Module "C#"

Submodule "C# Essentials"

OOP Part 1

UA Resource Development Unit

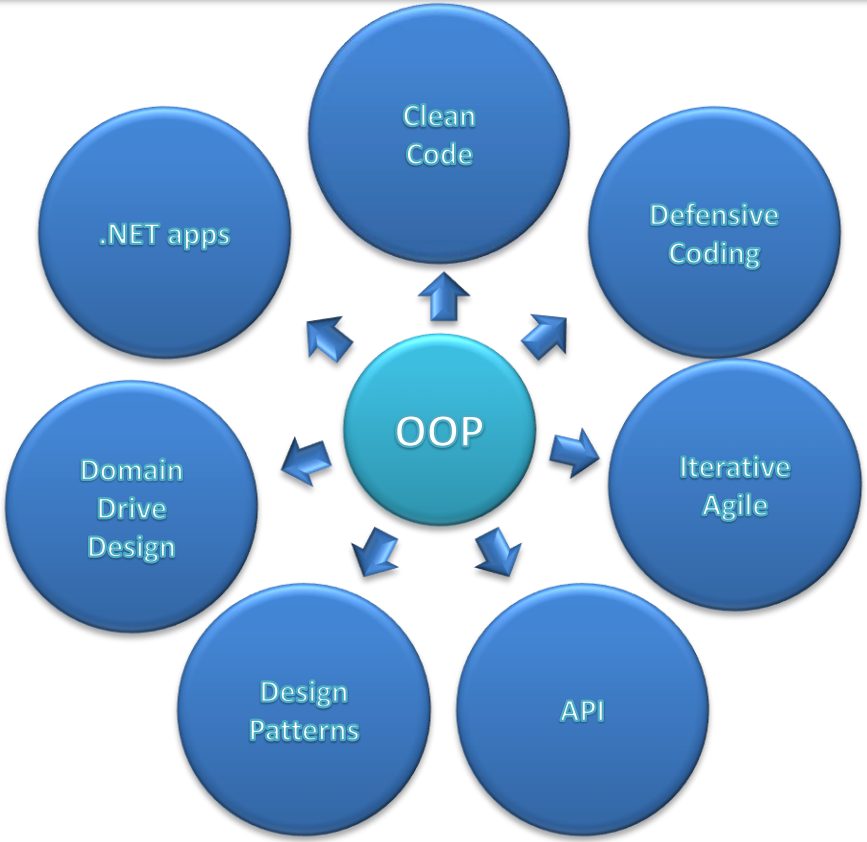
2021

AGENDA

1. OOP principles.
2. Class & Object. Constructors. Methods. Access modifiers.
3. Class instance destroying. Finalizer.
4. Static vs Instance classes and members. Properties and indexers.
5. Inheritance. Polymorphism.
6. Abstract classes.
7. Upcasting and downcasting.
8. new and override methods.
   Sealed methods and classes.

OOP principles

OOP is the Foundation



What is OOP?

OOP stands for Object-Oriented Programming.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "**D**on't **R**epeat **Y**ourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "**D**on't **R**epeat **Y**ourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application and place them at a single place and reuse them instead of repeating it.

Concepts of OOP

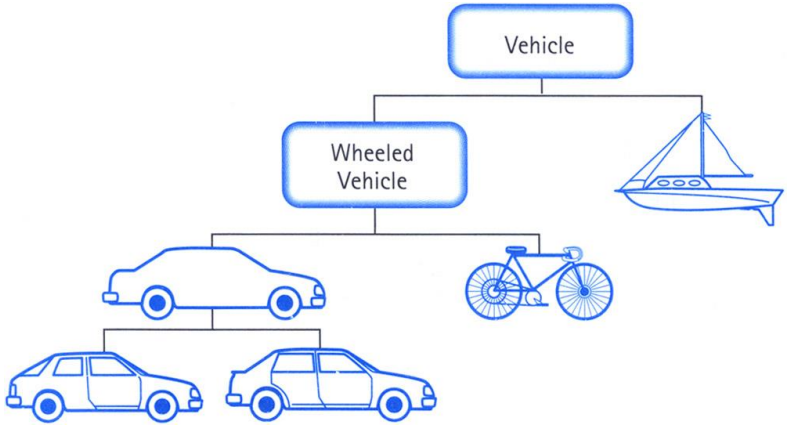
Object-oriented programming (OOP) is a programming methodology constructed around objects. OOP is associated with concepts such as

- Class
- Object
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism
- etc.

Definition of OOP

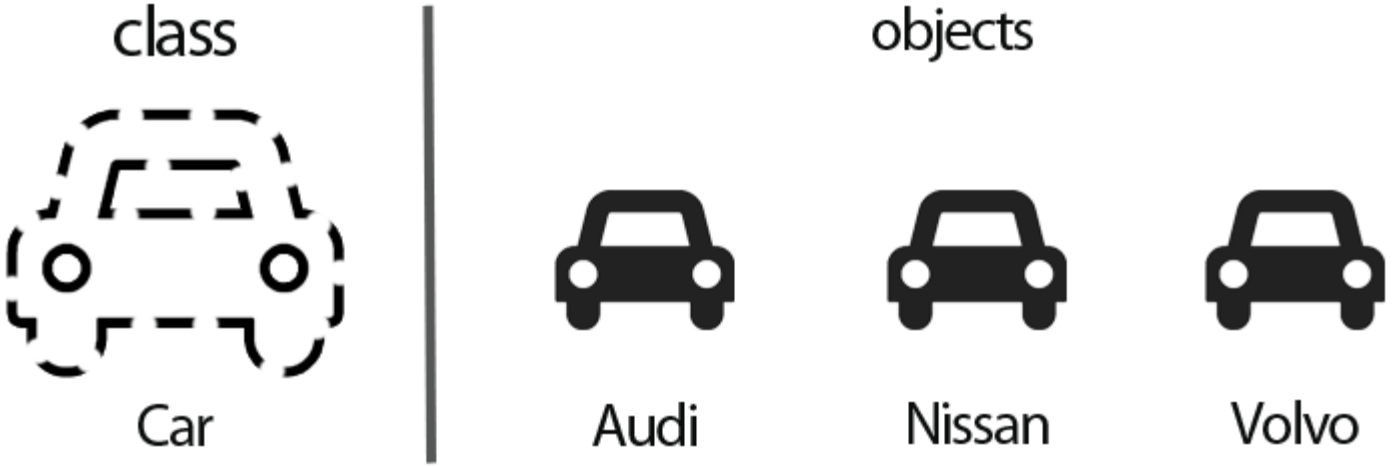
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this"). In OOP, computer programs are designed by making them out of objects that interact with one another.

https://en.wikipedia.org/wiki/Object-oriented_programming

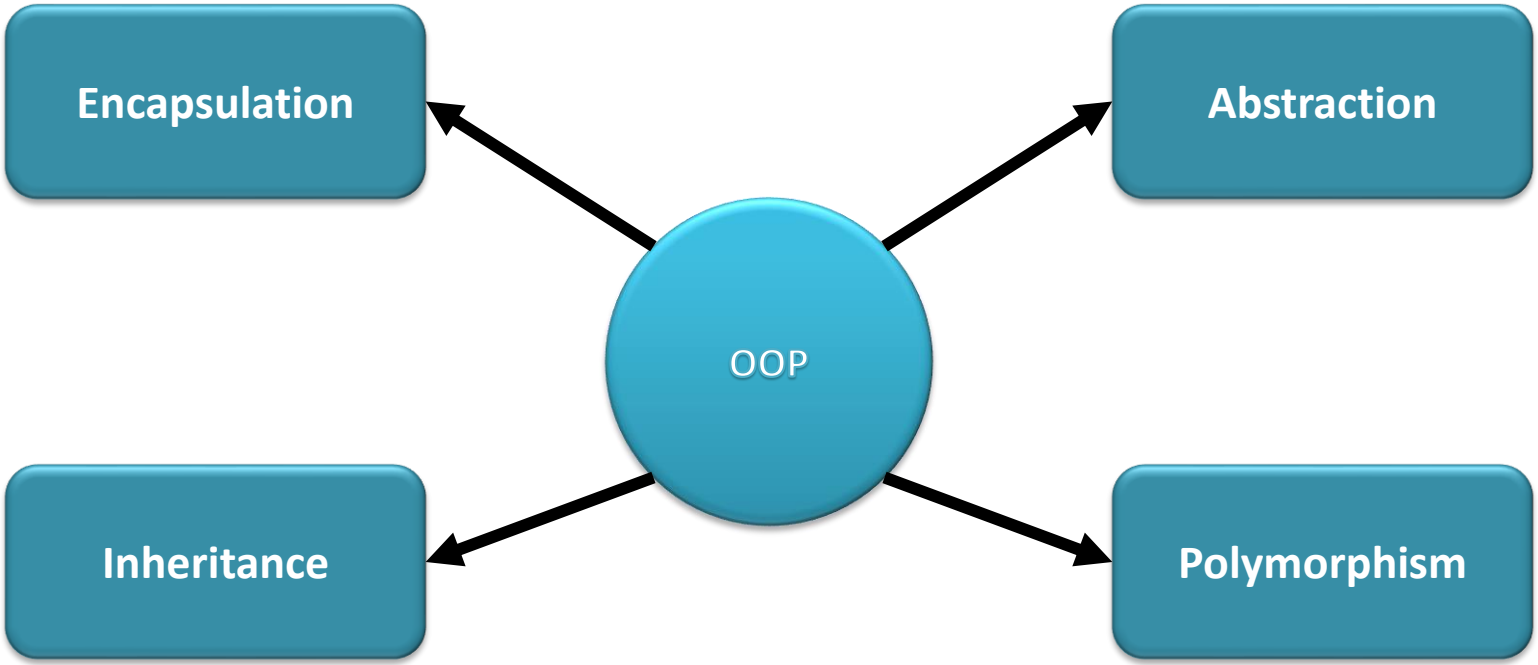


OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Class vs object



Four pillars of OOP



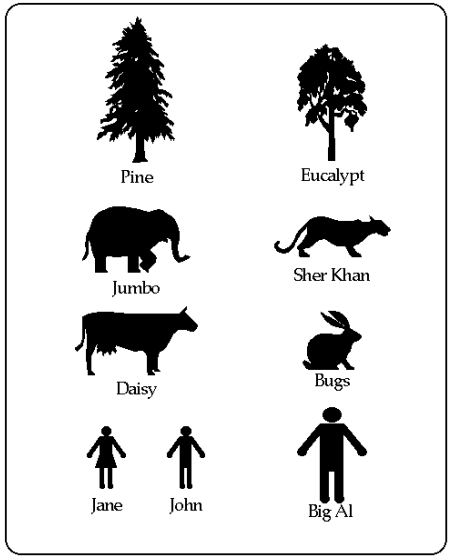
Abstraction

Abstract means a concept or an Idea which is not associated with any particular instance. Using abstract class/Interface we express the intent of the class rather than the actual implementation. In a way, one class should not know the inner details of another in order to use it, just knowing the interfaces should be good enough.

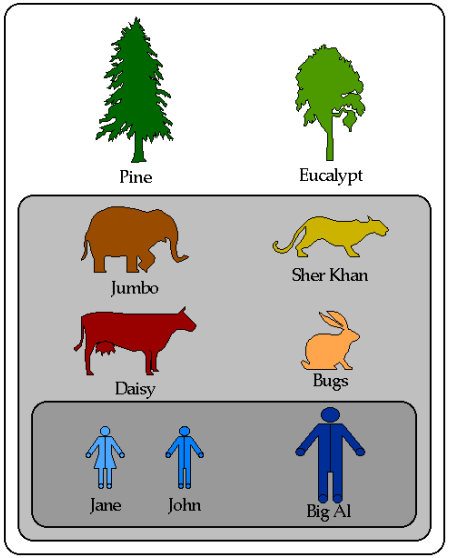
An abstraction includes the essential details relative to the perspective of the viewer

- ✓ Focus on essentials
- ✓ Ignore the irrelevant
- ✓ Ignore the unimportant

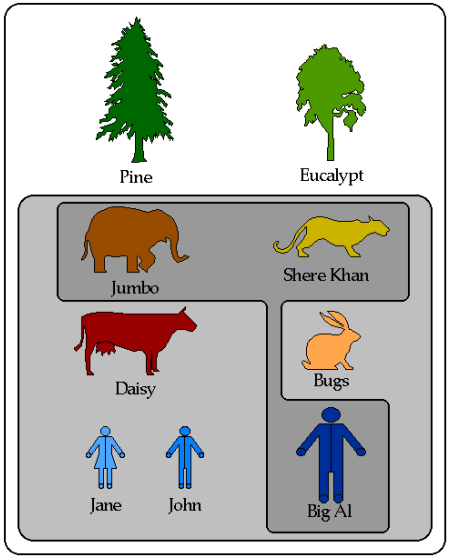
Abstraction



unclassified "things"



organisms,
mammals, humans



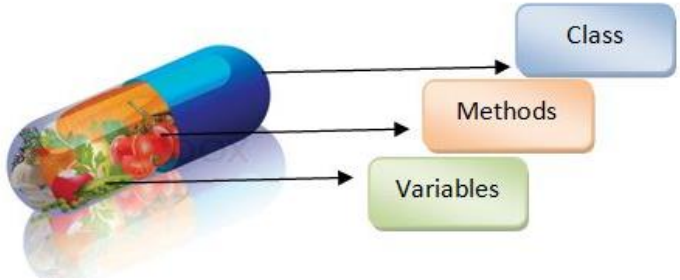
organisms, mammals,
dangerous mammals

Encapsulation

Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this.

Key Points of Encapsulation

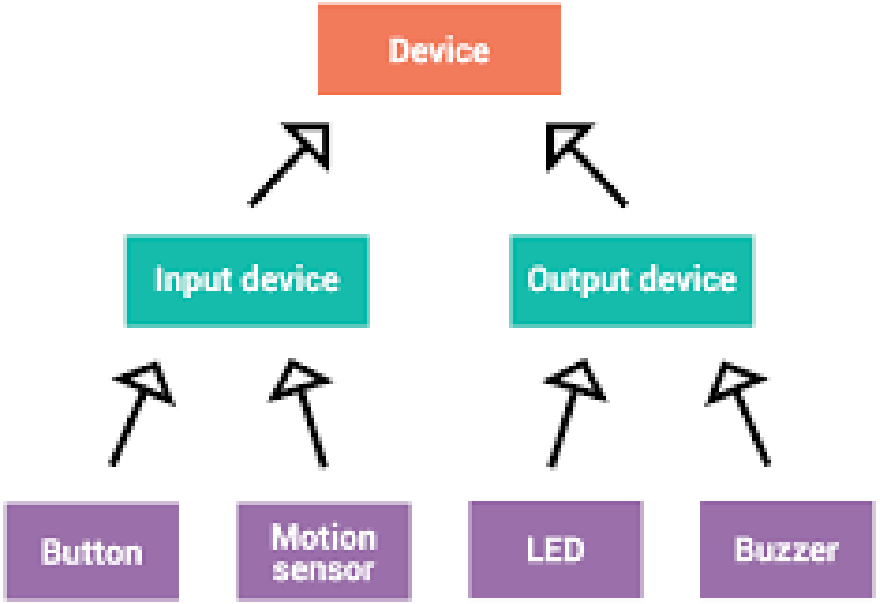
- Protection of data from accidental corruption
- Specification of the accessibility of each of the members of a class to the code outside the class
- Flexibility and extensibility of the code and reduction in complexity
- Encapsulation of a class can hide the internal details of how an object does something
- Using encapsulation, a class can change the internal implementation without affecting the overall functionality of the system
- Encapsulation protects abstraction

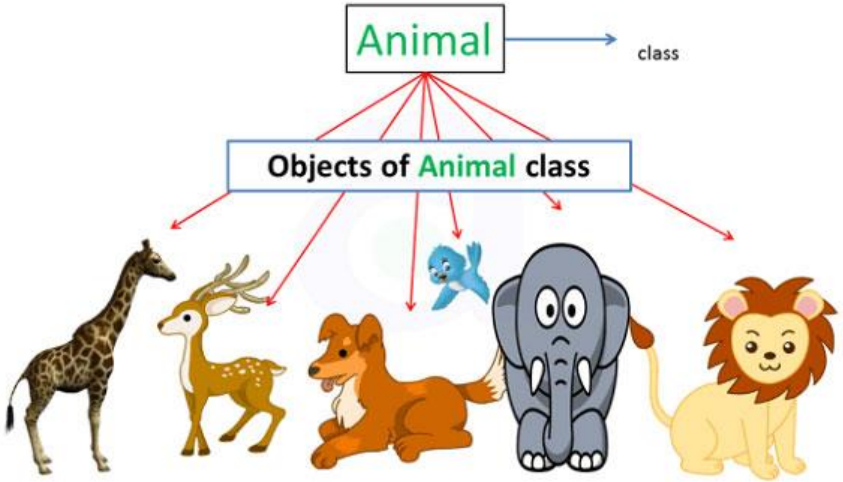


Encapsulation — private instance variable and public accessor methods.

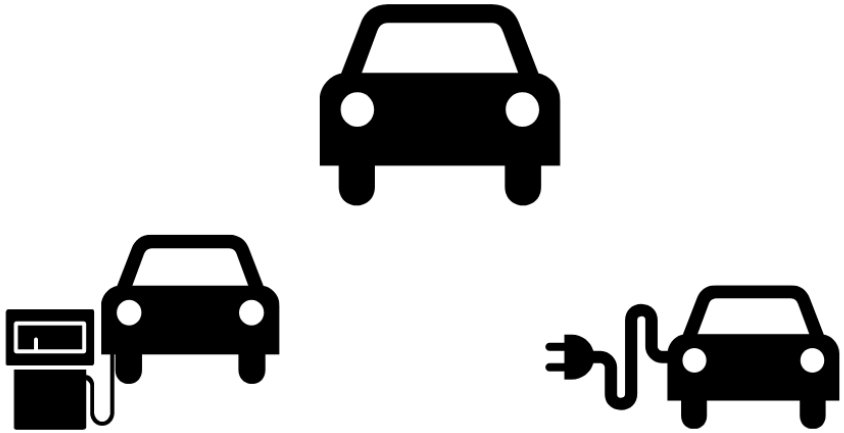
Inheritance

Inheritances expresses “is-a” and/or “has-a” relationship between two objects. Using Inheritance, In derived classes we can reuse the code of existing super classes.



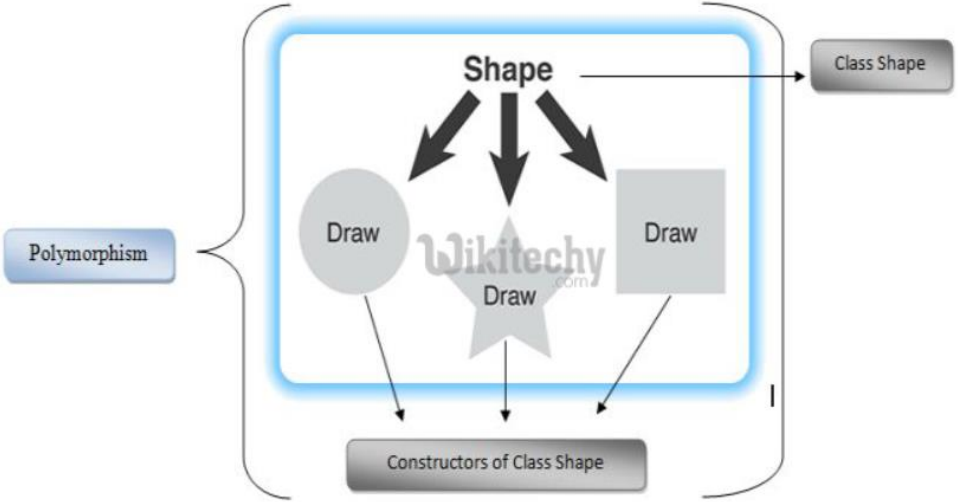


Polymorphism



ONE NAME FOR MANY FORMS

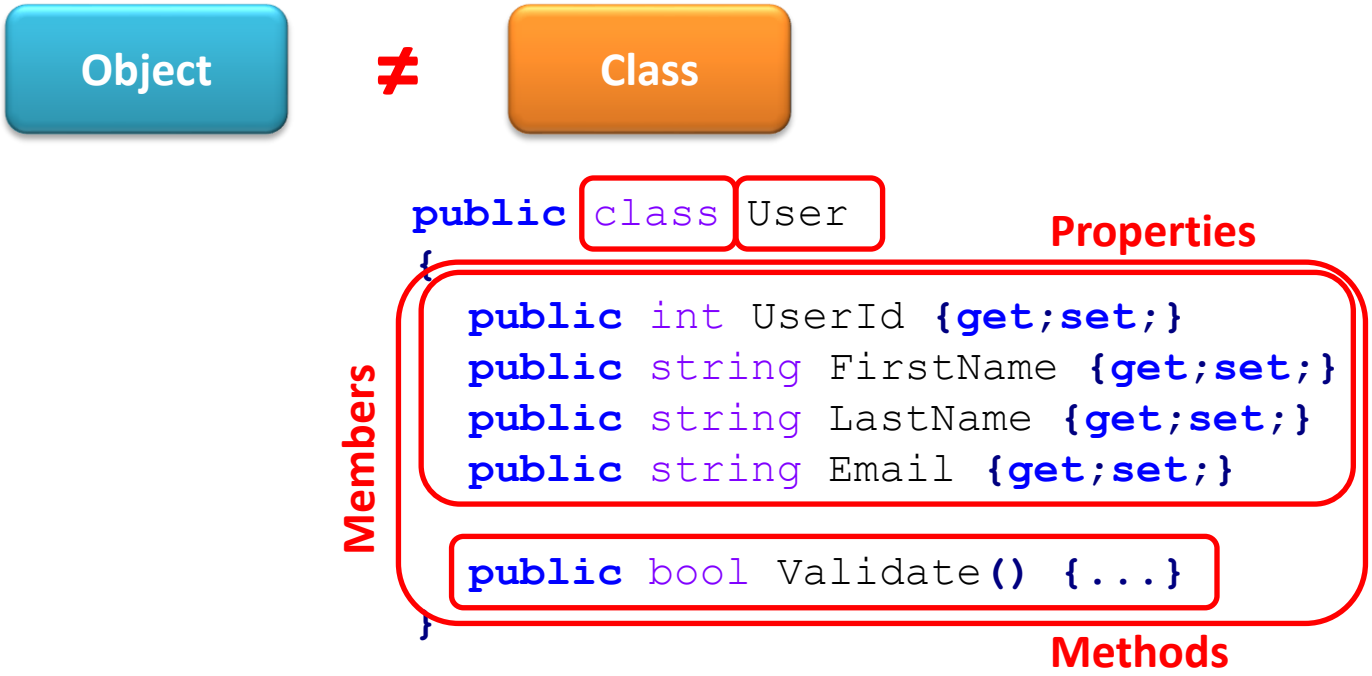
POLYMORPHISM



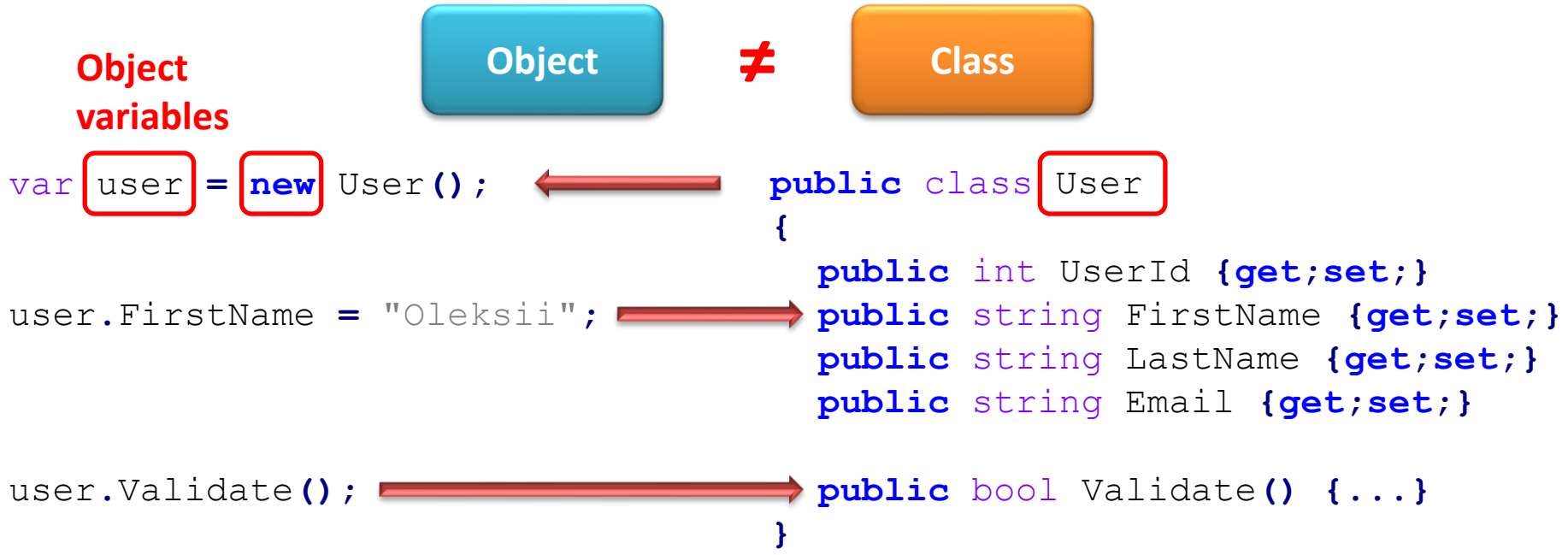
Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Class & Object
Constructors
Methods
Access modifiers

Class & Object

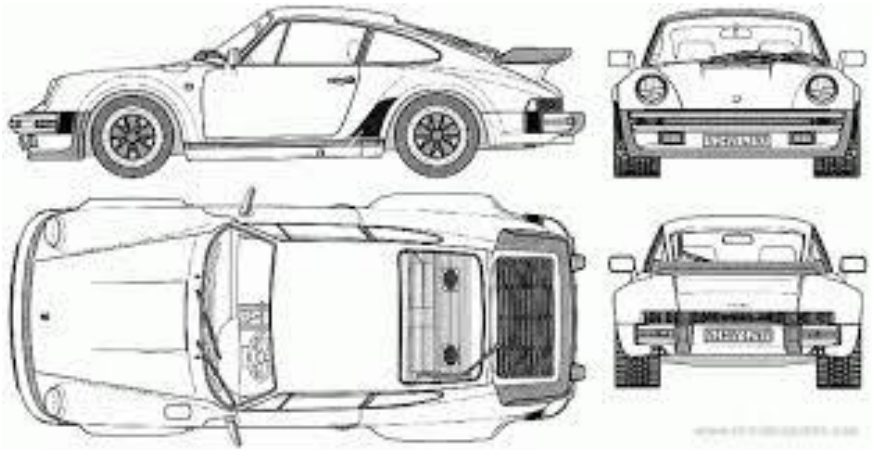


Class & Object

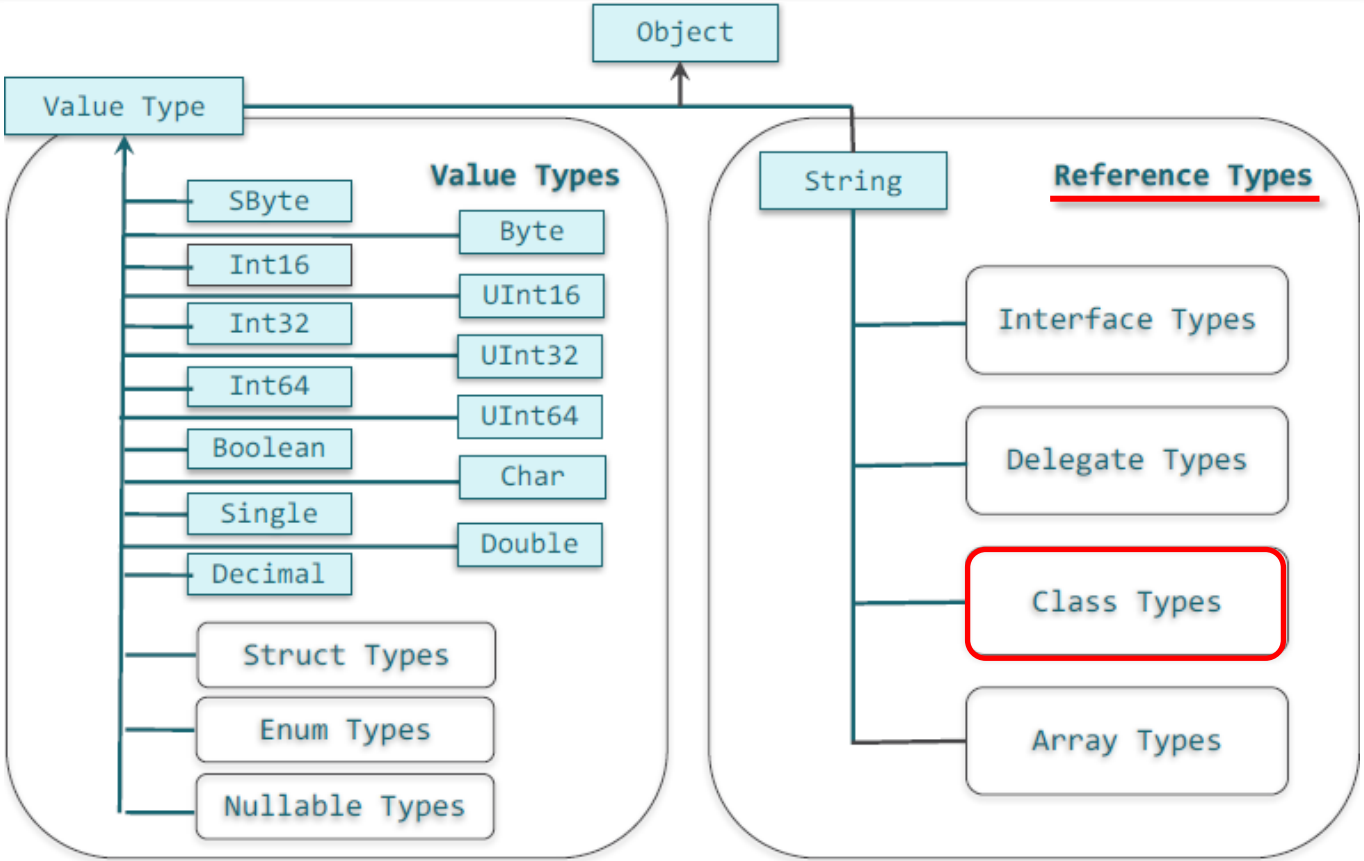


Class & Object

Object ≠ Class

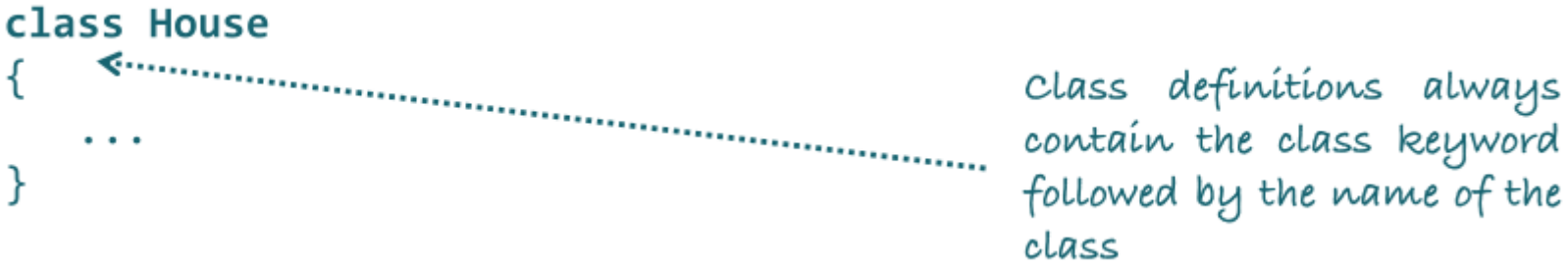


Types


Object
Value Type
Value Types
SByte
Byte
Int16
UInt16
Int32
UInt32
Int64
UInt64
Boolean
Char
Single
Double
Decimal
Struct Types
Enum Types
Nullable Types
String
Reference Types
Interface Types
Delegate Types
Class Types
Array Types

Class

A class can contain both code and data, and it can choose to make some of its features publicly available, while keeping other features accessible only to code within the class. So classes offer a mechanism for encapsulation—they can define a clear public programming interface for other people to use, while keeping in ternal implementation details inaccessible.



Class

```
[Attributes]
[Class modifiers] class ClassName [Generic type parameters, a base
                                   class, and interfaces]
{
        Class members – these are methods, properties, indexers,
                events, fields, constructors, overloaded operators,
                nested types, and a finalizer

}
```

public, internal, abstract, sealed, static, unsafe, partial

Access modifiers

Access modifiers are keywords used to specify the declared accessibility of a member or a type.

The following access modifiers are available:

- **public** - The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- **private -** The type or member can only be accessed by code in the same class.
- **protected** - The type or member can only be accessed by code in the same class or in a derived class.
- **internal** - The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal** - The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.
- **private protected** - The type or member can be accessed by code in the same class or in a derived class within the base class assembly.

Class

A **class** is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A **class** is like a blueprint. It defines the data and behavior of a type

- Classes are reference types
- Reference are copied on assignment.
- Can be instantiated only with using a **new** operator.
- Unlike structures, class can have explicit parameterless constructor
- Can inherit implementation from one explicit base class only. However, a class can implement more than one interface.

```
2 references
public class Customer
{
    // Fields, properties, methods and events go here...
}

0 references
static void Main()
{
    // Declaration of instance of Customer class
    Customer object1;

    // Declaration and initialization of instance of Customer class
    var object2 = new Customer();

    var object3 = object2;
}
```

Fields

A field is a variable of any type that is declared directly in a class or struct. Fields are members of their containing type.

```
2 references
public class Customer
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    0 references
    public DateTime Date
    {
        get { return date; }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year) date = value;
            else throw new ArgumentOutOfRangeException();
        }

    }
}
```

Class members. Fields

A field is a variable that is a member of a class or struct

Static modifier	static
Access modifier	public internal private protected
Inheritance modifier	new
Unsafe code modifier	unsafe
Read-only access modifier	readonly
The modifier of multithreading	volatile

Default value of fields

Type	Default value
bool	false
byte, int, sbyte, short, uint, ulong, ushort	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
long	0L
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null.
object	null

Const fields and readonly fields

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the **const** modifier.

```
1 reference
class Calendar
{
    // Multiple constants of the same type can be declared at the same time (Generally not recommended)
    // const int months = 12, weeks = 52, days = 365;

    const int months = 12;
    const int weeks = 52;
    const int days = 365;

    public const double daysPerWeek = (double)days / (double)weeks;
    public const double daysPerMonth = (double)days / (double)months;

    // Readonly field
    public readonly DateTime now = DateTime.Now;

    // Readonly field also can be assigned only in constructor
    0 references
    public Calendar()
    {
        now = DateTime.Now;
    }
}
```

The **readonly** keyword is a modifier that you can use on fields. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class.

Class members. Methods

A method is a procedure or function inside a class

Static modifier	static
Access modifier	public internal private protected
Inheritance modifier	new virtual abstract override sealed
Unmanaged code modifier	unsafe extern
Partial method modifier	partial
Asynchronous code modifier	async

Methods

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments.

```
2 references
abstract public class Motorcycle
{
    // Only this class can call this
    0 references
    private void Move() { /* Method statements here */ }

    // Anyone can call this.
    0 references
    public void StartEngine() { /* Method statements here */ }

    // Only derived classes can call this.
    0 references
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    0 references
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.

    public abstract double GetTopSpeed();
}
```

Methods

```
1 reference
public class MyMotorcycle : Motorcycle
{
    1 reference
    public override double GetTopSpeed()
    {
        AddGas(5);
        /* Method statements here */ return 1.5;
    }
}

0 references
static void Main()
{
    var motorcycle = new MyMotorcycle();
    motorcycle.
}
```

```
Drive                    int Motorcycle.Drive(int miles, int speed)
Equals
GetHashCode
GetTopSpeed
GetType
StartEngine
ToString
```

Access Modifiers

- **public**: Access is not restricted.
- **protected**: Access is limited to the containing class or types derived from the containing class.
- **internal**: Access is limited to the current assembly.
- **protected internal**: Access is limited to the current assembly or types derived from the containing class.
- **private**: Access is limited to the containing type.

Default and named parameters

```
0 references
static void Main()
{
    Console.WriteLine("1:");
    Calculate(1, 2);

    Console.WriteLine("2:");
    Calculate(b: 2, a: 1);

    Console.WriteLine("3:");
    Calculate(3, 6, "Other Value");

    Console.WriteLine("4:");
    Calculate(6, 3, secondDefault: "Other Value");
}
```

```
// Output:
//      1:
//      Result of (a + 1) * b = 4
//      firstDefault = First Default Value
//      secondDefault = Second Default Value
//      2:
//      Result of (a + 1) * b = 4
//      firstDefault = First Default Value
//      secondDefault = Second Default Value
//      3:
//      Result of (a + 1) * b = 24
//      firstDefault = Other Value
//      secondDefault = Second Default Value
//      4:
//      Result of (a + 1) * b = 21
//      firstDefault = First Default Value
//      secondDefault = Other Value
```

```
4 references
private static void Calculate(int a, int b, string firstDefault = "First Default Value", string secondDefault = "Second Default Value")
{
    var result = (a + 1) * b;

    Console.WriteLine("Result of (a + 1) * b = {0}", result);
    Console.WriteLine("firstDefault = {0}", firstDefault);
    Console.WriteLine("secondDefault * b = {0}", secondDefault);
}
```

“params” keyword

```
0 references
static void Main()
{
    // You can send a comma-separated list of arguments of the specified type.
    UseParams(1, 2, 3, 4);
    UseParams2(1, 'a', "test");

    // A params parameter accepts zero or more arguments.
    // The following calling statement displays only a blank line.
    UseParams();

    // An array argument can be passed, as long as the array
    // type matches the parameter type of the method being called.
    int[] myIntArray = { 5, 6, 7, 8, 9 };
    UseParams(myIntArray);

    object[] myObjArray = { 2, 'b', "test", "again" };
    UseParams2(2.2, myObjArray);
}

            // Output:
            //      1 2 3 4
            //      a test

            //
            //      5 6 7 8 9
            //      2 b test again
```

```
3 references
public static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
```

```
2 references
public static void UseParams2(double otherProperty, params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
```

“ref” and “out” parameters

- The **ref** keyword indicates a value that is passed by reference.
- The **out** keyword causes arguments to be passed by reference. It is like the **ref** keyword, except that **ref** requires that the variable be initialized before it is passed.

```
1 reference
static void Method(ref int i)
{
    i = i + 44;
}

0 references
static void Main()
{
    int val = 1;
    Method(ref val);
    Console.WriteLine(val);
}
// Output: 45
```

```
1 reference
static void Method(out int i, out string s1, out string s2)
{
    i = 44;
    s1 = "I've been returned";
    s2 = null;
}

0 references
static void Main()
{
    int value;
    string str1, str2;
    Method(out value, out str1, out str2);
}
// Output: 45
//      value is now 44
//      str1 is now "I've been returned"
//      str2 is (still) null;
```

Properties

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.

- Auto properties
- Classical properties

```
0 references
public class Person
{
    private int age;

    0 references
    public string Name { get; set; }

    0 references
    public int Age
    {
        get { return age; }
        set
        {
            if (value > 0)
            {
                this.age = value;
            }
        }
    }
}
```

```
0 references
static void Main()
{
    var person1 = new Person { Age = 3, Name = "Vlad" };

    var person2 = new Person() { Age = 3, Name = "Vlad" };

    var person3 = new Person();
    person3.Age = 3;
    person3.Name = "Vlad";
}
```

```
// public string Name { get { return name; } }
public string Name => name;
```

Properties with different access level

```
0 references
static void Main()
{
    var person = new Person();

    Console.WriteLine("Name is: {0}", person.Name);
    person.DoSomething();
    Console.WriteLine("Name is: {0}", person.Name);

    person.Name = "Ivan";
}
```

```
1 reference
public class Person
{
    4 references
    public string Name { get; private set; }

    // Also can be declared as:
    // public string Name { get; }

    1 reference
    public void DoSomething()
    {
        // Some code...
        Name = "Vlad";
    }
}
```

string Person.Name { get; private set; }

The property or indexer 'Program.Person.Name' cannot be used in this context because the set accessor is inaccessible

```
// Output:
//      Name is:
//      Name is: Vlad
```

Constructors

Whenever a class or struct is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

```
static void Main()
{
    CoOrds p1 = new CoOrds();
    CoOrds p2 = new CoOrds(5, 3);

    // Display the results using the overriden ToString method:
    Console.WriteLine("CoOrds #1 at ({0},{1})", p1.x, p1.y);
    Console.WriteLine("CoOrds #2 at ({0},{1})", p2.x, p2.y);
}
```

- Instance Constructors
- Private Constructors
- Static Constructors

```
// Output:
//      CoOrds #1 at (0,0)
//      CoOrds #2 at (5,3)
```

```
6 references
class CoOrds
{
    public int x, y;

    // Default constructor:
    1 reference
    public CoOrds()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments:
    1 reference
    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Class members. Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type

```
public class Residence
{
    public Residence(ResidenceType type, int numberOfBedrooms)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms, bool hasGarage,
                     bool hasGarden)
    {
    }
}
```

CLR calls the constructors automatically

Class members. Instance Constructors

```
public class Residence
{
    private ResidenceType type;
    private int numberOfBedrooms;
    private bool hasGarage;
    private bool hasGarden;
    private Residence(ResidenceType type, int numberOfBedrooms, bool
hasGarage, bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }
    public Residence() : this(ResidenceType.House, 3, true, true{ }
    ...
}
```

Private Constructors

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class.

```
0 references
class TestCounter
{
    0 references
    static void Main()
    {
        Counter.CurrentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.CurrentCount);

        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        Counter aCounter = new Counter();
    }
    // Output: New count: 101
}
```

```
6 references
public class Counter
{
    1 reference
    private Counter() { }

    3 references
    public static int CurrentCount { get; set; }

    1 reference
    public static int IncrementCount()
    {
        return ++CurrentCount;
    }
}
```

class ConsoleApp1.Program.Counter (+ 1 overload)

'Program.Counter.Counter()' is inaccessible due to its protection level

Objects Creating

```
public class Employee
{
    private int id;
    private string name;
    private static CompanyPolicy policy;

    public virtual void Work()
    {
        Console.WriteLine("Zzzz...");
    }

    public void TakeVacation(int days)
    {
        Console.WriteLine("Zzzz...");
    }

    public static void SetCompanyPolicy(CompanyPolicy plc)
    {
        policy = plc;
    }
}
```

Instance fields

Static field

Instance virtual method

Instance method

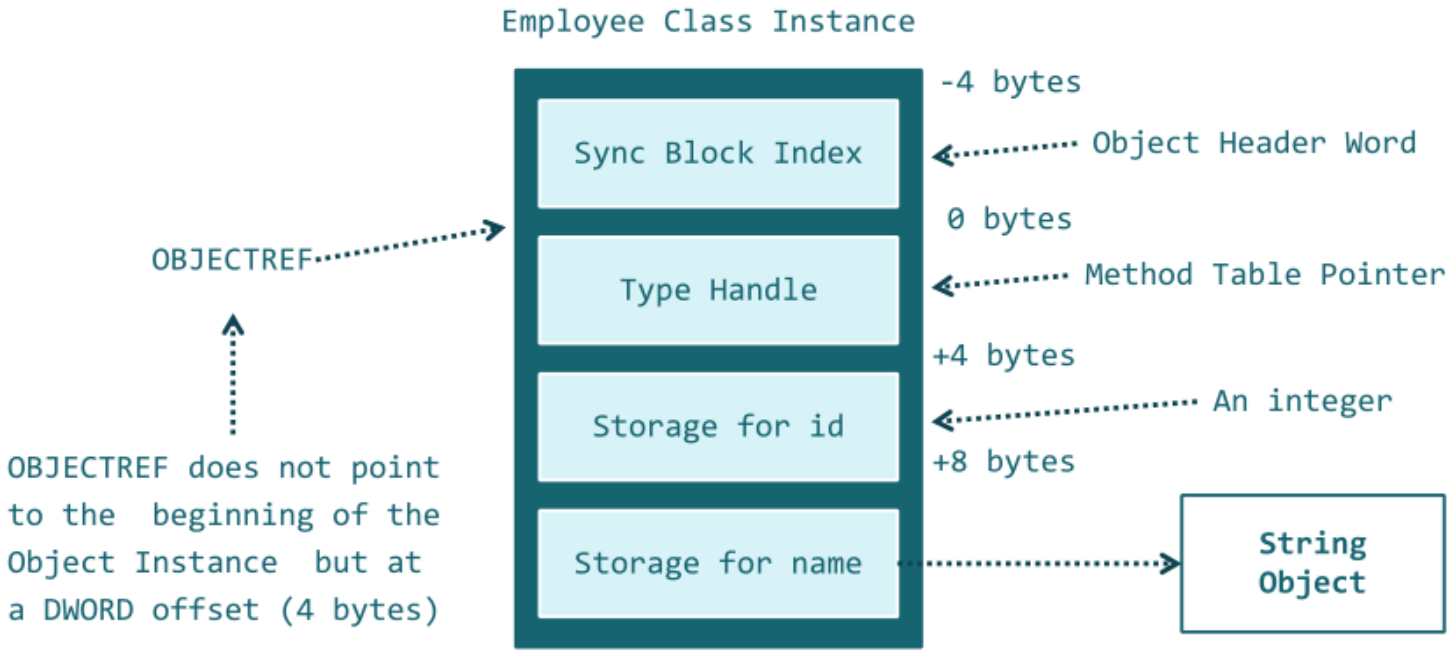
Static method

Objects Creating

Execution Engine as in MSCOR**EE**.dll

- EE allocates memory for the object
- The EE initializes the pointer to the method table
- EE lays the pointer to the object in the *ecx register* and passes control to the constructor specified in the *newobj* instruction that generated the object creation code
- If no unhandled exceptions occurred during the constructor's operation, the reference to the object is placed in one or another scope variable, from which the object creation code was called

Objects Creating



Using regions for grouping class members

```
public class Counter
{
    #region Constructors
    1 reference
    private Counter() { }
    #endregion

    #region Properties
    3 references
    public static int CurrentCount { get; set; }
    #endregion

    #region Methods
    1 reference
    public static int IncrementCount()
    {
        return ++CurrentCount;
    }

    0 references
    public void OtherMethod()
    {
        // Do somenthing
    }
    #endregion
}
```

```
6 references
public class Counter
{
    #region Constructors
    1 reference
    private Counter() { }
    #endregion

    Properties

    Methods
}
```

#region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. In longer code files, it is convenient to be able to collapse or hide one or more regions so that you can focus on the part of the file that you are currently working on.

Partial classes

It is possible to split the definition of a class or a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- To split a class definition, use the **partial** keyword modifier.

Partial Types and Methods

Partial types allow a type definition to be split—typically across multiple files. A common scenario is for a partial class to be auto-generated from some other source (such as a Visual Studio template or designer) and for that class to be augmented with additional hand-authored methods

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

 // PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

 // PaymentForm.cs - hand-authored
class PaymentForm { ... }
```

Partial types are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

Example of Partial classes

```
// file CoOrds1.cs
4 references
public partial class CoOrds : ICoOrds
{
    3 references
    public int Y { get; set; }

    1 reference
    public CoOrds(int x, int y)
    {
        X = x;
        Y = y;
    }
}

// file CoOrds2.cs
4 references
public partial class CoOrds : BaseCoOrds
{
    1 reference
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", X, Y);
    }
}
```

```
1 reference
public class BaseCoOrds
{
    2 references
    public int X { get; set; }
}

1 reference
public interface ICoOrds
{
    3 references
    int Y { get; set; }
}

0 references
class TestCoOrds
{
    0 references
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();
    }
}
// Output: CoOrds: 10,15
```

Partial methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

```
// file CoOrds1.cs
3 references
public partial class CoOrds
{
    1 reference
    partial void DoSomething();
}

// file CoOrds2.cs
3 references
public partial class CoOrds
{
    1 reference
    partial void DoSomething()
    {
        // method body
    }
}
```

Requirements to partial methods:

- Method declaration must begin with the contextual keyword **partial** and the method must return **void**.
- Method can have **ref** but not **out** parameters.
- Method are implicitly **private**, and therefore they cannot be **virtual**.
- Method can have **static** and **unsafe** modifiers.
- Method can be generic.

Partial Types and Methods

A partial type may contain partial methods. These let an auto-generated partial type provide customizable hooks for manual authoring

```
partial class PaymentForm // In auto-generated file
{ ...
        partial void ValidatePayment (ref decimal amount);
}
```

definition

```
partial class PaymentForm // In hand-authored file
{ ...
        partial void ValidatePayment (ref decimal amount)
        {
                if (amount > 100) ...
        }
}
```

implementation

Implicitly private

Anonymous classes

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

```
0 references
static void Main()
{
    var v = new { Amount = 108, Message = "Hello" };

    // Rest the mouse pointer over v.Amount and v.Message in the following
    // statement to verify that their inferred types are int and string.
    Console.WriteLine(v.Amount + " " + v.Message);

    // Anonymous type properties are readonly
    v.Amount = 25;
}
```

```
int 'a.Amount { get; }

Anonymous Types:
  'a is new { int Amount, string Message }

Property or indexer '<anonymous type: int Amount, string Message>.Amount' cannot be assigned to -- it is read only
```

Tuples

A tuple is a data structure that has a specific number and sequence of elements.

Tuples are commonly used in four ways:

- To represent a single set of data.
- To provide easy access to, and manipulation of, a data set.
- To return multiple values from a method without using out parameters.
- To pass multiple values to a method through a single parameter.

```
static void Main()
{
    // Create a 4-tuple.
    var population = new Tuple<string, int, int, int>("New York", 7891957, 7781984, 7894862);
    // Display the first and last elements.
    Console.WriteLine("Population of {0} in 2000: {1:N0}", population.Item1, population.Item4);
}
// Output:
//      Population of New York in 2000: 7,894,862
```

Class instance destroying. Finalizer

Class instance destroying

```
public class PathInfo
{
    public string DirectoryName { get; }
    public string FileName { get; }
    public string Extension { get; }

    public PathInfo(string path)
    {
        ...
    }

    public void Deconstruct( out string directoryName, out string fileName,
                             out string extension)
    {
        directoryName = DirectoryName;
        fileName = FileName;
        extension = Extension;
    }
    // ...
}
```

Class instance destroying

Value types

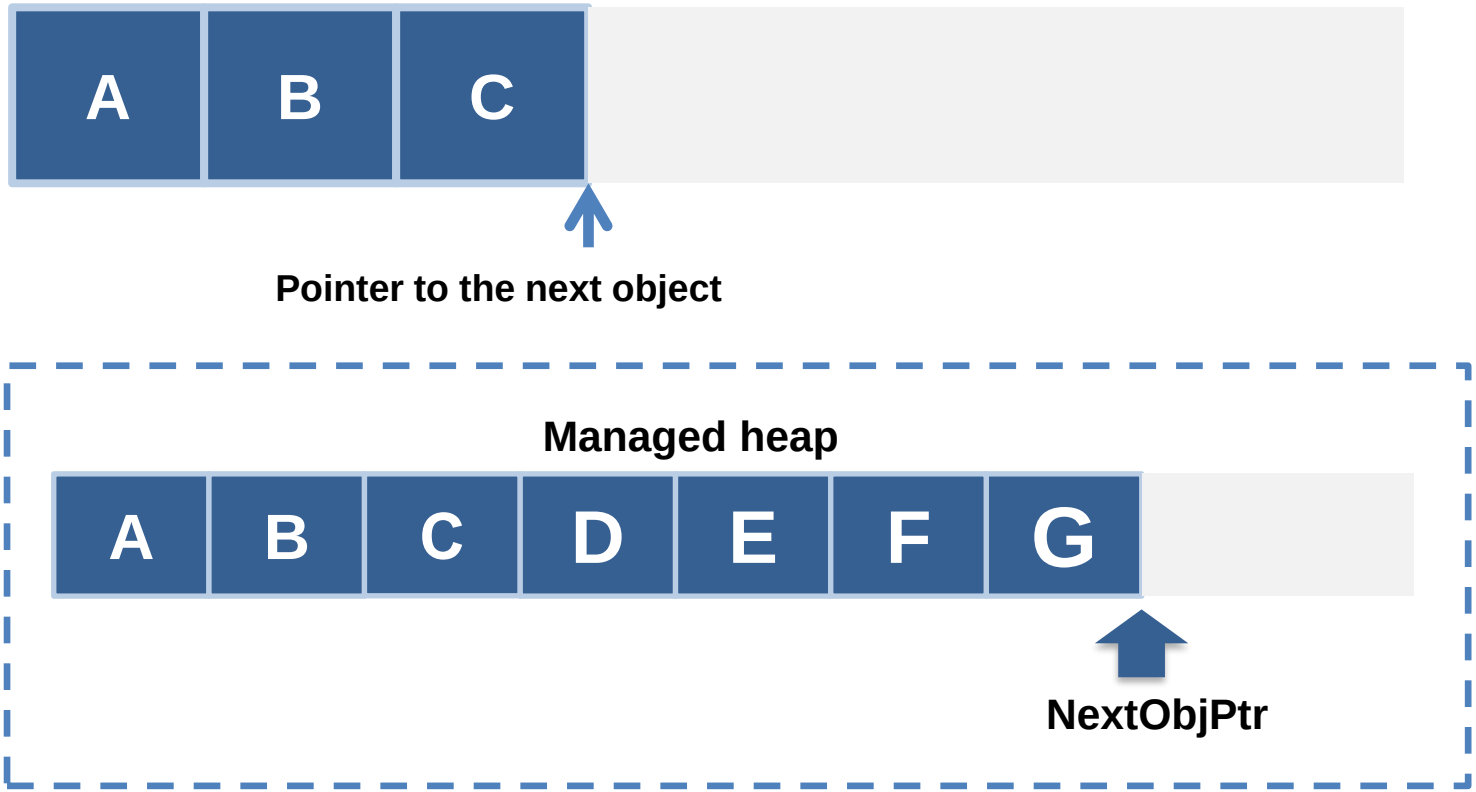
```
private void MyMethod(…)
{
     MyStruct a = …;
     DateTime z = …;
     double y = …;
     int x = …;
     …
}
```

Class instance destroying

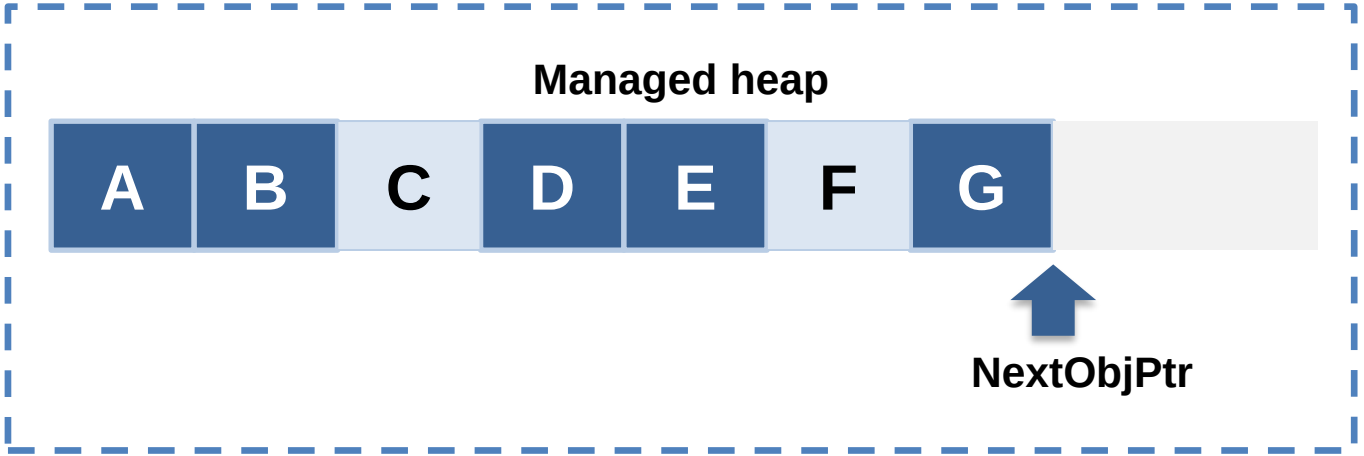
References types

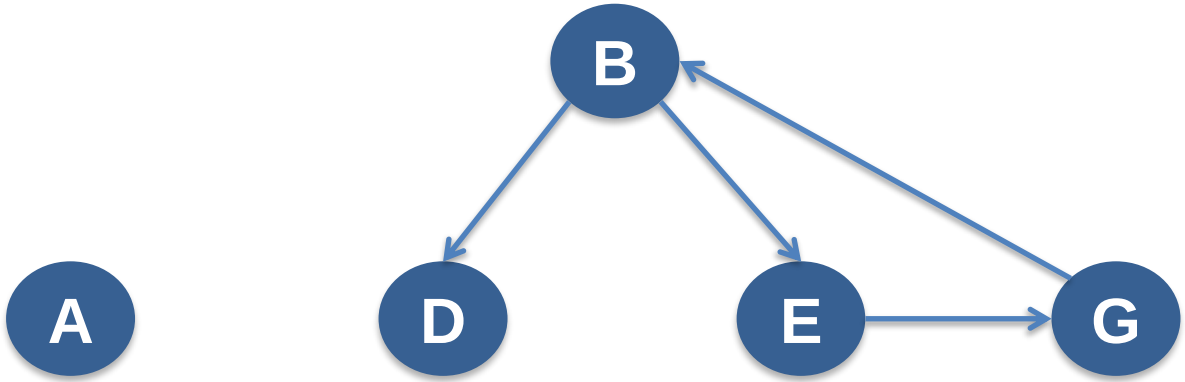
```
string s = …;
MyClass c = …;
private void MyMethod2(…)
{
    string t = s;
    MyClass d = c;
    …
}
```

Working GC

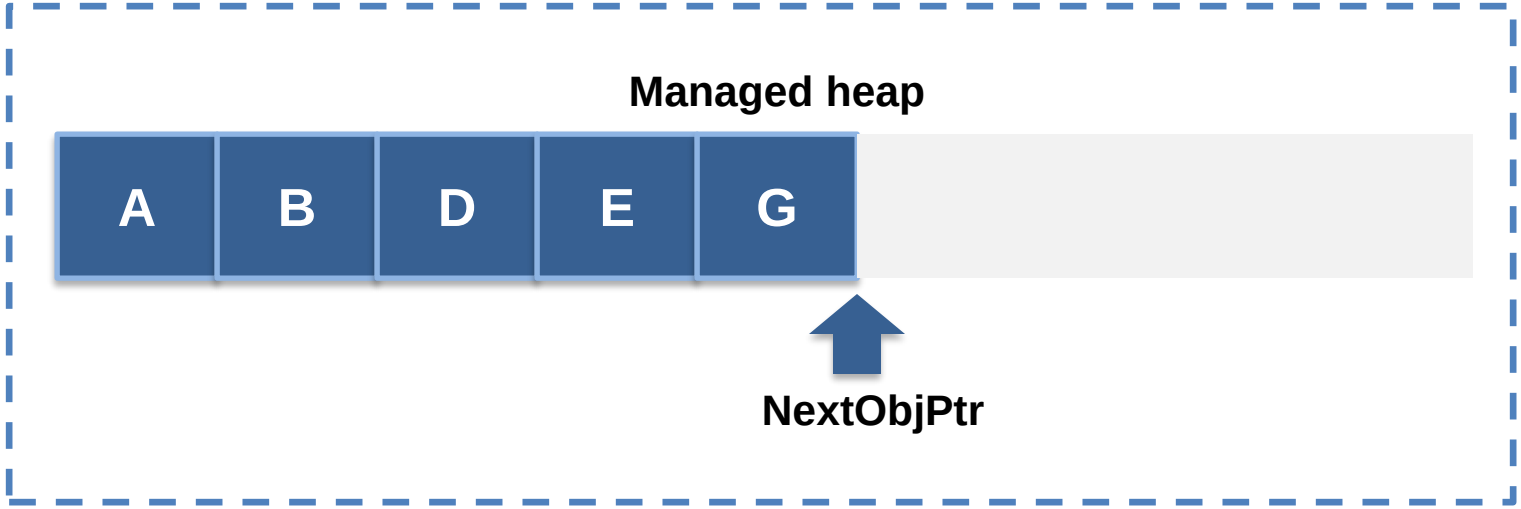

A
B
C
Pointer to the next object
Managed heap
A
B
C
D
E
F
G
NextObjPtr

Working GC

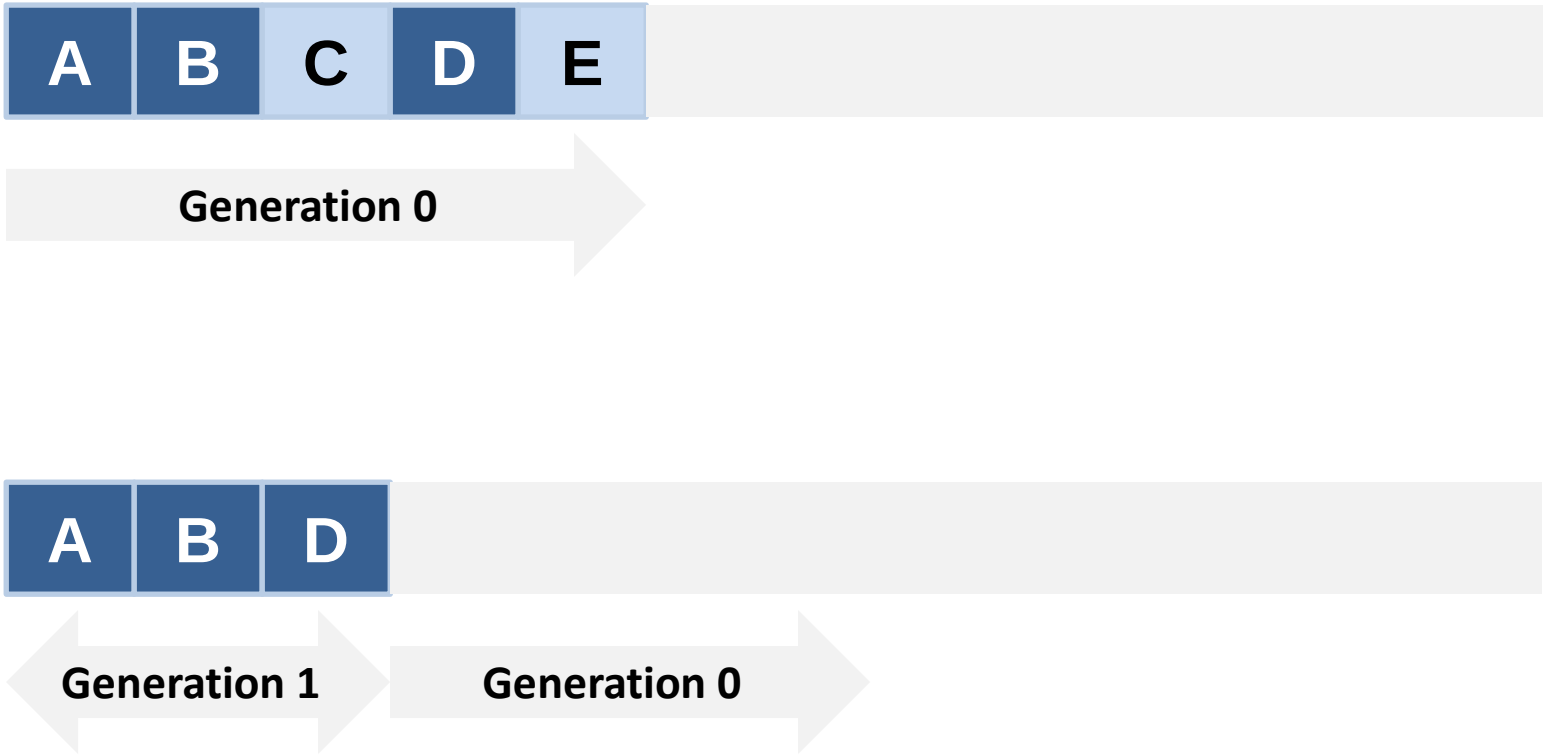




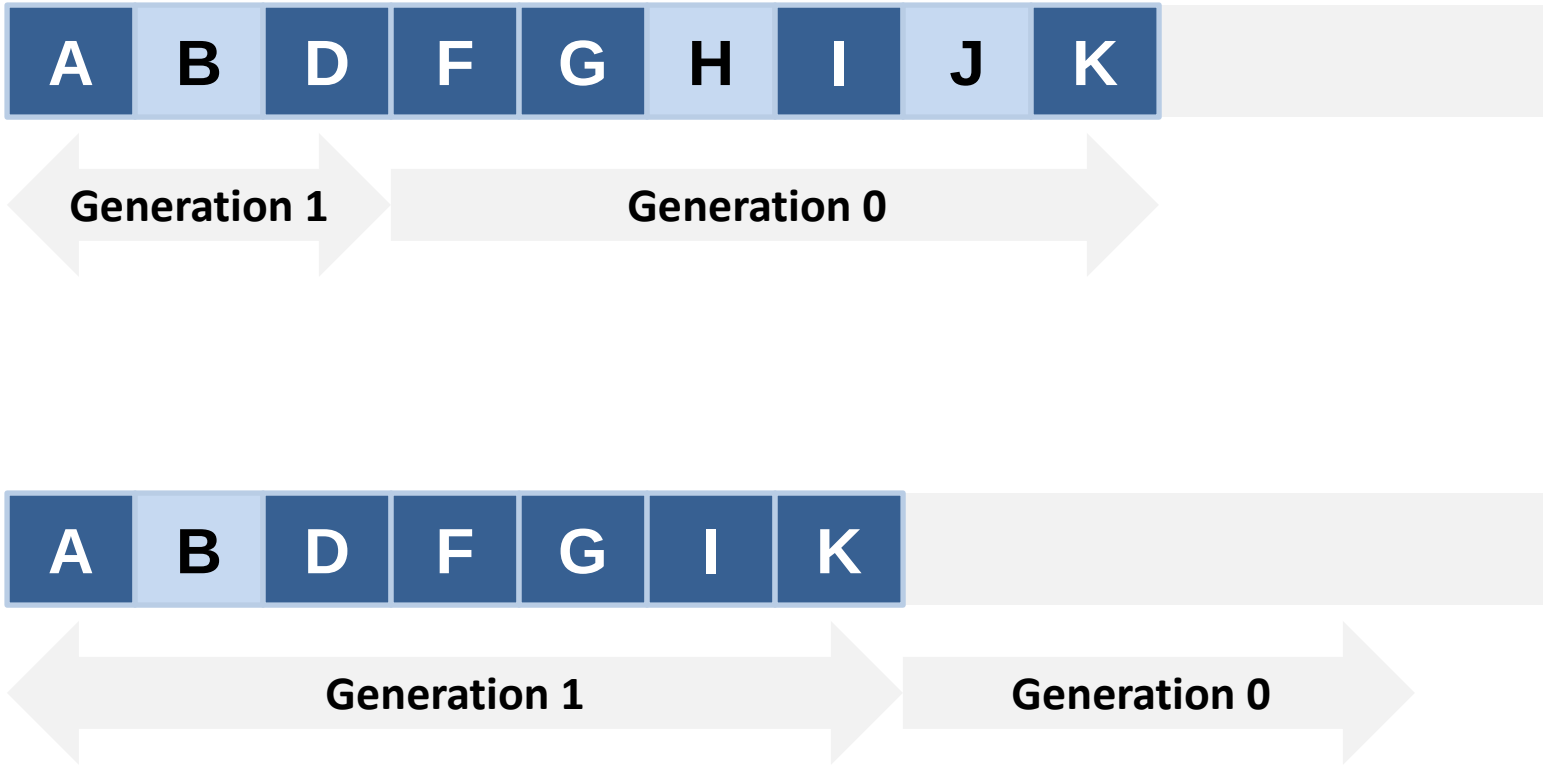
Working GC



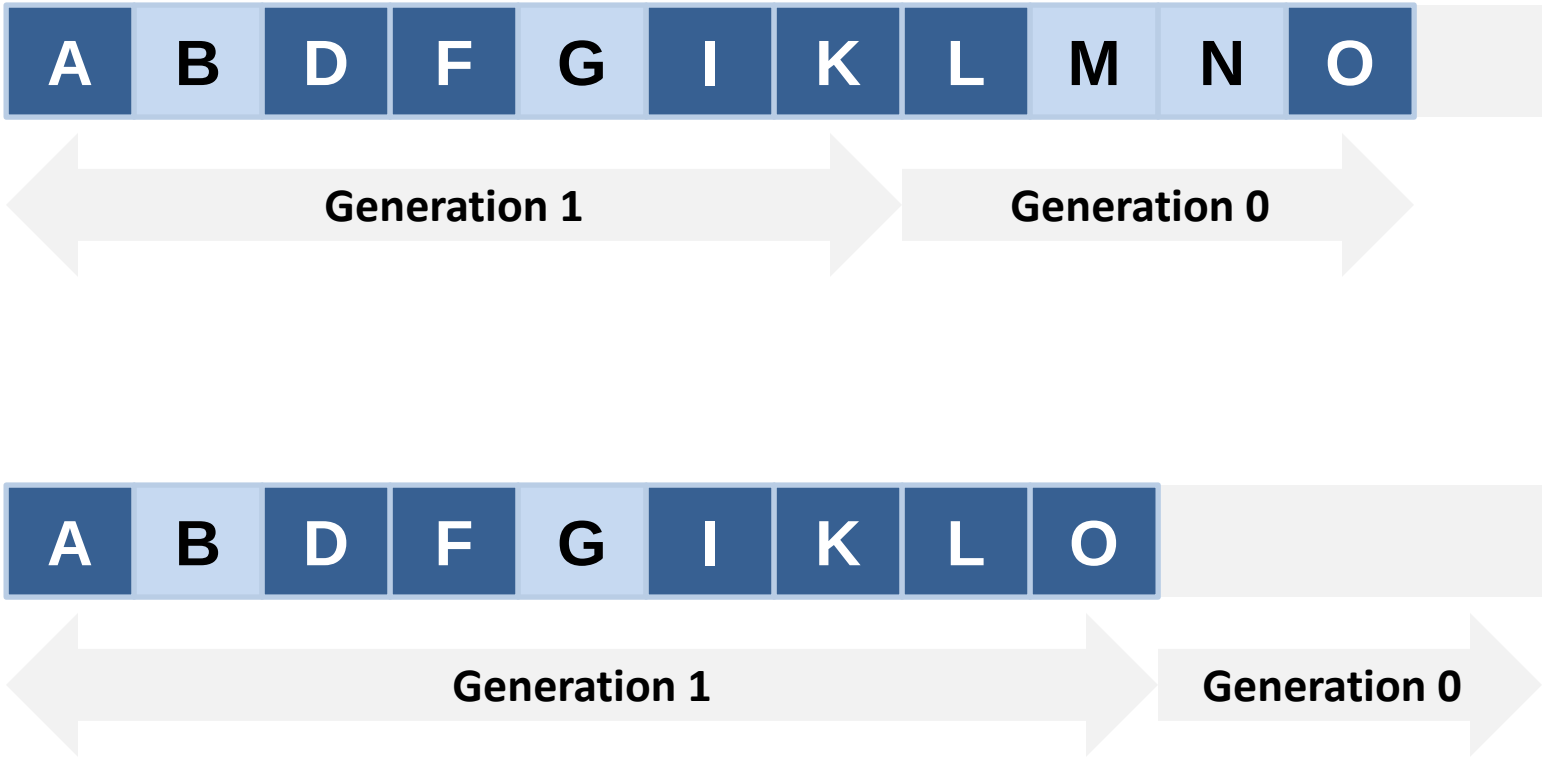
Working GC



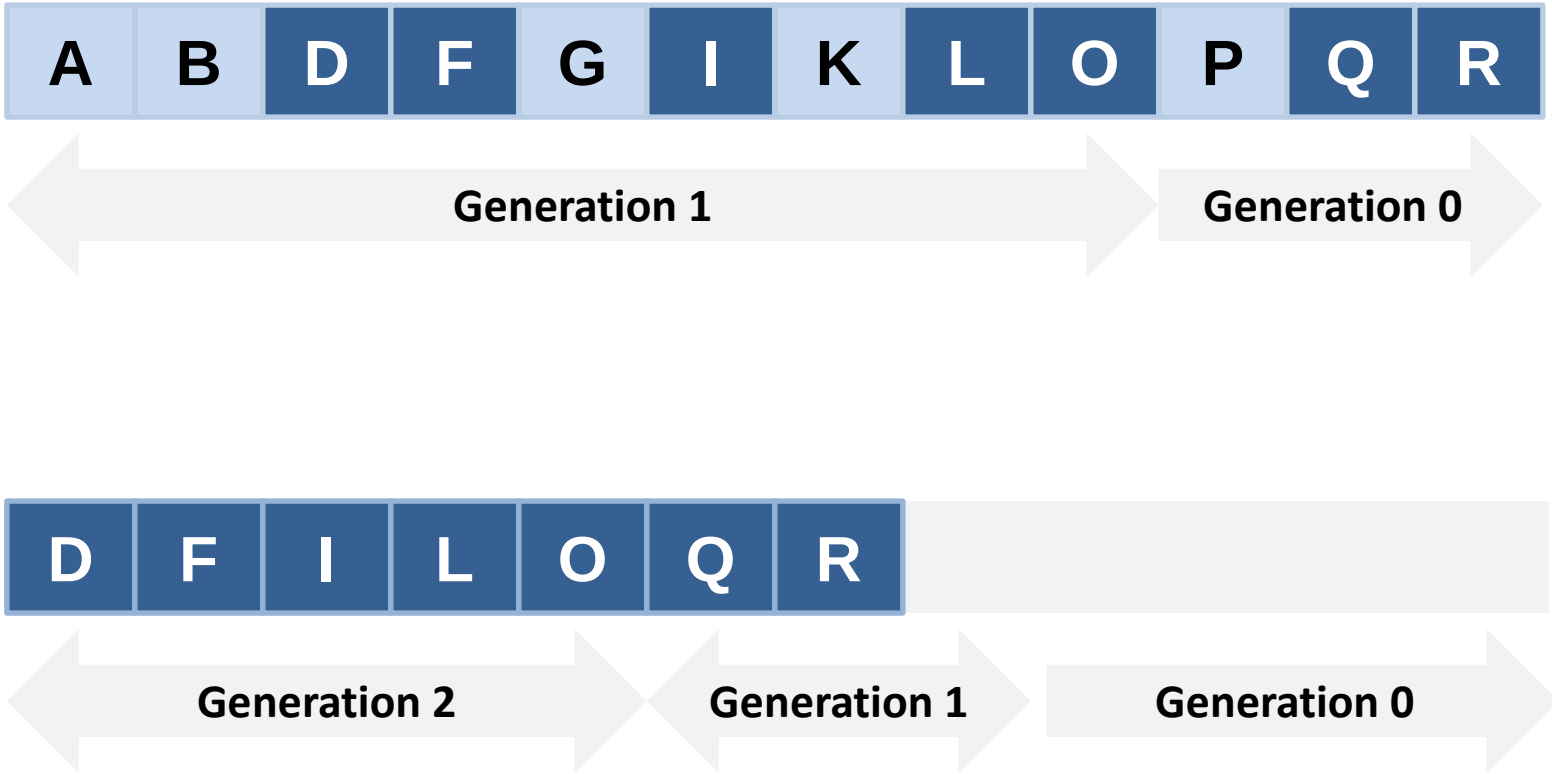
Working GC


A
B
D
F
G
H
I
J
K
Generation 1
Generation 0
A
B
D
F
G
I
K
Generation 1
Generation 0

Working GC


A
B
D
F
G
I
K
L
M
N
O
Generation 1
Generation 0
A
B
D
F
G
I
K
L
O
Generation 1
Generation 0

Working GC


A
B
D
F
G
I
K
L
O
P
Q
R
Generation 1
Generation 0
D
F
I
L
O
Q
R
Generation 2
Generation 1
Generation 0

Class GC

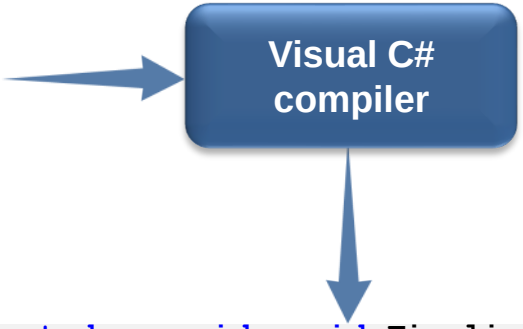
Method	Description
Collect	Forces an immediate garbage collection of all generations.
WaitForPendingFinalizers	Suspends the current thread until the thread that is processing the queue of finalizers has emptied that queue.
SupressFinalize	Requests that the common language runtime not call the finalizer for the specified object.
ReRegisterForFinalize	Requests that the system call the finalizer for the specified object for which SuppressFinalize(Object) has previously been called.
AddMemoryPressure	Informs the runtime of a large allocation of unmanaged memory that should be taken into account when scheduling garbage collection.
RemoveMemoryPressure	Informs the runtime that unmanaged memory has been released and no longer needs to be taken into account when scheduling garbage collection.

Destructor

```
class Employee
{
    ...
    // Destructor
    ~Employee
    {
        // Destructor logic.
    }
}
```

Class instance destroying

```
class Employee
{     ...
     // Destructor
     ~Employee
     {
          // Destructor logic.
     }
}
```




```
protected override void Finalize()
{
     try
     {
          // Destructor logic.
     }
     finally
     {
          base.Finalize();
     }
}
```

interface IDisposable

```
public interface IDisposable
{
    void Dispose();
}
```

Dispose Method

```
class MyResourceWrapper: IDisposable
{
    public void Dispose()
    {
        . . .
    }
}
. . .
MyResourceWrapper rw = new MyResourceWrapper();
rw.Dispose();
```

```
FileStream fs = new FileStream("myFile.txt",
FileMode.OpenOrCreate);
fs.Close();
fs.Dispose();
```

Static vs Instance classes and members. Properties and indexers.

Static classes and static class members

A **static** class is basically the same as a non-static class, but there is one difference: a **static** class cannot be instantiated. In other words, you cannot use the **new** keyword to create a variable of the class type.

```
public static class MyStaticClass
{
    public static int MyProperty { get; set; }

    public static double DoSomething()
    {
        // Some Code
        return MyProperty;
    }
}

public class EntryPoint
{
    static void Main()
    {
        MyStaticClass.MyProperty = 15;
        var d = MyStaticClass.DoSomething();
        Console.WriteLine("d is {0}", d);
    }
    // Output:
    //      d is 15
}
```

```
static void Main()
{
    MyNonStaticClass.MyProperty = 15;

    var obj1 = new MyNonStaticClass();
    Console.WriteLine("obj1.MyProperty is {0}", obj1.DoSomething());
    var obj2 = new MyNonStaticClass();
    Console.WriteLine("obj2.MyProperty is {0}", obj2.DoSomething());

    MyNonStaticClass.MyProperty = 10;

    Console.WriteLine("Now obj1.MyProperty is {0}",
        obj1.DoSomething());
    Console.WriteLine("Now obj2.MyProperty is {0}",
        obj2.DoSomething());

    Console.ReadKey();
}
// Output:
//          obj1.MyProperty is 15
//          obj2.MyProperty is 15
//          Now obj1.MyProperty is 10
//          Now obj2.MyProperty is 10
```

```
public class MyNonStaticClass
{
    public static int MyProperty { get; set; }

    public double DoSomething()
    {
        // Some Code
        return MyProperty;
    }
}
```

Static constructors

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

```
public class SimpleClass
{
    // Static variable that must be initialized at run time.
    public static readonly long StaticTicks;

    // SNon static variable
    public readonly long NonStaticTicks;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        StaticTicks = DateTime.Now.Ticks;
    }

    public SimpleClass()
    {
        // StaticTicks = DateTime.Now.Ticks;  // Error
        NonStaticTicks = DateTime.Now.Ticks;
    }
}
```

```
static void Main()
{
    var obj1 = new SimpleClass();
    Console.WriteLine("StaticTicks: {0}, NonStaticTicks: {1}",
        SimpleClass.StaticTicks, obj1.NonStaticTicks);

    Thread.Sleep(1000);

    var obj2 = new SimpleClass();
    Console.WriteLine("StaticTicks: {0}, NonStaticTicks: {1}",
        SimpleClass.StaticTicks, obj2.NonStaticTicks);

    Console.ReadKey();
}
// Output:
//      StaticTicks: 636447935867160327, NonStaticTicks: 636447935867430337
//      StaticTicks: 636447935867160327, NonStaticTicks: 636447935877460897
```

System.Object

Supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

Useful methods:

- **Equals(Object)** - Supports comparisons between objects.
- **Finalize()** - Performs cleanup operations before an object is automatically reclaimed.
- **GetHashCode()** - Generates a number corresponding to the value of the object to support the use of a hash table.
- **ToString()** - Manufactures a human-readable text string that describes an instance of the class.
- **GetType()** - Gets the Type of the current instance (cannot be overloaded).
- **MemberwiseClone()** - Creates a shallow copy of the current Object.

Inheritance
Polymorphism

Inheritance

```
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}
```

```
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```

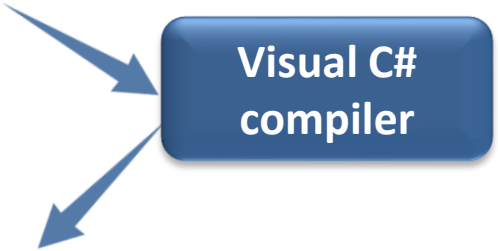
Inheritance

```
class Employee
{
    protected string empName;
    public Employee(string name)
    {
        this.empName = name;
    }
    ...
}
```

```
class Manager : Employee
{
    protected string empGrade;
    public Manager(string name, string grade) : base(name)
    {
        this.empGrade = grade;
    }
    ...
}
```

Inheritance

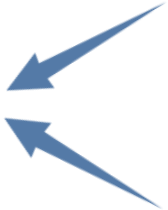
```
class Manager : Employee
{
    public Manager(string name, string grade)
    {   ...    }
    ...
}
```




```
class Manager : Employee
{
    public Manager(string name, string grade) : base()
    {
        ...
    }
    ...
}
```

Inheritance

```
class Employee
{
...
}
```



```
class Manager : Employee
{
...
}
```

```
class ManualWorker : Employee
{
...
}
```

```
 ...
// Manager constructor expects a name and a grade
Manager myManager = new Manager ("Fred", "VP");
ManualWorker myWorker = myManager;
```

```
Manager myManager = new Manager ("Fred", "VP");
Employee myEmployee = myManager;
// legal, Employee is the base class of Manager
```

Inheritance

```
Manager myManager = new Manager ("Fred", "VP");
Employee myEmployee = myManager; // myEmployee refers to a Manager
...
Manager myManagerAgain = myEmployee as Manager;
// OK - myEmployee is a Manager
...
ManualWorker myWorker = new ManualWorker("Bert");
myEmployee = myWorker; // myEmployee now refers to a ManualWorker
...
myManagerAgain = myEmployee as Manager;
// returns null - myEmployee is a ManualWorker
```

Abstract classes

Abstract classes

```
interface ISalaried
{
    void PaySalary();
}
```

```
class ManualWorker : Employee, ISalaried
{   ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Code as ManualWorker for paying salary.
    }
}
```

```
class Manager : Employee, ISalaried
{   ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Code as Manager for paying salary.
    }
}
```

Abstract classes

```
class SalariedEmployee : Employee, ISalaried
{
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}
```

```
class ManualWorker : SalariedEmployee , ISalaried
{
    ...
}
```

```
class Manager : SalariedEmployee , ISalaried
{
    ...
}
```

Abstract classes

```
abstract class SalariedEmployee : Employee, ISalaried
{
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}
```

```
SalariedEmployee myEmployee = new SalariedEmployee();
```

Abstract method

```
abstract class SalariedEmployee : Employee, ISalaried
{
    abstract void PayBonus();
    ...
}
```

Upcasting and downcasting

Type casting

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost.
- **Explicit conversions (casts)**: Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons.
- **User-defined conversions**: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.
- **Conversions with helper classes**: To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the Parse methods of the built-in numeric types, such as Int32.Parse.

```
int i = 5;
i = "Hello"; // Error: "Cannot implicitly convert type 'string' to 'int'"
```

Type Casting

Implicit Conversion

```
0 references
static void Main()
{
    // Implicit conversion. num long can
    // hold any value an int can hold, and more!
    int num = 2147483647;
    long bigNum = num;

    Derived d = new Derived();
    Base b = d; // Always OK.
}
```

Explicit Conversion

```
0 references
static void Main()
{
    double x = 1234.7;
    int a;
    // Cast double to int.
    a = (int)x;
    System.Console.WriteLine(a);

    Base baseClass = new Base();
    // Explicit conversion is required to cast back
    // to derived type. Note: This will compile but will
    // throw an exception at run time if the right-side
    // object is not in fact a Derived.
    Derived deliveredClass = (Derived)baseClass;
}
// Output: 1234
```

Conversion Operators overloading

C# enables programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types. Conversions are defined like operators and are named for the type to which they

```
0 references
static void Main()
{
    SampleClass sampleClass;

    int i = 3;
    sampleClass = i;
    Console.WriteLine("SampleValue is: {0}", sampleClass.SampleValue);

    double d = 5.7;
    sampleClass = (SampleClass)d;
    Console.WriteLine("SampleValue is: {0}", sampleClass.SampleValue);

    Console.ReadKey();
}
// Output:
//      SampleValue is: 3
//      SampleValue is: 5
```

```
class SampleClass
{
    4 references
    public int SampleValue { get; set; }

    public static implicit operator SampleClass(int i)
    {
        var temp = new SampleClass();
        // code to convert from int to SampleClass...
        temp.SampleValue = i;
        return temp;
    }

    public static explicit operator SampleClass(double d)
    {
        var temp = new SampleClass();
        // code to convert from double to SampleClass...
        temp.SampleValue = (int)d;
        return temp;
    }
}
```

Operators overloading

```
public class ComplexNumber
{
    private int real;
    private int imaginary;

    public ComplexNumber(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override bool Equals(object obj)
    {
        var other = obj as ComplexNumber;
        if (other == null) return false;

        return (this.real == other.real) && (this.imaginary == other.imaginary);
    }

    public static bool operator ==(ComplexNumber me, ComplexNumber other) => Equals(me, other);

    public static bool operator !=(ComplexNumber me, ComplexNumber other) => !Equals(me, other);

    public static ComplexNumber operator +(ComplexNumber c1, ComplexNumber c2)
    {
        return new ComplexNumber(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }
}
```

C# allows user-defined types to overload operators by defining static member functions using the **operator** keyword.

```
public class EntryPoint
{
    static void Main()
    {
        var a = new ComplexNumber(3, 5);
        var b = new ComplexNumber(1, 2);
        var c = new ComplexNumber(2, 3);

        if (a == b + c)
            Console.WriteLine("(3, 5) == (1, 2) + (2, 3)");

        if (b != a + c)
            Console.WriteLine("(1, 2) != (3, 5) + (2, 3)");

        Console.ReadKey();
    }
    // Output:
    //      (3, 5) == (1, 2) + (2, 3)
    //      (1, 2) != (3, 5) + (2, 3)
}
```

new and override methods.
Sealed methods and classes.

Virtual

```
class Employee
{     ...
      public virtual string GetTypeName()
     {
          return "This is an Employee";
     }
}
```

Virtual

```
class ManualWorker : Employee
{      ...
     // Does not override GetTypeName
}
class Manager : Employee
{     ...
     public virtual string GetTypeName()
     {
          return "This is a Manager";
     }
}

Employee myEmployee;
Manager myManager = new Manager(...);
ManualWorker myWorker = new ManualWorker(...);
myEmployee = myManager;
Console.WriteLine(myEmployee.GetTypeName());
myEmployee = myWorker;
Console.WriteLine(myEmployee.GetTypeName());
```

Override methods

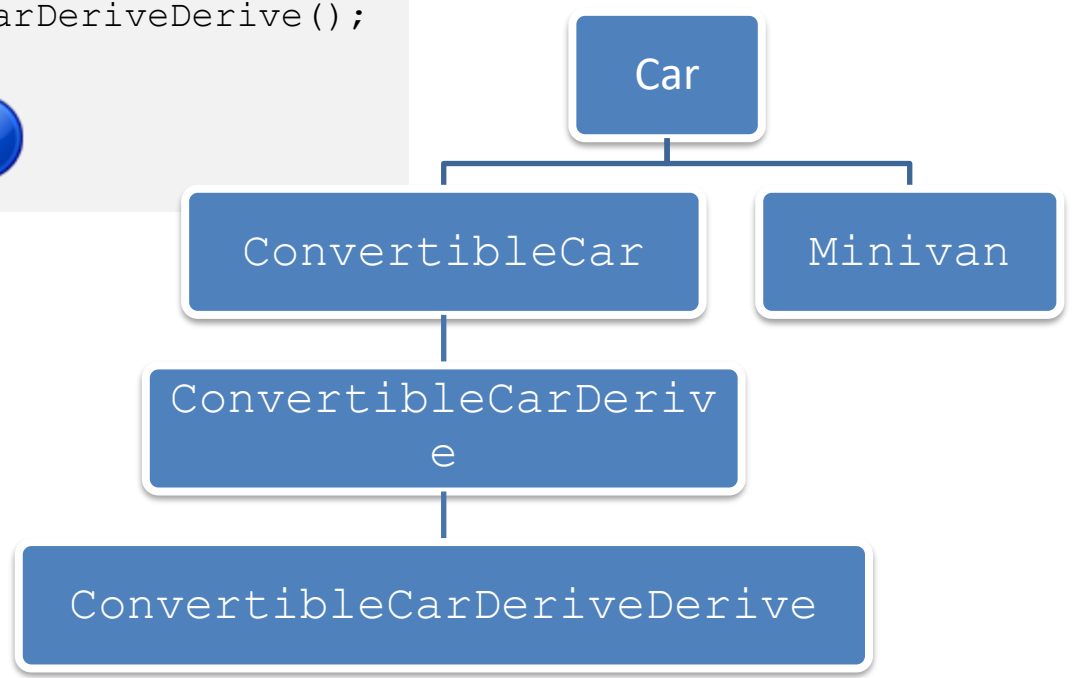
```
class Object
{
    public virtual string ToString()
    {
        // Return the type of the object as a string
    }
}
class Employee
{
    protected string empName;
    ...
    public override string ToString()
    {
        return string.Format("Employee: {0}", empName);
    }
}
```

Hide the method in the base class

```
class Employee
{
    protected void DoWork()
    {
        ...
    }
}

class Manager : Employee
{
    public new void DoWork()
    {
        // Hide the DoWork method in the base class
        ...
    }
    ...
}
```

```
Car car1 = new ConvertibleCar();
Car car2 = new Minivan();
Car car3 = new ConvertibleCarDerive();
Car car4 = new ConvertibleCarDeriveDerive();
car1.ShowDetails();
car2.ShowDetails();
car3.ShowDetails();
car4.ShowDetails();
```



base

```
class Employee
{
    protected virtual void DoWork()
    {   ...   }
}


class Manager : Employee
{
    protected override void DoWork()
    {
        // Do processing specific to Managers
        ...
        // Call the DoWork method in the base class
        base.DoWork();
    }
    ...
}
```

Sealed methods and classes


```
sealed class Manager : Employee
{
...
}
```

```
class Manager : Employee
{    ...
     protected sealed override void DoWork()
     {    ...     }
}
```

.NET Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

Oleksii_Leunenko@epam.com

With the note [.NET Online UA Training Course Feedback]

Thank you.

Q&A

UA .NET Online LAB