C# i .Net Framework

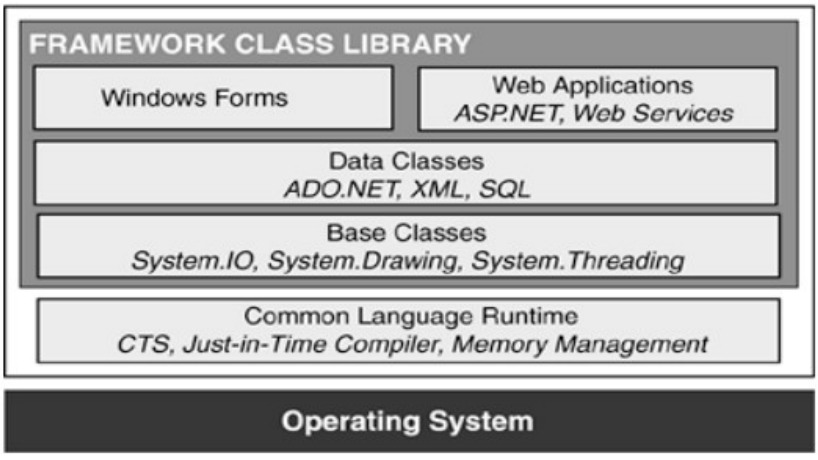
.NET Framework je platforma koja uključuje jezike, runtime, i framework biblioteke, omogućavajući developerima da stvore različite tipove aplikacija. C# je jedan od jezika .NET-a, koji uključuje i Visual Basic, F#, C++. Osnovne ideje pri dizajniranju .NET-a su bile:

- Da se formira konzistentno objektno orijentisano okruženje za različite aplikacije.
- Da se napravi okruženje koje minimizuje probleme za uvođenjem novih verzija programa ("DLL Hell") koji su postojali u Windows (COM) okruženju i da se pojednostavi proces distribucije/instalacije.
- Da se formira portabilno okruženje, bazirano na opšteprihvaćenim standardima, koje može da se izvršava na proizvoljnom operativnom sistemu.
- Da se obezbedi upravljano (managed) okruženje u kome će se kod lako i sigurno izvršavati.

Da bi se ostvarili ovi ciljevi, .NET Framework se oslanja na ***Common Language Runtime (CLR)*** i ***Framework Class Library (FCL)***. **Slika 1** prikazuje ove osnovne komponente .NET Framework-a.

CLR vodi računa o izvršavanju koda i svim ostalim zadacima u vezi sa tim: kompajliranje, upravljanje memorijom, sigurnost, upravljanje nitima i kontrolisanje tipova podataka. Kod koji se izvršava u CLR se naziva upravljani (managed) kod. Nasuprot tome, neupravljni (unmanaged) je kod koji ne implementira zahteve za rad u .NET Framework-u(kao što je COM ili Windows API).

Druga komponenta, Framework Class Library, predstavlja biblioteku klasa i struktura koje su na raspolaganju aplikacijama koje rade u .NET. Tu spadaju klase za pristup bazama podataka, grafički interfejs, interoperabilnost sa neupravljanim kodom, sigurnost podataka, kao i za Web i Windows forme. Svi programski jezici prilagođeni .NET-u koriste ovu bibilioteku.



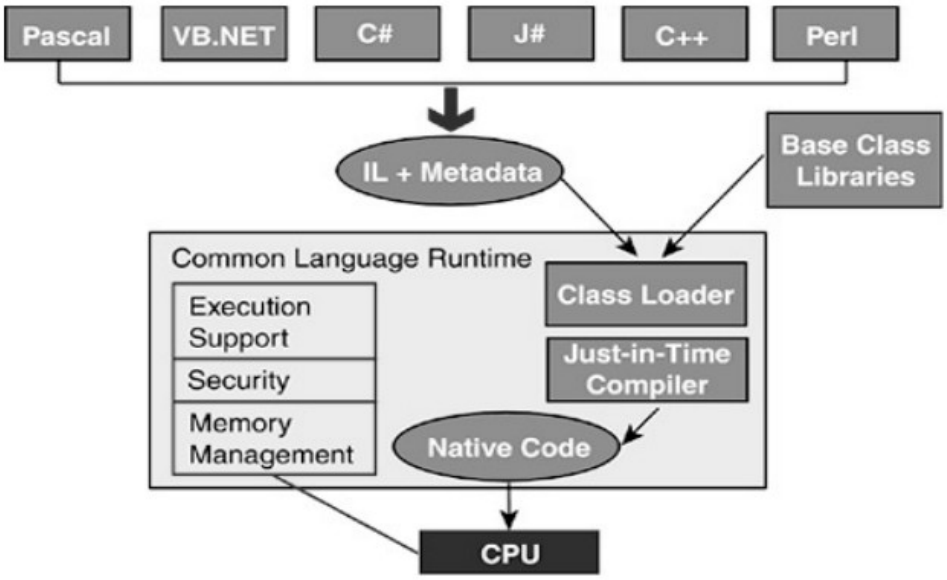
Slika 1: Osnovne komponente .NET Framework-a

Common Language Runtime

Common Language Runtime upravlja celim životnim vekom aplikacije: on pronalazi kod, prevodi ga, učitava potrebne klase, upravlja njihovim izvršavanjem i osigurava automatsko oslobađanje memorije. CLR je zadužen i za integraciju koda između različitih programskih jezika.

Prevođenje .NET koda

Programski prevodioci koji su kompatiblini sa CLR-om proizvode kod koji je pogodan za izvršavanje u runtime-u, za razliku od onih proizvedenih za specifični CPU. Ovaj kod, poznat pod različitim imenima ***Common Intermediate Language (CIL), Intermediate Language (IL),*** ili ***Microsoft Intermediate Language (MSIL)***, je asemblerski jezik koji se pakuje u EXE ili DLL datoteke. Ovaj kod (intermediate code) je ključna stvar za ispunjenje osnovnog cilja .NET Framework-a – interoperabilnost između programskih jezika. **Slika 2** ilustruje na koji način CLR komunicija sa jezički nezavisnim IL. Common Language Runtime ne mora da zna, niti treba da zna, u kom programskom jeziku je aplikacija napisana. Zahvaljujući tome što aplikacije komuniciraju kroz IL, izlaz iz jednog prevodioca može biti integrisan sa kodom koji je proizveo drugi prevodilac.



Slika 2: Mesto i funkcija CLR-a

C#

C# predstavlja objektno orjentisan programski jezik opšte namene. Kao jezik opšte namene, on može biti iskorišćen na mnoge načine, za izvršavanje različitih tipova zadataka. Neke od osnovnih primena su web aplikacije koristeći ASP.NET, desktop aplikacije uz WPF(Windows Presentation Foundation), ili mobilne aplikacije za Windows telefone/tablete.

Radno okruženje

Kao razvojno okruženje za C# aplikacije, Microsoft nudi Visual Studio(VS). U ovu svrhu se mogu koristiti i drugi editori, kao što su Notepad++, Visual Studio Code, Sublime, itd. Ipak, na ovim vežbama koristićemo Visual Studio, zbog jednostavnosti i efikasnosti.

Pokretanje programa

Za početak, potrebno je napraviti novi program. U VS-u potrebno je izabrati **File** > **New** > **Project,** zatim **Installed** > **Templates** > **Visual C#** u stablu sa leve strane, i na kraju odabrati tip projekta **Console Application.** Nazvati projekat Greetings i postaviti željenu lokaciju. Ovo kreira novo rešenje(solution) za nas. Obrisati Program.cs fajl i dodati Greeting.cs fajl. Sledi kod koji ispisuje "Greetings!" u komandnoj liniji.

```
using System;

class Greetings
{
    static void Main()
    {
        Console.WriteLine("Greetings!");
    }
}
```

Metoda **Main** označava mesto gde program počinje sa izvršavanjem. Modifikator **static** nam govori da postoji samo jedna insatnca **Greetings** klase, koja sadrži **Main** metodu. S obzirom da program ima samo jednu **Main** metodu, upotreba static modifikatora ima smisla.

Unutar **Main** metode se nalazi iskaz koji ispisuje string u komandnoj liniji. String se prosleđuje **WriteLine** metodi, koja ispisuje željeni string u komandnoj liniji i prebacuje je u novi red. **WriteLine** metoda pripada klasi **Console**. Klasa **Console** pripada namespace-u **System**, zbog čega kod počinje sa using naredbom. FCL je grupisan u namespace-ove, kako bi kod bio organizovan i kako bi se izbegli sukobi između tipova sa istim imenom. Ova **using** naredba nam omogućava da koristimo kod iz **System** namespace-a, bez čega kompajler ne bi znao šta znači **Console**. Na ovaj način, kompajler zna da se radi o klasi **System.Console**.

Imenski prostori i organizacija koda

Jedan od uobičajenih načina za organizaciju koda jeste pomoću C# opcije imenskih prostora(**eng. namespace**). Hijerarhijska organizacija koda je u tom slučaju sledeća:

Imenski prostor
    Klasa
        Član Klase

U ovoj hijerarhiji namespace je opcion, kao što se moglo videti u prethodnom programu, gde **Greeting** klasa nije pripadala nijednom imenskom prostoru. Ovo znači da je **Greeting** klasa pripadnik globalnog imenskog prostora. Ovu praksu bi trebalo izbegavati, jer bi se moglo desiti da drugi developeri, koji rade sa istim kodom, napišu svoju **Greetings** klasu u istom imenskom prostoru. Ovo bi dovelo do grešaka, jer C# kompajler ne bi znao o kojoj **Greetings** klasi se radi.

U sledećem primeru klasa **Calc** pripada **Syncfusion** imenskom prostoru. Metoda **Pythagorean** je član **Calc** klase. Ova organizacija koda prati **Imenski prostor-Klasa-Član Klase** organizaciju.

```
using static System.Math;

namespace Syncfusion
{
    public class Calc
    {
        public static double Pythagorean(double a, double b)
        {
            double cSquared = Pow(a, 2) + Pow(b, 2);
            return Sqrt(cSquared);
        }
    }
}
```

Naredba **using static** omogućava kodu da koristi statičke članove **System.Math** klase, bez pune kvalifikacije. Umesto pisanja **Math.Pow(a, 2)**, za kvadriranje broja a, može da se koristi skraćena sintaksa u **Pythagorean** metodi. Isto važi i za **Sqrt** metodu, koja nam vraća kvadratni koren prosleđenog broja. Sledeći primer nam pokazuje kako bi mogla izgledati primena ovog koda.

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
```

Main metoda poziva **Pythagorean** metodu iz **Calc** klase, prosleđujući parametre 2 i 3, uz povratni rezultat u promenljivoj **hypotenuse**. Kako se **Calc** klasa nalazi u **Syncfusion** imenskom prostoru, dodata je **using Syncfusion** naredba. Bez ovoga, **Main** bi bio primoran da koristi potpuno kvalifikovano ime **Syncfusion.Calc.Pythagorean**.

Još jedna opcija prethodnog programa je pseudonim imenskog prostora Crypto. Ova sintaksa dozvoljava korišćenje skraćene verzije potpunog imena imenskog prostora. Crypto je pseudonim za **System.Security.Cryptography** i **Main** ga koristi kao **Crypto.AesManaged,** kako bi učinio kod čitljivijim.

Pokretanje programa

U VS-u, program se pokreće klikom na zelenu strelicu Start u toolbar-u, što build-uje i pokreće program. Isto radi i pritisak na taster F5. Ukoliko se program pokrene i završi prebrzo, potrebno je stisnuti Ctrl + F5, kako bi komandna linija ostala otvorena.

Kodiranje i naredbe

Pisanje jednostavnih naredbi

Kombinovanjem jezičkih operatora i sintakse, prave se izrazi i naredbe u C#-u. Neki od osnovnih primera su sledeći.

```
int count = 7;
char keyPressed = 'Q';
string title = "Weekly Report";
```

Svaki od navedenih primera ima zajedničke sintaksne elemente: tip, identifikator promenljive, operator dodele, vrednost i kraj naredbe. Ovi primeri pokazuju kako deklarisati promenljivu i izvršiti dodelu vrednosti u jednom koraku. Ovo, međutim, nije neophodno. Dok god se promenljiva deklariše pre upotrebe, sve je u redu. Sledi i primer odvojene deklaracije.

```
string title;
```

A zatim i naknadne dodele vrednosti.

```
title = "Weekly Report";
```

C# tipovi i operatori

C# je strogo tipiziran jezik, što znači da kompajler neće implicitno konvertovati nekompatibilne tipove. Na primer, nije moguće pretvoriti **string** u **int**, kao ni **int** u **string**. Sledeći kod neće biti kompajliran.

```
int total = "359";
string message = 7;
```

Iako koverzija ne može da se odradi implicitno, postoje načini da se odradi eksplicitno. Primeri ovog postupka su dati u nastavku.

```
int total = int.Parse("359");
string message = 7.ToString();
```

U prethodnom primeru, **Parse** će konvertovati **string** u **int**, ako **string** predstavlja validan **int**. Pozivanjem **ToString** metode za bilo koju vrednost, dobićemo njenu **string** predstavu.

Uz ove primere, treba navesti da C# ima i operator za kastovanje. Ukoliko biste hteli da pretvorite **double** vrednost koja predstavlja 64-bitnu vrednost, i dodelite je **integer-u**, koji je 32-bitni broj, to bi izledalo ovako:

```
double preciseLength = 5.61;
int roundedLength = (int)preciseLength;
```

Bez operatora za kastovanje, dogodila bi se greška prilikom kompajliranja, jer **double** nije **int**. Suštinski, C# kompajler nas štiti od gubitka preciznosti usled pretvaranja **double** tipa u **int**.

U narednoj tabeli prikazani su ugrađeni tipovi:

Type (Literal Suffix)	Description	Values/Range
byte	8-bit unsigned integer	0 to 255
sbyte	8-bit signed integer	-128 to 127
short	16-bit signed integer	-32,768 to 32,767
ushort	16-bit unsigned integer	0 to 65,535
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647
uint	32-bit unsigned integer	0 to 4,294,967,295
long (l)	64-bit signed integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong (ul)	64-bit unsigned integer	0 to 18,446,744,073,709,551,615
float (f)	32-bit floating point	-3.4×10^{38} to $+3.4 \times 10^{38}$
double (d)	64-bit floating point	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
decimal (m)	128-bit, 28 or 29 digits of precision (ideal for financial)	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$
bool	Boolean	true or false
char	16-bit Unicode character (use single quotes)	U+0000 to U+FFFF
string	Sequence of Unicode characters (use double quotes)	E.g., “abc”

Uvek bi trebalo dodati sufiks broju, ukoliko njegova vrednost može biti dvosmislena. U sledećem primeru sufiks **m** osigurava da se literal 9.95 posmatra kao **decimal** broj.

```
decimal price = 9.95m;
```

Do sada, naredbe su sadržale samo operator dodele, ali C# ima i mnoge druge operatore, koji dozvoljavaju izvršavanje logičkih operacija. Naredna tabela prikazuje neke od dostupnih operatora.

Category	Description
Primary	`x.y  x?.y  f(x)  a[x]  x++  x--  new  typeof  default  checked  unchecked  nameof`
Unary	`+  -  !  ~  ++x  --x  (T)x  await x`
Multiplicative	`*  /  %`
Additive	`+  -`
Shift	`<<  >>`
Relational and Type Testing	`<  >  <=  >=  is  as`
Equality	`==  !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null Coalescing	`??`
Conditional	`?:`
Assignment	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=  =>`

Prefiks operatori menjaju vrednost varijabli pre dodele, a postfiks operatori menjaju promenljivu nakon dodele, kao što je prikazano u narednom primeru:

```
int val1 = 5;
int val2 = ++val1;
int val3 = 2;
int val4 = val3--;
```

Ovde **val1** i **val2** imaju vrednost 6, **val3** 1, a **val4** 2, jer postfiks operator menja promenljivu tek nakon dodele.

Ternarni operator nudi jednostavnu sintaksu za **if-then-else** logiku. Evo i primera:

```
decimal priceGain = 2.5m;
string action = priceGain > 2m ? "Buy" : "Sell";
```

Sa leve strane upitnika nalazi se **bool** izraz, **priceGain > 2m**. Ukoliko je ovo tačno, ternarni operator vraća prvu vrednost između upitnika i **:**, što je **"Buy"**. U suprotnom, ternarni operator vraća vrednost nakon **:**, što je **"Sell"**. Vraćena vrednost se dodeljuje promenljivoj **action**.

Pored ugrađenih tipova, **FCL** sadrži mnoge tipove koji se koriste gotovo svakodnevno. Jedan od ovih tipova je **DateTime**, koji predstavlja datum i vreme. Sledi nekoliko primera upotrebe ovog tipa.

```
DateTime currentTime = DateTime.Now;
string shortDateString = currentTime.ToShortDateString();
string longDateString = currentTime.ToLongDateString();
string defaultDateString = currentTime.ToString();
DateTime tomorrow = currentTime.AddDays(1);
```

Prethodni kod prikazuje redom kako se dolazi do trenutnog vremena, kratkog prikaza datuma(npr. 20/10/2018), dugački prikaz datuma i vremena(sve je spelovano), uobičajeni numeričku prikaz i kako upotrebiti **DateTime** metode za računanje.

Formatiranje stringova

Postoje različiti načini za izgradnju i formatiranje stringova u C#-u: konkatenacija, stringovi numeričkog formata ili interpolacija stringa. Sledeći kod prikazuje proces konkatenacije:

```
string name = "Joe";
string helloViaConcatenation = "Hello, " + name + "!";
Console.WriteLine(helloViaConcatenation);
```

Ovo ispisuje **"Hello, Joe!"** u konzoli. Sledeći primer radi istu stvar, samo koristeći **string.Format** metodu. Ova metoda postavlja redom vrednosti argumenata(navedenih u zagradi posle zareza) na mesto brojeva unutar vitičastih zagrada. Kako u ovo primeru postoji samo jedan broj i jedan argument, **{0}** će biti zamenjeno sa vrednošću promenljive **name**, što je **“Joe“**, tako da u konzoli ponovo dobijamo **"Hello, Joe!"**.

```
string helloViaStringFormat = string.Format("Hello, {0}!", name);
Console.WriteLine(helloViaStringFormat);
```

Što se dublje zadire u formatiranje stringova, to ono postaje moćnije, dozvoljavajući specificiranje dužine kolone, poravnanje, i formatiranje same vrednosti. Prikaz ovoga je dat u nastavku:

```
string item = "bread";
decimal amount = 2.25m;
Console.WriteLine("{0,-10}{1:C}", item, amount);
```

U datom primeru, prva vitičasta zagrada zauzima 10 karaktera u dužinu. Osnovno poravnanje je desno, ali znak minus menja to u levo. U drugim vitičastim zagradama **C** predstavlja format stringa za novac.

C# 6 je uveo nove načine za formatiranje stringa, putem interpolacije. Ovo je kratka sintaksa koja omogućava direktan upis vrednosti promenljivih, koje se nalaze u vitičastim zagradama.

```
Console.WriteLine($"{item}      {amount}");
```

Znak **$** je neophodan. Ovde, vrednost promenljive **item** zamenjuje **{item}**, a vrednost promenljive **amount** zamenjuje **{amount}**.

Grananje

Za grananje mogu da se koriste **if-else** ili **switch** naredbe. U nastavku dat je primer upotrebe **if** naredbe sa jednom, dve ili više grana.

```
string action2 = "Sell";
if (priceGain > 2m)
{
    action2 = "Buy";
}
```

```
string action3 = "Do Nothing";
if (priceGain <= 2m)
{
    action3 = "Sell";
}
else
{
    action3 = "Buy";
}
```

```
string action4 = null;
if (priceGain <= 2m)
{
    action4 = "Sell";
}
else if (priceGain > 2m && priceGain <= 3m)
{
    action4 = "Do Nothing";
}
else
{
    action4 = "Sell";
}
```

Za postavljanje uslova u **if** naredbi koriste se neki od relacionih operatora(**>, <, >=, <=**), kao i operatori jednakosti(**==, !=**). Ukoliko je potrebno povezati više uslova u jedan, koriste se uslovni operatori **&&** i **||**. Operator **&&** daje tačan rezultat ukoliko su svi izrazi tačni, a operator || daje tačan rezultat ukoliko je bar jedan od uslova tačan. Ovi operatori ne ispituju uslove sa desne strane, ukoliko oni sa leve strane čine čitav izraz netačnim. U slučaju **else if** naredbe u poslednjem primeru, gde se proverava uslov **priceGain > 2m**, operator **&&** ne bi ispitivao izraz sa desne strane.

If naredba je dobra za jednostavno grananje ili za kompleksne uslove, kao što je slučaj sa poslednjom **else if** naredbom. Međutim, ukoliko postoji više slučajeva, a svi izrazi su konstantne vrednosti, kao što su **int** ili **string**, bolje je koristiti **switch** naredbu. Sledeći primer koristi **switch** naredbu kako bi odredio pogodnu opremu u zavisnosti od vremenske prognoze.

```
string currentWeather = "rain";
string equipment = null;
switch (currentWeather)
{
    case "sunny":
        equipment = "sunglasses";
        break;
    case "rain":
        equipment = "umbrella";
        break;
    case "cold":
    default:
        equipment = "jacket";
        break;
}
```

Switch naredba pokušava da spoji vrednost, u ovom slučaju **currentWeather**, sa jednim od slučajeva**(case)**. U slučaju nepoklapanja ni sa jednim definisanim slučajem koristi se **default** slučaj. Jedini slučaj u kom je dozvoljeno "propadanje" kroz **switch** jeste kada jedan od slučajeva nema telo, odnosno ne sadrži nikakav kod, što je u našem primeru samo slučaj **"cold"**.

Osim naredbi za grananje, postoji i potreba za izvršavanjem nekog seta operacija više puta, gde na scenu stupaju C# petlje. Pre pregleda petlji, biće dat pregled nizova i kolekcija, koje mogu da sadrže podatke koji se koriste unutar petlji.

Nizovi i kolekcije

Ponekad je potrebno grupisati elemente u kolekciju, kako bi bili smešteni u memoriju. Za ovo mogu da se koriste nizovi, kao i mnogi drugi tipovi kolekcija iz .NET Framework-a. Naredni primer demonstrira jedan od načina za kreiranje niza.

```
int[] oddNumbers = { 1, 3, 5 };
```

```
int firstOdd = oddNumbers[0];
int lastOdd = oddNumbers[2];
```

Nakon kreiranje niza neparnih brojeva od 1 do 5, vršeno je indeksiranje njegovih elemenata pri čemu promenljiva **firstOdd** dobija vrednost prvog člana niza(1), a **lastOdd** dobija vrednost poslednjeg člana niza(5). Indeksi u C#-u kreću od **0**, a sintaksa koja se koristi za indeksiranje jeste **[x]**, gde **x** predstavlja željeni indeks. Sledi još jedan primer, ovoga puta sa stringom.

```
string[] names = new string[3];
names[1] = "Joe";
```

U prethodnom primeru, inicijalizovan je niz sa tri stringa. Svi stringovi imaju **null**, odnosno neinicijalizovanu vrednost na početku. Ovaj kod menja vrednost drugog stringa na **"Joe"**.

Pored nizova, mogu da se koriste i ostale strukture podataka, kao što su **List**, **Stack**, **Queue,** kao i mnoge druge, koje predstavljaju deo FCL-a. Naredni primer pokazuje kako bi trebalo koristiti listu. Ne treba zaboraviti **using System.Collections.Generic** naredbu, kako bi uopšte bilo moguće koristiti listu.

```
List<decimal> stockPrices = new List<decimal>();
stockPrices.Add(56.23m);
stockPrices.Add(72.80m);
decimal secondStockPrice = stockPrices[1];
```

U ovom primeru inicijalizovana je nova **List** kolekcija. **<decimal>** predstavlja generički tip, koji govori da je ovo strogo tipizirana lista, koja može da sadrži samo vrednosti tipa **decimal**. Lista sadrži dva elementa. Indeksiranje liste se može vršiti na isti način kao i kod nizova, ili upotrebom metode **List.ElementAt**.

Petlje

C# podržava nekoliko petlji, uključujući **for**, **foreach**, **while** i **do-while**. Kodovi koji slede, obavljaju slične poslove.

```
double[] temperatures = { 72.3, 73.8, 75.1, 74.9 };
for (int i = 0; i < temperatures.Length; i++)
{
    Console.WriteLine(i);
}
```

For petlja iz prethodnog primera inicijalizuje vrednost promenljive **i** na **0**, proverava da li je vrednost te promenljive manja od broja elemenata u **temperature** nizu, izvršava **Console.WriteLine** metodu, a zatim uvećava promenljivu **i** za jedan. Petlja se izvršava dok god je **i** manje od **temperature.Length**.

U narednom primeru, koristi se jednostavnija, **foreach** petlja, koja se izvršava za svaki član iz **temperature** niza.

```
foreach (int temperature in temperatures)
{
    Console.WriteLine(temperature);
}
```

Sledeći je primer **while** petlje. **While** petlja proverava uslov i izvršava se ukoliko je taj uslov tačan.

```
int tempCount = 0;
while (tempCount < temperatures.Length)
{
    Console.WriteLine(tempCount);
    tempCount++;
}
```

Za kraj, dolazimo do primera **do-while** petlje:

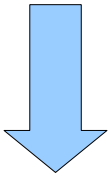
```
int tempCount2 = 0;
do
{
    Console.WriteLine(tempCount2++);
}
while (tempCount2 <= temperatures.Length);
```

Do-while petlja je dobra ukoliko nešto treba izvršiti barem jednom. Treba primetiti da za isti uslov, **while** i **do-while** petlja neće imati isti broj izvršenih ciklusa.

Pregled

Za kraj ovog dela dat je primer programa kalkulatora, koji spaja osnovne koncepte spomenute u dosadašnjem delu.

```
using System;
using System.Text;

/*
    Title: Calculator
    By: Joe Mayo
*/

class Calculator
{
    /// <summary>
    /// This is the entry point.
    /// </summary>
    static void Main()
    {
        char firstChar = 'Q';
        bool keepRunning = true;

        do
        {
            Console.WriteLine();
            Console.Write("What do you want to do? (Add, Subtract, Multiply, Divide,
Quit): ");
            string input = Console.ReadLine();
            firstChar = input[0];

            // This is used in both the if statement and the do-while loop.
            keepRunning = !(firstChar == 'q' || firstChar == 'Q');

            double firstNumber = 0;
            double secondNumber = 0;
```



```
            if (keepRunning)
            {
                Console.Write("First Number: ");
                string firstNumberInput = Console.ReadLine();
                firstNumber = double.Parse(firstNumberInput);

                Console.Write("Second Number: ");
                string secondNumberInput = Console.ReadLine();
                secondNumber = double.Parse(secondNumberInput);
            }

            double result = 0;
            switch (firstChar)
            {
                case 'a':
                case 'A':
                    result = firstNumber + secondNumber;
                    break;

                case 's':
                case 'S':
                    result = firstNumber - +secondNumber;
                    break;
                case 'm':
                case 'M':
                    result = firstNumber * secondNumber;
                    break;
                case 'd':
                case 'D':
                    result = firstNumber / secondNumber;
                    break;
                default:
                    result = -1;
                    break;
            }

            Console.WriteLine();
            Console.WriteLine("Your result is " + result);

        } while (keepRunning);
    }
}
```

Ovaj program demonstrira **do-while** petlju, **if** naredbu, **switch** naredbu, kao i osnovnu komunikaciju između konzole i korisnika. Ono što je još prikazano jeste unos teksta sa tastature pomoću **Console.ReadLine** metode. U dnu programa prikazana je upotreba konkatenacije, prilikom ispisa **"Your result is " + result.** Korišćenje + znaka je jednostavan način za izgradnju stringova. Treba napomenuti da je **string** tip nepromenljiv, odnosno, nije ga moguće modifikovati, što znači da se prilikom svake konkatenacije u memoriji kreira novi string. Još jedan način za izgradnju stringova je upotrebom StringBuilder klase, koja se koristi na sledeći način:

```
StringBuilder sb = new StringBuilder();
sb.Append("Your result is ");
sb.Append(result.ToString());
Console.WriteLine(sb.ToString());
```

Za upotrebu ove klase neophodno je dodati **using System.Text** naredbu. Ovaj program prikazuje i kako se ispisuju komentari, koristeći // i /* */.

Metode i svojstva

Za potrebe ovog poglavlja biće upotrebljena uprošćena verzija programa kalkulatora iz prethodnog dela. Ovaj kalkulator izvršava samo sabiranje i zaustavlja se nakon jedne operacije.

```
using System;

class Calculator1
{
    static void Main()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);

        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);

        double result = firstNumber + secondNumber;

        Console.WriteLine($"\n\tYour result is {result}.");

        Console.ReadKey();
    }
}
```

Modularizacija pomoću metoda

Iako je prethodni program mali, na prvi pogleda nam ne govori šta tačno radi. Da je kod duži, bilo bi ga još teže razumeti. Da bi se ovo rešilo potrebno je izdeliti kod, bez promene njegove funkcionalnosti. Sledeći primer pokazuje **refaktorisanje** koda u metode, zarad lakšeg razumevanja:

```
using System;

class Calculator2
{
    static void Main()
    {
        double firstNumber = GetFirstNumber();

        double secondNumber = GetSecondNumber();

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }
```

```
    static double GetSecondNumber()
    {
        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);
        return secondNumber;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Pogledom na novi **Main**, može se reći šta program radi. Čita dva broja, sabira ih, i onda ispisuje rezultat korisniku. Svaka od linija koda predstavlja poziv metode. Prve tri metode(**GetFirstNumber**, **GetSecondNumber**, i **AddNumbers**) vraćaju vrednost tipa **double**, koja se dodeljuje promenljivoj. Poslednja metoda, **PrintResult**, se izvršava bez povratne vrednosti i stoga se definiše kao **void** metoda.

Uprošćavanje koda pomoću metoda

Poslednja promena je unapredila program deljenjem velikog bloka koda na smislene delove. Međutim, ovaj program se dalje može unaprediti izbacivanjem **GetFirstNumber** i **GetSecondNumber** metoda, odnosno njihovim spajanjem u jednu metodu, kako bi se smanjila količina koda.

```
using System;

class Calculator3
{
    static void Main()
    {
        double firstNumber = GetNumber("First");
        double secondNumber = GetNumber("Second");

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);
        return number;
    }
```

```
    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Ovog puta uklonjene su **GetFirstNumber** i **GetSecondNumber** metode i zamenjene sa **GetNumber.** Jedina razlika između ovih metoda je, osim naziva promenljivih, **whichNumber string** parametar.

Dodavanje svojstava

Prethodni primeri su izvršavali sve operacije unutar iste klase. Sve je počinjalo od **Main-a** i obavljano putem metoda. Međutim, za ponovnu upotrebu funkcija kalkulatora je u ovom slučaju potrebno ponovo pozvati sve metode, jednu po jednu. Takođe, vrednosti izvršenih operacija se čuvaju samo kao privremene vrednosti unutar **Main** metode. Da bi se obezbedila bolja upotreba programa kalkulatora, potrebno je izdvojiti njegove metode u posebnu **Calculator** klasu, gde će se ujedno čuvati i informacije o značajnim vrednostima, odnosno stanjima klase.

Za dalji rad treba prvo osmisliti najbolji način za dobijanje stanja klase **Calculator**. Na primer, ukoliko nam je potrebno da očitamo trenutni rezultat, mogli bismo da upotrebimo metodu **GetResult**, koja nam vraća vrednost. Međutim, ovo se može uraditi i upotrebom svojstava(**eng. Properties**), koja je moguće upotrebiti kao polja klase, ali koja rade kao metode. Naredna verzija kalkulatora pokazuje kako refaktorisati metode u posebnu klasu i dodati svojstva.

```
using System;

public class Calculator4
{

    double[] numbers = new double[2];

    public double First
    {
        get
        {
            return numbers[0];
        }
    }

    public double Second
    {
        get
        {
            return numbers[1];
        }
    }
```

```
    double result;

    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    public void GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);

        if (whichNumber == "First")
            numbers[0] = number;
        else
            numbers[1] = number;
    }

    public void AddNumbers()
    {
        Result = First + Second;
    }

    public void PrintResult()
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

U prethodnom kodu, **First**, **Second** i **Result** predstavljaju svojstva. Metoda **AddNumbers** čita vrednosti **First** i **Second** svosjstva i sabira ih u **Result.**

Svako od ovih svojstava izgleda kao polje ili promenljiva, međutim, ukoliko se pogledaju njihove definicije, odmah se može videti da to nisu polja. Ova svojstva su definisana preko **get** i **set** metoda. Kada se čita vrednost svojstva, izvršava se **get** metoda. Kada se svojstvu dodeljuje vrednost, izvršava se **set** metoda. Može se primetiti da pored **Result** svojstva, postoji i **result** polje. Metode **get** i **set**, čitaju, odnosno upisuju vrednost **result** polja. Kada se korsiti **set**, ključna reč **value** predstavlja ono što će biti upisano kao vrednost svojstva.

Šablon čitanja i pisanja vrednosti iz jednog izvora(polja) je toliko uobičajen da C# ima sintaksnu prečicu za ovu primenu. Naredni kod predstavlja svojstvo **Result**, ovoga puta napisano kao auto-implementirajuće svojstvo:

```
    public double Result { get; set; }
```

Izvor koji stoji iza ovako definisanog auto-implementirajućeg svojstva je podrazumevan, a C# kompajler je taj koji se njime bavi u pozadini. Ukoliko postoji potreba za proverom validnosti dodeljene vrednosti ili postoji poseban način za njeno čuvanje, potrebno je definisati puno svojstvo, gde su definisane **get** metoda, **set** metoda, ili obe.

U našem primeru, **First** i **Second** svojstva imaju unikatan izvor, što zahteva punu implementaciju **get** metode. Ova svojstva čitaju vrednosti sa određene pozicije niza, pri čemu metoda **GetNumber** prethodno određuje na koju poziciju da upiše koji broj.

Svojstva omogućavaju ograđivanje unutrašnjih operacija klase, tako da postoji sloboda u izmeni implementacije, bez uništavanja interfejsa za korisnike klase.

Sledeći primer pokazuje kako korisnički kod može upotrebiti nove metode **Calculator4** klase.

```
using System;

class Program
{
    static void Main()
    {
        var calc4 = new Calculator4();

        calc4.GetNumber("First");
        calc4.GetNumber("Second");

        calc4.AddNumbers();

        PrintResult(calc4);

        Console.ReadKey();
    }

    static void PrintResult(Calculator4 calc)
    {
        Console.WriteLine($"Your result is {result}.");
    }
}
```

U ovom primeru, **Main** metoda kreira novu instancu **Calculator4** klase i poziva njene javne(**public)** metode. Sva unutrašnjost **Calculator4** klase je sakrivena i **Main** se interesuje samo za javni interfejs, koji izlaže **Calculator4** usluge. Metoda **PrintResult** čita vrednost **Result** svojstva nove instance **Calculator4** klase. Ono što vredi ponovo napomenuti, jeste da korisnik može da koristi klasu, bez brige o tome kako klasa obavlja svoj posao.

Rukovanje izuzecima

C# omogućava rad u situacijama u kojima neke metode nisu u stanju da ispune ono za šta su namenjene. Sintaksa koja se koristi u ovim situacijama jeste **try-catch** blok. Sav kod koji se nadgleda u potrazi za izuzecima(*eng. exceptions*) se smešta u **try** blok, dok se kod koji obrađuje potencijalne izuzetke smešta u **catch** blok. Naredni kod prikazuje primer ove sintakse.

```
static void HandleNullReference()
{
    Program prog = null;

    try
    {
        Console.WriteLine(prog.ToString());
    }
    catch (NullReferenceException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Svaki pokušaj upotrebe nekog člana **null** objekta izbacuje(*eng. throws)* **NullReferenceException**, što se može rešiti dodeljivanjem vrednosti problematičnoj promenljivoj. U prethodnom primeru izaziva se **NullReferenceException** unutar **try** bloka, pozivanjem **ToString** metode nad **prog** promenljivom, koja ima **null** vrednost.

Kako je deo koda koji izbacuje izuzetak unutar **try** bloka, izvršavanje koda unutar **try** bloka se zaustavlja i kreće se u potragu za rukovaocem(*eng. handler*) tog izuzetka. Parametar **catch** bloka naznačava da on može da uhvati(*eng. catch*) **NullReferenceException** ukoliko kod unutar **try** bloka izbaci izuzetak tog tipa. Telo **catch** bloka predstavlja mesto za obradu uhvaćenih izuzetaka.

Obrada izuzetaka može da podrazumeva i nekoliko **catch** blokova:

```
static void HandleUncaughtException()
{
    Program prog = null;

    try
    {
        Console.WriteLine(prog.ToString());
    }
    catch (ArgumentNullException ex)
    {
        Console.WriteLine("From ArgumentNullException: " + ex.Message);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("From ArgumentException: " + ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine("From Exception: " + ex.Message);
    }
    finally
    {
        Console.WriteLine("Finally always executes.");
    }
}
```

Metoda iz prethodnog primera se naziva **HandleUncaughtException**, jer nema određenog **catch** bloka za obradu **NullReferenceException** tipa. Ovaj izuzetak se obrađuje od strane **catch** bloka za **Exception** tip.

Izuzeci se nabrajaju prema hijerarhiji nasleđivanja, prilikom čega se izuzeci najvišeg nivoa postavljaju najniže u listi **catch** blokova. Izbačeni izuzetak će prolaziti kroz ovu listu rukovalaca, tražeći podudarajući tip izuzetka, i izvršiće se tek kada naiđe na rukovaoca koji to zadovoljava. U prethodnom primeru, **ArgumentNullException** tip je izveden iz **ArgumentException** tipa, a on je dalje izveden iz **Exception** tipa.

Ukoliko nijedan **catch** blok ne može da obradi izuzetak, kod ide uz stek tragajući za potencijalnim **catch** blokom unutar pozivajućeg koda. Ukoliko ni na ovaj način nije pronađen odgovarajući **catch** blok, izvršavanje koda se prekida.

Blok **finally** se uvek izvršava ukoliko počne izvršavanje koda unutar **try** bloka. Ukoliko se pojavi izuzetak i ne bude uhvaćen, **finally** blok će se izvršiti pre nego što program pređe na pozivajući kod u steku, u potrazi za odgovarajućim rukovaocem.

Moguće je napisati i **try-finally** blok (bez **catch** bloka), kako bi se garantovalo izvršavanje određenog koda, ako se krene sa izvršavanjem **try** bloka. Ovo je korisno prilikom otvaranja određenih resursa, jer se na ovaj način može garantovati njihovo zatvaranje, bez obzira na izuzetke.

Unutar **.NET FCL-a** postoje mnogi tipovi izuzetaka koji se mogu koristiti. Naredni primer objedinjuje nekoliko korisnih koncepata, poput validacije ulaznih parametara i izbacivanja **ArgumentNullException** izuzetka.

```
public class Address
{
    public string City { get; set; }
}

internal class Company
{
    public Address Address { get; set; }
}

        // Inside of a class...
        static void ThrowException()
        {
            try
            {
                ValidateInput("something", new Company());
            }
            catch (ArgumentNullException ex) when (ex.ParamName == "inputString")
            {
                Console.WriteLine("From ArgumentNullException: " + ex.Message);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("From ArgumentException: " + ex.Message);
            }
        }

        static void ValidateInput(string inputString, Company cmp)
        {
            if (inputString == null)
                throw new ArgumentNullException(nameof(inputString));

            if (cmp?.Address?.City == null)
                throw new ArgumentNullException(nameof(cmp));
        }
```

Prethodni kod prikazuje **Address** klasu, kao i **Company** klasu sa svojstvom **Address** tipa. Unutar **try** bloka **ThrowException** metode se prosleđuje nova instanca **Company** klase, ali se ne instancira **Address** svojstvo, odnosno ono ima **null** vrednost.

Unutar **ValidateInput** metode, **if** iskaz koristi operator **?.** kako bi proverio da li je bilo koja vrednost između kompanije **cmp**, njenog **Address** svojstva i **City** svojstva unutar tog **Address** svojstva **null**. Ukoliko je bilo koja od vrednosti **null**, kod izbacuje **ArgumentNullException**.

Prilikom prosleđivanja argumenta **ArgumentNullException** izuzetku, koristi se **nameof** operator, koji vraća ime varijable, tipa ili člana u vidu string konstante (u ovom slučaju **"cmp"**). Ovaj kod se ne nalazi unutar **try** bloka, tako da se vraćamo na kod koji poziva ovu metodu.

Nazad u **ThrowException** metodi, izbačeni izuzetak tera kod da traži rukovaoca odgovarajućeg tipa. Tip izuzetka je **ArgumentNullException**, ali se **catch** blok za **ArgumentNullException** neće izvršiti. To se dešava zato što **when** klauzula, koja prati **ArgumentNullException catch** blok parametar, traži **ParamName** koje ima vrednost **"inputString"**. Ova **when** klauzula se naziva filterom izuzetaka. Kao što je već pomenuto, ime parametra koje je prosleđeno **ArgumentNullException** izuzetku ima vrednost **"cmp"**, tako da ne postoji podudaranje. Zbog ovoga, kod nastavlja potragu za **catch** rukovaocima.

Kako je **ArgumentNullException** tip izveden iz **ArgumentException** tipa, za kojeg nema nikakvih filtera izuzetaka u kodu, **catch** rukovalac za **ArgumentException** se izvršava i izuzetak se obrađuje.