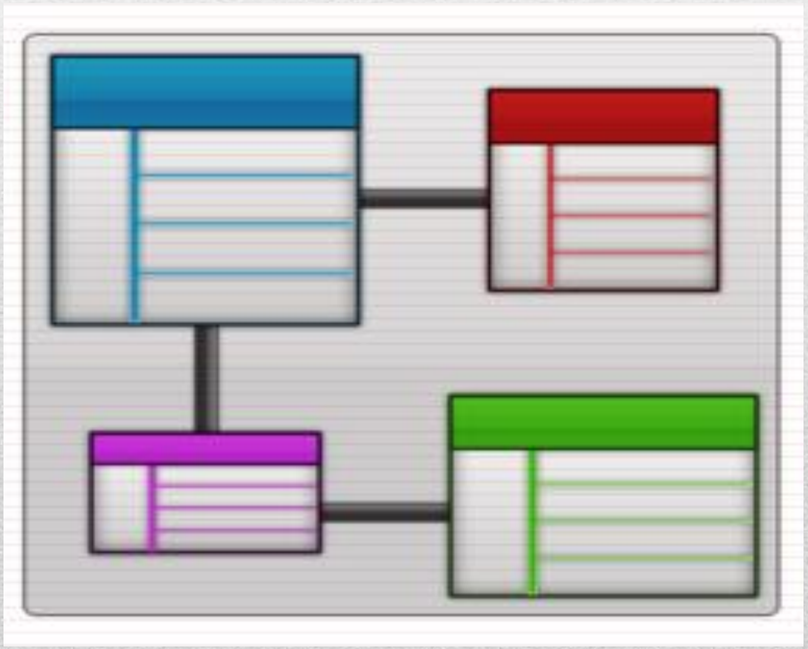
WEEK#1 OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

MARKS DISTRIBUTION TENTATIVE

Mid	20%	
Final	40%	
Project	15%	
Quiz	10%	(5)
Assignments	5%	(2)
Lab	10%	

TEXT BOOK

C Sharp How To Program (Prentice Hall)
Sixth Edition
DEITEL & DEITEL

C# PROGRAMMING LANGUAGE

- **In 2000, Microsoft announced the C# programming language.**
- **C# has roots in the C, C++ and Java programming languages.**
- **It has similar capabilities to Java and is appropriate for the most demanding app-development tasks, especially for building today’s desktop apps, large-scale enterprise apps, and web-based, mobile and cloud-based apps.**
- **It’s supported by .NET’s huge class library that makes development of modern Graphical User Interface applications for personal computers very easy.**

WHAT IS C#?

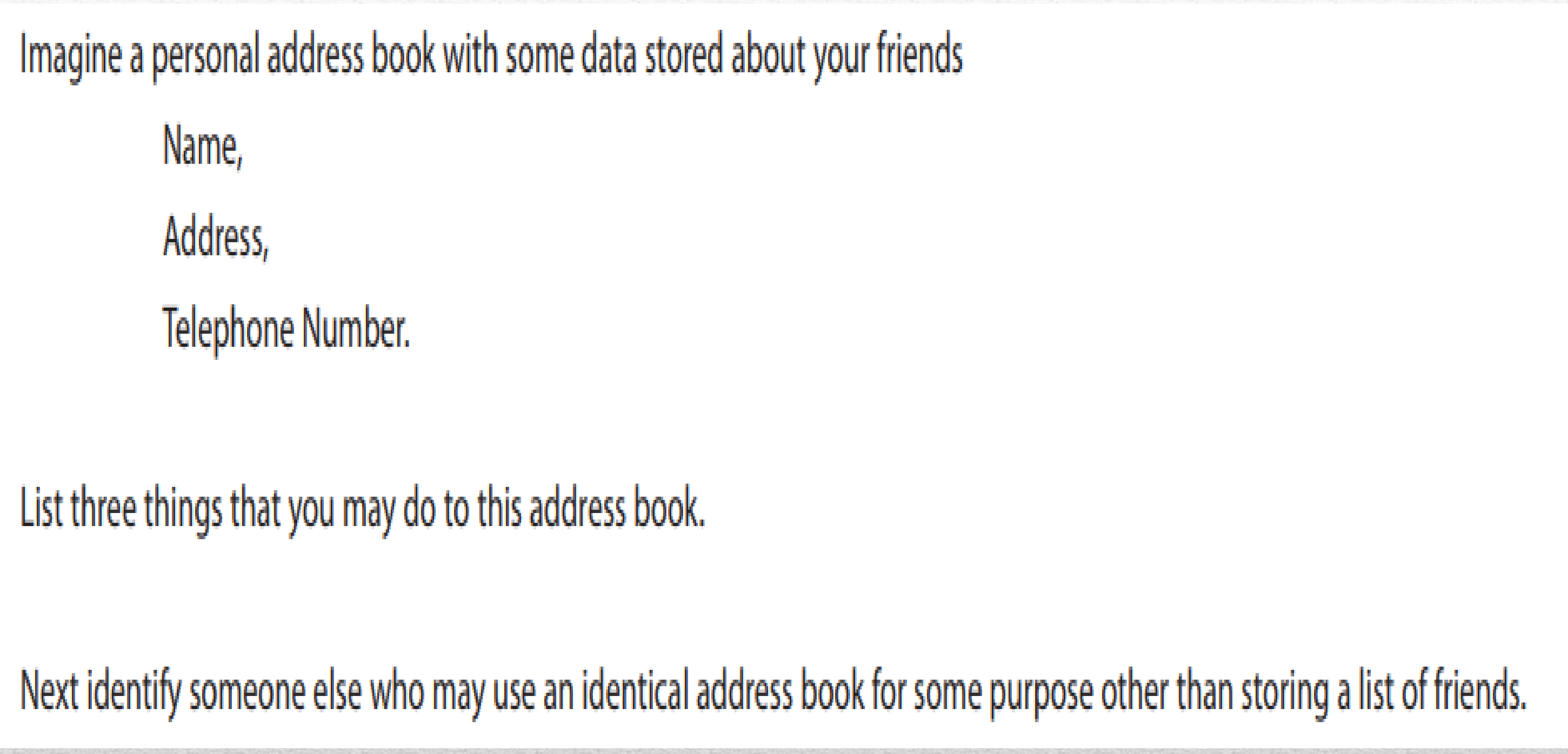
- C# is a general purpose object oriented programming language developed by Microsoft for program development in the .NET Framework.

- It’s a C-like language and many features resemble those of C++ and Java. For instance, like Java, it too has automatic garbage collection.

Object Oriented Programming

- Object Oriented Programming (OOP) is a programming concept used in several modern programming languages, like C++, Java and Python.
- Objects are the most important part of a program. Manipulating these objects to get results is the goal of Object Oriented Programming
- In OOP, the data is grouped together with the methods that operate upon it, which makes it easy to maintain the integrity of the data and provide a structure to the program.

Object Oriented Programming



Imagine a personal address book with some data stored about your friends

Name,

Address,

Telephone Number.

List three things that you may do to this address book.

Next identify someone else who may use an identical address book for some purpose other than storing a list of friends.

Object Oriented Programming

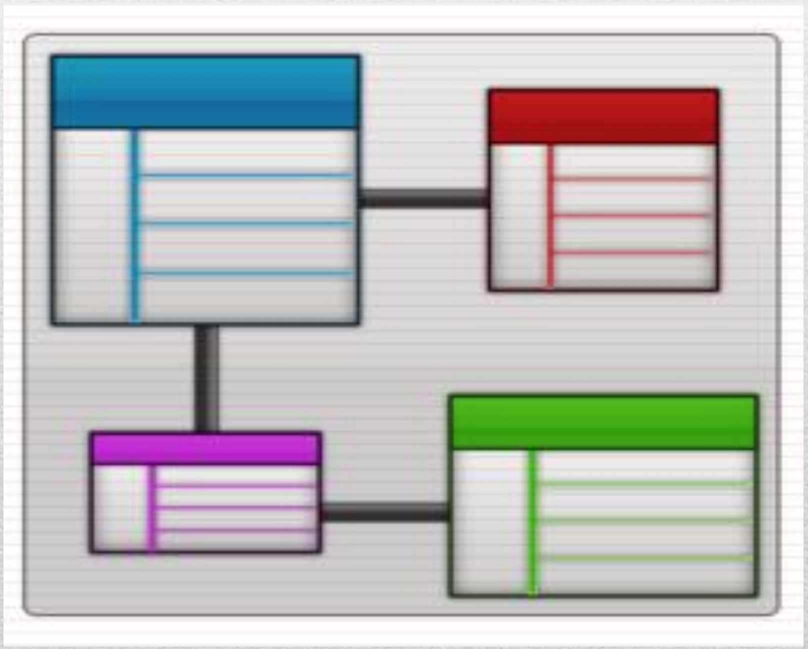
With an address book we would want to be able to perform the following actions :- find out details of a friend i.e. their telephone number, add an address to the address book and, of course, delete an address.

We can create a simple software component to store the data in the address book (i.e. list of names etc) and the operations, things we can do with the address book (i.e add address, find telephone number etc).

By creating a simple software component to store and manage addresses of friends we can reuse this in another software system i.e. it could be used by a business manager to store and find details of customers. It could also become part of a library system to be used by a librarian to store and retrieve details of the users of the library.

Thus in object oriented programming we can create re-usable software components (in this case an address book).

WEEK#2
OBJECT ORIENTED PROGRAMMING

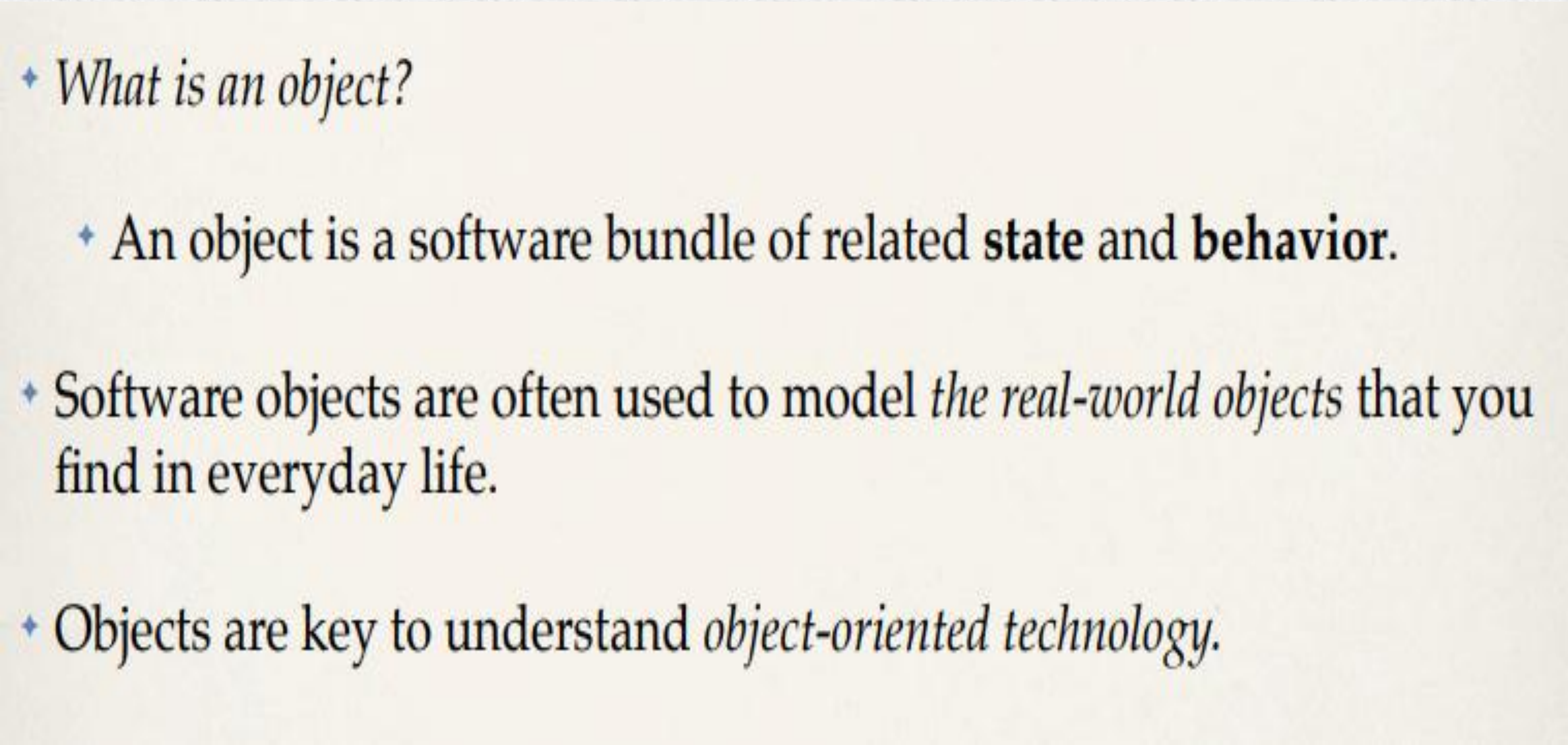


BY

SOLAT JABEEN SHEIKH

CONCEPTS OF OOP

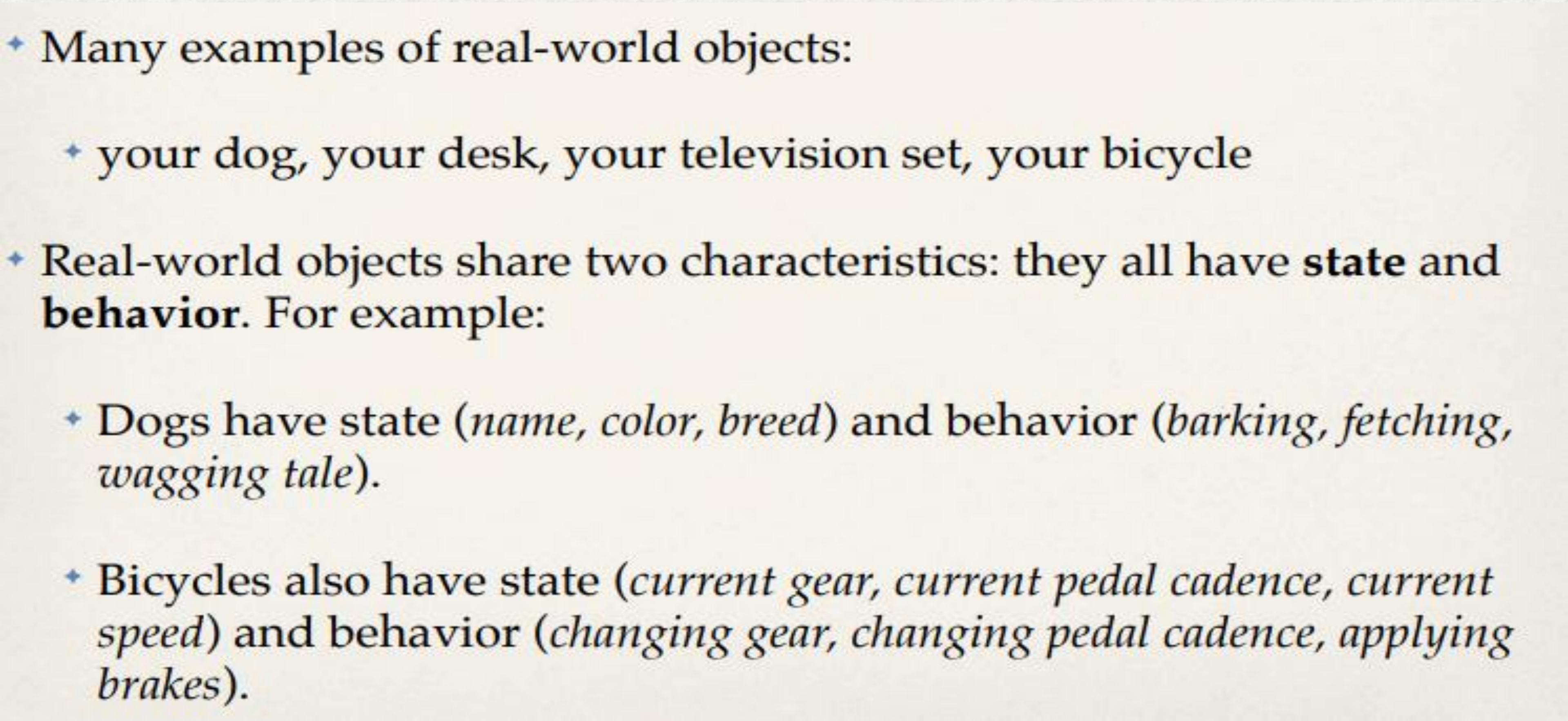
What Is OBJECT?



- *What is an object?*
  - An object is a software bundle of related **state** and **behavior**.
- Software objects are often used to model *the real-world objects* that you find in everyday life.
- Objects are key to understand *object-oriented technology.*

CONCEPTS OF OOP

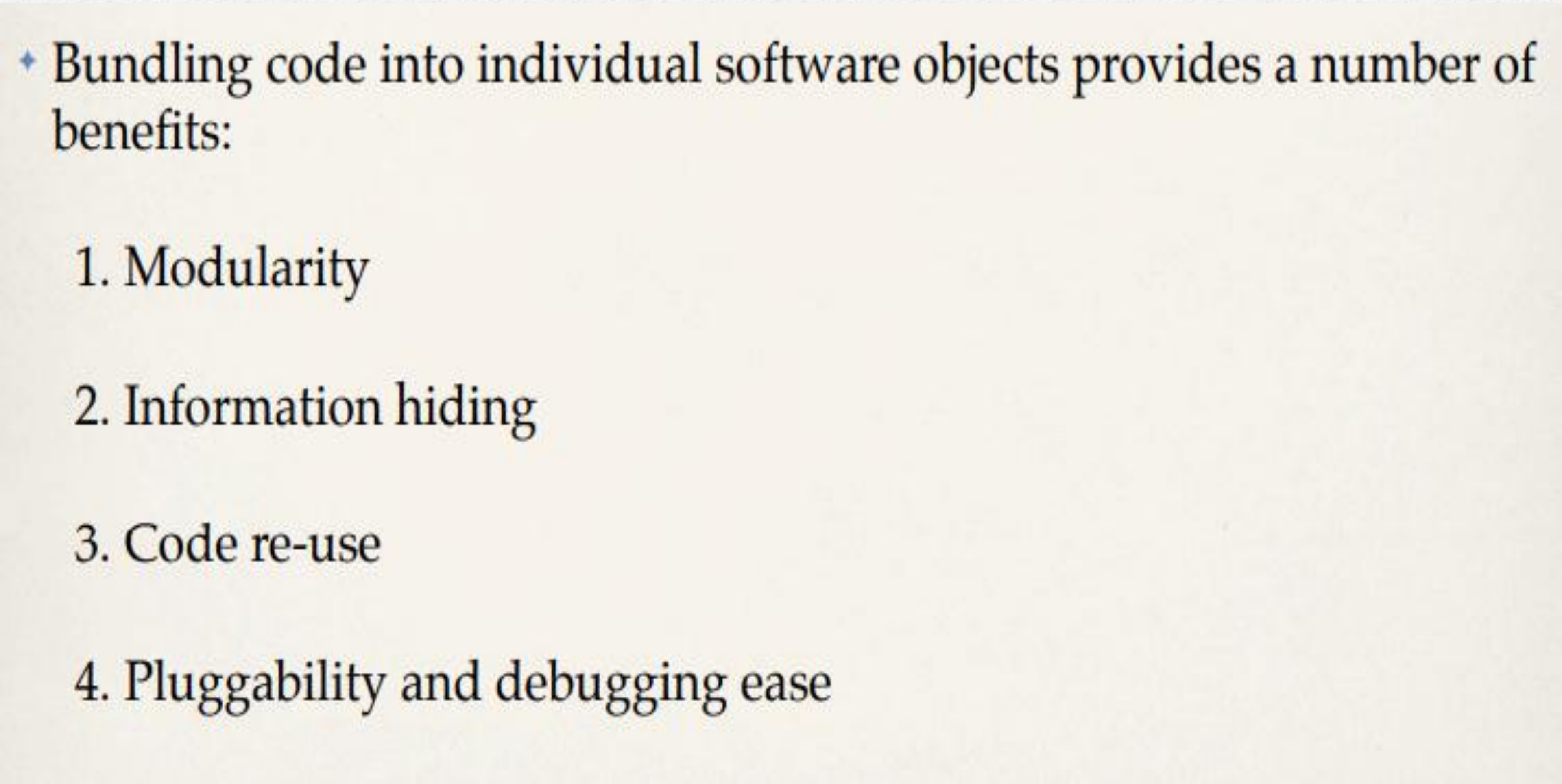
What Is OBJECT?



- Many examples of real-world objects:
  - your dog, your desk, your television set, your bicycle
- Real-world objects share two characteristics: they all have **state** and **behavior**. For example:
  - Dogs have state (*name, color, breed*) and behavior (*barking, fetching, wagging tale*).
  - Bicycles also have state (*current gear, current pedal cadence, current speed*) and behavior (*changing gear, changing pedal cadence, applying brakes*).

CONCEPTS OF OOP

What Is OBJECT?



- Bundling code into individual software objects provides a number of benefits:

  1. Modularity

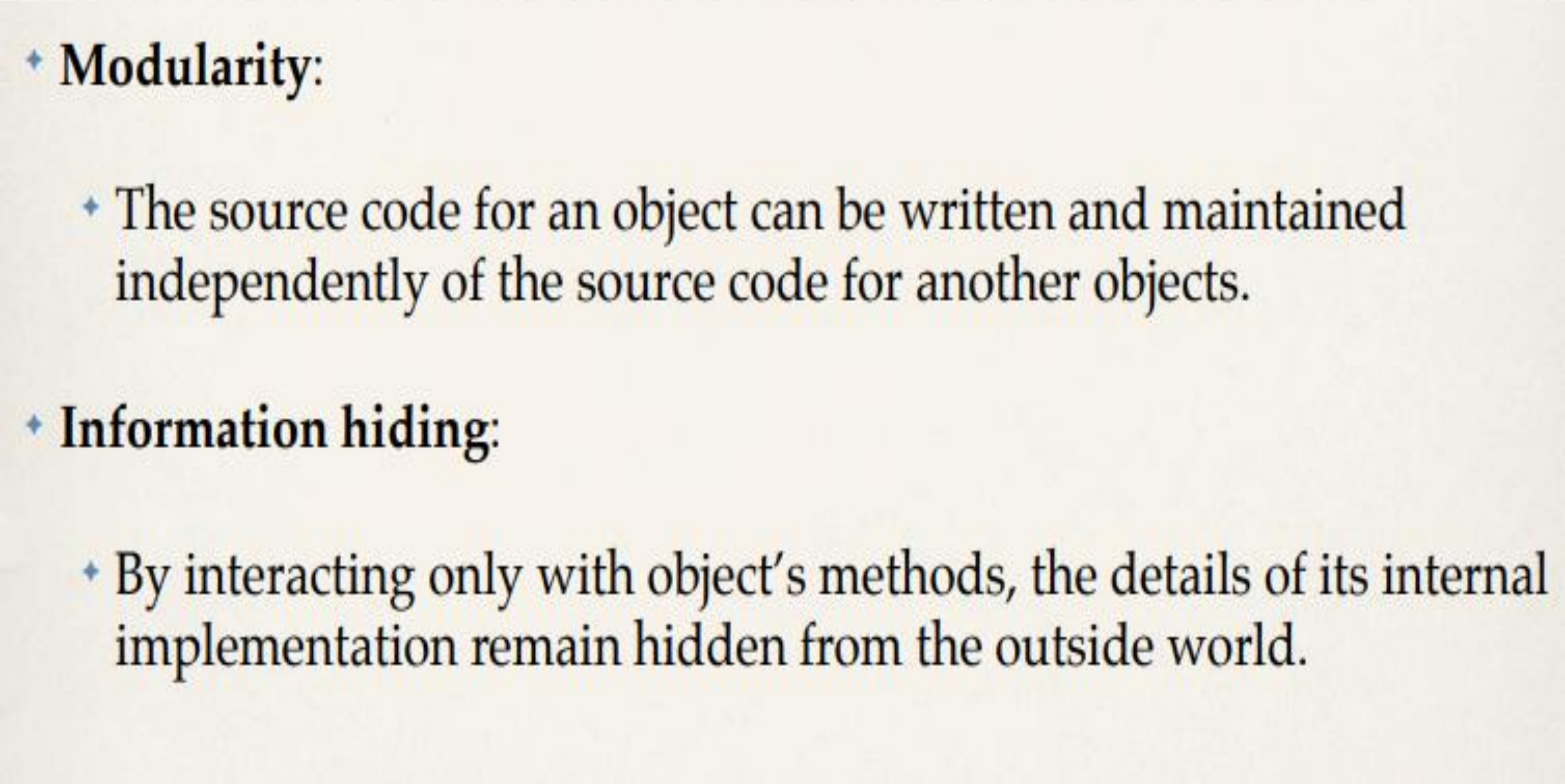
  2. Information hiding

  3. Code re-use

  4. Pluggability and debugging ease

CONCEPTS OF OOP

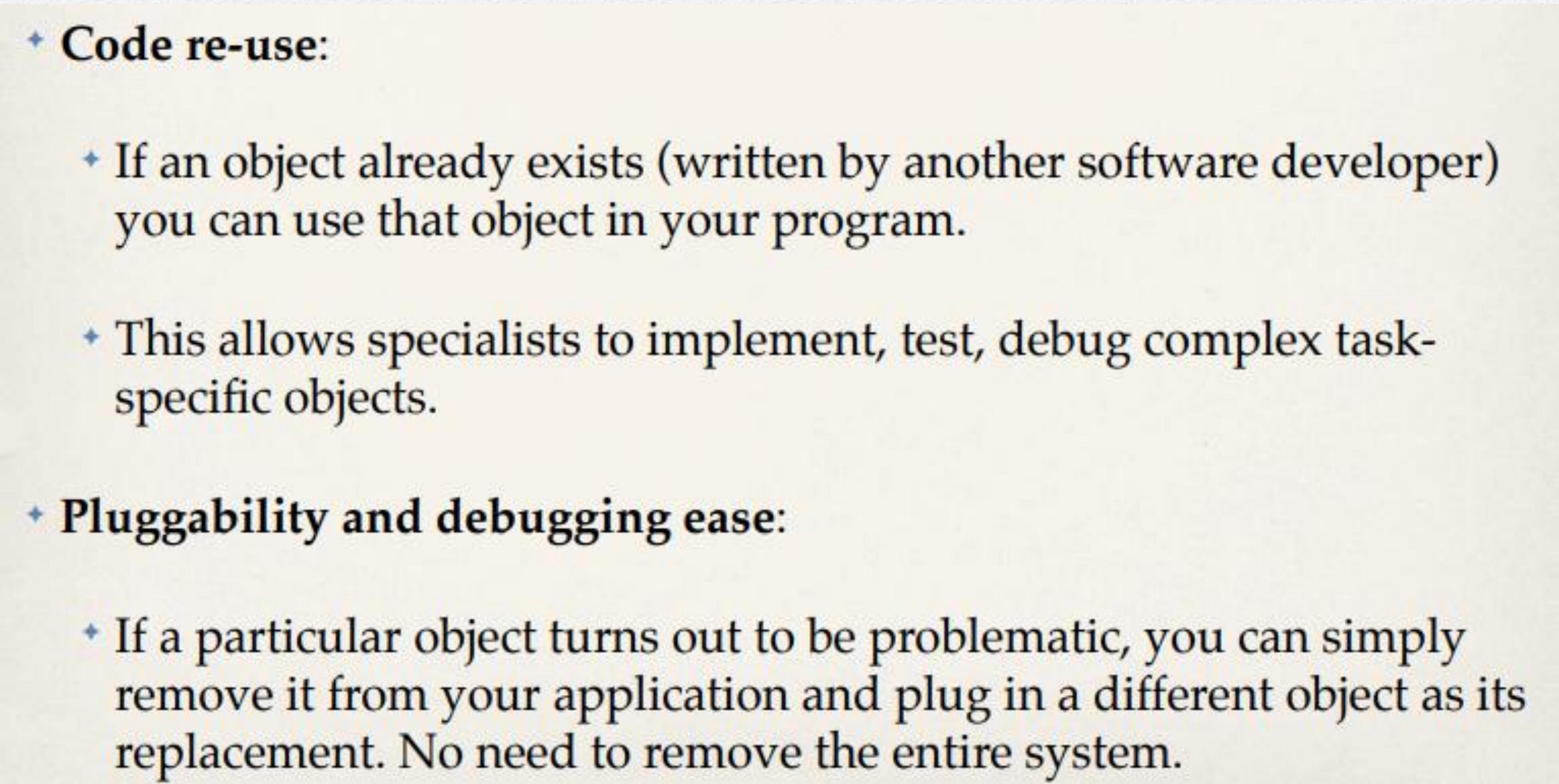
What Is OBJECT?



- **Modularity**:
  - The source code for an object can be written and maintained independently of the source code for another objects.
- **Information hiding**:
  - By interacting only with object's methods, the details of its internal implementation remain hidden from the outside world.

CONCEPTS OF OOP

What Is OBJECT?



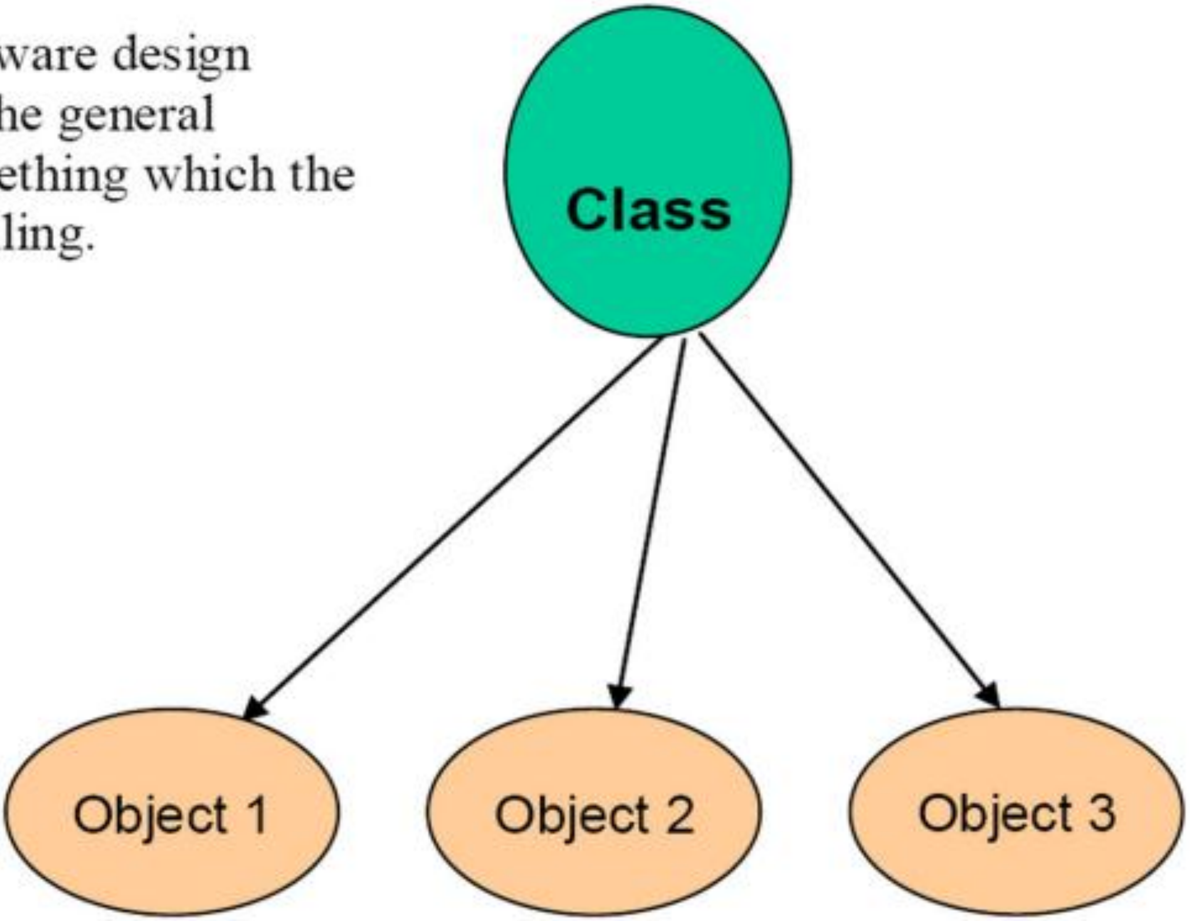
- **Code re-use**:
  - If an object already exists (written by another software developer) you can use that object in your program.
  - This allows specialists to implement, test, debug complex task-specific objects.
- **Pluggability and debugging ease**:
  - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. No need to remove the entire system.

CONCEPTS OF OOP

What Is CLASS?

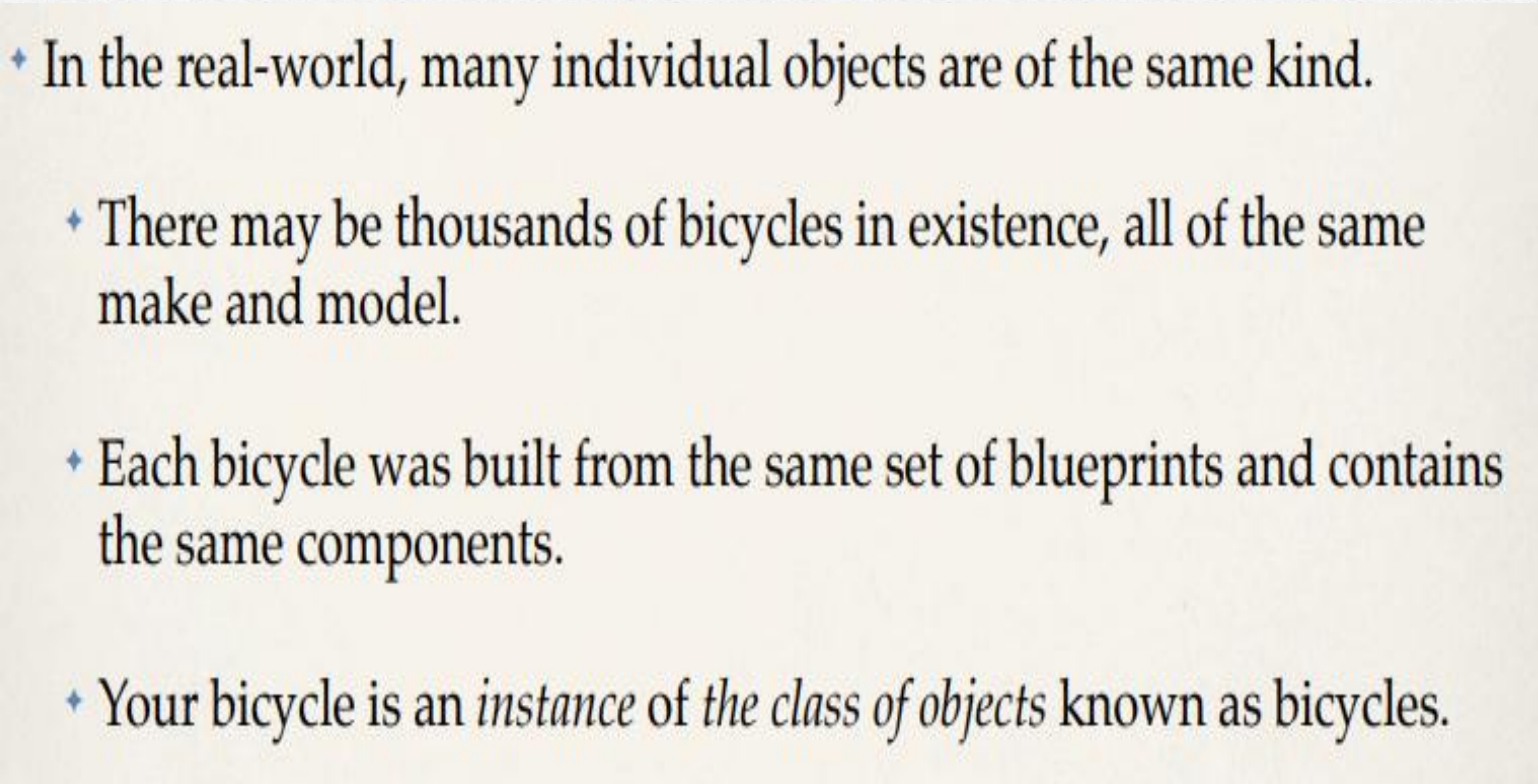
A 'class' is a software design which describes the general properties of something which the software is modelling.

Individual 'objects' are created from the class design for each actual thing



CONCEPTS OF OOP

What Is CLASS?



- In the real-world, many individual objects are of the same kind.
  - There may be thousands of bicycles in existence, all of the same make and model.
  - Each bicycle was built from the same set of blueprints and contains the same components.
  - Your bicycle is an *instance* of *the class of objects* known as bicycles.

CONCEPTS OF OOP

What Is CLASS?

- class is a blueprint
- A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.
- houses the set of methods that perform the class's tasks.

CONCEPTS OF OOP

- For example, a class that represents a bank account might contain one method to deposit money to an account and another to withdraw money from an account.

- A class is similar in concept to a car’s engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

CONCEPTS OF OOP

- For example, a class that represents a bank account might contain one method to deposit money to an account and another to withdraw money from an account.

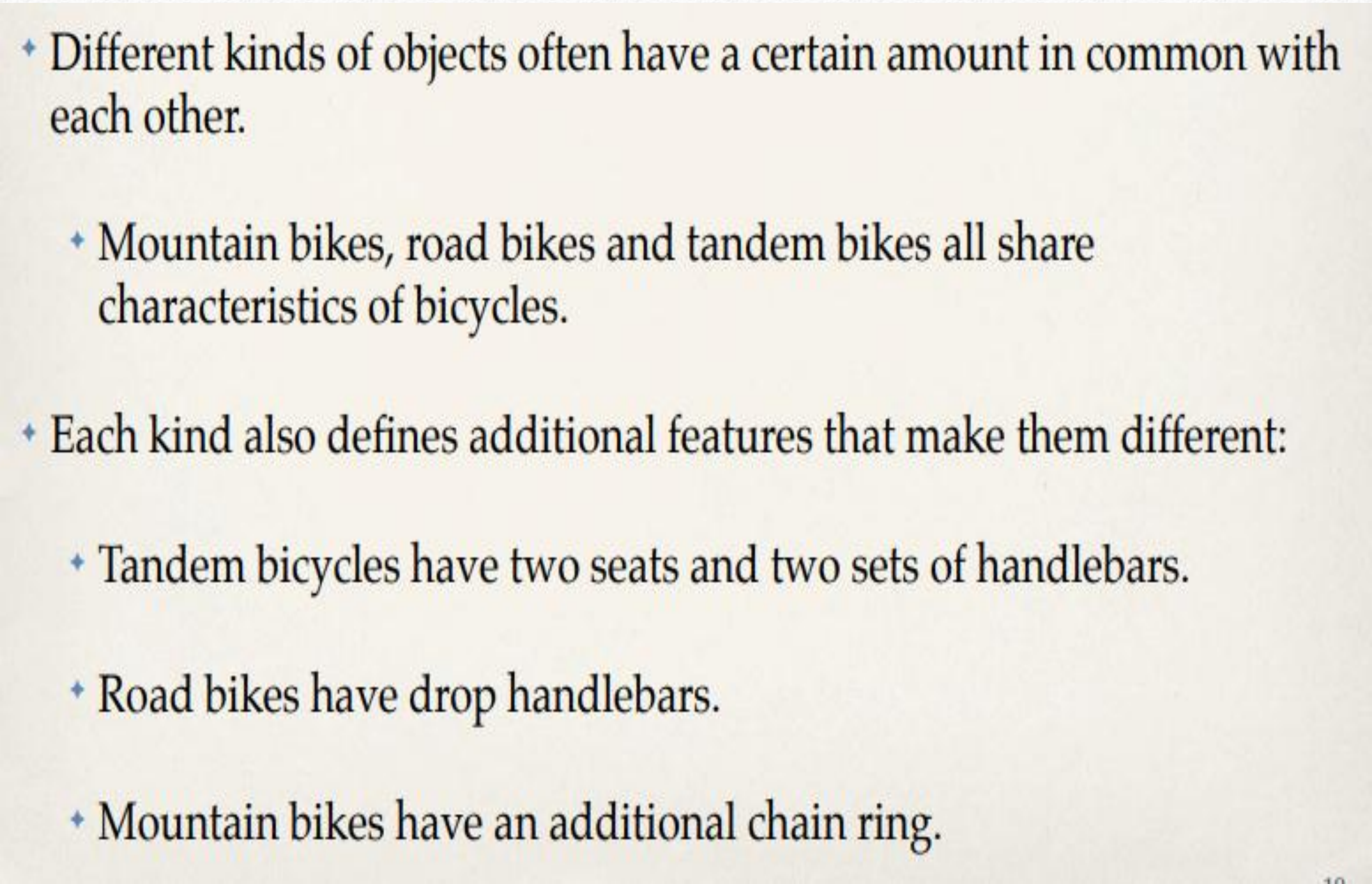
HOW TO DEFINE A CLASS?

```
public class Flower {
String name;
String color;
void pollination() {}
}
```

PILLARS

- ❑ INHERITANCE
- ❑ POLYMORPHISM
- ❑ ABSTRACTION
- ❑ ENCAPSULATION

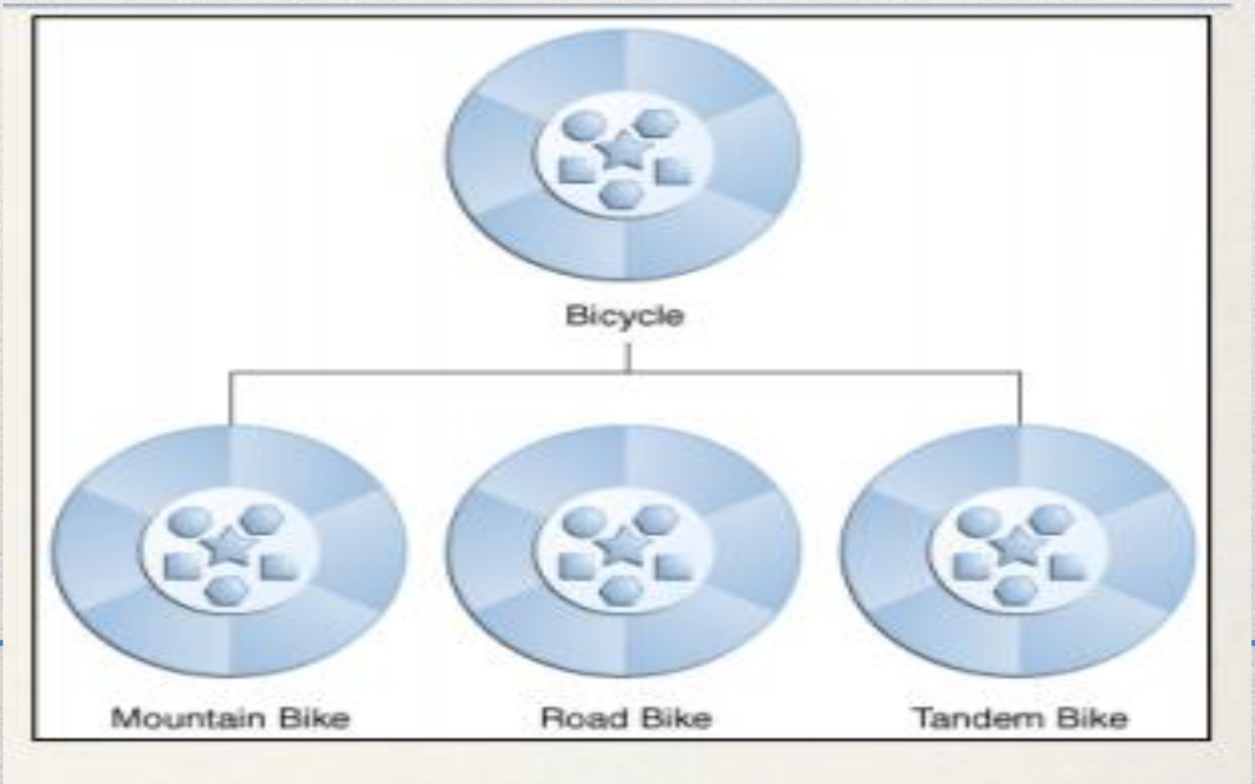
1. INHERITANCE



- Different kinds of objects often have a certain amount in common with each other.
  - Mountain bikes, road bikes and tandem bikes all share characteristics of bicycles.
- Each kind also defines additional features that make them different:
  - Tandem bicycles have two seats and two sets of handlebars.
  - Road bikes have drop handlebars.
  - Mountain bikes have an additional chain ring.

1. INHERITANCE

- Object-oriented programming allows class to *inherit* commonly used state and behavior from other classes.
- Bicycle now becomes the *superclass* of MountainBike, RoadBike and TandemBike.



1. INHERITANCE

- reuse existing code
- reduces the time
- derived class
- Base class
- Example: Windows operating system.

2. POLYMORPHISM

- different forms at different times (poly + morphos)

- Polymorphism is the ability of two different objects to respond to the same request message in their own unique way.

- For example, I could train my dog to respond to the command bark and my bird to respond to the command chirp. On the other hand, I could train them to both respond to the command speak. Through polymorphism I know that the dog will respond with a bark and the bird will respond with a chirp.

2. POLYMORPHISM

- In OOP you could send a print message to a printer object that would print the text on a printer, and you could send the same message to a screen object that would print the text to a window on your computer screen.

- use of words in the English language. Words have many different meanings, but through the context of the sentence you can deduce which meaning is intended.

OVERLOADING

- In OOP you implement this type of polymorphism through a process called overloading.
- You can implement different methods of an object that have the same name.
- The object can then tell which method to implement depending on the context (in other words, the number and type of arguments passed) of the message.

OVERLOADING

```
Class Addition
{
   int add(int a, int b)
   {
     return a+b;
   }
   int add(int a, int b, int c)
   {
     return a+b+c;
   }
}
```

OVERIDING

- occurs when the methods itself are changed.

- EXAMPLE: When you don’t need a President but a Prime Minister

3. ABSTRACTION

Activity 3 Consider your home and imagine you were going to swap your home for a week with a new friend.

Write down three essential things you would tell them about your home and that you would want to know about their home.

You presumably would tell them the address, give them a basic list of rooms and facilities (e.g. number of bedrooms) and tell them how to get in (i.e which key would operate the front door and how to switch off the burglar alarm (if you have one).

You would not tell them irrelevant details (such as the colour of the walls, seats etc) as this would overload them with useless information.

Abstraction allows us to consider the important high level details of your home, e.g. the address, without becoming bogged down in detail.

3. ABSTRACTION

- displaying relevant properties and methods
- reduces the complexity
- EXAMPLE: Phone

```
        public abstract class ExampleAbs
{
        // declare your fields
        abstract void delay();
}
```

4. ENCAPSULATION

- keeping objects with their methods in one place
- protects the integrity of the data
- prevents it from being needlessly altered
- A normal class in Java follows the encapsulation principle by default

4. ENCAPSULATION

```
public class Employeedetails
{
        private String employeeName;
        private String employeeDept;
        private int salary;
        private int hoursperday;

        public void work ()
        {
        }
        public void train ()
        {
        }
}
```

Why OOP?

PROBLEMS WITH PROCEDURAL LANGUAGES:

- Functions have unrestricted access to global data
- Unrelated Functions and data.

Why OOP?

Before Object Oriented Programming programs were viewed as procedures that accepted data and produced an output. There was little emphasis given on the data that went into those programs.

Procedure Oriented Programming	Object Oriented Programming
Program is divided into small parts called functions.	Program is divided into parts called objects.
Importance is not given to data but to functions as well as sequence of actions to be done.	Importance is given to the data rather than procedures or functions because it works as a real world.
Data can move freely from function to function in the system.	Objects can move and communicate with each other through member functions.
To add new data and function is not so easy.	OOP provides an easy way to add new data and function.
Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Examples are C, VB, FORTRAN, Pascal.	Examples are C++, JAVA, VB.NET, C#.NET.

WHAT IS .NET FRAMEWORK?

Language independent platform for creating platform independent software.

It was designed and is maintained by Microsoft Corporation.

WHAT IS .NET FRAMEWORK?

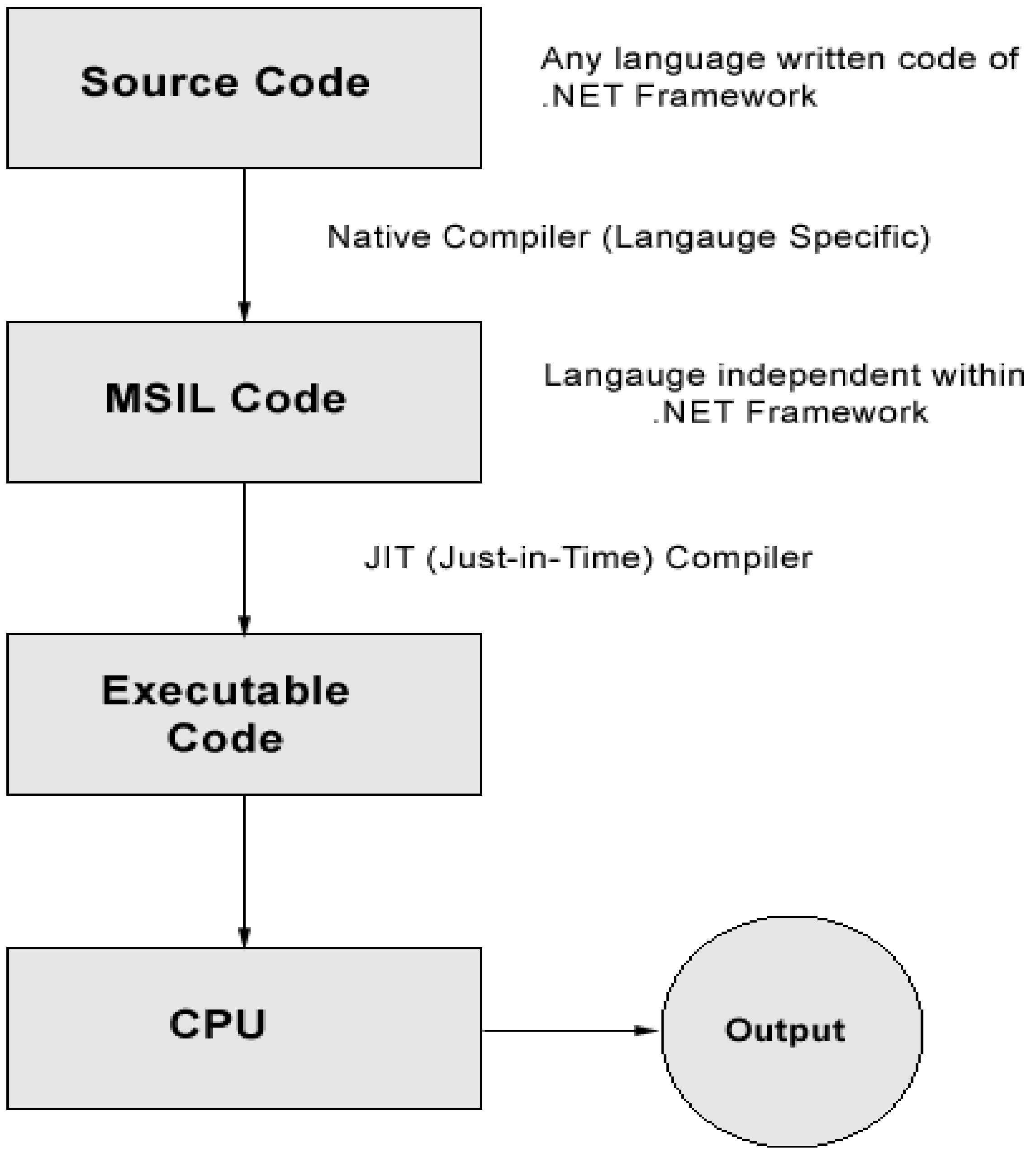
Language independence means code can be executed written in any language.

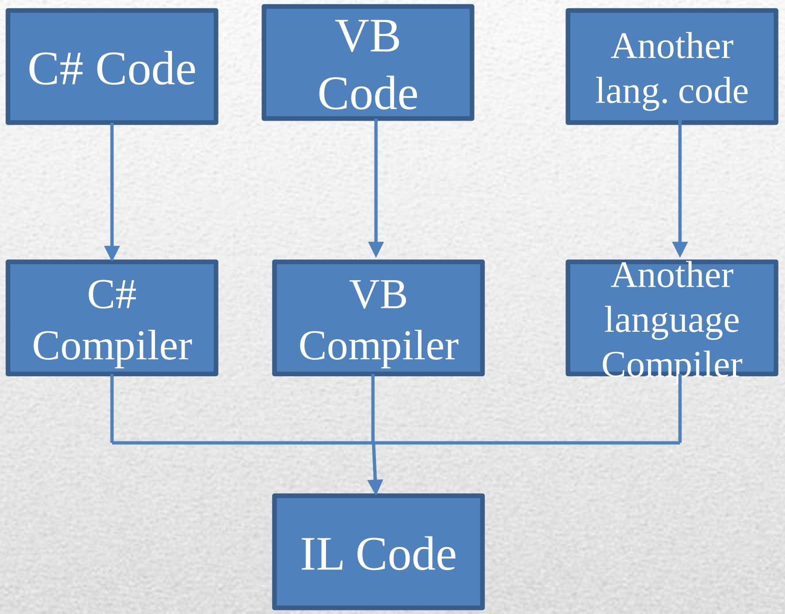
WHAT IS .NET FRAMEWORK?

Platform independence means that an application can be executed on any machine without being concerned with issues like what is the operating system of the machine etc.

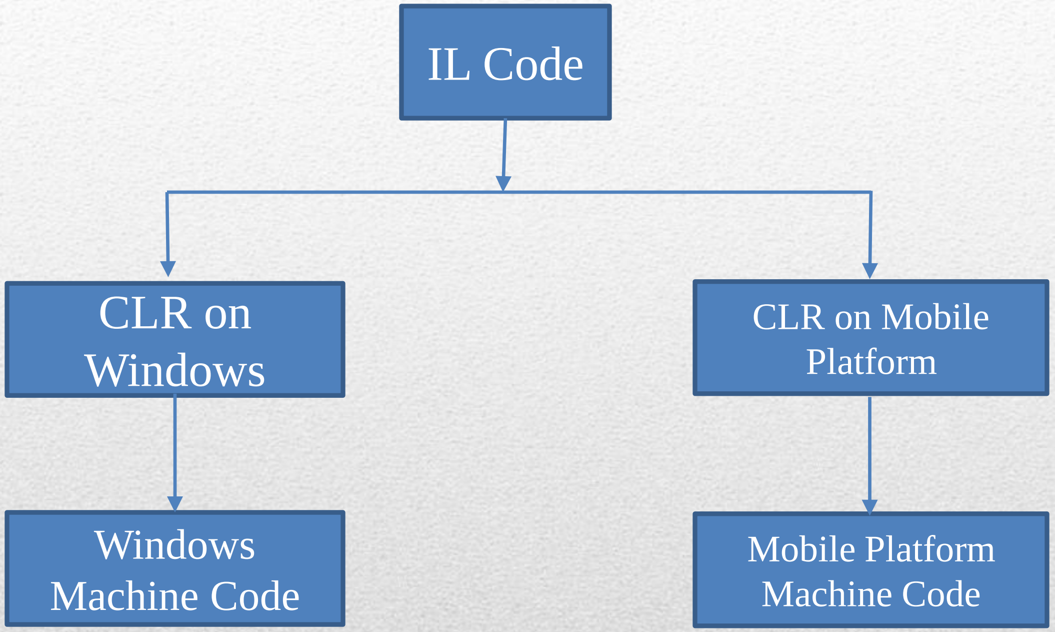
Platform independence refers in particular to the ability of the application to run on any machine having any operating system installed on it.

Managed Code Execution Process

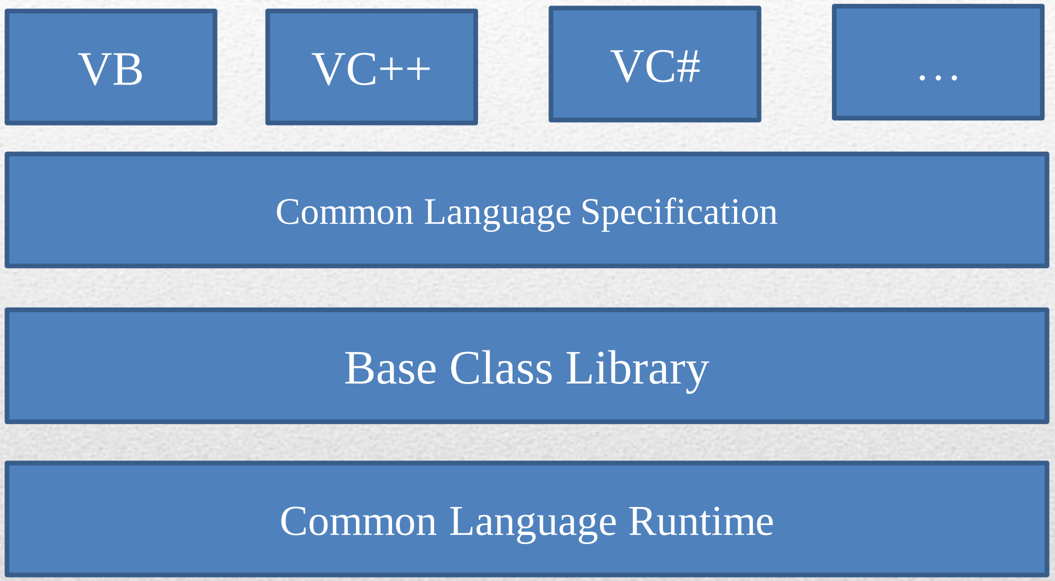

Source Code
Any language written code of .NET Framework
Native Compiler (Langauge Specific)
MSIL Code
Langauge independent within .NET Framework
JIT (Just-in-Time) Compiler
Executable Code
CPU
Output



CREATING .NET SOFTWARE “LANGUAGE INDEPENDENT”



EXECUTING .NET SOFTWARE “PLATFORM INDEPENDENT”



COMPONENTS OF .NET FRAMEWORK

.NET LANGUAGES

Any language that conforms to the Common Language Infrastructure (CLI) specification of the .NET, can run in the .NET run-time. Followings are some .NET languages:

- Visual Basic
- C#
- C++ (CLI version)
- J# (CLI version of Java)
- A# (CLI version of ADA)
- L# (CLI version of LISP)
- IronRuby (CLI version of RUBY)

TYPES OF SOFTWARES THAT CAN BE CREATED WITH .NET FRAMEWORK

Desktop Applications
WPF.Winforms
Word, excel, dropbox desktop UI

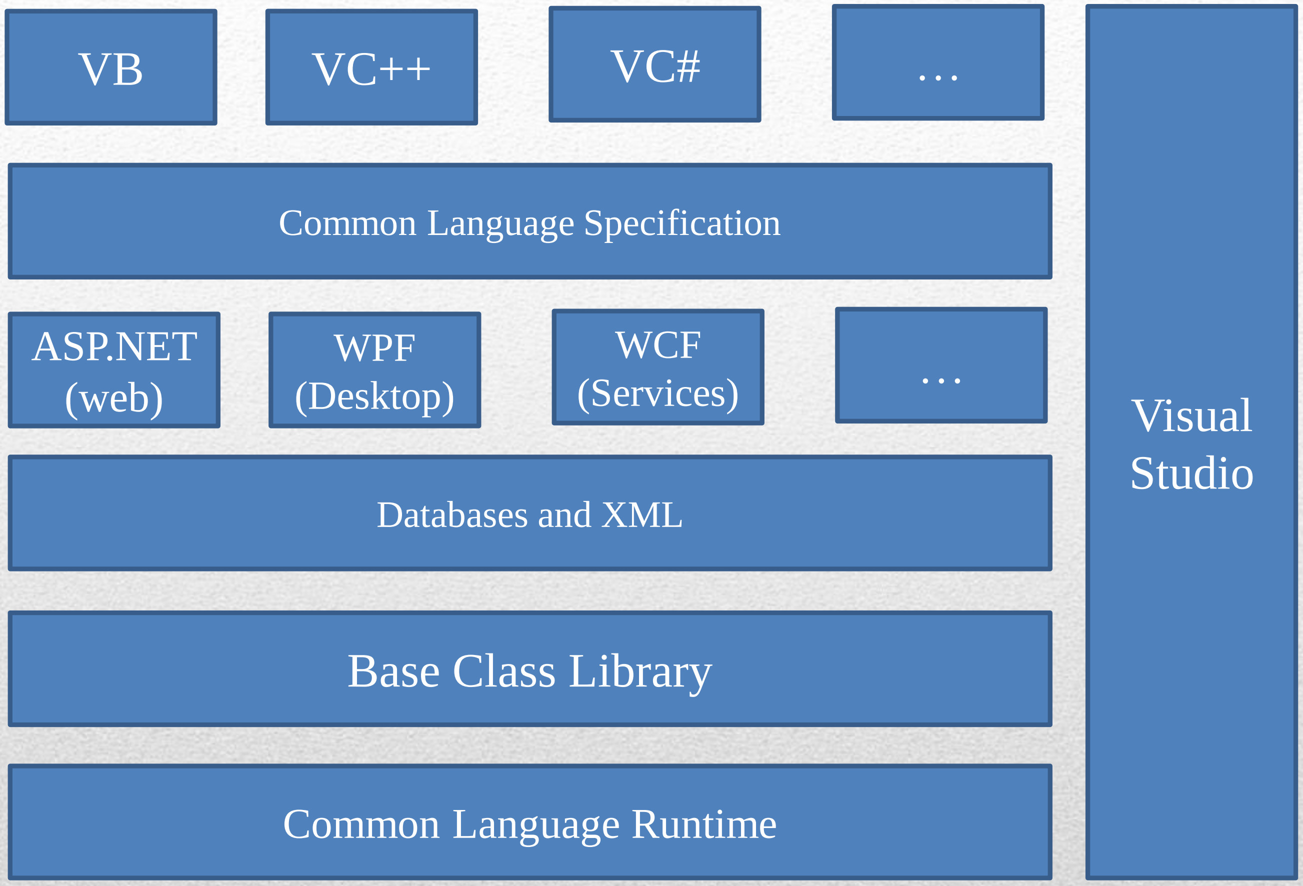
Web Applications
ASP.NET
Facebook, youtube, dropbox web UI

Service Based Backends
WCF Services
Whatsapp backend, skype backend, dropbox backend

.NET IDEs

Microsoft provides a comprehensive Integrated Development Environment (IDE) for the development and testing of software with .NET. Some IDEs are as follows:

- Visual Studio
- Visual Web Developer
- Visual Basic
- Visual C#

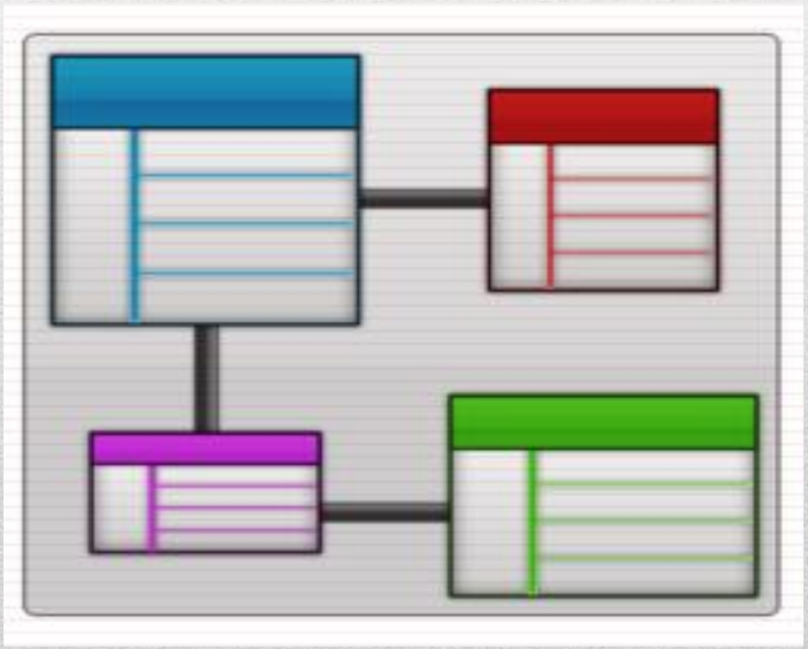


VS and .NET FRAMEWORK

ASSIGNMENT

- List down all the versions of .Net frameworks till date along with all the details.
- List down all the versions of C# programming language till date along with all the details.

WEEK#3 OBJECT ORIENTED PROGRAMMING

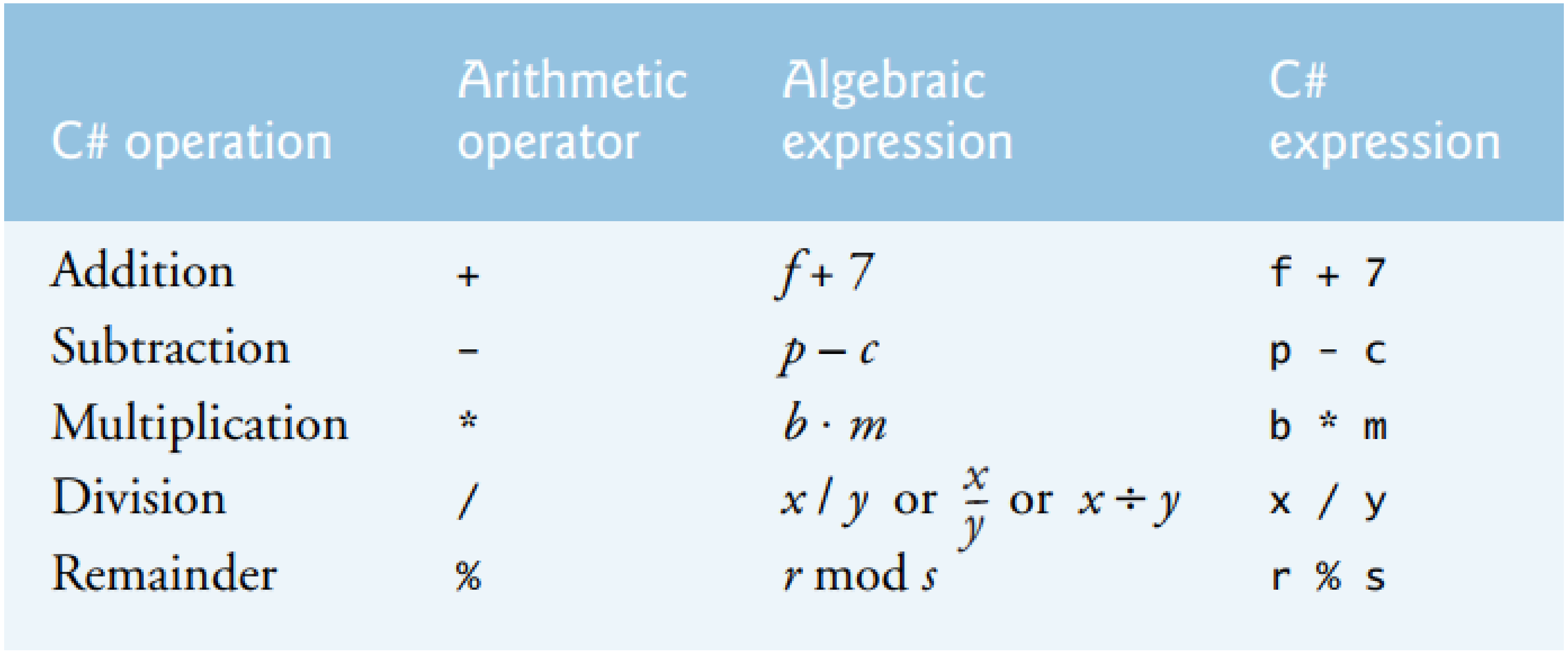


BY

SOLAT JABEEN

C# REVIEW

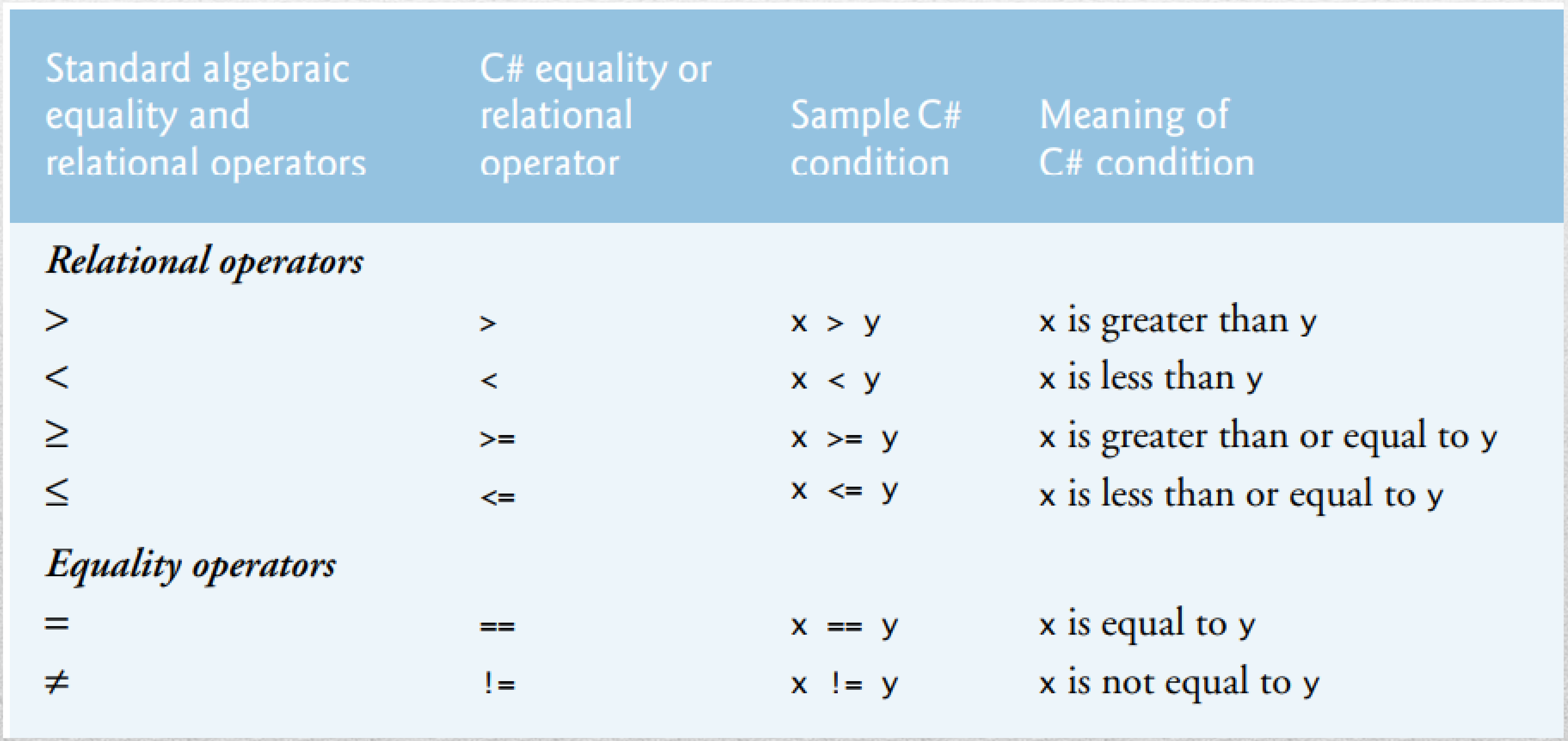
- **ARITHMETIC OPERATORS**



C# operation	Arithmetic operator	Algebraic expression	C# expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	$b \cdot m$	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

C# REVIEW

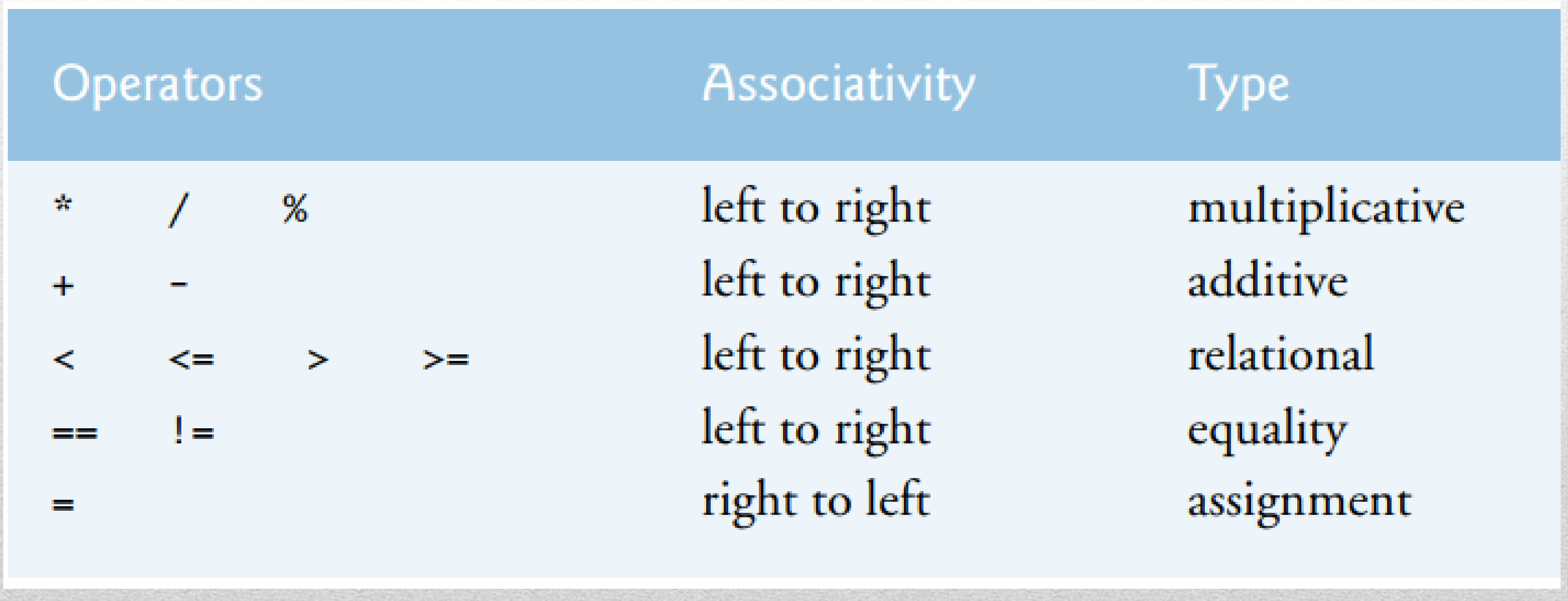
- **RELATIONAL AND EQUALITY OPERATOS**



Standard algebraic equality and relational operators	C# equality or relational operator	Sample C# condition	Meaning of C# condition
Relational operators			
$>$	`>`	`x > y`	x is greater than y
$<$	`<`	`x < y`	x is less than y
$\geq$	`>=`	`x >= y`	x is greater than or equal to y
$\leq$	`<=`	`x <= y`	x is less than or equal to y
Equality operators			
$=$	`==`	`x == y`	x is equal to y
$\neq$	`!=`	`x != y`	x is not equal to y

C# REVIEW

- **PRECEDENCE AND ASSOCIATIVITY**



Operators	Associativity	Type
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`=`	right to left	assignment

C# REVIEW

- Assignment
  - <Variable Name> = <Expression>
  - a = (1+4)*5

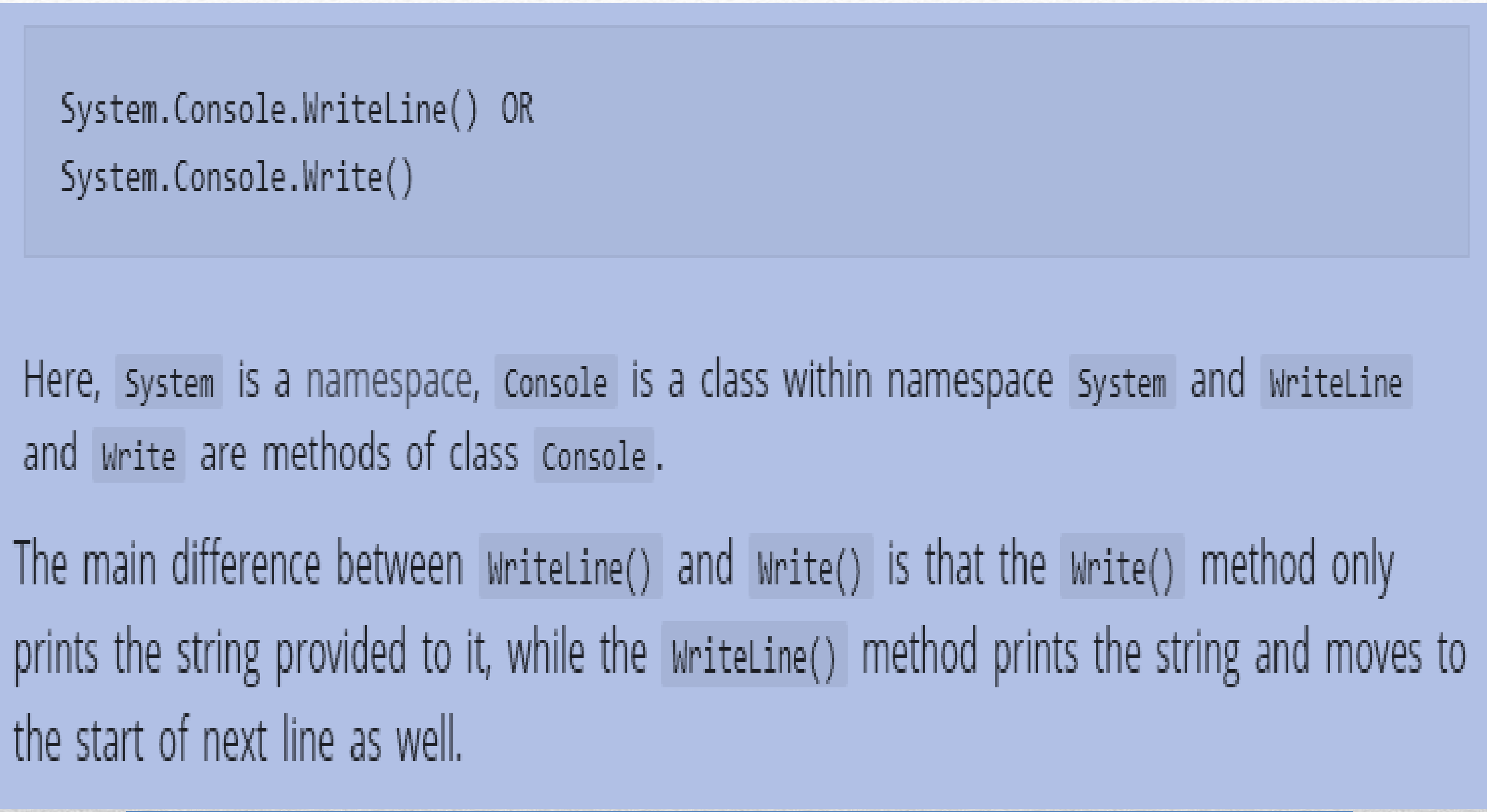
- if Statement
  - if (<condition>) {<statements>}
    - if (a < 2) {a = 5;}

- if (<condition>) {<statements>} else {<statements>}
  - if (a > 2) {a = 5;} else {a == 10;}

C# REVIEW

- Loops
  - For loop
    - for (<initialization>; <termination test>; <make progress>) {<statements>}
      - for (int i = 10; i >0; i--) {Console.WriteLine(i);}
  - While loop
    - while (<condition>) {<statements>}
      - while (i > 0) {Console.WriteLine(i); i--;}
  - Do-while loop
    - do {<statements>} while (<condition>)
      - do {Console.WriteLine(i); i--;} while (i > 0);

C# OUTPUT

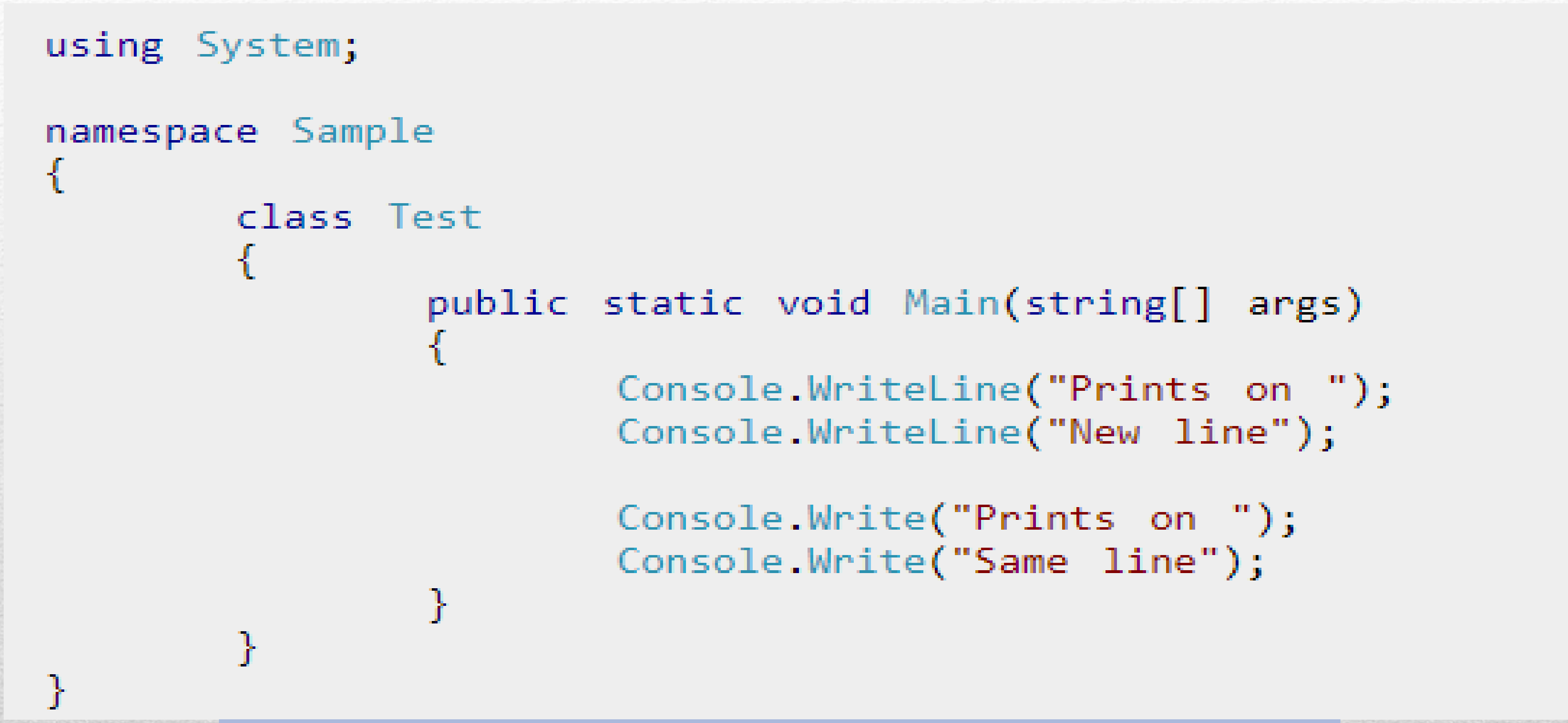


```
System.Console.WriteLine() OR
System.Console.Write()
```

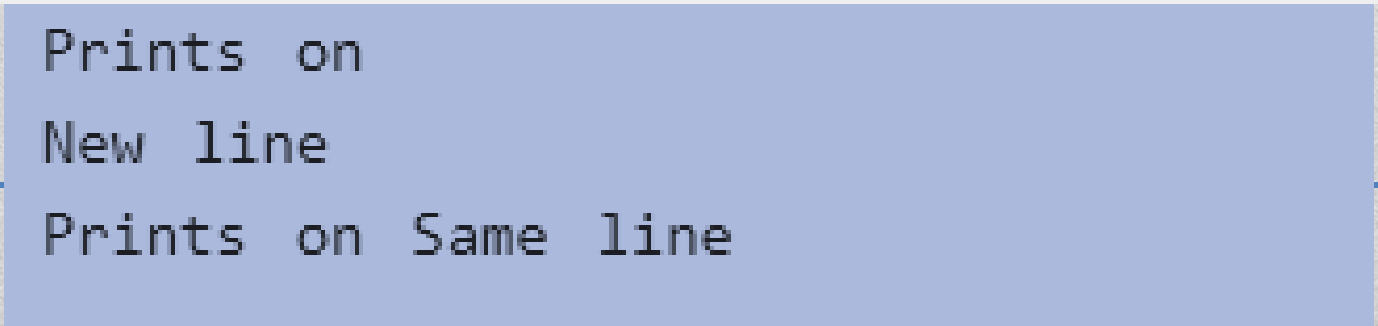
Here, `System` is a namespace, `Console` is a class within namespace `System` and `WriteLine` and `Write` are methods of class `Console`.

The main difference between `WriteLine()` and `Write()` is that the `Write()` method only prints the string provided to it, while the `WriteLine()` method prints the string and moves to the start of next line as well.

Console.WriteLine() vs. Console.Write()

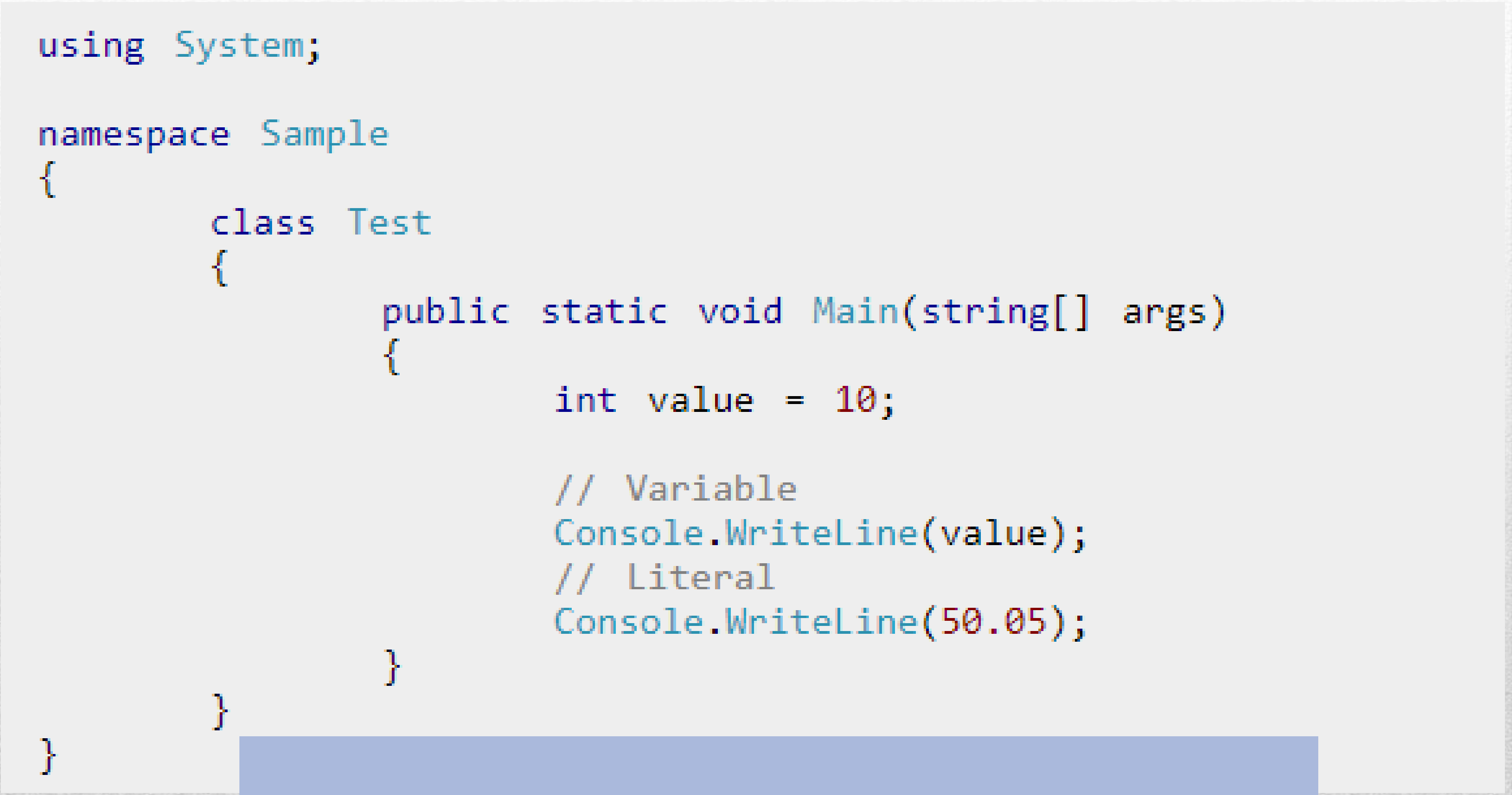


```
using System;

namespace Sample
{
        class Test
        {
                public static void Main(string[] args)
                {
                        Console.WriteLine("Prints on ");
                        Console.WriteLine("New line");

                        Console.Write("Prints on ");
                        Console.Write("Same line");
                }
        }
}
```

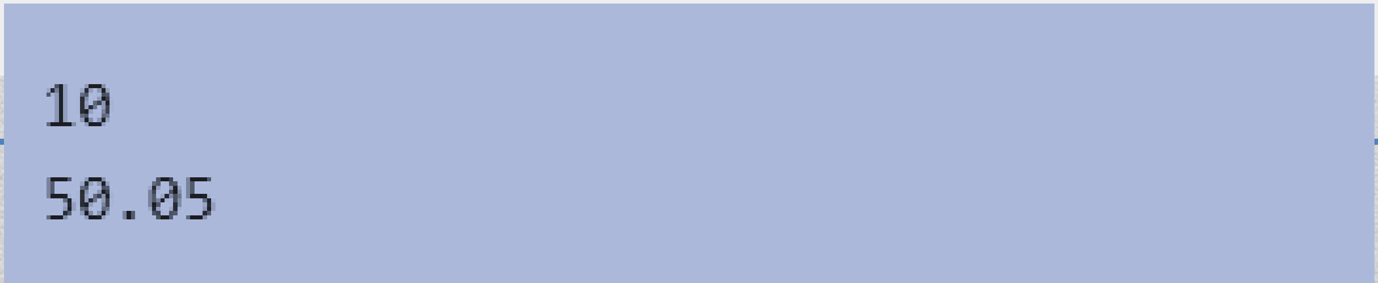


```
Prints on
New line
Prints on Same line
```

PRINTING VARIABLES & LITERALS

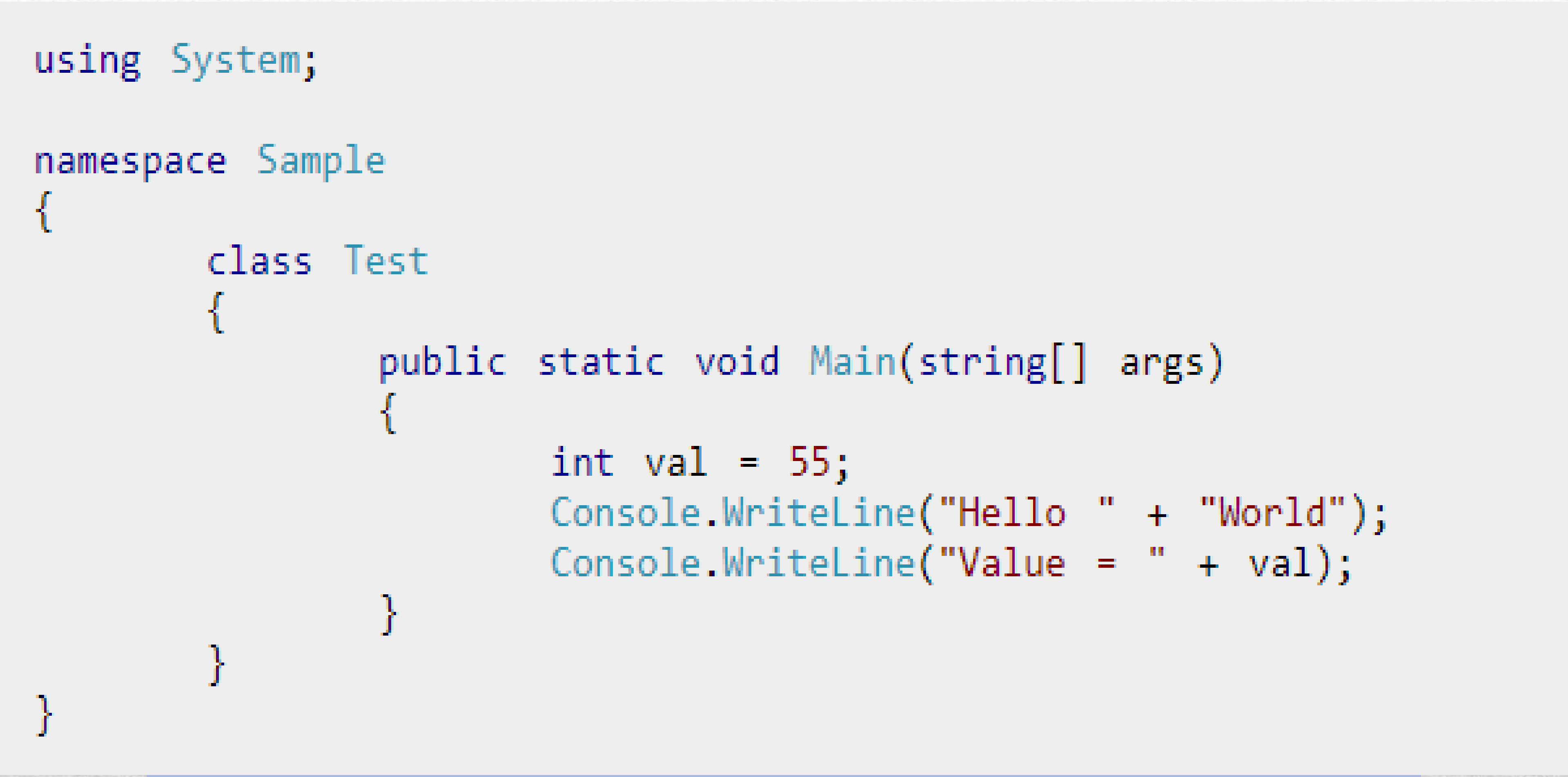


```
using System;

namespace Sample
{
        class Test
        {
                public static void Main(string[] args)
                {
                        int value = 10;

                        // Variable
                        Console.WriteLine(value);
                        // Literal
                        Console.WriteLine(50.05);
                }
        }
}
```

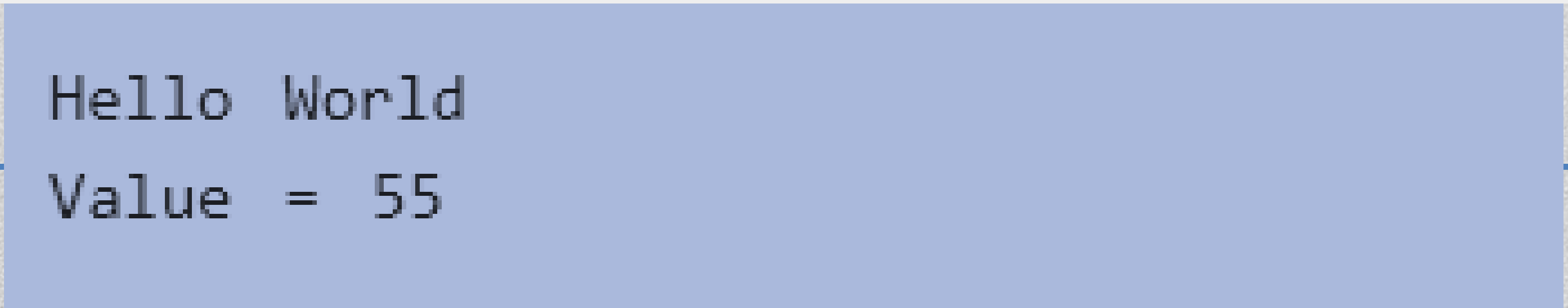


```
10
50.05
```

Printing Concatenated String using + operator

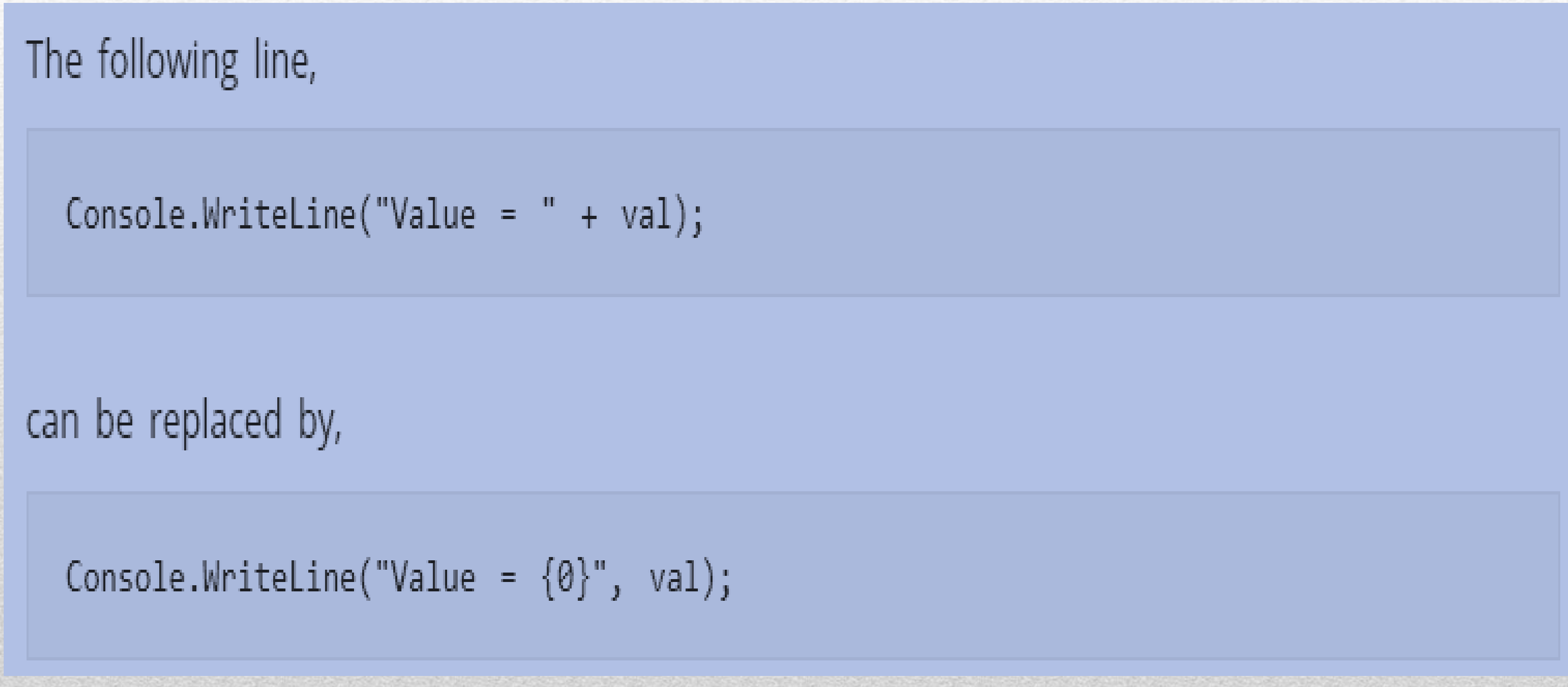


```
using System;

namespace Sample
{
        class Test
        {
                public static void Main(string[] args)
                {
                        int val = 55;
                        Console.WriteLine("Hello " + "World");
                        Console.WriteLine("Value = " + val);
                }
        }
}
```



```
Hello World
Value = 55
```

Printing concatenated string using Formatted String



The following line,

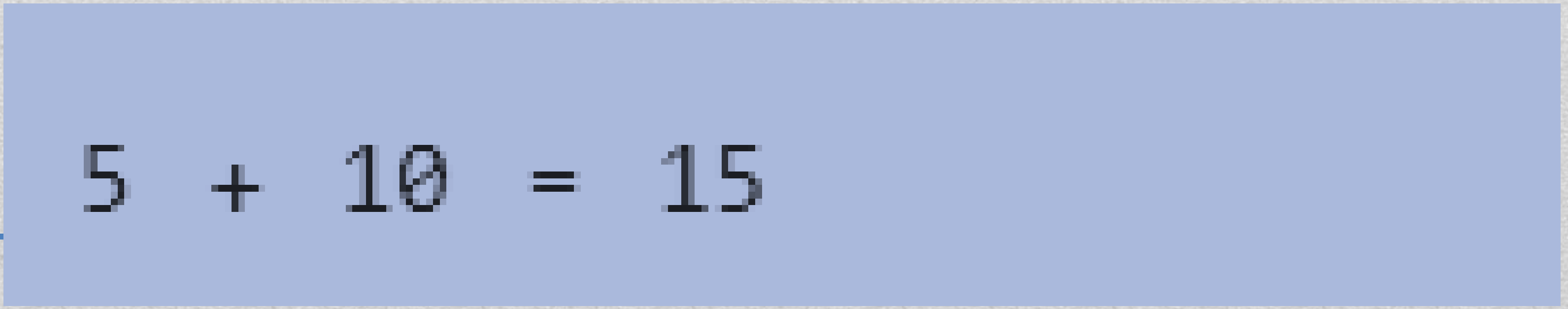
```
Console.WriteLine("Value = " + val);
```

can be replaced by,

```
Console.WriteLine("Value = {0}", val);
```

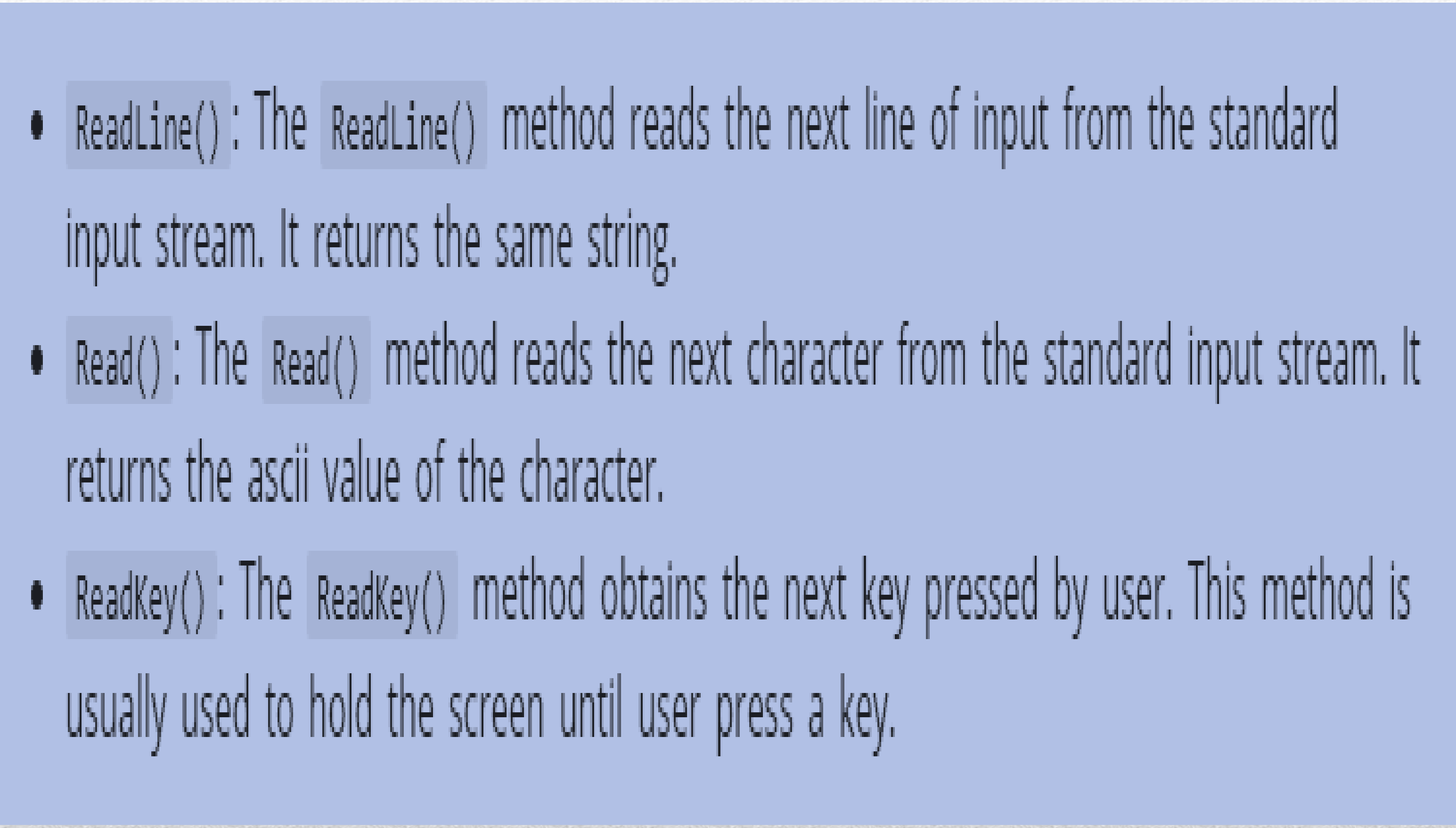
Printing concatenated string using Formatted String

```
public static void Main(string[] args)
{
        int firstNumber = 5, secondNumber = 10, result;
        result = firstNumber + secondNumber;
        Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);
}
```



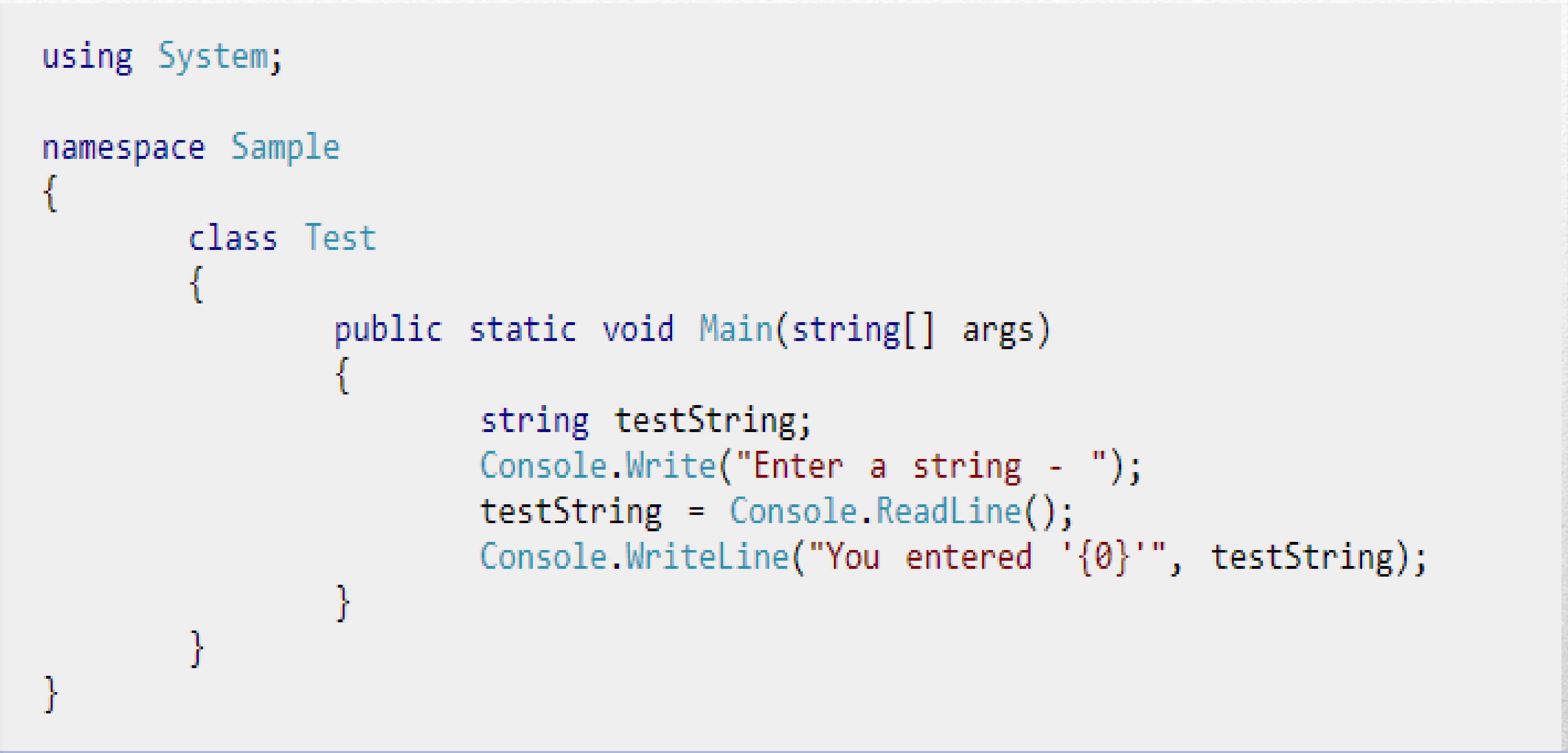
```
5 + 10 = 15
```

C# INPUT

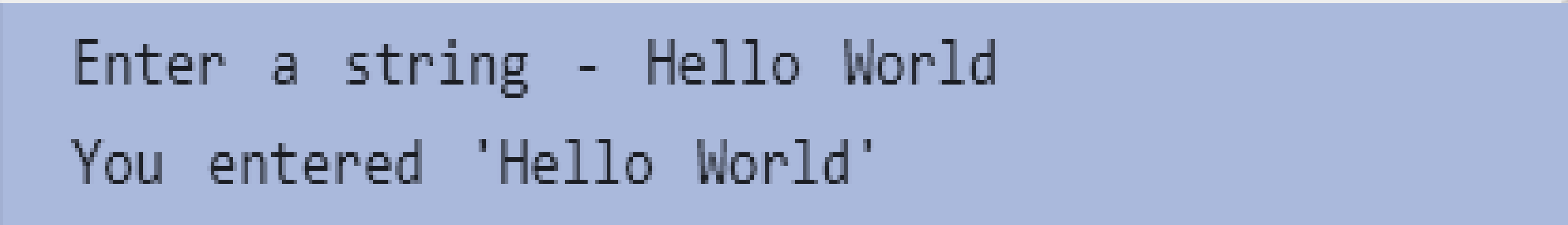


- `ReadLine()`: The `ReadLine()` method reads the next line of input from the standard input stream. It returns the same string.
- `Read()`: The `Read()` method reads the next character from the standard input stream. It returns the ascii value of the character.
- `ReadKey()`: The `ReadKey()` method obtains the next key pressed by user. This method is usually used to hold the screen until user press a key.

C# INPUT

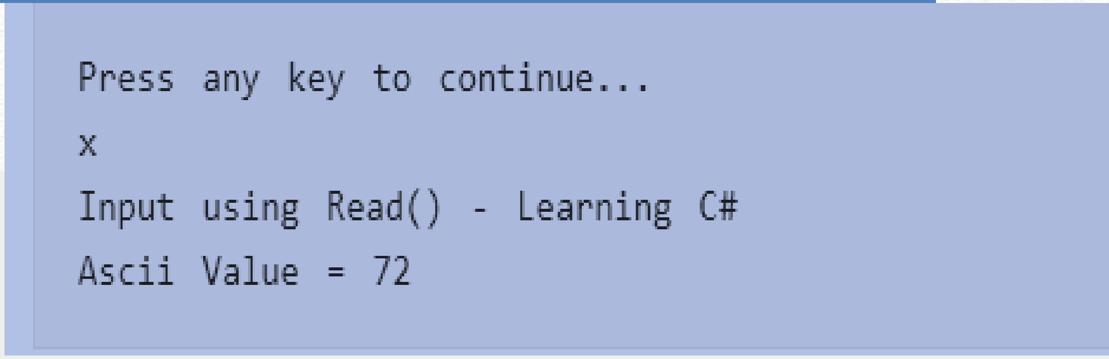


```
using System;

namespace Sample
{
        class Test
        {
                public static void Main(string[] args)
                {
                        string testString;
                        Console.Write("Enter a string - ");
                        testString = Console.ReadLine();
                        Console.WriteLine("You entered '{0}'", testString);
                }
        }
}
```

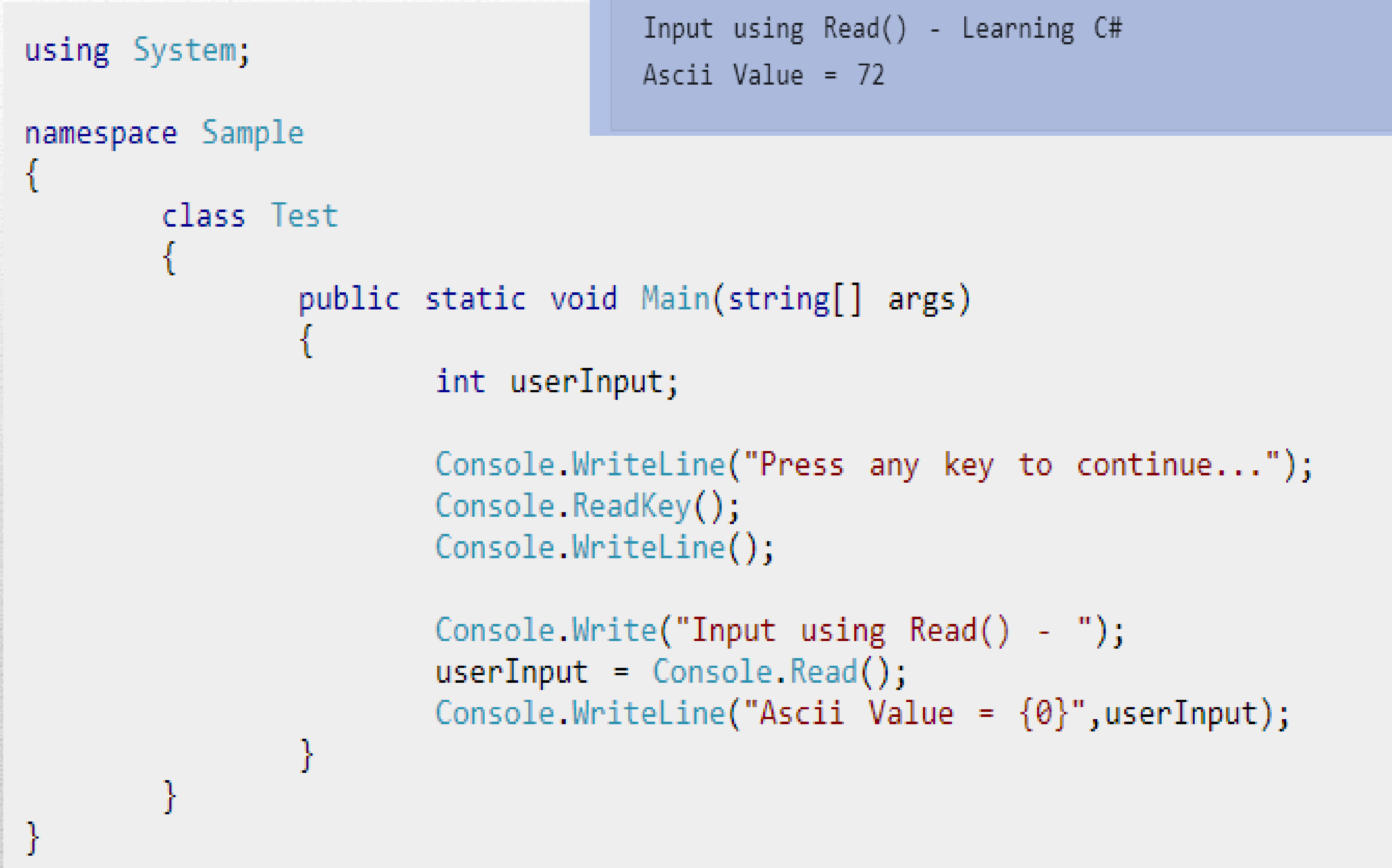


```
Enter a string - Hello World
You entered 'Hello World'
```

C# INPUT



```
Press any key to continue...
x
Input using Read() - Learning C#
Ascii Value = 72
```



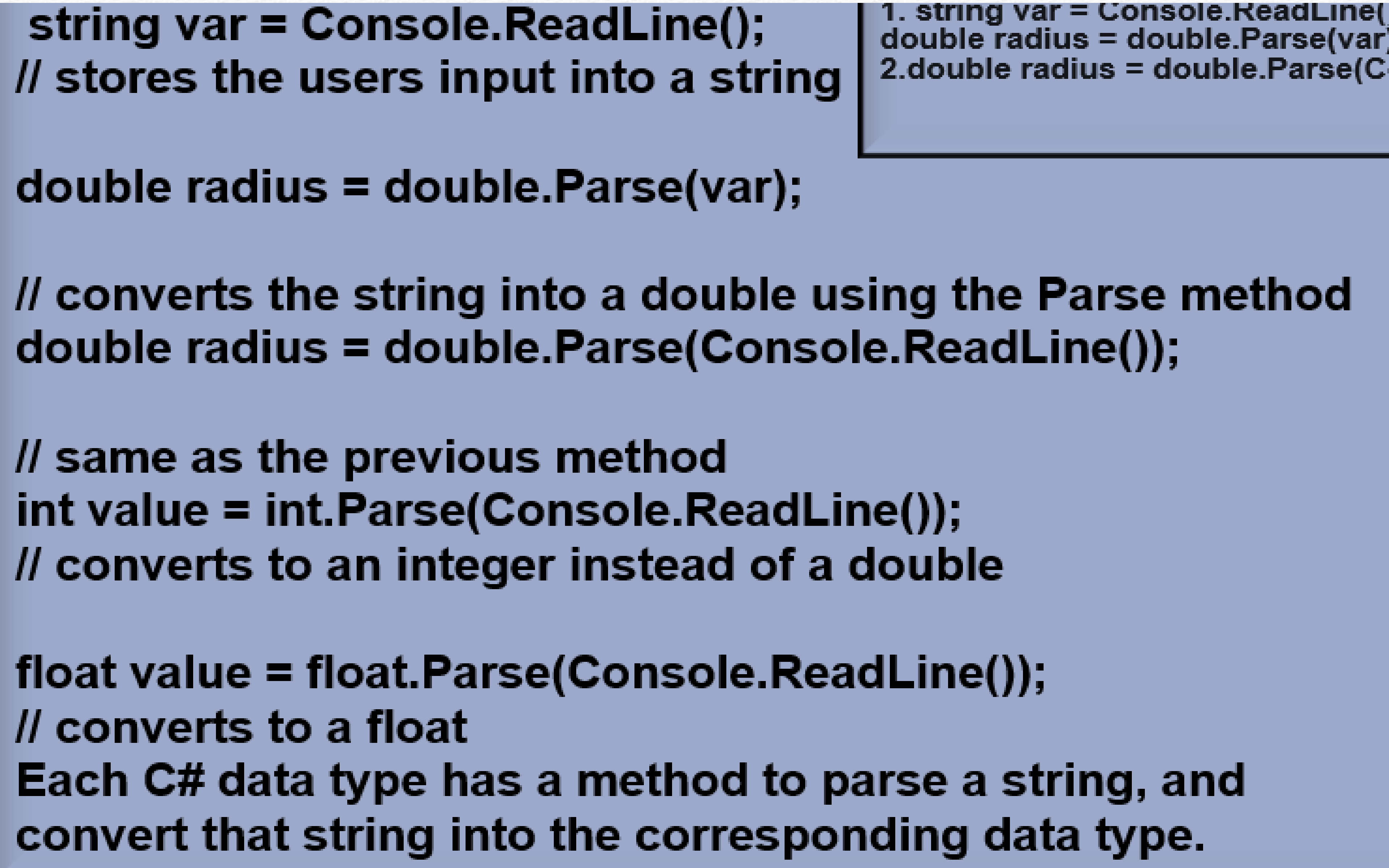
```
using System;

namespace Sample
{
        class Test
        {
                public static void Main(string[] args)
                {
                        int userInput;

                        Console.WriteLine("Press any key to continue...");
                        Console.ReadKey();
                        Console.WriteLine();

                        Console.Write("Input using Read() - ");
                        userInput = Console.Read();
                        Console.WriteLine("Ascii Value = {0}",userInput);
                }
        }
}
```

C# INPUT NUMERIC DATA

```
Enter integer value: 101
You entered 101
Enter double value: 59.412
You entered 59.412
```

```
using System;

namespace UserInput
{
        class MyClass
        {
                public static void Main(string[] args)
                {
                        string userInput;
                        int intVal;
                        double doubleVal;

                        Console.Write("Enter integer value: ");
                        userInput = Console.ReadLine();
                        /* Converts to integer type */
                        intVal = Convert.ToInt32(userInput);
                        Console.WriteLine("You entered {0}",intVal);

                        Console.Write("Enter double value: ");
                        userInput = Console.ReadLine();
                        /* Converts to double type */
                        doubleVal = Convert.ToDouble(userInput);
                        Console.WriteLine("You entered {0}",doubleVal);
                }
        }
}
```

C# INPUT NUMERIC DATA



```
 string var = Console.ReadLine();
// stores the users input into a string

double radius = double.Parse(var);

// converts the string into a double using the Parse method
double radius = double.Parse(Console.ReadLine());

// same as the previous method
int value = int.Parse(Console.ReadLine());
// converts to an integer instead of a double

float value = float.Parse(Console.ReadLine());
// converts to a float
```

Each C# data type has a method to parse a string, and convert that string into the corresponding data type.

```
1. string var = Console.ReadLine(
double radius = double.Parse(var
2.double radius = double.Parse(C
```

CREATING CLASSES

```
class Calculator
{
  public long Addem(int value1, int value2)
  {
    return value1 + value2;
  }
}
```

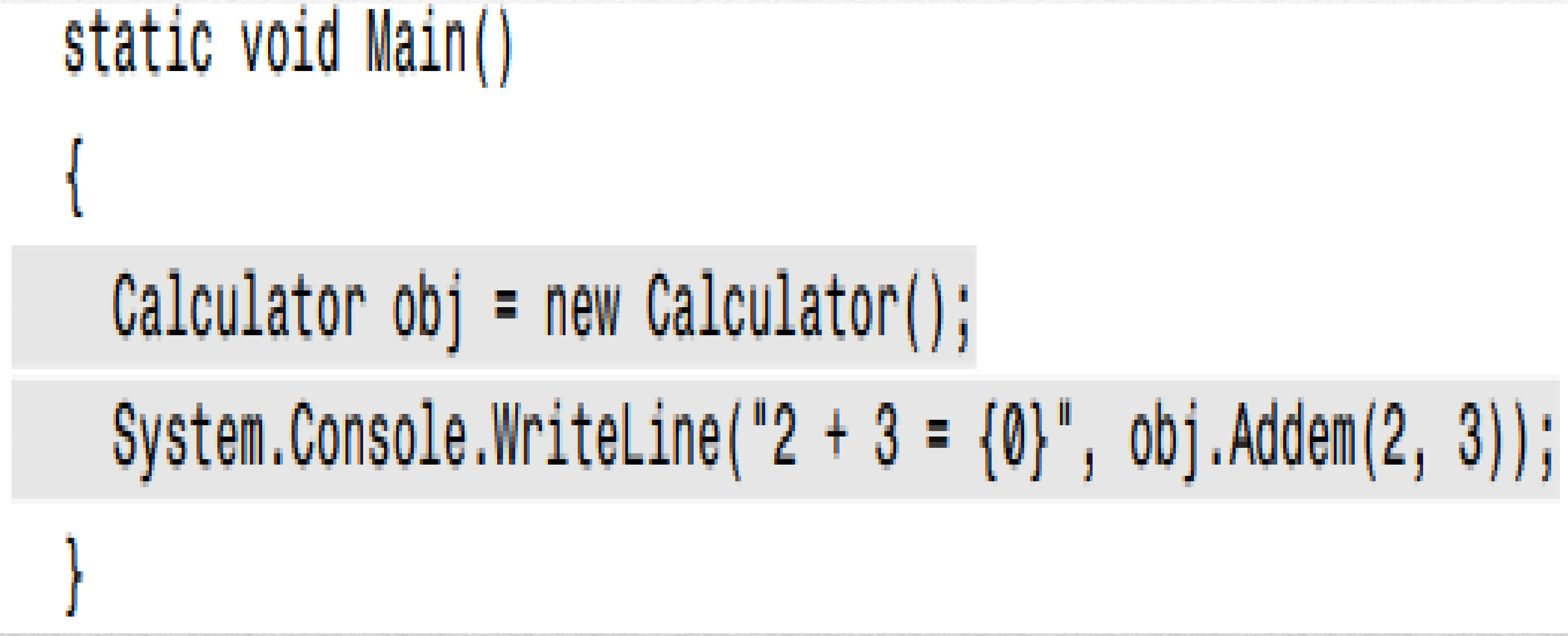
We've created a new class now—the `Calculator` class. To put that class to use, we simply have to create an object of that class.

CREATING OBJECTS/instantiation of classes

Calculator obj = new Calculator();

- **These parentheses are necessary when you use the new keyword.**
- **They let you pass data to a class's constructor, the special method that lets you initialize the data in a class.**
- **Classes must be instantiated dynamically with use of the new operator**
- **The new operator returns a reference to the new object**

CREATING OBJECTS/instantiation of classes



```
static void Main()
{
    Calculator obj = new Calculator();
    System.Console.WriteLine("2 + 3 = {0}", obj.Addem(2, 3));
}
```

CLASS ENCAPSULATION AND DATA HIDING

- **In object-oriented programming, encapsulation is the term for not only grouping data and behavior,**
- **but also hiding data within a class (the capsule)**
- **so that minimum access about the inner workings of a class is exposed outside the class.**
- **This reduces the chances that callers will modify the data inappropriately.**

ACCESS MODIFIERS

- **use access modifiers to set the allowed access to not only classes, but also to all members of those classes.**

- **five accessibility levels can be specified using the access modifiers: public,**
  - **protected,**
  - **internal,**
  - **internal protected, and**
  - **private.**

ACCESS MODIFIERS

- **The public keyword gives a type or type member public access, the most permissive access level. There are no restrictions on accessing public members.**

- **The protected keyword gives a type or type member protected access, which means it’s accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.**

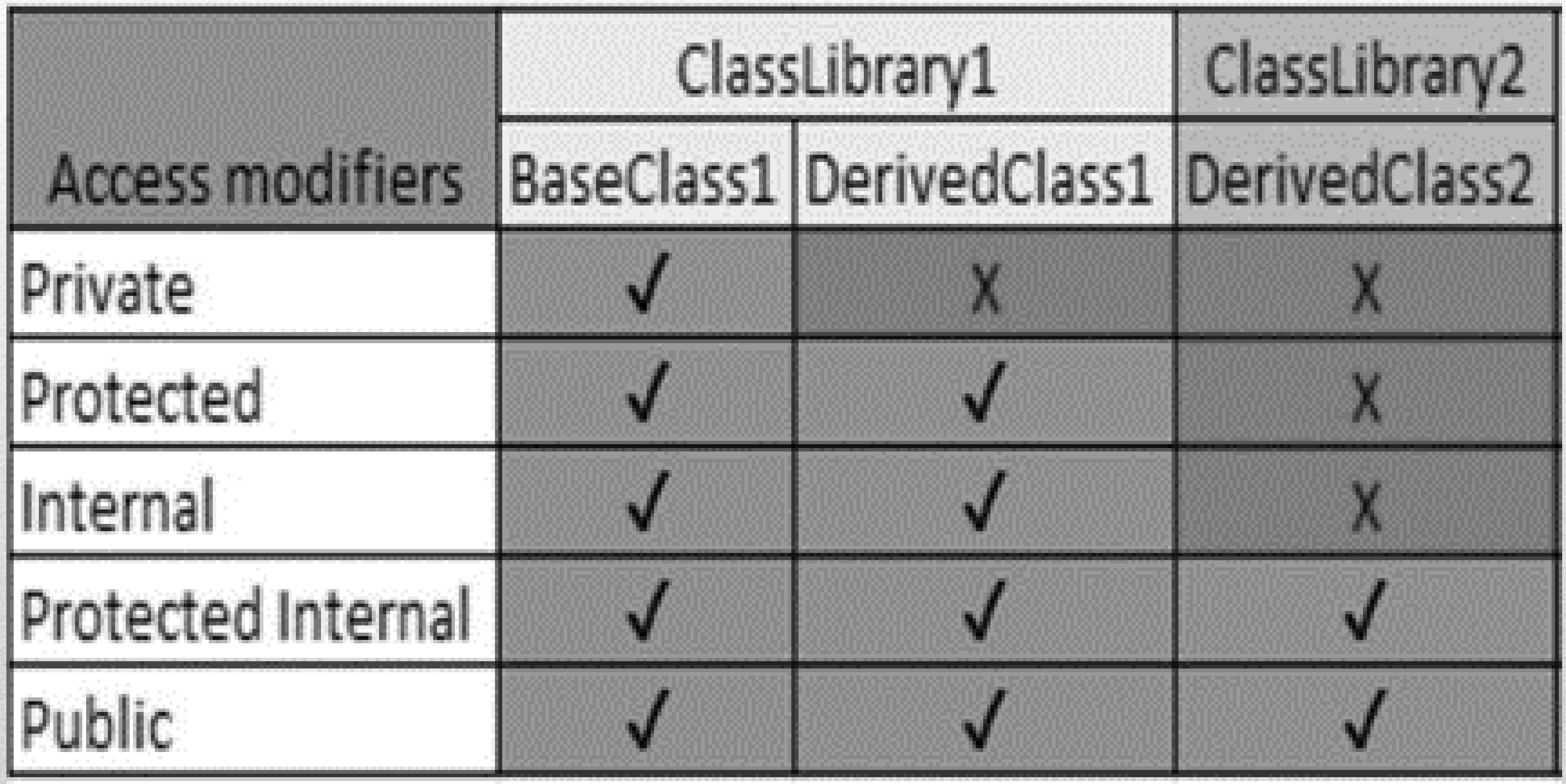
ACCESS MODIFIERS

- The internal keyword gives a type or type member internal access, which is accessible only within files in the same assembly. It is an error to reference a member with internal access outside the assembly within which it was defined.

- The private keyword gives a type or type member private access, which is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared.

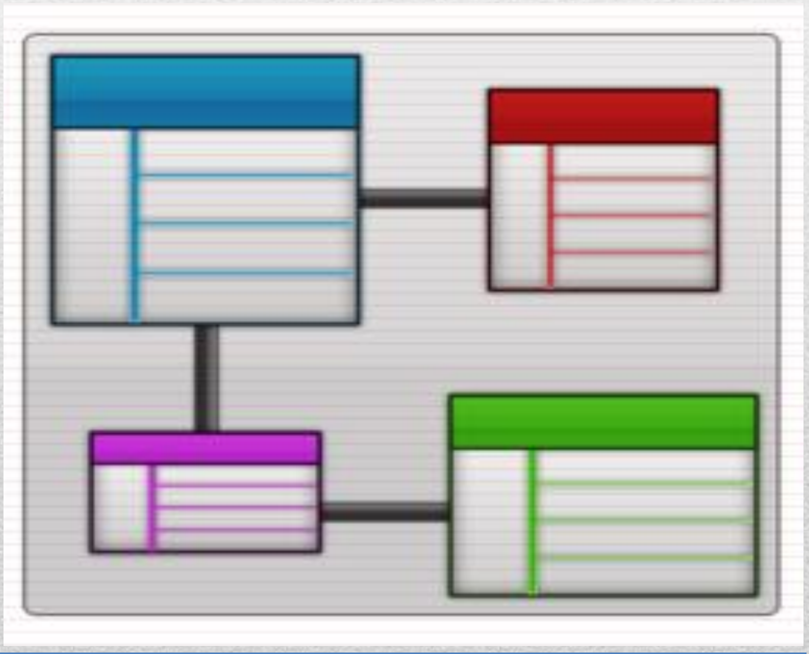
- If you don’t specify an access modifier for a type or type member, the default access modifier is private

ACCESS MODIFIERS VISIBILITY CHART



Access modifiers	ClassLibrary1		ClassLibrary2
	BaseClass1	DerivedClass1	DerivedClass2
Private	✓	X	X
Protected	✓	✓	X
Internal	✓	✓	X
Protected Internal	✓	✓	✓
Public	✓	✓	✓

WEEK#4
OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

SET & GET METHODS

- Set and Get methods can validate attempts to modify private data and control how that data is presented to the caller, respectively.

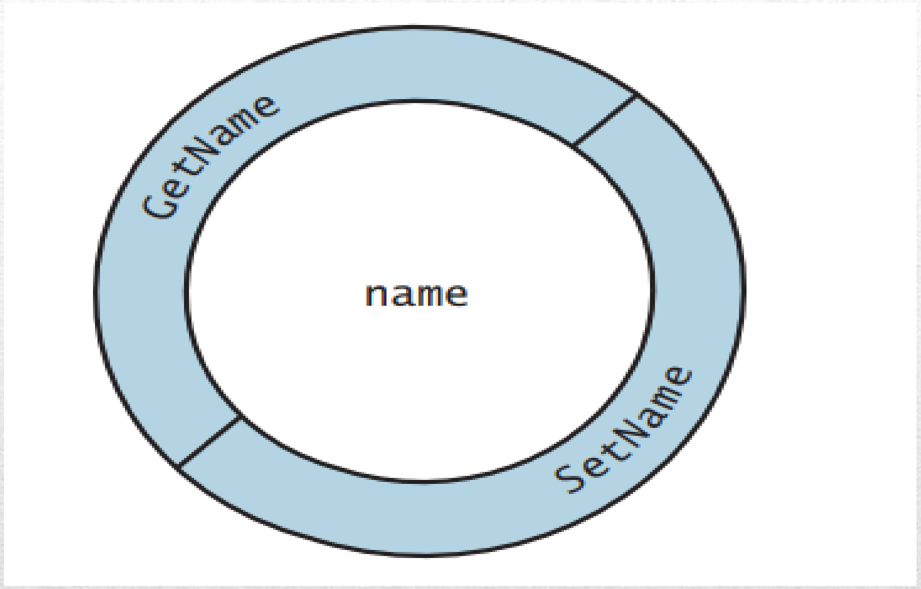
- Set methods can be programmed to validate their arguments and reject any attempts to Set the data to bad values, such as • a negative body temperature • a day in March outside the range 1 through 31 • a product code not in the company’s product catalog, etc

SET & GET METHODS

- A Get method can present the data in a different form, while the actual data representation remains hidden from the user.

- For example, a Grade class might store a grade instance variable as an int between 0 and 100, but a GetGrade method might return a letter grade as a string, such as "A" for grades between 90 and 100, "B" for grades between 80 and 89, …

SET & GET METHODS



- The private instance variable name is hidden inside the object (represented by the inner circle containing name) and guarded by an outer layer of public methods (represented by the outer circle containing GetName and SetName).

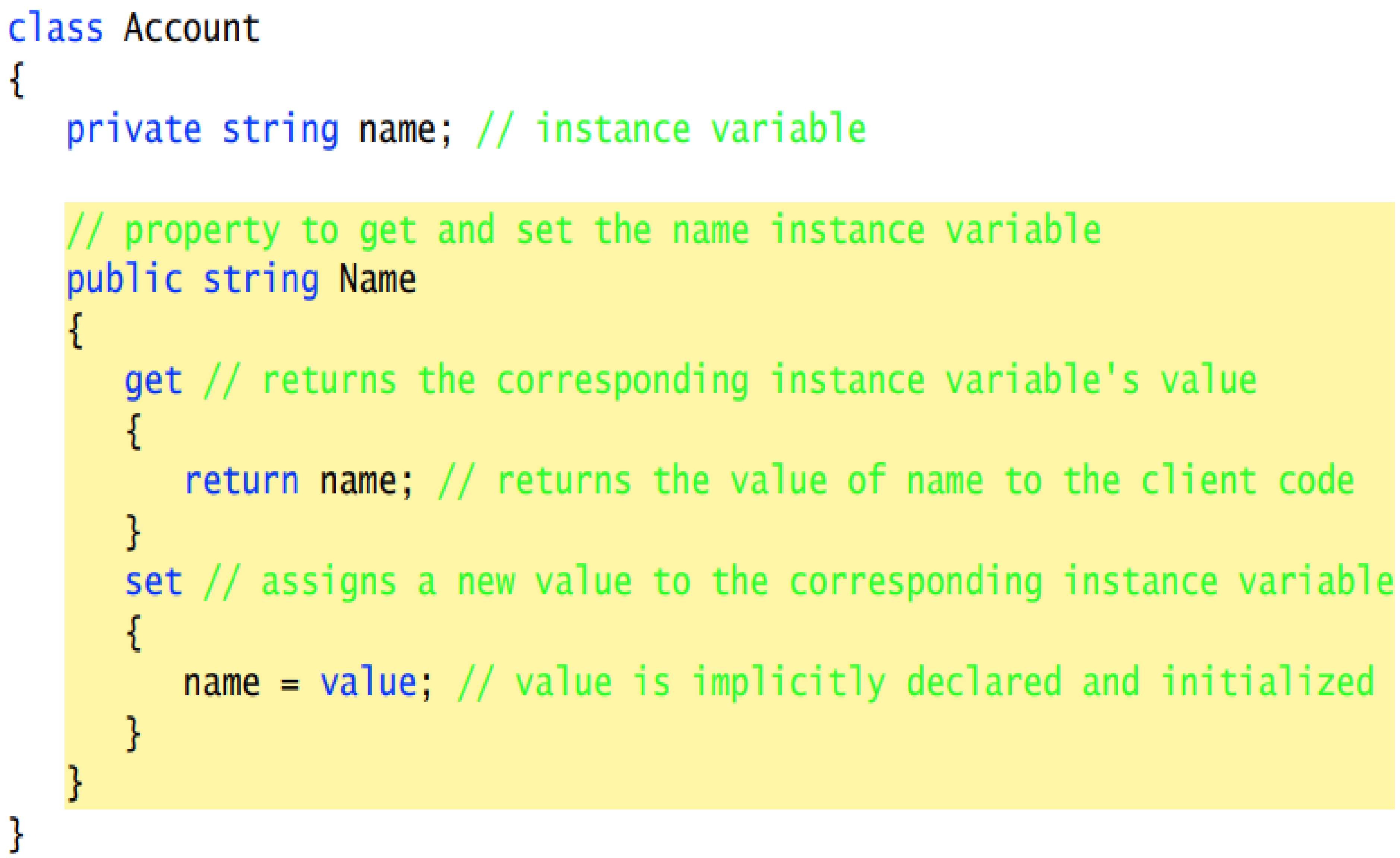
- Any client code that needs to interact with the Account object can do so only by calling the public methods of the protective outer layer.

PROPERTIES

- **C# provides a more elegant solution—called properties—to accomplish the same tasks as set and get methods.**

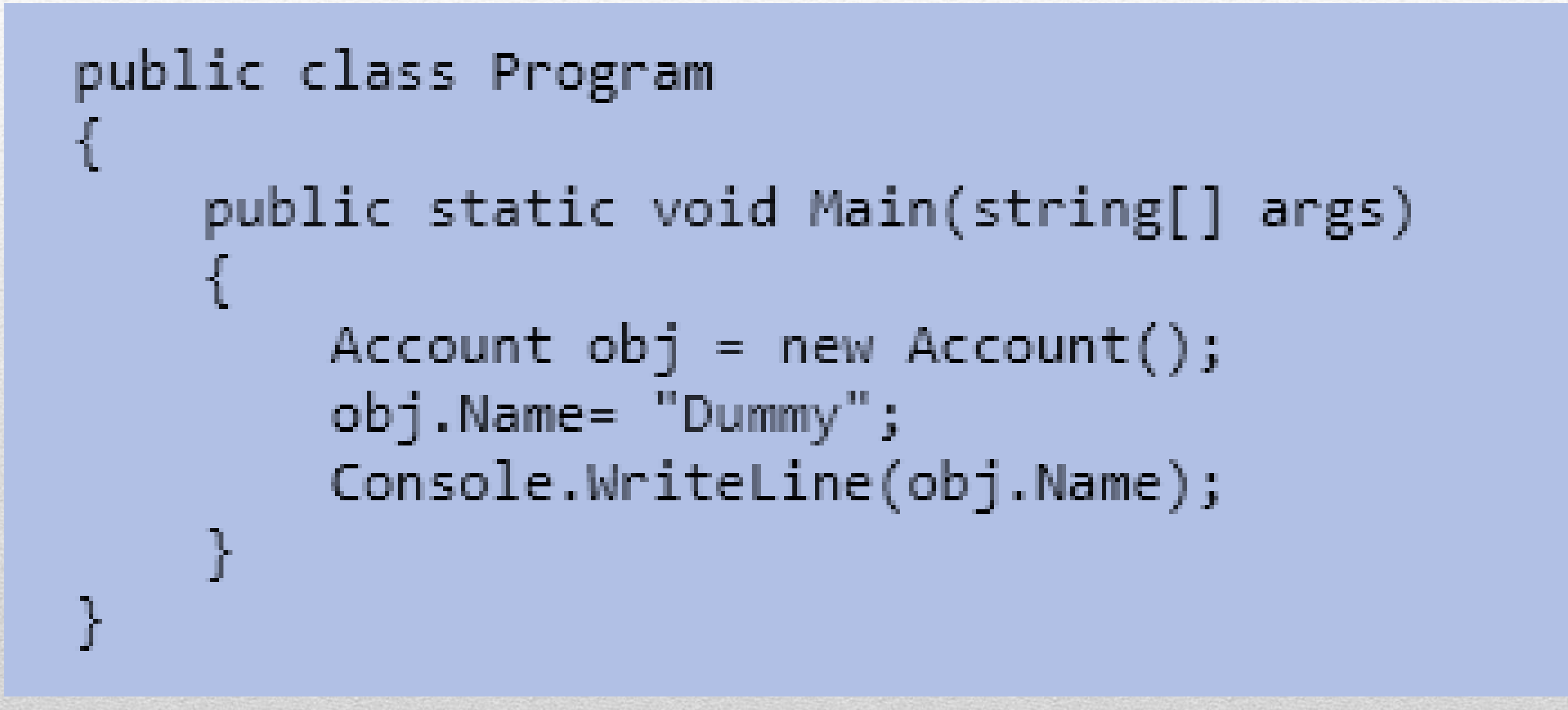
- **A property encapsulates a set accessor for storing a value into a variable and a get accessor for getting the value of a variable.**

PROPERTIES



```
class Account
{
   private string name; // instance variable

   // property to get and set the name instance variable
   public string Name
   {
      get // returns the corresponding instance variable's value
      {
         return name; // returns the value of name to the client code
      }
      set // assigns a new value to the corresponding instance variable
      {
         name = value; // value is implicitly declared and initialized
      }
   }
}
```

USING PROPERTY FROM MAIN()



```
public class Program
{
    public static void Main(string[] args)
    {
        Account obj = new Account();
        obj.Name= "Dummy";
        Console.WriteLine(obj.Name);
    }
}
```

Property Name’s Declaration

public string Name

specifies that

- the property is public so it can be used by the class’s clients,
- the property’s type is string
- the property’s name is Name.
- By convention, a property’s identifier is the capitalized identifier of the instance variable that it manipulates—
    - Name is the property that represents instance variable name.
    - C# is case sensitive, so Name and name are distinct identifiers.

- The property’s body is enclosed in the braces.

PROPERTIES

Read-only Properties

You can also create read-only properties if you omit the set accessor method.

```
class Customer
{
        private string name;
        public string Name
        {
                get
                {
                        return name;
                }
        }
}
```

CONSTRUCTOR

- A constructor is a special function that is a member of a class.
- C# requires a constructor call for each object that's created, which helps ensure that each object is initialized properly before it's used in a program.
- The constructor call occurs implicitly when the object is created. This ensures that objects will always have valid data to work on.
- Normally, constructors are declared public.
- Constructors can be overloaded, just like other functions.

DIFFERENCE BETWEEN CONSTRUCTORS AND OTHER FUNCTIONS

- **A constructor must be defined with the same name as the class.**

- **Constructors do not possess a return type (not even void).**

TYPES OF CONSTRUCTORS

DEFAULT CONSTRUCTOR

- A constructor without having any parameters is called default constructor.
- In this constructor every instance of the class will be initialized without any parameter values.

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;
public Sample()      // Default Constructor
{
param1 = “Hello";
param2 = “World";
}
}
class Program
{
static void Main(string[] args)
{
Sample obj=new Sample();    // Once object of class created automatically constructor will be
called
Console.WriteLine(obj.param1);
Console.WriteLine(obj.param2);
Console.ReadLine();
}
}
}
```

OUTPUT:

```
Hello
World
```

TYPES OF CONSTRUCTORS

PARAMETERIZED CONSTRUCTOR

- A constructor with at least one parameter is called as parameterized constructor.

- In parameterized constructor we can initialize each instance of the class to different values like as shown below

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;
public Sample(string x, string y)     // Declaring Parameterized constructor with Parameters
{
param1 = x;
param2 = y;
}
}
class Program
{
static void Main(string[] args)
{
Sample obj=new Sample("Welcome","dotnet");   // Parameterized Constructor Called
Console.WriteLine(obj.param1 +" to "+ obj.param2);
Console.ReadLine();
}
}
}
```

OUTPUT:

```
Welcome to dotnet
```

COPY CONSTRUCTOR

- **A parameterized constructor that contains a parameter of same class type is called as copy constructor.**

- **Main purpose of copy constructor is to initialize new instance to the values of an existing instance.**

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;
public Sample(string x, string y)
{
param1 = x;
param2 = y;
}
public Sample(Sample obj)      // Copy Constructor
{
param1 = obj.param1;
param2 = obj.param2;
}
}
class Program
{
static void Main(string[] args)
{
Sample obj = new Sample("Welcome", "dotnet");  // Create instance to class Sample
Sample obj1=new Sample(obj); // Here obj details will copied to obj1
Console.WriteLine(obj1.param1 +" to " + obj1.param2);
Console.ReadLine();
}
}}
```

OUTPUT:

```
Welcome to dotnet
```

CONSTRUCTOR OVERLOADING

- In c# we can overload constructor by creating another constructor with same method name and different parameters.

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;


public Sample()      // Default Constructor
{
param1 = "Hi";
param2 = "I am Default Constructor";
}
public Sample(string x, string y)      // Declaring Parameterized constructor with Parameters
{
param1 = x;
param2 = y;
}
}
class Program
{
static void Main(string[] args)
{
Sample obj = new Sample();    // Default Constructor will Called
Sample obj1=new Sample("Welcome","dotnet");    // Parameterized Constructor will Called
Console.WriteLine(obj.param1 + ", "+obj.param2);
Console.WriteLine(obj1.param1 +" to " + obj1.param2);
Console.ReadLine();
}
}
```

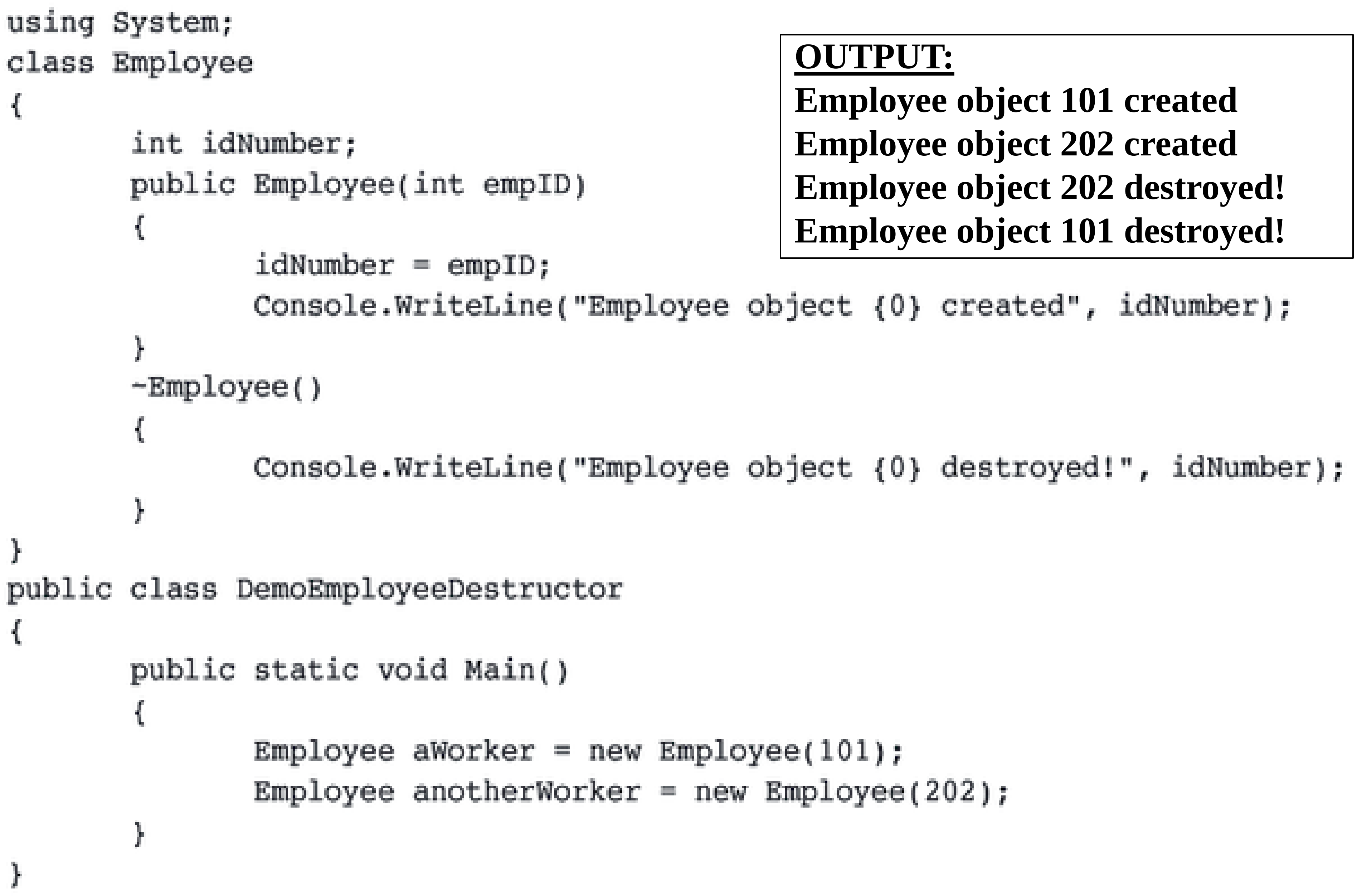
OUTPUT:

```
Hi, I am Default Constructor
Welcome to dotnet
```

Understanding Destructor Methods

- A **destructor method** contains the actions you require when an instance of a class is destroyed
- To explicitly declare a destructor method, use an identifier that consists of a tilde(~) followed by the class name
- Destructors cannot receive parameters and therefore cannot be overloaded
- A class can have at most one destructor

• Employee class with destructor method



```
using System;
class Employee
{
        int idNumber;
        public Employee(int empID)
        {
                idNumber = empID;
                Console.WriteLine("Employee object {0} created", idNumber);
        }
        ~Employee()
        {
                Console.WriteLine("Employee object {0} destroyed!", idNumber);
        }
}
public class DemoEmployeeDestructor
{
        public static void Main()
        {
                Employee aWorker = new Employee(101);
                Employee anotherWorker = new Employee(202);
        }
}
```

OUTPUT:

```
Employee object 101 created
Employee object 202 created
Employee object 202 destroyed!
Employee object 101 destroyed!
```

Destructor Methods

- Destructor methods are automatically called when an object goes out of scope
- You cannot explicitly call a destructor
- The last object created is the first object destroyed

METHOD OVERLOADING

- **Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters).**

- **Method overloading is commonly used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments**

```
public class Program
{
    // test overloaded square methods
    public static void Main(string[] args)
    {
        Program obj = new Program();
        Console.WriteLine("Square of integer 7 is {0}",obj.Square(7));
        Console.WriteLine("Square of double 7.5 is {0}",obj.Square(7.5));
    }

    // square method with int argument
    int Square(int intValue)
    {
         Console.WriteLine("Called square with int argument: {0}",intValue);
         return intValue * intValue;
    }

     // square method with double argument
    double Square(double doubleValue)
    {
         Console.WriteLine("Called square with double argument: {0}",doubleValue);
         return doubleValue * doubleValue;
    }
}
```

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

METHOD OVERLOADING

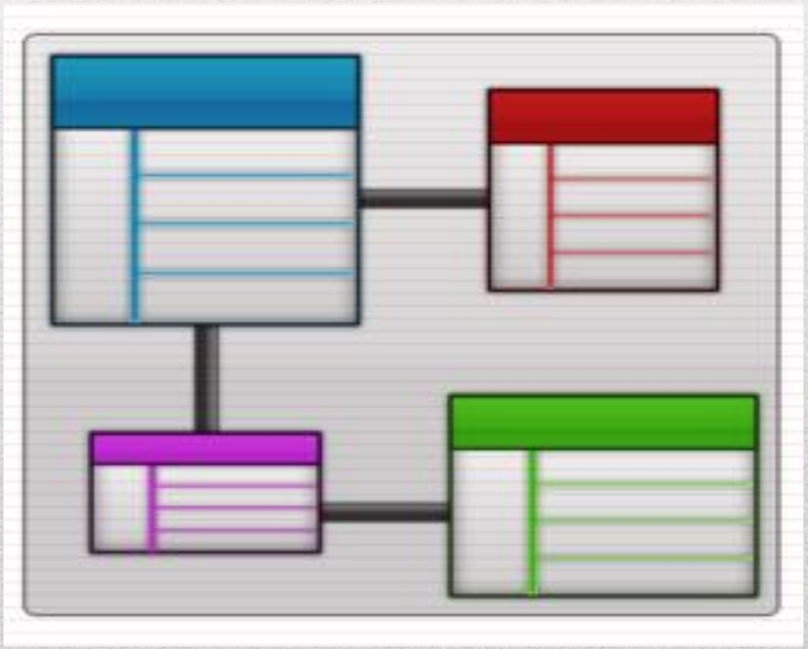
- Methods cannot be distinguished by return type. If in a class you define overloaded methods with the following headers:

```
int Square(int x)
double Square(int x)
```

the compiler generates error for the second Square method.

- Overloaded methods can have the same or different return types if the parameter lists are different. Also, overloaded methods need not have the same number of parameters.

WEEK#5
OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

STATIC MEMBERS IN A CLASS

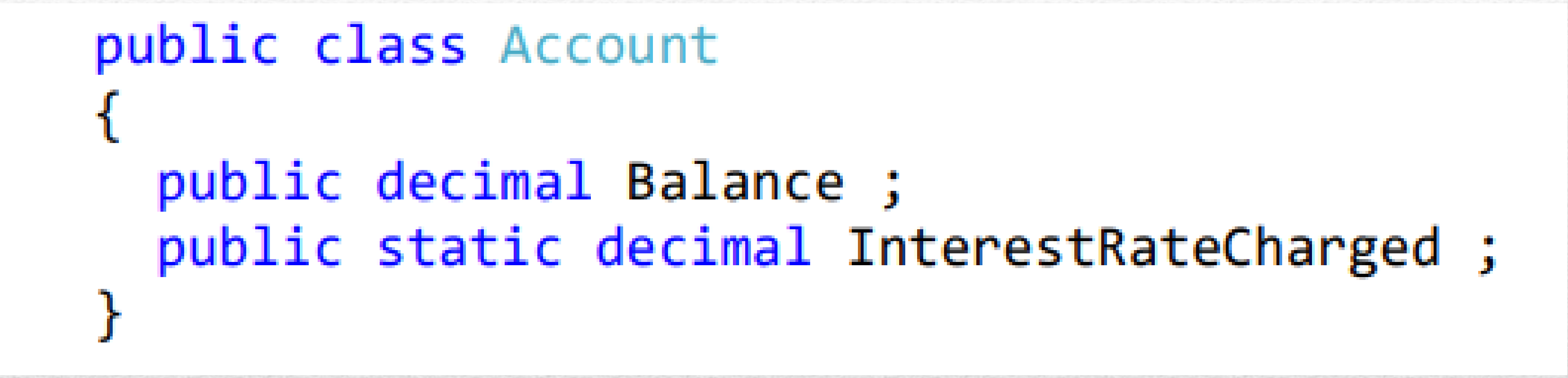
- There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A static data member is used for these and other reasons.

- Static members exist as members of the class rather than as an instance in each object of the class. So, this keyword is not available in a static member function.

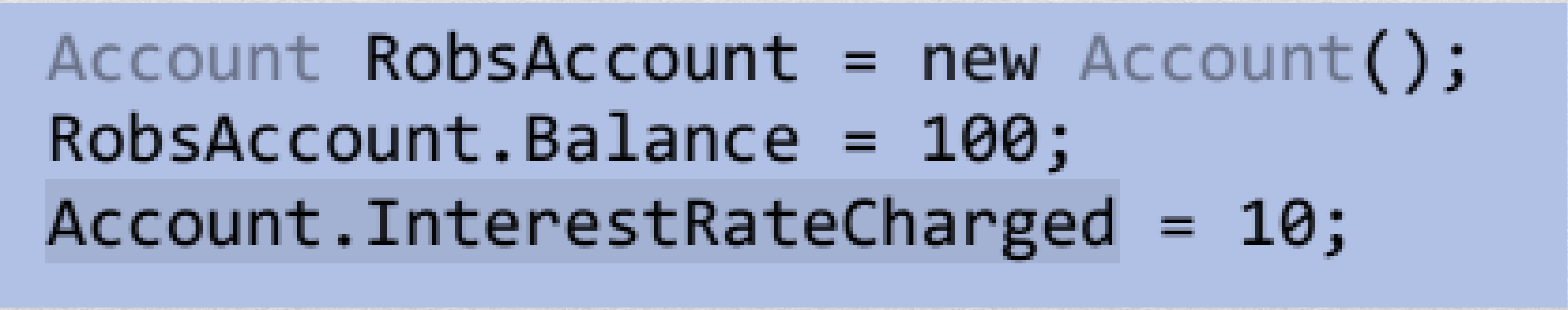
- A non-static member function can be called only after instantiating the class as an object. This is not the case with static member functions. A static member function can be called, even when a class is not instantiated.

- Static functions may access only static data members.

STATIC DATA MEMBERS IN A CLASS

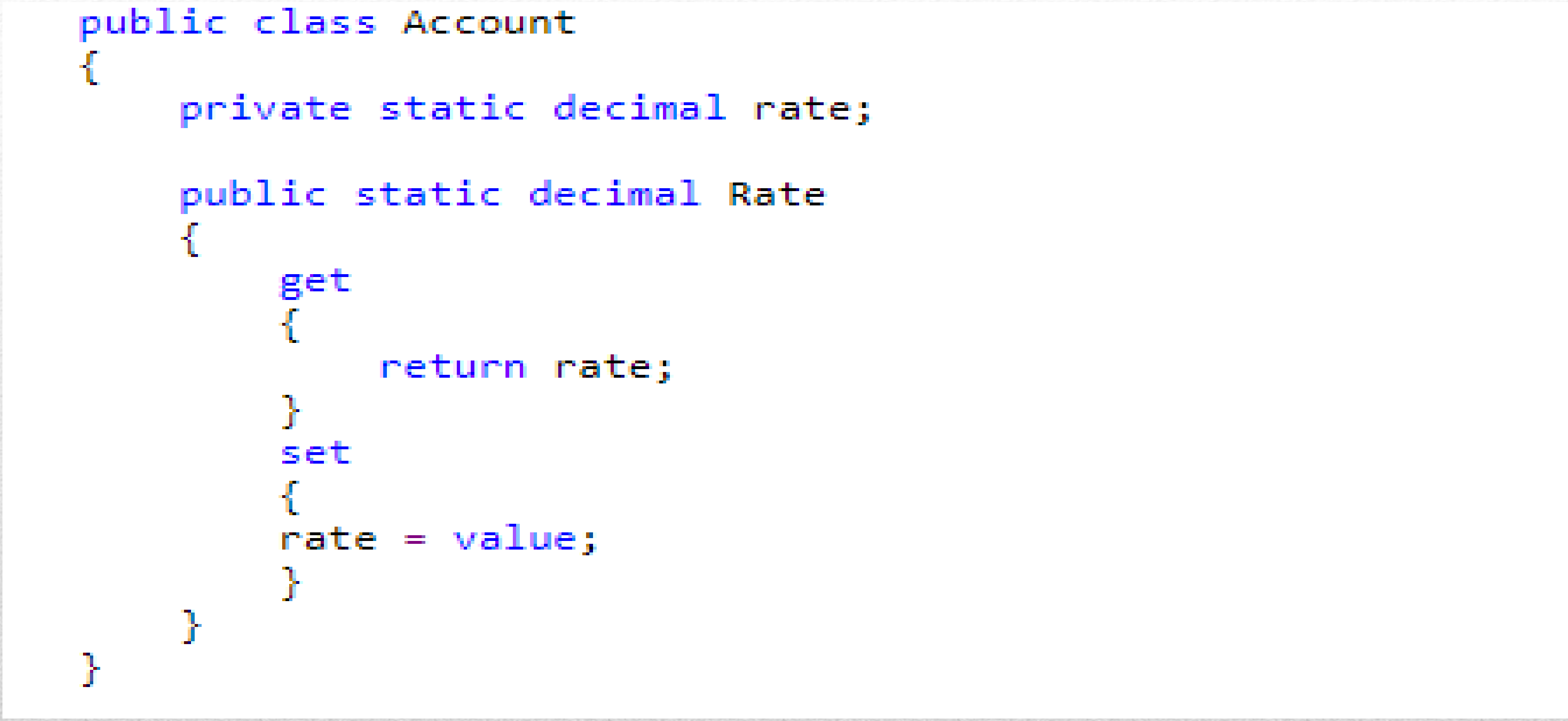


```
public class Account
{
  public decimal Balance ;
  public static decimal InterestRateCharged ;
}
```

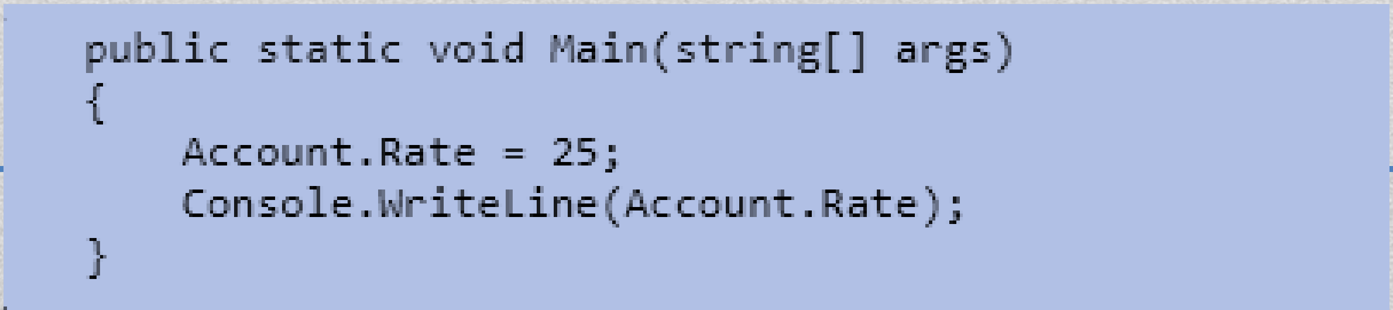


```
Account RobsAccount = new Account();
RobsAccount.Balance = 100;
Account.InterestRateCharged = 10;
```

STATIC PROPERTIES

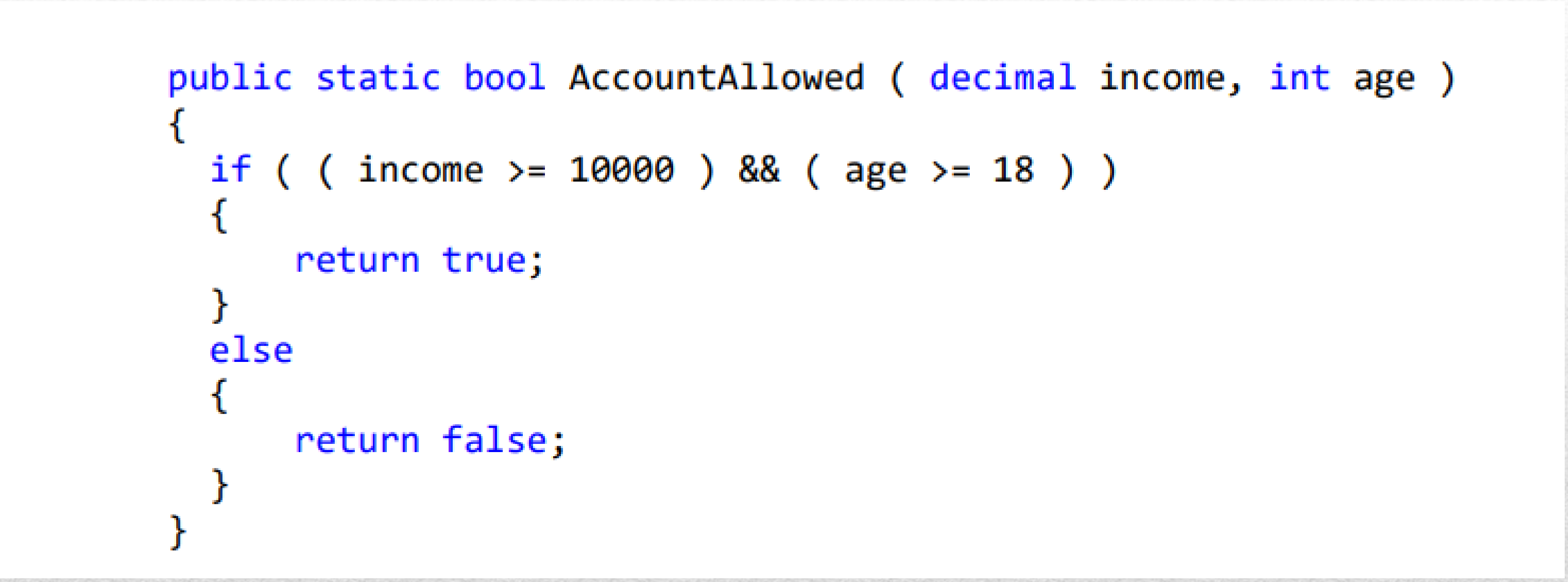


```
public class Account
{
    private static decimal rate;

    public static decimal Rate
    {
        get
        {
            return rate;
        }
        set
        {
        rate = value;
        }
    }
}
```

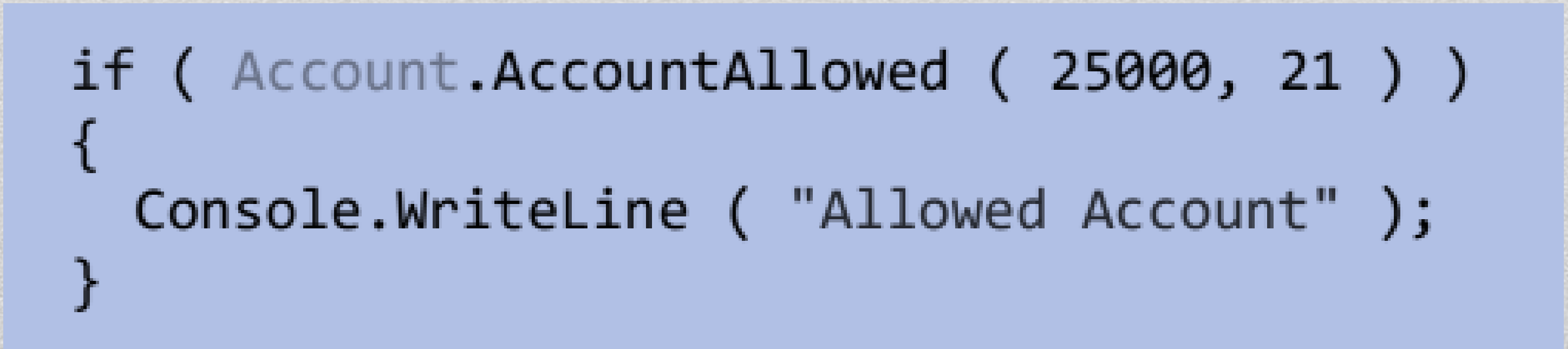


```
public static void Main(string[] args)
{
    Account.Rate = 25;
    Console.WriteLine(Account.Rate);
}
```

STATIC METHODS IN A CLASS



```
public static bool AccountAllowed ( decimal income, int age )
{
  if ( ( income >= 10000 ) && ( age >= 18 ) )
  {
      return true;
  }
  else
  {
      return false;
  }
}
```



```
if ( Account.AccountAllowed ( 25000, 21 ) )
{
  Console.WriteLine ( "Allowed Account" );
}
```

STATIC CLASSES

- This class does not have any instance fields (or methods) and therefore, creation of such a class would be pointless. Because of this, the class is decorated with the static keyword.

- Contains only static members means all the methods and members must be static

- It can have default constructor or Can have only one constructor without any parameter.

- No class can be derived from it or instantiate it(using new keyword).

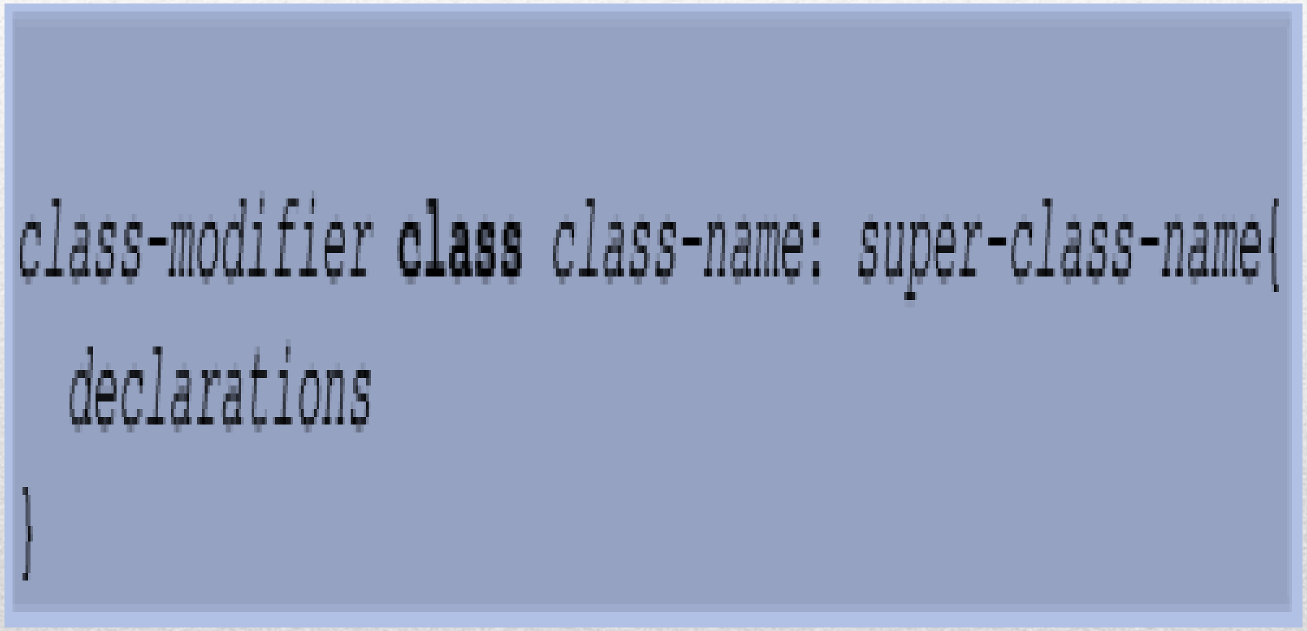
STATIC CLASSES

- **Static Classes Can only contain static constructor.**
- **Static constructor cannot have any parameter**
- **access modifiers are not allowed on static constructors**

WHEN TO HAVE STATIC CLASS?

- **We have a class Math in C#, let’s check it’s methods and operation. Say Math.Min, it takes two values of different data types and return minimum one, similarly Max, Round, Floor, Ceiling. If you notice all these methods take some values and operate only those provided values. So a static class should be container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields.**

INHERITANCE



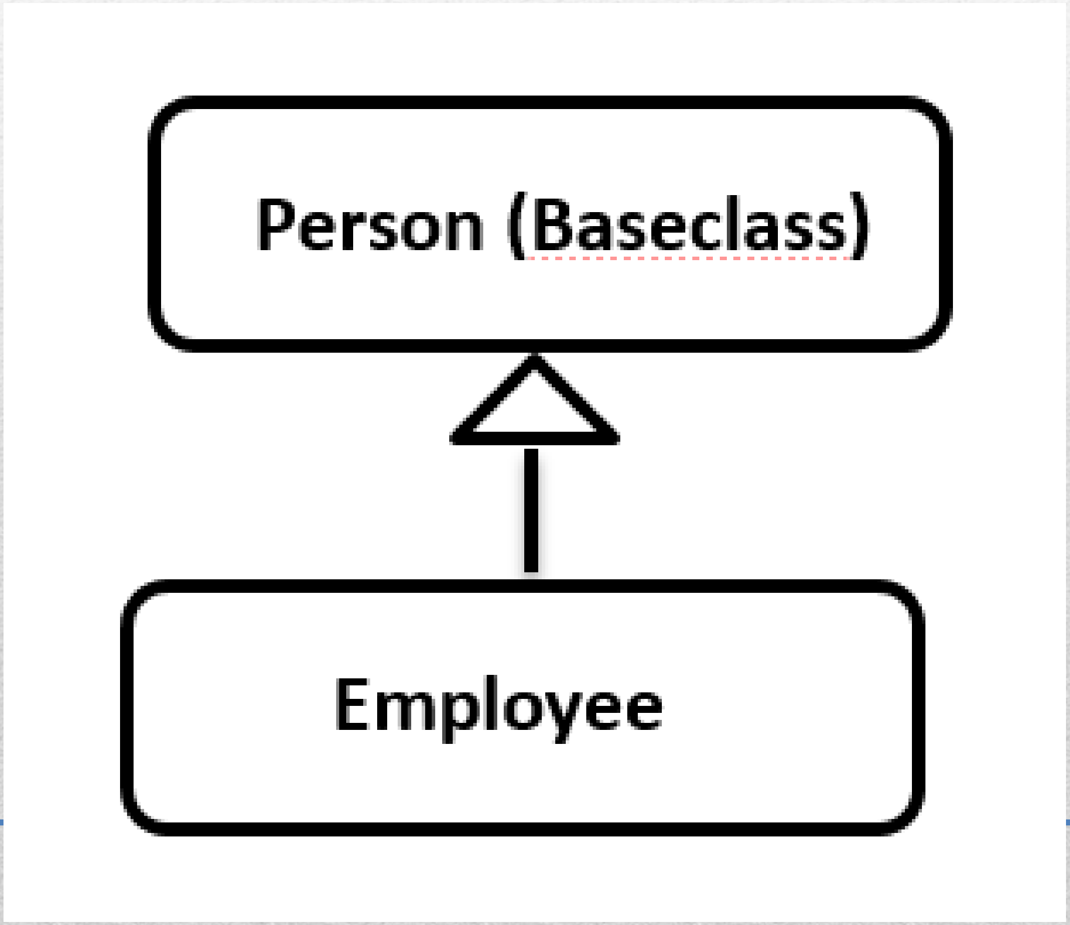
```
class-modifier class class-name: super-class-name{
  declarations
}
```

TYPES OF INHERITANCE

- **SINGLE INHERITANCE**
- **HIERARICAL**
- **MULTILEVEL**
- **MULTIPLE INHERITANCE**

TYPES OF INHERITANCE

- **SINGLE INHERITANCE**

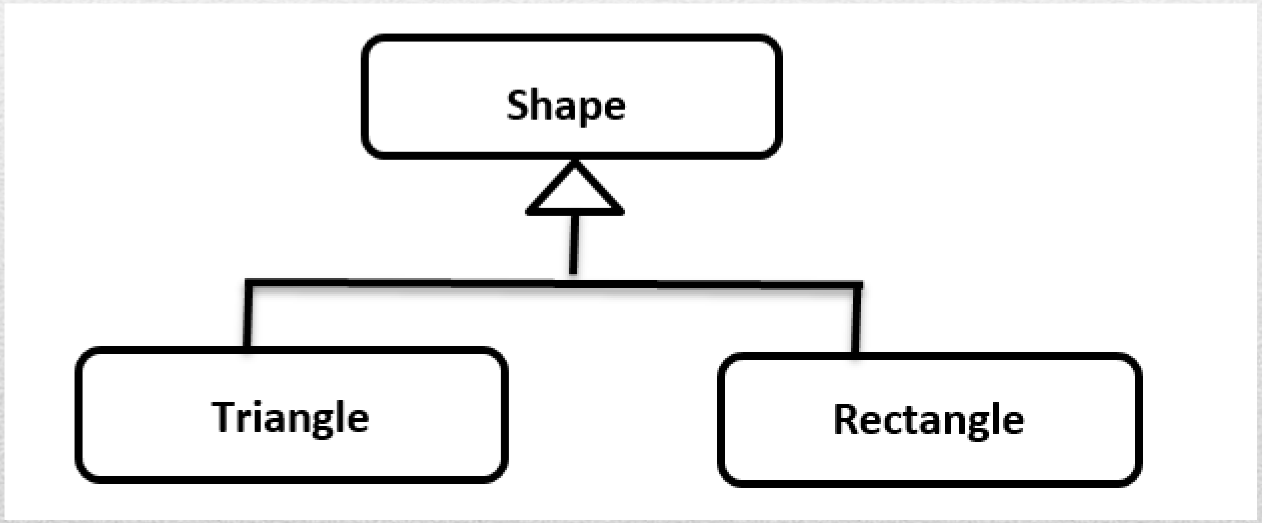


```
public class Person
    {
        private string name = "Dummy";
        private int age = 26;

        protected void Display()
        {
            Console.Write(name + "\n" + age);
        }

    }

    public class Employee : Person
    {
        public string company = "ABC Company";
        public float salary = 36000;

        public void Display()
        {
            base.Display();
            Console.Write("\n" + company + "\n" + salary + "\n");
        }
    }
```

```
public class Program
   {
       public static void Main(string[] args)
       {
          Employee emp = new Employee();
          emp.Display();
       }
   }
```

TYPES OF INHERITANCE

- **HIERARICAL INHERITANCE**

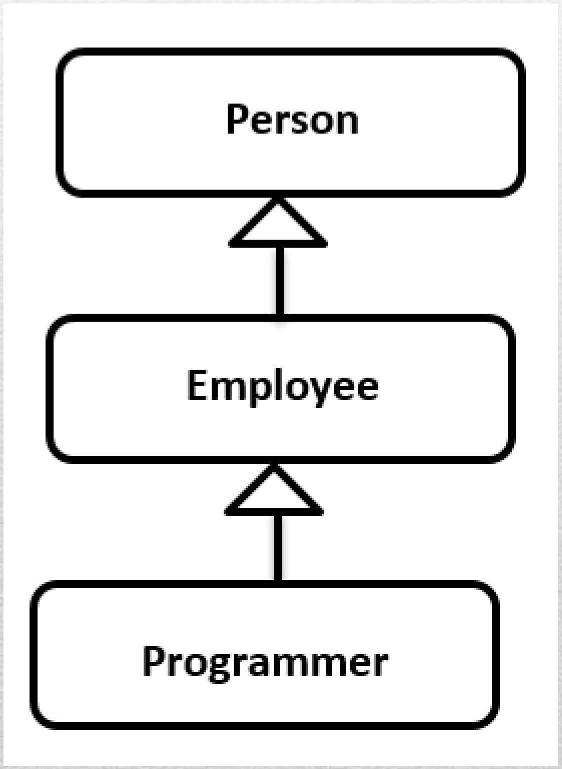


```
public class Shape
  {
     protected float width,height;

     public void SetData(float w,float h)
     {
        width = w;
        height = h;
     }
  }

  public class Rectangle : Shape
  {
     public float area()
     {
        return (width*height);
     }
  }

  public class Triangle : Shape
  {
     public float area()
     {
        return (width*height/2);
     }  }
```

```
public class Program
  {
     public static void Main(string[] args)
     {
        Rectangle rect = new Rectangle();
        Triangle tri = new Triangle();
        rect.SetData(5,3);
        tri.SetData(2,5);
        Console.WriteLine(rect.area() + "\n" +
tri.area());
     }
  }
```

TYPES OF INHERITANCE

MULTI LEVEL INHERITANCE



```
public class Person
{
   string name="Dummy",gender="M";
   int age=26;

  public void display()
  {
      Console.WriteLine("Name:  "+name);
      Console.WriteLine("Age:  "+age);
      Console.WriteLine("Gender:  "+gender);
  }
}
```

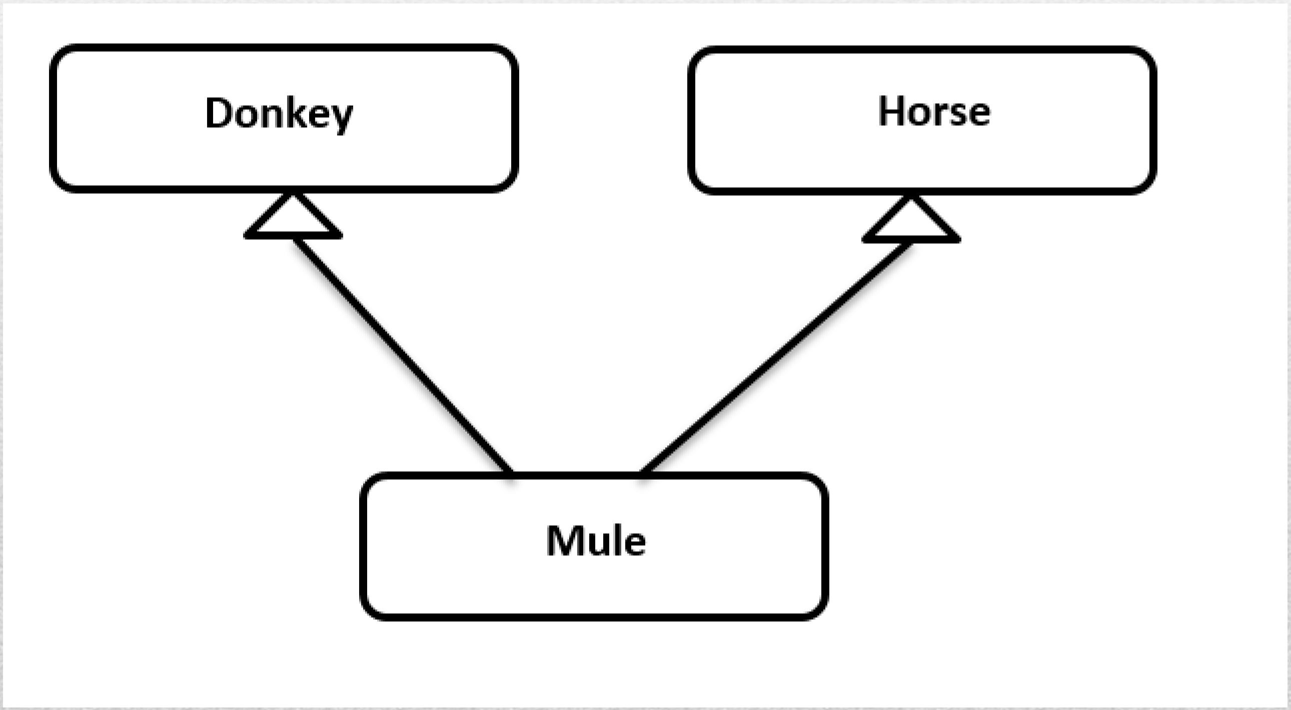
```
public class Employee:Person
{
   string company = "ABC";
   float salary = 25000f;

   public void display()
   {
      base.display();
      Console.WriteLine("Name of Company:
"+company);
      Console.WriteLine("Salary: Rs."+salary);
   }
}
```

```
public class Programmer: Employee
{
   int number=3;

   public void display()
   {
      base.display();
      Console.WriteLine("Number of programming
language known:"+number);
   }
}
```

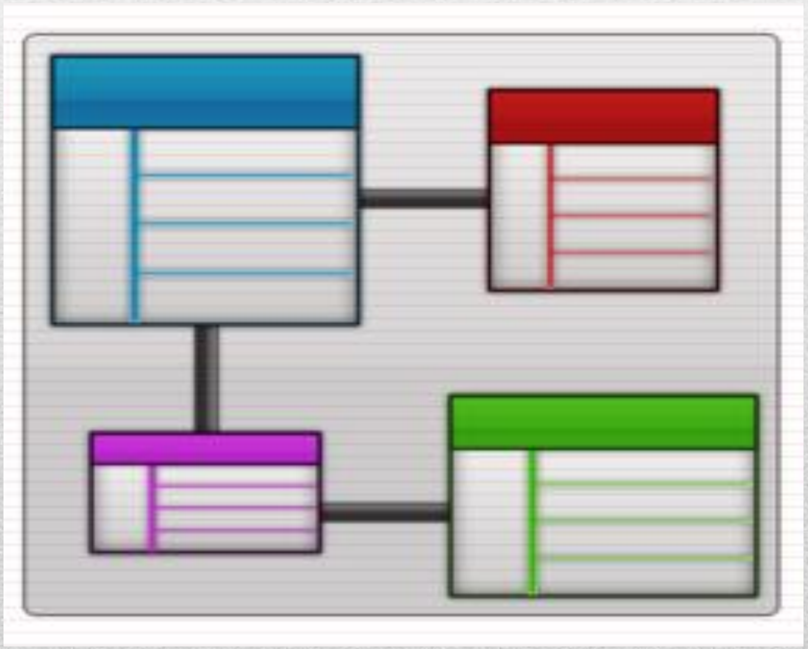
```
 public class Program
    {
        public static void Main(string[] args)
        {
                Programmer p = new Programmer();
                Console.WriteLine("Enter data");
                Console.WriteLine("\nDisplaying data");
                p.display();
          }
    }
```

TYPES OF INHERITANCE

MULTI PLE INHERITANCE



WEEK#6
OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

CONSTRUCTOR CALLING IN INHERITANCE

Each subclass can have its own constructor for specialised initialization but it must often invoke the behaviour of the base constructor. It does this using the keyword **base.**

```
class MySubClass : MySuperClass
{
    public MySubClass (sub-parameters) : base(super-parameters)
    {
        // other initialization
    }
```

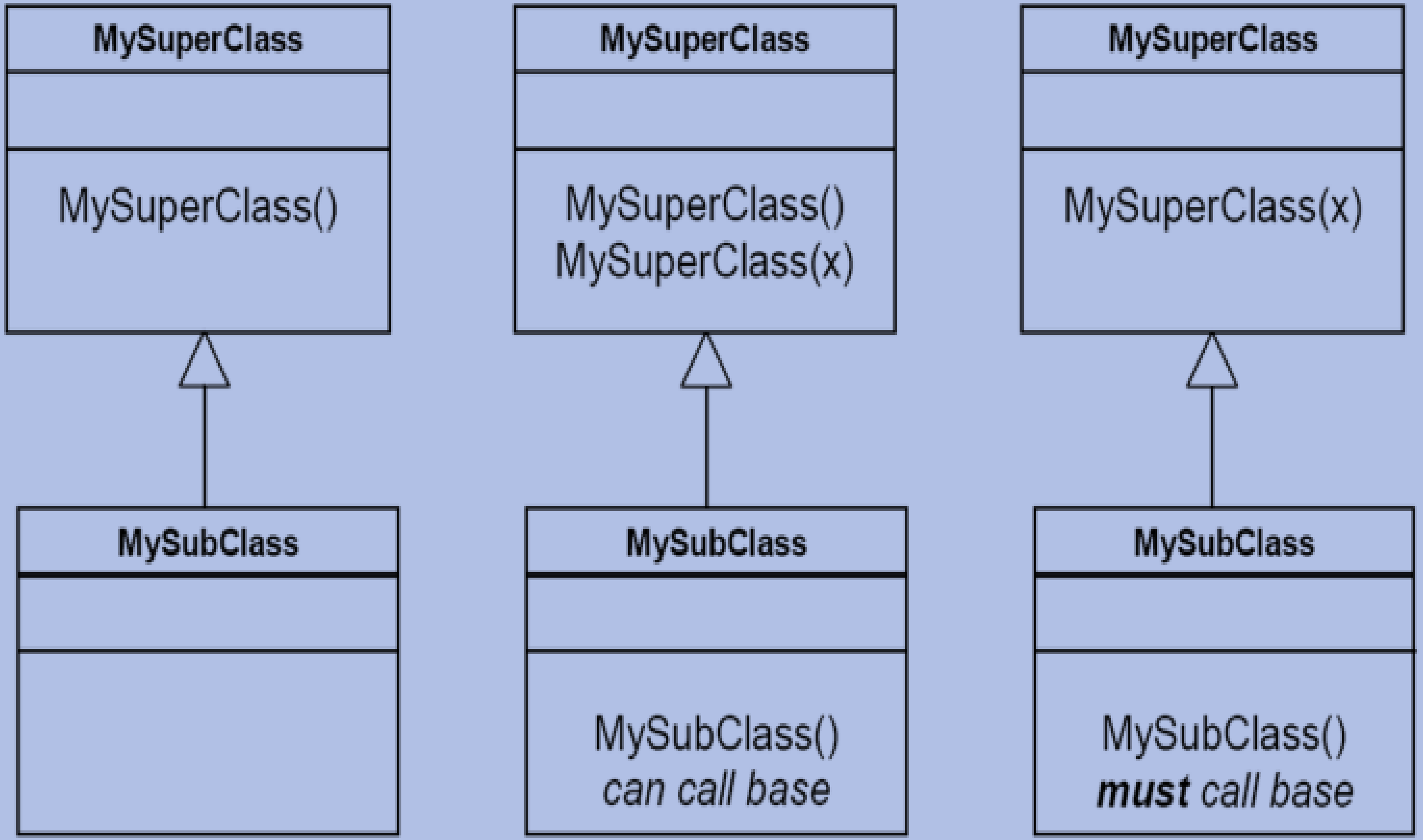
Usually some of the parameters passed to MySubClass will be initializer values for superclass instance variables, and these will simply be passed on to the superclass constructor as parameters. In other words *super-parameters* will be some (or all) of *sub-parameters*.

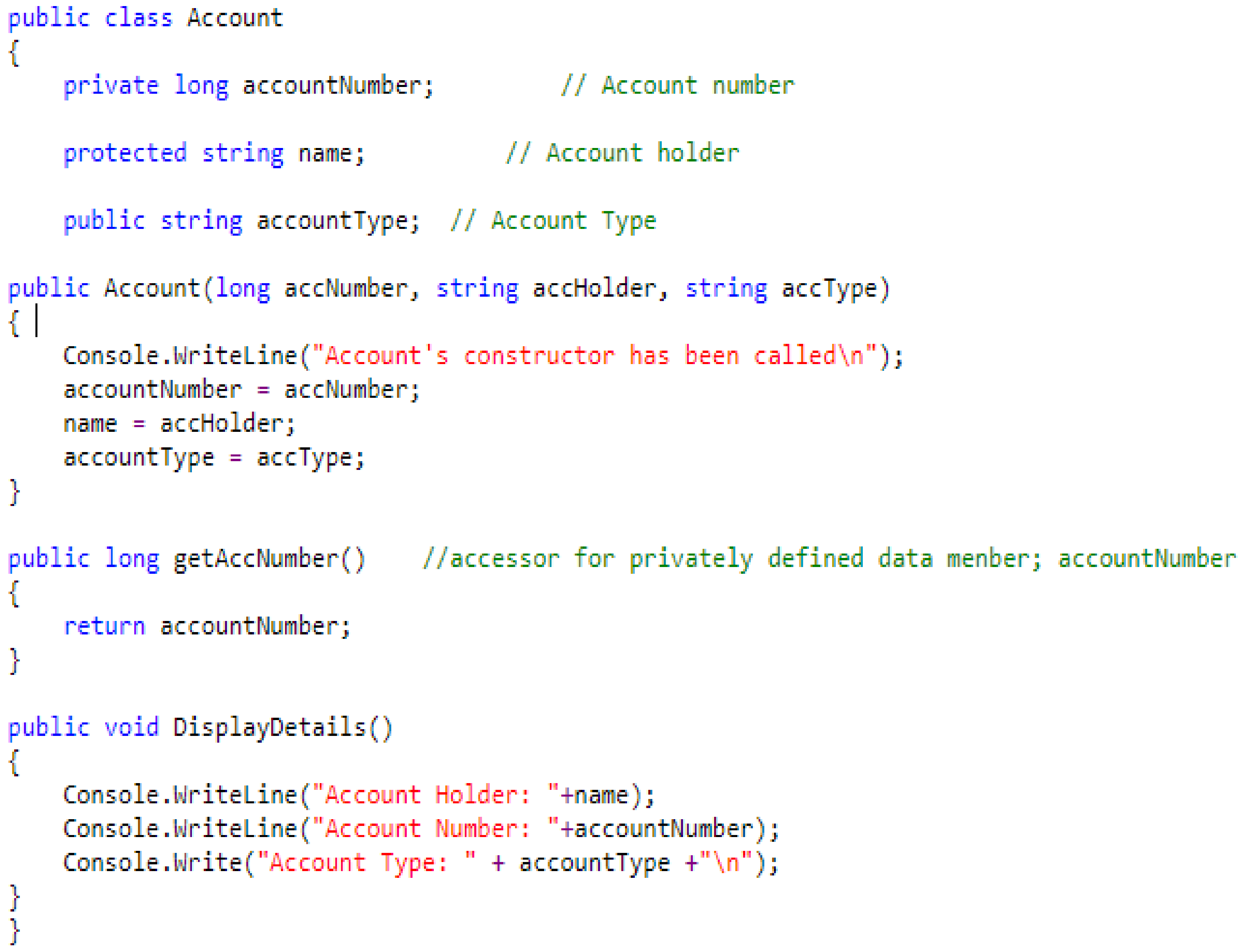
CONSTRUCTOR CALLING IN INHERITANCE

```
// a constructor for the Publication class
public Publication(String title, double price, int copies)
{
    this.title = title;
    // etc
}
```

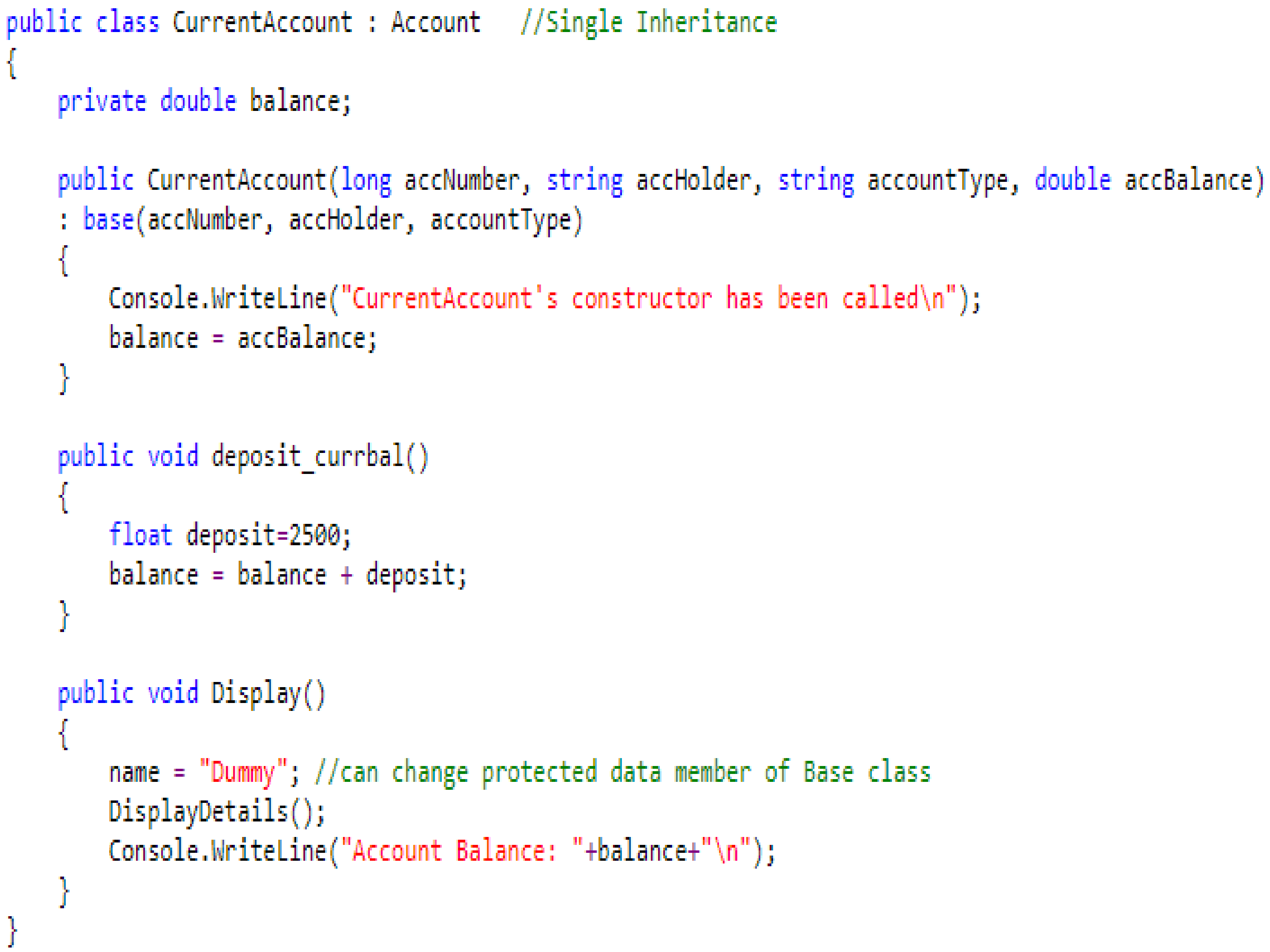
```
// a constructor for the Book class
public Book(String title, double price, int copies, String author)
        : base(title, price, copies)
{
    this.author = author;
}
```

CONSTRUCTOR RULES

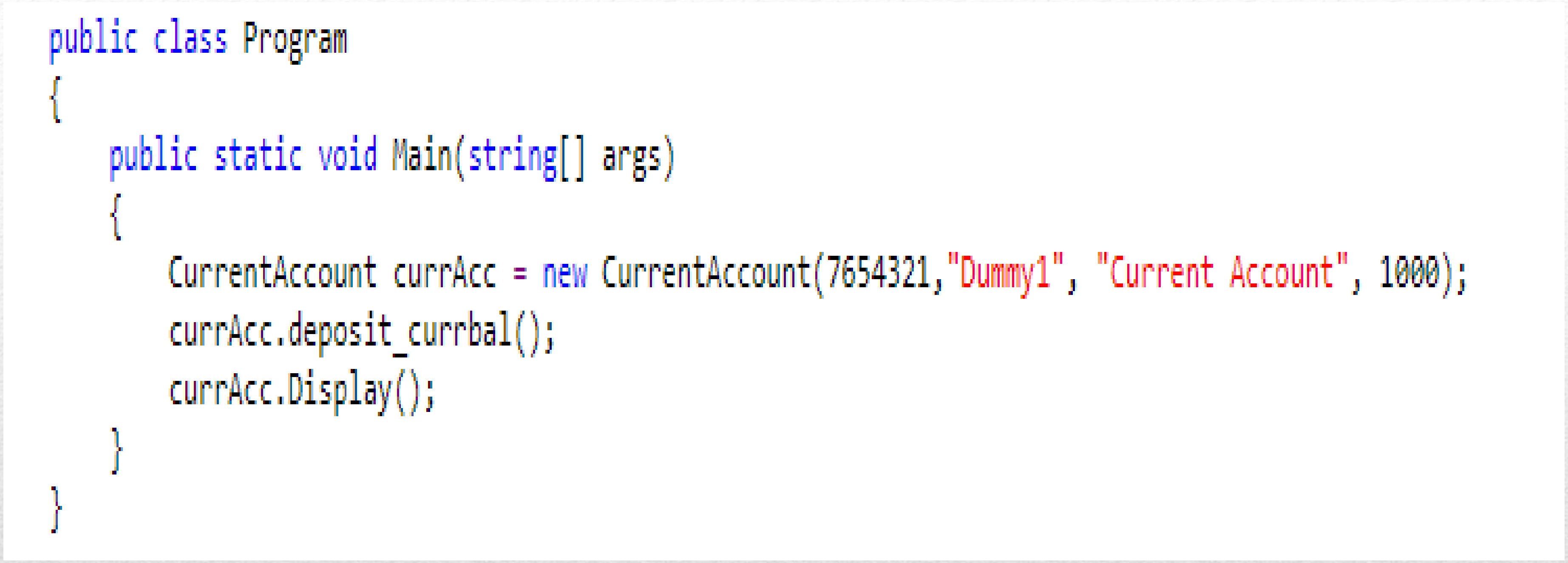




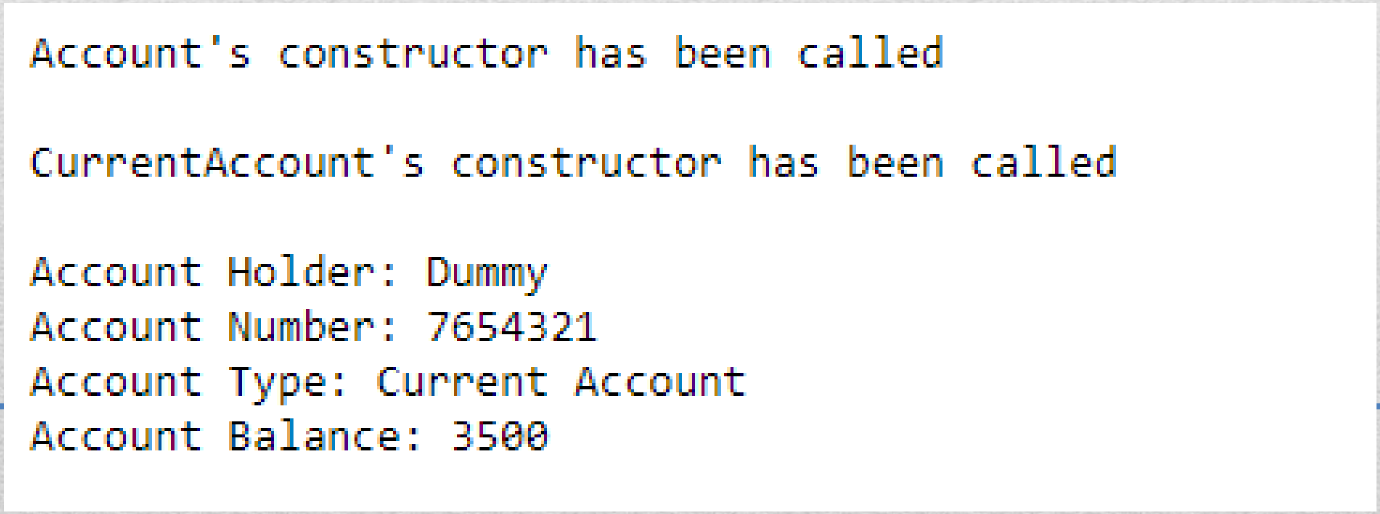
```
public class Account
{
    private long accountNumber;         // Account number

    protected string name;          // Account holder

    public string accountType;  // Account Type

public Account(long accNumber, string accHolder, string accType)
{
    Console.WriteLine("Account's constructor has been called\n");
    accountNumber = accNumber;
    name = accHolder;
    accountType = accType;
}

public long getAccNumber()    //accessor for privately defined data menber; accountNumber
{
    return accountNumber;
}

public void DisplayDetails()
{
    Console.WriteLine("Account Holder: "+name);
    Console.WriteLine("Account Number: "+accountNumber);
    Console.Write("Account Type: " + accountType +"\n");
}
}
```



```
public class CurrentAccount : Account   //Single Inheritance
{
    private double balance;

    public CurrentAccount(long accNumber, string accHolder, string accountType, double accBalance)
    : base(accNumber, accHolder, accountType)
    {
        Console.WriteLine("CurrentAccount's constructor has been called\n");
        balance = accBalance;
    }

    public void deposit_currbal()
    {
        float deposit=2500;
        balance = balance + deposit;
    }

    public void Display()
    {
        name = "Dummy"; //can change protected data member of Base class
        DisplayDetails();
        Console.WriteLine("Account Balance: "+balance+"\n");
    }
}
```



```
public class Program
{
    public static void Main(string[] args)
    {
        CurrentAccount currAcc = new CurrentAccount(7654321,"Dummy1", "Current Account", 1000);
        currAcc.deposit_currbal();
        currAcc.Display();
    }
}
```



```
Account's constructor has been called

CurrentAccount's constructor has been called

Account Holder: Dummy
Account Number: 7654321
Account Type: Current Account
Account Balance: 3500
```

METHOD OVERRIDING

```
public class Person
  {
    private string name = "Dummy";
    private int age = 26;

    public void Display()
    {
      Console.Write(name + "\n" + age);
    }
  }

  public class Employee : Person
  {
    public string company = "ABC Company";
    public float salary = 36000;

    public void Display()
    {
      base.Display();
      Console.Write("\n" + company + "\n" + salary + "\n");
    }
  }
```

```
public class Program
  {
    public static void Main(string[] args)
    {
      Employee emp = new Employee();
      emp.Display();
      Person pers = new Person();
      pers.Display();
    }
  }
```

const KEYWORD

- a const field contains a compile-time-determined value that cannot be changed at runtime.
- Values such as pi, Avogadro’s number, and the circumference of the Earth are good examples.
- Constant fields are static automatically, since no new field instance is required for each object instance.
- Declaring a constant field as static explicitly will cause a compile error.

const KEYWORD

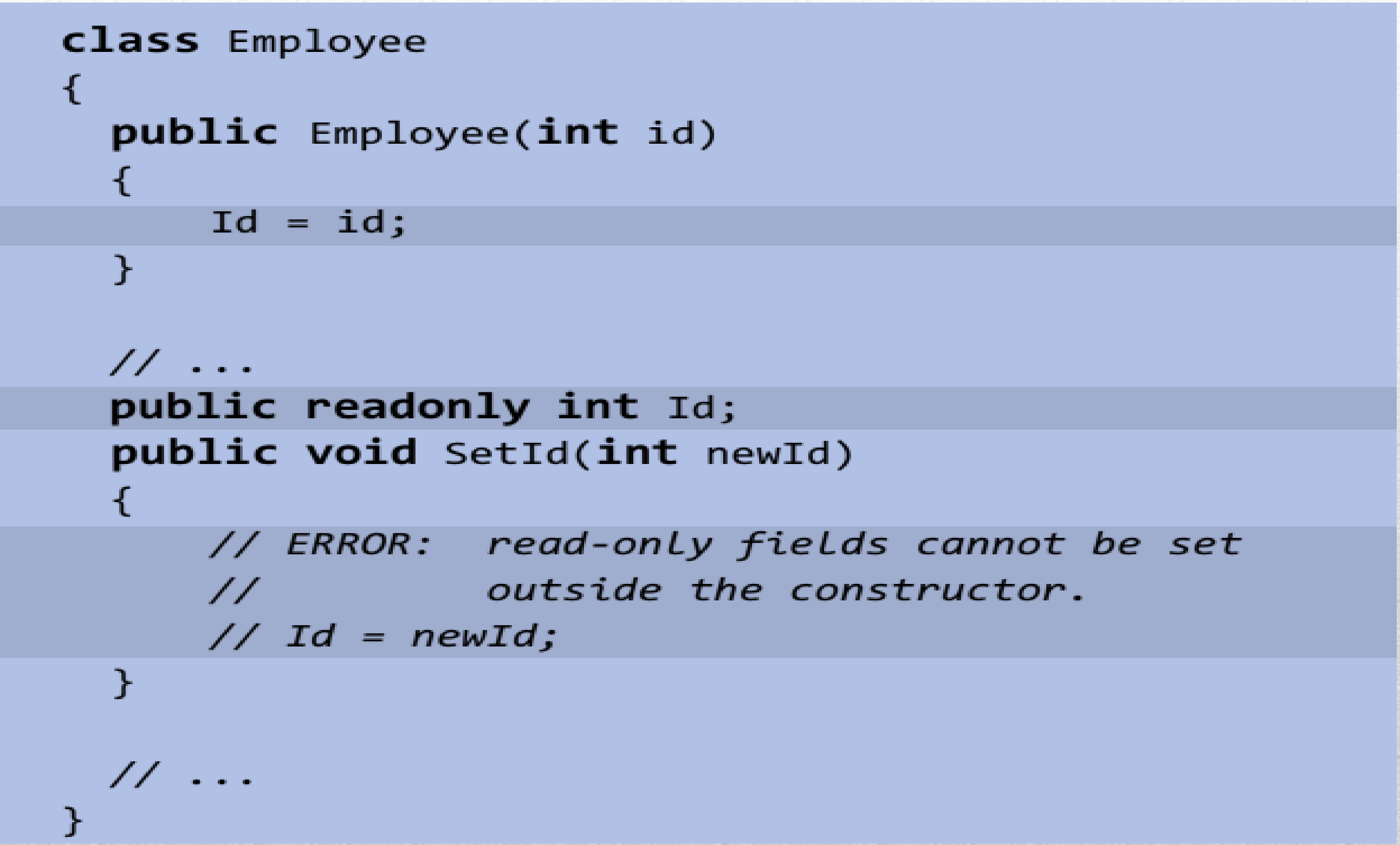
```
class ConvertUnits
{
   public const float CentimetersPerInch = 2.54F;
   public const int CupsPerGallon = 16;
   // ...
}
```

readonly KEYWORD

- it declares that the field value is modifiable only from inside the constructor or directly during declaration.

- Unlike constant fields, readonly fields can vary from one instance to the next.

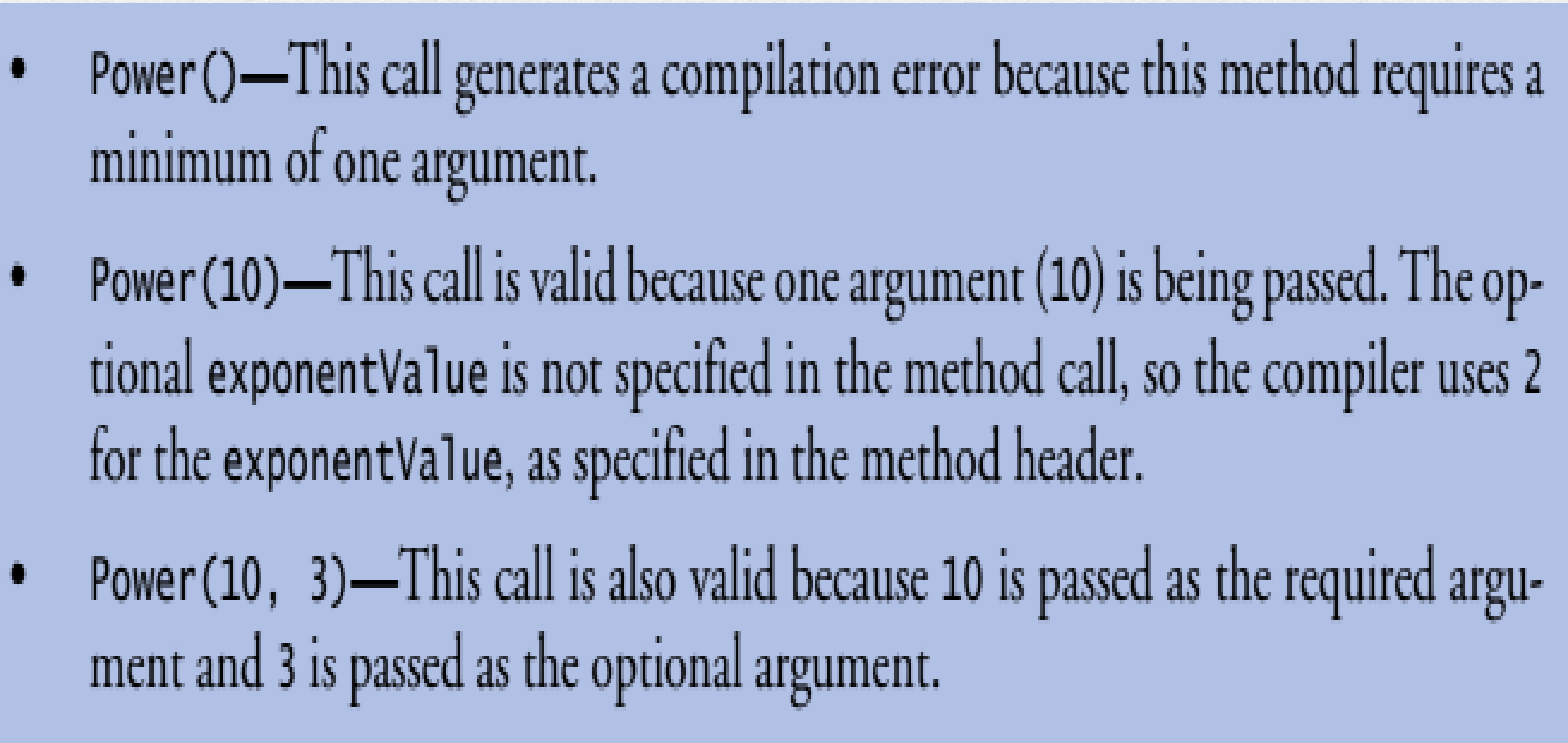
readonly KEYWORD



```
class Employee
{
  public Employee(int id)
  {
      Id = id;
  }

  // ...
  public readonly int Id;
  public void SetId(int newId)
  {
      // ERROR:  read-only fields cannot be set
      //         outside the constructor.
      // Id = newId;
  }

  // ...
}
```

OPTIONAL PARAMETERS

static int Power(int baseValue, int exponentValue = 2)

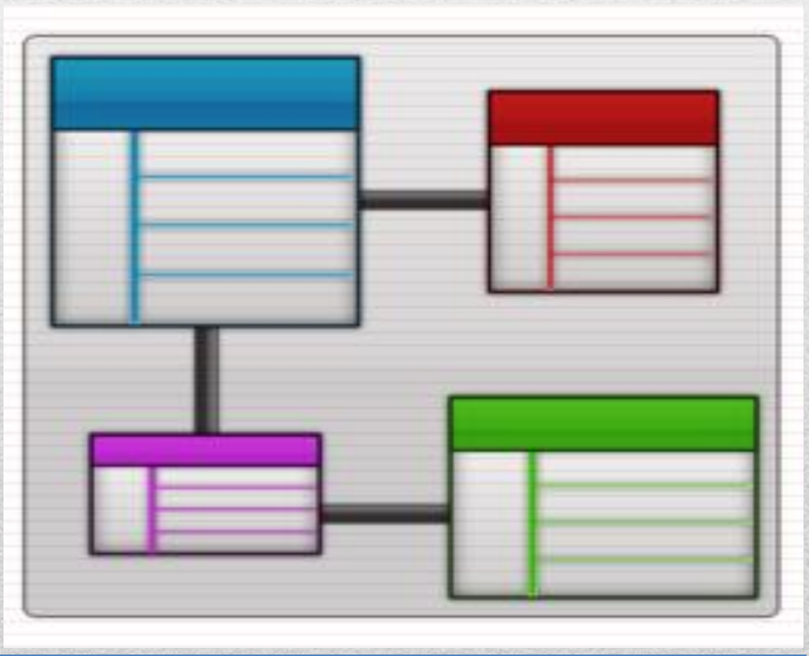


- `Power()`—This call generates a compilation error because this method requires a minimum of one argument.
- `Power(10)`—This call is valid because one argument (10) is being passed. The optional `exponentValue` is not specified in the method call, so the compiler uses 2 for the `exponentValue`, as specified in the method header.
- `Power(10, 3)`—This call is also valid because 10 is passed as the required argument and 3 is passed as the optional argument.

NAMED PARAMETERS

```
public void SetTime(int hour = 0, int minute = 0, int second = 0)

t.SetTime(12, , 22); // COMPILATION ERROR

t.SetTime(hour: 12, second: 22); // sets the time to 12:00:22
```

WEEK#7 OBJECT ORIENTED PROGRAMMING



BY

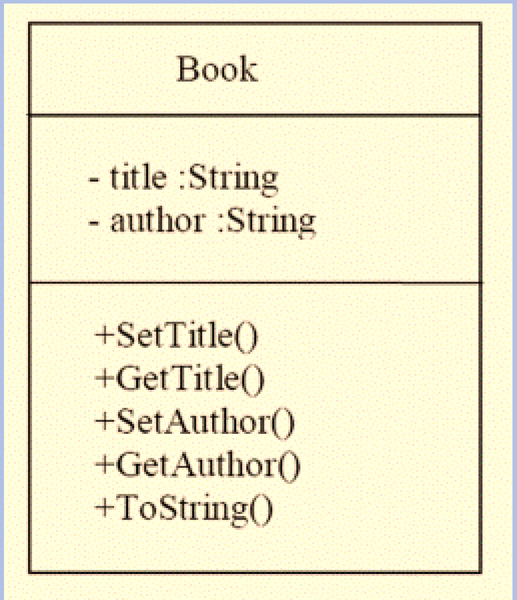
SOLAT JABEEN

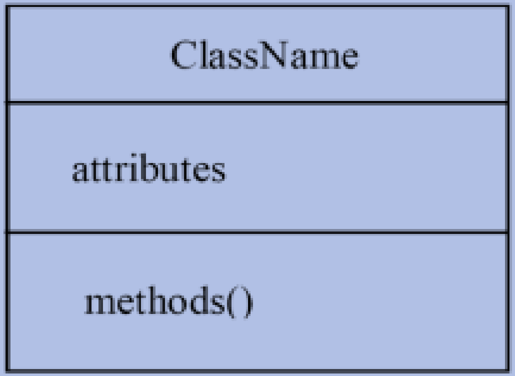
UML CLASS DIAGRAM

A class consists of :-

- a unique name (conventionally starting with an uppercase letter)
- a list of attributes (int, double, boolean, String etc)
- a list of methods

This is shown in a simple box structure…

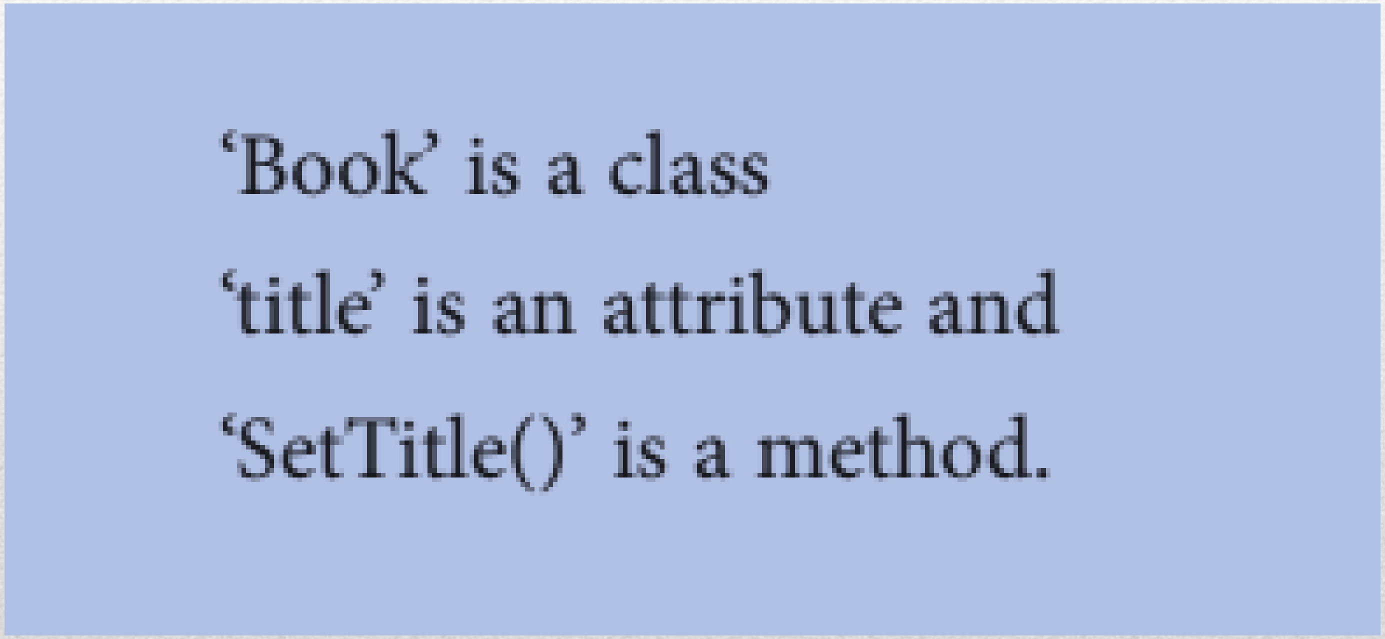




CLASS DIAGRAM EXAMPLE

It’s for an imaginary application that must model different kinds of vehicles such as bicycles, motor bike and cars. All Vehicles have some common attributes (speed and colour) and common behaviour (TurnLeft, TurnRight). Bicycle and MotorVehicle are both kinds of Vehicle. MotorVehicles have engines and license plates. MotorBike and Car are MotorVehicles. Both MotorBike and Car have additional attributes and behaviour which are specific to those kinds of object

NAMING CONVENTIONS



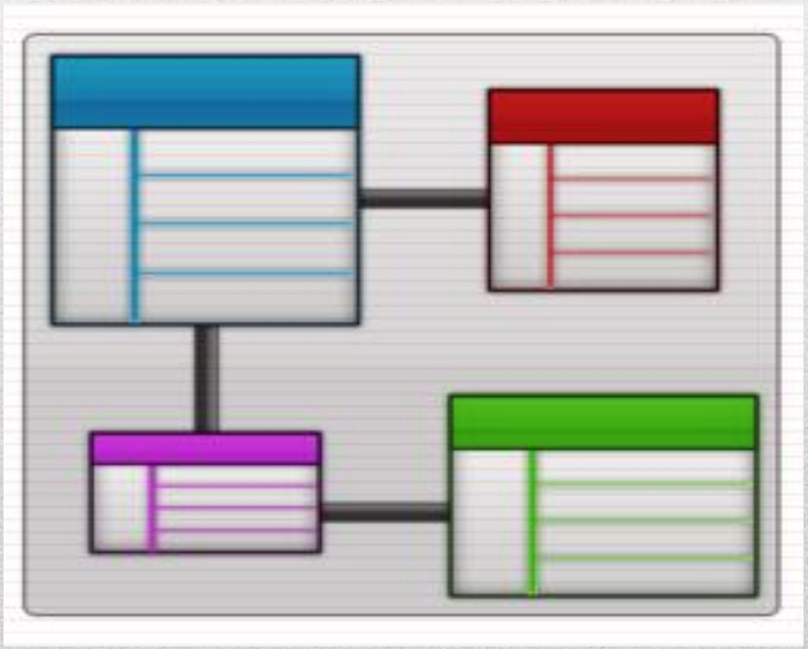
'Book' is a class

'title' is an attribute and

'SetTitle()' is a method.

Everything covered till this slide along with the assignment, quizzes, class discussions and book’s content is relevant for Midterm Exam

WEEK#8 OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

Association

- **Is the weakest link between objects**
- **Is a reference by which one object can interact with some other object**
- **Association can be one-to-one, one-to-many, many-to-one, many-to-many.**

Kinds of Association

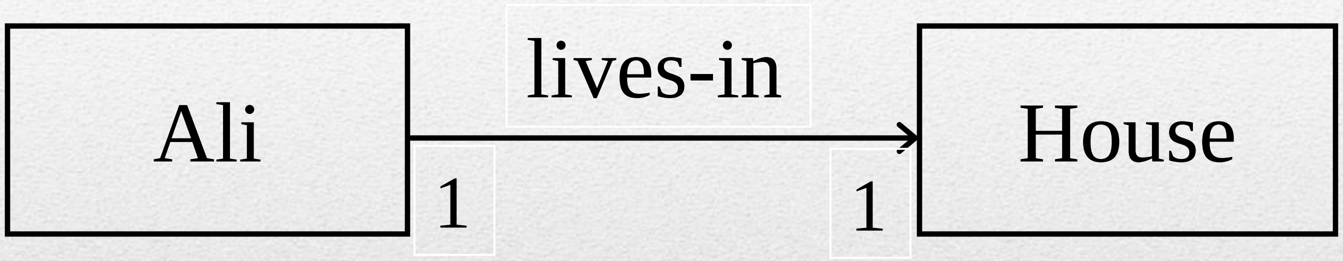
- **w.r.t navigation**
  - **One-way Association**
  - **Two-way Association**

- **w.r.t number of objects**
  - **Binary Association**
  - **Ternary Association**
  - **N-ary Association**

One-way Association

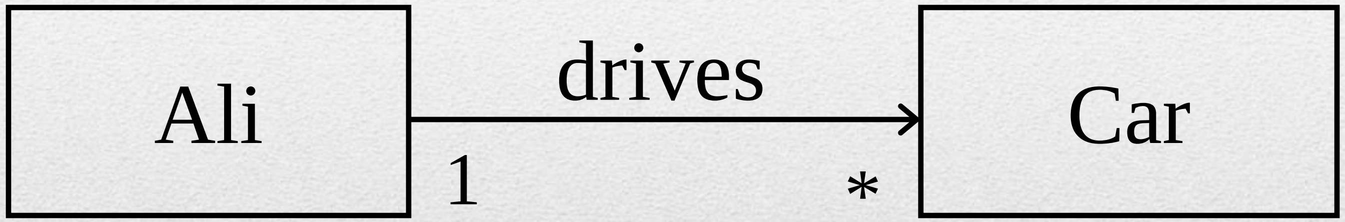
- **We can navigate along a single direction only**
- **Denoted by an arrow towards the server object**

Example – Association



Ali lives in a House

Example – Association

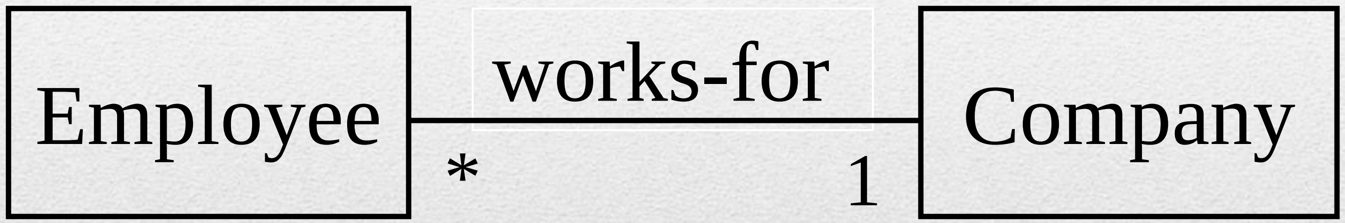


Ali drives his Car

Two-way Association

- We can navigate in both directions
- Denoted by a line between the associated objects

Two-way Association

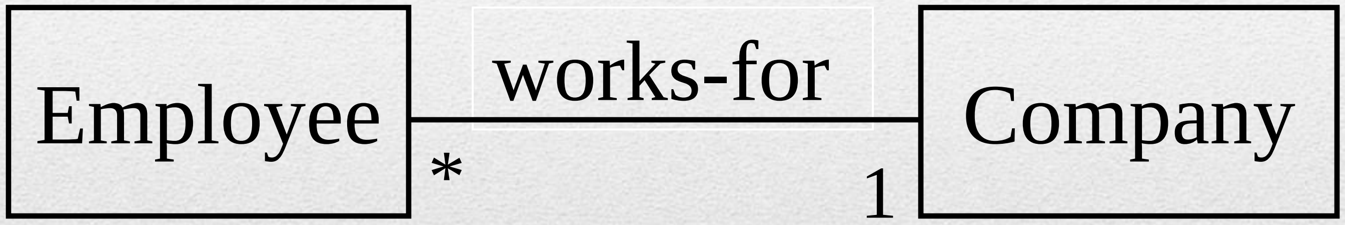


- **Employee works for company**
- **Company employs employees**

Binary Association

- Associates objects of exactly two classes
- Denoted by a line, or an arrow between the associated objects

Example – Binary Association



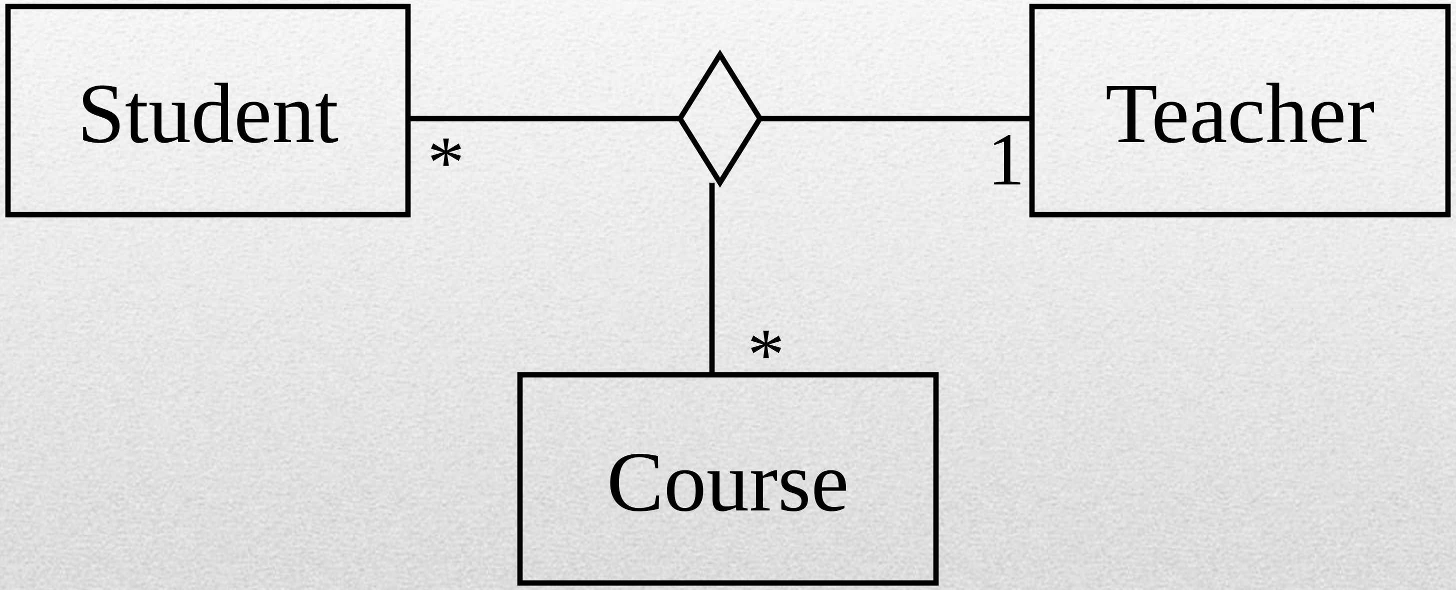
- **Association “works-for” associates objects of exactly two classes**

Ternary Association

- Associates objects of exactly three classes

- Denoted by a diamond with lines connected to associated objects

Example – Ternary Association



- **Objects of exactly three classes are associated**

N-ary Association

- An association between 3 or more classes
- Practical examples are very rare

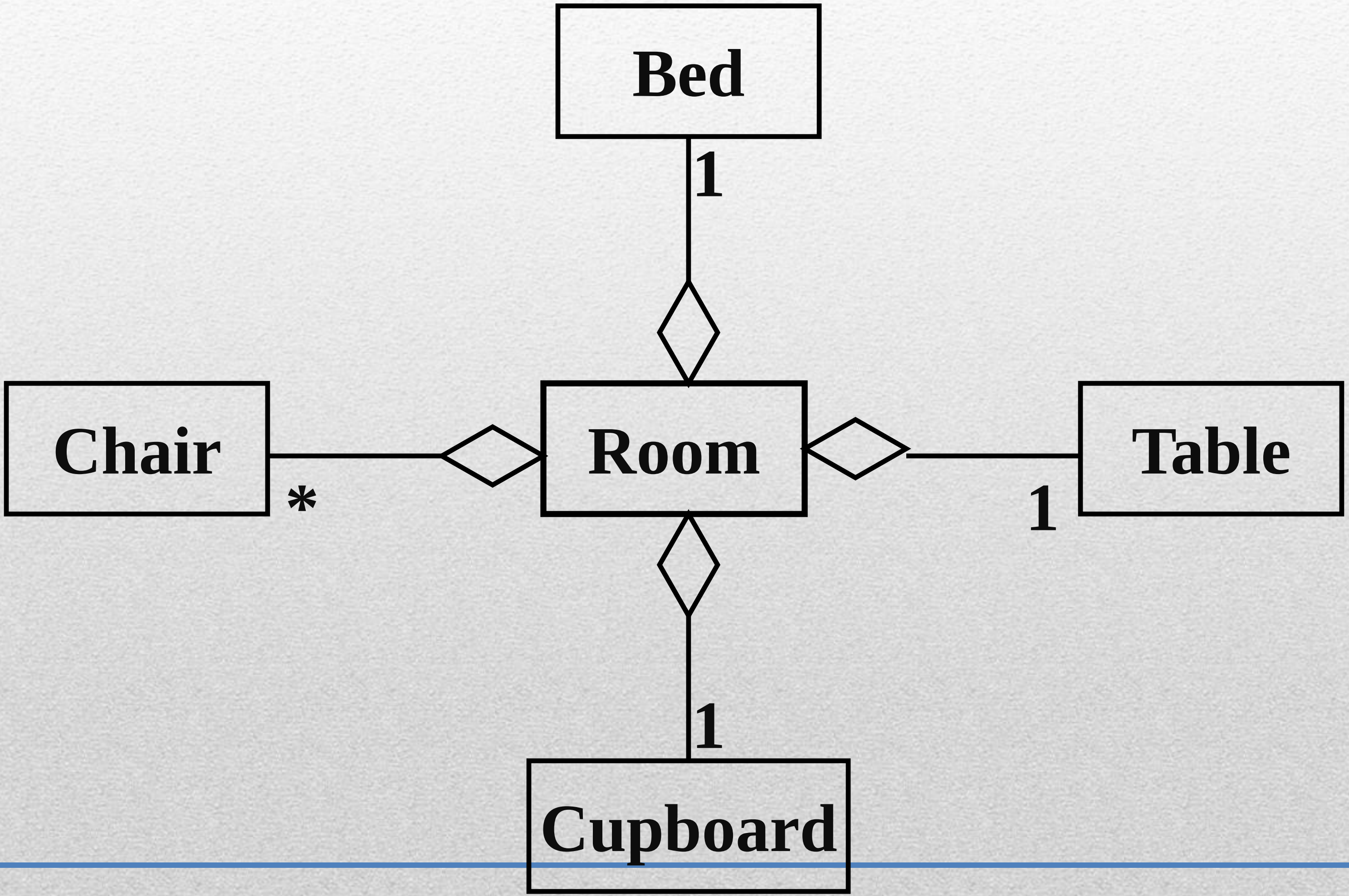
Forms of Association

- **Composition** and **Aggregation** are the two forms of association.

Aggregation

- **Aggregation is a special type of Association.**
- **Aggregation models a whole part relationship between an aggregate (the whole) and its parts.**
- **When an aggregate is destroyed, the sub objects are not destroyed.**

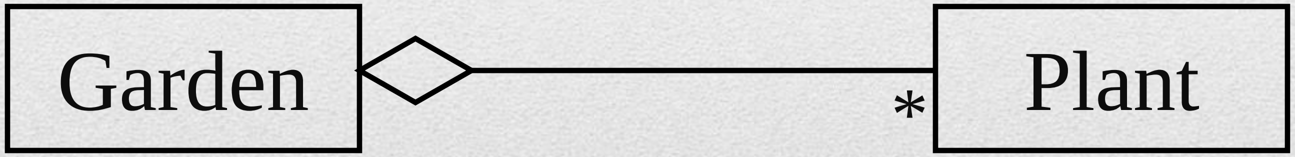
Example – Aggregation



Example – Aggregation

- **Furniture is not an intrinsic part of room**
- **Furniture can be shifted to another room, and so can exist independent of a particular room**

Example – Aggregation

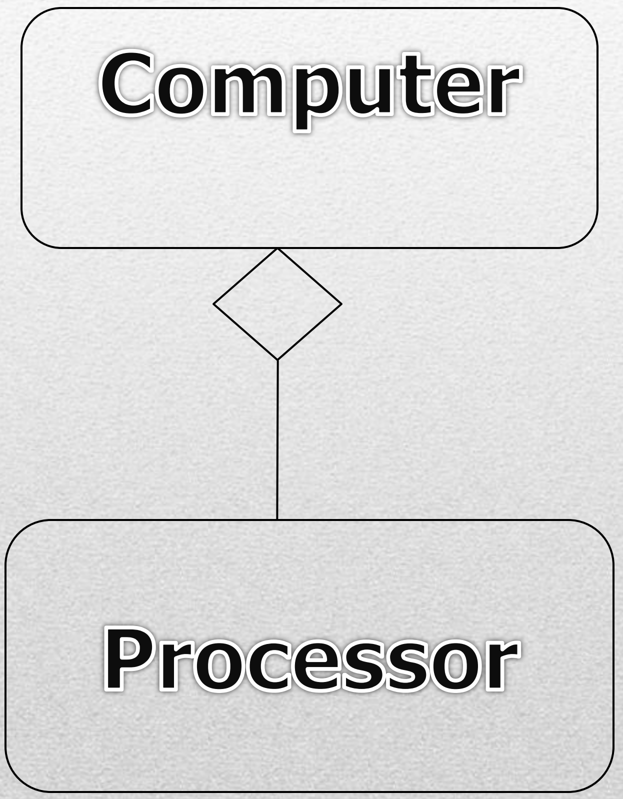


Example – Aggregation

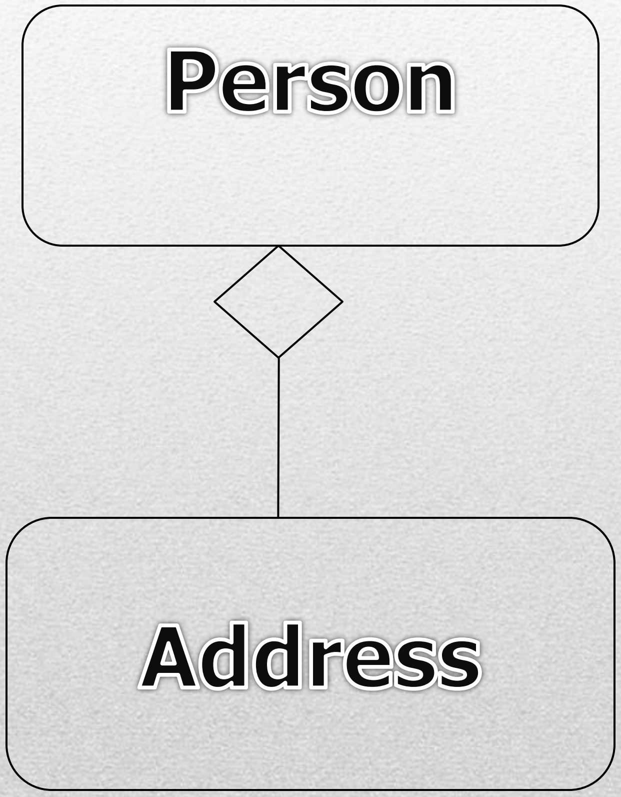
- A plant is not an intrinsic part of a garden

- It can be planted in some other garden, and so can exist independent of a particular garden

Example – Aggregation



Example – Aggregation



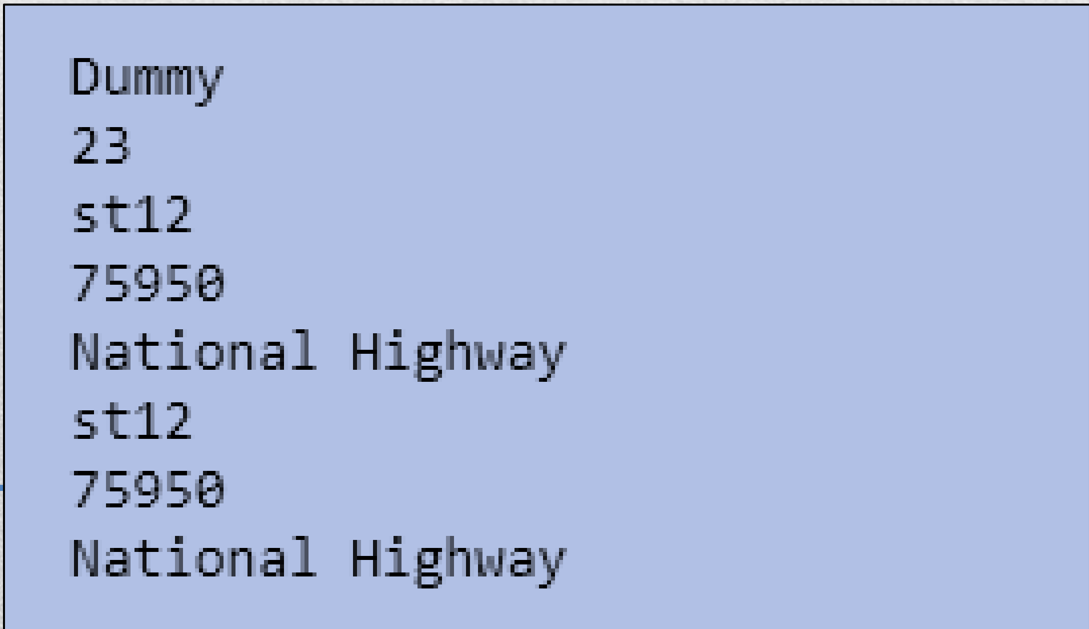
Aggregation - Implementation

```
public class Address
   {
       private string Street;
       private long postalCode;
       private string Area;

      public Address(string str, long p, string a)
      {
           Street = str;
           postalCode = p;
           Area = a;
      }

      public void Show()
      {
           Console.Write(Street+"\n"+postalCode+"\n"+Area+"\n");
      }
   }
```

```
public class Person
 {
    string Name;

    int Age;
    private Address address;

    public Person(string n, Address a, int age)
    {
        Name = n;
        Age = age;
        address = a;
    }

    public void Show()
    {
      Console.Write(Name+"\n"+Age+"\n");
      address.Show();
    }
 }
```

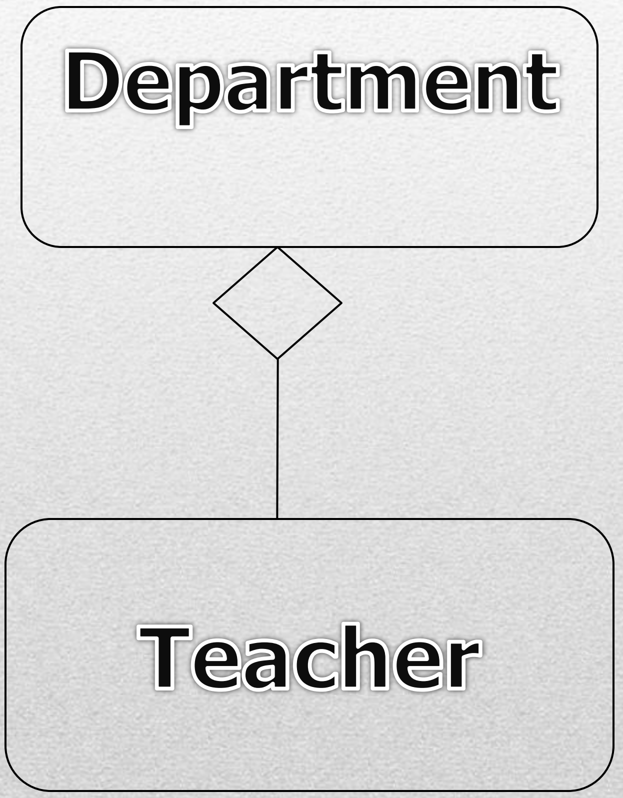
Aggregation - Implementation

```
public class Program
    {
        public static void Main(string[] args)
        {
            Address address = new Address("st12",75950,"National Highway");
            {
                Person person = new Person("Dummy",address,23);
                person.Show();
            }
           address.Show();
        }
    }
```



```
Dummy
23
st12
75950
National Highway
st12
75950
National Highway
```

Example – Aggregation



Provide the implementation for above example.

Composition

- Composition is special type of Aggregation. It is a strong type of Aggregation.
- An object may be composed of other smaller objects
- The relationship between the “part” objects and the “whole” object is known as Composition

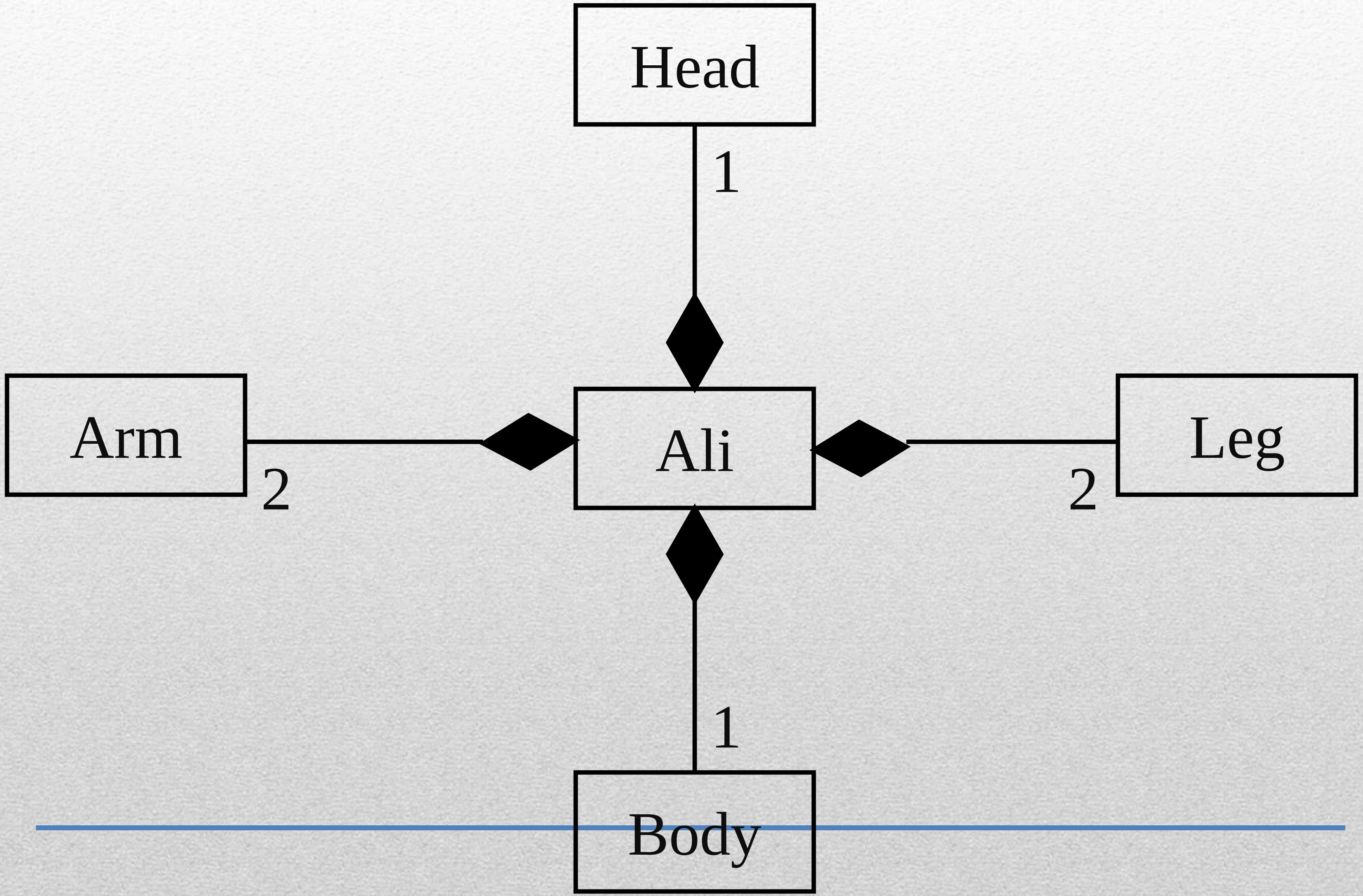
Composition

- Composition is a stronger relationship, because
  - Composed object becomes a part of the composer
  - Composed object can’t exist independently

- Composition is a "Part of" relationship.

- Composition is represented by a line with a filled-diamond head towards the composer object

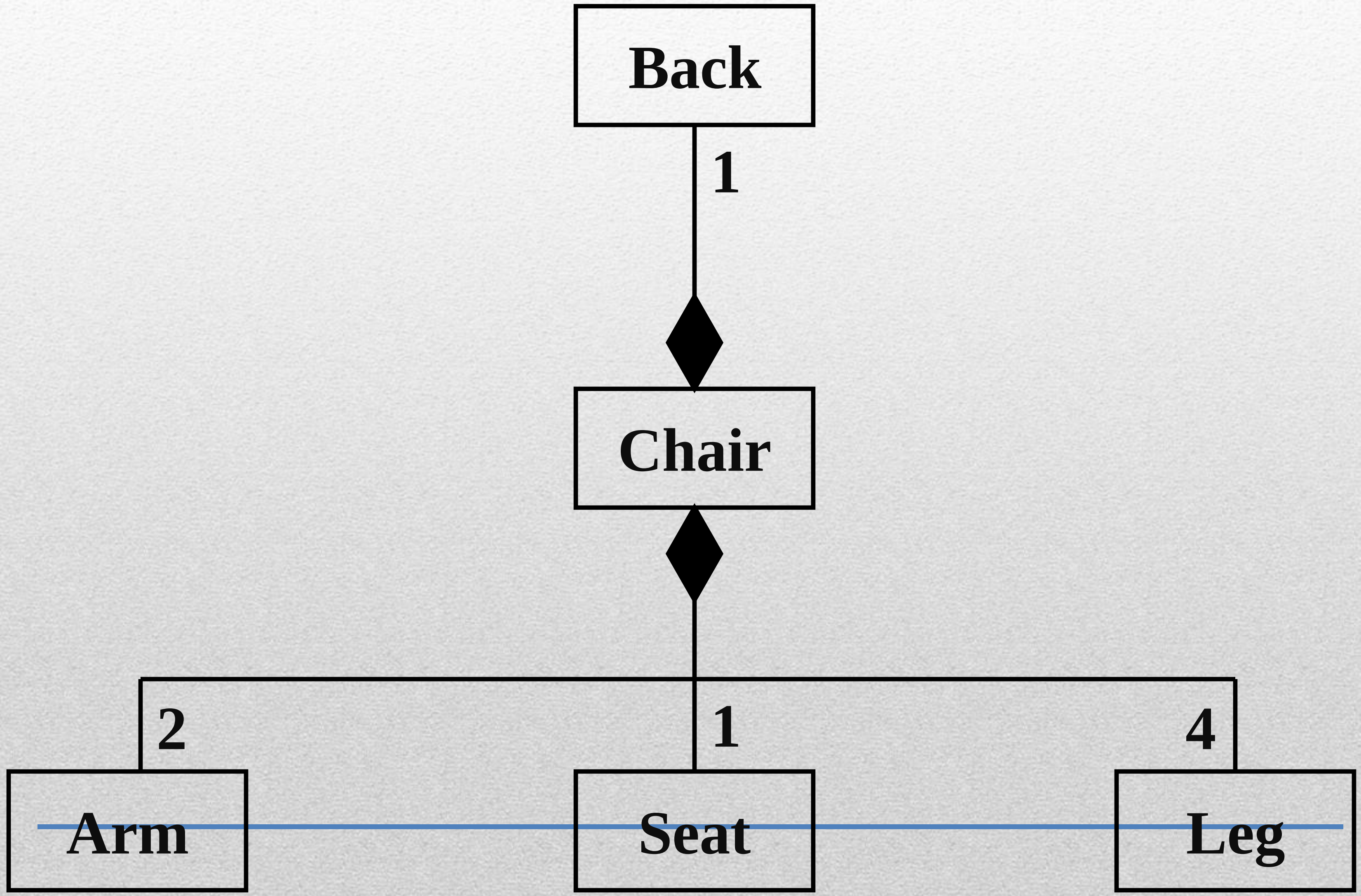
Example – Composition



Example – Composition

- **Ali is made up of different body parts**
- **They can’t exist independent of Ali**

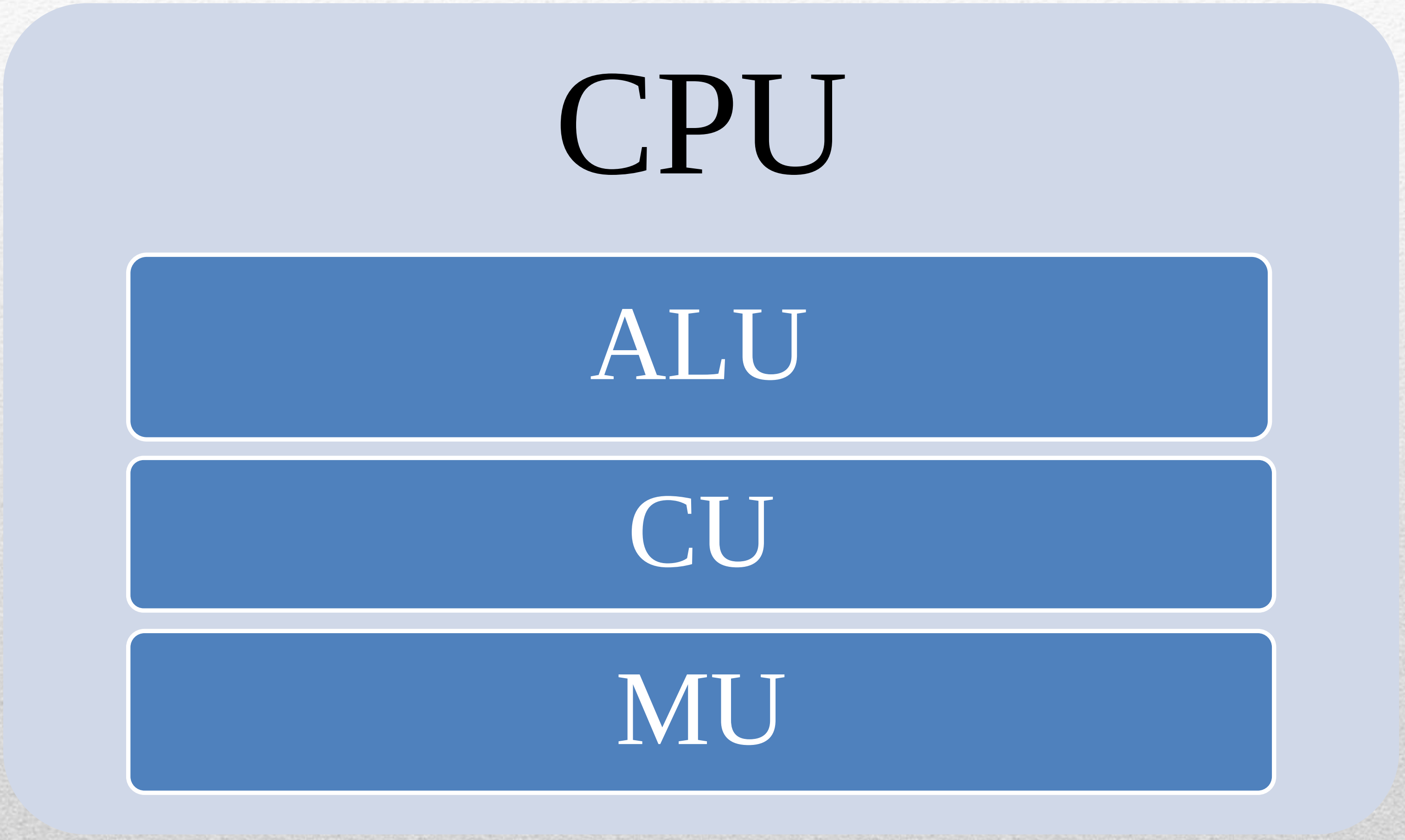
Example – Composition



Example – Composition

- Chair’s body is made up of different parts
- They can’t exist independently

Example – Composition



Composition - Implementation

```
public class ControlUnit
 {

    public ControlUnit()
    {
        Console.Write("Ctor: Control Unit\n");
    }

    public void PrintUnitName()
    {
        Console.Write("Control Unit\n");
    }
 }
```

Composition - Implementation

```
public class ArithmeticAndLogicUnit
  {

      public ArithmeticAndLogicUnit()
      {
          Console.Write("Ctor: Arithmetic & Logic Unit\n");
      }

      public void PrintUnitName()
      {
          Console.Write("Arithmetic & Logic Unit\n");
      }
  }
```

Composition - Implementation

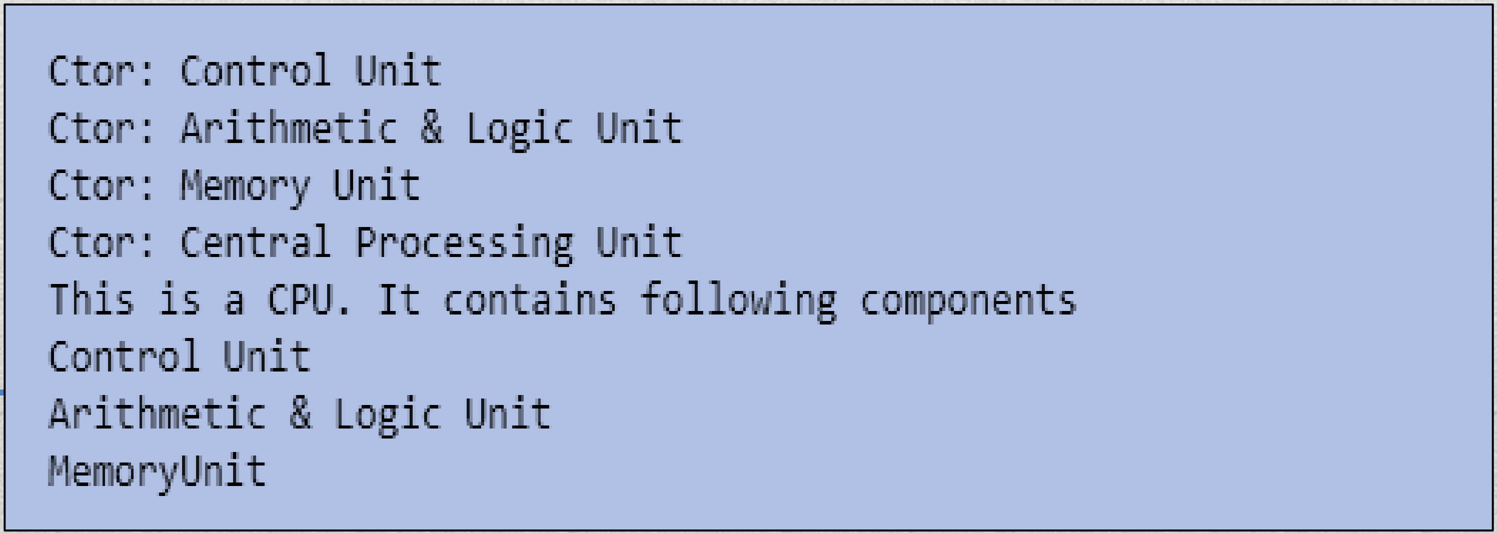
```
public class MemoryUnit
  {

     public MemoryUnit()
     {
          Console.Write("Ctor: Memory Unit\n");
     }

     public void PrintUnitName()
     {
          Console.Write("MemoryUnit\n");
     }
  }
```

Composition - Implementation

```
public class  CentralProcessingUnit
   {
        ControlUnit cu = new  ControlUnit();
        ArithmeticAndLogicUnit alu = new  ArithmeticAndLogicUnit();
        MemoryUnit mu = new MemoryUnit();

       public  CentralProcessingUnit()
       {
          Console.Write("Ctor: Central Processing Unit\n");
       }

       public void PrintSpec()
       {
         Console.Write("This is a CPU. It contains following components\n");
          cu.PrintUnitName();
          alu.PrintUnitName();
          mu.PrintUnitName();
       }
   }
```

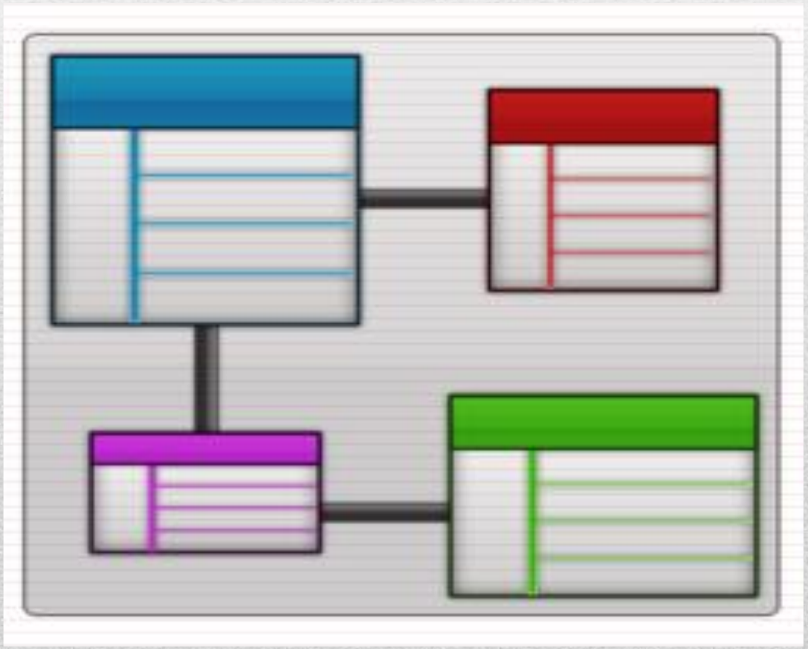
Composition - Implementation

```
public class Program
   {
      public static void Main(string[] args)
      {
         CentralProcessingUnit cpu = new  CentralProcessingUnit();
         cpu.PrintSpec();
      }
   }
```



```
Ctor: Control Unit
Ctor: Arithmetic & Logic Unit
Ctor: Memory Unit
Ctor: Central Processing Unit
This is a CPU. It contains following components
Control Unit
Arithmetic & Logic Unit
MemoryUnit
```

WEEK#9 OBJECT ORIENTED PROGRAMMING



BY

SOLAT JABEEN

POLYMORPHISM

- **Poly means many , morph means form**
  Polymorphic means having many forms

Hence Polymorphism is:

- **Ability for the same code to be used with different types of objects and behave differently with each.**
- **The same invocation can produce “many forms” of results**

TYPES OF POLYMORPHISM

- **Compile time polymorphism/Overloading**
- **Runtime polymorphism/Overriding**

Compile time polymorphism/Overloading

- **It is also called early binding.**
- **Method overloading is used in situation where we want a class to be able to do something, but there is more than one possibility for what information is supplied to the method that carries out the task.**
- **Consider overloading a method when for some reason a couple of methods are needed that take different parameters, but conceptually do the same thing.**
- **Overloading a method simply involves having another method with the same prototype.**
- **TYPES OF OVERLOADING**
    - **Constructor Overloading – Already discussed.**
    - **Function/Method Overloading – Already discussed.**
    - **Operator Overloading – to be discussed later.**

OPERATOR OVERLOADING

- **An operator is said to be overloaded if it is defined for multiple types. In other words, overloading an operator means making the operator significant for a new type.**

BUILT IN OVERLOADS

- Most operators are already overloaded for fundamental types.

- Example:
  In the case of the expression:
  a / b
  the operand type determines the machine code created by the compiler for the division operator. If both operands are integral types, an integral division is performed; in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

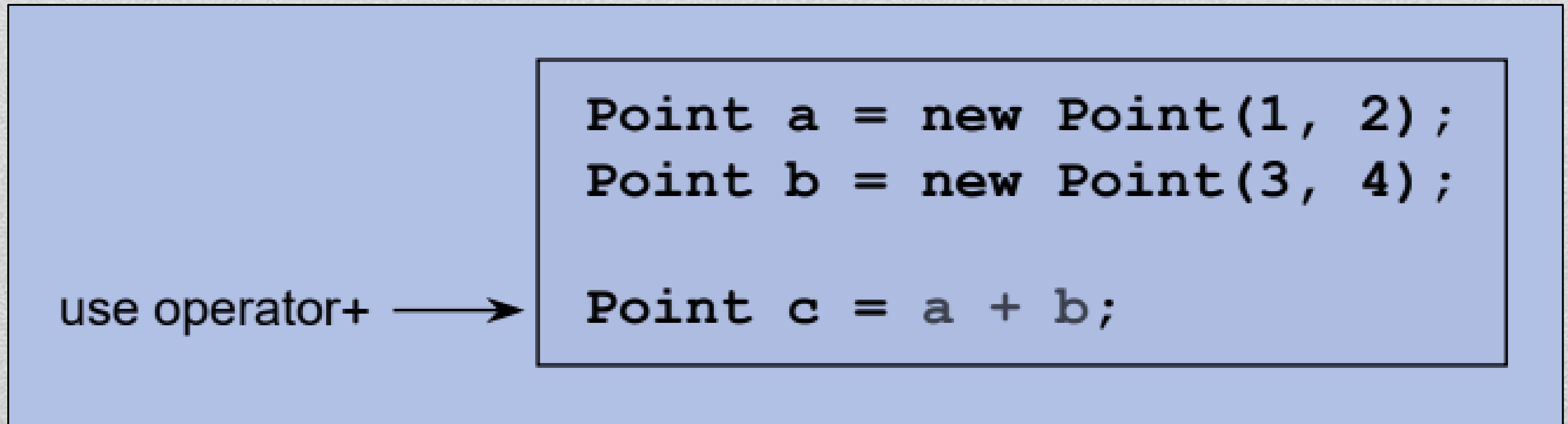
OVERLOADS FOR USER DEFINED TYPES

- Operators can be used with user-defined types as well.

- Although C# does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

- **Example:**

  The effect of + operator can be stipulated for the objects of a particular class.



```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

Point c = a + b;
```

Method format

- Overloaded operator must be member of class or struct

- Must have specific modifiers
  - public
  - static

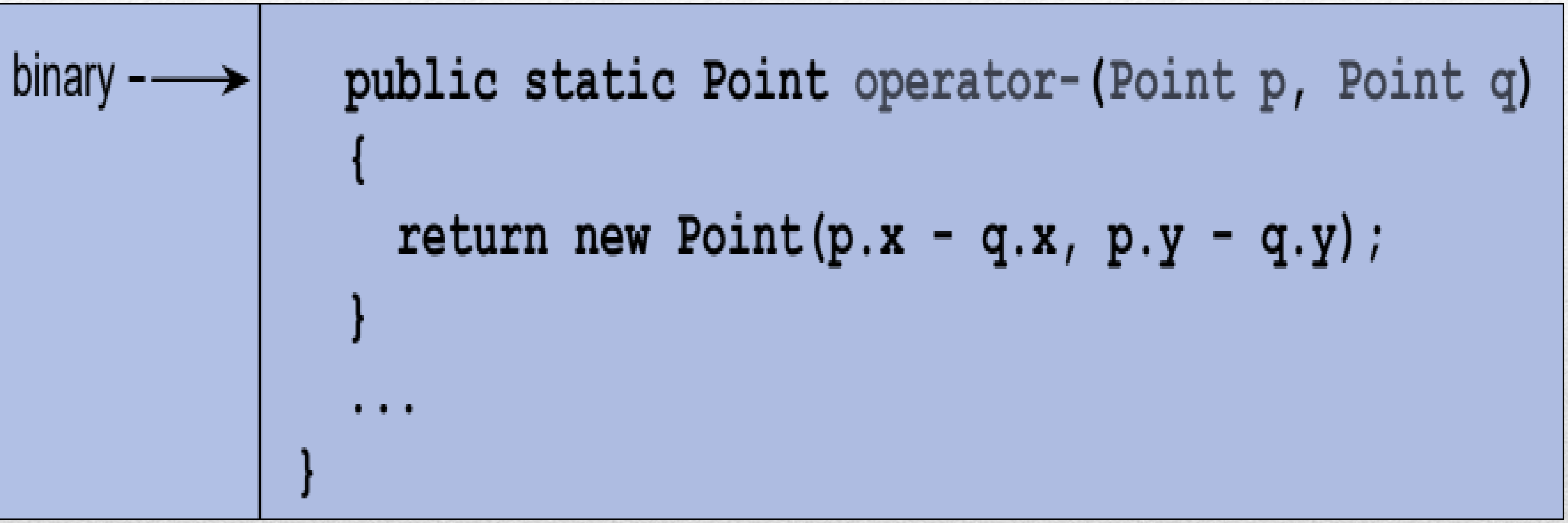
Binary operators

```
public class Point
        {
            public int x;
            public int y;

            public Point(int a, int b)
            {
                x=a;
                y=b;
            }

            public static Point operator+(Point p, Point q)
            {
                return new Point(p.x + q.x, p.y + q.y);
            }
        }
```

```
public static void Main(string[] args)
        {
            Point a = new Point(1, 2);
            Point b = new Point(3, 4);
            Point c = a+b;
            Point d = a+b+c;
            Console.WriteLine(c.x + ", " +c.y);
            Console.WriteLine(d.x + ", " +d.y);

        }
```

```
4, 6
8, 12
```

Binary operators



```
public static Point operator-(Point p, Point q)
{
   return new Point(p.x - q.x, p.y - q.y);
}
...
}
```

Unary operators

- Unary operators take single parameter.

```
public class Point
      {
         public int x;
         public int y;

         public Point(int a, int b)
         {
            x=a;
            y=b;
         }

         public static Point operator-(Point p)
         {
            return new Point(-p.x, -p.y);
         }
      }
```

```
public static void Main(string[] args)
      {
         Point a = new Point(1, 2);
         Point c = -a;
         Console.WriteLine(c.x + ", " +c.y);
      }
```

```
-1, -2
```

Operator pairs

Some operators are required to be present in pairs

- **== and !=**
- **> and <**
- **>= and <=**
- **true and false**

Equality

equality →

inequality →

```
public static bool operator==(Point p, Point q)
{
  return p.x == q.x && p.y == q.y;
}
public static bool operator!=(Point p, Point q)
{
  return !(p == q);
}
```

compare points →

```
Point a = new Point(1, 2);
Point b = new Point(3, 4);

if (a == b) ...
```

Compound assignment

- Compound assignment operator provided automatically when corresponding binary operator overloaded

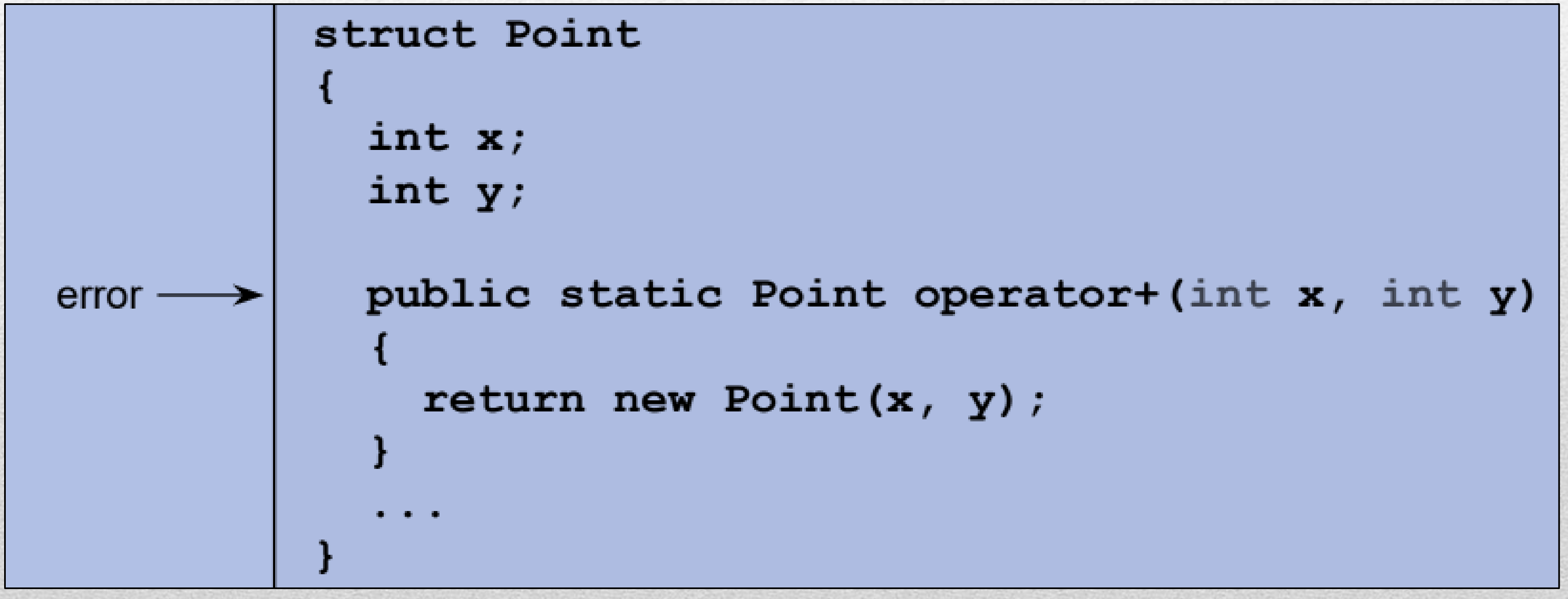
```
public class Point
    {
       public int x;
       public int y;

       public Point(int a, int b)
       {
          x=a;
          y=b;
       }

       public static Point operator+(Point p, Point q)
       {
          return new Point(p.x + q.x, p.y + q.y);
       }
    }
```

```
public static void Main(string[] args)
    {
       Point a = new Point(1, 2);
       Point b = new Point(3, 4);
       Point c = a+b;
       c += b;
       Console.WriteLine(c.x + ", " +c.y);
    }
```

```
7, 10
```

Parameter types

- At least one parameter must be of enclosing type
  - prevents redefinition of operators on existing type



```
struct Point
{
  int x;
  int y;

  public static Point operator+(int x, int y)
  {
    return new Point(x, y);
  }
  ...
}
```

Limitations

- Only some operators can be overloaded
    - unary: + - ! ~ ++ -- true false
    - binary: + - * / % & | ^ << >> == != > < >= <=

- Cannot
    - – create new operators
    - – change precedence
    - – change associativity
    - – change number of arguments
    - – overload prefix/postfix versions separately
    - – pass parameters ref or out

Runtime polymorphism/Overriding

- Method overriding is called runtime polymorphism.
- It is also called late binding.
- Runtime time polymorphism is done using inheritance and virtual functions.
- When overriding a method, the behavior of the method is changed for the derived class.

Example; Overriding

- Base class: Quadrilateral.

- Derived classes: Rectangle, Square, Trapezoid

- Polymorhpic "Quadrilateral.DrawYourself" method
  - 3 ("many") forms of DrawYourself for 3 classes derived from Quadrilateral

- "Quadrilateral.DrawYourself" morphs (re-forms, reshapes) to fit the class of the object for which it is called
  - Becomes Quadrilateral.Rectangle for a Rectangle object
  - Becomes Quadrilateral.Square for a Square object
  - Becomes Quadrilateral.Trapezoid for a Trapezoid object

HOW TO ACHIEVE RUN TIME POLYMORPHISM; VIRTUAL, OVERRIDE KEYWORDS

```
public class Quadrilateral
   {
       public void DrawYourself()
       {
           Console.WriteLine("Quadrilateral
              Drawn");
       }
   }
```

```
public class Rectangle : Quadrilateral
   {
       public void DrawYourself()
       {
            Console.WriteLine(" Rectangle
              Drawn");
       }
   }
```

```
public class Program
   {
       public static void Main(string[] args)
       {
           Quadrilateral quadrilateral;
           quadrilateral = new Rectangle();
           quadrilateral.DrawYourself();
       }
   }
```

Will still draw quadrilateral instead of rectangle; hence no overriding

```
public class Quadrilateral
   {
       public virtual void DrawYourself()
       {
          Console.WriteLine("Quadrilateral
             Drawn");
       }
   }
```

```
public class Rectangle : Quadrilateral
   {
       public override void DrawYourself()
       {
           Console.WriteLine(" Rectangle
             Drawn"); } }
```

```
public class Program
   {
       public static void Main(string[] args)
       {
          Quadrilateral quadrilateral;
          quadrilateral = new Rectangle();
          quadrilateral.DrawYourself();
       }
   }
```

Polymorphic methods are declared with "virtual" keyword.

The keyword virtual means "I might want to make another version of this method in a child class".

- **the "override" keyword is used to provide an implementation.**
- **The override modifier allows a method to override the virtual method of its base class at run-time.**

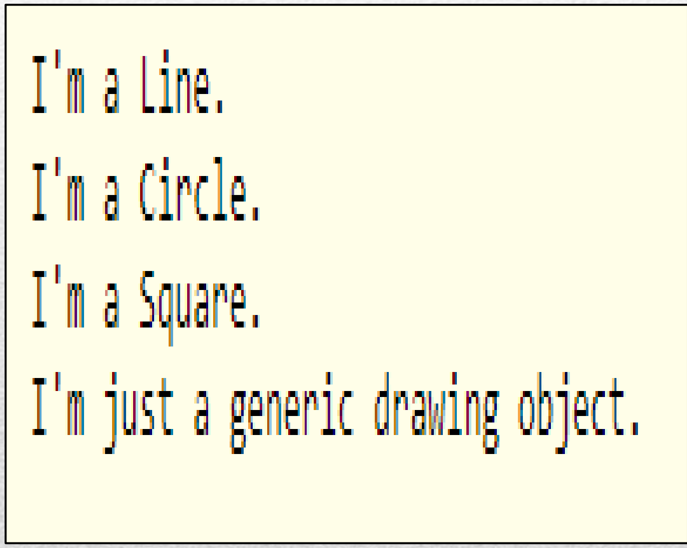
Will now draw rectangle.

Method Overriding is handy when a group of objects is stored in array/list and some method is invoked on all of them.

```
public class drawingobject
   {
       public virtual void Draw()
       {
           Console.WriteLine("I'm just a
generic drawing object.");
       } }
```

```
public class Circle : drawingobject
   {
       public override void Draw()
       {
           Console.WriteLine("I'm a Circle.");
       }
   }
```

```
public static void Main(string[] args)
       {
           drawingobject[] dobj = new drawingobject[4];
           dobj[0] = new Line(); dobj[1] = new Circle();
           dobj[2] = new Square(); dobj[3] = new drawingobject();
           foreach (drawingobject drawobj in dobj)
           {
              drawobj.Draw();
           }
       }
```



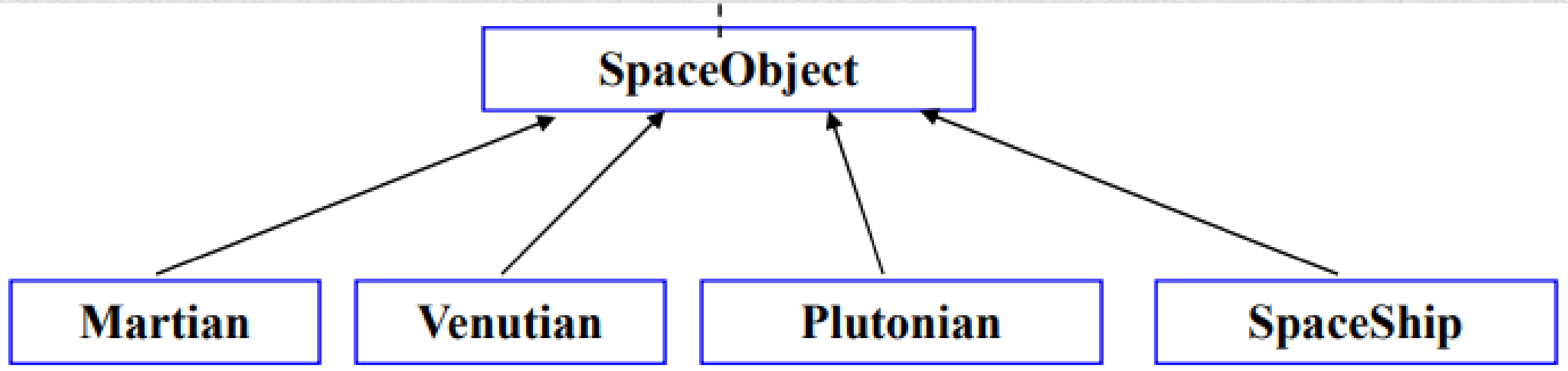
```
I'm a Line.
I'm a Circle.
I'm a Square.
I'm just a generic drawing object.
```

```
public class Line : drawingobject
   {
       public override void Draw()
       {
           Console.WriteLine("I'm a Line.");
       }
   }
```

```
public class Square : drawingobject
   {
       public override void Draw()
       {
           Console.WriteLine("I'm a Square.");
       }
   }
```

Overriding; Another Example

- SpaceObject base-class – contains method DrawYourself
  - Martian derived-class (implements DrawYourself)
  - Venutian derived-class (implements DrawYourself)
  - Plutonian derived-class (implements DrawYourself)
  - SpaceShip derived-class (implements DrawYourself)



Overriding; Another Example

- A screen-manager program may contain a SpaceObject array of references to objects of various classes that derive from SpaceObject

```
arrOfSpaceObjects[ 0 ] = martian1;
arrOfSpaceObjects[ 1 ] = martian2;
arrOfSpaceObjects[ 2 ] = venutian1;
arrOfSpaceObjects[ 3 ] = plutonian1;
arrOfSpaceObjects[ 4 ] = spaceShip1;
```

- To refresh the screen, the screen-manager calls DrawYourself on each object in the array

```
foreach(SpaceObject spaceObject in arrOfSpaceObjects)
{
        spaceObject.DrawYourself
}
```

The program polymorphically calls the appropriate version of DrawYourself on each object, based on the type of that object

RUN TIME POLYMORPHISM; SUMMARY

- **Polymorphism(run time) means to invoke derived class methods through base class reference during run-time.**

- **Its handy when a group of objects is stored in array/list and we invoke some method on all of them.**

- **In C#, for overriding the base class method in derived class declare base class method as virtual and derived class method as override.**

- **If a derived class wishes to hide a method in the parent class, it will use the new keyword.**

Overloading vs. Overriding

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data
- Example:

  int foo(*string*[] bar);

  int foo(*int* bar1, *float* a);

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types
- Example:

  class Base {

  public virtual int foo() {} }

  class Derived {

  public override int foo() {}}

ABSTRACT vs. CONCRETE CLASSES

Abstract classes

- **Are base classes (called abstract base classes)**
- **Cannot be instantiated**
- **Incomplete**
    - **subclasses fill in "missing pieces“**

Concrete classes

- **Can be instantiated**
- **Implement every method they declare**
- **Provide specifics**

ABSTRACT CLASSES

- Contains one or more abstract methods
  - Subclasses must override if the subclasses are to be concrete

- To define an abstract class, use keyword abstract in the declaration

ABSTRACT METHODS

- A method without an implementation is called an abstract method

- Abstract methods are often used to create an interface

- Any class with an abstract method or property must be declared abstract

- Abstract methods and abstract properties are implicitly virtual

- To declare a method or property abstract, use keyword abstract in the declaration

PURPOSE of ABSTRACT CLASSES

- **The purpose of an abstract class is primarily to provide an appropriate base class from which other classes can inherit, and thus share a common design.**

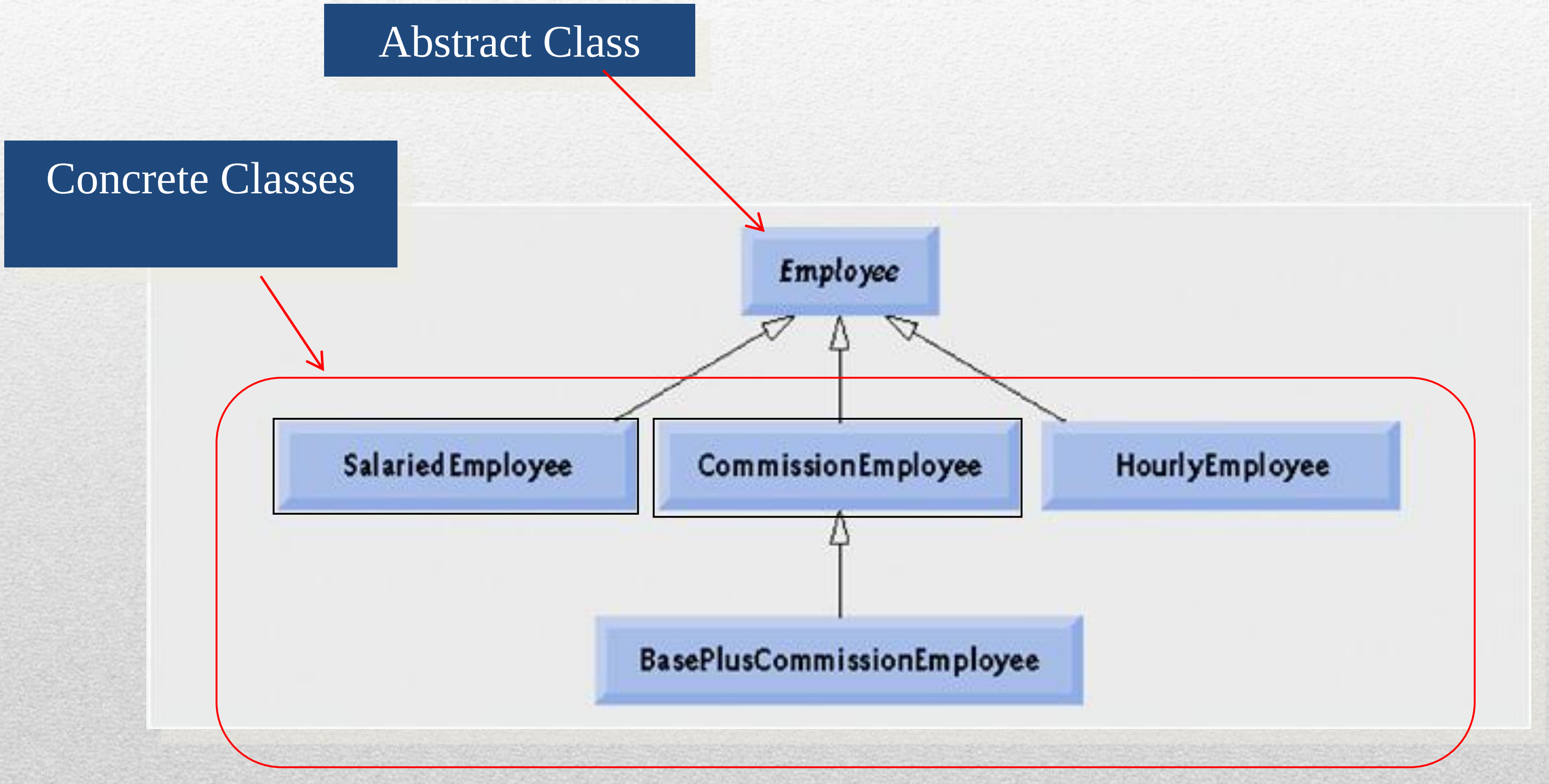
- **Many inheritance hierarchies have abstract super classes occupying the top few levels**

- **Apples, Oranges, Pears, and Avocados are types of Fruit: But what does “a fruit” taste like? How many calories are in “a fruit”? How much does “a fruit” cost?**

Why can’t abstract classes be instantiated

- Abstract classes are too general to create real objects—they specify only what is common among derived classes.
- We need to be more specific before we can create objects.
- For example, we could derive concrete classes Circle, Square and Triangle from abstract base class TwoDimensionalShape. Draw message is sent to abstract class TwoDimensionalShape, the class knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a real Draw method. Concrete classes provide the specifics needed to instantiate objects.

ABSTRACT CLASS; EXAMPLE



ABSTRACT CLASS; EXAMPLE

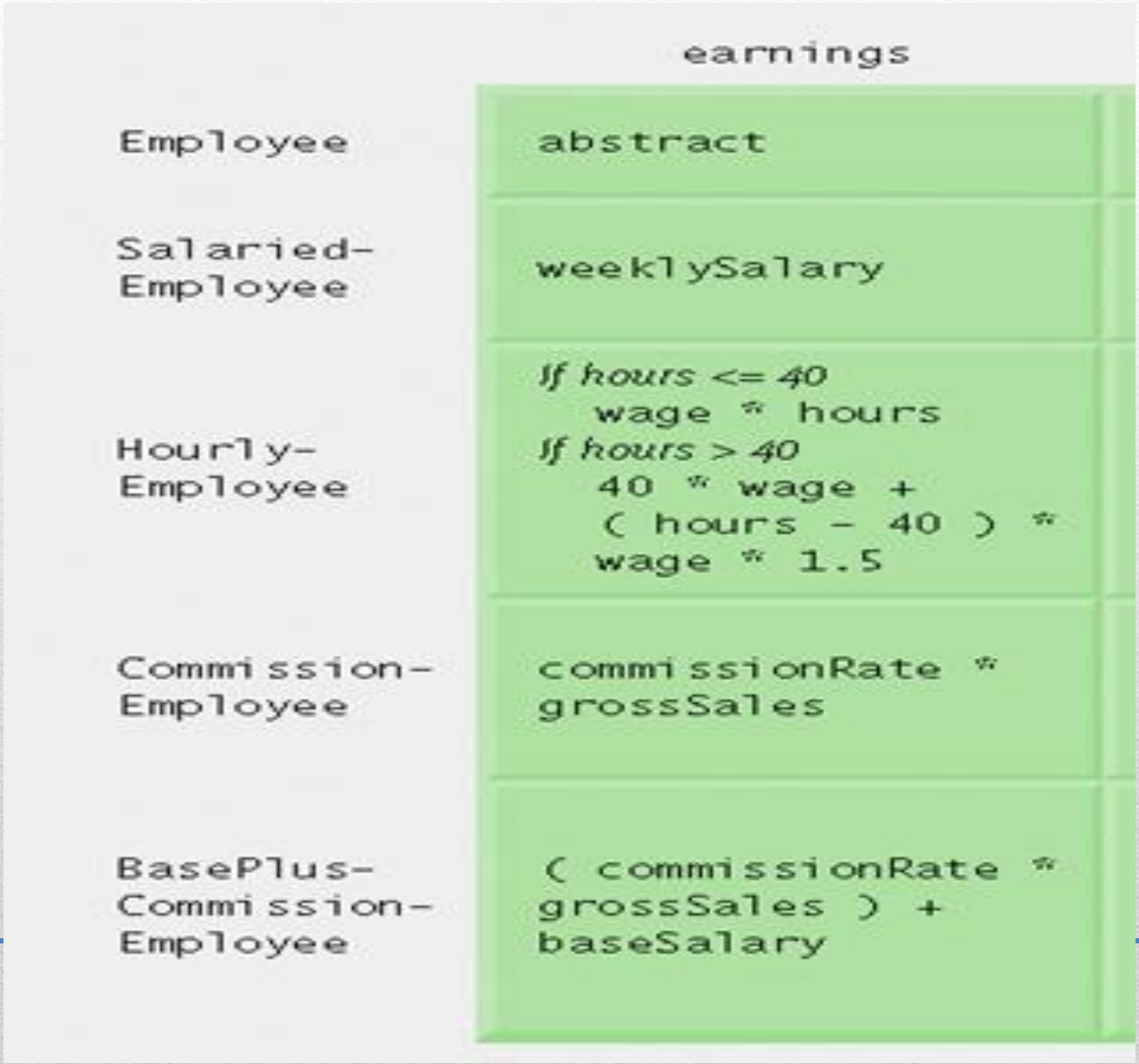
- **abstract super class Employee, earnings is declared abstract**

- **no implementation can be given for earnings in the Employee abstract class**

- **an array of Employee variables will store references to subclass objects**

- **earnings method calls from these variables will call the appropriate version of the earnings method**

ABSTRACT CLASS; EXAMPLE



	earnings
Employee	abstract
Salaried-Employee	weeklySalary
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5
Commission-Employee	commissionRate * grossSales
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary

ABSTRACT CLASS; EXAMPLE

```
public abstract class Employee
    {
        private string name;

        public Employee(string n)
        {
            name = n;
        }

        public abstract decimal Earning();
    }
```

ABSTRACT CLASS; EXAMPLE

```
public class SalariedEmployee : Employee
   {
      private decimal weeklySalary;

      public SalariedEmployee(string name, decimal wSalary): base (name)
      {
         weeklySalary = wSalary;
      }

      public override decimal Earning()
      {
         return weeklySalary;
      }

   }
```

```
public class HourlyEmployee : Employee
    {
        private decimal wagePerHour;
        private decimal hours;

        public HourlyEmployee(string name, decimal wPH, decimal hrs): base (name)
        {
            wagePerHour = wPH;
            hours = hrs;
        }

        public override decimal Earning()
        {
            if (hours <= 40) // no overtime
            {
                return wagePerHour * hours;
            }
            else
            {
                return (40 * wagePerHour) + ((hours - 40) * wagePerHour * 1.5M);
            }
        }
    }
```

ABSTRACT CLASS; EXAMPLE

```
public class Program
    {
        public static void Main(string[] args)
        {
            SalariedEmployee salariedEmployee = new SalariedEmployee("Dummy", 342.5M);
            Console.WriteLine("Earning for Salaried Employee is: "+salariedEmployee.Earning());
            HourlyEmployee hourlyEmployee = new HourlyEmployee("Dummy", 20,40);
            Console.WriteLine("Earning for Hourly Employee is: "+hourlyEmployee.Earning());
        }
    }
```

```
Earning for Salaried Employee is: 342.5
Earning for Hourly Employee is: 800
```

ABSTRACT CLASS; EXAMPLE (Polymorphic Processing)

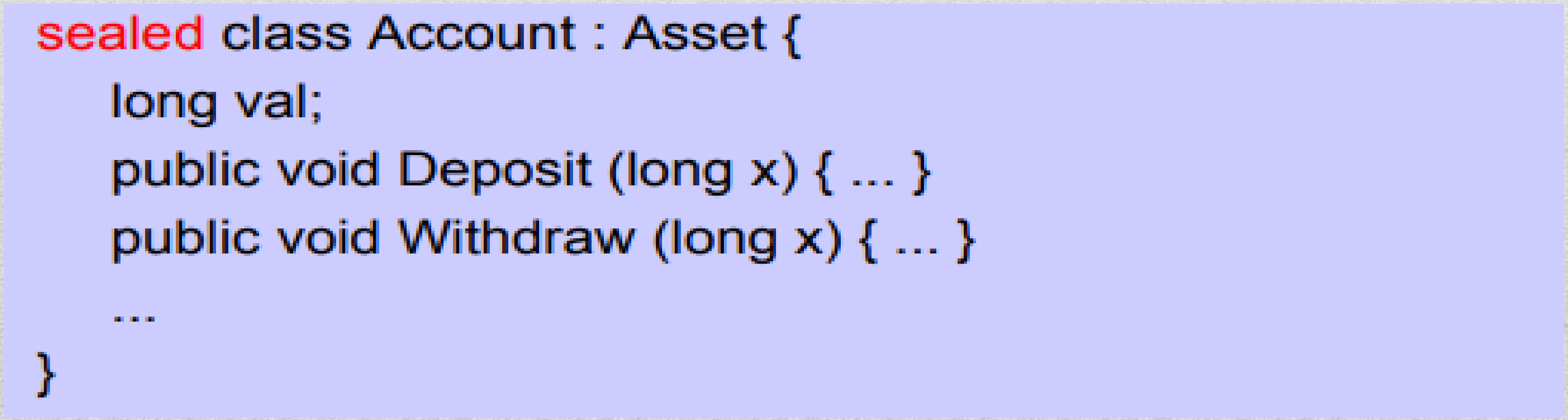
```
public class Program
    {
        public static void Main(string[] args)
        {
            SalariedEmployee salariedEmployee = new SalariedEmployee("Dummy", 342.5M);
            HourlyEmployee hourlyEmployee = new HourlyEmployee("Dummy", 20,40);
            Employee[] employees = {salariedEmployee,hourlyEmployee};
            foreach(Employee employee in employees)
            {
                Console.WriteLine("Earning for Employee is: "+ employee.Earning());
            }
        }
    }
```

```
Earning for Salaried Employee is: 342.5
Earning for Hourly Employee is: 800
```

SEALED CLASSES

- **To design a class correctly that others can extend via derivation can be a tricky task which requires testing with examples to verify that the derivation will work successfully. To avoid unexpected derivation scenarios and problems classes can be marked as sealed**

- **sealed classes cannot be extended but they can inherit from other classes.**



```
sealed class Account : Asset {
    long val;
    public void Deposit (long x) { ... }
    public void Withdraw (long x) { ... }
    ...
}
```

SEALED CLASS; EXAMPLE

- The string type is an example of a type that uses the sealed modifier to prevent derivation.

```
public sealed class String : IComparable,
    ICloneable, IConvertible,
    IEnumerable, IComparable<string>,
    IEnumerable<char>, IEquatable<string>
```