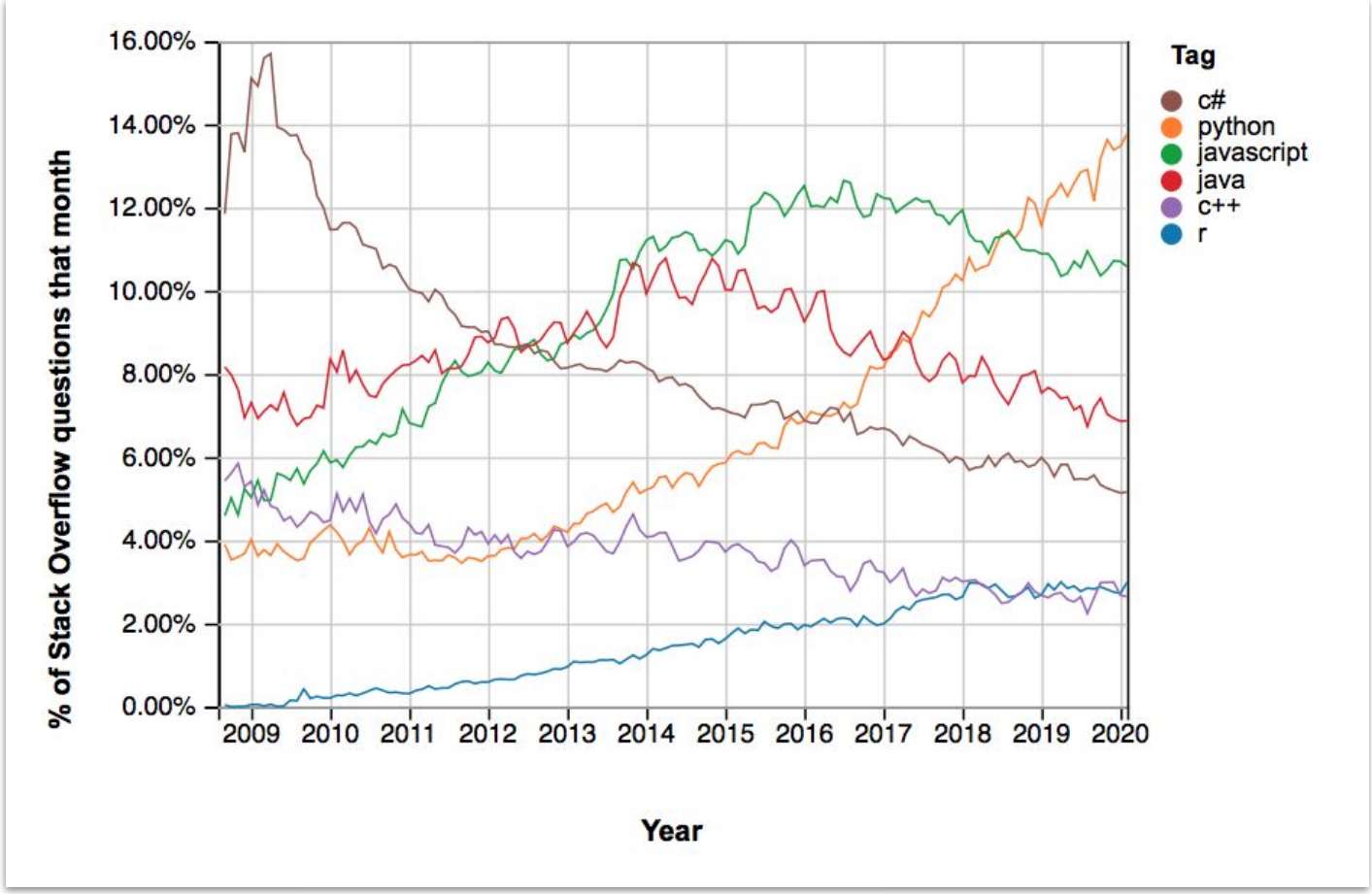
# C#

I deo

# Java vs C#




**Java**

Airbnb, Instagram, Spotify, Netflix…

**C#**

Stack Exchange, Microsoft, Coderus, Docplanner…

# Java vs C#

| Osobina | Java | C# |
|---|---|---|
| Paradigma | objektno-orijentisan jezik zasnovan na klasama | objektno-orijentisan jezik, funkcionalan |
| Platforma | nezavisan od platforme | Microsoft |
| Upotreba | kompleksne web aplikacije, aplikacije koje zahtevaju konkurentnost, android aplikacije, naučne aplikacije i softveri | *Windows* aplikacije, igrice, virtuelna stvarnost, web aplikacije |
| Garbage Collector | automatski i nedeterministički, *finalize*() metoda | automatski i nedeterministički, *IDisposable* interfejs i *Dispose* metoda |
| Lambda i LINQ izrazi | lambda izrazi podržani od Jave 8, LINQ ne postoje | podržano |

# Java vs C#

| Osobina | Java | C# |
|---|---|---|
| Izuzeci | *checked (compile time)* i *unchecked (run time)* | samo *unchecked* |
| Strukture | nisu podržane | podržane |
| Goto izraz | ne postoji | postoji |
| Polimorfizam | podrazumevano | Omogućava se upotrebom ključne reči *virtual* |
| Preklapanje operatora | nije podržano | podržano |
| Pokazivači | nisu podržani | podržani |
| Uslovno kompajliranje | nije podržano | podržano |
| Nizovi | direktna specifikacija objekta | specifikacija sistema |
| Bezbednost | veoma siguran, zato što ima statičke tipove i verifikaciju koda | mnoge karakteristike ga čine ranjivim na napade |

# Okruženje

- Visual Studio
- WPF (*Windows Presentation Foundation*) – UI radni okvir za kreiranje interfejsa desktop aplikacija

# Zadatak

Napraviti WPF projekat.

# Assembly

- Osnovni gradivni blok *.NET* aplikacija
- Kompajlirani kod koji može *CLR* da izvrši
- Predstavlja kolekciju tipova i resursa koji su napravljeni da rade zajedno i formiraju logičku jedinicu funkcionalnosti
- *dll* ili *exe* u zavisnosti od toga šta nam treba

# Namespace

- Sličan paketima u Javi
- Kontejner za skup povezanih klasa i prostora imena. Klase unutar jednog prostora imena moraju imati jedinstveno ime
- Može da sadrži druge prostore imena. Unutrašnji prostori imena moraju biti odvojeni `.`
- Klasi iz nekog prostora imena pristupamo pomoću `.`

Prostor imena u kojem se nalazi klasa `Person`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Primeri.Model
{
    1 reference
    class Person
    {
    }
}
```

```
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 Primeri
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 references
    public partial class MainWindow : Window
    {

        private Primeri.Model.Person pera;

        0 references
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Prostor imena u kojem koristimo klasu koju smo kreirali.

Način 1.

Upotreba klase iz drugog prostora imena

Upotreba `using` sintakse

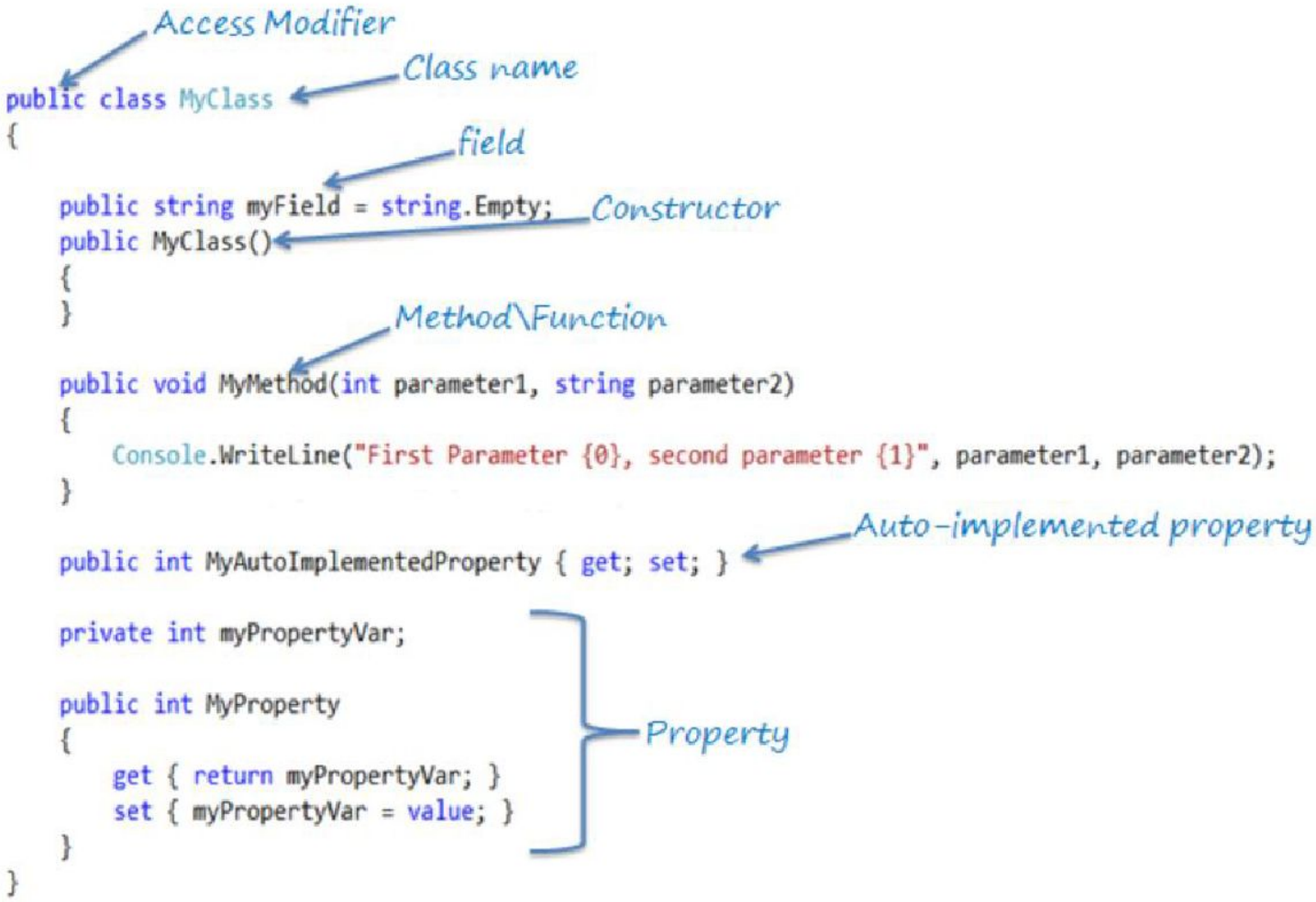
Način 2.

```
using Primeri.Model;
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 Primeri
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 references
    public partial class MainWindow : Window
    {

        private Person pera;

        0 references
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

# Klase

- Klasa je generalizacija nekog objekta, ona definiše određena svojstva, atribute, događaje, metode itd.
- Omogućava da kreirate sopstvene tipove podataka grupisanjem promenljivih drugih tipova, metoda i događaja.
- Definiše se upotrebom ključne reči ***class***.
- Gradivni elementi klase su:
  - Modifikator pristupa – definiše dostupnost klase
  - Ime klase
  - Atribut – promenljiva na nivou klase koja može da sadrži vrednost. Obično ima *private* modifikator pristupa i koristi se zajedno sa svojstvom.
  - Konstruktor – može da bude sa parametrima ili bez njih. Poziva se prilikom kreiranja instance neke klase.
  - Metoda – obična funkcija samo što se nalazi unutar klase i može se pozvati samo nad objektom.
  - Auto-implementirano svojstvo – jednostavna deklaracija svojstva kada nije potreba dodatna logika u pristupnicima svojstva.
  - Svojstvo – definiše se za privatne atribute. Sadrži *get* i *set* metodu. Pomoću *get* metode pristupamo vrednosti atributa, a pomoću *set* metode postavljamo vrednost atributa.

# Klase


Access Modifier
Class name
field
Constructor
Method\Function
Auto-implemented property
Property
public class MyClass
{
public string myField = string.Empty;
public MyClass()
{
}
public void MyMethod(int parameter1, string parameter2)
{
Console.WriteLine("First Parameter {0}, second parameter {1}", parameter1, parameter2);
}
public int MyAutoImplementedProperty { get; set; }
private int myPropertyVar;
public int MyProperty
{
get { return myPropertyVar; }
set { myPropertyVar = value; }
}
}

## C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Primeri.Model
{
    2 references
    class Person
    {
        // auto-implemented property
        0 references
        public string FirstName { get; set; }
        0 references
        public string LastName { get; set; }

        // property

        private string _personNo;

        0 references
        public string PersonNo
        {
            get { return _personNo; }
            set { _personNo = value; }
        }

        0 references
        public Person()
        {
        }

        0 references
        public Person(string firstName, string lastName, string personNo)
        {
            FirstName = firstName;
            LastName = lastName;
            this._personNo = personNo;
        }


    }
}
```

## Java

```
package com.examples;

public class Person {
    private String firstName;
    private String lastName;
    private String personNo;

    public Person() {
    }

    public Person(String firstName, String lastName, String personNo) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.personNo = personNo;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getPersonNo() {
        return personNo;
    }

    public void setPersonNo(String personNo) {
        this.personNo = personNo;
    }
}
```

# Modifikatori pristupa

- *Public*
  - Možemo da mu pristupimo u bilo kom delu programa
- *Private*
  - Možemo da mu pristupimo samo iz klase u kojoj je definisan
- *Protected*
  - Možemo da mu pristupimo iz klase u kojoj je definisan i iz klase naslednice
- *Internal*
  - Možemo da mu pristupimo iz bilo kog dela programa, koji se nalaze u istom *assembly-u*.

# Modifikatori pristupa

- Definišemo ih za klase, atribute, metode i funkcije.
- Podrazumevani modifikatori zavise od okolnosti:
  - za klase, strukture i interfejse definisane direktno u namespace-u je *internal*,
  - za članove klase i strukture *private*,
  - za članove interfejsa *public*...

- ***Izuzetno je mudro da modifikatori pristupa uvek budu eksplicitno navedeni!***

# Nasleđivanje

- Jedna od 3 glavne karakteristike objektno orijentisanog programiranja.
- Omogućava nam da kreiramo nove klase u kojima proširujemo i menjamo ponašanje postojećih klasa.
- Klasa čije članove nasleđujemo zove se osnova ili bazna klasa, a klasa koja je nasledila te članove je izvedena klasa.
- Podržano je jednostruko nasleđivanje, što znači da izvedena klasa može da ima samo jednu baznu klasu. Međutim, nasleđivanje je tranzitivno.
  - Primer: Ako imamo klase A, B i C. Klasa B nasledi klasu A, a klasa C nasledi B, onda klasa C direktno nasleđuje sve članove deklarisane i u klasi A i u klasi B.
- Konceptualno gledano izvedena klasa je specijalizacija osnovne klase. Kada definišete da klasa potiče iz neke druge klase, onda ona implicitno dobije sve članove osnovne klase, osim njenih konstruktora i finalizatora. Izvedena klasa koristi kod iz osnovne klase bez potrebe za ponovnim dodavanjem. U izvedenu klasu možete dodati nove članove i tako proširiti funkcionalnosti osnovne klase.

Java

```java
public class Student extends Person{

    private String indexNo;
    private int year;

    public Student() {
    }

    public Student(String firstName, String lastName, String personNo, String indexNo, int year) {
        super(firstName, lastName, personNo);
        this.indexNo = indexNo;
        this.year = year;
    }

    public String getIndexNo() {
        return indexNo;
    }

    public void setIndexNo(String indexNo) {
        this.indexNo = indexNo;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}
```

C#

```csharp
namespace Primeri.Model
{
    2 references
    class Student : Person
    {
        1 reference
        public String IndexNo { get; set; }
        1 reference
        public int Year { get; set; }


        0 references
        public Student():base()
        {
        }

        0 references
        public Student(string lastName, string firstName, string personNo, string indexNo, int year)
            : base(firstName, lastName, personNo)
        {
            IndexNo = indexNo;
            Year = year;
        }
```

# Polimorfizam

- Polimorfizam je važna osobina objektno orijentisanog programiranja jer omogućava da osnovna klasa definiše funkcije koje će biti zajedničke za sve izvedene klase, ali da izvedenim klasama ostavi slobodu da same implementiraju sve te funkcije.
- U C# programskom jeziku polimorfizam može da se napravi upotrebom:
  - Virtuelnih metoda
    - Kreira se u osnovnoj klasi upotrebom ključne reči *virtual*.
    - Ove metode imaju implementaciju u osnovnoj klasi, ali mogu i da se redefinišu u klasi naslednici upotrebom ključne reči *override*.
    - Kada se pozove virtuelna metoda, proverava se da li postoji preklapajuća metoda za tip objekta nad kojim je pozvana metoda. Ukoliko u izvedenoj klasi postoji redefinisana metoda onda će se pozvati ta metoda, a ako ne postoji onda će se pozvati metoda iz osnovne klase.
    - Podrazumevani tip metode je *non-virtual* i ne možete da ih redefinišemo.
    - Ključnu reč *virtual* ne možete upotrebiti sa statičkim, apstraktnim, privatnim ili *override* modifikatorom pristupa.
  - Abstraktnih metoda
    - Možete ih kreirati samo u abstraktnim klasama.
    - Ove metode nemaju implementaciju i za razliku od virtuelnih metoda koje ne morate da redefnišete, ove metode morate da redefinišete u bilo kojoj neabstraktnoj klasi koja direktno nasleđuje abstraktnu klasu.

## *Virtual metoda*

```
class Person
{
    // auto-implemented property
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }

    // property

    private string _personNo;

    0 references
    public string PersonNo
    {
        get { return _personNo; }
        set { _personNo = value; }
    }

    1 reference
    public Person()
    {
    }

    1 reference
    public Person(string firstName, string lastName, string personNo)
    {
        FirstName = firstName;
        LastName = lastName;
        this._personNo = personNo;
    }

    1 reference
    public virtual string Hello()
    {
        return "Hello, I am a person";
    }

}
```

## *Abstraktna klasa i metoda*

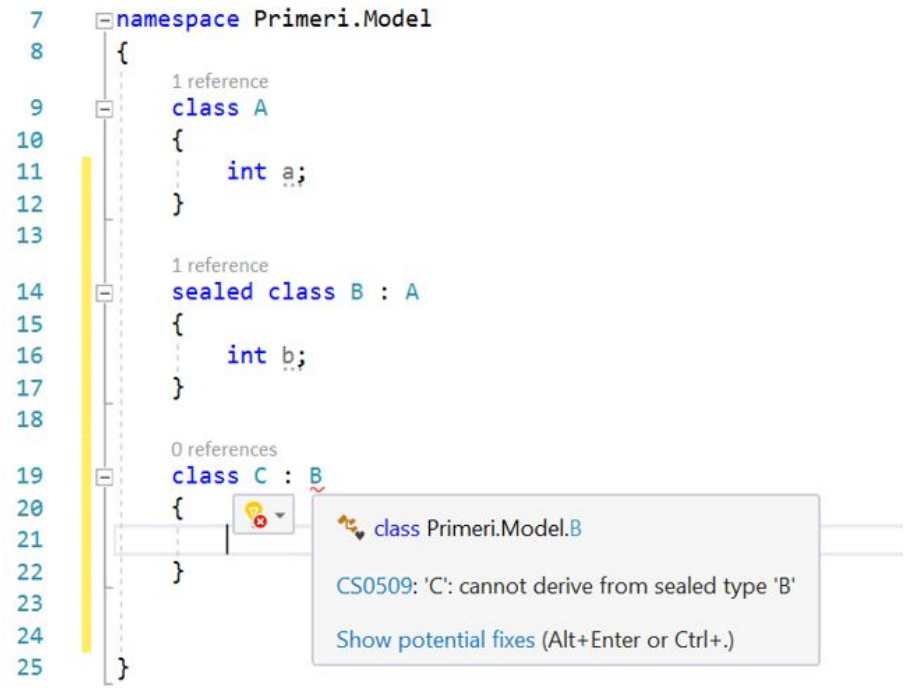
```
abstract class Person
{
    // auto-implemented property
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }

    // property

    private string _personNo;

    0 references
    public string PersonNo
    {
        get { return _personNo; }
        set { _personNo = value; }
    }

    1 reference
    public Person()
    {
    }

    1 reference
    public Person(string firstName, string lastName, string personNo)
    {
        FirstName = firstName;
        LastName = lastName;
        this._personNo = personNo;
    }

    2 references
    public abstract string Hello();
```

```
class Student : Person
{
    1 reference
    public String IndexNo { get; set; }
    1 reference
    public int Year { get; set; }


    0 references
    public Student():base()
    {
    }

    0 references
    public Student(string lastName, string firstName, string personNo, string indexNo, int year)
        : base(firstName, lastName, personNo)
    {
        IndexNo = indexNo;
        Year = year;
    }

    1 reference
    public override string Hello()
    {
        return "Hello, I am a student";
    }
}
```

```
0 references
class Teacher : Person
{
    2 references
    public override string Hello()
    {
        return "Hello, I am a teacher.";
    }
}
```

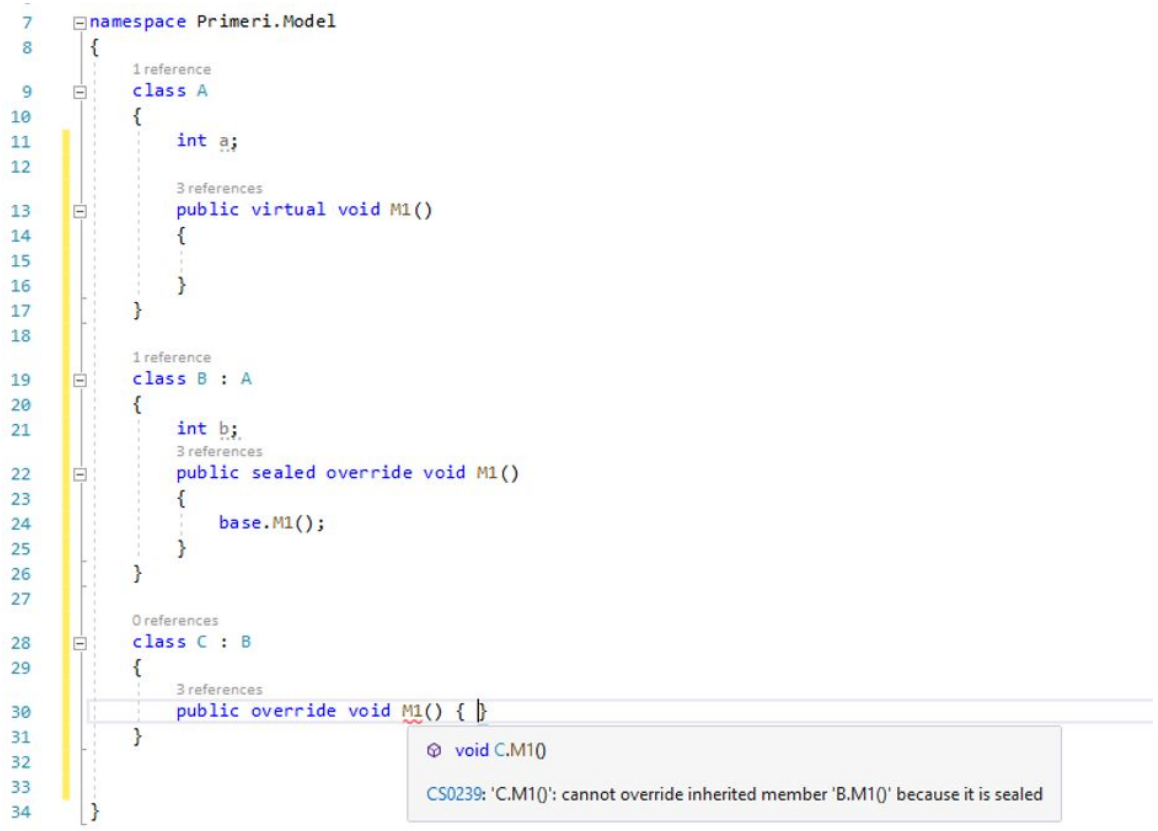
# Sealed modifikator

- Ne postoji u Javi.
- Možemo da ga definišemo nad:
  - Klasom
    - Klasa koja je označena sa ovim modifikatorom ne može da se nasleđuje.



# Sealed modifikator

- Nad atributom i metodom
  - Metode i atributi koji su označeni ovim modifikatorom ne smeju da se preklapaju, odnosno ne možete da ih *override-ujete*.



# Const

- Ovu ključnu reč koristite da označite da je neki atribut konstanta ili da je nešto lokalna konstanta. Konstantni atributi i lokalne konstante nisu promenljive i njihove vrednosti se ne mogu menjati.
- Konstante mogu da budu brojevi, boolean vrednosti, stringovi ili null reference.
- Inicijalizator konstante mora biti konstantan izraz, koji se implicitno može pretvoriti u ciljni tip. Konstantni izraz je izraz koji se u potpunosti može proceniti u toku kompilacije.
- Implicitno podrazumeva da je atribut i statički.
- **Vrednost konstanti se dodeljuje samo na mestu definisanja**.

```
const double Pi = 3.14;
const string ProductName = "Visual C#";
```

# Readonly

```
public readonly struct Point
{
    public readonly double X;
    public readonly double Y;

    public Point(double x, double y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}
```

```
readonly int year;
Age(int year)
{
    this.year = year;
}
```

- Može da se nađe uz:
  - Atribut
    - Ukoliko se nađe uz atribut onda označava da se dodeljivanje vrednosti tom atributu može dogoditi samo kao deo deklaracije ili u konstruktoru iste klase.
    - Označava da je neki atribut samo za čitanje i da njegova vrednost ne može da se menja.
    - Vrednost mu se može dodeliti više puta u okviru deklaracije polja ili u konstruktoru. Ovom atributu vrednost se ne može dodeliti nakon izlaska iz konstruktora.
  - Strukturu
    - Ukoliko se nađe uz strukturu onda označava da je struktura nepromenljiva.
  - Atribut unutar strukture
    - Ukazuje da član strukture ne mutira unutrašnjost strukture
  - Povratnu vrednost funkcije
    - Ukazuje da metoda vraća referencu u koju upisi nisu dozvoljeni.
- **Vrednost mu se dodeljuje na mestu definisanje ili u konstruktoru.**

# Readonly vs const modifikator

- Ove dve ključne reči se razlikuju.
- Atribut koji uz sebe ima modifikator *const* može se inicijalizovati samo kada ga deklarišemo, dok se *readonly* atribut može inicijalizovati prilikom deklaracije ili u konstruktoru. Zbog toga *readonly* može imati različite vrednosti u zavisnosti od konstruktora koji se koristi.

# Struktura

- Pomoću ključne reči ***struct*** generišemo novi tip podataka, koji je sličan klasi, ali je uvek *sealed*, odnosno ne može da se nasleđuje. Može da implementira interfejs, ali ne može da nasleđuje drugu strukturu ili klasu.
- Služi za enkapsulaciju malog broja obeležja.
- Deklariše se pomoću ključne reči struct sa public ili internal modifikatorom pristupa. Podrazumevani modifikator pristupa za strukturu i njene članove je internal.
- Može da sadrži samo parametrizovane konstruktore, konstante, atribute, metode, svojstva, indeksere, događaje. Ne može da sadrži konstruktor bez parametara i destruktor.
- Članovi strukutre ne mogu biti apstraktni, virtuelni ili zaštićeni
- Objekat strukture možete da kreirate sa ili bez upotrebe ključne reči new. Kada ga kreirate sa upotrebom ključne reči new, onda se poziva odgovarajući konstruktor i članovima strukture se dodele određene vrednosti. Ako ga kreirate bez upotrebe ključne reči new, onda se ne poziva ni jedan konstruktor i tada svi članovi strukture ostaju bez dodele. Dakle, ako objekat strukture kreirate bez ključne reči new, onda prvo svim članovima morate dodeliti vrednost, pa tek onda možete da pristupite članovima te strukture, jer će u suprotnom doći do greške.

```
struct Employee
{
    public int EmpId;
    public string FirstName;
    public string LastName;

    public Employee(int empid, string fname, string lname)
    {
        EmpId = empid;
        FirstName = fname;
        LastName = lname;
    }
}

Employee emp = new Employee(10, "Bill", "Gates");

Console.Write(emp.EmpId); // prints 10
Console.Write(emp.FirstName); // prints Bill
Console.Write(emp.LastName); // prints Gates
```

# Klasa vs struktura

- Klasa je referentni tip, a struktura vrednosni tip
- Struktura ne može da sadrži konstruktor i destruktor bez parametara
- Objekat strukture može da se inicijalizuje bez upotrebe ključne reči new, ali onda ne možete da koristite njegove metode, događaje ili atribute
- Struktura ne podržava nasleđivanje