Class and Objects in C#

What are Classes and Objects?

C# is an object-oriented program. In object-oriented programming (OOP), we solve complex problems by dividing them into objects.

To work with objects, we need to perform the following activities:

- create a **class**
- create **objects** from the class

C# Class

- Before we learn about objects, we need to understand the working of classes.
- Class is the blueprint for the object.
- For Example: We can think of the class as a sketch (prototype / house plan) of a house. It contains all the details about the floors, doors, windows, etc. We can build a house based on these descriptions. House is the object.
- Like many houses can be made from the sketch, we can create many objects from a class.

Create a class in C#

- We use the class keyword to create an object. For example,

  ```
  class Account
  {

  }
  ```

  Here, we have created a class named Account.
- A class can contain
  - **fields** - variables to store data(instance variables)
  - **methods** - functions to perform specific tasks (instance methods)
- Let's see an example,

  ```
  class Account
  {
      //fields
      int accno;
      string name;
      double balance;

      //methods
      public void showBalance()
      {

      }
  }
  ```

  In the above example, accno, name, balance are called as fields and showBalance() is a method. In C#, fields and methods inside a class are called members of a class.

C# Objects

- An object is an instance of a class. Suppose we have a class "Account" and John Account, Grace Account, Martin Account are object of the class "Account".

Creating an Object of a class

- In C#, here's how we create an object of the class.

  ```
      ClassName obj = new ClassName();
  ```

  Here, we have used the **new** keyword to create (memory for) an object of the class. And, **obj** is the name or reference of the object.

- Now, let us create an object from the Account class.

  ```
      Account obj = new Account();
  ```

  Now, the obj object can access the fields and methods of the Account class.

Access Class Members using Object

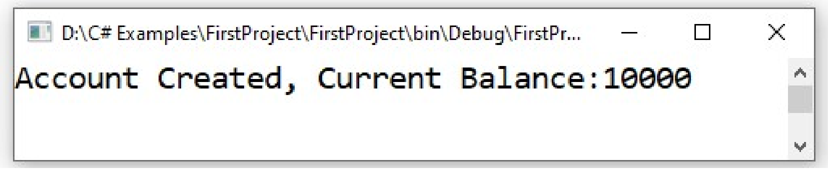
- We use the name or reference of object along with the .(dot) operator to access members of a class. For example,

  ```
    Account account = new Account();

     //Accessing fields
    account.balance = 10000;

   //Accessing method
   account.showBalance();
  ```

- **Example Program:-** Simple Bank Account Class with Fields and Methods

  ```
  using System;
  class Account
  {
      //fields
      int accno;
      string name;
      double balance;

      //methods
      public double showBalance()
      {
          return balance;
      }
      public static void Main(string[] args)
      {
          Account account = new Account();
          account.accno = 101;
          account.name = "John";
          account.balance = 10000;
          double currentBalance = account.showBalance();
          Console.WriteLine("Account Created, Current Balance:"  +
  ```

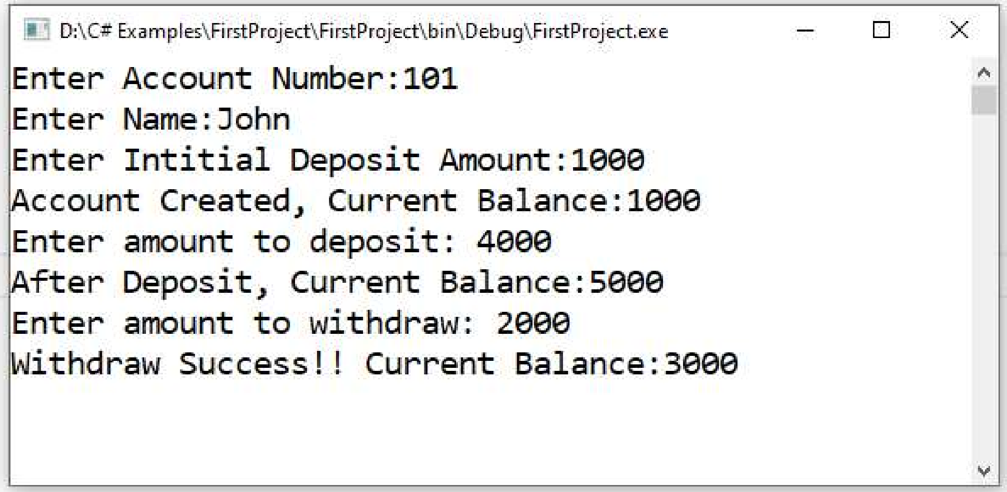
```
                                             currentBalance);
        Console.ReadKey();
    }
}
```


D:\C# Examples\FirstProject\FirstProject\bin\Debug\FirstPr...

```
Account Created, Current Balance:10000
```

- **Example Program2: Banking Operations using Class and Objects**

```
using System;
class Account {
    //fields
    int accno;
    string name;
    double balance;

    //methods
    public double showBalance()
    {
        return balance;
    }
    public void deposit(double amount)
    {
        balance += amount;
    }
    public void withdraw(double amount)
    {
        if (balance > amount) {
            balance -= amount;
        }
        else {
            Console.WriteLine("Error!!! Insufficient Balance");
        }
    }
    public static void Main(string[] args)
    {
        //create object
        Account account = new Account();

        Console.Write("Enter Account Number:");
        account.accno = int.Parse(Console.ReadLine());
        Console.Write("Enter Name:");
        account.name = Console.ReadLine();
        Console.Write("Enter Intitial Deposit Amount:");
        account.balance = double.Parse(Console.ReadLine());
        Console.WriteLine("Account Created, Current Balance:" +
                                          account.showBalance());

        //Deposit Operation
        Console.Write("Enter amount to deposit: ");
        double amount = double.Parse(Console.ReadLine());
        account.deposit(amount);
        Console.WriteLine("After Deposit, Current Balance:" +
                                          account.showBalance());

        //Withdraw Operation
        Console.Write("Enter amount to withdraw: ");
        amount = double.Parse(Console.ReadLine());
        account.withdraw(amount);
        Console.WriteLine("Withdraw Success!! Current Balance:" +
                                          account.showBalance());

        Console.ReadKey();
    }
}
```


D:\C# Examples\FirstProject\FirstProject\bin\Debug\FirstProject.exe

```
Enter Account Number:101
Enter Name:John
Enter Intitial Deposit Amount:1000
Account Created, Current Balance:1000
Enter amount to deposit: 4000
After Deposit, Current Balance:5000
Enter amount to withdraw: 2000
Withdraw Success!! Current Balance:3000
```

Why Objects and Classes?

- Objects and classes help us to divide a large project into smaller sub-problems.
- Suppose you want to create a game that has hundreds of enemies and each of them has fields like health, ammo, and methods like shoot() and run().
- With OOP we can create a single Enemy class with required fields and methods. Then, we can create multiple enemy objects from it.
- Each of the enemy objects will have its own version of health and ammo fields. And, they can use the common shoot() and run() methods.

- Now, instead of thinking of projects in terms of variables and methods, we can think of them in terms of objects.
- This helps to manage complexity as well as make our code reusable.

Creating Multiple Objects of a Class

- We can create multiple objects from the same class. For example,

```
using System;
class Account {
    //fields
    int accno;
    string name;
    double balance;

    //methods
    public double showBalance() {
        return balance;
    }
    public static void Main(string[] args) {
        Account account1 = new Account();
        account1.accno = 101;
        account1.name = "John";
        account1.balance = 10000;
        double currentBalance = account1.showBalance();
        Console.WriteLine("Current Balance:" + currentBalance);

        Account account2 = new Account();
        account2.accno = 102;
        account2.name = "David";
        account2.balance = 20000;
        currentBalance = account2.showBalance();
        Console.WriteLine("Current Balance:" + currentBalance);

        Console.ReadKey();
    }
}
```

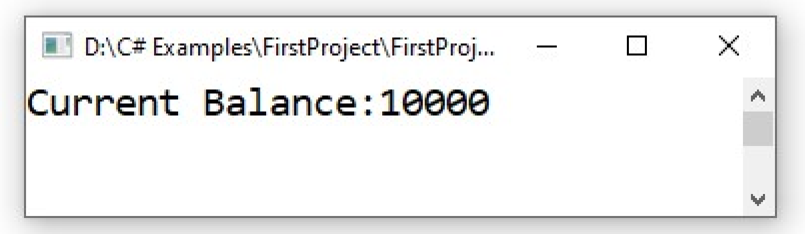
- In the above example, we have created two objects: account1 and account2 from the Account class. Here, you can see both the objects have their own memory of the accno, name and balance fields with different values.

Creating objects in a different class (Recommended for Real Time Application Development)

- In C#, we can also create an object of a class in another class. For example,

```
using System;
 class Account
    {
        public int accno;
        public string name;
```

```
            public double balance;

            public double showBalance()
            {
                return balance;
            }
        }
    class BankApp
    {
        public static void Main(string[] args)
        {
            Account account = new Account();
            account.accno = 101;
            account.name = "John";
            account.balance = 10000;
            double currentBalance = account.showBalance();
            Console.WriteLine("Current Balance:" + currentBalance);
            Console.ReadKey();
        }
    }
```



```
Current Balance:10000
```

- In the above example, we have two classes: **Account** and **BankApp**. Here, we are creating an object account of the Account class in the **BankApp** class.
- We have used the account object to access the members of the Account class from **BankApp**. This is possible because the members in the Account class are public.
- Here, public is an ***access specifier*** that means the class members are accessible from any other classes.

Access Specifiers

- In C#, access specifiers defines the accessibility of types (classes, interfaces, etc) and type members (fields, methods, etc). For example,

```
class Account {
        private int accno;
        private string name;
        private double balance;

        public double showBalance()
        {
            return balance;
        }
  }
```

Here, accno, name and balance are private data members which can be accessed only within the class. The method showBalance() method is specified as public which can be accessed from any class.

- **Types of Access Modifiers**

  In C#, there are 4 basic types of access modifiers.

  1. public - it can be accessed from anywhere.
  2. private - it can only be accessed within the same class.
  3. protected - it can only be accessed from the same class and its derived classes.
  4. internal - it can be accessed only within the same assembly(namespace).

C# Constructor

In C#, a constructor is similar to a method that is invoked when an object of the class is created. However, unlike methods, a constructor:

- has the same name as that of the class
- does not have any return type

Create a C# constructor

Here's how we create a constructor in C#

```
class Account
{
     Account()
     {

     }
}
```

Here, Account() is a constructor. It has the same name as its class.

Call a constructor

Once we create a constructor, we can call it using the **new** keyword. For example,

```
new Account();
```

In C#, a constructor is called when we try to create an object of a class. For example,

```
Account account = new Account();
```

Here, we are calling the Account() constructor to create an object **account**.

Types of Constructors

There are the following types of constructors:

- Parameterless Constructor
- Parameterized Constructor
- Default Constructor

Parameterless Constructor

When we create a constructor without parameters, it is known as a parameterless constructor. For example,

```
 using System;
 class Account
 {
        private int accno;
        private string name;
        private double balance;

        public Account() {
            Console.WriteLine("Inside Contructor..");
            Console.Write("Enter Acc.No:");
            accno = int.Parse(Console.ReadLine());
            Console.Write("Enter Acc.Name:");
            name = Console.ReadLine();
            Console.Write("Enter Initial Amount:");
            balance = double.Parse(Console.ReadLine());
            Console.WriteLine("------------------------");
         }
        public void viewAccount()
        {
           Console.WriteLine("Acc.No: " + accno);
           Console.WriteLine("Name: " + name);
           Console.WriteLine("Balance: " + balance);
        }
  }
 class BankApp
 {
     public static void Main(string[] args)
        {
            Account account = new Account();
            account.viewAccount();
            Console.ReadKey();
        }
 }
```

In the above example, the **public Account()** method is a Parameterless constructor which will be executed whenever we create object using new keyword like **new Account()** . In addition to that, the constructor does not take any parameter, hence the initialization of data members can be done inside the constructor.

Parameterized Constructor

A constructor can also accept parameters. It is called a parameterized constructor. For example,

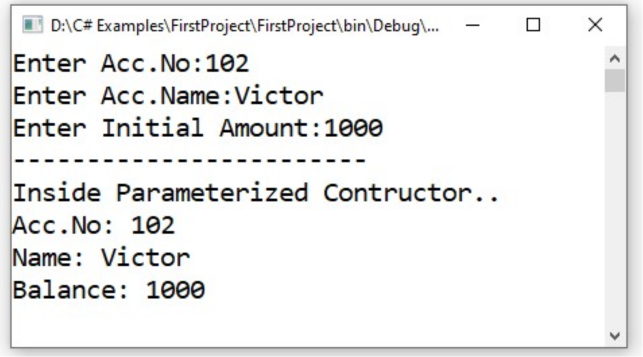
```
using System;
```

```
class Account
{
        private int accno;
        private string name;
        private double balance;

        public Account(int accno, string name, double amount)
        {
            Console.WriteLine("Inside Parameterized Contructor..");
            this.accno = accno;
            this.name = name;
            this.balance = amount;
         }
        public void viewAccount()
        {
           Console.WriteLine("Acc.No: " + accno);
           Console.WriteLine("Name: " + name);
           Console.WriteLine("Balance: " + balance);
        }
  }
 class BankApp
 {
     public static void Main(string[] args)
        {
            Console.Write("Enter Acc.No:");
            int accno = int.Parse(Console.ReadLine());
            Console.Write("Enter Acc.Name:");
            string name = Console.ReadLine();
            Console.Write("Enter Initial Amount:");
            double amount = double.Parse(Console.ReadLine());
            Console.WriteLine("------------------------");

            Account account = new Account(accno,name,amount);
            account.viewAccount();

            Console.ReadKey();
        }
 }
```

Here, the following code creates the parameterized constructor and it will be called when we create object using `new Account(accno,name,amount);`

```
     public Account(int accno, string name, double amount)
        {
            //initialization of data members
        }
```

Note: this keyword used to refer the current object

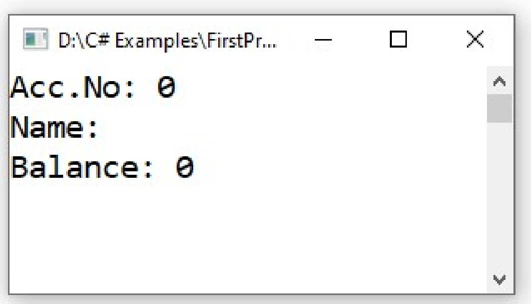


```
Enter Acc.No:102
Enter Acc.Name:Victor
Enter Initial Amount:1000
------------------------
Inside Parameterized Contructor..
Acc.No: 102
Name: Victor
Balance: 1000
```

Default Constructor

If we have not defined a constructor in our class, then the C# will automatically create a default constructor with an empty code and no parameters. For example,

```
using System;
class Account
{
        private int accno;
        private string name;
        private double balance;
        public void viewAccount()
        {
           Console.WriteLine("Acc.No: " + accno);
           Console.WriteLine("Name: " + name);
           Console.WriteLine("Balance: " + balance);
        }
  }
 class BankApp
 {
   public static void Main(string[] args)
    {
      Account account = new Account();//calls default constructor
      account.viewAccount();
      Console.ReadKey();
    }
 }
```



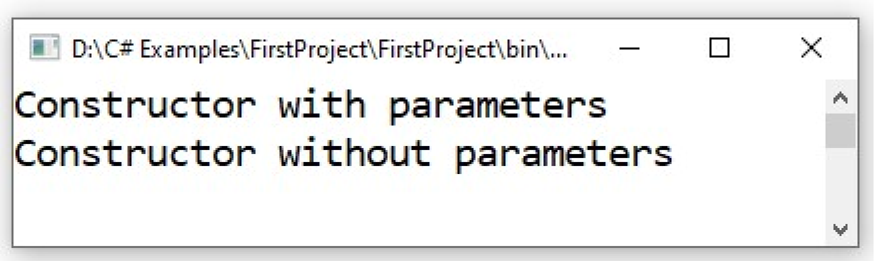
```
Acc.No: 0
Name:
Balance: 0
```

Here, we have not specified any constructor, so compiler includes the default constructor it will initialize the data members to its default values;

Constructor Overloading

We can create two or more constructor in a class with different type of parameters. It is known as constructor overloading. For example,

```
using System;
class Account
{
        private int accno;
        private string name;
        private double balance;

        public Account()
        {
            Console.WriteLine("Constructor without parameters");
            // initialization code
        }
        public Account(int accno, string name, double amount)
         {
            Console.WriteLine("Constructor with parameters");
            // initialization code
         }

  }
 class BankApp
 {
     public static void Main(string[] args)
        {
            Account account1 = new Account(857,"Joy",10000);
            Account account2 = new Account();
            Console.ReadKey();
        }
 }
```



```
Constructor with parameters
Constructor without parameters
```

In the above example, we have overloaded the Account constructor:

- one constructor has no parameters
- another has three parameters

Based on the number of the argument passed during the constructor call, the corresponding constructor is called.

Here,

account1 - calls constructor with three parameters
account2 - calls constructor with no parameters

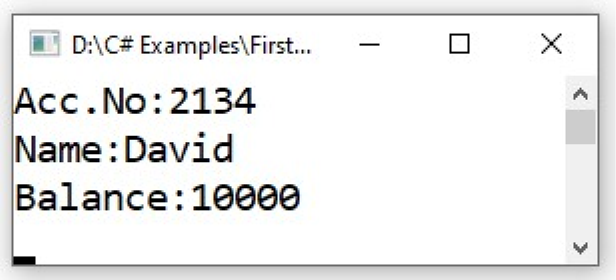
this Keyword

In C#, this keyword refers to the current instance of a class. For example,

```
using System;
class Account
{
        private int accno;
        private string name;
        private double balance;

        public Account(int accno, string name, double balance) {
            this.accno = accno;
            this.name = name;
            this.balance = balance;
         }
        public void view() {
           Console.WriteLine("Acc.No:" + accno + "\nName:" + name +
                                               "\nBalance:" + balance);
        }
  }
 class BankApp
 {
     public static void Main(string[] args)
        {
            Account account = new Account(2134,"David",10000);
            account.view();
            Console.ReadKey();
        }
 }
```

Here, **this** keyword is used to differentiate instance variables and local variables inside the constructor method. For example,

```
            this.accno = accno;

           //this.accno(left side) is instance variable
           //accno(right side) is a local variable
```

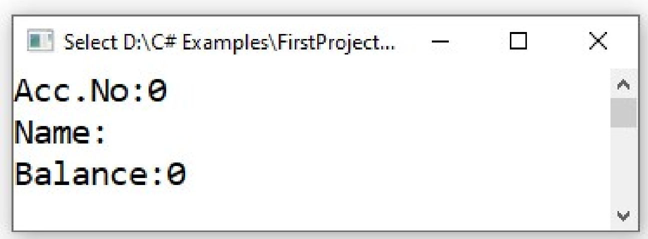


```
Acc.No:2134
Name:David
Balance:10000
```

Note: What if we create constructor as given below

```
public Account(int accno, string name, double balance)
   {
      accno = accno;
      name = name;
      balance = balance;
   }
```

Here, the local variables are initialized with local variables. Hence, the instance variables will not be initialized and takes the default value. The output will be as follows due to instance variable hiding.

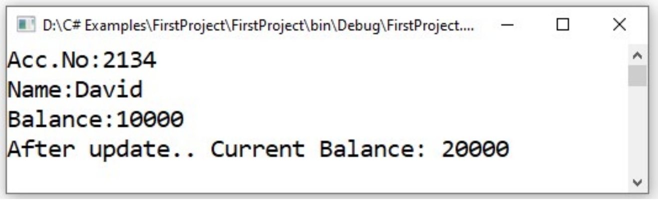


```
Acc.No:0
Name:
Balance:0
```

C# Setter and Getter Expressions

Getters give public access to private data. They may also make a small modification of the returned result.
Setters allow for a private variable to be modified. They are important since they can provide validation before a value is set.

```
using System;
class Account
{
        private int accno;
        private string name;
        private double balance;

        public Account(int accno, string name, double balance) {
            this.accno = accno;
            this.name = name;
            this.balance = balance;
         }
        public double Balance
        {
            get { return balance; }
            set { balance = value; }
        }
        public void view() {
          Console.WriteLine("Acc.No:" + accno + "\nName:" + name +
                                            "\nBalance:" + balance);
        }
  }
 class BankApp
 {
     public static void Main(string[] args)
        {
            Account account = new Account(2134,"David",10000);
            account.view();

            //update balance using setter
            account.Balance = 20000;

            //display Balance only using getter
            Console.WriteLine("After update.. Current Balance: " +
                                                 account.Balance);
            Console.ReadKey();
        }
 }
```


D:\C# Examples\FirstProject\FirstProject\bin\Debug\FirstProject....

```
Acc.No:2134
Name:David
Balance:10000
After update.. Current Balance: 20000
```

Here, the following code creates both setter and getter for the instance variable *balance*.

```
public double Balance
  {
      get { return balance; }
      set { balance = value; }
  }
```

So that, we can modify the balance using

```
            account.Balance = 20000;
```

Also, to access the balance using

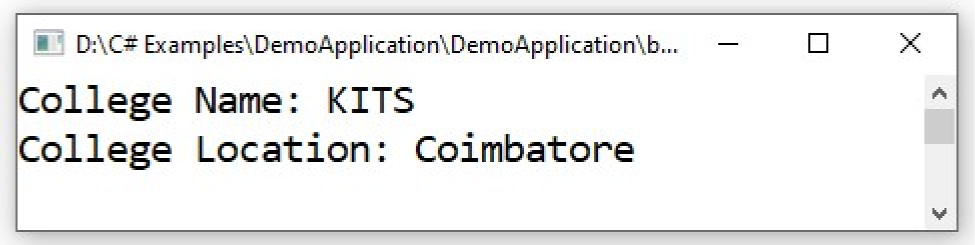
```
           Console.WriteLine("Current Balance: " + account.Balance);
```

static Class

In C#, one is allowed to create a static class, by using static keyword. A static class can only contain static data members, static methods, and a static constructor. It is not allowed to create objects of the static class. Static classes are sealed, means you cannot inherit a static class from another class.

- A **static constructor** is used to initialize any static data, or to perform a particular action that needs to be performed only once. It is called automatically before the first instance is created or any static members are referenced. A static constructor will be called at most once.
- **Static Data Members:** As static class always contains static data members, so static data members are declared using static keyword and they are directly accessed by using the class name. The memory of static data members is allocating individually without any relation with the object.
- **Static Methods:** As static class always contains static methods, so static methods are declared using static keyword and they are directly accessed by using the class name. These methods only access static data members, they cannot access non-static data members.

```
using System;

namespace DemoApplication
{
    static class College
    {
        static string name;
        static string location;

        static College()              //called automatically once
        {
            name = "KITS";
            location = "Coimbatore";
        }
        public static void viewDetails()
        {
            Console.WriteLine("College Name: " + name);
            Console.WriteLine("College Location: " + location);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            College.viewDetails();
            Console.ReadKey();
        }
    }
}
```



```
College Name: KITS
College Location: Coimbatore
```

Constraints:

1. It is not allowed to create objects of the static class.

   For Example in the previous program

```
class Program
 {
   static void Main(string[] args)
   {
    College obj = new College();  //Error!!! Cannot create object
    Console.ReadKey();
   }
 }
```

2. It is not allowed to define non static data or method inside static class.

   For example: The following static class is not valid since the data member and method is not static, which will generate error.

```
static class College
 {
     string name;              //Error
     string location;          //Error
```

```
        public void viewDetails()       //Error
        {
            Console.WriteLine("College Name: " + name);
            Console.WriteLine("College Location: " + location);
        }
    }
```