

INHERITANCE

An `IEnumerable` is a list or a container which can hold some items. You can iterate through each element in the `IEnumerable`. You can not edit the items like adding, deleting, updating, etc. instead you just use a container to contain a list of items. It is the most basic type of list container.

All you get in an `IEnumerable` is an enumerator that helps in iterating over the elements. An `IEnumerable` does not hold even the count of the items in the list, instead, you have to iterate over the elements to get the count of items.

`ICollection` is another type of collection, which derives from `IEnumerable` and extends its functionality to add, remove, update element in the list. `ICollection` also holds the count of elements in it and we do not need to iterate over all elements to get total number of elements.

`IList` extends `ICollection`. An `IList` can perform all operations combined from `IEnumerable` and `ICollection`, and some more operations like inserting or removing an element in the middle of a list.

The following method just returns null:

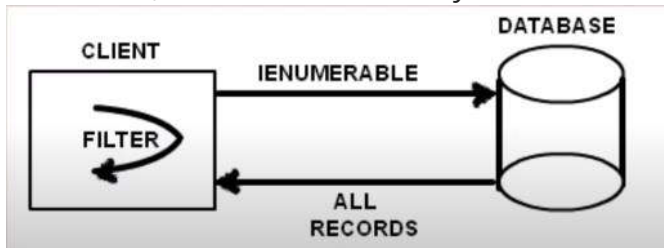
```
IEnumerable<object> Incorrect()
{
    return null;
}
```

IEnumerable vs IQueryable

IEnumerable

IEnumerable is suitable for querying data from in-memory collections like List, Array and so on.

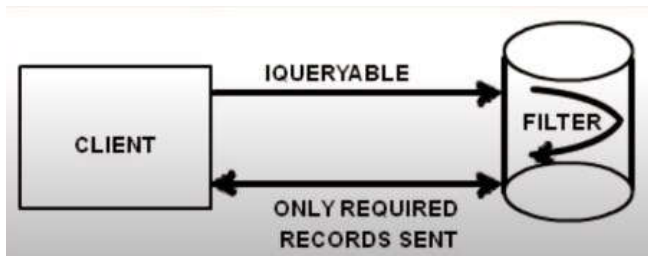
While querying data from the database, IEnumerable executes "select query" on the server-side, loads data in-memory on the client-side and then filters the data.



IQueryable

IQueryable is suitable for querying data from out-memory (like remote database, service) collections.

While querying data from a database, IQueryable executes a "select query" on server-side with all filters.



to execute

Execution is deferred until the query is iterated over

- foreach loop
- ToList(), ToArray(), ToDictionary()
- Singleton queries

count avg itn



smarter to get result first than

iterating

Dispose and Finalize

Method `dispose()` is used to free unmanaged resources whenever it is invoked.

Method `finalize()` is used to free unmanaged resources before the object is destroyed.

the method **dispose()** has to be explicitly invoked by the user whereas, the method **finalize()** is invoked by the garbage collector, just before the object is destroyed. As you have seen, finalizers can be used to release unmanaged resources when the garbage collector kicks in.

`IDisposable` provide a

way for the object user to clean up the object as soon as it is finished. However, if the caller forgets to call `Dispose()`, the unmanaged resources may be held in memory indefinitely.

As keyword

The operator `as` also is used for type conversion but invalid conversion returns `null`, not an exception.

Is keyword

The `is` operator is used to check if the run-time type of an object is compatible with the given type or not. It returns `true` if the given object is of the same type otherwise, return `false`. It also returns `false` for `null` objects for derived objects it will return `true`

?? Operator

it is placed between two operands and returns the left operand only if its value is not `null`, otherwise it returns the right operand.

Casting base to extend vs extend to base

It is possible to cast an **instance** of the *extended* class to the base class. It is however **not** possible to cast an instance of the base class to the extended.

//Possible

```
BaseClass baseObj = new ExtendedClass(); // Creating instance of extended and implicit up-cast it to base
```

```
ExtendedClass extendedObj = (ExtendedClass)baseObj; // Down-cast of the reference stored in "baseObj" to the extended obj.
```

//InvalidCastException

```
BaseClass baseObj1 = new BaseClass();
```

```
ExtendedClass extendedObj1 = (ExtendedClass)baseObj1;
```

//Possible

```
ExtendedClass ExtendedClass3 = new ExtendedClass();
```

```
BaseClass base123 = (BaseClass)ExtendedClass3;
```

foreach loop

The foreach-loop statement is used, when we do not need to change the elements, but just to read them.

always through all elements – from the start to the end(different from for-loop)

The loop variable in foreach-loops is read-only so we cannot modify the current loop item from the loop body.

must implement IEnumerable and provide

```
public IEnumerator GetEnumerator()
```

```
{
```

```
// Return the array object's IEnumerator.
```

```
return carArray.GetEnumerator();
```

```
}
```

the foreach construct will obtain the interface in the background when necessary.

object initializer syntax

// Make a Point by setting each property manually.

```
Point firstPoint = new Point();
```

```
firstPoint.X = 10;
```

```
firstPoint.Y = 10;
```

```
firstPoint.DisplayStats();
```

// Or make a Point via a custom constructor.

```
Point anotherPoint = new Point(20, 20);
```

```
anotherPoint.DisplayStats();
```

// Or make a Point using object init syntax. the default constructor is called implicitly.

```
Point finalPoint = new Point { X = 30, Y = 30 };
```

The final Point variable is not making use of a custom constructor (as one might do traditionally) but

is rather setting values to the public X and Y properties. Behind the scenes, the type's default constructor is invoked, followed by setting the values to the specified properties. To this end, object initialization syntax is just shorthand notation for the syntax used to create a class variable using a default constructor and to set the state data property by property

// Here, the default constructor is called explicitly.

```
Point finalPoint = new Point() { X = 30, Y = 30 };
```

/ Calling a custom constructor

```
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

the following Point declaration results in an X value of 100 and a Y value of 100, regardless of the fact that the constructor arguments specified the values 10 and 16

// Calling a more interesting custom constructor with init syntax.

```
Point goldPoint = new Point(PointColor.Gold) { X = 90, Y = 20 };
```

// Create and initialize a Rectangle.

```
Rectangle myRect = new Rectangle
```

```
{
```

```
TopLeft = new Point { X = 10, Y = 10 },
```

```
BottomRight = new Point { X = 200, Y = 200}
```

```
}
```

Properties

Be aware of course that if you are

building a property that requires additional code beyond getting and setting the underlying private field

(such as data validation logic, writing to an event log, communicating with a database, etc.), you will be

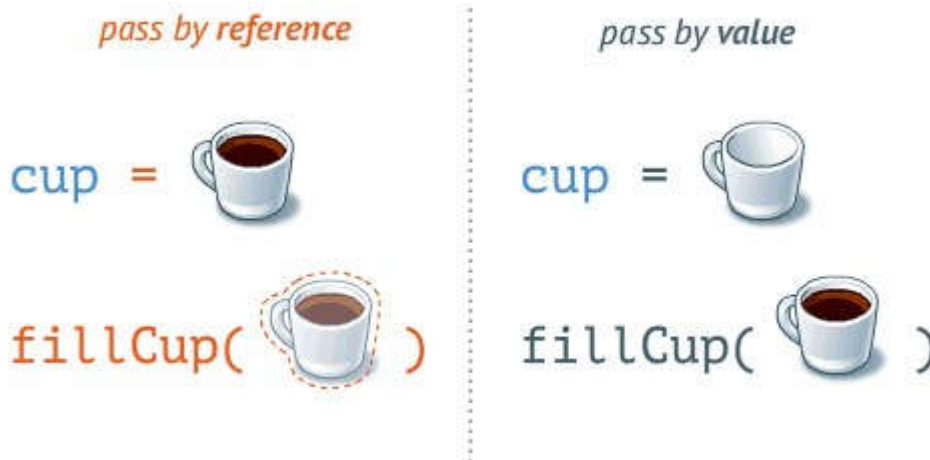
required to define a "normal" .NET property type by hand. C# automatic properties never

do more than
provide simple encapsulation for an underlying piece of (compiler-generated) private data.

Sealed methods

```
class X {  
    protected virtual void F() { Console.WriteLine("X.F"); }  
    protected virtual void F2() { Console.WriteLine("X.F2"); }  
}  
class Y : X {  
    sealed protected override void F() {  
        Console.WriteLine("Y.F"); }  
    protected override void F2() { Console.WriteLine("X.F3"); }  
}  
class Z : Y {  
    // Attempting to override F causes compiler error CS0239.  
    // protected override void F() { Console.WriteLine("C.F"); }  
    // Overriding F2 is allowed.  
    protected override void F2() { Console.WriteLine("Z.F2"); }  
}
```

Passing by reference



Hence, whenever an argument of a reference type is passed to a method, the method's parameter receives the reference itself
passing arguments of reference type, only the value of the variable that keeps the address to the object is copied. Note that this does not copy the object itself.
By passing the argument that are of reference type, the only thing that is copied is the variable that keeps the reference to the object, but not the object data.
primitive types are passed by their values, the objects, however, are passed by reference.
class Program

```

{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}

```

Coupling

Most of the classes from .NET Common Type System (CTS) and .NET Framework define methods that depend only on the data within their class and the passed arguments. In standard libraries, the methods dependencies from external classes are minimal and that is why they are easy to reuse. The .NET Framework class library strongly follows the idea of loose coupling. Whenever a method reads or modifies global data and depends on 10 additional objects, which must be initialized within the instance of its own class, it is considered a coupled to its environment

Sealed methods

Sealing of methods is done when we rely on a piece of functionality and we don't want it to be altered. We already know that methods are sealed by default. But if we want a base class virtual method to become sealed in a derived class, we use `override sealed`.

Polymorphism

Polymorphism allows treating objects of a derived class as objects of its base class. It is mostly related to overriding methods in derived classes, in order to change their original behavior.

```
Lion lion = new AfricanLion(false, 60);
```

```
lion.CatchPrey(null); // Will print "AfricanLion.CatchPrey", because // the variable lion has a value of type AfricanLion
```

the overwritten method is called and not the base method. This happens, because it is validated what the actual class behind the variable is and whether it implements (overwrites) that method. Rewriting of methods is also called overriding of virtual methods. Virtual methods as well as abstract methods can be overridden. Abstract methods are actually virtual methods without a specific implementation. All methods defined in an interface are abstract and therefore virtual, although this is not explicitly defined. When Should We Use Polymorphism? The answer to this question is simple: whenever we want to enable changing a method's implementation in a derived class.

method hiding

```
class A
{
    public virtual void show()
    {
        Console.WriteLine("Hello: Base Class!");
        Console.ReadLine();
    }
}

class B : A
{
    public override void show()
    {
        Console.WriteLine("Hello: Derived Class!");
        Console.ReadLine();
    }
}
```



```

class C : B
{
    public new void show()
    {
        Console.WriteLine("Am Here!");
        Console.ReadLine();
    }
}

class Polymorphism
{
    public static void Main()
    {
        A a1 = new A();
        a1.show();
        B b1 = new B();
        b1.show();
        C c1 = new C();
        c1.show();
        A a2 = new B();
        a2.show();
        A a3 = new C();
        a3.show();
        B b3 = new C();
        b3.show();
    }
}

```

```

hello base
hello derived
am here
hello derived
hello derived
helllo derived

```

za new ako napravis inatanca od svojata klasa ke bide toj metod ako ne go krie i bara drug

```

    C c1 = new C();

```

bez polimorfizam ke bide toj metod
so polimorfizam drug metod
ke odi gore i ke bara duri ako e new ke go zeme

Нема Полиморфизам

class Program

```
{
    class A {
        public void F() {
            Console.WriteLine("A.F");
        }
    }
    class B : A
    {
        public void F()
        {
            Console.WriteLine("B.F");
        }
    }
    static void Main(string[] args)
    {

        A a = new B();
        a.F();
        //print AF
        Console.ReadLine();
    }
}
```

Метод со virtual **може** да се препокрие но не **мора**
class Program

```
{
    class A {
        public virtual void F() {
            Console.WriteLine("A.F");
        }
    }
    class B : A
    {
    }
    static void Main(string[] args)
    {
        A a = new B();
        a.F();
        //print AF
        Console.ReadLine();
    }
}
```

Чист Полиморфизам

```
class Program
{
    class A {
        public virtual void F() {
            Console.WriteLine("A.F");
        }
    }
    class B : A
    {
        public override void F()
        {
            Console.WriteLine("B.F");
        }
    }
    static void Main(string[] args)
    {
        A a = new B();
        a.F();
        //print bf
        Console.ReadLine();
    }
}
```

Метод со **abstract** мора да се **препокрие**

```
abstract class A {
    public abstract void F();
}

class B : A //compile-error program B does not impement inherited memeber
{

}
```

Encapsulation

```
private Paw frontLeft;
```

```

private Paw frontRight;
private Paw bottomLeft;
private Paw bottomRight;

private void MovePaw(Paw paw)
{ // ...
}
public override void Walk()
{ this.MovePaw(frontLeft);
  this.MovePaw(frontRight);
  this.MovePaw(bottomLeft);
  this.MovePaw(bottomRight);
}
}

```

The public method Walk() calls some other private method 4 times. That way the base class is short – it consists of a single method. The implementation, however, calls another of its methods, which is hidden from the users of the class.

Interfaces

```
interface IFile
```

```
{
    void ReadFile();
    void WriteFile(string text);
}
```

```
class FileInfo : IFile
```

```
{
    public void ReadFile()
    {
        Console.WriteLine("Reading File");
    }

    public void WriteFile(string text)
    {
        Console.WriteLine("Writing to file");
    }
}
```

```
public class Program
```

```
{
    public static void Main()
    {
        IFile file1 = new FileInfo();
        FileInfo file2 = new FileInfo();
    }
}
```

```

        file1.ReadFile();
        file1.WriteFile("content");

        file2.ReadFile();
        file2.WriteFile("content");
    }
}
Reading File
Writing to file
Reading File
Writing to file

```

Above, we created objects of the FileInfo class and assign it to IFile type variable and FileInfo type variable. When interface implemented implicitly, you can access IFile members with the IFile type variables as well as FileInfo type variable.

Explicit Implementation

An interface can be implemented explicitly using <InterfaceName>.<MemberName>. Explicit implementation is useful when class is implementing multiple interfaces; thereby, it is more readable and eliminates the confusion. It is also useful if interfaces have the same method name coincidentally.

```

interface IFile
{
    void ReadFile();
    void WriteFile(string text);
}

class FileInfo : IFile
{
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading File");
    }

    void IFile.WriteFile(string text)
    {
        Console.WriteLine("Writing to file");
    }

    public void Search(string text)
    {
        Console.WriteLine("Searching in file");
    }
}

public class Program
{

```

```

public static void Main()
{
    IFile file1 = new FileInfo();
    FileInfo file2 = new FileInfo();

    file1.ReadFile();
    file1.WriteFile("content");
    //file1.Search("text to be searched");//compile-time error

    file2.Search("text to be searched");
    //file2.ReadFile(); //compile-time error
    //file2.WriteFile("content"); //compile-time error
}
}

```

When you implement an interface explicitly, you can access interface members only through the instance of an interface type

In the above example, file1 object can only access members of IFile, and file2 can only access members of FileInfo class. This is the limitation of explicit implementation.

Implementing Multiple Interfaces

A class or struct can implement multiple interfaces. It must provide the implementation of all the members of all interfaces.

Example: Implement Multiple Interfaces

```

interface IFile
{
    void ReadFile();
}

interface IBinaryFile
{
    void OpenBinaryFile();
    void ReadFile();
}

class FileInfo : IFile, IBinaryFile
{
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading Text File");
    }

    void IBinaryFile.OpenBinaryFile()
    {
        Console.WriteLine("Opening Binary File");
    }
}

```

```

void IBinaryFile.ReadFile()
{
    Console.WriteLine("Reading Binary File");
}

public void Search(string text)
{
    Console.WriteLine("Searching in File");
}
}

public class Program
{
    public static void Main()
    {
        IFile file1 = new FileInfo();
        IBinaryFile file2 = new FileInfo();
        FileInfo file3 = new FileInfo();

        file1.ReadFile();
        //file1.OpenBinaryFile(); //compile-time error
        //file1.SearchFile("text to be searched"); //compile-time error

        file2.OpenBinaryFile();
        file2.ReadFile();
        //file2.SearchFile("text to be searched"); //compile-time error

        file3.Search("text to be searched");
        //file3.ReadFile(); //compile-time error
        //file3.OpenBinaryFile(); //compile-time error
    }
}

```

Above, the FileInfo implements two interfaces IFile and IBinaryFile explicitly. It is recommended to implement interfaces explicitly when implementing multiple interfaces to avoid confusion and more readability.

An interface can only declare methods and constants

The members of an interface never specify an access modifier (as all interface members are implicitly public and abstract)

```
public interface IDrawToForm
```



```

{
    void Draw();
}
// Draw to buffer in memory.
public interface IDrawToMemory
{
    void Draw();
}
// Render to the printer.
public interface IDrawToPrinter
{
    void Draw();
}
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
    void IDrawToForm.Draw()
    {
        Debug.WriteLine("Drawing the IDrawToForm...");
    }

    public void Draw()
    {
        // Shared drawing logic.
        Debug.WriteLine("Drawing the Octagon...");
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Octagon oct = new Octagon();
        oct.Draw();//ke se povika Drawing the Octagon
        IDrawToForm oct1 = (IDrawToForm)oct;
        oct1.Draw();//ke se povika Drawing the IDrawToForm
    }
}

```

explicitly implemented members are always implicitly private
implicit are public

Explicit can not be called outside the class cz they are private
and must be casted to be used

```
IDrawToForm oct1 = (IDrawToForm)oct
```

or

```

public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {

```

```

        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}

```

The class member IControl.Paint is only available through the IControl interface, and ISurface.Paint is only available through ISurface. Both method implementations are separate, and neither are available directly on the class. For example:

// Call the Paint methods from Main.

```

SampleClass obj = new SampleClass();
//obj.Paint(); // Compiler error.

```

```

IControl c = obj;
c.Paint(); // Calls IControl.Paint on SampleClass.

```

```

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

```

```

// Output:
// IControl.Paint
// ISurface.Paint

```

abstarct vs interface

od abstrakna klasa ne moze da se napravi objekt
dodeka od interface moze da se napravi objekt kakde toj objekt ke bide od klasata sto go implementira toj interface(toj objekt ke gi ima samo metodite bez data fileds)

Abstract classes Vs Interfaces

Abstract classes can have implementations for some of its members (Methods), but the interface can't have implementation for any of its members.

Interfaces cannot have fields where as an abstract class can have fields.

An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.

A class can inherit from multiple interfaces at the same time, where as a class cannot inherit from multiple classes at the same time.

Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Why use abstract class

```
public class FullTimeEmployee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int AnnualSalary { get; set; }

    public string GetFullName()
    {
        return this.FirstName + " " + LastName;
    }

    public int GetMonthlySalary()
    {
        return this.AnnualSalary / 12;
    }
}
```

```
public class ContractEmployee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int HourlyPay { get; set; }
    public int TotalHoursWorked { get; set; }

    public string GetFullName()
    {
        return this.FirstName + " " + LastName;
    }

    public int GetMonthlySalary()
    {
        return this.HourlyPay * this.TotalHoursWorked;
    }
}
```

Notice that, we have designed FullTimeEmployee and ContractEmployee classes as stand-alone classes. Regardless of the employee type, all employees in the organisation are going to have ID, FirstName and LastName properties. We also compute the FullName of any employee by concatenating their FirstName and LastName. This means that, the above two classes (FullTimeEmployee & ContractEmployee) are related and there is, a lot of common functionality duplicated in them.

Why use abstract class

```
public class FullTimeEmployee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int AnnualSalary { get; set; }

    public string GetFullName()
    {
        return this.FirstName + " " + LastName;
    }

    public int GetMonthlySalary()
    {
        return this.AnnualSalary / 12;
    }
}
```

```
public class ContractEmployee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int HourlyPay { get; set; }
    public int TotalHoursWorked { get; set; }

    public string GetFullName()
    {
        return this.FirstName + " " + LastName;
    }

    public int GetMonthlySalary()
    {
        return this.HourlyPay * this.TotalHoursWorked;
    }
}
```

The problem with this design is that, tomorrow, if want to introduce **MiddleName** property and if we have to include it in the computation of **FullName**, then we have to make the same change in both the classes. So code maintainability is going to be a big issue with this design.

To avoid these issues, we can move the common functionality into a base class. Using a common base class, we are going to get rid of the duplicated code.

Why use abstract class

The obvious next question is, How should we design the base class?

1. Should we design it as an abstract class

OR

2. Should we design it as a Concrete (Non abstract) class

If we create "**BaseEmployee**" class as a concrete (Non abstract) class, there is nothing stopping us from creating an instance of **BaseEmployee** class. We only have 2 types of employees in our organisation - **ContractEmployee** & **FullTimeEmployee**. The developers should only able to instantiate **ContractEmployee** & **FullTimeEmployee** classes and not **BaseEmployee** class.

So, in short, we would create an abstract class, when want to move the common functionality of 2 or more related classes into a base class and when, we don't want that base class to be instantiated.

Var keyword

var cannot be used as field data!

var cannot be used as a return value or parameter type!

Must assign value at exact time of declaration

Can't assign null as initial value!

static int GetAnInt()//it is allowed if is same type var and int

{var retVal = 9; return retVal;}

Virtual

A method, which can be overridden, is called virtual. In .NET, methods are not virtual by default.

In Java, you don't have to write `@Override` in order to override a method with the same signature. It is like that *by default*. That is not the case in C#, as you *have to* prefix `virtual`

A method, which can be overridden in a derived class, is called a virtual method. Methods in .NET by default aren't virtual. If we want to make a method virtual, we mark it with the keyword `virtual`. Then the derived class can declare and define a method with the same signature.

Virtual methods as well as abstract methods can be overridden. Abstract methods are actually virtual methods without a specific implementation. All methods defined in an interface are abstract and therefore virtual, although this is not explicitly defined

```
public class Object // zatoa MOZE DA SE MENUVA ToString
```

```
public Object();
```

```
[...] public virtual bool Equals(object obj);
```

```
[...] public static bool Equals(object objA, object objB);
```

```
[...] public virtual int GetHashCode();
```

```
[...] public Type GetType();
```

```
[...] protected object MemberwiseClone();
```

```
[...] public virtual string ToString();
```

```
A test = new B();
```

- **at compile time:** the compiler only knows that the variable `test` is of the type `A`. He does not know that we are actually giving him an instance of `B`. Therefore the compile-type of `test` is `A`.
- **at run time:** the type of `test` is known to be `B` and therefore has the run time type of `B`

Consider the following code statement:

```
((A)new B()).Test();
```

We are creating an instance of `B` casting it into the type `A` and invoking the `Test()` method on that object. The compiler type is `A` and the runtime type is `B`.

When the compiler wants to resolve the `.Test()` call he has a problem.

Because `A.Test()` is `virtual` the compiler can not simply call `A.Test` because the instance stored might have overridden the method.

The compiler itself can not determine which of the methods to call `A.Test()` or `B.Test()`. The method which is getting invoked is determined by the runtime and not "hardcoded" by the compiler

Virtual -the most derievered

10.5.3 Virtual Methods

When an instance method declaration includes a **virtual** modifier, that method is said to be a **virtual** method. When no **virtual** modifier is present, the method is said to be a **non-virtual** method.

The implementation of a **nonvirtual** method is invariant: The implementation is the same whether the method is invoked on an instance of the **class** in which it is declared or an instance of a **derived class**. In contrast, the implementation of a **virtual** method can be superseded by derived classes. The process of superseding the implementation of an inherited **virtual** method is known as **overriding** that method (§10.5.4).

In a **virtual** method invocation, the **runtime type** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a **nonvirtual** method invocation, the **compile-time type** of the instance is the determining factor. In precise terms, when a method named **N** is invoked with an argument list **A** on an instance with a **compile-time type C** and a **runtime type R** (where **R** is either **C** or a **class** derived from **C**), the invocation is processed as follows.

- Then, if **M** is a **nonvirtual** method, **M** is invoked.
- Otherwise, **M** is a **virtual** method, and the most derived implementation of **M** with respect to **R** is invoked.

but the latest override(of that method (same signature)) is taken into consideration.

Base

:base(some params)

The idea is that the fields of the base class should be initialized before we start initializing fields of the inheriting class, because they might depend on a base class field.

Static

In C#, unlike VB.NET and Java, you can't access static members with instance syntax. use of static methods and constants, which do not belong to any particular object.

All objects, created by the description of a given class (that is, instances of a given class), share the static fields of the class.

This initialization will complete during the first invocation to the static field.

od staticen context moze samo staticni raboti(metod,variabli)

The problem with the access to non-static elements of the class of static method has a

single solution – these non-static elements are accessed by reference to an object

Like static methods, the keyword `this` cannot be used in the static properties, as the static property is associated only with the class and does not “recognize” objects of a class.

Static properties can be accessed only through dot notation, applied to the name of the class in which they are declared.

When a class is declared as static, it is an indication that this class contains only static members (i.e. static fields, methods, properties) and cannot be instantiated.

Static constructors can be declared both in static and in non-static classes. They are executed only once when the first of the following two events occurs for the first time:

1. An object of class is created.

2. A static element of the class is accessed (field, method, property).

Most often static constructors are used for initialization of static fields. A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically. The user has no control on when the static constructor is executed in the program.

A static constructor does not take access modifiers or have parameters.

The static constructor executes before any instance-level constructors

Constructor

If we declare at least one constructor in a given class, the compiler will not create a default constructor for us.

When we do not declare any constructor in a given class, the compiler will create one, known as a default implicit constructor.

If a class has private constructors only, then it cannot be inherited

If a class has private constructors only, then this could indicate many other things. For example, no-one (other than that class itself) can create instances of such a class. Actually, that's how one of the most popular design patterns (Singleton) is implemented.

A class that has only private constructors cannot be instantiated. Such class usually has only static members and is called "utility class"

if we did not specify `public` the constructor is private and can not be instantiated

Singleton

```
public class Singleton {  
    // The single instance private static Singleton instance;  
    // Initialize the single instance  
    static Singleton() { instance = new Singleton(); }  
    // The property for retrieving the single instance
```

```

public static Singleton Instance { get { return instance; } }
// Private constructor: protects against direct instantiation
private Singleton() { }
}

```

We have a hidden (private) constructor in order to limit external instantiations. We have a static variable, which holds the only instance. We initialize it only once in the static constructor of the class. The property for retrieving the single instance is usually called Instance.

Constants

constants always have the same value

the readonly fields are called run-time constants – constants, because their values cannot be changed after assignment and run-time. Fields, declared as readonly, allow one-time initialization either in the moment of the declaration or in the class constructors.

const

compile-time constants, because they are replaced with the value during the compilation process.

They can be accessed without to create an instance (an object) of the class

Although the constants declared with a modifier const are static fields, they must not and cannot use the static modifier in their declaration.

Constants declared with modifier const must be of primitive, enumeration or reference type, and if they are of reference type, this type must be either a string or the value, that we assign to the constant, must be null.

When we want to declare reference type constants, which cannot be calculated during compilation of the program, we must use a combination of static readonly modifiers, instead of const modifier.

The compiletime constants (const) must be initialized at the moment of declaration, while the run-time constants (static readonly) can be initialized at a later stage, for example in one of the constructors of the class in which they are defined.

readonly mora? static ako treba da bide pristapena od druga klasa

Class

Without specifying **public** the **class is** implicitly internal . This **means** that the **class is** only visible inside the same assembly(project). When you specify **public** , the **class is** visible outside the assembly

Just to know, if we want to use a class with access level public from other namespace, different from the current, we should use the reserved word for including different namespaces using or every time we should write the full name of the class
class can be:

1. Static class
2. Abstract class

3. Partial class
4. Sealed class
5. private class only nested

It is also possible to apply the static keyword directly on the class level. When a class has been defined as static, it is not creatable using the new keyword, and it can contain only members or data fields marked with the static keyword. If this is not the case, you receive compiler errors.

Abstract class can not be instantiated.

A class that has only private constructors cannot be instantiated (other than that class itself)

When a class is declared as static, it is an indication that this class contains only static members (i.e. static fields, methods, properties) and cannot be instantiated.

Inner Class

Consider an example. Let's have a class for car – Car. Each car has an engine and doors. Unlike the car's door, however, the engine makes no sense regarded as being outside the car, because without it, the car cannot run, i.e. we have composition

When the connection between the two classes is a composition, the class, which consequently is a part of another class, is convenient to be declared as inner class.

Therefore, if you declare the class for a car: Car would be appropriate to create an inner class Engine, which will reflect the appropriate concept for the car engine:

Sealed Class

The string class has no virtual methods. In fact, inheriting string is entirely forbidden for inheritance through the keyword sealed in its declaration.

public sealed class String //zatoa NE MOZE DA SE NASLEDUVA OD STRING I DA SE
MENUVAAT FUNKCIITE

```
[...] public String(char* value);
[...] public int IndexOf(string value);
[...] public string Normalize();
[...] public string[] Split(params char[] separator);
[...] public string Substring(int startIndex);
[...] public string ToLower(CultureInfo culture);
```

Partial Class

we created a partial class called User in User1.cs class we created a partial class called User in User2.cs class. When you execute the above code, the compiler will combine these two partial classes into one User class the compiler will combine all the partial classes into single class while executing the application in c# programming language.

Utility Class

a class (or structure) that exposes only static functionality is often termed a *utility class*. When designing a utility class, it is good practice to apply the static keyword to the class definition.

Abstract Class

Each class with at least one abstract method must be abstract. Makes sense, right? However, the opposite is not true. It is possible to define a class as an abstract one, even when there are no abstract methods in it.

it is perfectly fine (and common) for abstract classes to define any number of constructors that are called *indirectly* when derived classes are allocated.

Namespaces

the namespaces in C# are named group of classes

String

to change the value, it will be saved to a new location in the dynamic memory and the variable will point to it. Access to the character of a certain position in a string is done with the operator [] (indexer), but it is allowed only to read characters (and not to write to them). The change of either variable will affect only itself because of the immutability of the type string, as when a change occurs, a copy of the changed string will be created. This is not true for the rest of the reference types (the normal, mutable types) because with them the changes are made directly in the address in memory and all references point to this changed address.

Memory Optimization for Strings (Interning)

When not initializing the strings with literals, no interning is used. However, if we want to use interning specifically, we can make it through the use of the method `Intern(...)`:

Override

It is used to modify a virtual or abstract method

Exception

Exceptions in .NET are two types – system and application. System exceptions are defined in .NET libraries and are used by the framework, while application exceptions are defined by application developers and are used by the application software. When we, as developers, design our own exception classes, it is a good practice to inherit from ApplicationException and not directly from SystemException (or even worse – directly from Exception). SystemException should only be inherited internally within the .NET Framework.

Stack and Heap

Stack is used for static **memory** allocation and **Heap** for dynamic **memory** allocation, both stored in the computer's **RAM**. Variables allocated on the **stack** are stored directly to the **memory** and access to this **memory** is very fast, and it's allocation is dealt with when the program is compiled

ArrayList

array list i list imaat kapacitet 4 koga ke se stavi barem edno
koga ke se napravi insert vo arraylist se pretvara vo object
vo C# moze primitivni tipovi da se cuvaat vo List i vo ArrayList i vo staticarray

List

list e genericka
array list i list imaat kapacitet 4 koga ke se stavi barem edno
vo C# moze primitivni tipovi da se cuvaat vo List i vo ArrayList i vo staticarray

Task

A Task can be seen as a convenient and easy way to execute something asynchronously and in parallel. In other programming languages and frameworks this may be known as a promise - "I promise will return to you at some point". A task will by default use the Threadpool, which saves resources as creating threads can be expensive. A threadpool is..

a pool of threads, which are ready to carry out instructions (if they are not occupied of course). You can see a Task as a higher level abstraction upon threads. Which could be a reason why they are under the System.Threading namespace.

Inheritance properties

In addition to Yacoub's answer, in this case, Enemy would not contain the properties, and methods that Ogre has.

```
public class Enemy
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
}
```

```
public class Ogre : Enemy
{
    public int Property3 { get; set; }
}
```

Let's say you inherit Enemy in your Ogre class. This means that your Ogre will effectively contain 3 properties: 1, 2 and 3.

In your example you're assigning an Ogre to an Enemy type. The Enemy type doesn't contain a "Property3" and therefore you won't be able to work with the extended class "Ogre" in an Enemy cast object.

```
//This will work
Ogre newOgre = new Ogre();
int newInt = newOgre.Property3;
```

```
//This won't.
Enemy newOgre = new Ogre();
int newInt = newOgre.Property3;
```

LINQ

LINQ comes in two syntactical flavors: The Query and the **Method** syntax. They can do almost the same, but while the query syntax is almost a new language within C#, the **method** syntax looks just like regular C# **method** calls.

In LINQ, Method Syntax is used to call the extension methods of the Enumerable or Queryable static classes. It is also known as **Method Extension Syntax** or **Fluent**. However, the compiler always converts the query syntax in method syntax at compile time