Session: 6

Classes and Methods

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.
- The object-oriented approach defines objects as entities having a defined set of values and a defined set of operations that can be performed on these values.

Features

- Object-oriented programming provides the following features:

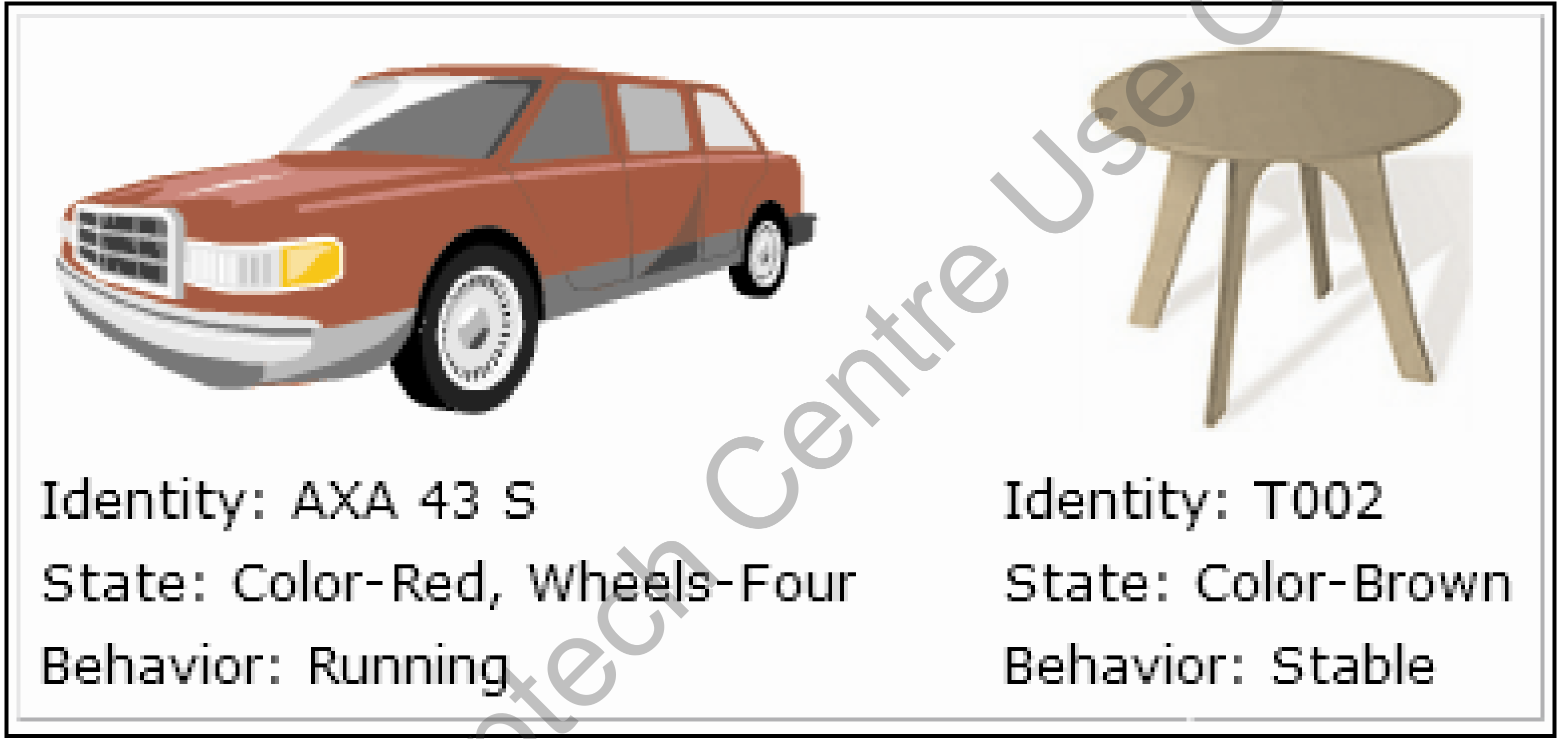
Abstraction	•Abstraction is the feature of extracting only the required information from objects. For example, consider a television as an object. It has a manual stating how to use the television. However, this manual does not show all the technical details of the television, thus, giving only an abstraction to the user.
Encapsulation	•Details of what a class contains need not be visible to other classes and objects that use it. Instead, only specific information can be made visible and the others can be hidden. This is achieved through encapsulation, also called data hiding. Both abstraction and encapsulation are complementary to each other.
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. This is a very important concept of object-oriented programming as it helps to reuse the inherited attributes and methods.
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 1-2

- 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 functionalities of the program.
- An object is a tangible entity such as a car, a table, or a briefcase. Every object has some characteristics and is capable of performing certain actions.
- The concept of objects in the real world can also be extended to the programming world. An object in a programming language has a unique identity, state, and behavior.
- The state of the object refers to its characteristics or attributes whereas the behavior of the object comprises its actions.
- An object has various features that can describe it which could be the company name, model, price, mileage, and so on.

Classes and Objects 2-2

- The following figure shows an example of objects:



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

Identity: T002
State: Color-Brown
Behavior: Stable

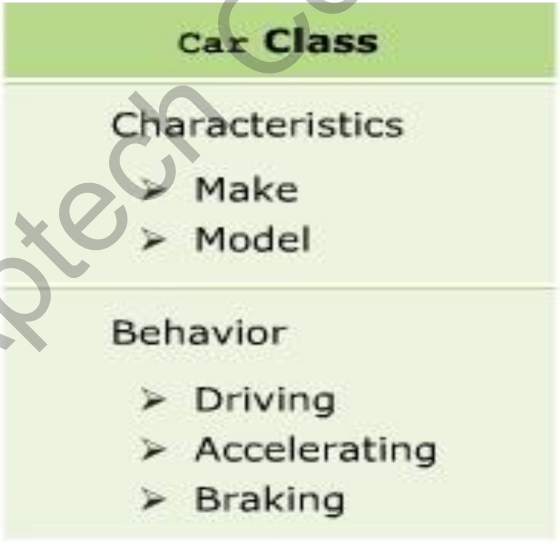
- An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

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.
- The following figure displays the class Car:



Creating Classes 1-2

- The concept of classes in the real world can be extended to the programming world, similar to the concept of objects.
- 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. In C#, the class declaration starts with the `class` keyword followed by the name of the class.
- The following syntax is used to declare a class:

Syntax

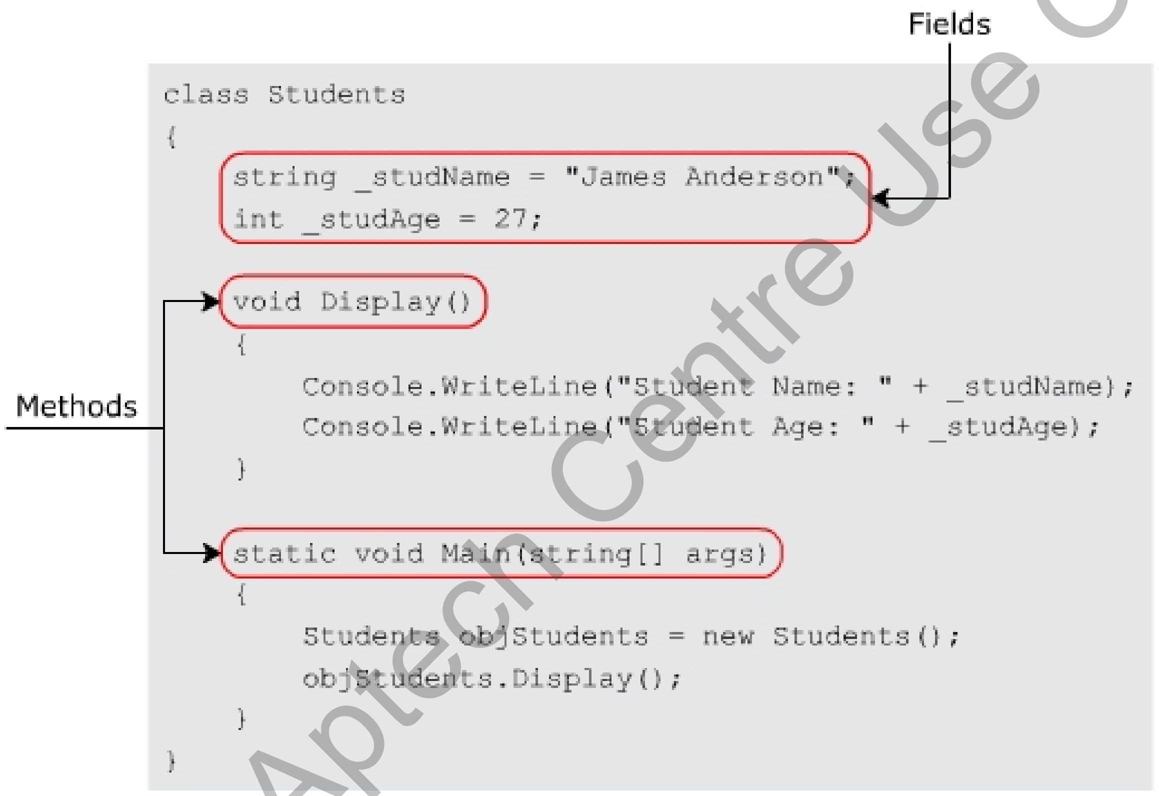
```
{
// class members
}
```

where,

`ClassName`: Specifies the name of the class.

Creating Classes 2-2

- The following figure displays a sample class:



```
class Students
{
    string _studName = "James Anderson";
    int _studAge = 27;

    void Display()
    {
        Console.WriteLine("Student Name: " + _studName);
        Console.WriteLine("Student Age: " + _studAge);
    }

    static void Main(string[] args)
    {
        Students objStudents = new Students();
        objStudents.Display();
    }
}
```

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`**, **`Acccount`**, 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 1-2

- It is necessary to create an object of the class to access the variables and methods defined within it.
- 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.
- The following syntax is used to instantiate an object.

Syntax

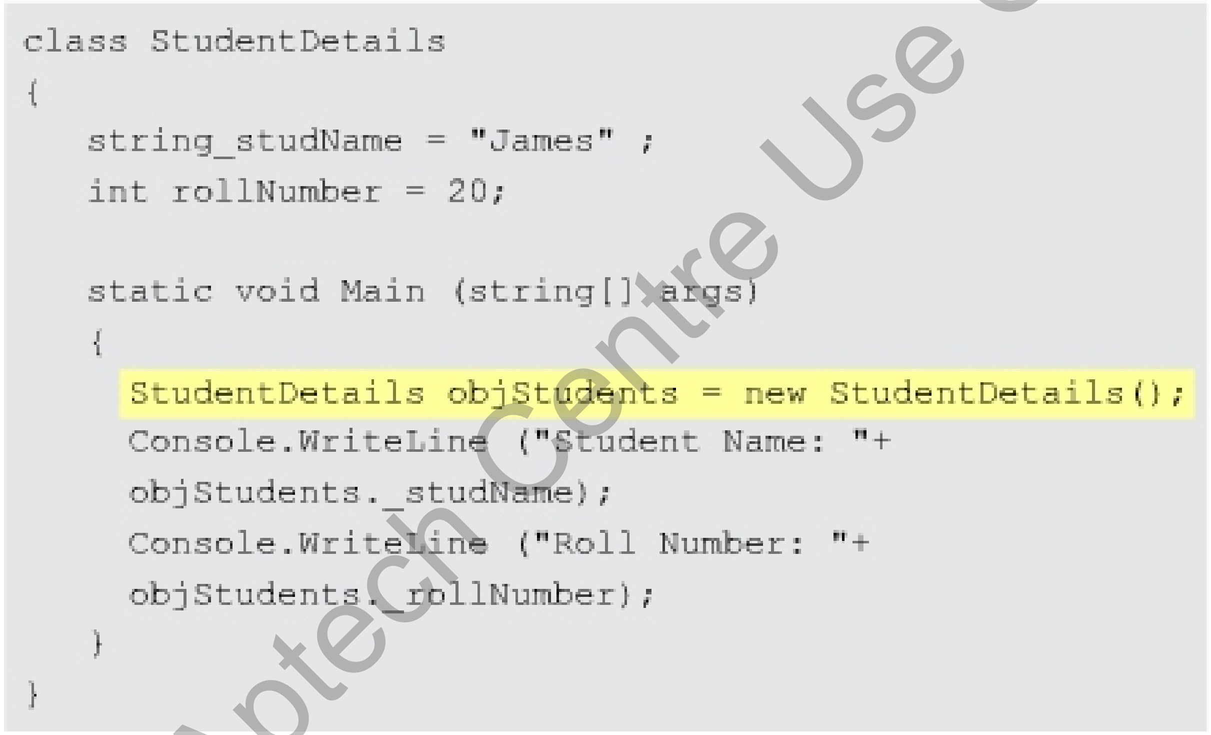
```
<ClassName><objectName> = new <ClassName>();
```

where,

- `ClassName`: Specifies the name of the class.
- `objectName`: Specifies the name of the object.

Instantiating Objects 2-2

- The following figure displays an example of object instantiation:



```
class StudentDetails
{
   string_studName = "James" ;
   int rollNumber = 20;

   static void Main (string[] args)
   {
     StudentDetails objStudents = new StudentDetails();
     Console.WriteLine ("Student Name: "+
     objStudents._studName);
     Console.WriteLine ("Roll Number: "+
     objStudents._rollNumber);
   }
}
```

Methods

- Methods are functions declared in a class and may be used to perform operations on class variables.
- They are blocks of code that can take parameters and may or may not return a value.
- A method implements the behavior of an object, which can be accessed by instantiating the object of the class in which it is defined and then invoking the method.
- 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 1-2

- 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 examples of valid method names are: **`Add()`, `Sum_Add()`, and `@Add()`.**

Example

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

- The following syntax is used to create a method:

Syntax

```
<access_modifier><return_type><MethodName> ([list of parameters]){
// body of the method
}
```

- where,
  - `access_modifier`: Specifies the scope of access for the method.
  - `return_type`: Specifies the data type of the value that is returned by the method and it is optional.
  - `MethodName`: Specifies the name of the method.
  - `list of parameters`: Specifies the arguments to be passed to the method.

Creating Methods 2-2

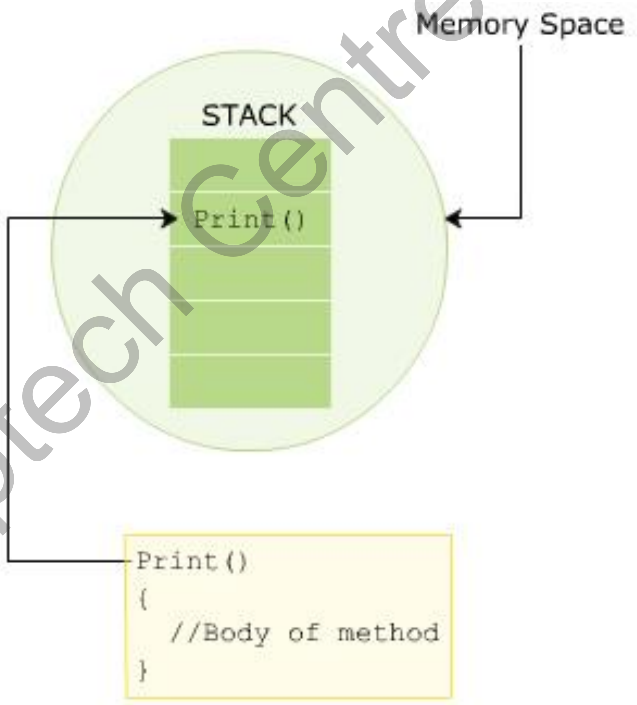
- The following code shows the definition of a method named **`Add()`** that adds two integer numbers:

```
using System;
class Sum
{
int Add(int numOne, int numTwo)
{
int addResult = numOne + numTwo;
Console.WriteLine("Addition = " + addResult);
. . .
}
}
```

- In the code:
  - The **`Add()`** method takes two parameters of type `int` and performs addition of those two values.
  - Finally, it displays the result of the addition operation.

Invoking Methods 1-2

- 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.
- The following figure displays how a method invocation or call is stored in the stack in memory and how a method body is defined:



Invoking Methods 2-2

Snippet

```
class Book{
string _bookName;
public string Print() {
return _bookName;
}
public void Input(string bkName) {
_bookName = bkName;
}
static void Main(string[] args)
{
Book objBook = new Book();
objBook.Input("C#-The Complete Reference");
Console.WriteLine(objBook.Print());
}
}
```

- In the code:
  - The `Main()` method is the calling method and the **`Print()`** and **`Input()`** methods are the called methods.
  - The **`Input()`** method takes in the book name as a parameter and assigns the name of the book to the **`bookName`** variable.
  - Finally, the **`Print()`** method is called from the `Main()` method and it displays the name of the book as the output.

Output

```
C#-The Complete Reference
```

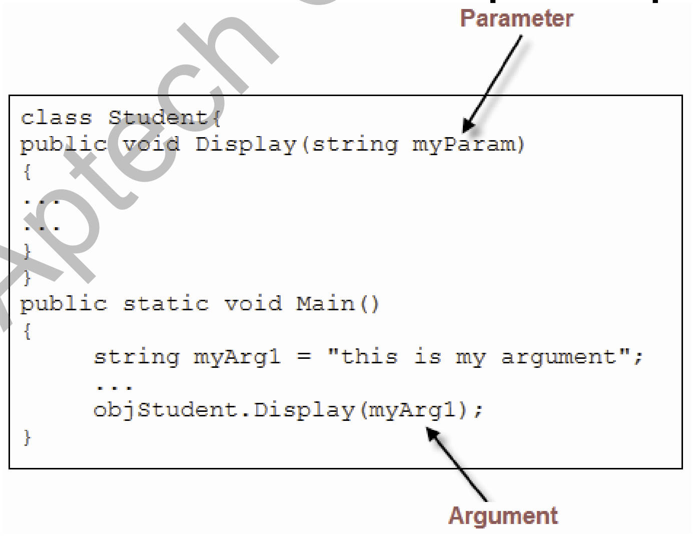
Method Parameters and Arguments

- 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. If the method takes no parameters, it is indicated by empty parentheses.

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

- The 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);
}
```

Named and Optional Arguments 1-6

- A method in a C# program can accept multiple arguments that are passed based on the position of the parameters in the method signature.
- A method caller can explicitly name one or more arguments being passed to the method instead of passing the arguments based on their position.
- 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.
- Named arguments are beneficial because you do not have to remember the exact order of parameters in the parameter list of methods.

Named and Optional Arguments 2-6

- The following code demonstrates how to use named arguments:

Snippet

```
using System;

class Student
{
voidprintName(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 3-6

- 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
```

Named and Optional Arguments 4-6

- The following code shows another example of using named arguments:

Snippet

```
using System;
class TestProgram
{
    void Count(int boys, int girls)
    {
Console.WriteLine(boys + girls);
    }
    static void Main(string[] args)
    {
TestProgramobjTest = new TestProgram();
objTest.Count(boys: 16, girls: 24);
    }
}
```

- C# also supports optional arguments in methods and can be emitted by the method caller.
- Each optional argument has a default value.

Named and Optional Arguments 5-6

- The following code shows how to use optional arguments:

Snippet

```
using System;
classOptionalParameterExample
{
   void printMessage(String message="Hello user!") {
         Console.WriteLine("{0}", message);
   }
   static void Main(string[] args)
   {
OptionalParameterExampleopExample = new
OptionalParameterExample();
opExample.printMessage("Welcome User!");
opExample.printMessage();
   }
}
```

Named and Optional Arguments 6-6

- In the code:
    - The **`printMessage`**`()` method declares an optional argument `message` with a default value `Hello user!`.
    - The first call to the **`printMessage`**`()` method passes an argument value that is printed on the console.
    - The second call does not pass any value and therefore, the default value gets printed on the console.

Output

```
Welcome User!
Hello user!
```

Static Classes 1-3

- 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. The main features of static classes are as follows:
  - They can only contain static members.
  - They cannot be instantiated or inherited and cannot contain instance constructors. However, the developer can create static constructors to initialize the static members.
- The code creates a static class **`Product`** having static variables **`_productId`** and price, and a static method called **`Display()`**.
- It also defines a constructor **`Product()`** which initializes the class variables to 10 and 156.32 respectively.
- Since there is no need to create objects of the static class to call the required methods, the implementation of the program is simpler and faster than programs containing instance classes.

Static Classes 2-3

Snippet

```
using System;
static class Product
{
staticint _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 3-3

- 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 1-3

- 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.
- A static method is declared using the `static` keyword. For example, the `Main()` method is a static method and it does not require any instance of the class for it to be invoked.
- A static method can directly refer only to static variables and other static methods of the class but can refer to non-static methods and variables by using an instance of the class.
- The following syntax is used to create a static method:

Syntax

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

Static Methods 2-3

- The following code creates a static method **Addition()** in the class **Calculate** and then invokes it in another class named **StaticMethods**:

Snippet

```
using System;

class Calculate
{
public static void Addition(int val1, int val2)
{
Console.WriteLine(val1 + val2);
}
public void Multiply(int val1, int val2)
{
Console.WriteLine(val1 * val2);
}
}
classStaticMethods
{
static void Main(string [] args)
{
Calculate.Addition(10, 50);
Calculate objCal = new Calculate();
objCal.Multiply(10, 20);
}
}
```

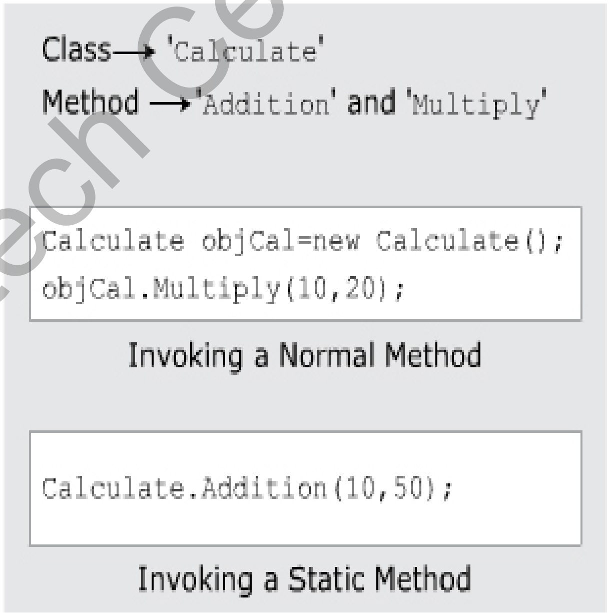
Static Methods 3-3

- In the code, the static method **`Addition()`** is invoked using the class name whereas the **`Multiply()`** method is invoked using the instance of the class.
- Finally, the results of the addition and multiplication operations are displayed in the console window:

Output

```
60

200
```

- The following figure displays invoking a static method:

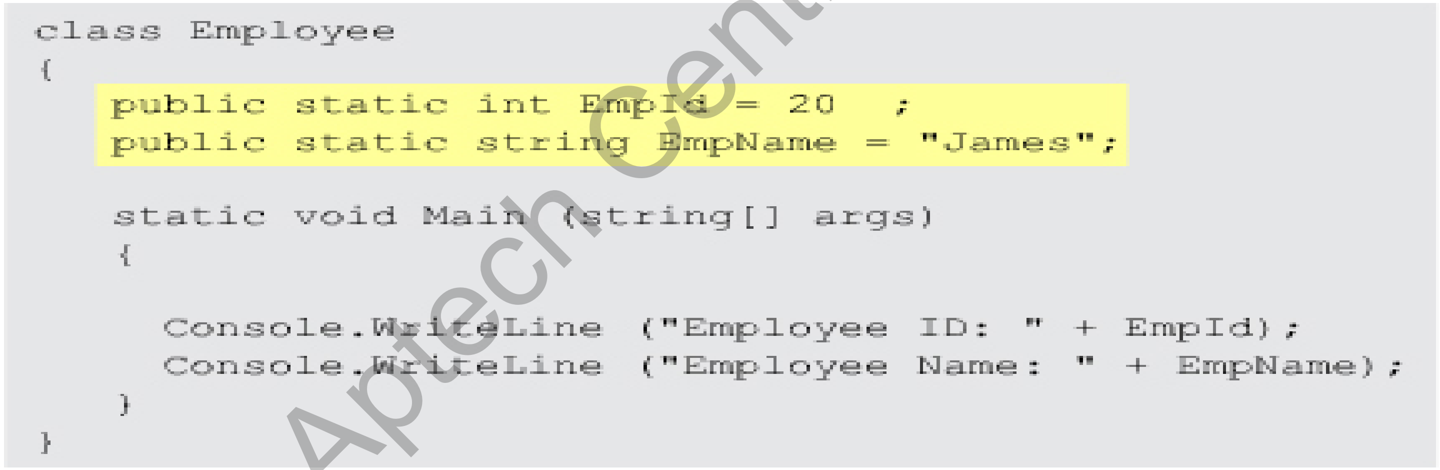


```
Calculate objCal=new Calculate();
objCal.Multiply(10,20);
```

```
Calculate.Addition(10,50);
```

Static Variables

- In addition to static methods, you can also have static variables in C#.
- 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.
- Therefore, a change in the value of such a variable is reflected by all the objects of the class.
- An instance of a class cannot access static variables. Figure 6.8 displays the static variables.
- The following 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 1-4

- 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.

- These are described as follows:
  - **`public`**: The `public` access modifier provides the most permissive access level. The members declared as public can be accessed anywhere in the class as well as from other classes.

    The following code declares a public string variable called **`Name`** to store the name of the person which can be publicly accessed by any other class:

```
class Employee
{
// No access restrictions.
public string Name = "Wilson";
}
```

  - **`private:`** The `private` access modifier provides the least permissive access level. Private members are accessible only within the class in which they are declared.

Access Modifiers 2-4

- Following code declares a variable called **`_salary`** as `private`, which means it cannot be accessed by any other class except for the **`Employee`** class.

Snippet

```
class Employee
{
// No access restrictions.
public string Name = "Wilson";
}
```

- **`protected`:**

  The `protected` access modifier allows the class members to be accessible within the class as well as within the derived classes.

Snippet

```
class Employee
{
// Accessible only within the class
private float _salary;
}
```

Access Modifiers 3-4

- The following code declares a variable called **`Salary`** as protected, which means it can be accessed only by the **`Employee`** class and its derived classes:

Snippet

```
class Employee
{
// Protected access
protected float Salary;
}
```

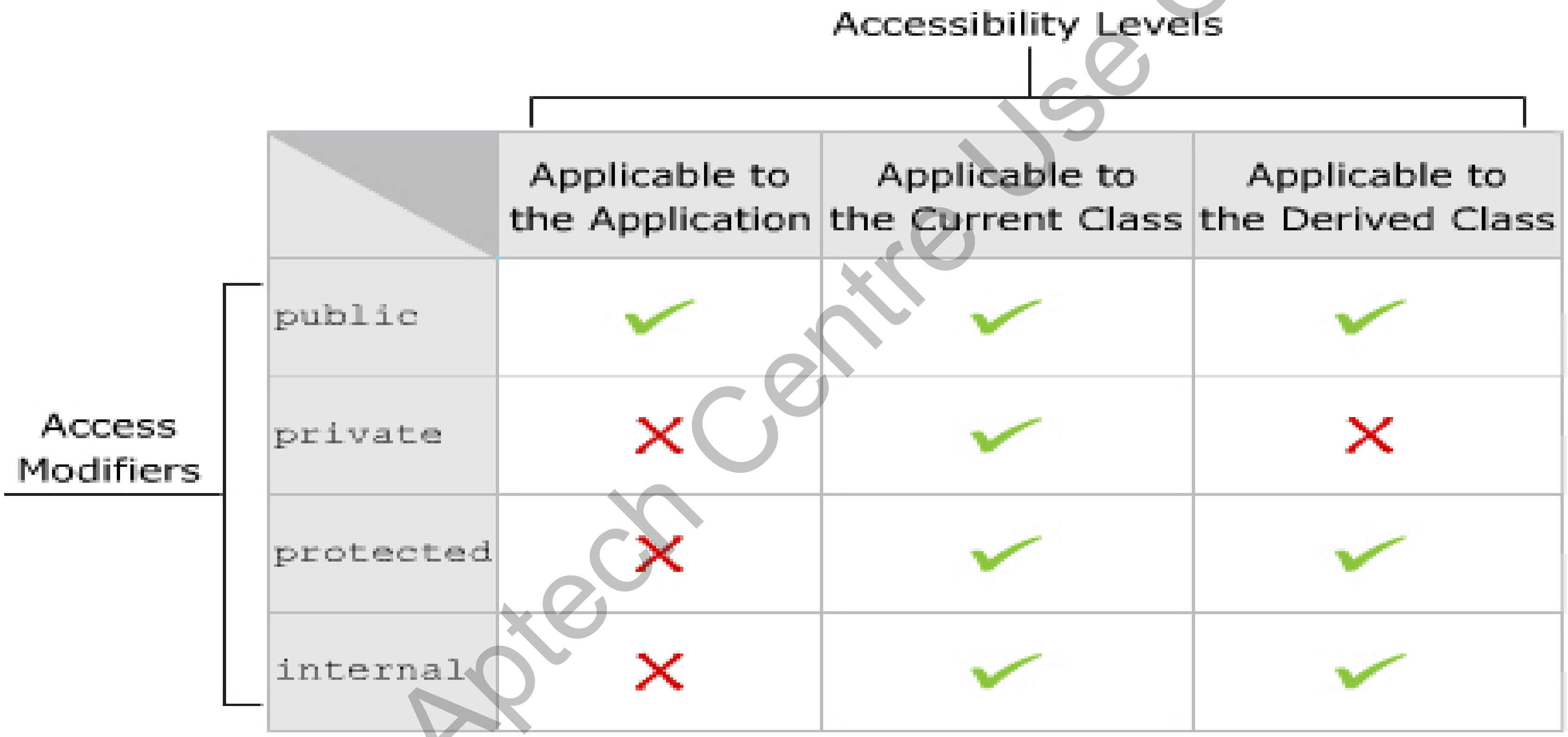
  - **`internal`**: The internal access modifier allows the class members to be accessible only within the classes of the same assembly. An assembly is a file that is automatically generated by the compiler upon successful compilation of a .NET application. The code declares a variable called **`NumOne`** as internal, which means it has only assembly-level access.

Snippet

```
public class Sample
{
// Only accessible within the same assembly
internal static intNumOne = 3;
}
```

Access Modifiers 4-4

- The following figure displays the various accessibility levels:



Access Modifiers	Accessibility Levels: Applicable to the Application	Accessibility Levels: Applicable to the Current Class	Accessibility Levels: Applicable to the Derived Class
`public`	✓	✓	✓
`private`	✗	✓	✗
`protected`	✗	✓	✓
`internal`	✗	✓	✓

`ref` and `out` Keywords 1-7

- 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.
- Any changes made to the parameters in the called method will be reflected in the parameters passed from the calling method when control passes back to the calling method.
- It is necessary that both the called method and the calling method must explicitly specify the `ref` keyword before the required parameters.
- The variables passed by reference from the calling method must be first initialized.
- The 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
}
```

where,

`parameter 1...parameterN:` Specifies that there can be any number of parameters and it is not necessary for all the parameters to be ref parameters.

ref and out Keywords 2-7

- The following code uses the ref keyword to pass the arguments by reference:

Snippet

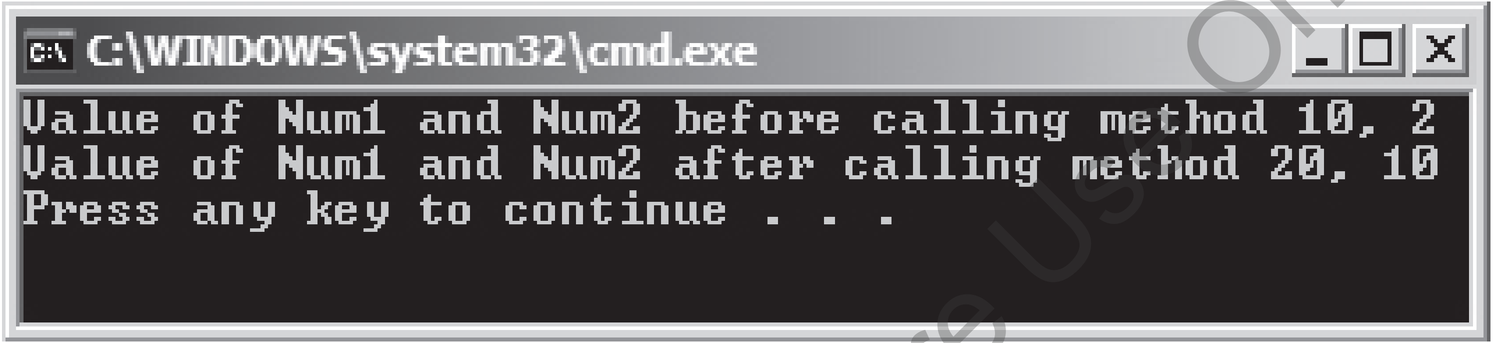
```
using System;

classRefParameters
{
static void Calculate(ref intnumValueOne, ref int
numValueTwo)
{
numValueOne = numValueOne * 2;
numValueTwo = numValueTwo / 2;
]
}
static void Main(string[] args)
{
intnumOne = 10;
intnumTwo = 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);
}
}
```

`ref` and `out` Keywords 3-7

- 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 **`numValueTwovariables`** 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 4-7

- The following figure displays the use of `ref` keyword:



```
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.

ref and out Keywords 5-7

- The following syntax is used to pass values by reference using the `out` keyword:

Syntax

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

- where,

  `parameter 1...parameterN`: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be `out` parameters.

ref and out Keywords 6-7

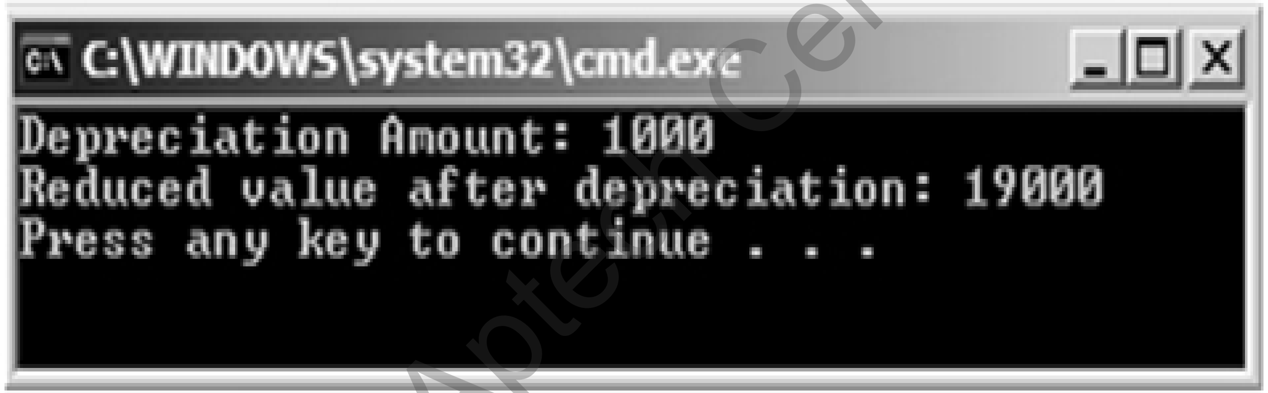
- The following code uses the out keyword to pass the parameters by reference:

Snippet

```
using System;
classOutParameters
{
static void Depreciation(out intval)
{
{
val = 20000;
intdep = val * 5/100;
intamt = val - dep;
Console.WriteLine("Depreciation Amount: " + dep);
Console.WriteLine("Reduced value after depreciation: " +
amt);
}
}
static void Main(string[] args)
{
int value;
Depreciation(out value);
}
}
```

`ref` and `out` Keywords 7-7

- In the code:
  - the **`Depreciation()`** method is invoked from the `Main()` method passing the **`val`** parameter using the `out` keyword. In the **`Depreciation()`** method, the depreciation is calculated and the resultant depreciated amount is deducted from the **`val`** variable. The final value in the **`amt`** variable is displayed as the output.
- The following figure shows the use of `out` keyword:



```
Depreciation Amount: 1000
Reduced value after depreciation: 19000
Press any key to continue . . .
```

Method Overloading in C# 1-3

- 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.

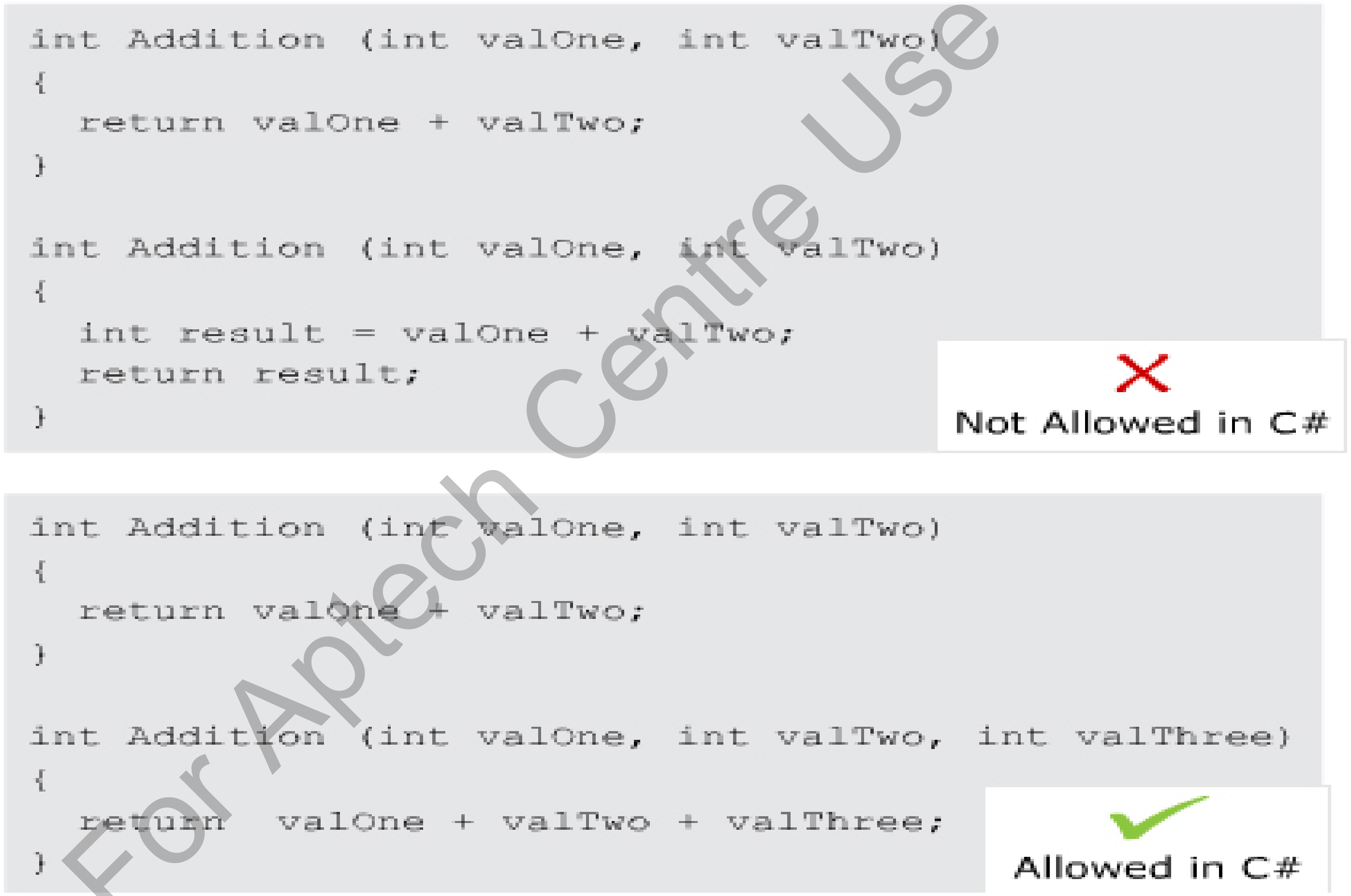
While declaring a method, the signature of the method is written in parentheses next to the method name.

No class is allowed to contain two methods with the same name and same signature, but it is possible for a class to have two methods having the same name but different signatures.

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

Method Overloading in C# 2-3

- The 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;
}
```

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

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

Method Overloading in C# 3-3

- The following code overloads the **Square()** method to calculate the square of the given `int` and `float` values:

Snippet

```
using System;
classMethodOverloadExample
{
static void Main(string[] args)
{
Console.WriteLine("Square of integer value " + Square(5));
Console.WriteLine("Square of float value " + Square(2.5F));
}
staticint Square(intnum)
{
returnnum * num;
}
static float Square(float num)
{
returnnum * num;
}
}
```

- In the code:
  - Two methods with the same name but with different parameters are declared in the class.
  - The two **`Square()`** methods take in parameters of `int` type and `float` type respectively.
  - Within the `Main()` method, depending on the type of value passed, the appropriate method is invoked and the square of the specified number is displayed in the console window.

Output

```
Square of integer value 25
Square of float value 6.25
```

Guidelines and Restrictions

- Guidelines to be followed while overloading methods in a program, to ensure that the overloaded methods function accurately are as follows:

The methods to be overloaded should perform the same task.

The signatures of the overloaded methods must be unique.

When overloading methods, 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.

The this keyword 2-3

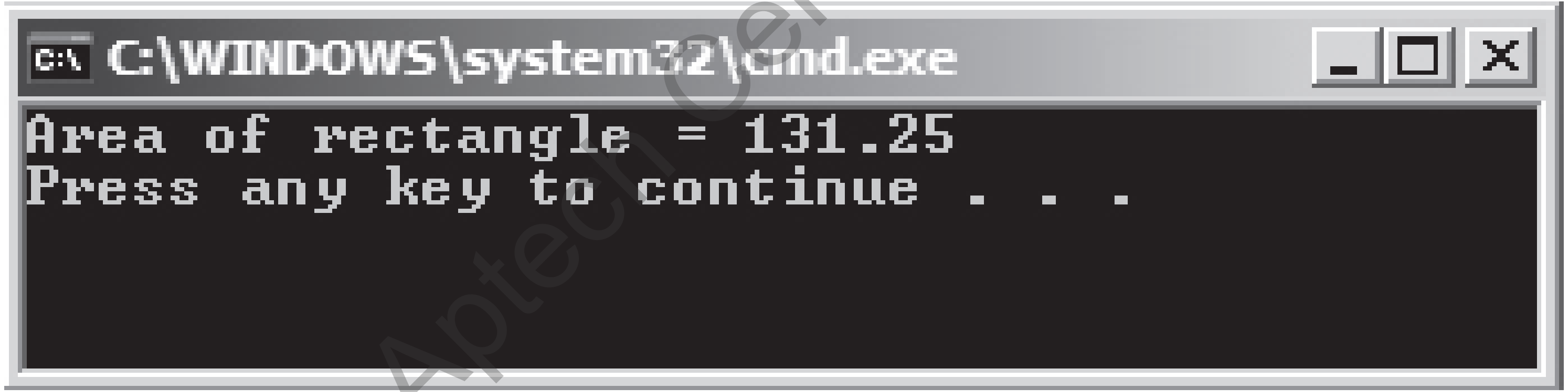
- In the following code, the `this` keyword refers to the **`length`** and **`_breadth`** fields of the current instance of the class **`Dimension`**:

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.
- The following figure displays the use of the `this` keyword:



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

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

Constructors

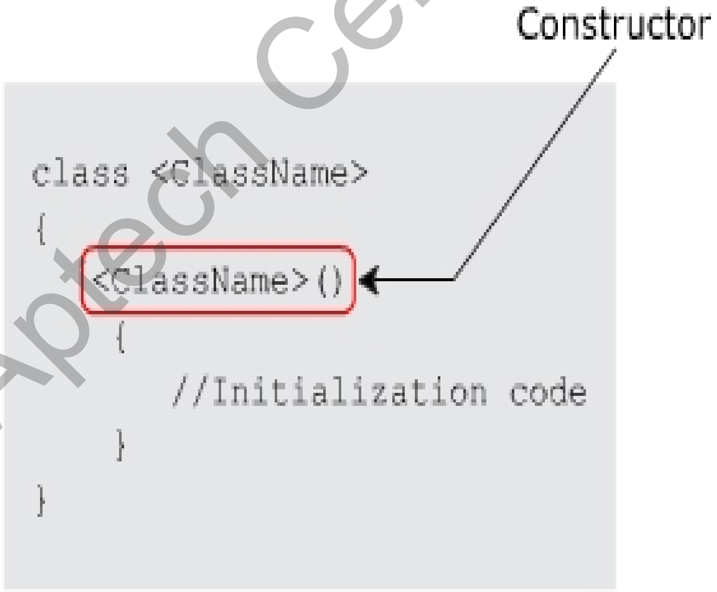
- A C# class can contain one or more special member functions having the same name as the class, called constructors.
- A constructor is a method having the same name as that of the class.

Destructors

- A C# class can also have a destructor (only one is allowed per class), which is a special method and also has the same name as the class but prefixed with a special symbol ~.
- A destructor of an object is executed when the object is no longer required in order to de-allocate memory of the object.

Constructors 1-4

- 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.
- The following figure shows the constructor declaration:



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

Constructors 2-4

- 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. This instantiation can be done from anywhere and from any assembly.
  - **`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. Here, the class object can only be created in the derived classes.
  - **`internal`**: Specifies that the constructor has its access limited to the current assembly. It cannot be accessed outside the assembly.
- The following code creates a class **`Circle`** with a `private` constructor:

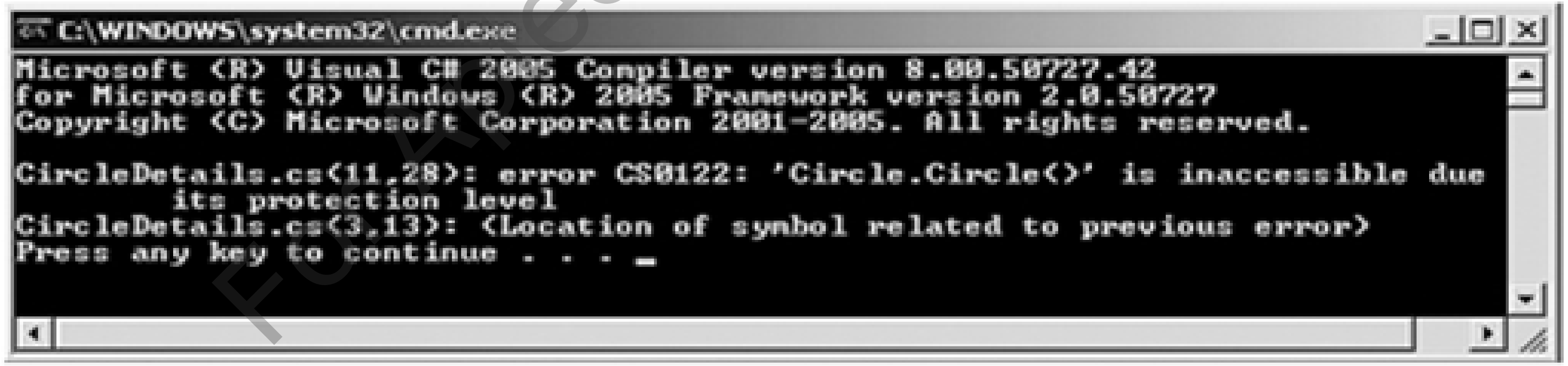
Snippet

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

Constructors 3-4

- 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.
  - Private constructors are used to prevent class instantiation.
  - If a class has defined only private constructors, the `new` keyword cannot be used to instantiate the object of the class.
  - This means no other class can use the data members of the class that has only private constructors.
  - Therefore, private constructors are only used if a class contains only static data members.
  - This is because static members are invoked using the class name.
- The following figure shows the output for creating a class **`Circle`** with a private constructor:

Output



```
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 . . . _
```

Constructors 4-4

- The following code used to initialize the values of **_empName**, **_empAge**, and **_deptName** with the help of a constructor:

Snippet

```
using System;

class Employees
{
string _empName;
int _empAge;
string _deptName;
Employees(string name, intnum)
{
_empName = name;
_empAge = num;
_deptName = "Research & Development";
}
static void Main(string[] args)
{
Employees objEmp = new Employees("John", 10);
Console.WriteLine(objEmp._deptName);
}
}
```

- In the code:
  - A constructor is created for the class **Employees**. When the class is instantiated, the constructor is invoked with the parameters **John** and **10**.
  - These values are stored in the class variables **empName** and **empAge** respectively. The department of the employee is then displayed in the console window.

Default and Static Constructors

Default Constructors

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

The default constructor automatically initializes all the numeric data type instance variables of the class to zero.

If you define a constructor in the class, the default constructor is no longer used.

Static Constructors

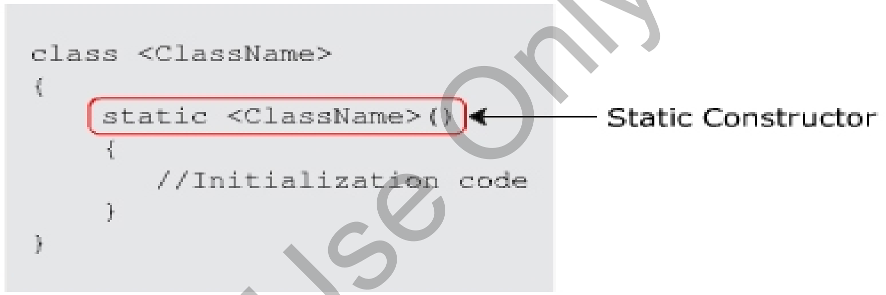
A static constructor is used to initialize static variables of the class and to perform a particular action only once.

It is invoked before any static member of the class is accessed.

A static constructor does not take any parameters and does not use any access modifiers because it is invoked directly by the CLR instead of the object.

Static Constructors 1-2

- The following figure illustrates the syntax for a static constructor:



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

- The following code shows how static constructors are created and invoked.

Snippet

```
using System;

class Multiplication
{
staticint _valueOne = 10;
staticint _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.
- Multiple constructors in a class can be declared wherein each constructor will have different signatures.
- Constructor overloading is used when different objects of the class might want to use different initialized values.
- Overloaded constructors reduce the task of assigning different values to member variables each time when needed by different objects of the class.

Constructor Overloading 2-3

- The 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.
  - Destructors cannot be explicitly invoked.
  - Destructors cannot specify access modifiers and cannot take parameters.
- The following code demonstrates the use of destructors:

Snippet

```
using System;

class Employee
{
privateint _empId;
private string _empName;
privateint _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;
```

Destructors 2-3

```
}
~Employee()
using System;

class Employee
{
privateint _empId;
private string _empName;
privateint _age;
private double _salary;
Employee(int id, string name, int age, double sal)
{
Console.WriteLine("Constructor 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 needed to be used.
  - However, you have no control on when the destructor is going to be executed.

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.