radi o *composite* predlošku. Ako metode ovih klasa delegiraju rad svojim podobjektima stabla i rade preko baznih klasa/sučelja hijerarhije, onda definitivno govorimo o kompozitima.

Slijedi primjer composite predloška:

```

using System;
using System.Collections.Generic;

namespace Composite
{
    // Bazna klasa Component deklarira uobičajene operacije jednostavnih i
    // kompleksnih objekata kompozicije.
    abstract class Component
    {
        public Component() {}

        // Bazni Component može implementirati neko uobičajeno ponašanje ili
        // ostaviti konkretnoj klasi da metodu koja sadržava ponašanje proglasi
        // apstraktnom
        public abstract string Operation();

        // U nekim slučajevima bilo bi korisno definirati upravljanje
        // operacija podčvorova u baznoj klasi komponente. Na ovakav način ne
        // moramo izložiti konkretnu komponentu klase klijentskom kodu, čak
        // ni prilikom slaganja stabla objekata. Loša je stvar što će te metode
        // biti prazne za krajnje listove stabla hijerarhije
        public virtual void Add(Component component)
        {
            throw new NotImplementedException();
        }

        public virtual void Remove(Component component)
        {
            throw new NotImplementedException();
        }

        // Možemo implementirati metodu koja nam omogućuje da klijentski kod
        // odluči može li komponenta imati podčvorove.
        public virtual bool IsComposite()
        {
            return true;
        }
    }
}

```



```

// Leaf (list) klasa predstavlja krajnji objekt kompozicije. List ne može imati
// potkomponente.
// Uobičajeno, objekt lista taj je koji radi nešto konkretno, a Composite
// objekti samo delegiraju svojim potkomponentama
class Leaf : Component
{
    public override string Operation()
    {
        return "List";
    }

    public override bool IsComposite()
    {
        return false;
    }
}

// Composite klasa predstavlja kompleksne dijelove koji mogu imati
// potkomponente
// Uobičajeno, Composite objekt delegira posao svojim potkomponentama i
// onda sumira rezultat.
class Composite : Component
{
    protected List<Component> _children = new List<Component>();

    public override void Add(Component component)
    {
        this._children.Add(component);
    }

    public override void Remove(Component component)
    {
        this._children.Remove(component);
    }

    // Composite izvršava svoju primarnu logiku na sebi svojstven način.
    // Prolazi rekurzivno kroz sve potkomponente, skuplja i sumira njihov
    // rezultat. Potkomponente isto to rade i svojim potkomponentama, sve
    // dok ne dođu do krajnjeg lista hijerarhije stabla.

```

```

public override string Operation()
{
    int i = 0;
    string result = "Grana(";

    foreach (Component component in this._children)
    {
        result += component.Operation();
        if (i != this._children.Count - 1)
        {
            result += "+";
        }
        i++;
    }

    return result + ")";
}

}

class Client
{
    // Klijentski kod radi sa svim komponentama preko baznog sučelja.
    public void ClientCode(Component leaf)
    {
        Console.WriteLine($"REZULTAT: {leaf.Operation()}\n");
    }

    // Zahvaljujući činjenici da su operacije upravljanja podčvorovima
    // deklarirane u baznoj Component klasi, klijentski kod može raditi s bilo
    // kojom komponentom,
    // jednostavnom ili kompleksnom, bez da ovisi o konkretnoj klasi.
    public void ClientCode2(Component component1, Component component2)
    {
        if (component1.IsComposite())
        {
            component1.Add(component2);
        }

        Console.WriteLine($"REZULTAT: {component1.Operation()}");
    }
}

```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        // Na ovakav način klijentski kod može podržavati komponente
        // jednostavnog lista
        Leaf leaf = new Leaf();
        Console.WriteLine("Klijent: Dobivam jednostavnu komponentu:");
        client.ClientCode(leaf);

        // ...isto tako i kompleksne kompozite.
        Composite tree = new Composite();
        Composite branch1 = new Composite();
        branch1.Add(new Leaf());
        branch1.Add(new Leaf());
        Composite branch2 = new Composite();
        branch2.Add(new Leaf());
        tree.Add(branch1);
        tree.Add(branch2);
        Console.WriteLine("Klijent: Sad ima kompozitno stablo:");
        client.ClientCode(tree);

        Console.WriteLine("Klijent: Ne trebam provjeriti komponentne klase čak i kad
        upravljam stablom:\n");
        client.ClientCode2(tree, leaf);
    }
}

// Ispis programa:
/*
Klijent: Dobivam jednostavnu komponentu:
RESULT: List
Klijent: Sad ima kompozitno stablo:

```

RESULT: Grana(Grana(List+List)+Grana(List))

Klijent: Ne trebam provjeriti komponentne klase čak i kad upravljam stablom:

RESULT: Grana(Grana(List+List)+Grana(List)+List)

***/**

6.2.4. Decorator

Decorator ili **oblikovanje** dekoratera predložak omogućuje nam dodavanje novog ponašanja objektu dinamički, tako da ga stavljamo u specijalni objekt omotača. Korištenjem dekoratora možemo omotati objekt koliko god puta želimo jer ciljani objekt i dekorator prate isto sučelje. Objekt koji je rezultat dobit će sva ponašanja od svih omota.

Ovaj predložak čest je u C#-u, pogotovo u kodu u relaciji sa *streamovima*. *Decorator* ćemo prepoznati po stvaranju metoda ili konstruktora, koji primaju objekte iste klase ili sučelja kao trenutna klasa.

Primjer korištenja dekoratora:

using System;

namespace Decorator

{

// Bazno Component sučelje definira operacije koje mogu biti promijenjene dekoratorima.

public abstract class Component

{

public abstract string Operation();

}

// ConcreteComponent pruža osnovnu implementaciju operacija.

// Može biti više varijanti ove klase.

class ConcreteComponent : Component

{

public override string Operation()

```

    {
        return "ConcreteComponent";
    }
}

```

// Bazna Decorator klasa prati isto sučelje kao i ostale komponente.
 // Primarna svrha ove klase jest da definira sučelje omotača za sve konkretne
 // dekoratore. Osnovna implementacija omotačkog koda može sadržavati
 // polje za pohranu omotane komponente i načina na koji se inicijalizira.

```

abstract class Decorator : Component
{
    protected Component _component;

    public Decorator(Component component)
    {
        this._component = component;
    }

    public void SetComponent(Component component)
    {
        this._component = component;
    }

    // Decorator delegira sav rad omotač komponenti.
    public override string Operation()
    {
        if (this._component != null)
        {
            return this._component.Operation();
        }
        else
        {
            return string.Empty;
        }
    }
}

// Konkretni Decoratori zovu omotani objekt i mijenjanju rezultat na neki način
class ConcreteDecoratorA : Decorator
{

```

```

public ConcreteDecoratorA(Component comp) : base(comp)
{
}

// Dekoratori mogu pozivati roditeljsku implementaciju operacije, umjesto
// da zovu omotani objekt direktno. Ovaj način pojednostavljuje proširenje
// dekorator klasa.
public override string Operation()
{
    return $"ConcreteDecoratorA({base.Operation()})";
}
}

// Dekoratori mogu izvršiti svoje ponašanje bilo prije ili nakon poziva
// omotanog objekta

class ConcreteDecoratorB : Decorator
{
    public ConcreteDecoratorB(Component comp) : base(comp)
    {
    }

    public override string Operation()
    {
        return $"ConcreteDecoratorB({base.Operation()})";
    }
}

public class Client
{
    // Klijentski kod radi sa svim objektima koristeći Component sučelje.
    // Ovako može ostati nezavisan naspram konkretne klase komponenti s
    // kojima radi.
    public void ClientCode(Component component)
    {
        Console.WriteLine("REZULTAT: " + component.Operation());
    }
}

class Program

```

```

{
    static void Main(string[] args)
    {
        Client client = new Client();

        var simple = new ConcreteComponent();
        Console.WriteLine("Klijent: Dobivam jednostavnu komponentu:");
        client.ClientCode(simple);
        Console.WriteLine();

        // Primijetite kako dekoratori mogu omotati ne samo jednostavne
        // komponente, nego i druge dekoratore isto tako.
        ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(simple);
        ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(decorator1);
        Console.WriteLine("Klijent: Sad imam dekoriranu komponentu:");
        client.ClientCode(decorator2);
    }
}
}
// Ispis programa:
/*
Klijent: Dobivam jednostavnu komponentu:
REZULTAT: ConcreteComponent

Klijent: Sad imam dekoriranu komponentu:
REZULTAT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))
*/

```

6.2.5. Facade

Facade ili **oblikovanje** pročelja predložak pruža pojednostavljena (no ujedno i limitirana) sučelja prema kompleksnom sustavu klasa ili biblioteka. Ovaj predložak smanjuje kompleksnost aplikacija, ali pomaže i u pomicanju neželjenih ovisnosti na jedno mjesto.

Facade predložak često se koristi u C# aplikacijama, a pogotovo je dobar kad radimo s kompleksnim API-jima i bibliotekama. Može se prepoznati u klasi koja ima jednostavno sučelje, ali delegira većinu posla drugim klasama. Uobičajeno upravlja punim ciklusom objekata koje koriste.

Primjer *facade* predloška:

```
using System;
```

```
namespace RefactoringGuru.DesignPatterns.Facade.Conceptual
```

```
{  
    // Facade klasa pruža jednostavno sučelje kompleksnoj logici jednog ili  
    // više podsustava. Facade delegira klijentski zahtjev odgovarajućem  
    // objektu unutar podsustava. Facade je odgovoran i za upravljanje  
    // njihovim životnim ciklusom. Sve ovo štiti klijent od neželjene  
    // kompleksnosti podsustava  
    public class Facade  
    {  
        protected Subsystem1 _subsystem1;  
  
        protected Subsystem2 _subsystem2;  
  
        public Facade(Subsystem1 subsystem1, Subsystem2 subsystem2)  
        {  
            this._subsystem1 = subsystem1;  
            this._subsystem2 = subsystem2;  
        }  
  
        // Facade metode je prečac prema sofisticiranoj funkcionalnosti  
        // podsustava. No klijenti mogu doći samo do dijela mogućnosti
```



```

// podsustava
public string Operation()
{
    string result = "Facade inicijalizira podsustav:\n";
    result += this._subsystem1.operation1();
    result += this._subsystem2.operation1();
    result += "Facade naređuje podsustavu da izvrši:\n";
    result += this._subsystem1.operationN();
    result += this._subsystem2.operationZ();
    return result;
}
}

```

// Subsystem (podsustav) može primiti zahtjeve od Facade ili klijenta
// direktno. U svakom slučaju, podsustavu je Facade samo još jedan
// klijent i nije dio podsustava.

```

public class Subsystem1
{
    public string operation1()
    {
        return "Subsystem1: Spreman!\n";
    }

    public string operationN()
    {
        return "Subsystem1: Kreni!\n";
    }
}

```

// Neki facades mogu raditi s višestrukim podsustavima istovremeno.

```

public class Subsystem2
{
    public string operation1()
    {
        return "Subsystem2: Spremi se!\n";
    }

    public string operationZ()
    {

```

```

        return "Subsystem2: Pucaj!\n";
    }
}

class Client
{
    // Klijentski kod radi s kompleksnim podsustavima preko jednostavnog
    // sučelja koje pruža Facade. Kad Facade upravlja životnim ciklusom
    // podsustava, klijent ne mora čak ni znati za postojanost podsustava.
    // Ovaj pristup pomaže da kontroliramo kompleksnost.
    public static void ClientCode(Facade facade)
    {
        Console.Write(facade.Operation());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod može imati stvorene neke podsustavne objekte.
        // Kako god bilo, moglo bi biti korisno inicijalizirati Facade
        // s tim objektima umjesto dopustiti da Facade stvori nove instance
        Subsystem1 = new Subsystem1();
        Subsystem2 = new Subsystem2();
        Facade = new Facade(subsystem1, subsystem2);
        Client.ClientCode(facade);
    }
}

// Ispis programa:
/*
Facade inicijalizira podsustav:
Subsystem1: Spreman!
Subsystem2: Spremi se!
Facade naređuje podsustavu da izvrši:
Subsystem1: Kreni!
Subsystem2: Pucaj!
*/

```

6.2.6. Flyweight

Flyweight ili **predložak muhavac** omogućuje programima da podržavaju ogromnu količinu objekata tako da drže njihove memorijske potrebe niskima. Ovaj predložak postiže to tako da dijeli dijelove stanja objekata između višestrukih objekata. Drugim riječima, štedi nam RAM tako da predmemorira (*caching*) iste podatke koje koriste različiti objekti.

Flyweight predložak ima jednu jedinu ulogu, smanjivanje potrebe za memorijom. Ako program nema problema s manjkom RAM-a, ovaj predložak najbolje je ignorirati. Ovaj predložak ćemo prepoznati po tome što metode stvaranja objekata vraćaju predmemorirane objekte umjesto da stvaraju nove.

Primjer korištenja:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.Json;

namespace Flyweight
{
    // Flyweight pohranjuje zajednički dio stanja (također zvano intrinzično
    // stanje) koje pripada više stvarnih poslovnih entiteta. Flyweight
    // prihvaća ostatak stanja (ekstrinzična stanja, jedinstvena za svaki entitet)
    // preko parametara metoda.
    public class Flyweight
    {
        private Automobil _sharedState;

        public Flyweight(Automobil Automobil)
        {
            this._sharedState = Automobil;
        }

        public void Operation(Automobil uniqueState)
```

```

    {
        string s = JsonSerializer.Serialize(this._sharedState);
        string u = JsonSerializer.Serialize(uniqueState);
        Console.WriteLine($"Flyweight: Prikaz dijeljenog {s} i jedinstvenog {u}
stanja.");
    }
}

// FlyweightFactory stvara i upravlja Flyweight objektima. To osigurava
// da flyweightovi budu dijeljeni ispravno. Kad klijent zahtijeva
// flyweight, tvornica vraća postojeću instancu ili stvara novu ako
// jedna još ne postoji
public class FlyweightFactory
{
    private List<Tuple<Flyweight, string>> flyweights = new List<Tuple<Flyweight,
string>>();

    public FlyweightFactory(params Automobil[] args)
    {
        foreach (var elem in args)
        {
            flyweights.Add(new Tuple<Flyweight, string>(new Flyweight(elem), this.
getKey(elem)));
        }
    }

    // Vraća Flyweight string hash za dano stanje.
    public string getKey(Automobil key)
    {
        List<string> elements = new List<string>();

        elements.Add(key.Model);
        elements.Add(key.Boja);
        elements.Add(key.Proizvodjac);

        if (key.Vlasnik != null && key.Broj != null)
        {
            elements.Add(key.Broj);
            elements.Add(key.Vlasnik);
        }
    }
}

```

```

    }

    elements.Sort();

    return string.Join("_", elements);
}

// vraća postojeći flyweight s danim stanjem ili stvara novi
public Flyweight GetFlyweight(Automobil sharedState)
{
    string key = this.getKey(sharedState);

    if (flyweights.Where(t => t.Item2 == key).Count() == 0)
    {
        Console.WriteLine("FlyweightFactory: Ne mogu naći flyweight, stvaram
novi.");

        this.flyweights.Add(new Tuple<Flyweight, string>(new Flyweight(shared-
State), key));
    }
    else
    {
        Console.WriteLine("FlyweightFactory: Ponovno korištenje postojećeg
flyweight.");
    }
    return this.flyweights.Where(t => t.Item2 == key).FirstOrDefault().Item1;
}

public void listFlyweights()
{
    var count = flyweights.Count;
    Console.WriteLine($"\\nFlyweightFactory: Imam {count} flyweightova:");
    foreach (var flyweight in flyweights)
    {
        Console.WriteLine(flyweight.Item2);
    }
}
}

```

```

public class Automobil
{
    public string Vlasnik { get; set; }

    public string Broj { get; set; }

    public string Proizvodjac { get; set; }

    public string Model { get; set; }

    public string Boja { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod stvara nekoliko flyweightova
        // u inicijalizacijskoj fazi aplikacije.
        var factory = new FlyweightFactory(
            new Automobil { Proizvodjac = "Chevrolet", Model = "Camaro2018",
                           Boja = "roza" },
            new Automobil { Proizvodjac = "Mercedes Benz", Model = "C300",
                           Boja = "crni" },
            new Automobil { Proizvodjac = "Mercedes Benz", Model = "C500",
                           Boja = "crveni" },
            new Automobil { Proizvodjac = "BMW", Model = "M5", Boja = "crveni" },
            new Automobil { Proizvodjac = "BMW", Model = "X6", Boja = "bijeli" }
        );
        factory.listFlyweights();

        addAutomobilToPoliceDatabase(factory, new Automobil
        {
            Broj = "CL234IR",
            Vlasnik = "James Doe",
            Proizvodjac = "BMW",
            Model = "M5",
            Boja = "crveni"
        });
    }
}

```

```

    });

    addAutomobilToPoliceDatabase(factory, new Automobil
    {
        Broj = "CL234IR",
        Vlasnik = "James Doe",
        Proizvodjac = "BMW",
        Model = "X1",
        Boja = "crveni"
    });

    factory.listFlyweights();
}

public static void addAutomobilToPoliceDatabase(FlyweightFactory factory,
Automobil Automobil)
{
    Console.WriteLine("\nKlijent: Dodajem Automobil u bazu podataka.");

    var flyweight = factory.GetFlyweight(new Automobil
    {
        Boja = Automobil.Boja,
        Model = Automobil.Model,
        Proizvodjac = Automobil.Proizvodjac
    });

    // Klijentski kod pohranjuje ili računa ekstrinzično stanje i
    // prosljeđuje flyweight metodi.
    flyweight.Operation(Automobil);
}
}
}

// Ispis programa:

/*
FlyweightFactory: Imam 5 flyweightova:
Camaro2018_Chevrolet_roza
crni_C300_Mercedes Benz

```

C500_Mercedes Benz_crveni
BMW_M5_crveni
BMW_bijeli_X6

Klijent: Dodajem Automobil u bazu podataka.

FlyweightFactory: Ponovno korištenje postojećeg flyweighta.

Flyweight: Prikaz dijeljenog {"Vlasnik":null,"Broj":null,"Proizvodjac":"BMW","Model":"M5","Boja":"crveni"} i jedinstvenog {"Vlasnik":"James Doe","Broj":"CL234IR","Proizvodjac":"BMW","Model":"M5","Boja":"crveni"} stanja.

Klijent: Dodajem Automobil u bazu podataka.

FlyweightFactory: Ne mogu naći flyweight, stvaram novi.

Flyweight: Prikaz dijeljenog {"Vlasnik":null,"Broj":null,"Proizvodjac":"BMW","Model":"X1","Boja":"crveni"} i jedinstvenog {"Vlasnik":"James Doe","Broj":"CL234IR","Proizvodjac":"BMW","Model":"X1","Boja":"crveni"} stanja.

FlyweightFactory: Imam 6 flyweightova:

Camaro2018_Chevrolet_roza
crni_C300_Mercedes Benz
C500_Mercedes Benz_crveni
BMW_M5_crveni
BMW_bijeli_X6
BMW_crveni_X1
*/

6.2.7. Private class data

Private class data ili **podaci privatne klase** predložak pokušava smanjiti ekspoziciju atributa tako da limitira njegovu vidljivost. Smanjuje broj atributa klase enkapsulirajući ih u jedan podatkovni objekt. Ovo omogućuje dizajneru klase da ukloni privilegiju pisanja atributima za koje je predviđeno da budu postavljeni samo tijekom konstrukcije, čak i od metoda njihove vlastite klase.

6.2.8. Proxy

Proxy ili **predložak opunomoćenik** pruža objekt koji se ponaša kao zamjena za st-

varni servisni objekt korišten od strane klijenta. *Proxy* prima zahtjev klijenta, napravi neki posao (kontrolu pristupa, predmemoriranje itd.) i zatim proslijeđuje zahtjev servisnom objektu. *Proxy* objekt ima isto sučelje kao i servis, što ga čini zamjenjivim stvarnim objektom kad se proslijedi klijentu.

Ovaj predložak nije tako čest u C# aplikacijama, ali svakako je dobar u nekim specijalnim slučajevima. Nezamjenjiv je kad želite dodati dodatna ponašanja objektu neke već postojeće klase bez da mijenjate klijentski kod. *Proxy* delegira sav rad nekom drugom objektu. Svaka *proxy* metoda treba se na kraju referirati servisnom objektu, osim ako *proxy* nije podklasa servisa.

Primjer *proxyja*:

```
using System;
```

```
namespace Proxy
```

```
{
```

```
    // Subject sučelje deklarira uobičajene operacije za RealSubject i Proxy.
```

```
    // Dokle god klijent radi s RealSubject koristeći ovo sučelje,
```

```
    // moći ćete mu proslijediti proxy umjesto pravog subjekta.
```

```
    public interface ISubject
```

```
    {
```

```
        void Request();
```

```
    }
```

```
    // RealSubject (stvarni subjekt) sadržava neku osnovnu poslovnu logiku.
```

```
    // Uobičajeno, RealSubjects su u stanju napraviti neki koristan rad koji može
```

```
    // biti vrlo spor ili osjetljiv (ispravak uzlaznih podataka). Proxy može riješiti
```

```
    // ove probleme bez da mijenja kod RealSubjecta.
```

```
    class RealSubject : ISubject
```

```
    {
```

```
        public void Request()
```

```
        {
```

```
            Console.WriteLine("RealSubject: Obrađujem zahtjev.");
```

```
        }
```

```
    }
```

```

// Proxy ima sučelje identično RealSubjectu.
class Proxy : ISubject
{
    private RealSubject _realSubject;

    public Proxy(RealSubject realSubject)
    {
        this._realSubject = realSubject;
    }

    // Najčešća upotreba proxy predloška jest lijeno učitavanje, predmemoriranje,
    // kontrola pristupa, logiranje itd. Proxy može izvršiti jedan od tih
    // zadataka, a zatim ovisno o rezultatu proslijediti izvođenje
    // istoj metodi u povezanom RealSubject objektu
    public void Request()
    {
        if (this.CheckAccess())
        {
            this._realSubject.Request();

            this.LogAccess();
        }
    }

    public bool CheckAccess()
    {
        // Neka stvarna provjera treba ići ovdje.
        Console.WriteLine("Proxy: Provjera pristupa prije pokretanja pravog zaht-
jeva.");

        return true;
    }

    public void LogAccess()
    {
        Console.WriteLine("Proxy: Logiranje vremena zahtjeva.");
    }
}

```

```

public class Client
{
    // Klijentski kod treba raditi sa svim objektima (sa subjectima i proxyjima)
    // preko Subject sučelja da bi podržao oba stvarna subjekta i proxyje.
    // U stvarnom životu, ipak, klijent uglavnom radi sa stvarnim subjektima
    // direktno. U tom slučaju, da implementiramo predložak jednostavnijem,
    // možemo proširiti proxy od klase stvarnog subjekta.
    public void ClientCode(ISubject subject)
    {
        // ...

        subject.Request();

        // ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        Console.WriteLine("Klijent: Izvršavam klijentski kod sa stvarnim subjek-
tom:");
        RealSubject = new RealSubject();
        client.ClientCode(realSubject);

        Console.WriteLine();

        Console.WriteLine("Klijent: Izvršavam isti klijentski kod s proxyjem:");
        Proxy proxy = new Proxy(realSubject);
        client.ClientCode(proxy);
    }
}

// Ispis programa:
/*

```

Klijent: Izvršavam klijentski kod sa stvarnim subjektom:

RealSubject: Obrađujem zahtjev.

Klijent: Izvršavam isti klijentski kod s proxyjem:

Proxy: Provjera pristupa prije pokretanja pravog zahtjeva.

RealSubject: Obrađujem zahtjev.

Proxy: Logiranje vremena zahtjeva.

*/

6.3. Predlošci ponašanja

Ovi predlošci ponašanja specifično se odnose na komunikaciju među objektima, odnosno na efektivnu komunikaciju i dodjeljivanje odgovornosti među objektima.

6.3.1. Chain of responsibility

Chain of responsibility ili **lanac odgovornosti** predložak omogućuje prosljeđivanje zahtjeva kroz lanac potencijalnih rukovatelja zahtjeva dokle god jedan ne rukuje zahtjevom. Ovaj predložak omogućuje višestrukim objektima da mogu rukovati zahtjevom bez odspajanja klase pošiljatelja prema konkretnoj klasi primatelja. Lanac može biti dinamički komponiran tijekom izvođenja s bilo kojim rukovateljem koji prati standarde sučelja rukovatelja (*handler*).

Lanac odgovornosti uzorak nije čest u C# programima jer je relevantan samo kad kod radi s lancem objekata. Uzorak se može prepoznati po ponašanju metoda jedne grupe objekata koji indirektno pozivaju istu metodu u drugim objektima, dok istovremeno svi objekti prate isto zajedničko sučelje.

Primjer rada:

```

using System;
using System.Collections.Generic;

namespace ChainOfResponsibility
{
    // Handler sučelje deklarira metodu za izgradnju lanca handlera.
    // Također deklarira metodu za izvršavanje zahtjeva
    public interface IHandler
    {
        IHandler SetNext(IHandler handler);

        object Handle(object request);
    }

    // Ponašanje lanca može se implementirati unutar bazne handler klase.
    abstract class AbstractHandler : IHandler
    {
        private IHandler _nextHandler;

        public IHandler SetNext(IHandler handler)
        {
            this._nextHandler = handler;

            // Vraćanje handlera odatle će nam omogućiti da povežemo handlera
            // na jednostavan način poput ovog:
            // majmun.SetNext(vjeverica).SetNext(pas);
            return handler;
        }

        public virtual object Handle(object request)
        {
            if (this._nextHandler != null)
            {
                return this._nextHandler.Handle(request);
            }
            else
            {
                return null;
            }
        }
    }
}

```

```
}  
}
```

```
class MajmunHandler : AbstractHandler  
{  
    public override object Handle(object request)  
    {  
        if ((request as string) == "Banana")  
        {  
            return $"Majmun: Pojest cú {request.ToString()}.\\n";  
        }  
        else  
        {  
            return base.Handle(request);  
        }  
    }  
}
```

```
class VjevericaHandler : AbstractHandler  
{  
    public override object Handle(object request)  
    {  
        if (request.ToString() == "Orah")  
        {  
            return $"Vjeverica: Pojest cú {request.ToString()}.\\n";  
        }  
        else  
        {  
            return base.Handle(request);  
        }  
    }  
}
```

```
class PasHandler : AbstractHandler  
{  
    public override object Handle(object request)  
    {  
        if (request.ToString() == "Mesna okruglica")  
        {
```

```

        return $"Pas: Pojest ću {request.ToString()}.\\n";
    }
    else
    {
        return base.Handle(request);
    }
}
}

```

class Client

```

{
    // Klijentski kod uobičajeno je prilagođen da radi s jednim handlerom.
    // U većini slučajeva nije ni svjestan da je taj handler dio lanca.
    public static void ClientCode(AbstractHandler handler)
    {
        foreach (var food in new List<string> { "Orah", "Banana", "Šalica kave" })
        {
            Console.WriteLine($"Klijent: Tko želi {food}?");

            var result = handler.Handle(food);

            if (result != null)
            {
                Console.Write($" {result}");
            }
            else
            {
                Console.WriteLine($" {food} nije konzumirano.");
            }
        }
    }
}

```

class Program

```

{
    static void Main(string[] args)
    {
        // Drugi dio klijentskog koda gradi lanac.
    }
}

```

```

var Majmun = new MajmunHandler();
var Vjeverica = new VjevericaHandler();
var Pas = new PasHandler();

Majmun.SetNext(Vjeverica).SetNext(Pas);

// Klijent bi trebao moći poslati zahtjev bilo kojem handleru,
// ne samo prvom u lancu.
Console.WriteLine("Lanac: Majmun > Vjeverica > Pas\n");
Client.ClientCode(Majmun);
Console.WriteLine();

Console.WriteLine("Podlanac: Vjeverica > Pas\n");
Client.ClientCode(Vjeverica);
    }
}
}

// Ispis programa:
/*
Lanac: Majmun > Vjeverica > Pas

Klijent: Tko želi Orah?
    Vjeverica: Pojest ću Orah.
Klijent: Tko želi Banana?
    Majmun: Pojest ću Banana.
Klijent: Tko želi Šalica kave?
    Šalica kave nije konzumirano.

Podlanac: Vjeverica > Pas

Klijent: Tko želi Orah?
    Vjeverica: Pojest ću Orah.
Klijent: Tko želi Banana?
    Banana nije konzumirano.
Klijent: Tko želi Šalica kave?
    Šalica kave nije konzumirano.
*/

```


6.3.2. Command

Command ili **predložak naredba** pretvara zahtjev ili jednostavnu operaciju u objekt. Konverzija omogućuje odgođeno ili udaljeno izvršavanje naredbi, spremanje povijesti komandi itd.

Komanda predložak relativno je česta u C#-u, uobičajeno kao alternativa pozivima parametriziranih ulazno-izlaznih elemenata s akcijama. Također se koristi za redovanje zadataka, praćenje povijesti operacija itd.

Command predložak prepoznatljiv je po metodama ponašanja u apstraktnim/sučelje tipskim pošiljaocima, koji priziva metodu u implementaciji različitih apstraktno/sučelje tipa primatelja koji su enkapsulirani pomoću naredbene implementacije prilikom njihovog stvaranja. Komandne klase uobičajeno su limitirane na specifične akcije.

Primjer korištenja:

```
using System;
```

```
namespace Command
```

```
{  
    // Command sučelje deklarira metodu za izvršavanje naredbe.  
    public interface ICommand  
    {  
        void Execute();  
    }  
  
    // Neke komande mogu implementirati jednostavne operacije.  
    class SimpleCommand : ICommand  
    {  
        private string _payload = string.Empty;  
  
        public SimpleCommand(string payload)  
        {  
            this._payload = payload;  
        }  
    }  
}
```

```

public void Execute()
{
    Console.WriteLine($"SimpleCommand: Vidiš, mogu raditi jednostavne stvari poput ispisa ({this._payload})");
}
}

```

// Ipak, neke komande mogu delegirati kompleksnije operacije drugim objektima koje zovemo primatelji (reciever)

```

class ComplexCommand : ICommand
{
    private Receiver _receiver;

    // Podaci potrebni za pokretanje metode primatelja.
    private string _a;

    private string _b;

    // Kompleksne komande mogu primiti jedan ili više primateljskih objekata
    // s bilo kojim podacima preko konstruktora.
    public ComplexCommand(Receiver, string a, string b)
    {
        this._receiver = receiver;
        this._a = a;
        this._b = b;
    }

    // Naredbe mogu delegirati bilo kojoj metodi primatelja.
    public void Execute()
    {
        Console.WriteLine("ComplexCommand: Komplicirana stvar treba biti izvršena s objektom primatelja.");
        this._receiver.DoSomething(this._a);
        this._receiver.DoSomethingElse(this._b);
    }
}

```

// Receiver klasa sadrži neku važnu poslovnu logiku. Znaju

// kako izvršiti više operacija povezanih s izvršavanjem zahtjeva.

// Bilo koja klasa može služiti kao Reciever

class Receiver

```
{  
    public void DoSomething(string a)  
    {  
        Console.WriteLine($"Receiver: Radim na ({a}.)");  
    }  
  
    public void DoSomethingElse(string b)  
    {  
        Console.WriteLine($"Receiver: Također radim na ({b}.)");  
    }  
}
```

// Invoker se povezuje s jednom ili više naredbi. Šalje zahtjev naredbi.

class Invoker

```
{  
    private ICommand _onStart;  
  
    private ICommand _onFinish;  
  
    // Inicijaliziraj naredbe.  
    public void SetOnStart(ICommand command)  
    {  
        this._onStart = command;  
    }  
  
    public void SetOnFinish(ICommand command)  
    {  
        this._onFinish = command;  
    }  
}
```

// Invoker ne ovisi o konkretnoj naredbi ili klasi primatelja.

// Invoker prosljeđuje zahtjev primatelju indirektno tako da

// izvršava naredbe.

```
public void DoSomethingImportant()  
{
```

```

    Console.WriteLine("Invoker: Želi li netko da se nešto napravi prije nego  

što počnem?");

    if (this._onStart is ICommand)
    {
        this._onStart.Execute();
    }

    Console.WriteLine("Invoker: ...radi nešto važno...");

    Console.WriteLine("Invoker: Želi li netko nešto napraviti nakon što  

završim?");

    if (this._onFinish is ICommand)
    {
        this._onFinish.Execute();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod može parametrizirati invokera bilo kojim naredbama.
        Invoker invoker = new Invoker();
        invoker.SetOnStart(new SimpleCommand("Reci Pozdrav!"));
        Receiver = new Receiver();
        invoker.SetOnFinish(new ComplexCommand(receiver, "Pošalji email",
        "Spremi izvješće"));

        invoker.DoSomethingImportant();
    }
}

// Ispis programa:
/*
Invoker: Želi li netko da se nešto napravi prije nego što počnem?
SimpleCommand: Vidiš, mogu raditi jednostavne stvari poput ispisa (Reci Pozdrav!)

```

Invoker: ...radi nešto važno...

Invoker: Želi li netko nešto napraviti nakon što završim?

ComplexCommand: Komplicirana stvar treba biti izvršena s objektom primatelja.

Receiver: Radim na (Pošalji email.)

Receiver: Također radim na (Spremi izvješće.)

*/

6.3.3. Interpreter

Interpreter ili **predložak tumač** pruža način da evaluiramo jezičnu gramatiku ili izraz. Čest je u parsiranju SQL-a, simbolnim procesima itd. Ukratko, koristi se kad imamo neki jezik za interpretirati, a najbolje radi kad je gramatika jednostavna i učinkovitost nije kritična za rad.

Najbolji primjer primjene ovog predloška jest sam C# kompajler koji interpretira C# izvorni kod u *byte* kod koji CLR razumije. Daljnji primjeri ovog predloška bili bi Google Translate ili bilo koji drugi prevoditelj koji uzima za ulaz neki jezik i za izlaz daje neki drugi jezik.

U C# programskom jeziku ovaj predložak rijetko se samostalno koristi, no čest je na razini *frameworka* u smislu procesiranja simbola i prepoznavanja tekstualnog unosa. Tipičan primjer bila bi interpretacija datuma i vremena te njihov prikaz u specifičnom formatu.

6.3.4. Iterator

Iterator ili **predložak ponavljač** omogućuje sekvencijski prolazak kroz kompleksnu podatkovnu strukturu bez izlaganja njegove unutarnje strukture. Zahvaljujući iterator predlošku možemo prolaziti elemente različitih kolekcija na sličan način koristeći jedinstveno sučelje. Najbolji primjer ovoga nam je prolaz kroz kolekcije korištenjem *foreach* petlje.

Ovaj predložak čest je u C# kodu. Mnoge biblioteke koriste ga za prolaz kroz njihove kolekcije.

Iterator uzorak lagano je prepoznati po navigacijskim metodama poput *next*, *previous* i drugih. Klijentski kod koji koristi iteratore ne mora imati direktan pristup kolekciji koju prolazimo.

Rad s iteratorom najbolje je demonstrirati primjerom:

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace Iterator
{
    abstract class Iterator : IEnumerator
    {
        object IEnumerator.Current => Current();

        // Vraća ključ trenutnog elementa
        public abstract int Key();

        // Vraća trenutni element
        public abstract object Current();

        // Pomak prema sljedećem elementu
        public abstract bool MoveNext();

        // Vraća se na prvi element
        public abstract void Reset();
    }

    abstract class IteratorAggregate : IEnumerable
    {
        // Vraća iterator ili drugi IteratorAggregate za implementirani objekt
        public abstract IEnumerator GetEnumerator();
    }
}
```

// Konkretni iterator implementira razne algoritme prolaska. Ove klase
// pohranjuju trenutnu poziciju u svakom trenutku.

```
class AbecedniRedIterator : Iterator
```

```
{
```

```
    private KolekcijaRijeci _collection;
```

// Pohranjuje trenutnu poziciju. Iterator može imati mnogo članova

// za pohranu svoga stanja, ovisno o tipu kolekcije s

// kojim radi

```
    private int _position = -1;
```

```
    private bool _reverse = false;
```

```
    public AbecedniRedIterator(KolekcijaRijeci collection, bool reverse = false)
```

```
    {
```

```
        this._collection = collection;
```

```
        this._reverse = reverse;
```

```
        if (reverse)
```

```
        {
```

```
            this._position = collection.getItems().Count;
```

```
        }
```

```
    }
```

```
    public override object Current()
```

```
    {
```

```
        return this._collection.getItems()[_position];
```

```
    }
```

```
    public override int Key()
```

```
    {
```

```
        return this._position;
```

```
    }
```

```
    public override bool MoveNext()
```

```
    {
```

```
        int updatedPosition = this._position + (this._reverse ? -1 : 1);
```

```

        if (updatedPosition >= 0 && updatedPosition < this._collection.getItems().
Count)
        {
            this._position = updatedPosition;
            return true;
        }
        else
        {
            return false;
        }
    }

    public override void Reset()
    {
        this._position = this._reverse ? this._collection.getItems().Count - 1 : 0;
    }
}

// Konkretna kolekcija pruža jednu ili više metoda za dohvat nove iterator
//instance
// koja je kompatibilna s klasom kolekcije
class KolekcijaRijeci : IteratorAggregate
{
    List<string> _collection = new List<string>();

    bool _direction = false;

    public void ReverseDirection()
    {
        _direction = !_direction;
    }

    public List<string> getItems()
    {
        return _collection;
    }

    public void AddItem(string item)
    {

```



```

        this._collection.Add(item);
    }

    public override IEnumerator GetEnumerator()
    {
        return new AbecedniRedIterator(this, _direction);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod može, a ne mora znati za konkretni iterator
        // ili kolekciju klasa
        var collection = new KolekcijaRijeci();
        collection.AddItem("Prvi");
        collection.AddItem("Drugi");
        collection.AddItem("Treći");

        Console.WriteLine("Prolazak kolekcije:");

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }

        Console.WriteLine("\nObrnuti prolaz kolekcije:");

        collection.ReverseDirection();

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }
    }
}

```

```
// Ispis programa:
/*
Prolazak kolekcije:
Prvi
Drugi
Treći

Obrnuti prolaz kolekcije:
Treći
Drugi
Prvi
*/
```

6.3.5. Mediator

Mediator ili **predložak posrednik** smanjuje spajanje među komponentama programa tako da napravi indirektnu komunikaciju između njih preko specijalnih posredničkih objekata. Omogućuje da lagano modificiramo, proširimo i ponovno iskoristimo individualne komponente jer nisu više ovisne o drugim klasama.

Najčešća upotreba mediator predloška u C#-u leži u komunikaciji između komponenti grafičkog sučelja u aplikaciji. Sinonim za mediator jest kontroler u MVC predlošku.

Primjer korištenja:

```
using System;
```

```
namespace Mediator
```

```
{
```

```
    // Mediator sučelje deklarira metodu korištenu od strane komponenti koje će
    // obavijestiti mediator o raznim događajima. Mediator može reagirati
    // na te događaje i proslijediti izvršavanje drugim komponentama
```

```
    public interface IMediator
```

```
    {
```

```
        void Notify(object sender, string ev);
```

```
    }
```

```

// Konkretni mediatori implementiraju suradničko ponašanje koordinirajući
// više komponenti
class ConcreteMediator : IMediator
{
    private Component1 _component1;

    private Component2 _component2;

    public ConcreteMediator(Component1 component1, Component2 component2)
    {
        this._component1 = component1;
        this._component1.SetMediator(this);
        this._component2 = component2;
        this._component2.SetMediator(this);
    }

    public void Notify(object sender, string ev)
    {
        if (ev == "A")
        {
            Console.WriteLine("Mediator A reagira i okida sljedeće operacije:");
            this._component2.DoC();
        }
        if (ev == "D")
        {
            Console.WriteLine("Mediator reagira na D i okida sljedeće operacije:");
            this._component1.DoB();
            this._component2.DoC();
        }
    }
}

// Bazna komponenta pruža osnovnu funkcionalnost za spremanje
// mediatorove instance unutar komponente objekata.
class BaseComponent
{
    protected IMediator _mediator;

```

```

public BaseComponent(IMediator mediator = null)
{
    this._mediator = mediator;
}

public void SetMediator(IMediator mediator)
{
    this._mediator = mediator;
}
}

```

// Konkretné komponente implementiraju razne funkcionalnosti. Ne ovise
// o drugim komponentama. Također, ne ovise ni o jednoj konkretnoj mediator
// klasi

```

class Component1 : BaseComponent
{
    public void DoA()
    {
        Console.WriteLine("Komponenta 1 izvodi A.");

        this._mediator.Notify(this, "A");
    }

    public void DoB()
    {
        Console.WriteLine("Komponenta 1 izvodi B.");

        this._mediator.Notify(this, "B");
    }
}

```

```

class Component2 : BaseComponent
{
    public void DoC()
    {
        Console.WriteLine("Komponenta 2 izvodi C.");
    }
}

```

```

        this._mediator.Notify(this, "C");
    }

    public void DoD()
    {
        Console.WriteLine("Komponenta 2 izvodi D.");

        this._mediator.Notify(this, "D");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // The client code.
        Component1 component1 = new Component1();
        Component2 component2 = new Component2();
        new ConcreteMediator(component1, component2);

        Console.WriteLine("Klijent okida operaciju A.");
        component1.DoA();

        Console.WriteLine();

        Console.WriteLine("Klijent okida operaciju D.");
        component2.DoD();
    }
}

```

// Ispis programa:

/*

Klijent okida operaciju A.

Komponenta 1 izvodi A.

Mediator A reagira i okida sljedeće operacije:

Komponenta 2 izvodi C.

Klijent okida operaciju D.

Komponenta 2 izvodi D.

Mediator reagira na D i okida sljedeće operacije:

Komponenta 1 izvodi B.

Komponenta 2 izvodi C.

*/

6.3.6. Memento

Memento ili **predložak spomena** dopušta izradu snimki stanja objekta i njegovo vraćanje u budućnosti, pri tome ne kompromitirajući internu strukturu objekta s kojim radi, kao ni podataka koji se drže u snimci.

Memento princip može se postići korištenjem serijalizacije, koja je dosta česta u C#-u. Iako nije najefikasniji ni jedini način da bismo radili snimke stanja objekata, omogućuje spremanje stanja sigurnosnih kopija istovremeno štiteći strukturu originala od ostalih objekata.

Primjer korištenja:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Memento
{
    // Originator drži važna stanja koja se vremenom mogu mijenjati. On također
    // definira metode za pohranu stanja unutar mementa i drugu
    // metodu za vraćanje stanja iz njega.
    class Originator
    {
        // Radi jednostavnosti stanje ćemo spremiti u jednu varijablu.
        private string _state;
        public Originator(string state)
        {
```

```

    this._state = state;
    Console.WriteLine("Originator: Moje početno stanje je: " + state);
}

// Originatorova logika može utjecati na unutarnja stanja.
// Radi toga klijent treba napraviti sigurnosnu kopiju stanja
// prije nego što pokrene metode logike rada preko save() metode
public void DoSomething()
{
    Console.WriteLine("Originator: Radim nešto važno.");
    this._state = this.GenerateRandomString(30);
    Console.WriteLine($"Originator: i moje se stanje promijenilo u: {_state}");
}

private string GenerateRandomString(int length = 10)
{
    string allowedSymbols = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-
NOPQRSTUVWXYZ";
    string result = string.Empty;

    while (length > 0)
    {
        result += allowedSymbols[new Random().Next(0, allowedSymbols.Length)];

        Thread.Sleep(12);

        length--;
    }

    return result;
}

// Pohranjuje trenutno stanje u memento.
public IMemento Save()
{
    return new ConcreteMemento(this._state);
}

// Vraća originatorovo stanje iz memento objekta.
public void Restore(IMemento memento)

```

```

{
    if (!(memento is ConcreteMemento))
    {
        throw new Exception("Nepoznata memento klasa " + memento.ToString());
    }

    this._state = memento.GetState();
    Console.WriteLine($"Originator: Moje se stanje promijenilo u: {_state}");
}
}

// Memento sučelje pruža način da povratimo mementove metapodatke,
// poput stvaranja datuma ili imena. No ne pokazuje originatorovo
// originalno stanje.
public interface IMemento
{
    string GetName();

    string GetState();

    DateTime GetDate();
}

// Konkretni memento sadrži infrastrukturu za spremanje
// originatorova stanja.
class ConcreteMemento : IMemento
{
    private string _state;

    private DateTime _date;

    public ConcreteMemento(string state)
    {
        this._state = state;
        this._date = DateTime.Now;
    }

    // Originator koristi ovu metodu kad vraća stanje
    public string GetState()

```



```

    {
        return this._state;
    }
    // Ostatak metoda koristi se od strane Caretekera da prikažu metapodatke
    public string GetName()
    {
        return $"{this._date} / ({this._state.Substring(0, 9)})...";
    }

    public DateTime GetDate()
    {
        return this._date;
    }
}
// Caretaker ne ovisi o konkretnoj memento klasi. Time nema pristup
// originatorovu stanju, spremljenom u mementu. Radi sa svim
// mementima preko baznog Memento sučelja.
class Caretaker
{
    private List<IMemento> _mementos = new List<IMemento>();

    private Originator _originator = null;

    public Caretaker(Originator originator)
    {
        this._originator = originator;
    }

    public void Backup()
    {
        Console.WriteLine("\nCaretaker: Pohranjujem Originatorovo stanje...");
        this._mementos.Add(this._originator.Save());
    }

    public void Undo()
    {
        if (this._mementos.Count == 0)
        {

```

```

        return;
    }

    var memento = this._mementos.Last();
    this._mementos.Remove(memento);

    Console.WriteLine("Caretaker: Vraćam stanje u: " + memento.GetName());

    try
    {
        this._originator.Restore(memento);
    }
    catch (Exception)
    {
        this.Undo();
    }
}

public void ShowHistory()
{
    Console.WriteLine("Caretaker: Evo popis mementa:");

    foreach (var memento in this._mementos)
    {
        Console.WriteLine(memento.GetName());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Client code.
        Originator originator = new Originator("Super-duper-super-puper-super.");
        Caretaker caretaker = new Caretaker(originator);

        caretaker.Backup();
        originator.DoSomething();
    }
}

```

```

    caretaker.Backup();
    originator.DoSomething();

    caretaker.Backup();
    originator.DoSomething();

    Console.WriteLine();
    caretaker.ShowHistory();

    Console.WriteLine("\nClient: Sad se vratimo!\n");
    caretaker.Undo();

    Console.WriteLine("\n\nClient: Još jednom!\n");
    caretaker.Undo();

    Console.WriteLine();
}
}
}

```

// Ispis programa:

/*

Originator: Moje je početno stanje: Super-duper-super-puper-super.

Caretaker: Pohranjujem Originatorovo stanje...

Originator: Radim nešto važno.

Originator: I moje se stanje promijenilo u: oGyQlIatIDDWNgYYqJATTmdwnnG-ZQj

Caretaker: Pohranjujem Originatorovo stanje...

Originator: Radim nešto važno.

Originator: I moje se stanje promijenilo u: jBtMDDWogzzRJbTTmEwOOhZ-rjjBULe

Caretaker: Pohranjujem Originatorovo stanje...

Originator: Radim nešto važno.

Originator: I moje se stanje promijenilo u: exoHyyRkbuuNEXOhhArKccUmexP-PHZ

Caretaker: Evo popisa mementa:

18.06.2022 15:52:45 / (Super-dup...)

18.06.2022 15:52:46 / (oGyQllatl...)

18.06.2022 15:52:46 / (jBtMDDWog...)

Client: Sad se vratimo!

Caretaker: Vraćam stanje u: 12.06.2018 15:52:46 / (jBtMDDWog...)

Originator: Moje se stanje promijenilo u: jBtMDDWogzzRJbTTmEwOOhZrjBULe

Client: Još jednom!

Caretaker: Vraćam stanje u: 12.06.2018 15:52:46 / (oGyQllatl...)

Originator: Moje se stanje promijenilo u: oGyQllatIDDWNgYYqJATTmdwnnGZQj
*/

6.3.7. Null Object

Null object ili **prazan objekt** predložak poznat je kao Stub, kao aktivno ništa ili aktivno null. Pomaže u programskoj logici da se riješimo provjere je li objekt null gdje god je to moguće. Problem koji imamo kad pokušamo pozvati neku metodu na objektu koji je null jest to da će se dignuti iznimka `NullPointerException` i ako nismo osigurali oporavak od greške, program će puknuti i srušiti se.

Ovaj uzorak nije jako čest u C#-u, a često se rješava ovaj korištenjem drugih predložaka ili jednostavnom provjerom je li objekt null.

6.3.8. Observer

Observer ili **predložak promatrač** omogućuje da neki objekti obavijeste druge objekte o promjenama u njihovu stanju. Ovaj predložak pruža mogućnost da dobivamo ili ne dobivamo obavijesti o ovim događajima za svaki objekt koji implementira sučelje promatrača.

Ovaj predložak čest je u C#-u, pogotovo kod komponenti grafičkog sučelja. Pruža način da reagiramo na događaje u drugim objektima bez vezanja na njihovu klasu. Predložak se može prepoznati po metodi `pretplate`, koja pohranjuje objekte u listu i pozivima metode za ažuriranje daje objekt toj listi.

Primjer korištenja promatrača:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace Observer
{
    public interface IObserver
    {
        // Prima promjene subjekta
        void Update(ISubject subject);
    }

    public interface ISubject
    {
        // Povezuje promatrača na subjekt
        void Attach(IObserver observer);

        // Uklanja promatrača sa subjekta.
        void Detach(IObserver observer);

        // Obavještava sve promatrače o događaju.
        void Notify();
    }

    // Subject je vlasnik nekih važnih stanja i obavještava promatrača kad se
    // stanja promijene.
    public class Subject : ISubject
    {
        // Zbog jednostavnosti stanje subjekta sprema se u ovu varijablu.
        public int State { get; set; } = -0;
    }
}
```

```

// Lista pretplatnika. U stvarnosti lista pretplatnika može biti detaljnije
// pohranjena.
private List<IObserver> _observers = new List<IObserver>();

// Metoda upravljanja pretplatom.
public void Attach(IObserver observer)
{
    Console.WriteLine("Subject: Povezao promatrača.");
    this._observers.Add(observer);
}

public void Detach(IObserver observer)
{
    this._observers.Remove(observer);
    Console.WriteLine("Subject: Uklonio promatrača.");
}

// Okidač za ažuriranje svakog pretplatnika.
public void Notify()
{
    Console.WriteLine("Subject: Obavještavam promatrače...");

    foreach (var observer in _observers)
    {
        observer.Update(this);
    }
}

// Uobičajeno, logika pretplate samo je dio onoga što subjekt može napraviti.
// Subjekt obično sadrži neke važne logike rada, koje okidaju obavijest
// metodu
// kad se treba nešto važno dogoditi.
public void SomeBusinessLogic()
{
    Console.WriteLine("\nSubject: Radim nešto važno.");
    this.State = new Random().Next(0, 10);
    Thread.Sleep(15);

    Console.WriteLine("Subject: Moje se stanje promijenilo u: " + this.State);
}

```

```

        this.Notify();
    }
}

// Konkretni promatrač reagira na ažuriranja koja je dao subjekt s kojim je
// povezan.
class ConcreteObserverA : IObservable
{
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State < 3)
        {
            Console.WriteLine("ConcreteObserverA: Reagirao na događaj.");
        }
    }
}

class ConcreteObserverB : IObservable
{
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
        {
            Console.WriteLine("ConcreteObserverB: Reagirao na događaj.");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // klijentski kod
        var subject = new Subject();
        var observerA = new ConcreteObserverA();
        subject.Attach(observerA);

        var observerB = new ConcreteObserverB();
        subject.Attach(observerB);
    }
}

```

```

        subject.SomeBusinessLogic();
        subject.SomeBusinessLogic();

        subject.Detach(observerB);

        subject.SomeBusinessLogic();
    }
}
}

```

// Ispis programa:

/*

Subject: Povezao promatrača.

Subject: Povezao promatrača.

Subject: Radim nešto važno.

Subject: Moje se stanje promijenilo u: 2

Subject: Obavještavam promatrače...

ConcreteObserverA: Reagirao na događaj.

ConcreteObserverB: Reagirao na događaj.

Subject: Radim nešto važno.

Subject: Moje se stanje promijenilo u: 1

Subject: Obavještavam promatrače...

ConcreteObserverA: Reagirao na događaj.

Subject: Uklonio promatrača.

Subject: Radim nešto važno.

Subject: Moje se stanje promijenilo u: 5

Subject: Obavještavam promatrače...

*/

6.3.9. State

State ili **predložak stanje** omogućuje objektu da promijeni ponašanje kad se njegovo unutrašnje stanje promijeni. Ovaj predložak pomiče ponašanje vezano uz stanje u zasebnu klasu i prisiljava originalni objekt da delegira rad instanci te druge klase.

Predložak stanja uobičajeno se koristi u C#-u da se pretvori ogromna switch naredba u objekte. Ovaj predložak može biti prepoznat po metodama koje mijenjaju svoje ponašanje na temelju stanja samog objekta, kontrolirano izvan klase.

Primjer korištenja:

```
using System;
```

```
namespace State
```

```
{
```

```
    // Context definira sučelje interesa klijenata. Također sadrži  
    // referencu na instancu State podklase, koja predstavlja  
    // trenutno stanje Contexta.
```

```
    class Context
```

```
    {
```

```
        // Referenca na trenutno stanje Contexta.
```

```
        private State _state = null;
```

```
        public Context(State state)
```

```
        {
```

```
            this.TransitionTo(state);
```

```
        }
```

```
        // Context dozvoljava promjenu stanja State objekta tijekom izvođenja.
```

```
        public void TransitionTo(State state)
```

```
        {
```

```
            Console.WriteLine($"Context: Prijelaz u {state.GetType().Name}.");
```

```
            this._state = state;
```

```
            this._state.SetContext(this);
```

```
        }
```

```

// Context delegira dio svog ponašanja na trenutno stanje objekta.
public void Request1()
{
    this._state.Handle1();
}

public void Request2()
{
    this._state.Handle2();
}
}

// Bazna State klasa deklarira metode koje sve konkretne State klase trebaju
// implementirati i također pruža pozadinsku referencu na Context objekt,
// povezan sa State objektom. Ova pozadinska referenca može se koristiti od
// strane stanja da prebaci Context u drugo stanje.
abstract class State
{
    protected Context _context;

    public void SetContext(Context context)
    {
        this._context = context;
    }

    public abstract void Handle1();

    public abstract void Handle2();
}

// Konkretno stanje implementira različita ponašanja, povezana sa stanjem
// Contexta.
class ConcreteStateA : State
{
    public override void Handle1()
    {
        Console.WriteLine("ConcreteStateA obrađuje zahtjev1.");
        Console.WriteLine("ConcreteStateA želi promijeniti stanje sadržaja.");
    }
}

```

```

        this._context.TransitionTo(new ConcreteStateB());
    }

    public override void Handle2()
    {
        Console.WriteLine("ConcreteStateA obrađuje zahtjev2.");
    }
}

class ConcreteStateB : State
{
    public override void Handle1()
    {
        Console.WriteLine("ConcreteStateB obrađuje zahtjev1.");
    }

    public override void Handle2()
    {
        Console.WriteLine("ConcreteStateB obrađuje zahtjev2.");
        Console.WriteLine("ConcreteStateB želi promijeniti stanje sadržaja.");
        this._context.TransitionTo(new ConcreteStateA());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod.
        var context = new Context(new ConcreteStateA());
        context.Request1();
        context.Request2();
    }
}

// Ispis programa:
/*
Context: Prijelaz u ConcreteStateA.

```

```
ConcreteStateA obrađuje zahtjev1.  
ConcreteStateA želi promijeniti stanje sadržaja.  
Context: Prijelaz u ConcreteStateB.  
ConcreteStateB obrađuje zahtjev2.  
ConcreteStateB želi promijeniti stanje sadržaja.  
Context: Prijelaz u ConcreteStateA.  
*/
```

6.3.10. Strategy

Strategy ili **predložak strategija** pretvara set ponašanja u objekte i čini ih međusobno zamjenjivima unutar objekta originalnog sadržaja. Originalni objekt drži referencu prema objektu strategije i delegira ga izvršavajući željeno ponašanje. Da bi se promijenio način na koji originalni objekt izvršava svoj posao, drugi objekti mogu zamijeniti trenutno povezanu strategiju s objektom druge strategije.

Ovaj predložak čest je u C# kodu i često se koristi u mnogim bibliotekama da pruži korisnicima način da promijene ponašanje klase bez da je prošire. Predložak strategije može se prepoznati po metodama koje dozvoljavaju ugniježđenim objektima da obavljaju stvarni rad, ali i postavljacima koji dozvoljavaju zamjenu tog objekta drugim objektom.

Primjer korištenja predloška strategije:

```
using System;  
using System.Collections.Generic;  
  
namespace Strategy  
{  
    // Context definira sučelje interesa klijenata.  
    class Context  
    {  
        // Context održava referencu na jedan od objekata strategije. Također ne zna  
        // konkretnu klasu strategije. Trebao bi raditi sa svim strategijama preko  
        // Strategy sučelja.  
    }  
}
```

```

private IStrategy _strategy;

public Context()
{
    // Uobičajeno Context prima strategiju preko konstruktora, ali također
    // pruža postavljач da ga promijeni tijekom izvođenja.
    public Context(IStrategy strategy)
    {
        this._strategy = strategy;

        // Uobičajeno Context dozvoljava zamjenu Strategy objekta tijekom
        // izvođenja.
        public void SetStrategy(IStrategy strategy)
        {
            this._strategy = strategy;
        }

        // Context delegira dio posla Strategy objektu umjesto da
        // implementira višestruke verzije algoritma.
        public void DoSomeBusinessLogic()
        {
            Console.WriteLine("Context: Slaganje podataka korištenjem strategije
(nisam siguran kako ću to napraviti)");
            var result = this._strategy.DoAlgorithm(new List<string> { "a", "b", "c", "d",
"e" });

            string resultStr = string.Empty;
            foreach (var element in result as List<string>)
            {
                resultStr += element + ",";
            }

            Console.WriteLine(resultStr);
        }
    }

    // Strategy sučelje deklarira operacije uobičajene za sve podržane verzije

```

```

//istog algoritma
// Context koristi ovo sučelje da poziva algoritam definiran u konkretnoj
// strategiji.
public interface IStrategy
{
    object DoAlgorithm(object data);
}

// Konkretna strategija implementira algoritam dok prati bazno Strategy
// sučelje.
// Sučelje ih čini zamjenjivim unutar Contexta.
class ConcreteStrategyA : IStrategy
{
    public object DoAlgorithm(object data)
    {
        var list = data as List<string>;
        list.Sort();

        return list;
    }
}

class ConcreteStrategyB : IStrategy
{
    public object DoAlgorithm(object data)
    {
        var list = data as List<string>;
        list.Sort();
        list.Reverse();

        return list;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Klijentski kod odabire konkretnu strategiju i prosljeđuje je sadržaju.
    }
}

```

// Klijent bi trebao biti svjestan razlika između strategija da bi mogao napraviti ispravnu odluku

```
var context = new Context();
```

```
Console.WriteLine("Klijent: Strategija postavljena za normalno sortiranje.");  
context.SetStrategy(new ConcreteStrategyA());  
context.DoSomeBusinessLogic();
```

```
Console.WriteLine();  
Console.WriteLine("Klijent: Strategija postavljena za obrnuto sortiranje.");  
context.SetStrategy(new ConcreteStrategyB());  
context.DoSomeBusinessLogic();
```

```
    }  
}  
}
```

// Ispis programa:

/*

Klijent: Strategija postavljena za normalno sortiranje.

Context: Slaganje podataka korištenjem strategije (nisam siguran kako ću to napraviti)

a,b,c,d,e

Klijent: Strategija postavljena za obrnuto sortiranje.

Context: Slaganje podataka korištenjem strategije (nisam siguran kako ću to napraviti)

e,d,c,b,a

*/

6.3.11. Template method

Template method ili **predložak metode** dozvoljava da definiramo kostur algoritma u baznoj klasi i dopustimo podklasama da pregaze korake bez da mijenjaju općenitu strukturu algoritma.

Ovaj predložak relativno je čest u C#-u. Programeri ga često koriste da pruže korisnicima biblioteka jednostavan način da prošire standardnu funkcionalnost korištenjem nasljeđivanja. Može se prepoznati po ponašanju metoda koje već imaju neko početno ponašanje definirano unutar bazne klase.

Primjer rada:

```
using System;
```

```
namespace TemplateMethod
```

```
{
```

```
    // AbstractClass definira predložak metodu koja sadrži kostur nekog  
    // algoritma, čija je struktura poziv apstraktnim primitivnim operacijama.  
    // Konkretna podklasa treba implementirati ove operacije, ali ostaviti  
    // metodu i predložak netaknutima.
```

```
    abstract class AbstractClass
```

```
    {
```

```
        // definicija kostura algoritma
```

```
        public void TemplateMethod()
```

```
        {
```

```
            this.BaseOperation1();
```

```
            this.RequiredOperations1();
```

```
            this.BaseOperation2();
```

```
            this.Hook1();
```

```
            this.RequiredOperation2();
```

```
            this.BaseOperation3();
```

```
            this.Hook2();
```

```
        }
```

```
        // Ove operacije su implementirane.
```

```
        protected void BaseOperation1()
```



```

{
    Console.WriteLine("AbstractClass kaže: radim većinu posla");
}

protected void BaseOperation2()
{
    Console.WriteLine("AbstractClass kaže: ali dajem podklasama da pregaze
neke operacije");
}

protected void BaseOperation3()
{
    Console.WriteLine("AbstractClass kaže: ali radim većinu posla svakako");
}

// Ove operacije moraju biti implementirane u podklasama.
protected abstract void RequiredOperations1();

protected abstract void RequiredOperation2();

// Ovo su "kuke". Podklase ih mogu pregaziti, ali nije obavezno,
// budući da kuke već imaju svoju početnu (ali praznu) implementaciju.
// Kuke pružaju dodatne točke proširenja u nekim ključnim mjestima
// algoritma.
protected virtual void Hook1() {}

protected virtual void Hook2() {}
}

// Konkretna klasa moraju implementirati sve apstraktne operacije bazne
//klase.
// Mogu pregaziti i neke operacije bez početne implementacije.
class ConcreteClass1 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass1 kaže: implementirao Operation1");
    }
}

```

```

protected override void RequiredOperation2()
{
    Console.WriteLine("ConcreteClass1 kaže: implementirao Operation2");
}
}

```

// Uobičajeno, konkretna klasa pregaži samo dio bazne klase operacija

```

class ConcreteClass2 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass2 kaže: implementirao Operation1");
    }

    protected override void RequiredOperation2()
    {
        Console.WriteLine("ConcreteClass2 kaže: implementirao Operation2");
    }

    protected override void Hook1()
    {
        Console.WriteLine("ConcreteClass2 kaže: pregaženo Hook1");
    }
}

```

class Client

```

{
    // Klijentski kod poziva predložak metoda da izvrši algoritam.
    // Klijentski kod ne mora znati konkretne klase objekata s kojima radi
    // dokle god radi s objektima preko sučelja njihove bazne klase
    public static void ClientCode(AbstractClass abstractClass)
    {
        // ...
        abstractClass.TemplateMethod();
        // ...
    }
}

```

class Program

```

{
    static void Main(string[] args)
    {
        Console.WriteLine("Isti klijentski kod može raditi s različitim podklasama");

        Client.ClientCode(new ConcreteClass1());

        Console.WriteLine("\n");

        Console.WriteLine("Isti klijentski kod može raditi s različitim podklasama");
        Client.ClientCode(new ConcreteClass2());
    }
}
}

```

// Ispis programa:

/*

Isti klijentski kod može raditi s različitim podklasama

AbstractClass kaže: radim većinu posla

ConcreteClass1 kaže: implementirao Operation1

AbstractClass kaže: ali dajem podklasama da pregaze neke operacije

ConcreteClass1 kaže: implementirao Operation2

AbstractClass kaže: ali radim većinu posla svakako

Isti klijentski kod može raditi s različitim podklasama

AbstractClass kaže: radim većinu posla

ConcreteClass2 kaže: implementirao Operation1

AbstractClass kaže: ali dajem podklasama da pregaze neke operacije

ConcreteClass2 kaže: pregaženo Hook1

ConcreteClass2 kaže: implementirao Operation2

AbstractClass kaže: ali radim većinu posla svakako

*/

6.3.12. Visitor

Visitor ili **predložak posjetitelj** omogućuje dodavanje novog ponašanja postojećoj klasnoj hijerarhiji bez mijenjanja postojećeg koda. Ovaj uzorak nije popularan ni često korišten zbog velike kompleksnosti i jako uskog područja korištenja.

6.4. Predlošci dizajna programskih rješenja

Ovi predlošci opisuju način kako dizajnirati cijelo programsko rješenje. Pokazuju nam kako bi trebala ići relacija i interakcija klasa ili objekata, bez davanja specifične finalne implementacije klasa ili objekata uključenih u dizajn.

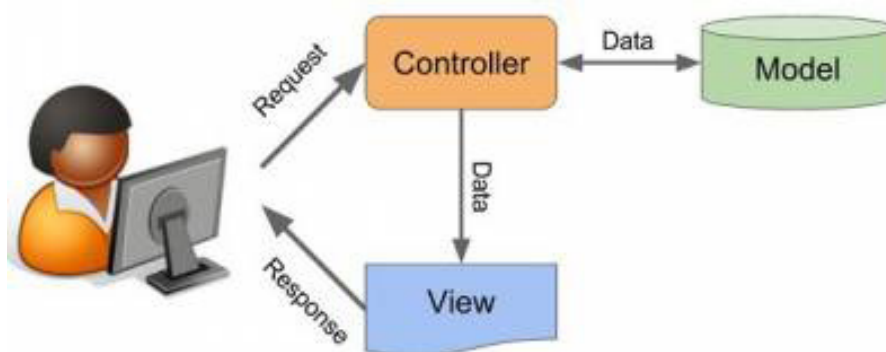
6.4.1. MVC

MVC (Model View Controller) predložak je koji odvaja aplikaciju u tri glavne logičke komponente:

- model
- view (pogled)
- controller (upravljač ili kontroler).

Svaka od ovih komponenti napravljena je da rukuje specifičnim dijelovima aplikacije. MVC je jedan od najčešćih predložaka koji se danas koristi i možemo reći da je industrijski standard za razvoj web-aplikacija.

Sljedeći dijagram predstavlja način na koji komuniciraju određeni dijelovi MVC predloška s korisnikom.



Slika 67. Vizualizacija MVC predloška

Model predstavlja logiku rada s podacima. Pod tim mislimo na to da model komunicira s kontrolerom kojem šalje potrebne podatke. Model enkapsulira način čitanja i rada s podacima, bilo da komunicira s bazom podataka, datotečnim sustavom ili nekim drugim servisom.

View ili pogled koristi se za svu ulaznu izlaznu logiku aplikacije. U pogledima radimo korisničko sučelje preko kojeg će korisnik interagirati s aplikacijom.

Za *controller* ili kontroler možemo reći da je središnji dio sustava. Kontroler je taj koji procesira svu logiku rada aplikacije, zahtjeve koji dolaze (*request*), manipulira podacima koristeći model i interaktira s pogledima da bi generirali finalni izlaz koji se prezentira korisniku u obliku odgovora (*response*).

Logiku rada možemo podijeliti u nekoliko koraka:

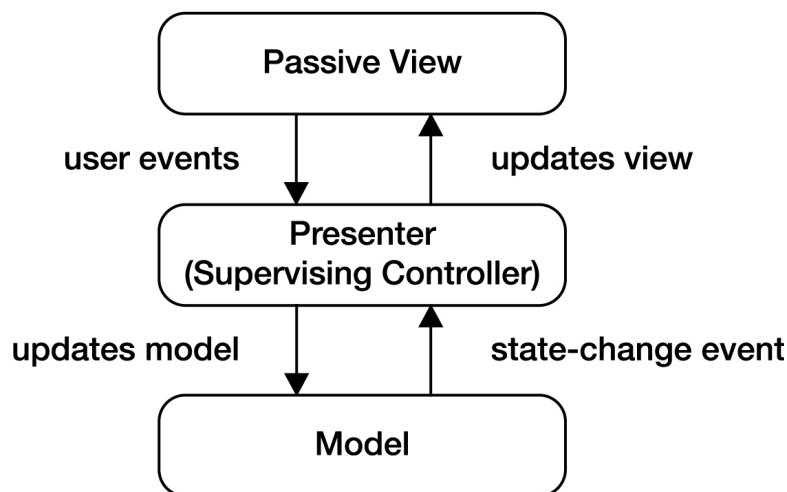
- korisnik šalje zahtjev (*request*)
- kontroler zaprima zahtjev i određuje što će s njime
- kontroler komunicira preko modela da dođe do potrebnih podataka
- kontroler podatke šalje pogledu i s pogledom generira izlaz
- izlaz se šalje korisniku u obliku odgovora (*response*).

Kako smo spomenuli, MVC je praktički postao standard za razvoj web-aplikacija. MVC predložak koriste:

- ASP.NET MVC
- ASP.NET Core
- ASP.NET WEB API
- Google Web Kit
- Spring MVC
- Laravel
- AngularJS
- Node.js
- Backbone.js
- Ember.js
- ...

6.4.2. MVP

MVP (*Model View Presenter*) predložak je derivat MVC predloška i koristi se većinom za izgradnju korisničkih sučelja. U MVP-u presenter uzima funkcionalnost posrednika, a sva se prezentacijska logika gura u presenter.



Slika 68. Vizualizacija MVP predloška

MVP je osmišljen da automatizira testiranje i poboljša odvajanje prezentacijske logike od ostatka programa.

Model je sučelje koje definira podate koji se trebaju prikazati ili na koje treba reagirati korisničko sučelje.

Pogled (*view*) pasivno je sučelje koje prikazuje podatke i preusmjerava korisničke naredbe (događaje odnosno *user events*) prezenteru da reagira na podatke.

Prezenter reagira na model i pogled. Dohvaća podatke iz repozitorija (modela) i formatira ih za prikazivanje u pogledu.

Standardno implementacija pogleda instancira konkretan prezenter objekt, dajući referencu na sebe samoga. Sljedeći kod demonstrira jednostavan konstruktor pogleda, gdje `ConcreteDomainPrezenter` implementira `IDomainPresenter` sučelje:

```

public class DomainView : IDomainView
{
    private IDomainPresenter _domainPresenter = null;

    public DomainView()
    {
        _domainPresenter = new ConcreteDomainPresenter(this);
    }
}

```

Stupanj logike koji je dozvoljen u pogledu varira između različitih implementacija. S jedne strane pogled je u potpunosti pasivan i prosljeđuje svu interakciju presenteru. Na ovakav način, kad korisnik okine metodu događaja pogleda, ne radi ništa osim što pokreće metodu presentera koja nema parametara i nema povratne vrijednosti. Presenter dohvaća podatke iz pogleda preko metoda definiranih preko sučelja pogleda. Na kraju presenter „operira” na modelu i ažurira pogled s rezultatom operacije.

Ovaj način rada pogodan je u nekim slučajevima za *web*-aplikacije, gdje pogled koji se izvršava na strani klijenta u pregledniku može biti mjesto gdje se odrađuju određene interakcije ili naredbe.

Sa stajališta raslojavanja aplikacije za presenter klasu može se smatrati da pripada aplikacijskom sloju, ali može biti viđena kao prezentacijski sloj sam po sebi, između aplikacijskog sloja i sloja korisničkog sučelja.

.NET ima podršku za MVP predložak. Isti model i presenter klasa mogu biti podržani da koriste višestruka sučelja poput ASP.NET *web*-aplikacija, Windows formi aplikacija itd.

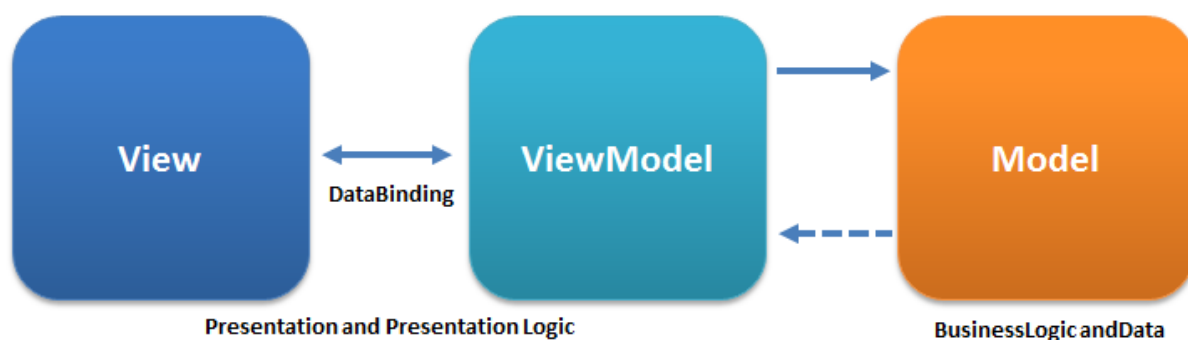
6.4.3. MVVM

MVVM (*Model-View-ViewModel*) predložak podržava razdvajanje razvoja grafičkog korisničkog sučelja (pogleda preko nekog markup jezika) od razvoja poslovne logike ili logike rada tako da pogled nije ovisan o bilo kojoj specifičnoj platformi. ViewModel, zadnji VM dio, konverter je vrijednosti, što znači da je odgovoran za konverziju podatkovnih objekata iz modela na način da objekti budu lako upravljani i prezentirani. Ovako gledajući ViewModel je više model nego pogled i odrađuje većinu logike prikaza pogleda. ViewModel može implementirati *mediator* predložak i tako organizirati pristup pozadinskoj logici oko seta slučajeva korištenja koje podržava pogled.

Sama pozadina razvoja MVVM-a jest da se pojednostavi programiranje utemeljeno na događajima korisničkih sučelja. MVVM predložak ukomponiran je u WPF i Silverlight – sjetite se XAML sintakse kad smo prolazili WPF aplikaciju.

MVVM se ponekad zove i MVB (*Model-View-Binder*), pogotovo u implementacijama na .NET platformi. Također, MVB koriste ZK i KnockoutJS.

MVM grafički prikazano izgleda ovako:



Slika 69. Vizualizacija MVVM predloška

Model reprezentira stvarno stanje, odnosno to je sadržaj.

View (pogled) struktura je i prikaz onoga što korisnik vidi na ekranu. Prikazuje prezentaciju modela i prima događaje interakcije s korisnikom u obliku klikova miša, pritisaka

tipki tipkovnice itd. Tu interakciju prosljeđuje ViewModelu preko podatkovnih veza (*Databinding*), koje su definirane da povežu view i viewmodel.

ViewModel je apstrakcija pogleda koja eksponira javna svojstva i komande. Umjesto kontrolera MVC predložka, MVVM ima poveznicu koja automatski komunicira između pogleda i njegovih vezanih svojstava unutar *viewmodela*. *Viewmodel* je opisan kao stanje podataka u modelu. Glavna razlika između *viewmodela* i prezentera u MVP predlošku jest to da prezenter ima referencu na pogled, a viewmodel je nema. Umjesto toga pogled direktno veže svojstva na *viewmodel* da bi slao i primao izmjene.

Binder ili povezivač je markup jezik. U slučaju .NET-a koristimo XAML. Binder oslobađa programera od toga da piše logiku sinkronizacije pogleda i model pogleda. Ako gledamo izvan Microsoftovih tehnologija, prisutnost tehnologije deklarativnog vezanja podataka ovaj predložak čini mogućim i bez povezivača, tipično bi se koristio MVP ili MVC predložak.

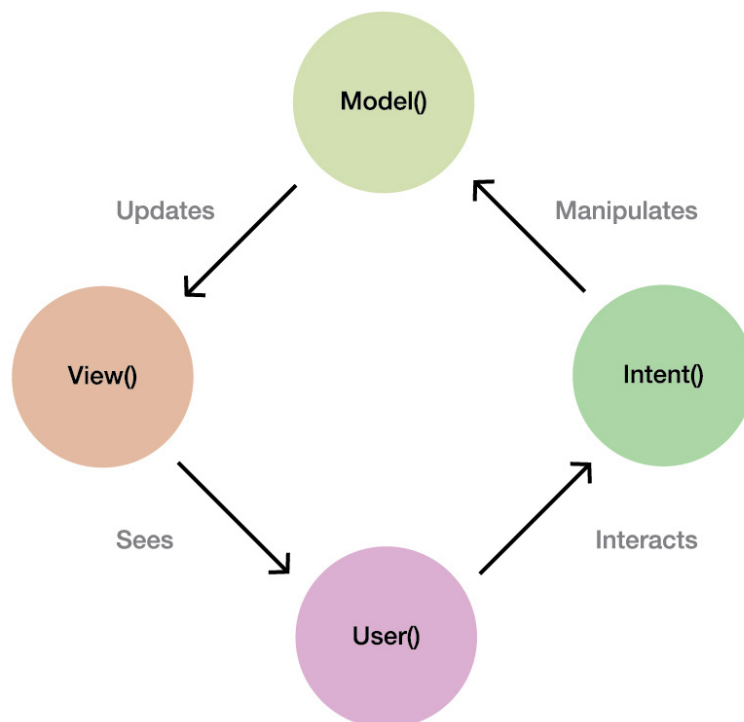
6.4.4. MVW

Za MVW (*Model View Whatever*) ili model pogled možemo reći da je naziv za predložak koji uzima dijelove prijašnje spomenutih. Nije idealno implementiran prema smjernicama drugih predložaka, no na kraju sve funkcionira kako treba. Ukratko, imamo implementiran model, implementiran pogled, no ispada da je svejedno na koji ćemo ih način vezati, ali bitno da radi. Nemamo detalje u koje bi trebalo ulaziti ovdje, jer možemo reći da su MVC, MVVM, MVP i praktički svaki model view predložak u skupini MVW predložka, s različitim implementacijama.

6.4.5. MVI

MVI (*Model View Intent*) predložak mješavina je MVP i MVVM predložaka adaptiran za reaktivno programiranje. Eliminira korištenje povratnih poziva i smanjuje broj metoda ulaza/izlaza. Također je odlično rješenje za sinkronizaciju stanja između pogleda i sloja poslovne logike.

Arhitektura je prikazana na slici 70.



Slika 70. Arhitektura MVI predloška

Sloj pogleda promatra korisničke akcije i sistemske događaje. Kao rezultat postavlja namjenu da se događaj okine te sluša i reagira na promjenu stanja u modelu.

Model je reprezentacija stanja pogleda. Sadrži sve ključne informacije potrebne da bi se pogled pravilno prikazao.

Intent ili namjena predstavlja buduću akciju koja mijenja stanje modela.

User ili korisnik upravo je to, korisnik aplikacije. Korisnik promatra i reagira na promjene stanja pogleda tako da nešto radi s aplikacijom.

Ovaj predložak čest je u mobilnim aplikacijama, no generalno gledano s .NET-om i C#-om ovaj se predložak ne koristi često.

6.4.6. MHVC

MHVC (*Hierarchical Model View Controller*) varijacija je MVC predloška. U ovoj varijaciji kontroler ima neki nadzor u smislu odabira – prvo model, a onda pogled, gdje realiziramo mehanizam potvrđivanja unutar kontrolera. Model sprečava pogled od pristupa podacima direktno.

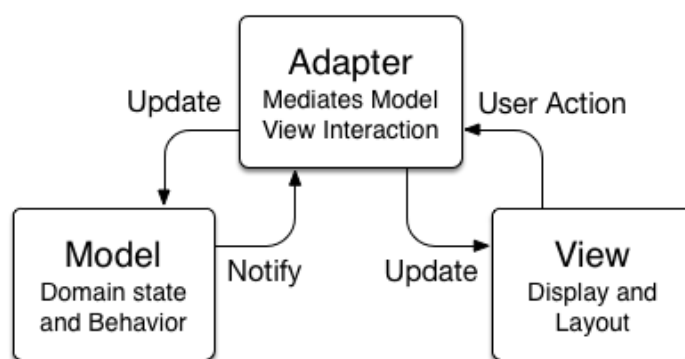
6.4.7. MMV

MMV (*Multiuse Model View*) predložak je koji je nastao kao poboljšanje MVVM predloška. Ovaj predložak specifičan je za WPF i WCF aplikacije. MMV predložak oslanja se na refleksiju da bi olakšao gradnju objekata, da bismo olakšali integraciju logičko-centričkih modela s pogledocentričkim objektima modela da smanjimo količinu dupliciranog koda. MMV je dizajniran da iskoristi specifične funkcije u C#-u, WPF-u i WCF-u da bismo bolje podržali ponovno korištenje koda između poslužitelja i klijenta.

Da bi se izveo, MMV se oslanja na set baznih klasa koje pružaju generičku funkcionalnost za primanje podataka od klijenta ili do klijenta te zatim te podatke prikazuje preko nekog korisničkog sučelja.

6.4.8. MVA

MVA (*Model View Adapter*), ili još zvan posrednovni kontroler, MVC je predložak gdje se slažu model, adapter ili posrednovni kontroler i pogled linearno bez ikakve veze direktno između modela i pogleda.



Slika 71. Vizualizacija MVA predloška

Pogled i model ne komuniciraju direktno. Pogled je potpuno odvojen od modela tako da pogled i model mogu interagirati samo preko nekog posrednika između njih. Na ovakav način samo adapter ima spoznaju o tome da postoje model i pogled jer je zadužen za posredovanje između modela i pogleda.

7. POGLAVLJE:

WEB API

U OVOM POGLAVLJU NAUČIT ĆETE:

- što su REST servisi
- primjena REST servisa
- WEB API za izradu servisa
- izraditi servis korištenjem WEB API-ja.

ASP.NET WEB API (engl. *Application Programming Interface*) *framework* je za izradu http servisa koji mogu biti konzumirani od strane većeg broja različitih klijenata kao što su preglednici i drugi programi na različitim uređajima i različitim operativnim sustavima. Svi uređaji koji podržavaju komunikaciju putem http protokola mogu konzumirati http servis.

Web-servise možemo gledati kao *web*-aplikacije koje nemaju svoje korisničko sučelje, nego sadrže samo logiku za rad s podacima. *Web*-servisi najčešće su zaduženi za CRUD operacije, odnosno Create, Read, Update i Delete, koje su osnovne operacije obrade podataka. Izrada korisničkog sučelja na klijentskoj je aplikaciji ili sustavu koji će koristiti servis. Uz ovakav način rada dobivamo u potpunosti modularan sustav s boljim mogućnostima skaliranja i možemo programskom logikom obuhvatiti veći broj različitih aplikacija/sustava.

U današnje vrijeme obična *web*-aplikacija nije dovoljna da bi dosegla sve potencijalne korisnike i platforme. Korisnici danas sve više koriste pametne telefone i uređaje, tablete, PC računala, IoT uređaje i ostale uređaje u svakodnevnom životu. No umjesto da se za svaki uređaj posebno piše kompletna aplikacija, uz pomoć *web*-servisa i *web* API-ja možemo napraviti zajedničku komponentu za rad s podacima, a za svaku pojedinu platformu napraviti korisničko sučelje.

7.1. REST

REST (engl. *Representational state transfer*) programski je arhitekturni stil stvoren da usmjeri dizajn i razvoj arhitekture za WWW. REST definira set ograničenja kako bio se trebala ponašati arhitektura sistema predviđena za rad na internetu.

Za web API koji se pridržava smjernica vodilja za rad često kaže da je RESTful. Takvi API-ji tipično su bazirani na http metodama za pristup resursima putem url enkodiranih parametara i koriste JSON ili XML za prijenos podataka.

Primjeri primjene REST servisa mnogobrojne su. Primjenjujemo ih u mobilnim aplikacijama, web-aplikacijama, Internet of Things uređajima, desktop aplikacijama itd. Ukratko, u svim vrstama aplikacija u kojima želimo putem interneta (odnosno s nekog udaljenog mjesta) raditi s podacima.

Npr. Facebook, Instagram, Twitter, Google i druge kompanije koriste REST servise za pristup podacima. Sjetite se da, kad otvorite Facebook ili neku mobilnu aplikaciju, prvo i najvažnije jest da ne funkcioniraju bez interneta. No kad se učitaju podaci, prikazu vam sve obavijesti na početnom zaslonu, a kako se krećete kroz aplikaciju, tako se učitava sve više i više vijesti preko web-servisa.

Na isti način u pozadini mogu raditi web-stranice. Iako većina web-stranica ima svoju bazu podataka, događa se da iz nekih drugih izvora uzimaju druge podatke. Tako recimo možemo preko servisa doći do tečajne liste na određeni dan ili do vremenske prognoze koju bismo integrirali u prikaz korisniku.

Kod IoT uređaja, iako neko konkretno sučelje ne vidimo često, kad ih podesimo, oni mogu slati očitavanja svojih senzora te preko pozadinskog servisa možemo slati naredbe za, na primjer, otključavanje nekih vrata, puštanje zvučnog signala, uključivanje nekog drugog uređaja i slično.

7.2. Serijalizacija i deserijalizacija

Serijalizacija je proces pretvorbe podatkovne strukture ili stanja objekta i format koji se može pohraniti ili poslati, a kasnije rekonstruirati. Kad se rezultirajući niz bitova ponovno pročita, prema uputama formata serijalizacijom, može se koristiti da bismo napravili semantički identičan klon originalnog objekta ili strukture podataka. Taj proces zove se deserijalizacija, kad od dobivenog niza bitova napravimo podatkovnu strukturu ili rekreiramo stanje objekta.

Serijalizacija i deserijalizacija korisni su jer održavaju arhitekturnu neovisnost. To znači da uređaji koji rade na različitim operativnim sustavima i imaju različit *hardware* trebaju pouzdano vršiti proces serijalizacije i deserijalizacije. Arhitekturnom neovisnosti sprečavamo probleme sljedovanja bajtova, memorijske arhitekture ili drukčijeg načina prezentiranja podataka u različitim programskim jezicima. Na primjer, poslužitelj koji ima servis napisan u C#-u i radi serijalizaciju može koristiti aplikaciju napisanu u JavaScriptu za dohvat podataka i deserijalizaciju istih.

Budući da se serijalizacija i deserijalizacija mogu koristiti iz istog zajedničkog koda, moguće je da zajednički kod radi oboje istovremeno. To nam omogućuje da detektiramo razlike među objektima koji se serijaliziraju i njihovih prijašnjih kopija, no istovremeno pruža povratnu informaciju za sljedeću takvu detekciju. Nije nužno raditi prijašnju kopiju jer se razlike mogu otkriti u hodu. Ta tehnika zove se diferencijalno izvršavanje. Korisno je za programe korisničkih sučelja, čiji sadržaj vremenski varira u smislu da se grafički objekti mogu stvarati, uklanjati ili napraviti da rukuju ulaznim događajima, a bez pisanja različitog koda za takve stvari.

Nedostatak serijalizacije jest to što krši nevidljivost apstraktnih tipova, gdje potencijalno eksponira privatne implementacijske detalje. Jednostavne implementacije koje serijaliziraju sve podatke mogu kršiti i principe enkapsulacije.

Da bi se onemogućilo da netko napravi kompatibilan proizvod u smislu servisa, mora se zaštititi način komunikacije između programa i servisa za koji ne želimo da bude javno dostupan. Tvrtke iz tih razloga često u svojim programima rade vlastite serijalizacijske formate. To može značiti da naprave neki naizgled čudan format ili enkriptira-

ju serijalizirane podatke. No svakako je zbog interoperabilnosti potrebno da aplikacije koje koriste servise mogu razumjeti takav format.

Postoji mnogo formata koji se koriste za serijalizaciju i deserijalizaciju. Prvi format koji je imao dosta široku upotrebu dolazi nam još početkom 1980-ih godina od Xerox Network Systems Couriera. Kasnije Sun Microsystems 1987. objavljuje XDR format, koji je otvorenog tipa i standardiziran kao takav. Krajem 1990-ih počele su se pojavljivati popularne alternativne verzije serijalizacijskih protokola, među kojima je počeo i XML. XML u početku nije bio popularan jer nije kompaktan kao prije toga kodiranje pojedinih bajtova. No budući da su internet i prijenos podataka tada već dovoljno napredovali, brzo je stekao popularnost zbog svoje čitljivosti. Početkom 2000-ih XML je često korišten za asinkrone AJAX pozive unutar web-aplikacija.

Za velike količine podataka, razvijeni su specifični binarni serijalizacijski standardi popu HDF-a, netCDF-a i drugih.

7.2.1. XML i JSON

XML i JSON najpopularniji su i najčešće korišteni formati serijalizacije i deserijalizacije. Nećemo ponovno prolaziti taj dio ovdje jer smo to već prošli u poglavlju „4.6 Pretvorba podataka”.

7.3. WEB API

ASP.NET WEB API odličan je alat za izlaganje podataka raznovrsnim uređajima. Web API je *open source framework* za izgradnju RESTful servisa pomoću .NET tehnologije.

ASP.NET WEB API dizajniran je da bi nam olakšao izradu servisa. Krajnje točke rada imaju i automatsku serijalizaciju klase u JSON, bez potrebe za posebnim konfiguracijama. Naravno, serijalizacija može biti podešena za krajnje točke korištenja koje imaju specifične potrebe.

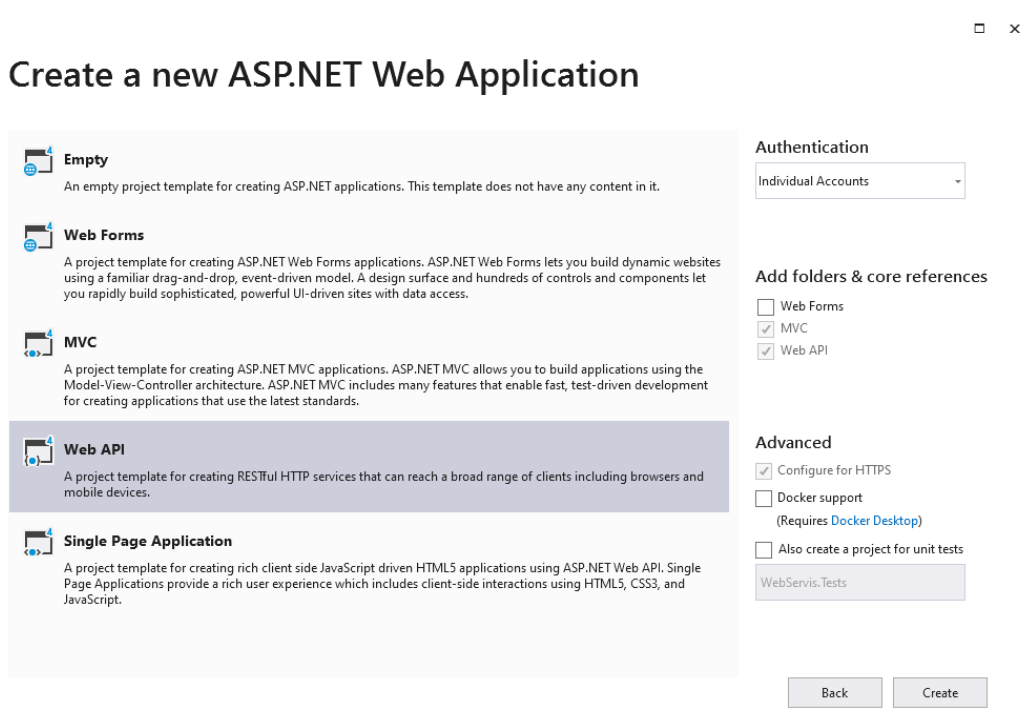
Pomoću Secure API-ja imamo ugrađenu podršku za industrijski standard JSON Web Tokens (JWT), koji nam omogućuje da definiramo pristupna prava.

Dodatno, što se tiče sigurnosti, ASP.NET može se konfigurirati tako da podržava https protokol, što nam daje dodatnu razinu sigurnosti servisa.

Ukratko, korištenjem WEB API-ja i C#-a možemo relativno lagano i brzo graditi moćne servise koje mogu koristiti sve vrste aplikacija koje trebamo.

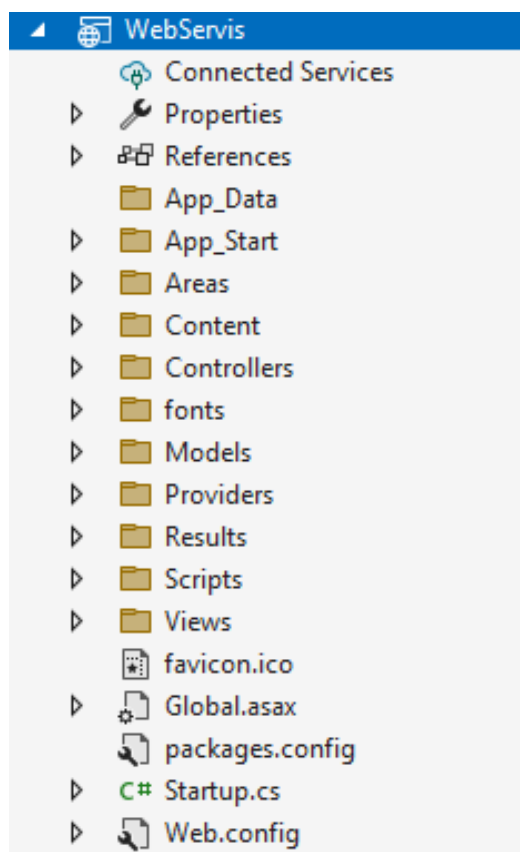
7.3.1. WEB API projekt

Stvaranje WEB API projekta relativno je jednostavno, moramo se referirati na poglavlje 4.7.3 o *web-sučeljima*. Prilikom odabira Web Forms ili MVC predloška imali smo Web API.



Slika 72. Odabir WEB API predloška u ASP.NET Web Application okviru

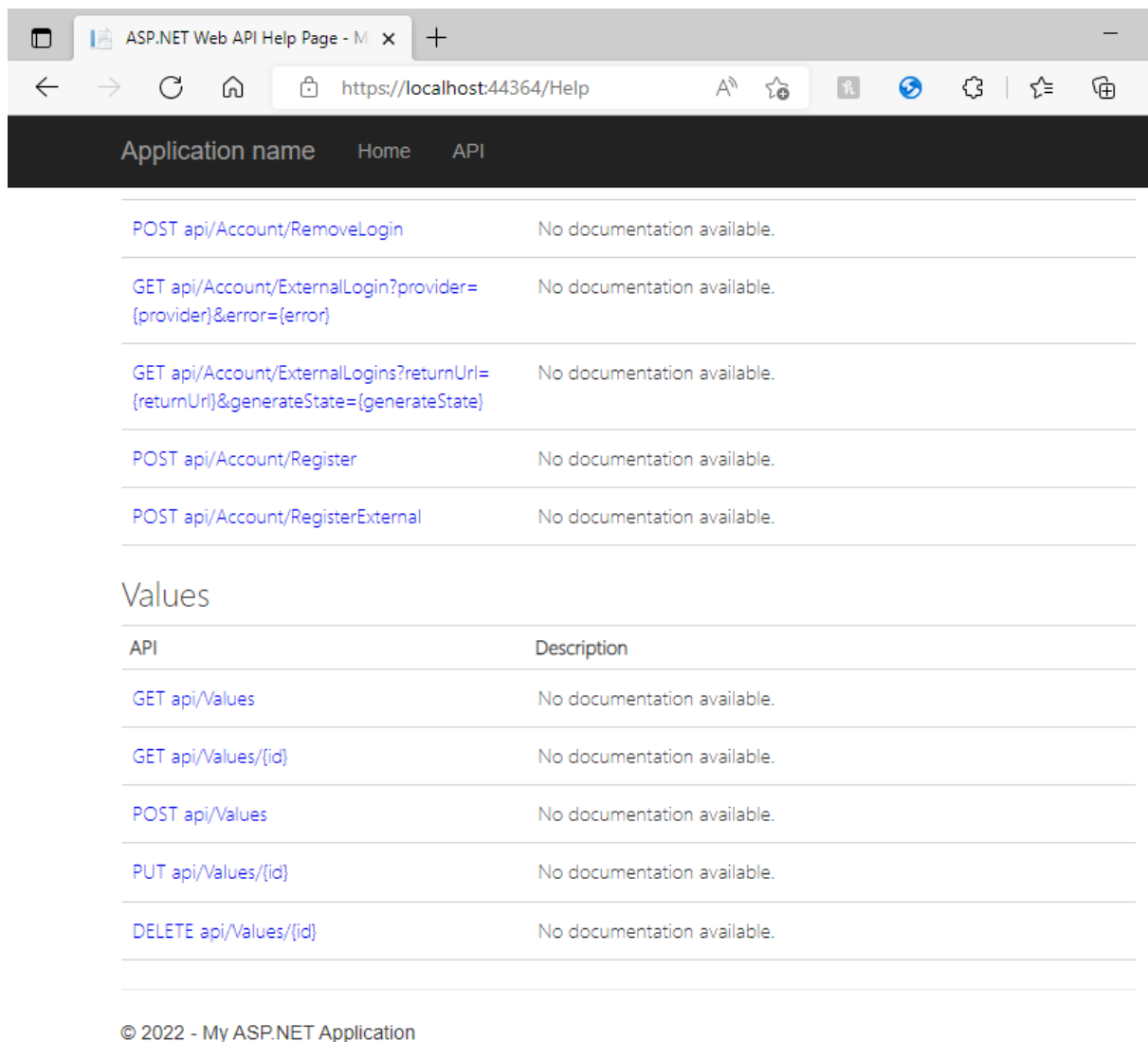
Potvrđivanjem na gumb „Create” Visual Studio automatski će nam generirati projekt i povezati sve potrebne reference. Ako pogledate konstrukciju napravljenog projekta, odnosno mape i datoteke, primijetit ćete da je projekt skoro identičan ASP.NET MVC aplikaciji.



Slika 73. Hijerarhija WEB API projekta

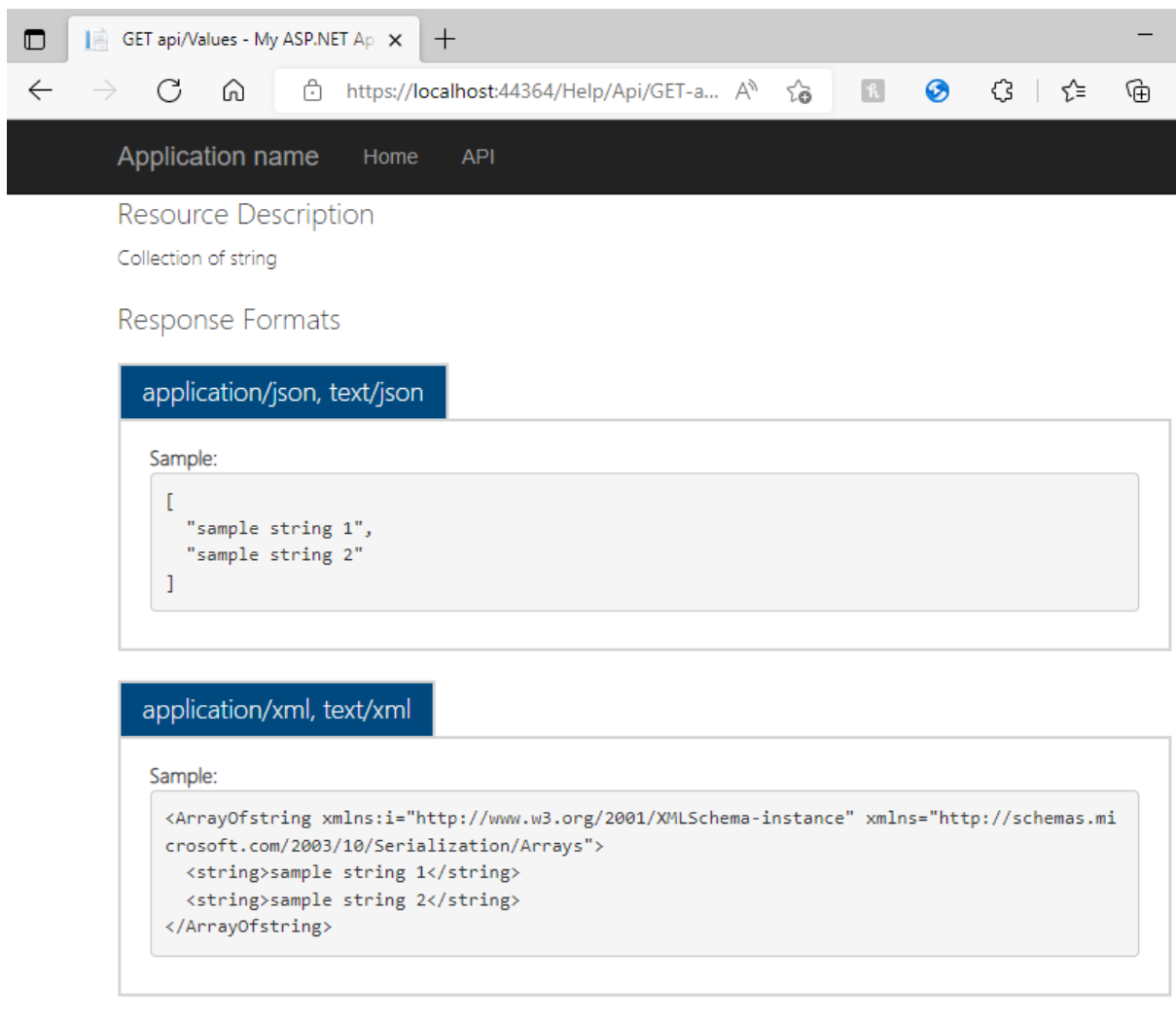
I u pravu ste kad donesete takav zaključak. U pozadini rada ASP.NET WEB API razvijen je na temelju smjernica MVC predloška. U ovom tipu projekta možete paralelno raditi MVC web-aplikaciju, ali i u pozadini servis za usluživanje kako ove aplikacije, tako i bilo koje aplikacije drugog tipa (mobilna, desktop, IoT...) koja bi vam mogla zatrebati ili ako netko treba koristiti servis iz svoje aplikacije.

Ukoliko pokrenete ovu aplikaciju, odnosno projekt, vidjet ćete isti predložak, odnosno sučelje, kao i kad pokrenete MVC aplikaciju. Zasad naizgled ništa nije drukčije, sve dok ne kliknete u aplikaciji u izborniku na API opciju. Klikom na tu opciju otvorit će vam se, kao na slici 74., pomoćna stranica, koja će dokumentirano pokazati kako da koristite pojedinu metodu web-servisa, kojom od http metoda i preko kojeg url-a pristupate.



Slika 74. Popis metoda web-servisa na pomoćnoj stranici

Klikom na pojedini link dobivate opis kako koristiti pojedini poziv. Pa klikom na „GET [api/Values](#)” dobivamo informacije poziva. Informacije poziva sadrže informacije o tome koji su parametri, kako treba izgledati tijelo parametara, ali i koje će nam biti informacije odgovora. Informacije odgovora sadrže opis resursa i kako će nam izgledati izlaz ovisno o formatu u kojem zatražimo serijalizaciju. Prikaz mogućih formata na slici je 75.

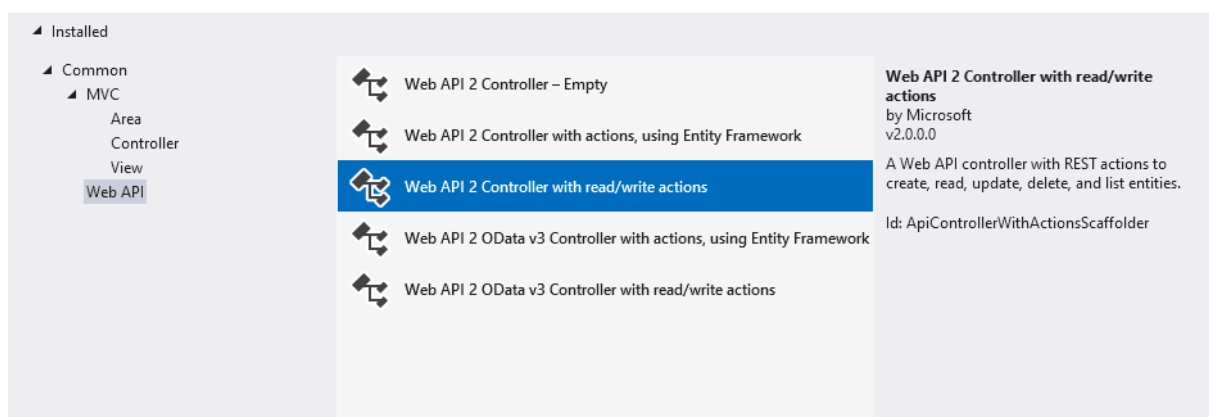


Slika 75. Formati serijalizacije

7.3.2. Kontroler

Dodavanje novog kontrolera bit će nam važno kad želimo dodati novu funkcionalnost. Dodavanje kontrolera radimo tako da kliknemo desnom tipkom miša na mapu Controllers i zatim pod opcijom Add odaberemo na vrhu opciju Controller. Budući da dodajemo Web API Controller, trebamo u izborniku desno odabrati Web API i dobit ćemo kao na slici 76.

Add New Scaffolded Item



Slika 76. Dodavanje Web API kontrolera

Imamo više različitih kontrolera, odnosno vrsta koje odabiremo. Odabir će ovisiti o tome što želimo napraviti s kontrolerom, odnosno koju razinu već pripremljenog koda želimo.

Controller – Empty znači da nemamo praktički ništa, odnosno da u njemu nemamo nijednu akciju.

Controller with actions, using Entity Framework – napraviti će kontroler i osnovne metode za izvođenje osnovnih CRUD operacija (stvaranje, čitanje, izmjena, brisanje i dohvat liste entiteta) korištenjem Entity Framework data contexta (Entity Framework, poglavlje 5.2.2). Podrazumijeva da već imamo DbContext deriviranu klasu i preko nje automatizira sve operacije. Ukoliko imate dobro podešen DbContext, ovdje ne morate ni liniju koda dodatno napisati, već imate servis koji je funkcionalan.

Controller with read/write actions – radi slično kao gore navedeno, samo je na nama da implementiramo točno tok CRUD operacija. Sve operacije već su definirane kao akcije, no na nama je da ih implementiramo i prilagodimo.

OData v3 Controller – slično kao i gore navedena dva, samo koristi OData protokol.

Kad pritisnemo add, moramo samo navesti ime kontrolera. Svi kontroleri u MVC projektu, radi raspoznavanja, imaju u imenu Controller. Tako da, ako želimo napraviti RW

kontroler, ime kontrolera bit će RWController.

U primjeru ćemo raditi s Controller with read/write actions.

7.3.3. Akcije

Akcije, isto kao kod MVC projekta, predstavljaju metode koje se mapiraju direktno na http akcije. Ako pogledamo prethodno napravljen kontroler, on će imati sljedeći kod:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebServis.Controllers
{
    public class RWController : ApiController
    {
        // GET: api/RW
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET: api/RW/5
        public string Get(int id)
        {
            return "value";
        }

        // POST: api/RW
        public void Post([FromBody]string value)
        {
        }
    }
}
```

```

// PUT: api/RW/5
public void Put(int id, [FromBody]string value)
{
}

// DELETE: api/RW/5
public void Delete(int id)
{
}
}
}

```

Visual Studio automatski nam generira i komentar da bismo znali koja se akcija veže na koju http metodu, čak i pokazuje kojim linkom pristupamo željenoj metodi.

Budući da ovdje već imamo sve http akcije mapirane, ukoliko želimo drukčije mapiranje, odnosno akcije koje rade dohvat drugih podataka, morat ćemo definirati drugi kontroler. Slično kao u MVC aplikacijama, ovdje će se WEB API pridržavati standardnog izgleda linkova za rukovanje API-jem, koji ima sljedeći uzorak:

```
{domena}/api/{web_api_controler}/{parametar}
```

Domena je u našem slučaju localhost s brojem porta, no u produkciji to bi bili domena ili poslužitelj na kojima se nalazi naš servis. Možemo reći da je „api” tu da odvoji sadržaj web stranice od servisa. Parametri su opcionalni i prosljeđujemo ih ukoliko ih neka od metoda treba. Ukoliko imamo više parametara, moramo ih adekvatno nabrojati u parametarsku listu metode preko koje im želimo pristupiti. Imena metoda ne možemo mijenjati ako želimo ostaviti da se odnose na http metode. Ovdje ne smijemo napraviti standardno preopterećivanje metoda . Npr. u sljedećem kodu:

```

public string Get(int id)
{
    return "value";
}
public string Get(string id)
{
    return id;
}

```

Iako je po pravilima pisanja C# koda ovo sasvim dozvoljeno, pokušaj izvršavanja zahtjeva, neovisno o tome proslijedimo li broj ili niz znakova, rezultat će time da nam servis pošalje poruku greške:

```
<Error>
<Message>An error has occurred.</Message>
<ExceptionMessage>Multiple actions were found that match the
request: Get on type WebServis.Controllers.RWController Get on
type WebServis.Controllers.RWController</ExceptionMessage>
<ExceptionType>System.InvalidOperationException</ExceptionType>
<StackTrace> at System.Web.Http.Controllers.ApiControllerAc-
tionSelector.ActionSelectorCacheItem.SelectAction(HttpControl-
lerContext controllerContext) at System.Web.Http.ApiController.
ExecuteAsync(HttpControllerContext controllerContext, Cancellat-
ionToken cancellationToken) at System.Web.Http.Dispatcher.Http-
ControllerDispatcher.<SendAsync>d__15.MoveNext()</StackTrace>
</Error>
```

Ukoliko izmijenimo i promijenimo metode get s parametrima na sljedeći način:

```
public string Get(int id)
{
    return "value";
}
public string Get(string id, string ime)
{
    return id;
}
```

Moći ćemo pristupiti i drugoj akciji, no na način da na drugi način dobijemo parametar ime. U tom slučaju poziv te akcije bit će sljedeći:

```
GET api/RW/{id}?ime={ime}
```

Sve akcije koje dodamo u kontroler bit će nam vidljive i dokumentirane na pomoćnoj stranici jednom kad pokrenemo projekt.

RW

API	Description
GET api/RW	No documentation available.
GET api/RW/{id}	No documentation available.
GET api/RW/{id}?ime={ime}	No documentation available.
POST api/RW	No documentation available.
PUT api/RW/{id}	No documentation available.
DELETE api/RW/{id}	No documentation available.

Slika 77. Dodane metode novog kontrolera

7.4. Mapiranje složenih podataka

Složene podatke u WEB API projektu mapiramo slično kao i u MVC aplikacijama, preko modela. Receptura je jednostavna, pa idemo dodati novu klasu u model, koja će nam služiti za demonstraciju.

Na Models mapu kliknemo desnom tipkom miša, odaberemo opciju Add, pa zatim na dnu Class. Odaberemo ime (u primjeru dodao sam klasu osoba) i kliknemo Add. Zatim ćemo u klasu dodati sljedeća svojstva:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
  
namespace WebServis.Models  
{
```



```

public class Osoba
{
    public int Id { get; set; }
    public string Ime { get; set; }
    public string Prezime { get; set; }
    public int Godina_rodjenja { get; set; }
}

```

Ovu klasu koristit ćemo za rad s podacima u novom kontroleru.

7.4.1. Mapiranje HTTP metoda

Http metode nema potrebe posebno mapirati jer su one već mapirane s obzirom na ime metode unutar Web API kontrolera.

No idemo napraviti osnovne CRUD operacije s gore navedenom klasom. Dodat ćemo novi WEB API Controller with read/write actions imena OsobaController i modificirati ga na sljedeći način:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using WebServis.Models;

```

```

namespace WebServis.Controllers
{
    public class OsobaController : ApiController
    {
        private List<Osoba> _list= new List<Osoba>
        {
            new Osoba {Id=1,Ime="Josip",Prezime="Jelačić",Godina_rodjenja=1801},

```

```

    new Osoba {Id=2,Ime="Marko",Prezime="Marulić",Godina_rodjenja=1450},
    new Osoba {Id=3,Ime="Nikola",Prezime="Tesla",Godina_rodjenja=1856}
};

// GET: api/Osoba
public IEnumerable<Osoba> Get()
{
    return _list;
}

// GET: api/Osoba/5
public Osoba Get(int id)
{
    Osoba o = _list.Where(x=>x.Id==id).FirstOrDefault();
    return o;
}

// POST: api/Osoba
public void Post([FromBody]Osoba value)
{
    _list.Add(value);
}

// PUT: api/Osoba/5
public void Put(int id, [FromBody]Osoba value)
{
    _list[id] = value;
}

// DELETE: api/Osoba/5
public void Delete(int id)
{
    var o = _list.Where(x => x.Id == id).FirstOrDefault();
    if (o == null) return;
    _list.Remove(o);
}
}
}

```

Ako sad pogledamo kako će nam izgledati pozivi preko helpera, dobit ćemo rezultat kao na slici 78.

Osoba

API	Description
GET api/Osoba	No documentation available.
GET api/Osoba/{id}	No documentation available.
POST api/Osoba	No documentation available.
PUT api/Osoba/{id}	No documentation available.
DELETE api/Osoba/{id}	No documentation available.

Slika 78. Rezultat mapiranih metoda

Automatski su nam definirane sve potrebne metode. Ako otvorimo neku od metoda, vidjet ćemo detalje kako pristupiti tim metodama, odnosno kojeg će podatkovnog tipa biti i kako će nam izgledati rezultat. Ako trebamo nekome eksponirati servis, da napravi klase za brzo mapiranje prema izlazu servisa, može to napraviti. Čak i ukoliko dođe do greške, opis metode pokazat će nam kako bi greška izgledala. Ukoliko odaberemo POST api/Osoba, dobit ćemo detalje kao na slici 79.

POST api/Osoba

Request Information

URI Parameters

None.

Body Parameters

[Osoba](#)

Name	Description	Type	Additional information
Id		integer	None.
Ime		string	None.
Prezime		string	None.
Godina_rodjenja		integer	None.

Request Formats

application/json, text/json

Sample:

```
{
  "Id": 1,
  "Ime": "sample string 2",
  "Prezime": "sample string 3",
  "Godina_rodjenja": 4
}
```

application/xml, text/xml

Sample:

```
<Osoba xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/WebServis.Model
s">
  <Godina_rodjenja>4</Godina_rodjenja>
  <Id>1</Id>
  <Ime>sample string 2</Ime>
  <Prezime>sample string 3</Prezime>
</Osoba>
```

application/x-www-form-urlencoded

Sample:

Failed to generate the sample for media type 'application/x-www-form-urlencoded'. Cannot use formatter 'jQueryMvcFormUrlEncodedFormatter' to write type 'Osoba'.

Response Information

Resource Description

None.

Slika 79. Detalji POST api/Osoba

7.4.2. Čitanje i pisanje u bazu podataka

Čitanje i pisanje u bazu podataka možemo napraviti na nekoliko načina. Možemo napraviti čitanje i pisanje u datoteku, kako smo napravili u poglavlju „4.4 Rad s datotekama” ili u bazu podataka kao u poglavlju „5.2 ORM i ADO.NET upotrebom EF”, bilo preko klasičnog direktnog pristupa ili entity frameworka.

Kod bilo kojeg pristupa osim EF-a najlakši je način rada taj da kao i u prošlom primjeru definiramo WEB API Controller with read/write actions pa modificiramo potrebne metode da rade s odabranim načinom spremanja.

Ukoliko radimo s Entity Frameworkom, uvelike nam je pojednostavljena situacija. Prilikom definiranja novog kontrolera odaberemo opciju WEB API Controller with actions, using Entity Framework. Nakon klika na *add*, odaberemo model klasu i *data context* klasu koje odgovaraju mapiranju iz EF klase. Nakon toga, s obzirom na odabrane klase, Visual Studio ponudit će nam ime kontrolera, koje može ostati kakvo jest. Pritiskom na tipku *add* Visual Studio sam će odraditi potreban posao i generirati nam sljedeći kod:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using System.Web.Http.Description;
using WebService;

namespace WebService.Controllers
{
    public class OsobasController : ApiController
    {
        private SkriptaEntities db = new SkriptaEntities();
    }
}
```

```

// GET: api/Osobas
public IQueryable<Osoba> GetOsoba()
{
    return db.Osoba;
}

// GET: api/Osobas/5
[ResponseType(typeof(Osoba))]
public IHttpActionResult GetOsoba(int id)
{
    Osoba = db.Osoba.Find(id);
    if (osoba == null)
    {
        return NotFound();
    }

    return Ok(osoba);
}

// PUT: api/Osobas/5
[ResponseType(typeof(void))]
public IHttpActionResult PutOsoba(int id, Osoba osoba)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (id != osoba.IDOsoba)
    {
        return BadRequest();
    }

    db.Entry(osoba).State = EntityState.Modified;

    try
    {
        db.SaveChanges();
    }
}

```

```

        catch (DbUpdateConcurrencyException)
        {
            if (!OsobaExists(id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return StatusCode(HttpStatusCode.NoContent);
    }

    // POST: api/Osobas
    [ResponseType(typeof(Osoba))]
    public IHttpActionResult PostOsoba(Osoba osoba)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        db.Osoba.Add(osoba);
        db.SaveChanges();

        return CreatedAtRoute("DefaultApi", new { id = osoba.IDOsoba }, osoba);
    }

    // DELETE: api/Osobas/5
    [ResponseType(typeof(Osoba))]
    public IHttpActionResult DeleteOsoba(int id)
    {
        Osoba = db.Osoba.Find(id);
        if (osoba == null)
        {
            return NotFound();
        }
    }

```

```

        db.Osoba.Remove(osoba);
        db.SaveChanges();

        return Ok(osoba);
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            db.Dispose();
        }
        base.Dispose(disposing);
    }

    private bool OsobaExists(int id)
    {
        return db.Osoba.Count(e => e.IDOsoba == id) > 0;
    }
}

```

Koristio sam bazu podataka iz 2. poglavlja – rada sa SQL serverom. Koristio sam Osoba tablicu za rad.

Kao što vidimo, Visual Studio podesio nam je sve što trebamo, tako da, ako ne treba dodavati neke kompliciranije pretrage ili slično, imamo sve što nam treba za rad. Mapiranje http metoda isto je kao i prije, odnosno možemo provjeriti preko api helpera.

7.5. Povezivanje s aplikacijom

Povezivanje možemo napraviti preko bilo koje vrste aplikacije. Kako što smo rekli, RESTful servisima može se pristupati preko bilo koje vrste aplikacije ili platforme, ono što je bitno jest da uređaj, odnosno program, vidi servis, odnosno da ima pristup servisu koji trebamo. Za potrebe kontrole pristupa opet ćemo koristiti najjednostavniju konzolnu aplikaciju da opišemo korake rada. Za početak morate imati pokrenut servis

u pozadini. Najbolje je da projekt koji ste radili za servis pokrenete s „Start without debugging”. Važno nam je da ostane pokrenut u pozadini jer ćemo ga trebati ga testirati, odnosno bitno nam je da znamo točan uri servisa. Kod izgleda ovako:

```
using System;
using System.Collections.Generic;
using System.Net.Http.Json;

namespace Aplikacija
{
    class Program
    {
        // http klijent instance za rad upita
        private static readonly HttpClient client = new HttpClient();
        static async Task Main()
        {
            var stringTask = client.GetStringAsync("https://localhost:44364/api/
                                                    osoba");
            var poruka = await stringTask; // čekanje odgovora i spremanje rezultata
            u                                     varijablu
            Console.WriteLine(poruka); // ispis odgovora na konzolu
        }
    }
}
```

Ispis će biti sljedeće:

```
{"Id":1,"Ime":"Josip","Prezime":"Jelacic","Godina_rodjenja":1801},{"Id":2,"Ime":"Marko","Prezime":"Marulic","Godina_rodjenja":1450},{"Id":3,"Ime":"Nikola","Prezime":"Tesla","Godina_rodjenja":1856}
```

Ako malo analiziramo kod, vidjet ćemo da teško može biti jednostavnije od ovoga. Moramo imati http klijent instancu preko koje se spajamo na naš web-servis. Može naravno biti član klase ili kao varijabla unutar metode. U http klijentu pozivamo metodu GetStringAsync s parametrom url-a kojemu pristupamo. Napomena, nećete uvijek imati isti port u url-u, ovo je tijekom testiranja bio port koji je VS dao u trenutku testiranja aplikacije. No svakako morate samo proslijediti url kojem pristupate. Nakon što se napravili zahtjev, budući da se radi o asinkronom pozivu, sačekat ćemo

odgovor poziva. Odgovor nam dolazi u json formatu, s kojim onda možemo raditi kako želimo. U ovom slučaju samo ispisujemo kompletan pristigli rezultat, no inače bismo mogli deserijalizirati podatke, spremiti u neku listu i onda raditi s njima kako želimo.

HttpClient klasa ovdje je najbitniji faktor, preko nje radimo sve potrebne zahtjeve prema bilo kojem servisu, no možemo napraviti i zahtjev prema nekoj *web*-stranici ako želimo kao rezultat dobiti kompletan html. HttpClient klasa ima sve http metode koje nam trebaju, pa samo odabirom metode dobivamo željenu http metodu. Neke od metoda jesu:

- HttpClient – konstruktor kojih ima više
- CancelPendingRequest – otkazivanje trenutnog zahtjeva
- GetAsync – http GET za dobivanje odgovora proslijeđenim uri-jem
- GetStringAsync – http GET metoda za dohvat podataka proslijeđenim uri-jem
- GetStreamAsync – http GET metoda za dohvat streama proslijeđenim uri-jem
- GetByteArrayAsync – http GET metoda za dohvat polja bajtova proslijeđenim uri-jem
- DeleteAsync – http DELETE metoda za proslijeđeni uri
- PostAsync – http Post metoda za proslijeđeni uri
- ...

Svaki programski jezik ima različit način rada s http zahtjevima, odnosno s REST servisima. Danas je rad s REST servisima maksimalno pojednostavljen, pa bi tako i JavaScript poziv na ovaj servis s Fetch API-jem izgledao ovako:

```
const userAction = async () => {  
  const response = await fetch('https://localhost:44364/api/osoba');  
  const myJson = await response.json(); // izvuci JSON iz http respon  
  // napraviti nešto s myJson  
}
```

Za testiranje REST API-ja postojе još servisi poput Postmana, no za rad s njim potrebno je da servis bude na nekom web-poslužitelju, svoj lokalni servis ne možete testirati takvim *online* alatima.

8. POGLAVLJE:

Automatizirano testiranje

U OVOM POGLAVLJU NAUČIT ĆETE:

- kako raditi automatsko testiranje koda
- definirati pravila rada
- pokrenuti testove
- pregledati rezultate.

8.1. Testiranje koda

Testiranje koda provjera je radi li kod kao što se očekuje tako da stvaramo i izvodimo testove jedinica (*unit test*). Zove se unit test (test jedinice) jer raščlanjujemo funkcionalnost programa u diskretne dijelove (jedinice) koje možemo individualno testirati. Visual Studio Test Explorer omogućuje nam fleksibilan i efikasan način da izvodimo testove i da pregledamo njihov rezultat u Visual Studiu. Visual Studio instalira Microsoft *unit testing framework* prilikom instalacije. Ovo nam omogućuje da u bilo kojem trenutku pokrenemo testiranje naših aplikacija. Inače, VS Enterprise ima takozvani Live Unit Test, koji se tijekom pisanja koda pokreće automatski u pozadini te detektira kako promjene na kodu utječu na testove.

Testiranje ima dobar utjecaj na kvalitetu koda kad je integralni dio razvoja programa. Čim napišete funkciju ili neki drugi blok koda, stvorite unit test da verificira ponašanje koda sukladno standardima, ograničenjima, neispravnim slučajevima unosa podataka ili provjere da se ne rade neke pretpostavke unutar koda. S *test driven develop-*

mentom (razvoj potican testovima) možete stvarati *unit testove* prije nego što napišete kod, pa koristite testove za dizajn dokumentacije i funkcionalne specifikacije.

Test eksplorer može izvršavati testove treće strane i *open source unit test frameworke* koji imaju implementirano Test Explorer *add-on* sučelje. Možete dodati koji god hoćete *framework* preko Visual Studio Extension Managera i Visual Studio Galleryja.

Testni projekti mogu se brzo generirati, kao i test metode iz koda. Također, mogu se ručno napraviti dodatni testovi po potrebi. Kad koristimo IntelliTest da pregledamo .NET kod, možemo generirati testne podatke za testove. Za svaku naredbu u kodu generira se testni ulaz koji će izvršiti naredbu.

Najbolje je pokazati kako unit testovi rade na primjeru, pa ćemo napraviti testove na konzolnoj aplikaciji. Napravimo konzolnu aplikaciju imena „TestiranaAplikacija” i u Program.cs dodamo sljedeći kod:

```
using System;
```

```
namespace BankovniRacun
```

```
{
```

```
    /// <summary>
```

```
    /// Bankovni racun demo klasa.
```

```
    /// </summary>
```

```
public class BankovniRacun
```

```
{
```

```
    private readonly string _vlasnikracuna;
```

```
    private double _bilanca;
```

```
    private BankovniRacun() {}
```

```
    public BankovniRacun(string imeKlijenta, double bilanca)
```

```
{
```

```
        _vlasnikracuna = imeKlijenta;
```

```
        _bilanca = bilanca;
```

```
}
```

```

public string ImeKlijenta
{
    get { return _vlasnikracuna; }
}

public double Bilanca
{
    get { return _bilanca; }
}

public void Debit(double iznos)
{
    if (iznos > _bilanca)
    {
        throw new ArgumentOutOfRangeException("iznos");
    }

    if (iznos < 0)
    {
        throw new ArgumentOutOfRangeException("iznos");
    }

    _bilanca += iznos; // namjerno netočan kod!!
}

public void Credit(double iznos)
{
    if (iznos < 0)
    {
        throw new ArgumentOutOfRangeException("iznos");
    }

    _bilanca += iznos;
}

public static void Main()
{
    BankovniRacun br = new BankovniRacun("Nikola Tesla", 300.13);
}

```

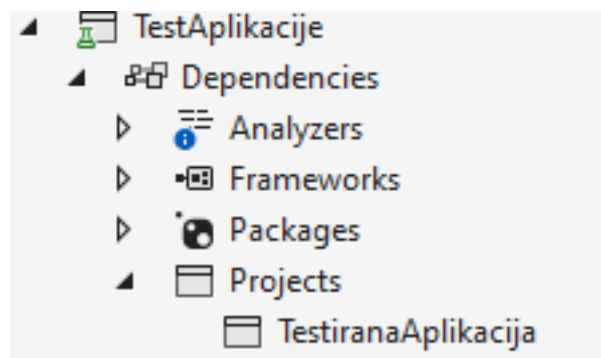
```

        br.Credit(5.77);
        br.Debit(11.22);
        Console.WriteLine("trenutna bilanca je {0}kn", br.Bilanca);
    }
}
}

```

Sad imamo projekt na kojem možemo raditi testove.

Sad ćemo dodati „MSTest Test Project” u Solution i imenovat ćemo ga „TestAplikacije”. Ukoliko koristite VS 2022, template se zove „Unit Test Project”. Kad je projekt za testiranje napravljen, moramo napraviti referencu na testirani projekt u Dependencies. Kliknemo desnom tipkom miša na Dependencies i odaberemo „Add Project Reference”. Odaberemo TestiranaAplikacija i kliknemo OK. Provjeru napravite tako da proširite Dependencies, zatim Project opciju i provjerite postoji li traženi projekt. Rezultat će izgledati kao na slici 80.



Slika 80. Referenca na testirani projekt

Sad ćemo napisati testove. Možemo to napraviti dodavanjem nove datoteke ili možemo preurediti već napravljenu UnitTest1.cs datoteku. Ovdje ćemo preimenovati UnitTest1 u TestiranjeRacuna, a kad preimenujemo datoteku, VS će nam ponuditi da automatski preimenuje klasu, što možemo potvrditi klikom na OK. Kod će sad izgledati ovako:

```

namespace TestAplikacije
{
    [TestClass]
    public class TestiranjeRacuna
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}

```

Minimalni zahtjevi za test klasu jesu sljedeći:

- [TestClass] atribut obavezan je na bilo kojoj klasi koja sadrži unit test metodu koju želimo pokrenuti u test exploreru
- svaka test metoda za koju želimo da je test explorer prepozna mora imati [TestMethod] atribut.

Možemo imati druge klase u unit test projektu, koje ne trebaju imati TestClass atribut i možemo imati druge metode u testnim klasama, koje ne moraju imati TestMethod atribut. Te klase i metode mogu se pozivati iz testnih klasa i metoda.

Postoje bar tri slučaja ponašanja koje moramo testirati:

- da metoda baci ArgumentOutOfRangeException kad je debit iznos veći od balance
- da metoda baci ArgumentOutOfRangeException ako je iznos debita manji od nule
- ako je debitni iznos validan, metoda oduzima debitni iznos iz balance računa.

Napravit ćemo test za provjeru povlačenja iznosa s računa. Dodat ćemo novu metodu u klasu TestiranjeRacuna, koja će sad izgledati ovako:


```

namespace TestAplikacije
{
    [TestClass]
    public class TestiranjeRacuna
    {
        [TestMethod]
        public void TestMethod1()
        {
        }

        [TestMethod]
        public void Debit_sValidnimIznosom_AzuriraBilancu()
        {
            // Priprema
            double pocetnaBilanca = 11.99;
            double debitIznos = 4.55;
            double ocekivano = 7.44;
            BankovniRacun.BankovniRacun racun = new BankovniRacun.BankovniRa-
            cun("Nikola Tesla", pocetnaBilanca);

            // Rad
            racun.Debit(debitIznos);

            // Potvrda
            double stvarno = racun.Bilanca;
            Assert.AreEqual(ocekivano, stvarno, 0.001, "Racun nije debitiran ispravno");
        }
    }
}

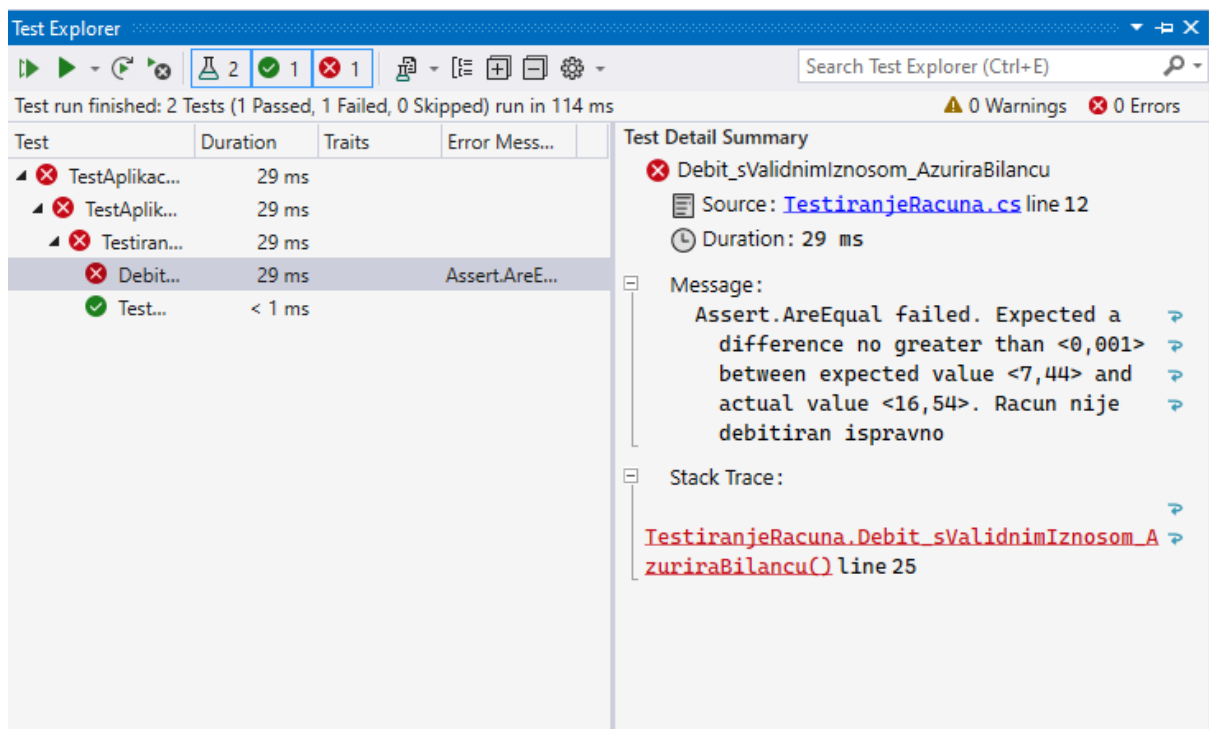
```

Metoda je direktna, radimo novu instancu BankovniRacun s početnom bilancom, a zatim povlačimo s računa određeni iznos. Koristimo Assert.AreEqual metodu da bismo verificirali očekivanu krajnju bilancu. Metode poput Assert.AreEqual, Assert.IsTrue i druge često se koriste u *unit* testovima.

Test metoda mora imati sljedeće preduvjete:

- ima [TestMethod] atribut
- vraća void
- ne može imati parametre.

Da bismo pokrenuli testove, prvo moramo *buildati* projekt. Test Explorer, ukoliko ga već nemate u jednoj alatnoj traci, možete otvoriti tako da kliknete u glavnom izborniku view pa ćete u popisu naći „Test Explorer”. Sad možemo pokrenuti testove klikom na zeleni trokutić ili u glavnom izborniku klikom na Test pa „Run All”. Nakon pokretanja dobit ćemo u Test Exploreru ono što je prikazano na slici 81.

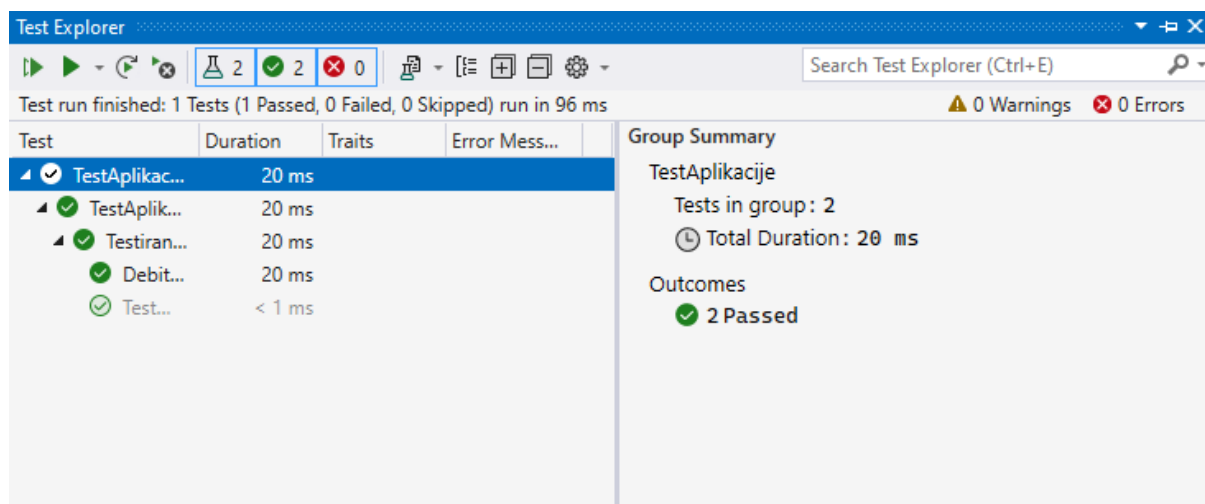


Slika 81. Test Explorer nakon neuspjelih testova

Rezultat testa sadrži poruku koja opisuje grešku koja je nastala. AreEqual metoda pokazuje što je očekivano i što je stvarno primljeno. Očekivano je da se bilanca smanji, no ona se povećala za iznos koji smo htjeli povući s računa.

Unit test pokazao je da postoji greška, sad trebamo naći kod koji je neispravan (a namjerno smo ga napravili u primjeru tako) i ispraviti ga. Znači liniju „_bilanca += iznos” treba zamjeniti linijom „_bilanca -= iznos”. Nakon toga opet možemo pokrenu-

ti testove i dobit ćemo rezultat kao na slici 82.



Slika 82. Uspješni testovi

Idemo sada još stvoriti metodu za provjeru je li debitni iznos manji od nule:

[TestMethod]

```
public void Debit_KadJelznosManjiOdNule_Trebalzbaciti_ArgumentOutOfRangeException()
{
    double pocetnaBilanca = 11.99;
    double debitIznos = -100.00;
    BankovniRacun.BankovniRacun racun = new BankovniRacun.Ban
    kovniRacun("Nikola Tesla", pocetnaBilanca);

    // Rad i potvrda
    Assert.ThrowsException<System.ArgumentOutOfRangeException>(() =>
    racun.Debit(debitIznos));
}
```

Koristimo ThrowException metodu asserta da budemo sigurni da je bačena ispravna iznimka. Ukoliko nije bačena ispravna iznimka, kako je definirana u ThrowsException, potvrda će biti neispravna, odnosno test neće uspjeti. Ako modificiramo metodu testa tako da baci općenitiju iznimku poput ApplicationException kad je debitni iznos manji od nule, test će se ispravno ponašati, odnosno neće prolaziti.

Da bismo provjerili slučajeve kad je povučena suma veća od bilance računa, možemo napraviti sljedeće:

- stvoriti novu testnu metodu
Debit_KadJelznosVeciNegoBilanca_Trebalzbaciti_ArgumentOutOfRangeException
- kopirati tijelo metode
Debit_KadJelznosManjiOdNule_Trebalzbaciti_ArgumentOutOfRangeException
- postaviti debitIznos na broj koji je veći od bilance.

Metoda koja se testira može se dalje poboljšati. S trenutnom implementacijom nemamo nikakav način da znamo koji je uvjet doveo do iznimke (iznos veći od bilance ili iznos manji od 0). Samo znamo da je bačeno `ArgumentOutOfRangeException` negdje u metodi. Bilo bi bolje kad bismo uspjeli znati koji je uvjet u `BankovniRacun.Debit` uzrok iznimke, pa da bismo mogli biti sigurni da je naša metoda ispravno testirala argumente.

Ako pogledamo metodu koju testiramo, možemo vidjeti da oba uvjeta bacanja iznimke koriste istu konstrukciju `ArgumentOutOfRangeException`, odnosno uzimaju samo parametar. Postoje konstruktori koji će u konstrukciji primiti više poruka, pa nam omogućuju da napravimo vlastite poruke tijekom bacanja iznimke. Te poruke možemo izmijeniti. Recimo da u `BankovniRacun` promijenimo unutar `Debit` metode bacanje iznimke na sljedeće:

```
if (iznos > _bilanca)
{
    throw new ArgumentOutOfRangeException("iznos", iznos, "Iznos
    povlačenja je veći od bilance!");
}
if (iznos < 0)
{
    throw new ArgumentOutOfRangeException("iznos", iznos, "iznos ne
    može biti negativan!");
}
```

Sad možemo promijeniti metodu testiranja da izbacimo poziv `Assert.ThrowsException` tako da za poziv unutar testa `Debit` metodi umotamo u `try/catch` blok, koji hvata

specifičnu iznimku i provjerava je s obzirom na poruku.

Test `Debit_KadJelznosVeciNegoBilanca_Trebalzbaciti_ArgumentOutOfRangeException` sad možemo prepraviti da izgleda ovako:

[TestMethod]

```
public void Debit_KadJelznosManjiOdNule_Trebalzbaciti_ArgumentOutOfRangeException()
{
    double pocetnaBilanca = 11.99;
    double debitIznos = -100.00;
    BankovniRacun.BankovniRacun racun = new BankovniRacun.Ban
    kovniRacun("Nikola Tesla", pocetnaBilanca);

    // Rad i potvrda
    try
    {
        racun.Debit(debitIznos);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        StringAssert.Contains(e.Message, "Iznos povlačenja je veći
        od bilance!");
    }
}
```

Trenutno testna metoda ne obrađuje sve slučajeve koje bi trebala. Ako metoda koju testiramo ne uspije baciti `ArgumentOutOfRangeException` kad je iznos veći od bilance ili manji od nule, test prolazi. To nije dobro jer želimo da testna metoda ne uspije ako nijedna iznimka nije bačena. Ovo je greška u testnoj metodi. Da bismo to riješili, trebamo dodati `Assert.Fail` na kraju testne metode da odradi slučajeve gdje se iznimka nije bacila.

Ponovno pokretanje testova pokazuje da test sada ne prolazi ako je korektna iznimka uhvaćena. *Catch blok* hvata iznimku, ali metoda se nastavlja izvoditi i ne uspijeva jer je `Assert.Fail` potvrđen. Da bismo riješili ovaj problem, dodat ćemo `return` naredbu nakon `StringAssert` u *catch bloku*. Povratak iz testa potvrđuje da smo riješili problem. Finalna verzija metode testa `Debit_KadJelznosVeciNegoBilanca_Trebalzbaciti_Argu-`

mentOutOfRange treba izgledati na sljedeći način:

```
[TestMethod]
public void Debit_KadJelznosManjiOdNule_Trebalzbaciti_ArgumentOutOfRange()
{
    double pocetnaBilanca = 11.99;
    double debitIznos = -100.00;
    BankovniRacun.BankovniRacun racun = new BankovniRacun.Ban
    kovniRacun("Nikola Tesla", pocetnaBilanca);

    // Rad i potvrda
    try
    {
        racun.Debit(debitIznos);
    }
    catch (System.ArgumentOutOfRangeException e)
    {
        StringAssert.Contains(e.Message, "Iznos povlačenja veći je
        od bilance!");
        return;
    }
    Assert.Fail("Očekivana iznimka nije se digla!");
}
```

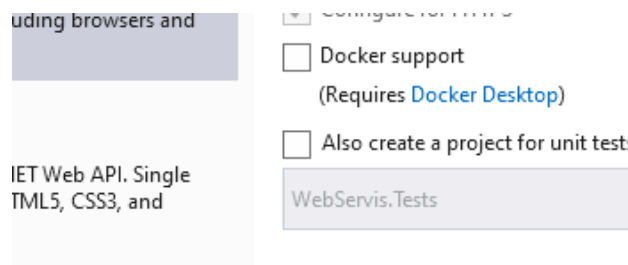
Poboljšanja na testnom kodu dovela su do robusnije i informativnije testne metode. No najvažnije je da su unaprijedile kod koji se testira, što i jest bit ovih testova.

Tijekom rada, pogotovo na većim projektima, rijetko ćemo imati mogućnost testiranja koda koji radimo i događat će se da radimo gomilu koda prije nego što uopće pokrenemo projekt. Zahvaljujući unit testovima ispravnost napisanog koda možemo odmah testirati i ispravljati bez da program u njegovoj finalnoj verziji uopće pokrenemo. Najvažnije nam je da smanjuje problem kad testiramo dio koda za koji bi u aplikaciji trebalo dosta vremena da mu pristupimo s korisničke strane.

8.2. Unit test u MVC aplikacijama

Već smo prilikom spominjanja MVC aplikacija rekli da su pogodne za *unit* testove, odnosno da za razliku od *web-formi* s MVC aplikacijama možemo lagano raditi testove.

Prvo svakako treba napomenuti da kod većine projekata pri stvaranju projekta možemo odabrati da se napravi i projekt za testiranje. Opcija se kod ASP.NET Web Application nalazi u doljnjem desnom uglu.



Slika 83. Dodavanje Unit Test projekta uz MVC aplikaciju

Ako projekt nema ovu mogućnost, uvijek se može zasebno dodati testni projekt, odnosno definirati projekt, dodati referenca na projekt i krenuti s izradom testova.

Svakako, kojom god da metodom napravimo projekt za testiranje, kako dodajemo nove kontrolere i nove dijelove aplikacije, testovi se neće automatski izrađivati. Iako će automatski generirani testni projekt napraviti reference i testove na predefinirane komponente, za naknadno dodane komponente mi moramo definirati nove testove, bilo preko novih klasa ili dodavanjem testova u istu klasu kao i prvotni testovi.

Glavno i temeljno testiranje u MVC aplikacijama jest testiranje rada kontrolera. U kontroleru možemo testirati nekoliko slučajeva:

- testiranje koji se pogled vraća kao rezultat akcije kontrolera
- testiranje je li vraćen pogled ili samo neki podatak u slučaju premošćivanja ili preopterećivanja akcija
- testiranje hoće li se ispitana metoda (akcija) uopće pozvati

- testiranje pristupa podacima
- testiranje privremenih podataka.

Testove u MVC aplikacijama možemo podijeliti na:

- testiranje kontrolera
- testiranje akcija
- testiranje podataka.

No prilikom te podjele nastaju problemi. Testiranje kontrolera i testiranje akcija često su isprepleteni i skoro je nemoguće napraviti čistu granicu gdje jedno počinje, a drugo prestaje. U testu kontrolera praktički testiramo pojedine metode unutar kontrolera. No nisu sve metode akcije, pa time testiranje akcija podrazumjeva testiranje kontrolera, no testiranje kontrolera ne podrazumjeva uvijek testiranje akcija.

8.2.1. Testiranje kontrolera

Testove koje pišemo da budu rađeni na razini kontrolera zovemo testovi kontrolera. Svodimo ih na provjeru ispravnosti kontrolera, no i na provjeru dijelova kontroler klase koji nisu eksponirani kao akcije. No taj dio spada pod standardne testove.

Prilikom definiranja test projekta uz MVC aplikaciju ili WEB API aplikaciju, budući da je rađena na sličnom konceptu, za kod predloška Valuescontroller.cs automatski je generiran ValuesControllerTest.cs u testnom projektu. Da se podjetimo kako ValuesController izgleda:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Net;  
using System.Net.Http;  
using System.Web.Http;  
  
namespace MVCTestiranje.Controllers
```



```

{
    [Authorize]
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        public string Get(int id)
        {
            return "value";
        }

        // POST api/values
        public void Post([FromBody]string value)
        {
        }

        // PUT api/values/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
        public void Delete(int id)
        {
        }
    }
}

```

Na temelju toga je automatski generiran ValuesControllerTest.cs:

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using MVCTestiranje;
using MVCTestiranje.Controllers;

```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Web.Http;

namespace MVCTestiranje.Tests.Controllers
{
    [TestClass]
    public class ValuesControllerTest
    {
        [TestMethod]
        public void Get()
        {
            // Arrange
            ValuesController controller = new ValuesController();

            // Act
            IEnumerable<string> result = controller.Get();

            // Assert
            Assert.IsNotNull(result);
            Assert.AreEqual(2, result.Count());
            Assert.AreEqual("value1", result.ElementAt(0));
            Assert.AreEqual("value2", result.ElementAt(1));
        }

        [TestMethod]
        public void GetById()
        {
            // Arrange
            ValuesController controller = new ValuesController();

            // Act
            string result = controller.Get(5);

            // Assert
            Assert.AreEqual("value", result);
        }
    }
}

```

```

    }

    [TestMethod]
    public void Post()
    {
        // Arrange
        ValuesController controller = new ValuesController();

        // Act
        controller.Post("value");

        // Assert
    }

    [TestMethod]
    public void Put()
    {
        // Arrange
        ValuesController controller = new ValuesController();

        // Act
        controller.Put(5, "value");

        // Assert
    }

    [TestMethod]
    public void Delete()
    {
        // Arrange
        ValuesController controller = new ValuesController();

        // Act
        controller.Delete(5);

        // Assert
    }
}

```

U zadnjih nekoliko metoda testa možemo primijetiti da nam je Assert dio prazan, odnosno nema ga. U kontroleru često je teško testirati sve metode jer neke metode samo primaju određene parametre ili podatke, a budući da nema povratnog rezultata ili nekog odgovora, ovdje možemo reći da je test uspješan ako ne dođe do neke iznimke.

8.2.2. Testiranje podataka

Testove koje pišemo za rad s podacima zovemo testovi podataka. Bilo kakvo testiranje podataka nezgodno je jer često nemamo konzistentnost. Da bi testovi bili uspješni, kod mora za poznati ulaz dati predviđeni izlaz. No što ukoliko dođe do promjene podataka npr. u bazi podataka pa podatak na koji smo se referirali u testovima bude obrisani ili izmijenjen? Testovi nam postaju nepouzdati. Zato za potrebe testiranja uvijek stvaramo neki „lažni” podatkovni skup koji znamo da se neće mijenjati. Testovi podataka mogu biti sljedećeg tipa:

- testiranje privremenih podataka
- testiranje podataka koji se razmjenjuju između modela i kontrolera
- testiranje modela.

Testiranje privremenih podataka radimo tako da unutar kontrolera postavimo „lažne” podatke koje koristimo prilikom vraćanja pogleda. Tako je efikasnije testirati jer smo sigurni u konzistentnost i sadržaj podataka koje koristimo. Ako bismo koristili bazu ili neki drugi dohvat podataka, trebali bismo osigurati da za isti ulaz imamo isti izlaz.

Testiranje podataka koji se razmjenjuju između modela i kontrolera ponekad je nezgodno. Mi možemo testirati razmjenu podataka, napraviti instancu kontrolera jednostavno je, no da bismo mogli testirati prijenos, opet moramo kontrolirati koji se podaci prenose i izvesti assert metode.

Testiranje modela često se svodi na testiranje ispravnosti pristupa podacima. Opet, budući da često ne možemo kontrolirati koji su točno podaci u bazi podataka, a lažne

podatke nema smisla postavljati u slučajevima rada s pristupom, kontroliramo samo možemo li se spojiti na odabrani podsustav. U ovim slučajevima nakon dohvata podataka samo radimo *assert* s obzirom na broj entiteta koje dobivamo kao povrat rezultata.

8.2.3. Testiranje akcija

Testove koje pišemo za specifične akcije zovemo testovima akcija. Svode se na testiranje točno određenih metoda. Kod akcije možemo testirati odrađuju li posao koji treba. Kod akcija testiramo:

- povratni pogled akcije
- uspješnost izvođenja.

Kod povratnog pogleda akcije ispitujemo samo je li vraćen ispravan pogled. Budući da često ne znamo koji će dinamički kreirani podaci biti u pogledu, pouzdano je jedino osloniti se na ime pogleda.

Ako imamo kontroler s jednom akcijom u kontroleru definiran na sljedeći način:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVCTestiranje.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Title = "Home Page";

            return View();
        }
    }
}
```

Kod za testiranje je li vraćen pogled „Index” bio bi sljedeći:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MVCTestiranje;
using MVCTestiranje.Controllers;
using System.Web.Mvc;

namespace MVCTestiranje.Tests.Controllers
{
    [TestClass]
    public class HomeControllerTest
    {
        [TestMethod]
        public void Index()
        {
            // Arrange
            HomeController controller = new HomeController();

            // Act
            ViewResult result = controller.Index() as ViewResult;

            // Assert
            Assert.IsNotNull(result);
            Assert.AreEqual("Home Page", result.ViewBag.Title);
            Assert.AreEqual("Index", result.ViewName);
        }
    }
}
```

Kako smo rekli, testiranje se svodi na provjeru jesmo li dobili dobar pogled, koji možemo dobiti preko naslova, preko gledanja imena pogleda ili nekog drugog proizvoljnog načina.

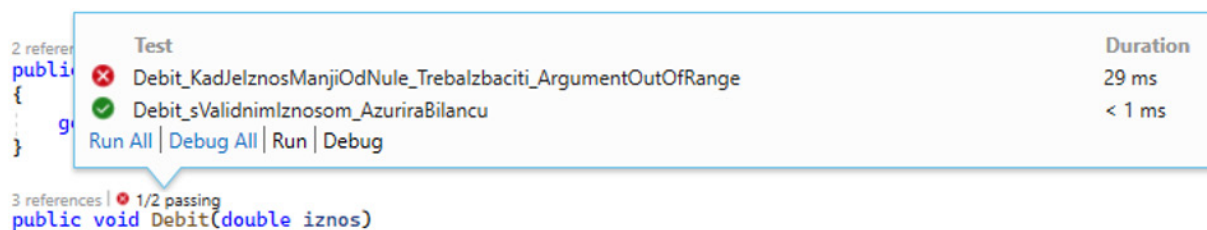
Provjeru uspješnosti izvođenja drugih akcija koje nemaju povratni tip, poput spremanja i slično, možemo ispitati jedino tako da postavimo granične slučajeve i provjerimo bacaju li iznimke. Opet ću samo napomenuti, prilikom testiranja treba koristiti „lažne” podatke, koji nisu u bazi podataka. Testove na „živim” podacima nije preporučljivo raditi, pogotovo ako radite na produkcijskim sustavima.

8.3. Mjerenje uspjeha i stvaranje izvješća

Uspjeh izvedenih testova možemo vidjeti na više mjesta. Prvo i najbitnije jest da sumirano izvješće za cijeli *solution* vidimo u Test Exploreru. Ako imamo više različitih testnih projekata, testovi su strukturno postavljeni. Rezultati su jednostavni kad gledamo, evaluirano je da li je napisani kod test prošao ili nije. Treća opcija može biti jedino ako se test uopće nije izvršio ili ako imamo grešku u pisanju testa. Tijekom pregleda testova metrika vremena trajanja testa može biti dobar način da evaluiramo ima li potencijalnih problema u testovima ako predugo traju. Ta metrika može biti jednako bitna kao i informacija o prolasku ili padu testa jer može ukazati na potencijalne probleme unutar koda.

Tijekom izrade nekog sumiranog izvještaja testovi su odlični u slučajevima kad radimo u timovima. Voditelj tima zahvaljujući testovima može pratiti rad tima, ali je i, ako testovi ne prolaze, lagano naći dio koda koji ne prolazi, budući da se svaki test referira na kod.

Također, tijekom kodiranja Visual Studio pokazat će nam postoje li kod metoda koje kodiramo definirani testovi metode i koliko ih ima s rezultatima zadnjeg testiranja.



Slika 84. Prikaz rezultata povezanih testova

Zahvaljujući Visual Studio alatu omogućeno nam je da kod metoda koje implementiramo da izvedemo testove po potrebi i ukoliko prođu, jednostavno znamo da smo dobro implementirali metode, ako smo prvo napravili sve potrebne testove za kod.

Nema unificiranog načina mjerenja uspjeha, to ovisi o detaljima testova. Savjet je da ne pretjerujete s testovima, ne treba svaku sitnicu testirati, ali svakako ne treba ni zanemariti određene slučajeve korištenja.

Što se tiče stvaranja izvješća, ono ovisi o radnoj okolini i tvrtki gdje radite. Postoje automatizirani sustavi koji će automatski ažurirati radne zadatke s obzirom na uspjeh (ili neuspjeh) testova. No to je više dio domene DevOpsa, odnosno CI/CD rada projekta.

9. POGLAVLJE:

Zaštita na radu

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati mjere zaštite na radu
- razlikovati različite izvore opasnosti pri radu s računalom
- razlikovati različite izvore opasnosti pri radu s računalom
- objasniti upotrebu računala na siguran način primjenom mjera zaštite od ozljeda
- opisati i demonstrirati vježbe opuštanja i razgibavanja radi sprečavanja ozljeda na radnom mjestu.

9.1. Osnove zaštite na radu

Zaštita na radu u Republici Hrvatskoj regulirana je [Zakonom o zaštiti na radu \(„Narodne novine”, br. 071/2014, 118/2014, 094/2018, 096/2018\)](#) te podzakonskim propisima donesenima na temelju Zakona o zaštiti na radu. Svi poslodavci trebaju je organizirati i provoditi.

Zaštita na radu sadrži sustav pravila, mjera i aktivnosti čijom se primjenom osiguravaju sigurni uvjeti na radu te zaštita zdravlja na radu. Cilj zaštite na radu prevencija je, odnosno sprečavanje ozljeda na radu te profesionalnih bolesti koje se mogu javiti prilikom obavljanja nekog posla.

Poslodavac može na različite načine organizirati zaštitu na radu, a to ovisi o djelatnosti kojom se bavi i o broju djelatnika. Razina opasnosti utvrđuje se postupkom procjene rizika, što poslodavac provodi za sve poslove, odnosno sva postojeća radna mjesta unutar ustanove.

Osnovni pojmovi vezani uz zaštitu na radu jesu:

- **Poslodavac** je fizička ili pravna osoba za koju radnik obavlja posao.
- **Radnik** je fizička osoba koja obavlja poslove za poslodavca.
- **Mjesto rada** svako je mjesto na kojemu se obavljaju poslovi iz djelokruga poslodavca.
- **Izdvojeno mjesto rada** mjesto je na kojemu radnik obavlja posao, a nije prostor poslodavca, može biti kod kuće ili u drugom prostoru.
- **Povjerenik** za zaštitu na radu osoba je koja je izabrana da zastupa prava radnika iz područja zaštite na radu.
- **Stručnjak zaštite na radu** osoba je koju je poslodavac odredio za obavljanje poslova zaštite na radu.
- **Ozljeda na radu** ozljeda je koju je radnik zadobio tijekom rada u prostoru rada, bilo kod poslodavca ili drugom prostoru gdje se obavlja rad.
- **Profesionalna bolest** jest bolest nastala kao posljedica djelovanja štetnosti u tijeku rada i/ili radnoj okolini.
- **Stres** predstavljaju zdravstvene i psihičke promjene koje su nastale kao posljedica kontinuiranog utjecaja izvora stresa na radu kroz duži vremenski period.



Slika 85. Ilustracija zaštite na radu

Nezgodom na radu smatra se svaki neželjeni, nepredviđeni događaj koji za posljedicu ima ozljedu na radu koja dovodi do bolovanja, materijalnih gubitaka radnika i poslodavca zbog izgubljenih sati rada ili materijalni gubitak zbog oštećenja opreme ili zastoja u proizvodnom procesu. Nezgode na radu nastaju najčešće kao posljedica ljudske pogreške jer radnik ne zna sigurno raditi, ne može raditi na siguran način ili ne želi raditi na siguran način.

Može biti da radnik nije osposobljen za rad na siguran način i da ne zna do kojih posljedica njegov način rada može dovesti.

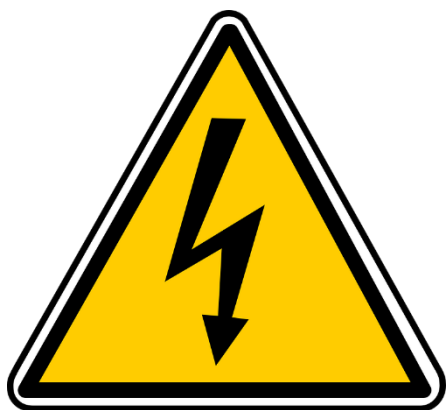
Najčešće zbog zdravstvenih razloga radnik ne može raditi na siguran način. U tom slučaju poslodavac je dužan rasporediti ga na poslove koje je u mogućnosti obavljati.

Kad radnik ne želi raditi na siguran način iako poznaje mjere zaštite na radu, on ugrožava sebe i druge djelatnike i poslodavac mora reagirati na adekvatan način, upozorenjem, razgovorom ili ga udaljiti s tog radnog mjesta.

9.2. Opasnosti i način zaštite na radnom mjestu

Izvori opasnosti pri radu s računalom mogu biti:

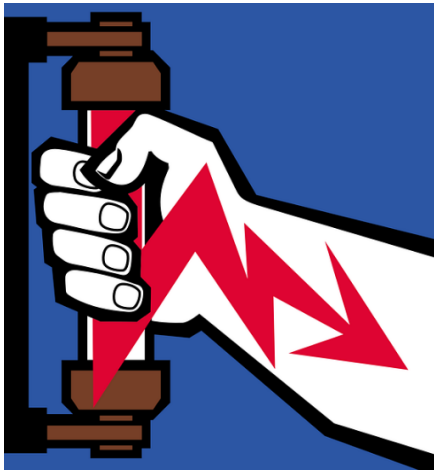
- opasnost od električkog udara
- opasnosti od buke
- opasnosti od štetnih zračenja
- opasnosti od požara i eksplozija, sredstva za zaštitu od požara.



Opasnosti od električkog udara izloženi su svi zaposlenici i posjetitelji ustanove kada električne instalacije i informatička oprema i drugi uređaji nisu ispravni ili pravilno održavani.

Električna struja ako prolazi kroz ljudski organizam može izazvati teške ozljede ili smrtne posljedice.

Slika 86. Opasnost od strujnog udara



Slika 87. Ilustracija strujnog udara

Nezgode od strujnog udara događaju se zbog izravnog dodira s neispravnom opremom ili oštećenom izolacijom na kablovima, sklopka-ma, utičnicama i slično. Elektroničku opremu i električne instalacije potrebno je periodički redovito pregledavati i održavati i na taj način osigurati da ne postoji opasnost od izravnog slučajnog dodira.

Kod uočenih smetnji ili kvarova potrebno je odmah isključiti napon ili izvući utikač uređaja iz utičnice. Isto tako ne smiju se dirati oštećeni prekidači ili utikači jer mogu biti pod naponom. Prilikom zamjene žarulje potrebno je isključiti napon. U slučaju pregrijane ili na dodir vruće utičnice ili uređaja potrebo ih je isključiti kako ne bi izazvali požar, a nakon toga pozvati električara da otkloni kvar.

U slučaju ozljede električnim udarom i ako se ozlijeđeni nalazi u strujnom krugu, potrebno ga je osloboditi od strujnog kruga poštujući mjere sigurnosti za sebe i ozlijeđenog. Najsigurnija opcija isključiti je strujni krug prekidačem ili uklanjanjem vodiča iz blizine ozlijeđenog. U tome vam može pomoći komad suhog drveta.

Osiguranje od buke jedan je segment koji također utječe na sigurnost zaposlenika. Bukom se smatra svaki neželjeni zvuk koji je zaposleniku neugodan ili mu smeta. Djelovanje buke na čovjeka ovisi o njezinoj jačini i frekvenciji. Buka može izazvati gubitak sluha, napetost, smanjenje koncentracije, probleme s razumijevanjem govora, umor i razdražljivost. Posebno je opasno oštećenje sluha jer ima trajnu posljedicu. Sluh izgubljen zbog buke ne može se liječiti. Buka se razlikuje po visini frekvencije, a buka visokih frekvencija štetnija je od buke dubokih tonova.

Buku izražavamo u mjernim jedinicama:

- decibelima (dB) – intenzitet buke
- Hertzima (Hz) – frekvencija.

Prag čujnosti je intenzitet buke 0 dB kod frekvencije 1000 Hz. Kod opremanja ureda treba paziti da uređaji koji se koriste ne stvaraju buku veću od 60 dB, koliko je propisano pravilnikom za takve prostore. Ukoliko je za rad potrebna oprema koja stvara buku veću od propisane, potrebno ju je smjestiti u zasebnu prostoriju. Stalna izloženost buci manje je opasna za zdravlje zaposlenika od povremene zato što se organizam čovjeka prilagođava okolini u kojoj stalno boravi. Osobna zaštitna sredstva koja se mogu koristiti slušalice su i čepići.

Čovjek je svakodnevno izložen nekim vrsta štetnog zračenja i one mogu utjecati na njegovo zdravstveno stanje. Dijelev se na ionizirajuća zračenja, čiji su izvor radioaktivni elementi i susreću se u zdravstvenoj djelatnosti, te neionizirajuća zračenja, koja proizvode određeni uređaji. U neionizirajuća zračenja spadaju elektromagnetsko polje i elektromagnetski valovi.

Izvori neionizirajućih elektromagnetskih polja kojima su zaposlenici u uredu svakodnevno izloženi jesu antene baznih postaja, radarski sustava, televizijskih i radijskih postaja, mikrovalne pećnice, mobilni telefoni, bežični internet, bežični telefoni, daljinski upravljači, indukcijske ploče za kuhanje te protuprovalni sustavi. Udaljenost od takvih uređaja smanjuje razinu zračenja.



Slika 88. Opasnost od elektromagnetskog polja

Preporuke za prevenciju odnose se na izbjegavanje prekomjerne upotrebe mobilnih uređaja, izbjegavanje boravka pored uređaja koji su izvor elektromagnetskih polja, izbjegavanje kontinuiranog nošenja mobilnih uređaja uz tijelo, korištenje slušalica tijekom razgovora ili pisanje poruka umjesto telefonskih razgovora.

Uzrok požara može biti neodgovarajuća upotreba radne opreme ili kvar na elektroinstalacijama, kao i nepažnja zaposlenika ili posjetitelja. U slučaju curenja plina može doći do eksplozije jer čak i iskra prekidača može izazvati eksploziju. Mjere prevencije redovito su održavanje plinskih uređaja i instalacija, kao i postavljanje detektora.



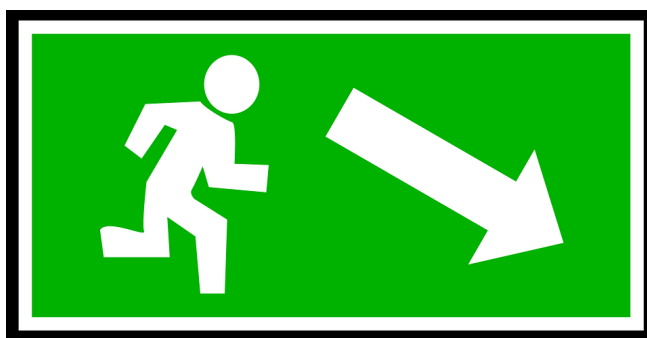
Slika 89. Opasnost od požara

Najvažnije je da zaposlenici budu osposobljeni za zaštitu na radu iz područja zaštite od požara kako bi, u slučaju nastanka požara, mogli intervenirati brzo i učinkovito. Uredi moraju imati vatrogasne aparate za slučaj požara, a zaposlenici trebaju znati kako njima rukovati. Vatrogasni aparati moraju se servisirati jednom godišnje, pristup do njih uvijek mora biti slobodan. Ustanova mora imati jasno istaknut plan evakuacije, a putevi predviđeni za evakuaciju u svakom trenutku trebaju

biti prohodni. Ista pravila vrijede i za hidrante, kojima uvijek mora biti dostupan pristup i u blizini smješten ormarić s opremom za gašenje požara.

Mjere prevencije osposobljenost su zaposlenika za brzu reakciju i početno gašenje požara, svakodnevni pregled radnog mjesta, isključivanje opreme i rasvjete i slično.

U slučaju opekline opečeni dio staviti pod tekuću čistu vodu ili u posudu s hladnom vodom. Potrebno je skinuti nakit i odjeću s opečenog dijela tijela. Ako postoji dio odjeće koji je zalijepljen za opeklinu, ne diramo ga. Iznimno dio odjeće uklanjamo s opeklina kada je riječ o kemikalijama koje mogu izazvati dodatna oštećenja kože. Nastale plikove ne dirati i ne bušiti. Opeklinu zaštititi sterilnom gazom i previti zavojem ili maramom. Ukoliko se radi o većim opeklinama, ozlijeđenom treba dati da pije što više tekućine.



Slika 90. Oznaka puta za evakuaciju

Obveza je poslodavca osigurati sredstva za rad i osobnu zaštitnu opremu zaposlenika kako bi bili sigurni pri radu, kao i održavati istu. Oštećenu opremu ili sredstva osobne zaštite mora ukloniti i isključiti iz upotrebe ako na njoj postoje oštećenja i kao takva predstavljaju rizik za djelatnike.

9.3. Ergonomija i mjere prevencije pri radu s računalom

U radu s računalom treba osigurati da radnik može često mijenjati svoj položaj te da oprema ne bude izvor opasnosti od ozljede ili oštećenja zdravlja djelatnika. Poteškoće uzrokuju dugotrajno sjedenje u istom položaju, dugotrajno gledanje u zaslon, ponavljajući pokreti izazivaju bolove u mišićima i zglobovima. Bolesti povezane uz rad s računalom jesu bolesti vida, mišićnog sustava, koštano-zglobnog i kardiovaskularnog sustava. Nažalost, posljedice se osjete tek u kroničnoj fazi bolesti.

Ergonomija treba rad učiniti što ugodnijim i sigurnijim te smanjiti broj profesionalnih oboljenja. Ergonomija se koristi znanstvenim informacijama prilikom dizajniranja radne opreme i radne okoline.



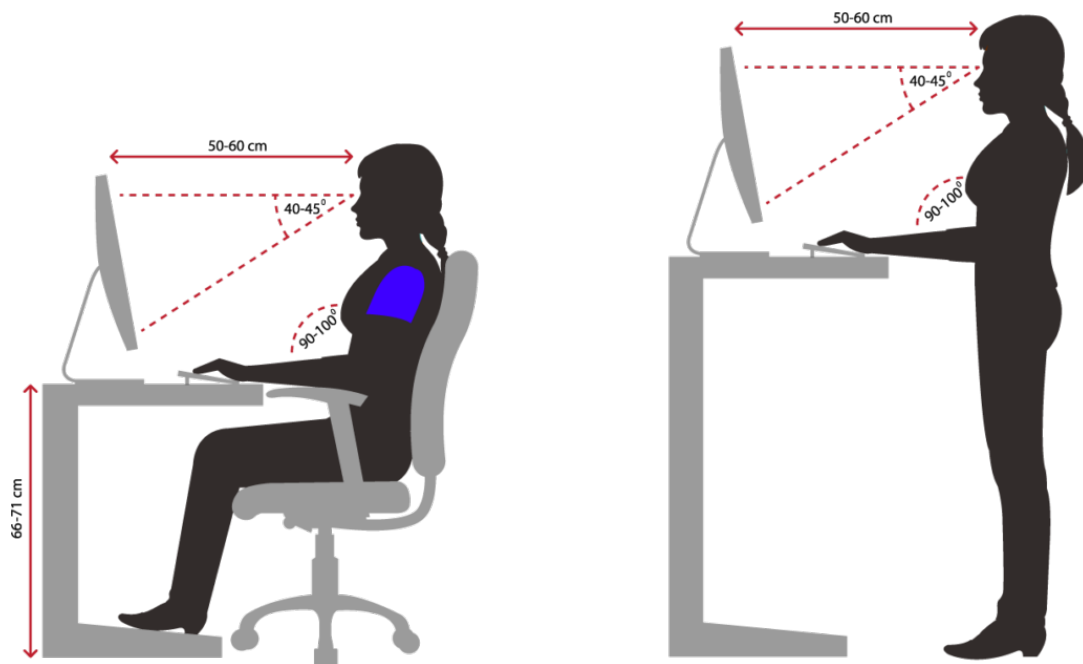
Slika 91. Ispravan i neispravan položaj tijela pri radu s računalom

Radno mjesto potrebno je prilagoditi svakom djelatniku. Pravilnim držanjem tijekom rada s računalom moguće je izbjeći zdravstvene probleme.

Preporučeni položaji za:

- **ruke i nadlaktice** – vodoravan položaj, paralelan s površinom poda
- **glava** – ravan položaj ili lagano nagnjanje prema naprijed
- **ramena** – opuštena s nadlakticama koje vise uz tijelo
- **laktovi** – položeni uz tijelo pod kutom između 90° i 120°
- **stopala** – potpuno spuštена na pod ili oslonac za noge
- **leđa** – u potpunosti naslonjena na naslon stolice
- **koljena** – u ravnini s bokovima, savijena pod 90° .

Osnovni element uredske opreme je radni stol jer određuje raspored ostalih uređaja, a i položaj tijela tijekom rada. Radni stol mora imati dovoljno mjesta da se na njega smjeste svi potrebni uređaji za rad i da omogući djelatniku komotan rad s monitorom na njemu. Odnosno, dubina stola mora omogućiti da na stol stanu ruke i podlaktica dok se koristi tipkovnica. Poželjno je da radni stol bude podesiv po visini i da ima stalak za miš i tipkovnicu. Preporučena visina sjedećeg stola je između 66 i 71 cm. Minimum širine stola je 120 cm. Ispod stola treba biti dovoljno prostora za udobno sjedenje.



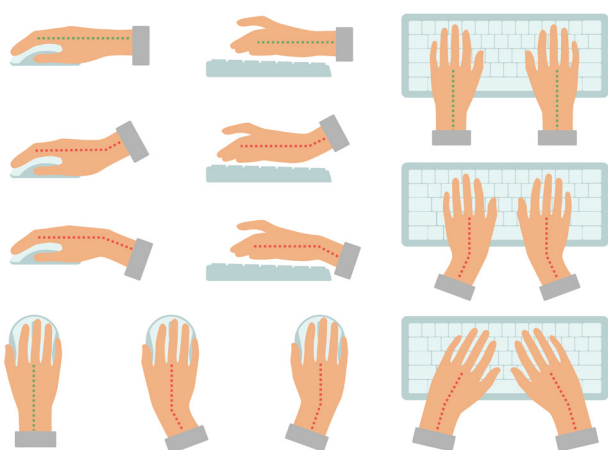
Slika 92. Ispravan položaj tijela kod podesivih stolova

Kod stola za rad u stajaćem položaju visina radnog stola mora biti između 95 i 118 cm, ovisno o visini zaposlenika. Kod ovakve vrste stolova potrebno je paziti da se osigura potrebna dubina za stopala, 15 cm u dubinu i 12 cm u visinu.

Zaslon monitora od očiju treba biti udaljen najmanje 50 cm. Gornji rub zaslona treba biti u visini očiju. Monitor treba biti pomičan kako bi svaki djelatnik, ovisno o položaju za vrijeme rada, mogao prilagoditi njegov smjer i nagib.

Uredska stolica treba biti stabilna, prilagodljive visine, tako da svakom djelatniku može omogućiti udoban položaj i prilagodbu. Naslon mora biti potpora cijelom dužinom leđa i imati mogućnost podešavanja nagiba i visine. Stolica mora biti pokretna na 5 kotačića.

Zadovoljavajuću razinu osvijetljenosti prostorije treba osigurati općom i/ili lokalnom rasvjetom, kao i primjeren kontrast između zaslona i pozadine na računalu, poštujući specifične zahtjeve djelatnika i njegova radnog mjesta. Minimalna propisna razina osvijetljenosti za radna mjesta za računalom jest 300 luxa. Zaslon mora biti namješten i nagnut tako da se izbjegne zrcaljenje rasvjete na zaslonu. Problem refleksije i blještanja moguće je spriječiti pravilnim smještajem radnog mjesta i uređaja sa zaslonom u odnosu na izvori svjetlosti, prozore, druge otvore ili svijetle površine. Kako bi se spriječio ulazak Sunčeve svjetlosti, na prozorima trebaju biti odgovarajući zastori ili kapci. Zaslon treba biti postavljen pod kutom od 90° prema prozoru ili nekom drugom izvoru svjetla.



Slika 93. Pravilan i nepravilan položaj ruku pri upotrebi miša i tipkovnice

Tijekom upotrebe miša i tipkovnice treba paziti na pravilan položaj ruku kako ne bi došlo do ozljede zbog opterećenja zglobova, ruke i šake prilikom ponavljajućih pokreta (engl. *repetiton strain injury*).

Za prirodno držanje ruku poželjno je da tipkovnica bude pokretna na radnoj površini stola. Poželjno

je koristiti ergonomске tipkovnice koje imaju raspored tipki prilagođen prirodnom položaju ruku. Nagib tipkovnice na podlogu trebao bi iznositi 5 – 10°, visina tipkovnice 3 cm, a visina tipki 12 – 15 mm s razmakom između tipki 18 – 20 mm.

Preporuke za prevenciju ozljeda ruku jesu:

- šaka i prsti moraju biti u ravnini s podlakticom
- prsti trebaju biti ravni u odnosu na šaku, bez savijanja zglobova
- razgibavati zglobove i prste više puta tijekom dana
- tipkovnice moraju biti na ravnoj i tvrdoj podlozi i udaljene tako da ispred njih na stol stane cijela podlaktica
- miš i tipkovnica moraju biti dostupni iz istog položaja
- pomicati cijelu ruku pri upotrebi miša, a ne samo zglobove
- ramena trebaju biti opuštena
- stolica treba imati naslon za ruke.



Slika 94. Primjer vježbi rasterećenja

Za očuvanje zdravlja kod rada u uredu pravilno sjedenje nije dovoljno, već je od velike važnosti promijeniti položaj tijela ili aktivnosti tijekom rada. Kod organizacije posla treba omogućiti radniku promjenu aktivnosti ili stanku od 5 minuta nakon svakog sata rada. U toj pauzi mogu se organizirati vježbe rasterećenja. Preporuka je više puta tijekom dana uraditi vježbe rasterećenja, koje će pomoći u smanjenju napora tijekom rada.

Vijeće ministara europske zajednice 1989. i 1990. godine donijelo je odluku o un-ošenju Direktive o zdravlju i zaštiti na radu u povelju o socijalnim pravima (engl. Social Charter). Direktiva 89/391/EEC ima namjeru poticanja poboljšanja zdravstvenih i sig-urnosnih uvjeta na radu. Odnosi se na radni prostor, radnu opremu, osobnu zaštitnu opremu, ručno rukovanje teškim predmetima, rad s monitorima, rad s kancerogenim tvarima te rad s biološkim agentima.

Posebnu važnost ima direktiva 90/270/EEC o minimalnim zahtjevima u pogledu sig-urnosti i zaštite zdravlja pri radu sa zaslonima, ali isključuje prijenosna računala.

Ova direktiva definira zahtjeve za:

- **monitore** – oblik slova, veličina i definicija, stabilnost i treperenje slike, kontrola svjetline i kontrasta, podešavanje kuta zaslona, sjaj i refleksija
- **tipkovnice** – podešavanje nagiba i odvajanje od zaslona, podrška za šaku i ruku, sjaj i refleksija, raspored tipki i čitljivost
- **radne površine** – sjaj i refleksija, prostor za opremu, držači dokumenata
- **stolice** – podešavanje i oslonac za noge
- **radnu okolinu** – rasvjeta, sjaj i refleksija, buka, toplina, zračenje i vlažnost zraka
- **korisnička sučelja** – prikladnost softvera, lakoća korištenja prilagođena vještini korisnika, vrijeme odziva i svojstva, izgled.

„Sadržaj publikacije/emitiranog materijala isključiva je odgovornost RCK ELPROS“



Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.