

# Session 5:

# Classes and Methods

For APtech Centre Use Only

# Objectives

- **Explain classes and objects**
- **Define and describe methods**
- **List the access modifiers**
- **Explain method overloading**
- **Define and describe constructors and destructors**

# Object-Oriented Programming

- ◆ Programming languages are based on two fundamental concepts, data and ways to manipulate data.
- ◆ Traditional languages such as Pascal and C used the procedural approach which focused more on ways to manipulate data rather than on the data itself.
- ◆ This approach had several drawbacks such as lack of re-use and lack of maintainability.
- ◆ To overcome these difficulties, OOP was introduced, which focused on data rather than the ways to manipulate data.

# Definition

## Abstraction

- Abstraction is the feature of extracting only the required information from objects. For example, consider a television as an object. It has a manual stating how to use the television.

## Encapsulation

- Details of what a class contains do not have to be visible to other classes and objects that use it. Instead, only specific information can be made visible and the others can be hidden.

## Inheritance

- Inheritance is the process of creating a new class based on the attributes and methods of an existing class. The existing class is called the base class whereas the new class created is called the derived class.



## Polymorphism

- Polymorphism is the ability to behave differently in different situations. It is basically seen in programs where you have multiple methods declared with the same name, but with different parameters and different behavior.

# Classes and Objects

- ◆ C# programs are composed of classes that represent the entities of the program which also include code to instantiate the classes as objects.
- ◆ When the program runs, objects are created for the classes and they may interact with each other to provide the required functionalities.
- ◆ An object is a tangible entity such as a car, a table, or a briefcase.
- ◆ An object in a programming language has a unique identity, state, and behavior.
- ◆ An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

## Example of Object

	
Identity: AXA 43 S State: Color-Red, Wheels-Four Behavior: Running	Identity: T002 State: Color-Brown Behavior: Stable

# Classes

- ◆ Several objects have a common state and behavior and thus, can be grouped under a single class.

## Example

- ◆ A Ford Mustang, a Volkswagen Beetle, and a Toyota Camry can be grouped together under the class Car. Here, Car is the class whereas Ford Mustang, Volkswagen Beetle, and Toyota Camry are objects of the class Car.
- ◆ Following figure displays the class Car:



# Creating Classes

- ◆ In object-oriented programming languages like C#, a class is a template or blueprint which defines the state and behavior of all objects belonging to that class.
- ◆ A class comprises fields, properties, methods, and so on, collectively called data members of the class.
- ◆ Following syntax is used to declare a class:

## Syntax

```
class ClassName
{
    // class members
}
```

where,

ClassName: Specifies the name of the class.

# Guidelines for Naming Classes

- ◆ There are certain conventions to be followed for class names while creating a class that help you to follow a standard for naming classes.
- ◆ These conventions state that a class name:

Should be a noun and written in initial caps and not in mixed case.

Should be simple, descriptive, and meaningful.

Cannot be a C# keyword.

Cannot begin with a digit but can begin with the '@' character or an underscore (\_).

## Example

- ◆ Valid class names are: **Account**, **@Account**, and **\_Account**.
- ◆ Invalid class names are: **2account**, **class**, **Account**, and **Account123**.



# Main Method

- ◆ The `Main ()` method indicates to the CLR that this is the first method of the program which is declared within a class and specifies where the program execution begins.
- ◆ Every C# program that is to be executed must have a `Main ()` method as it is the entry point to the program.
- ◆ The return type of the `Main ()` in C# can be `int` or `void`.

# Instantiating Objects

- ◆ In C#, an object is instantiated using the new keyword. On encountering the new keyword, the Just-in-Time (JIT) compiler allocates memory for the object and returns a reference of that allocated memory.
- ◆ Following syntax is used to instantiate an object:

## Syntax

```
<ClassName><objectName> = new <ClassName>();
```

where,

- ◆ **ClassName:** Specifies the name of the class.
- ◆ **objectName:** Specifies the name of the object.

# Methods

- ◆ Methods are functions declared in a class and may be used to perform operations on class variables, and implements the objects behavior.
- ◆ Methods specify the manner in which a particular operation is to be carried out on the required data members of the class.

## Example

- ◆ The class Car can have a method **Brake()** that represents the 'Apply Brake' action.
- ◆ To perform the action, the method **Brake()** will have to be invoked by an object of class **Car**.

# Creating Methods

- ◆ Conventions to be followed for naming methods state that a method name:

Cannot be a C# keyword, cannot contain spaces, and cannot begin with a digit

Can begin with a letter, underscore, or the "@" character

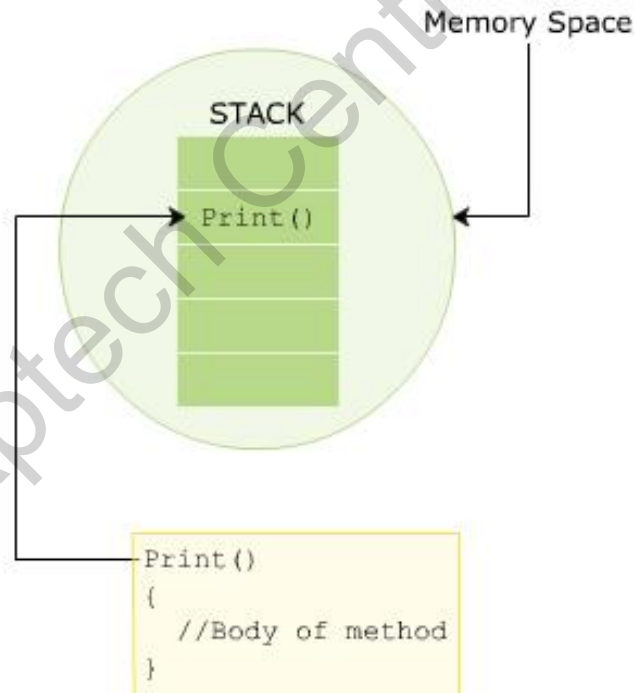
Some of the examples are:  
**Add()**, **Sum\_Add()**, and **@Add()**.

Example

Invalid method names include **5Add**, **AddSum()**, and **int()**.

# Invoking Methods

- ◆ A method can be invoked in a class by creating an object of the class where the object name is followed by a period (.) and the name of the method followed by parentheses.
- ◆ In C#, a method is always invoked from another method. This is referred to as the **calling** method and the invoked method is referred to as the **called** method.



# Method Parameters and Arguments

- ◆ Method parameters and arguments:

## Parameters

- The variables included in a method definition are called parameters. Which may have zero or more parameters, enclosed in parentheses and separated by commas.

## Arguments

- When the method is called, the data that you send into the method's parameters are called arguments.

- ◆ Following figure shows an example of parameters and arguments:

```
class Student{
public void Display(string myParam)
{
...
}
}
public static void Main()
{
    string myArg1 = "this is my argument";
    ...
    objStudent.Display(myArg1);
}
```

Parameter

Argument

# Named and Optional Arguments 1-2

- ◆ A method in a C# program can accept multiple arguments that are passed based on the position of the parameters in the method signature.
- ◆ An argument passed by its name instead of its position is called a named argument.
- ◆ While passing named arguments, the order of the arguments declared in the method does not matter.
- ◆ Following code demonstrates how to use named arguments:

## Snippet

```
using System;
class Student {
    void printName(String firstName, String lastName) {
        Console.WriteLine("First Name = {0}, Last Name = {1}", firstName, lastName);
    }
    static void Main(string[] args) {
        Student student = new Student();
        /*Passing argument by position*/
        student.printName("Henry", "Parker");
        /*Passing named argument*/
        student.printName(firstName: "Henry", lastName: "Parker");
        student.printName(lastName: "Parker", firstName:
            "Henry");
        /*Passing named argument after positional argument*/
        student.printName("Henry", lastName: "Parker");
    }
}
```

# Named and Optional Arguments 2-2

- ◆ In the code:
  - ◆ The first call to the **printNamed()** method passes positional arguments.
  - ◆ The second and third call passes named arguments in different orders.
  - ◆ The fourth call passes a positional argument followed by a named argument.

## Output

```
First Name = Henry, Last Name = Parker  
First Name = Henry, Last Name = Parker  
First Name = Henry, Last Name = Parker  
First Name = Henry, Last Name = Parker
```



# Static Classes 1-2

- ◆ Classes that cannot be instantiated or inherited are known as classes and the `static` keyword is used before the class name that consists of static data members and static methods.
- ◆ It is not possible to create an instance of a static class using the `new` keyword.
- ◆ Main features of static classes:
  - ◆ They can only contain static members.
  - ◆ They cannot be instantiated or inherited and cannot contain instance constructors.

## Snippet

```
using System;
static class Product {
    Static int _productId;
    static double _price;
    static Product() {
        _productId = 10;
        _price = 156.32;
    }
    public static void Display() {
        Console.WriteLine("Product ID: " + _productId);
        Console.WriteLine("Product price: " + _price);
    }
}
class Medicine {
    static void Main(string[] args) {
        Product.Display();
    }
}
```

# Static Classes 2-2

- ◆ In the code:
  - ◆ Since the class **Product** is a static class, it is not instantiated.
  - ◆ So, the method **Display()** is called by mentioning the class name followed by a period (.) and the name of the method.

## Output

```
Product ID: 10  
Product price: 156.32
```

# Static Methods

- ◆ A method is called using an object of the class, but it is possible for a method to be called without creating any objects of the class by declaring a method as static.

## Syntax

```
static<return_type><MethodName> ()  
{  
    // body of the method  
}
```

# Static Variables

- ◆ A static variable is a special variable that is accessed without using an object of a class.
- ◆ A variable is declared as static using the static keyword. When a static variable is created, it is automatically initialized before it is accessed.
- ◆ Only one copy of a static variable is shared by all the objects of the class.
- ◆ Figure displays the static variables:

```
class Employee
{
    public static int EmpId = 20 ;
    public static string EmpName = "James";

    static void Main (string[] args)
    {
        Console.WriteLine ("Employee ID: " + EmpId);
        Console.WriteLine ("Employee Name: " + EmpName);
    }
}
```

# Access Modifiers

- ◆ C# provides you with access modifiers that allow you to specify which classes can access the data members of a particular class.
- ◆ In C#, there are four commonly used access modifiers.



- ◆ Figure displays various accessibility levels:

Accessibility Levels

Access Modifiers		Applicable to the Application	Applicable to the Current Class	Applicable to the Derived Class
	public	✓	✓	✓
	private	✗	✓	✗
	protected	✗	✓	✓
	internal	✗	✓	✓

# ref and out Keywords 1-3

- ◆ The `ref` keyword causes arguments to be passed in a method by reference.
- ◆ In call by reference, the called method changes the value of the parameters passed to it from the calling method.
- ◆ The variables passed by reference from the calling method must be first initialized.
- ◆ Following syntax is used to pass values by reference using the `ref` keyword:

## Syntax

```
<access_modifier><return_type><MethodName> (ref parameter1, ref parameter2,  
parameter3, parameter4, ...parameterN)  
{  
    // actions to be performed  
}
```

# ref and out Keywords 2-3

- ◆ Following code uses the `ref` keyword to pass the arguments by reference:

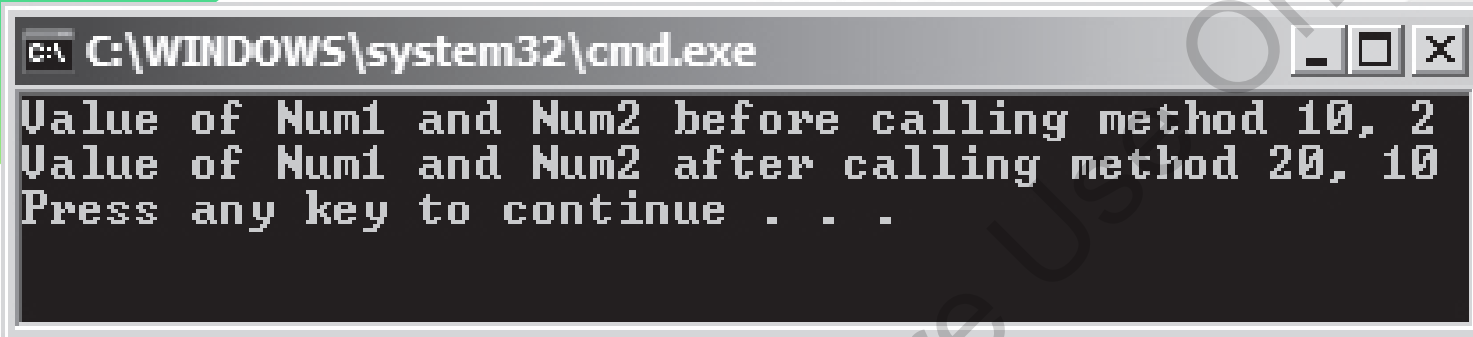
## Snippet

```
using System;
class RefParameters {
    static void Calculate(ref int numValueOne, ref int numValueTwo) {
        numValueOne = numValueOne * 2;
        numValueTwo = numValueTwo / 2;
    }
    static void Main(string[] args) {
        int numOne = 10;
        int numTwo = 20;
        Console.WriteLine("Value of Num1 and Num2 before calling method "
            + numOne +
            ", " + numTwo);
        Calculate(ref numOne, ref numTwo);
        Console.WriteLine("Value of Num1 and Num2 after calling method "
            + numOne +
            ", " + numTwo);
    }
}
```

- ◆ In the code:
  - ◆ The **Calculate()** method is called from the **Main()** method, which takes the parameters prefixed with the `ref` keyword.
  - ◆ The same keyword is also used in the **Calculate()** method before the variables **numValueOne** and **numValueTwo**.
  - ◆ In the **Calculate()** method, the multiplication and division operations are performed on the values passed as parameters and the results are stored in the **numValueOne** and **numValueTwo** variables respectively.
  - ◆ The resultant values stored in these variables are also reflected in the **numOne** and **numTwo** variables respectively as the values are passed by reference to the method **Calculate()**.

# ref and out Keywords 3-3

- ◆ Following figure displays the use of `ref` keyword:



```
C:\WINDOWS\system32\cmd.exe
Value of Num1 and Num2 before calling method 10, 2
Value of Num1 and Num2 after calling method 20, 10
Press any key to continue . . .
```

- ◆ The `out` keyword is similar to the `ref` keyword and causes arguments to be passed by reference.
- ◆ The only difference between the two is that the `out` keyword does not require the variables that are passed by reference to be initialized.
- ◆ Both the called method and the calling method must explicitly use the `out` keyword.



# Method Overloading in C# 1-2

- ◆ In object-oriented programming, every method has a signature which includes:

The number of parameters passed to the method, the data types of parameters and the order in which the parameters are written.

While declaring a method, the signature of the method is written in parentheses next to the method name.

No class is allowed to contain two methods with the same name and same signature.

The concept of declaring more than one method with the same method name, but different signatures is called method overloading.

# Method Overloading in C# 2-2

- ◆ Following figure displays the concept of method overloading using an example:

```
int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo)
{
    int result = valOne + valTwo;
    return result;
}
```



Not Allowed in C#

```
int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo, int valThree)
{
    return valOne + valTwo + valThree;
}
```



Allowed in C#

# Guidelines and Restrictions

The methods to be overloaded should perform the same task.

The signatures of the overloaded methods must be unique.

The return type of the methods can be the same as it is not a part of the signature.

The ref and out parameters can be included as a part of the signature in overloaded methods.

# The this keyword 1-3

- ◆ The `this` keyword is used to refer to the current object of the class to resolve conflicts between variables having same names and to pass the current object as a parameter.
- ◆ You cannot use the `this` keyword with static variables and methods.

For Aptech Centre Use Only

# The this keyword 2-3

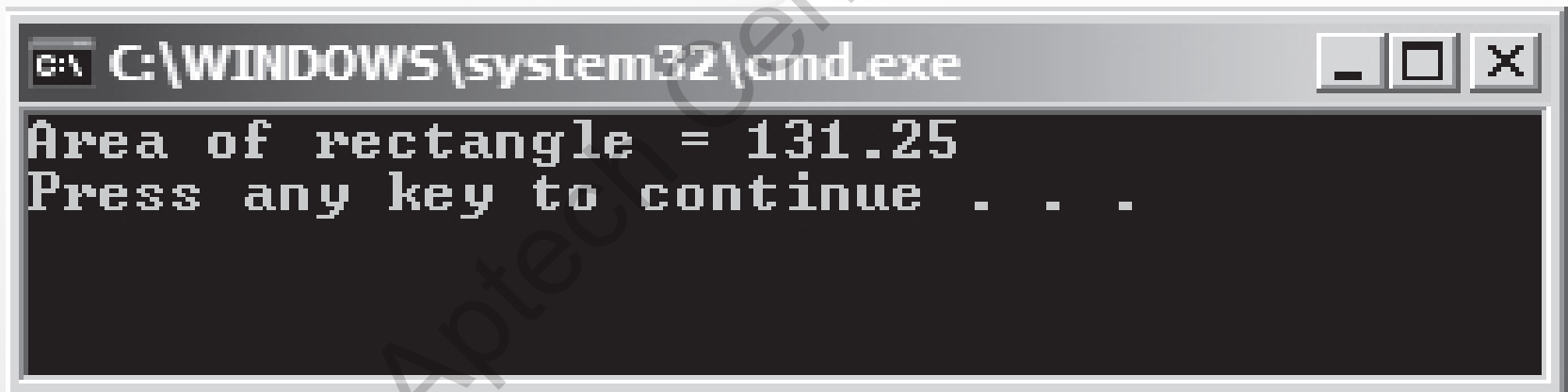
- ◆ In the Following code, the `this` keyword refers to the **length** and **\_breadth** fields of the current instance of the class **Dimension**:

## Snippet

```
using System;
class Dimension
{
    double _length;
    double _breadth;
    public double Area(double _length, double _breadth)
    {
        this._length = _length;
        this._breadth = _breadth;
        return _length * _breadth;
    }
    static void Main(string[] args)
    {
        Dimension objDimension = new Dimension();
        Console.WriteLine("Area of rectangle = " +
            objDimension.Area(10.5, 12.5));
    }
}
```

# The this keyword 3-3

- ◆ In the code:
  - ◆ The **Area()** method has two parameters **\_length** and **\_breadth** as instance variables. The values of these variables are assigned to the class variables using the `this` keyword.
  - ◆ The method is invoked in the **Main()** method. Finally, the area is calculated and is displayed as output in the console window.
- ◆ Following figure displays the use of the `this` keyword:

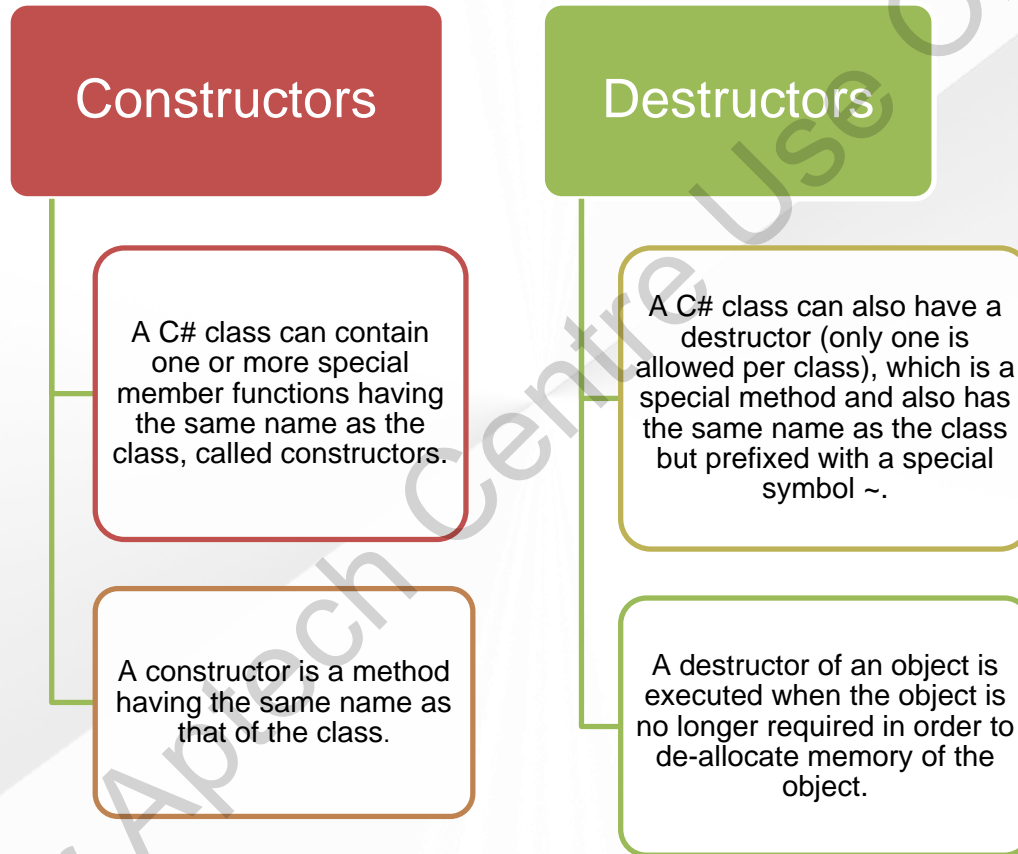


A screenshot of a Windows command prompt window. The title bar shows the path `C:\WINDOWS\system32\cmd.exe`. The window contains the following text:

```
Area of rectangle = 131.25  
Press any key to continue . . .
```

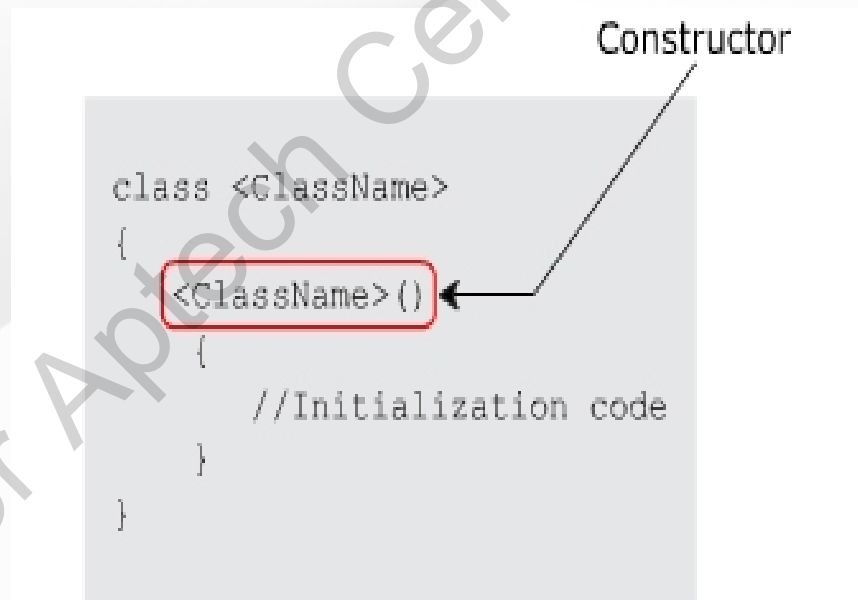
# Constructors and Destructors

- ◆ A C# class can define constructors and destructors as follows:



# Constructors 1-3

- ◆ Constructors can initialize the variables of a class or perform startup operations only once when the object of the class is instantiated.
- ◆ They are automatically executed whenever an instance of a class is created.
- ◆ Following figure shows the constructor declaration:



```
class <ClassName>
{
    <ClassName>()
    {
        //Initialization code
    }
}
```

The diagram illustrates the declaration of a constructor within a C++ class. A red rectangular box highlights the constructor signature `<ClassName>()` inside the class body. An arrow points from the label "Constructor" to this highlighted signature. The class structure shows the constructor as a member function without a return type, followed by an initialization block containing a comment `//Initialization code`.



# Constructors 2-3

- ◆ It is possible to specify the accessibility level of constructors within an application by using access modifiers such as:
  - ◆ **public**: Specifies that the constructor will be called whenever a class is instantiated.
  - ◆ **private**: Specifies that this constructor cannot be invoked by an instance of a class.
  - ◆ **protected**: Specifies that the base class will initialize on its own whenever its derived classes are created.
  - ◆ **internal**: Specifies that the constructor has its access limited to the current assembly.
- ◆ Following code creates a class **Circle** with a `private` constructor:

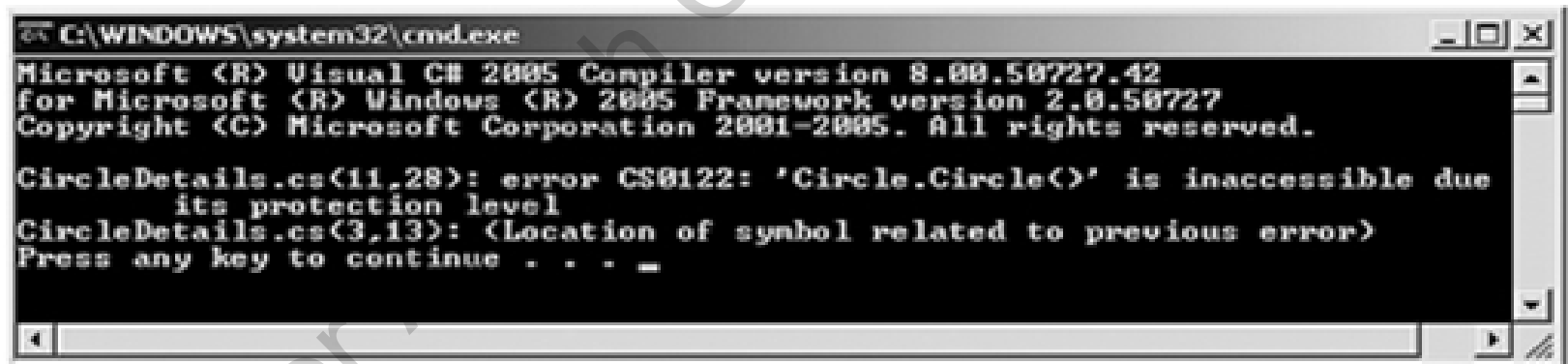
## Snippet

```
using System;
public class Circle
{
    private Circle()
    {
    }
}
class CircleDetails
{
    public static void Main(string[] args)
    {
        Circle objCircle = new Circle();
    }
}
```

# Constructors 3-3

- ◆ In the code:
  - ◆ The program will generate a compile-time error because an instance of the **Circle** class attempts to invoke the constructor which is declared as private. This is an illegal attempt.
  - ◆ If a class has defined only private constructors, the new keyword cannot be used to instantiate the object of the class.
- ◆ Following figure shows the output for creating a class **Circle** with a private constructor:

## Output



```
C:\WINDOWS\system32\cmd.exe
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

CircleDetails.cs(11,28): error CS0122: 'Circle.Circle()' is inaccessible due
    its protection level
CircleDetails.cs(3,13): (Location of symbol related to previous error)
Press any key to continue . . . _
```

# Default and Static Constructors

## Default Constructors

C# creates a default constructor for a class if no constructor is specified within the class.

Automatically initializes all the numeric data type instance variables of the class to zero.

## Static Constructors

A static constructor is used to initialize static variables of the class and to perform a particular action only once.

Invoked before any static member of the class is accessed.

# Static Constructors 1-2

- ◆ Following figure illustrates the syntax for a static constructor:

```
class <ClassName>
{
    static <ClassName>() ← Static Constructor
    {
        //Initialization code
    }
}
```

- ◆ Following code shows how static constructors are created and invoked:

## Snippet

```
using System;
class Multiplication {
static int _valueOne = 10;
static int _product;
static Multiplication() {
Console.WriteLine("Static Constructor initialized");
_product = _valueOne * _valueOne;
}
public static void Method()
{
Console.WriteLine("Value of product = " + _product);
}
static void Main(string[] args) {
Multiplication.Method();
}
}
```

# Static Constructors 2-2

- ◆ In the code:

- ◆ The static constructor **Multiplication()** is used to initialize the static variable **\_product**.
- ◆ Here, the static constructor is invoked before the static method **Method()** is called from the **Main()** method.

## Output

```
Static Constructor initialized  
Value of product = 100
```

# Constructor Overloading 1-3

- ◆ Declaring more than one constructor in a class is called constructor overloading.
- ◆ The process of overloading constructors is similar to overloading methods where every constructor has a signature similar to that of a method.
- ◆ Overloaded constructors reduce the task of assigning different values to member variables each time when required by different objects of the class.

# Constructor Overloading 2-3

- ◆ Following code demonstrates the use of constructor overloading:

## Snippet

```
using System;
public class Rectangle
{
    double _length;
    double _breadth;
    public Rectangle()
    {
    }
    _length = 13.5;
    _breadth = 20.5;
}
public Rectangle(double len, double wide)
{
    _length = len;
    _breadth = wide;
}

public double Area()
{
    return _length * _breadth;
}
static void Main(string[] args)
{
    Rectangle objRect1 = new Rectangle();
    Console.WriteLine("Area of rectangle = " + objRect1.Area());
    Rectangle objRect2 = new Rectangle(2.5, 6.9);
    Console.WriteLine("Area of rectangle = " + objRect2.Area());
}
}
```

# Constructor Overloading 3-3

- ◆ In the code:
  - ◆ Two constructors are created having the same name, **Rectangle**.
  - ◆ However, the signatures of these constructors are different. Hence, while calling the method **Area ()** from the **Main ()** method, the parameters passed to the calling method are identified.
  - ◆ Then, the corresponding constructor is used to initialize the variables **\_length** and **\_breadth**. Finally, the multiplication operation is performed on these variables and the area values are displayed as the output.

## Output

```
Area of rectangle1 = 276.75
```

```
Area of rectangle2 = 17.25
```



# Destructors 1-3

- ◆ A destructor is a special method which has the same name as the class, but starts with the character ~ before the class name and immediately de-allocate memory of objects that are no longer required.
- ◆ Following are the features of destructors:
  - ◆ Destructors cannot be overloaded or inherited and explicitly invoked.
  - ◆ Destructors cannot specify access modifiers and cannot take parameters.

# Destructors 2-3

Following code demonstrates the use of destructors:

## Snippet

```
using System;
class Employee {
private int _empId;
private string _empName;
private int _age;
private double _salary;
Employee(int id, string name, int age, double sal) {
Console.WriteLine("Constructor for Employee called");
_empId = id;
_empName = name;
_age = age;
_salary = sal;
}
~Employee() {
Console.WriteLine("Destructor for Employee called");
}
static void Main(string[] args) {
Employee objEmp = new Employee(1, "John", 45, 35000);
Console.WriteLine("Employee ID: " + objEmp._empId);
Console.WriteLine("Employee Name: " + objEmp._empName);
Console.WriteLine("Age: " + objEmp._age);
Console.WriteLine("Salary: " + objEmp._salary);
}
}
```

# Destructors 3-3

## ◆ In the code:

- ◆ The destructor **~Employee** is created having the same name as that of the class and the constructor.
- ◆ The destructor is automatically called when the object **objEmp** is no longer required to be used.
- ◆ However, you have no control on when the destructor is going to be executed.

For Aptech Centre Use Only

# Summary

- ◆ The programming model that uses objects to design a software application is termed as OOP.
- ◆ A method is defined as the actual implementation of an action on an object and can be declared by specifying the return type, the name and the parameters passed to the method.
- ◆ It is possible to call a method without creating instances by declaring the method as static.
- ◆ Access modifiers determine the scope of access for classes and their members.
- ◆ The four types of access modifiers in C# are public, private, protected and internal.
- ◆ Methods with same name but different signatures are referred to as overloaded methods.
- ◆ In C#, a constructor is typically used to initialize the variables of a class.