

Razvoj primijenjene programske potpore

3. Kratki pregled posebnosti platforme .NET (Core) i C#-a

Nadogradnja na OOP u Javi

.NET Framework

- Microsoft .NET Framework
 - nastao s idejom iste osnovice za izradu lokalnih i Internet aplikacija
 - začetak krajem 90-tih, prva verzija 2002. g
 - zadnja verzija 4.8, daljnji razvoj obustavljen
 - neovisnost o jeziku (C++, C#, Visual Basic .NET, F#, ...) i neovisnost o platformi
 - .. Sve dok platforma podržava Common Language Runtime (CLR)
- Suprotno zamisli, .NET je namijenjen uglavnom Windows platformi
 - Mono – (djelomična) implementacija .NET Frameworka za Linux

.NET (.NET Core)

- Open source varijanta nastala 2014. godine
 - nije podskup .NET Frameworka, ali dijelio dio funkcionalnosti propisane formalnom specifikacijom API-a (.NET Standard)
 - Podjela na modularne pakete dohvatljive korištenjem alata NuGet
- Dostupan za Windowse, OSX i razne distribucije Linuxa
 - CoreCLR
 - .NET 6 (preciznije 6.0.10, SDK 6.0.402), listopad 2022.
 - Od verzije 5 umjesto naziva .NET Core, koristi se naziv .NET i napušta koncept .NET standarda
- C#
 - Razvoj započet 1999., prva verzija 2002.
 - trenutna verzija C# 10.0 (studenj 2021.)

.NET status i budućnost

- Što trenutno može .NET Core?
 - konzolne aplikacije
 - web aplikacije i web servisi pisani u ASP.NET Core-u
 - objektno-relacijsko preslikavanje prema nekoliko tipova sustava za upravljanjem bazama podataka korištenjem alata Entity Framework Core
- Samostalne (desktop i mobilne) aplikacije za
 - Android, iOS, macOS, Windows, Tizen
 - .NET MAUI (.NET Multi-platform App UI)

Osnovni pojmovi

- IL – Intermediate language – jezik u kojeg se kompiliraju viši jezici
- CLR (Common language runtime)
 - Virtualno računalo koje izvršava naredbe nastale iz IL-a pretvorbom u strojni jezik (JIT – Just-in-time compiler)
- Common Type System (CTS) definira tipove podataka za jezike koji se mogu pretvoriti (kompajlirati) u IL
 - Klase, strukture, enumeracije, sučelja, delegati, modifikatori pristupa, ...
 - <https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system>
- Common Language System (CLS) – podskup CTS-a prisutan u svim jezicima
 - nazivi tipova u pojedinom jeziku ne moraju biti isti
 - npr int vs integer → int32 u IL-u
 - može postojati neki tip u C#-u koji ne postoji u Visual Basicu
- Base Class Library (BCL) – osnovni skup biblioteka

Stvaranje prvog programa (.NET Core)

- Komanda linija
 - u nekoj mapi pokrenuti `dotnet new console -n Naziv`
 - programski kod urediti u proizvoljnom uređivaču teksta
 - `dotnet restore` (dovlači pakete uključene unutar projekta)
 - `dotnet run`
- Visual Studio Code
 - u nekoj mapi pokrenuti `dotnet new console -n Naziv`
 - Visual Studio Code → Open Folder
 - Potvrdi dohvat paketa, a zatim F5
- Visual Studio 2022
 - File → New project → C# Console Application (NET Core)
 - Unijeti naziv i lokaciju projekta
 - F5 (Debug) ili CTRL+F5 (Execute)

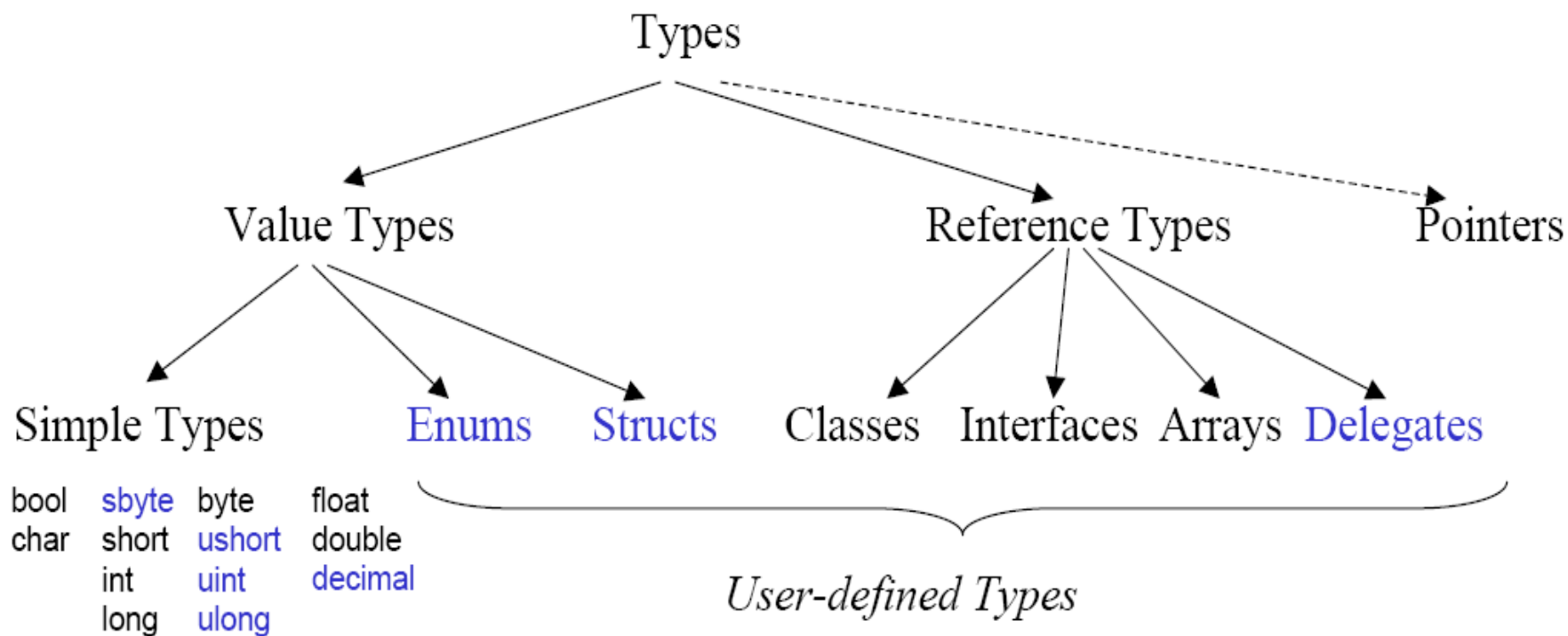
Java -> C# - prvi koraci

- Ime datoteke ne mora odgovarati nazivu klase, ali je poželjno
 - U istoj datoteci može se nalaziti više klasa
 - Ista klasa se može definirati u više datoteka te se takva klasa mora označiti modifikatorom `partial`
- `package` → `namespace` `import` → `using`
- 1 projekt = 1 dll i/ili exe (*assembly*)
- Sintaksa nalik C-u i Javi. Ključne riječi (neke ovisne o kontekstu):
 - <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords>
- Standardni ulaz/izlaz
 - `Console.ReadLine`, `Console.WriteLine` ...
 - Više o formatiranju ispisa: <https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings>
- Pravokutna i nazubljena (jagged) polja

Tipovi podataka

- Svi tipovi izvode se iz osnovnog tipa `System.Object`
 - primitivni tipovi su strukture
 - Moguće npr.: `3.ToString()`; `"abc".ToUpper()`;

<https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system>



Modifikatori vidljivosti

- Modifikatori pristupa razredima i članovima
 - `public` – pristup nije ograničen
 - `private` - pristup ograničen na razred u kojem je član definiran
 - `protected` – pristup ograničen na razred i naslijeđene razrede
 - `private protected` – slično kao `protected`, ali samo za naslijeđene razrede u istom programu/projektu
 - `internal` – pristup ograničen na program u kojem je razred definiran
 - `protected internal` - pristup dozvoljen naslijeđenim razredima (bez obzira gdje su definirani) i svima iz programa u kojem je razred definiran

Pretpostavljeni modifikatori

- Ako modifikator nije naveden onda se smatra da je
 - `internal` za razrede, sučelja, delegate i događaje
 - `private` za članske varijable, svojstva i postupke i ugniježdene razrede
 - `public` za postupke sučelja i članove enumeracija
 - do verzije 8 C#-a nije ni bio dozvoljen drugačiji modifikator
- Izvedeni razred ne može imati veću dostupnost od baznog razreda

Neki od značajnijih modifikatora

- `abstract` – razred može biti samo osnovni razred koji će drugi nasljeđivati
- `const` – atribut (polja) ili lokalna varijabla je konstanta
- `new` – modifikator koji skriva naslijeđenog člana od člana osnovnog razreda
- `readonly` – polje poprima vrijednost samo u deklaraciji ili pri instanciranju (u konstruktoru)
- `sealed` – razred ne može biti naslijeđen
- `static` – član definiran na razini razreda, a ne instance
- `virtual` – postupak ili dostupni član koji može biti nadjačan u naslijeđenom razredu – (prilikom nadjačavanja dodaje se modifikator `override`)


Finalizatori (Destruktori)

- Finalizator je postupak (*Finalize*) koji se automatski poziva neposredno prije uništenja objekta od strane sakupljača smeća (Garbage Collector)
- Piše se nalik destrukturu u C++-u - nazivu razreda prefiksiran s ~
 - Ne vraća vrijednost
 - Koristi se u rijetkim slučajevima za brisanje *unmanaged* resursa (ako su korišteni)

```
class Razred {  
    ~Razred(){  
        Console.WriteLine("Finalizer");  
    }  
    ...  
}
```

- Pogledati sadržaj klase s ILDASM (*Intermediate Language Disassembler*)

Svojstva


- Svojstvo je postupak pristupa zaštićenim varijablama instance
 - Pandan getteru i setteru u Javi, ali praktičnije sintakse
- Automatska svojstva
 - Koristi se u slučaju kad svojstvo služi samo kao omotač oko privatne varijable
 - Može se postaviti modifikator pristupa (npr. private) za get i/ili set
 - Interno se stvara varijabla za pohranu i kod za dohvat i pridruživanje
 - Za svojstva koja imaju samo get dio kod se može napisati i lambda izrazom
- Primjer  SomeOfCSharpFeatures \ PropertiesIndexersRefOut \ Triple.cs
 - Proučiti sadržaj klase s ILDASM

Indekseri


- Indekseri
 - omogućavaju korištenje objekta kao niza (pristup elementima operatorom [])
 - sintaksa uobičajena za svojstva (get i set)
- Sadržaj su uglatim zagradama može biti proizvoljan

```
public int this[string s, int pos] {  
    get {  
        ...  
    }  
    set { ...  
}
```

```
x["A", 5] = 1 + x["B", 3];
```

- Primjer  SomeOfCSharpFeatures \ PropertiesIndexersRefOut \ Triple.cs
 - Proučiti sadržaj klase s ILDASM


Ref, out i varijabilni broj argumenata

- ref modifikator – argumenti su reference (*call by reference*)
 - **Prije poziva postupka** argumenti **moraju** biti inicijalizirani
 - Mora se navesti i prilikom poziva postupka
- out modifikator – izlazni argument
 - U trenutku poziva out postupka argumenti ne moraju biti inicijalizirani.
 - **Pri izlasku iz postupka** out argumenti **moraju** biti postavljeni.
- Varijabilni broj argumenata se definira ključnom riječi params i poljem određenog tipa
- Primjer  SomeOfCSharpFeatures \ PropertiesIndexersRefOut \ Program.cs

Imenovanje argumenata

- Postupci mogu imati opcionalne argumente s pretpostavljenim vrijednostima
- Prilikom poziva argumenti se mogu imenovati (umjesto pridruživanja po redoslijedu argumenata)
 - Priliko poziva navodi se naziv argumenta i vrijednost odvojeni dvotočkom
 - Imenovani argumenti se navode zadnji

Preopterećenje operatora

- Operatori koji se mogu preopteretiti
 - unarni: `+` `-` `!` `~` `++` `--` `true` `false`
 - binarni: `+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>` `==` `!=` `>` `<` `>=` `<=`
- Primjer  `SomeOfCSharpFeatures \ PropertiesIndexersRefOut \ Triple.cs`
- Posljedično sadržaj stringova uspoređujemo s `==`

Enumeracije

- Pobrojani tip (enumerator)
 - Korisnički tip vrijednosti koji nasljeđuje System.Enum
 - Sastoji se od imenovanih konstanti
 - Temeljni tip podataka pobrojanog tipa je int, ali može se promijeniti
 - Može im se pridružiti vrijednost (ali ne mora)
- Korištenja atributa [Flags] dopušta kombinacije vrijednosti

```
[Flags]
public enum DayPeriod
{
    Morning = 1, Evening = 2, Afternoon = 4, Night = 8
}
```

```
DayPeriod p = DayPeriod.Evening | DayPeriod.Night
```

Nulabilni tipovi

- Varijable mogu imati nedefiniranu vrijednost
- Tip podatka `Nullable<T>` pri čemu T mora biti vrijednosni tip (*value type*), npr. `int`
 - Skraćeno se može zapisati `T?`, npr. `int?`

```
int? num = null;  
if (num.HasValue) // isto što i if (num != null)  
    Console.WriteLine("num = " + num.Value);  
else  
    Console.WriteLine("num is Null");
```

- Genericsi implementirani u CLR-u
 - ne koristi se brisanje tipa kao u Javi

Ograničenja na tip parametriziranog razreda

- Ograničenja se postavljaju kontekstualnom ključnom riječi `where`
- Moguća ograničenja:
 - `where T:struct` – tip `T` mora biti value type (*Nullable* nije dozvoljen iako je struktur)
 - `where T:class` – tip `T` mora biti reference type
 - `where T:new()` – tip `T` mora imati konstruktor bez argumenata
 - `where T:naziv baznog razreda` – tip `T` mora biti navedeni razred ili razred koji nasljeđuje taj razred
 - `where T:naziv sučelja` – tip `T` mora implementirati navedeno sučelje
 - `where T:U` – tip `T` mora tip `U` ili izveden iz tipa `U` pri čemu je `U` drugi tip po kojem se vrši parametrizacija

Nulabilne reference

- C# < 8.0
 - String se ne deklarira kao nulabilan
 - String može biti prazan (tzv. null-string) ali to ne znači da je null
 - `string s = string.Empty` // isto što i `s = ""`
 - postavljanje na null znači da nije ni prazan
 - `string s = null;` // vrijedi `s != ""`, `s == null`
- C# 8.0 uvodi koncept nulabilnih referenci, ali ga je potrebno eksplicitno uključiti u projektu
 - <https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references>
 - `string?` bi na taj način predstavljao referencu koja smije biti null, a `string` bi bila referenca koja nije null i za koju nije potrebno provjeravati je li null

Tipovi podataka *var* i *dynamic*

- Varijable definirane kao tip *var*
 - Deklaracija je moguća samo unutar određenog postupka
 - Stvarni tip podatka se određuje prilikom kompilacije


```
var sb = new StringBuilder();  
StringBuilder sb = new StringBuilder();
```

- Varijable definirane kao *Dynamic* varijable
 - Stvarni tip podatka se određuje prilikom izvršavanja

```
dynamic s;  
s = "Neki tekst"; Console.WriteLine(s.GetType());  
s = 12; Console.WriteLine(s.GetType());
```

- Zaobilazi se provjera prilikom kompilacije (većinom nepoželjno osim u iznimnim slučajevima) te se pretpostavlja da podržava bilo koju operaciju (metodu, varijablu)
- Može biti argument funkcije


Proširenja (eng. extensions)

- Razred za koji se piše proširenje je naveden kao prvi parametar statičke metode u statičkom razredu, prefiksiran s *this*
 - Poziva se kao da se radi o postupku unutar tog razreda (iako nije)
 - Primjer  SomeOfCSharpFeatures \ Extensions \ Extensions.cs

```
public static class Extensions { //može se zvati proizvoljno
    public static V GetOrCreate<K, V>(this Dictionary<K, V> dict,
                                     K key) where V : new() {
        if (!dict.TryGetValue(key, out V value)) {
            value = new V();
            dict[key] = value;
        }
        return value;
    }
}
```

```
var dict = new Dictionary<string, List<int>>;
List<int> list = dict.GetOrCreate("Some string");
```

Nasljeđivanje i polimorfizam

- Za označavanje nasljeđivanje ili implementiranja nekog sučelja se koristi dvotočka
 - `class DerivedClass: BaseClass, Interface1, Interface2 { ... }`
- Sučelja po standardu imenovanja počinju slovom `I`
- `sealed` sprječava daljnje nasljeđivanje
- `virtual` deklarira virtualni postupak roditelja koji može biti nadjačan
- `override` deklarira postupak djeteta koji nadjačava, a može koristiti nadjačani postupak
 - Što ako se pokuša nadjačati metoda koja nije virtualna ili ako se definira nova metoda bez nadjačavanja
 - Primjer  `SomeOfCSharpFeatures \ Inheritance \ *.cs`

Odnosi razreda u složenijim tipovima

- Razred `Car` je podrazred razreda `Vehicle`.
- U kojem su odnosu `List<Car>` i `List<Vehicle>` ?
 - Nisu hijerarhijski povezani
- U kojem su odnosu:
 - `IEnumerable<Car>` i `IEnumerable<Vehicle>`
 - `IComparer<Car>` i `IComparer<Vehicle>`
- Odgovor nije očit (jednostavan) !!

Invarijantnost, kovarijantnost i kontravarijantnost

- Invarijantnost (engl. invariance)
 - Mora se koristiti samo specificirani tip
- Kovarijantnost (engl. covariance)
 - Mogućnost korištenja nekog izvedenog tipa umjesto originalno navedenog
 - Neki tip je kovarijantan ako zadržava postojeće odnose među tipovima
- Kontravarijantnost (engl. contravariance)
 - Mogućnost korištenja nekog općenitijeg tipa umjesto originalno navedenog
 - Neki tip je kontravarijantan ako stvara suprotan odnos među postojećim tipovima

Kovarijantnost

- Primjer  SomeOfCSharpFeatures \ CovarianceContravariance

```
static void PrintVehicles(IEnumerable<Vehicle> vehicles) {  
    foreach (var vehicle in vehicles)  
        Console.WriteLine("\t " + vehicle.Model);  
}
```

- Kao argument moguće je poslati `List<Vehicle>`, jer `List<T>` implementira sučelje `IEnumerable<T>`
- Ali moguće je poslati i `List<Car>` !!
 - `List<Car>` se može pretvoriti u `IEnumerable<Car>`, a sučelje **`IEnumerable<T>` je kovarijantno**

```
public interface IEnumerable<out T>
```

Kontravarijantnost


- Primjer  SomeOfCSharpFeatures \ CovarianceContravariance

```
void PrintBetterCar(Car a, Car b, IComparer<Car> comparer) {  
    int result = comparer.Compare(a, b);  
    string betterModel = result <= 0 ? a.Model : b.Model;  
    string worseModel = result <= 0 ? b.Model : a.Model;  
    Console.WriteLine($"\\t{betterModel} ... {worseModel}");  
}
```

- Kao komparator moguće je upotrijebiti objekt tipa `IComparer<Vehicle>`, jer je **sučelje `IComparer<T>` kontravarijantno**

```
public interface IComparer<in T>
```

Delegati (1)

- Delegati su objekti koje sadrže reference na postupke
 - Omogućavaju metodama da budu argumenti neke druge metode
 - Nalik pokazivačima na funkcije u C-u
- Delegati koji sadrže reference na više postupaka nazivaju se *MultiCastDelegate*
 - Pozivom delegata redom se pozivaju referencirane metode
- Primjer  SomeOfCSharpFeatures \ CovarianceContravariance

```
void PrintBetterCar(Car a, Car b, Comparison<Car> comparer) {  
    int result = comparer(a, b);  
    string betterModel = result <= 0 ? a.Model : b.Model;  
    string worseModel = result <= 0 ? b.Model : a.Model;  
    Console.WriteLine($"\\t{betterModel} ... {worseModel}");  
}
```

Delegati (2)

- Delegat se definira kao varijabla određenog tipa delegata
- Tip delegata definira se sljedećom sintaksom

```
public delegate PovratniTip NazivTipaDelegata (args)
```

- Primjer:

```
public delegate double Tip(int a, string b)
```

bi definirao tip delegata koji omogućava pohranu referenci na sve postupke kojima imaju dva argumenta tipa `int` i `string`, a vraćaju `double`

- Interno se stvara novi razred *NazivTipaDelegata* koji nasljeđuje razred *MultiCastDelegate*
- Nakon toga bi se mogao definirati delegat na sljedeći način
`Tip nazivdelegata;`
- Delegati imaju definirane operacije `=`, `+=`, `-=`

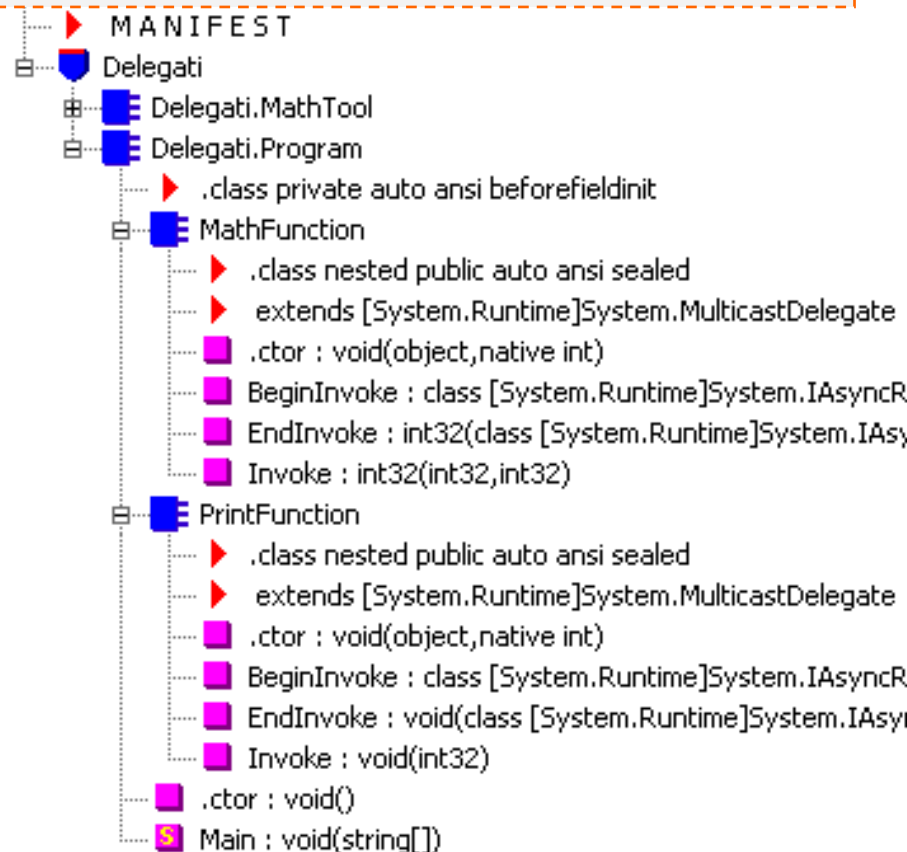
Primjer tipa delegata

- Primjer  SomeOfCSharpFeatures \ Delegates \ Program.cs


```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
}
```

- Definirana su 2 tipa delegata
 - MathFunction* za postupke koji primaju dva cjelobrojna argumenta i vraćaju cijeli broj
 - PrintFunction* za postupke koji primaju cijeli broj i ne vraćaju ništa
- U glavnom programu definirane dvije varijable (delegata)

```
MathFunction mf = ...  
PrintFunction pf =
```




Primjer pridruživanja postupka delegatu (1)

- Primjer  SomeOfCSharpFeatures \ Delegates \ MathTool.cs
 - Vlastiti razred MathTool sadrži nekoliko postupaka koji svojim potpisom odgovaraju tipova delegata


```
public class MathTool {  
    public static int Sum(int x, int y) {  
        return x + y;  
    }  
    public static int Diff(int x, int y) {  
        return x - y;  
    }  
    public static void PrintSquare(int x) {  
        Console.WriteLine("x^2 = " + x * x);  
    }  
    public static void PrintSquareRoot(int x) {  
        ...  
    }  
}
```


Primjer pridruživanja postupka delegatu (2)

- Primjer  SomeOfCSharpFeatures \ Delegates \ Program.cs
 - Varijabla *mf* je delegat koji sadrži reference na postupke koji primaju 2 cijela broja (kao što je npr. postupak Sum iz razreda MathTool)

```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
    static void Main(string[] args) {  
        int x = 16, y = 2;  
        MathFunction mf = MathTool.Sum;  
        Console.WriteLine("mf({0}, {1}) = {2}",  
                           x, y, mf(x, y));  
        mf = MathTool.Diff;  
        Console.WriteLine("mf({0}, {1}) = {2}",  
                           x, y, mf(x, y));  
        ...  
    }  
}
```

Primjer pridruživanja postupka delegatu (3)

- Primjer  SomeOfCSharpFeatures \ Delegates \ Program.cs
 - Delegatu se može pridružiti više postupaka (mogu se i ukloniti)
 - Obično ima smisla za postupke koje ne vraćaju nikakvu vrijednost

```
class Program {  
    public delegate int MathFunction(int a, int b);  
    public delegate void PrintFunction(int n);  
    static void Main(string[] args) {  
        int x = 16, y = 2;  
        ...  
        PrintFunction pf = MathTool.PrintSquare;  
        pf += MathTool.PrintSquareRoot;  
        pf(x);  
        pf -= MathTool.PrintSquare;  
        Console.WriteLine();  
        pf(y);  
    }  
}
```

Primjeri postojećih tipova delegata

- Func i Action kao dva najpoznatija tipa delegata
- `Action<in T1>`, `Action<in T1, in T2>`,
`Action<in T1, in T2, in T3>`, ...,
`Action<in T1,..., in T16>`
 - Referenca na postupke koji ne vraćaju ništa, a primaju 1, 2, 3, ..., ili 16 argumenata
 - Argumenti su kontravarijantni (vidi sljedeći slajd)
- `Func<in T1, out TResult>`,
`Func <in T1, in T2, out TResult>`, ...,
`Func<in T1, in T2, .. in T16, out TResult>`
 - Referenca na postupke primaju do 16 argumenata i vraća vrijednost tipa `TResult`
 - `TResult` je kovarijantan, a ostali su kontravarijantni
- U prethodnim primjerima se umjesto *MathFunction* mogao koristiti *Func<int, int, int>*, a umjesto *PrintFunction* *Action<int>*

Func i varijantnost

- Povratni tip je kovarijantan → postupak koji se pridružuje delegatu može vraćati izvedeni tip onog koji je predviđen pri parametrizaciji
- Ulazni tipovi su kontravarijantni → postupak koji se pridružuje za ulazne argumente može imati traženi tip ili njemu nadređene.
- Npr. Ako je definiran delegat tipa *Func<Car, Car>* tada se delegatu tog tipa mogu pridružiti reference na postupke npr
 - `Car Test (Car a) { ... }`
 - `ElectricCar Test (Vehicle a) { ... }`

ali ne i npr.

- `Vehicle Test (Car a) { ... }`
- `Car Test (ElectricCar a) { ... }`

- Navedeno ima smisla, jer ako je

Func<Car, Car> f = nešto od navedenog

tada kasnije u programu slijedi *Car c = f(neki automobil)* pa je potpuno svejedno je li povratna vrijednost *Car* ili nešto izvedeno iz njega, odnosno prima li pridruženi postupak *Car* ili nešto općenitije.