Osnove programskog jezika C#

C# programski jezik i .Net framework

C# je objektno-orijentisani programski jezik opšte namene. Osnovni cilj C# je povećati produktivnost programera. .Net framework je oslonjen na arhitekturu koja deli framework na dva dela: *Common Language Runtime* (CLR) i *Framework Class Library* (FCL).

CLR vodi računa o zadacima koji su vezani za izvršavanje koda: kompajliranje, upravljanje memorijom, sigurnosti, upravljanje nitima, i kontrola tipovA podataka. Kod koji se izvršava u CLR se naziva upravljani (eng. *managed*) kod. Nasuprot tome, neupravljani (eng. *unmanaged*) je kod koji ne implementira zahteve za rad u .Net framework-u —kao što je COM ili Windows API.

Programski prevodioci koji su kompatiblini sa CLR-on proizvode kod pod imenom *Intermediate Language* (IL), koji je pogodan za izvršavanje u runtime-u. Ovaj kod omogućava integraciju prevodilaca različitih programskih jezika.

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.

Konstrukcija C# programa

Sva programska logika i podaci moraju biti ubačeni u klase, strukture, enumeracije, intefejse ili delegate. Za razliku od nekih drugih programskih jezika, C# nema globalne promenjive koje bi postojale van navedenih tipova podataka. Pristup tipovima i njihovim podacima je striktno kontrolisano.

```
using System;

class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Metod *main* je obavezan za svaku izvršnu C# aplikaciju. Ovaj metod uvek je tipa *static* i služi kao ulazna tačka za aplikaciju, Metod main može imati ulazne parameter i povratnu vrednost.

Deklaracije i definicije

Deklaracija definiše osnovne atribute simbola: njegov tip i ime. Definicija pruža informacije o implementaciji simbola, za funkcije šta rade, za klase koja polja i metode imaju, za varijable gde su smeštene.

```
List<int> students;                         //field declaration
List<int> students = new List<int>();       //field definition
```

Prostori imena

Prostor imena (eng. *Namespace*) je domen u kome imena tipova moraju biti jedinstvena. Tipovi su obično organizovani u vidu hijerarhije prostora imena, sa ciljem da se izbegnu konflikti između imena i da se omogući lakše pronalaženje imena tipa. Korišćenjem tačke prikaz hijerarhije u prostoru imena moguće je pojednostaviti.

```
namespace Outer
{
   namespace Middle
   {
      namespace Inner
      {
         class Dog {}
      }
   }
}
```

```
namespace Outer.Middle.Inner
{
   class Dog {}
}
```

Da bi se koristio neki prostor imena potrebno ga je importovati sa **using** direktivom. U tom slučaju moguće je prilikom deklarisanja navesti samo ime tipa umesto navođenja imena tipa sa kompletnom hijerarhijom.

```
using Outer.Middle.Inner;

Dog mydog = new Dog("Laika");
// same as Outer.Middle.Inner.Dog mydog = new Outer.Middle.Inner.Dog("Laika");
```

Komentari

U C# postoje dve vrste komentara:

1. Komentari u jednom redu

```
int x = 3;      // Comment about assigning 3 to x
```

2. Komentari u više redova

```
int x = 3;      /* This is a comment that
                spans two lines */
```

Kontrola toka

Blok predstavlja skup naredbi koje se ograničene {} zagradama. Upravljanje tokom naredbi vrši se pomoću naredbi selekcije, iteracije i skokova.

Tip naredbe	Naredba	Namena	Primer

Naredbe selekcije	if	U zavisnosti od postavljenog uslova bira se odgovarajući blok naredbi	`if (a > b)`
			`{`
			`  Console.WriteLine("greater");`
			`}`
			`else`
			`{`
			`  Console.WriteLine("less or equal");`
			`}`

	switch	U zavisnosti od mogućih vrednosti koje ulazna promenljiva može imati bira se odgovarajući blok naredbi	`switch (coffeeSize)`
			`{`
			`  case "1":`
			`  case "small":`
			`  cost += 25;`
			`  break;`
			`  case "2":`
			`  case "medium":`
			`  cost += 25;`
			`  goto case "1";`
			`  case "3":`
			`  case "large":`
			`  cost += 50;`
			`  goto case "1";`
			`  default:`
			`  break;`
			`}`

Naredbe iteracije	while	Petlja sa nepoznatim brojem iteracija. Blok naredbi se izvršava dokle god je uslov zadovoljen.	`while (a < b)`
			`{`
			`  Console.WriteLine("value: {0}", a);`
			`  n++;`
			`}`

	for	Petlja sa poznatim brojem iteracija. Najčešće se koristi za iteraciju kroz sve članove niza.	`for (int i = 0; i < 3; i++)`
			`{`
			`  a[i] = 2 * (i + 1);`
			`}`

	foreach	Petlja koja omogućava iteraciju kroz sve članove kolekcije ili niza koji implementiraju *System.Collections.Enumerable*	`foreach (char c in "beer")`
			`{`
			`  Console.WriteLine (c);`
			`}`

Naredbe skoka	break	Prekida izvršavanje petlje ili switch selekcije	`for(int i=0; i<10; i++)`
			`{`
			`  if (x++ > 5)`
			`  break ;`
			`}`

	continue	Prenosi izvršavanje na prvu narebu iduće iteracije	`for (int i = 0; i < 10; i++)`
			`{`
			`  if ((i % 2) == 0) // If i is even,`
			`  {`
			`  continue; // go to next iteration`
			`  }`
			`  Console.Write (i + " ");`
			`}`
			`OUTPUT: 1 3 5 7 9`

	goto	Bezuslovan skok na naredbu zadatu labelom	`int i = 1;`
			`startLoop:`
			`if (i <= 5)`
			`{`
			`  Console.Write (i + " ");`
			`  i++;`

			goto startLoop; } OUTPUT: 1 2 3 4 5

	return	Završava izvršavanje metode i vraća odgovarajuću povratnu vrednost	static decimal AsPercentage (decimal d)
			{
			decimal p = d * 100m;
			return p;
			}

	throw	Ukazuje da se desila greška u programu.	throw new ArithmeticException ("reason");

Tipovi podataka

Sistem tipova u C# je sjedninjen, jer svi tipovi nasleđuju klasu *System.Object*. Osnovna podela tipova je na proste tipove i složene tipove, koji su izvedeni iz prostih tipova.

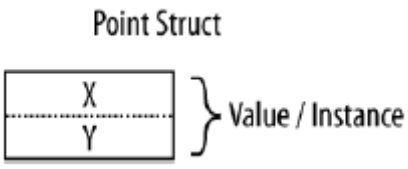
Tip	Ključne reči	Dužina [byte]	Primer
Numerički	sbyte	8	sbyte val = -9;
	short	16	short val = -9;
	int	32	int val = -9;
	long	64	long val = -9l;
	byte	8	byte val = 9;
	ushort	16	ushort val = 9;
	uint	32	uint val = 9u;
	ulong	64	ulong val = 9ul
	float	32	float val = 4.5f;
	double	64	double val = 4.5d;
	decimal	128	decimal val = 4.5m;
Logički	bool	8	bool val = true;
Karakter	char	8	char val = 'A';
Tekstualni	string	(n.+1)* 8	string val = "Hello" + "world";

Vrednosni i referentni tipovi

Osnovna razlika između vrednosnih i referentnih tipova je u načinu na koji ti tipovi rukuju memorijom.

Vrednosni tipovi

Sadržaj promenljive ili konstante vrednosnog tipa je sama vrednost.



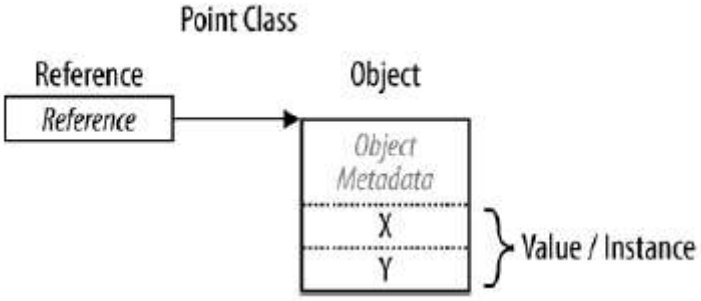
Dodeljivanje instance vrednosnog tipa realizuje se kopiranjem same instance.

```
public struct Point { public int X, Y; }

static void Main()
{
        Point p1 = new Point();
        p1.X = 7;
        Point p2 = p1;                // Assignment causes COPY
        Console.WriteLine (p1.X);     // 7
        Console.WriteLine (p2.X);     // 7
        p1.X = 9;                     // Change p1.X
        Console.WriteLine (p1.X);     // 9
        Console.WriteLine (p2.X);     // 7
}
```

Referentni tipovi

Referentni tipovi zahtevaju posebno alociranje memorije za objekat i za referencu na taj objekat. Sadržaj promenljive ili konstante referentnog tipa je referenca na objekat koji sadrži vrednost.



Dodeljivanje instance referentnog tipa predstavlja kopiranje reference, a ne objekta na koji referenca pokazuje. Time se otvara mogućnost da više referenci pokazuju na isti objekat.

```
Point p1 = new Point();
p1.X = 7;
Point p2 = p1;                // Copies p1 reference
Console.WriteLine (p1.X);     // 7
Console.WriteLine (p2.X);     // 7
p1.X = 9;                     // Change p1.X
Console.WriteLine (p1.X);     // 9
Console.WriteLine (p2.X);     // 9
```

Vrednosni tipovi	Referentni tipovi
brojevi	stringovi
logički tip	nizovi
karakteri	klase
enumeracije	interfejsi
strukture	delegati

Instance i statički tipovi

Osnovna razlika između instance i statičkog tipa je što se instanca vezuje za konkretan objekat neke klase, dok se statički tip vezuje za sam tip klase. Modifikator **static** može se koristiti za klase, polja, metode, svojstva..

```
public class Panda
{
        public string Name; // Instance field
        public static int Population; // Static field

        public Panda (string n) // Constructor
        {
                Name = n; // Assign the instance field
                Population = Population + 1; // Increment the static Population field
        }
}

class Program
{
        static void Main()
        {
                Panda p1 = new Panda ("Pan Dee");
                Panda p2 = new Panda ("Pan Dah");
                Console.WriteLine (p1.Name); // Pan Dee
                Console.WriteLine (p2.Name); // Pan Dah
                Console.WriteLine (Panda.Population); // 2
        }
}
```

Implicitni tip lokalne promenljive

U c# postoji mogućnost da se ne navede eksplicitno tip promenljive u slučaju da se promenljiva u istom koraku i deklariše i definiše.

```
var x = "hello";
```

Default vrednosti

Sve instance tipova imaju svoje podrazumevane vrednosti.

Tip	default vrednost
referentni tipovi	null
brojevi i enumeracije	0
karakteri	'\0'
logički tip	false

Koristeći ključnu reč *default* moguće je dobiti default vrednost za zadati tip.

```
decimal d = default (decimal);
```

1.1. Operatori

Operatori poređenja

Poređenje brojeva implementirano je preko oepratora ==, !=, <, >, >=, i <=.

```
Int x = 5;
Console.WriteLine (x <= 7+1); // True
```

Vrednosni tipovi se porede po vrednosti:

```
int x = 1;
int y = 1;
Console.WriteLine (x == y);   // OUTPUT: true
```

Kod referentnih tipova poredi se referenca objekta na koji pokazuju, a ne vrednosti polja samog objekta.

```
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2); // OUTPUT: false
Dude d3 = d1;
Console.WriteLine (d1 == d3); // OUTPUT: true
```

Operatori inkrementa i dekrementa

Operatori inkrementa (dekrementa) omogućavaju uvećanje (umanjenje) vrednosti promenljive za jedan. U zavisnosti od potrebe, vrednost promenljive može biti osvežena pre ili posle izvršenja naredbe.

```
int x = 0;
Console.WriteLine (x++); // Outputs 0; x is now 1
Console.WriteLine (++x); // Outputs 2; x is now 2
```

Operatori uslova

Operator	Značenje
!	negacija
&&	i
||	ili

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
        return !windy && (rainy || sunny);
}
```

Operatori sa bitima

Operator	Značenje	Primer	
~	komplement	~0xfu	0xfffffff0u
&	i	0xf0 & 0x33	0x30
|	ili	0xf0 | 0x33	0xf3
^	eksluzivno ili	0xff00 ^ 0x0ff0	0xf0f0
<<	pomeranje u levo	0x20 << 2	0x80
>>	pomeranje u desno	0x20 >> 1	0x10

Nizovi

Nizovi predstavljaju fiksan broj elemenata određenog tipa. Elementi niza se čuvaju u memoriji po redosledu, čime se omogućava visoka efikasnost pristupa.

```
char[] vowels = new char[5]; // Declare an array of 5 characters
char[] vowels = new char[] {'a','e','i','o','u'};   // An array initialization
```

1.2. Višedimenzionalni nizovi

Postoje dva tipa višedimenezionalnih nizova u c#: pravougani i nazubljeni.

Pravougaoni nizovi (eng. *rectangular arrays*) predstavljaju n-dimenzionalne blokove memorije.

```
int[,] rectangularMatrix2 =
{
        {0,1,2},
        {3,4,5},
        {6,7,8}
};
```

Nazubljeni nizovi (eng. *jagged arrays*) prestavljaju nizove nizova. Za razliku od pravougaonih nizova, svaki unutrašnji niz može imati proizvoljan broj članova.

```
int[][] jaggedMatrix =
{
        new int[] {0,1,2},
        new int[] {3,4,5},
        new int[] {6,7,8}
};
```

Klase

Klase omogućavaju kreiranje izvedenog tipa podataka grupisanjem varijabli, svojstava i metoda. Definicija klase u C# ima sledeće elemente:

1. **Atributi –** opcioni elementi koji dodatno opisuju klasu
2. **Deklaracija** – definiše ime klase, prava pristupa, navodi ime klase čije osobine nasleđuje i interfejsi koje implementira
3. **Polje** (eng. *field)* – promenljiva ili konstanta
4. **Svojstvo** (eng. *property*) – kontroliše prava pristupa polju klase
5. **Metoda** - predstavlja funkciju koja se može pozvati nad objektom klase.
6. **Kontruktor** – služi za inicijalizaciju klase.

```
[Serializable()]                              // attribute
public class Parking:                         // declaration
{
        private uint name = "SNP Parking";    // constant field
        private uint cars;                    // variable field

        public uint Cars                      // property
        {
                get{return cars;}};
                private set{ cars = value}};

        }

        public Parking () {}                  // contructors
        public Parking (uint numOfCars){ cars = numOfCars}

        // Methods
        void EmptyParking(){cars = 0};        // methods

}
```

Referenca **this** predstavlja samu instancu određene klase.

```
public class Panda
{
        public Panda Mate;
        public void Marry (Panda partner)
        {
                Mate = partner;
                partner.Mate = this;
        }
}
```

Metode

Metoda predstavlja blok koji sadrži seriju naredbi. Parametri metode definišu set argumenata koji bi trebalo da bude obezbeđen za tu metodu. U C# funkcija je mnogo širi pojam od metode i pored metode ona označava i svojstva, konstruktore, događaje, operatore...

U okviru jedne klase više metoda može imati isto ime, ali mora imati različiti skup tipova parametara pri pozivu.

```
void Add (double x, double y, double z);
void Add (int x, int y);
void Add (float x, float t y);
```

Prosleđivanje argumenata po vrednosti

Ukoliko se eksplicitno drugačije ne navede, argumenti se u C# prosleđuju po vrednosti.

```
class Test
{
static void Foo (int p)
{
        p = p + 1; // Increment p by 1
        Console.WriteLine (p); // Write p to screen
}
        static void Main()
        {
                int x = 8;
                Foo (x); // Make a copy of x
                Console.WriteLine (x); // x will still be 8
        }
}
```

Prosleđivanjem po vrednosti argumenta koji je tip reference, kopira se referenca, ali ne i objekat.

```
class Test
{
        static void Foo (StringBuilder fooSB)
        {
                fooSB.Append ("test");
                fooSB = null;
        }
        static void Main()
        {
                StringBuilder sb = new StringBuilder();
                Foo (sb);
                Console.WriteLine (sb.ToString()); // test
        }
}
```

Prosleđivanje parametra po referenci

Za prosleđivanje parametra po referenci koristi se ključna reč *ref*.

```
class Test
{
        static void Foo (ref int p)
        {
                p = p + 1; // Increment p by 1
                Console.WriteLine (p); // Write p to screen
        }

        static void Main()
        {
                int x = 8;
                Foo (ref x);          // Ask Foo to deal directly with x
                Console.WriteLine (x); // x is now 9
        }
}
```

Prosleđivanje izlaznih parametara

Ovaj tip prosleđivanja parametra najčešće se koristi u slučaju kada je potrebno više vrednosti vratiti kao izlaz metode. Prosleđivanje izlaznih parametara slično je kao i prosleđivanje parametara po referenci, osim što parametrima nije potrebno dodeliti vrednost pre poziva funkcije.

```
class Test
{
        static void Split (string name, out string firstNames, out string lastName)
        {
                int i = name.LastIndexOf (' ');
                firstNames = name.Substring (0, i);
                lastName = name.Substring (i + 1);
        }
        static void Main()
        {
                string a, b;
                Split ("Stevie Ray Vaughn", out a, out b);
                Console.WriteLine (a);        // Stevie Ray
                Console.WriteLine (b);        // Vaughn
        }
}
```

Promenljiv broj parametara

Ukoliko broj parametara funkcije koja se poziva nije unapred poznat, moguće je funkciju definisati tako da poslednji parametar metode očekuje bilo koji broj parametara određenog tipa.

```
class Test
{
        static int Sum (params int[] ints)
        {
                int sum = 0;
                for (int i = 0; i < ints.Length; i++)
                        sum += ints[i];               // Increase sum by ints[i]
                return sum;
        }
        static void Main()
        {
                int total = Sum (1, 2, 3, 4);
                Console.WriteLine (total);    // 10
        }
}
```

Opcioni parametri

Ukoliko se neki parametar metode definiše kao opcioni, moguće je metodu pozvati bez navedenog parametra. U tom slučaju parametar će dobiti vrednost koja je definisana kao podrazumevana.

```
void Foo (int x = 23) { Console.WriteLine (x); }

Foo(); // 23
```

Modifikatori pristupa

Prava pristupa regulišu se korišćenjem modifikatora pristupa:

1. **public** – javni pristup bez ograničenja
2. **protected** – zaštićeni pristup, samo članovi klase i članovi izvedenih klasa
3. **private** – privatni pristup, samo članovi klase
4. **internal** – interni pristup, samo elementi u okviru asemblerskog sklopa

Nasleđivanje

Nasleđivanjem klase omogućava se korišćenje funkcionalnosti klase koja se nasleđuje uz mogućnost proširenja i prilagođavanja njene funkcionalnosti potrebama korisnika, umesto da se funkcionalnost klase implementira od početka.

Klasa može naslediti samo jednu klasu, ali može biti nasleđena od više klasa, čime se formira hijerarhija klasa u vidu stabla.

```
public class Node
{
        public string lid;
        public Node(lid){this.lid = lid;}
}

public class BusNode : Node
{
        byte phases;
        BusNode(uint lid, byte phases) : base(lid){this.phases = phases};
}
```

Ključna reč **base** omogućava pristup kontruktoru bazne klase.

Virtuelne metode

Metode koje su obeležene sa ključnom reči *virtual* mogu biti prepisane (eng. *overridden*) od izvedenih klasa sa ciljem da se napiše specijalizovanija implementacija.

```
public class Asset
{
        public string Name;
        public virtual decimal Liability { get { return 0; } }
}

public class House : Asset
{
        public decimal Mortgage;
        public override decimal Liability { get { return Mortgage; } }
}
```

Operator **new** omogućava izvedenoj klasi da ima isto polje ili metodu kao i roditeljska klasa.

```
public class A { public int Counter = 1; }
public class B : A { public new int Counter = 2; }   //hide parent's Counter
```

Pomoću klučne reči **sealed** moguće je izvedenim klasama zabraniti menjanje nekog člana metode.

```
public sealed override decimal Liability { get { return Mortgage; } }
```

Apstraktne klase

Klase koje su deklarisane kao apstraktne ne mogu se instancirati, već se moraju implementirati u izvedenoj klasi.

```
public abstract class Asset
{
        // Note empty implementation
        public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
        public long SharesOwned;
        public decimal CurrentPrice;

        // Override like a virtual method.
        public override decimal NetValue
        {
                get { return CurrentPrice * SharesOwned; }
        }
}
```

Generički tipovi

Generički tipovi klasa omogućavaju pisanje koda koji će moći koristiti više različitih tipova.

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T> // A simple Stack implementation
{
        int position;
        T[] data = new T[100];
        public void Push (T obj) { data[position++] = obj; }
        public T Pop() { return data[--position]; }
}
```

```
Stack<Bear> bears = new Stack<Bear>();
```

Interfejsi

Za razliku od klasa, interfejsi sadrže samo specifikaciju svojih članova. Klasa ili struktura koja implementira interfejs mora implementirati sve članove koji su specificirani definicijom interfejsa.

```
public interface IEnumerator
{
        object Current { get; }
        bool MoveNext();
        void Reset();
}
```

Interfejs može biti izveden iz nekog drugog interfejsa.

```
public interface IUndoable { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

Implementiranje više interfejsa može dovesti do kolizije između imena članova. U tom slučaju potrebno je prilikom implementacije člana eksplicitno definisati kojem iterfejsu član pripada.

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
        public void Foo ()
        {
                Console.WriteLine ("Widget's implementation of I1.Foo");
        }
        int I2.Foo()
        {
                Console.WriteLine ("Widget's implementation of I2.Foo");
                return 42;
        }
}
```

1.3. Strukture

Struktura je vrednosni tip koja se najčešće koristi da se grupišu promenljive koje su na neki način logički povezane. Pored polja, strukture mogu da sadrže i svojstva, metode, operatore, ali se ova mogućnost vrlo retko koristi.

```
public struct Point
{
        int x, y;
        public Point (int x, int y) { this.x = x; this.y = y; }
}
…
Point p2 = new Point (1, 1); // p1.x and p1.y will be 1
```

Enumeracije

Enumeracije su vrednosni tipovi koji omogućuju specifikaciju grupe imenovanih numeričkih konstanti.

```
public enum BorderSide { Left, Right, Top, Bottom }
// public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }

BorderSide topSide = BorderSide.Top;
```

Pomoću eksplicitnog kastovanja moguće je konverzija između vrednosti enumeracije i celobrojnih vrednosti.

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
```

Konverzije

C# dozvoljava konverziju između kompatibilnih tipova. Konverzija može biti *implicitna* ukoliko se dešava automatski, ili *eksplicitna* ukoliko se koristi kastovanje (eng. *cast*).

Kod konverzije vrednosti kreira se nova vrednost iz postojeće.

```
int x = 12345;         // int is a 32-bit integer
long y = x;            // Implicit conversion to 64-bit integer
short z = (short)x;    // Explicit conversion to 16-bit integer
```

Kod konverzije referenci kreira se nova referenca koja pokazuje na isti objekat.

```
Dog dog = new Dog ();
Animal animal = dog;          // Upcast
Dog mydog = (Dog)animal;      // Downcast
```

Takođe, moguće je izvršiti konverziju između klase i interfejsa koji klasa implementira.

Izuzeci

Mehanizam izuzetaka (eng. *exceptions*) omogućava rukovanje očekivanim i neočekivanim situacijama, koje se mogu dogoditi u toku izvršavanja programa. Izuzeci se predstavljaju klasama koje su izvedene iz klase Exception.

Blokovi naredbi za rukovanje izuzecima:

1. **try** – predstavlja blok naredbi koji je potrebno kontrolisano izvršiti
2. **catch** – blok naredbi koji se izvršava u slučaju detektovanja izuzetka. Svaki catch blok definisan je za određeni tip izuzetka. Ukoliko postoji više catch blokova, važno je prvo poslagati izvedene izuzetke pa tek onda bazne.
3. **finally** – blok naredbi koji se izvršava bez obzira da li se izuzetak dogodio ili ne. Ovaj blok služi da se kod vrati u prvobitni oblik.

```
try
{
        reader = File.OpenText ("file.txt");
        …
}
catch (FileNotFoundException ex)
{
        Console.WriteLine ("File not found. ");
}
catch (Exception ex)
{
        Console.WriteLine (ex.message);
}
finally
{
        if (reader != null) reader.Dispose();
}
Console.WriteLine ("Done.");
```

Izuzeci mogu biti generisani od strane CLR izvršnog okruženja, od nekih spoljašnjih biblioteka ili preko koda aplikacije korišćenjem **throw** ključne reči

```
private static void TestThrow()
{
    System.ApplicationException ex =
        new System.ApplicationException("Demonstration exception in TestThrow()");

    throw ex;
}
```

Izuzetak se može proslediti i njime rukovati u nekom drugom delu programa.

```
Try
{
...
}
catch (Exception ex)
{
        // Log error
        ...
        throw; // Rethrow same exception
}
```

'using' naredba

Naredba using omogućava eksplicitno upravljanje oslobađanjem resursa, ukoliko klasa koja je zauzela resurs implementira interfejs *IDisposable*

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
...
}
```

U nastavku je dat primer koji je ekvivalentan prethodnom, a napisan je pomoću try-catch blokova.

```
StreamReader reader = File.OpenText ("file.txt");
try
{
...
}
finally
{
        if (reader != null)
        {
                ((IDisposable)reader).Dispose();
        }
}
```

Delegati

Delegat je tip koji predstavlja referencu na metodu sa određenom listom parametara i povratnom vrednosti.

```
delegate int NumberChanger(int n);

NumberChanger nc1 = new NumberChanger(AddNum);
NumberChanger nc2 = new NumberChanger(MultNum);

public static int AddNum(int p)
{
        num += p;
        num;
}

public static int MultNum(int q)
{
       num *= q;
       return num;
}
```

Događaji (eng. *Events*)

Mehanizam događaja koristi delegate za sistem obaveštenja, koje slušaoci registruju kod izvora događaja. Izvor događaja izvršava metode svih registrovanih delegata kada se događaj desi.

```
using System;

public delegate void EventHandler();

class Program
{
    public static event EventHandler _show;

    static void Main()
    {
        // Add event handlers to Show event.
        _show += new EventHandler(Dog);
        _show += new EventHandler(Cat);

        // Invoke the event.
        _show.Invoke();
    }

    static void Cat()
    {
        Console.WriteLine("Cat");
    }

    static void Dog()
    {
        Console.WriteLine("Dog");
    }
}

OUTPUT:
Dog
Cat
```

Kolekcije

.Net okruženje obezbeđuje stadardni set tipova za čuvanje i upravljanje kolekcijama objekata.

Lista

Lista predstavlja dinamički niz objekata kojima je moguće pristupati prema indeksu.

```
List<string> words = new List<string>();
```

Osnovne operacije:

a) Dodavanje novog elementa u listu
```
words.Add ("melon");
```

b) Dodavanje novog elementa u listu na poziciju definisanu indeksom
```
words.Insert (0, "lemon");
```

c) Uklanjanje elementa iz liste preko njegove vrednosti
```
words.Remove ("melon");
```

d) Uklanjanje elementa iz liste preko njegovog indeksa
```
words.RemoveAt (3); // Remove the 4th element
```

e) Čitanje elemenata iz liste

```
var word = words [index];
```

f) Promena vrednosti elementu iz liste

```
words [index] = "melon"
```

g) Iteracija kroz članove liste

```
foreach (string word in words)
   Console.WriteLine (word);
```

h) Provera broja elemenata u listi

```
words.Count
```

Rečnik (eng. *Dictionary*)

Rečnik predstavlja kolekciju u kojoj je svaki elemenat predstavljen parom *ključ-vrednost*.

```
var dict = new Dictionary<int,string>();
```

Osnovne operacije:

a) Dodavanje novih elemenata u rečnik

```
dict.Add(key, value);
```

b) Promena vrednosti postojećem element rečnika

```
dict[key] = value;
```

c) Brisanje postojećeg elementa rečnika

```
dict.Remove(key);
```

d) Čitanje vrednosti iz rečnika

```
var value = dict[key];
```

e) Provera da li ključ postoji u rečniku

```
if(dict.ContainsKey(key))
...
```

f) Provera da li vrednost postoji u rečniku

```
if(dict.ContainsValue(value))
...
```

g) Dobavljanje vrednosti iz rečnika ukoliko ključ postoji

```
if(dict.TryGetValue (key, out value))
   Console.WriteLine ("Key not found.");
```

h) Iteracija kroz parove ključ-vrednost

```
foreach (KeyValuePair<int, string > kv in dict)
   Console.WriteLine (kv.Key + ":" + kv.Value);
```

i) Iteracija kroz ključeve

```
foreach (string key in dict.Keys)
   Console.WriteLine(key);
```

j) Iteracija kroz vrednosti

```
foreach (int value in dict.Values)
   Console.WriteLine(value);
```

i) Provera broja elemenata u rečniku

```
dict.Count
```

Zadatak

Napisati program koji implementira bankomat za podizanje novca. Za isplatu koristiti novčanice od 10, 20, 50, 100, 200, 500, 1000, 2000 i 5000 dinara. Prilikom isplate traženog iznosa novca trebalo bi voditi računa o stanju na računu klijenta i raspoloživom broju novčanica u automatu.