Abstract Classes

A class declared as abstract can never be instantiated. Instead, only its concrete sub- classes can be instantiated.

If a classis defined as abstract then we can't create an instance of that class. By the creation of the derived class object where an abstract class is inherit from, we can call the method of the abstract class.

For example,

```
abstract class mcn {
    public int add(int a, int b) {
        return (a + b);
    }
}
class mcn1: mcn {
    public int mul(int a, int b) {
        return a * b;
    }
}
class test {
    static void Main(string[] args) {
        mcn1 ob = new mcn1();
        int result = ob.add(5, 10);
        Console.WriteLine("the result is {0}", result);
    }
}
```

Abstract Members

An Abstract method is a method without a body. A derived class does the implementation of an abstract method. When the derived class inherits the abstract method from the abstract class, it must override the abstract method. This requirement is enforced at compile time and is also called dynamic polymorphism. **Abstract members are used to achieve total abstraction.**

The syntax of using the abstract method is as follows:

<access-modifier>abstract<return-type>method name (parameter)

The abstract method is declared by adding the abstract modifier the method.

```
using System;
public abstract class Shape {
        public abstract void draw();
}
public class Rectangle : Shape {
```

Downloaded from Saral Notes Learn something new today. Join Group

```
        public override void draw() {
                Console.WriteLine("Drawing rectangle...");
                }
}
public class TestAbstract {
        public static void Main() {
                Rectangle s = new Rectangle(); s.draw();
        }
}
```

Output:

Drawing rectangle...

Another Example

```
abstract class test1 {
    public int add(int i, int j) {
        return i + j;
    }
    public abstract int mul(int i, int j);
}
class test2: test1 {
    public override int mul(int i, int j) {
        return i * j;
    }
}
class test3: test1 {
    public override int mul(int i, int j) {
        return i - j;
    }
}
class test4: test2 {
    public override int mul(int i, int j) {
        return i + j;
    }
}
class myclass {
    public static void main(string[] args) {
        test2 ob = new test4();
        int a = ob.mul(2, 4);
        test1 ob1 = new test2();
        int b = ob1.mul(4, 2);
        test1 ob2 = new test3();
        int c = ob2.mul(4, 2);
        Console.Write("{0},{1},{2}", a, b, c);
        Console.ReadLine();
    }
}
```

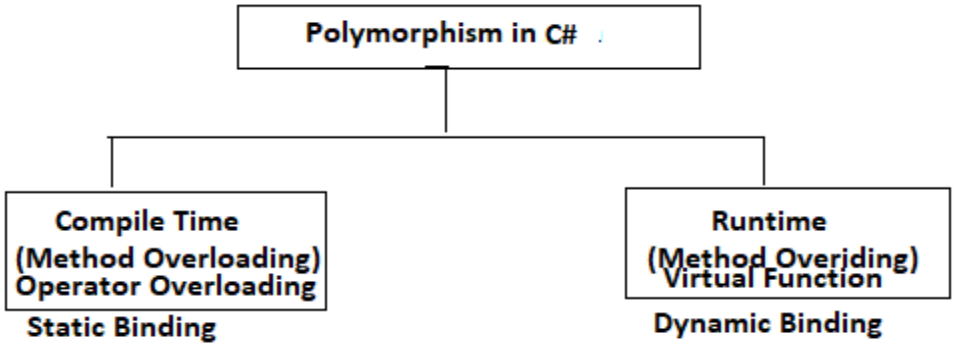
Downloaded from Saral Notes **Learn something new today.** **Join Group**

In the above program, one method i.e. mul can perform various functions depending on the value passed as parameters by creating an object of various classes which inherit other classes. Hence we can achieve dynamic polymorphism with the help of an abstract method.

Polymorphism

Polymorphism in C# is a concept by which we can perform a *single action in different ways.* Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in C# by method overloading and method overriding.



Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in C#, that allows a class to have more than one constructor having different argument lists. In order to overload a method, the argument lists of the methods must differ in either of these:

a) Number of parameters.

add(int, int)

add(int, int, int)

b) Data type of parameters.

add(int, int)

add(int, float)

c) Sequence of Data type of parameters.

add(int, float)

add(float, int)

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Example of Method Overloading

```
public class Methodoveloading
  {
    public int add(int a, int b)  //two int type Parameters method
    {
        return a + b;

    }
    public int add(int a, int b,int c)  //three int type Parameters with same method
    {
        return a + b+c;

    }
    public float add(float a, float b,float c,float d)  //four float type Parameters
    {
        return a + b+c+d;

    }
  }
```

Method Overriding

Method overriding in C# allows programmers to create base classes that allows its inherited classes to override same name methods when implementing in their class for different purpose. This method is also used to enforce some must implement features in derived classes.

Important points:

- Method overriding is only possible in derived classes, not within the same class where the method is declared.
- Base class must use the virtual or abstract keywords to declare a method. Then only can a method be overridden.

Here is an example of method overriding.

```
public class Account {
        public virtual int balance() {
                return 10;
                }
        }
        public class Amount : Account {
                public override int balance() {
                        return 500;
                        }
```

Saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

```
        }
 class Test {
        static void Main() {
        Amount obj = new Amount();
        int balance = obj.balance();
        Console.WriteLine("Balance is: "+balance);
        Console.ReadKey();
        }
}
```

Output:

Balance is 500

Virtual Method

A virtual method is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived the class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overridden in the derived class using the override keyword.

Features of virtual method

- By default, methods are non-virtual. We can't override a non-virtual method.
- We can't use the virtual modifier with the static, abstract, private or override modifiers.
- **If class is not inherited, behavior of virtual method is same as non-virtual method, but in case of inheritance, it is used for method overriding.**

Behavior of virtual method without inheritance – same as non-virtual

```
class Vir {
        public virtual void message() {
                Console.WriteLine("This is test");
                }
        }
class Program {
```

saralnotes.com

```
        static void Main(string[] args) {
                Vir obj = new Vir(); obj.message();
                Console.ReadKey();
                }
}
```

Behavior of virtual method with inheritance – used for overriding

```
class Vir {
        public virtual void message() {
                Console.WriteLine("This is test");
                }
}
class Vir1 : Vir {
        public override void message() {
                Console.WriteLine("This is test1");
        }
}
class Program {
        static void Main(string[] args) {
                Vir1 obj = new Vir1();
                obj.message();
                Console.ReadKey();
        }
}
```

Output:

This is test1

Up casting and Down casting

Upcasting converts an object of a specialized type to a more general type. **An upcast operation creates a base class reference from a subclass reference**.

For example: *Stock msft = new Stock();*

Asset a = msft; // Upcast

Downcasting converts an object from a general type to a more specialized type. **A downcast operation creates a subclass reference from a base class reference.**

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

For example: Stock msft = new Stock();

Asset a = msft; // Upcast

Stock s = (Stock)a; // Downcast


```
BankAccount    ba1,
               ba2 =   new BankAccount("John", 250.0M, 0.01);
LotteryAccount la1,
               la2 =   new LotteryAccount("Bent", 100.0M);

ba1 = la2;                  // upcasting   - OK
//  la1 = ba2;              // downcasting - Illegal
                            //   discovered at compile time
//  la1 = (LotteryAccount)ba2;  // downcasting - Illegal
                            //   discovered at run time
la1 = (LotteryAccount)ba1;  // downcasting - OK
                            //   ba1 already refers to a LotteryAccount
```

The as operator

The as operator performs a downcast that evaluates to null (rather than throwing an exception) if the downcast fails:

Asset a = new Asset();

Stock s = a as Stock; // s is null; no exception thrown

The is operator

The is operator tests whether a reference conversion would succeed; in other words, whether an object derives from a specified class (or implements an inter- face). It is often used to test before downcasting.

if (a is Stock)

Console.WriteLine (((Stock)a).SharesOwned);

Operator Overloading

The concept of overloading a function can also be applied to operators. Operator overloading gives the ability to use the same operator to do various operations. It provides additional capabilities to C# operators when they are applied to user-defined data types. It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.

Saralnotes.in

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Only the predefined set of C# operators can be overloaded. To make operations on a user- defined data type is not as simple as the operations on a built-in data type. To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement. An operator can be overloaded by defining a function to it. The function of the operator is declared by using the operator keyword.

Syntax:

```
access specifier className operator Operator_symbol (parameters) {
        // Code
}
```

The following table describes the overloading ability of the various operators available in C#:

OPERATORS	**DESCRIPTION**
+, -, !, ~, ++, – –	unary operators take one operand and can be overloaded.
+, -, *, /, %	Binary operators take two operands and can be overloaded.
==, !=, =	Comparison operators can be overloaded.
&&, ||	Conditional logical operators cannot be overloaded directly.
+=, -+, *=, /=, %=, =	Assignment operators cannot be overloaded.

saralnotes.com

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Overloading Unary Operators

The following program overloads the **unary - operator** inside the class Complex.

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
    {
    }
    public Complex(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine("{0} {1}", x, y);
    }
    public static Complex operator -(Complex c)
    {
        Complex temp = new Complex();
        temp.x = -c.x;
        temp.y = -c.y;
        return temp;
    }
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex();
        c2.ShowXY(); // displays 0 & 0
        c2 = -c1;
        c2.ShowXY(); // diapls -10 & -20
    }
}
```

Overloading Binary Operators

An overloaded binary operator must take two arguments; at least one of them must be of the type class or struct, in which the operation is defined.

Downloaded from Saral Notes **Learn something new today.** **Join Group**

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
}
public Complex(int i, int j)
{
    x = i;
    y = j;
}
public void ShowXY()
{
    Console.WriteLine("{0} {1}", x, y);
}
public static Complex operator +(Complex c1, Complex c2)
{
    Complex temp = new Complex();
    temp.x = c1.x + c2.x;
    temp.y = c1.y + c2.y;
    return temp;
}
}
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10, 20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex(20, 30);
        c2.ShowXY(); // displays 20 & 30
        Complex c3 = new Complex();
        c3 = c1 + c2;
        c3.ShowXY(); // dislplays 30 & 50
    }
}
```

Downloaded from Saral Notes Learn something new today. Join Group

Sealing Functions and Classes

C# Sealed Class

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, this class cannot be inherited. In C#, the sealed modifier is used to declare a class as sealed. If a class is derived from a sealed class, compiler throws an error.

If you have ever noticed, structs are sealed. You cannot derive a class from a struct.

// Sealed class

sealed class SealedClass{

}

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        SealedClass sealedCls = new SealedClass();
        int total = sealedCls.Add(4, 5);
        Console.WriteLine("Total = " + total.ToString());
    }
}
// Sealed class
sealed class SealedClass
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Sealed Methods and Properties

You can also use the sealed modifier on a method or a property that overrides a virtual method or property in a base class.

This enables you to allow classes to derive from your class and prevent other developers that are using your classes from overriding specific virtual methods and properties.

Downloaded from Saral Notes **Learn something new today.** **Join Group**

```
class X
{
    protected virtual void F()
    {
        Console.WriteLine("X.F");
    }
    protected virtual void F2()
    {
        Console.WriteLine("X.F2");
    }
}
class Y : X
{
    sealed protected override void F()
    {
        Console.WriteLine("Y.F");
    }
    protected override void F2()
    {
        Console.WriteLine("X.F3");
    }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    //
    protected override void F()
    {
         Console.WriteLine("C.F");
    }
    // Overriding F2 is allowed.
    protected override void F2()
    {
        Console.WriteLine("Z.F2");
    }
}
```

The base Keyword

We can use the base keyword to access the fields of the base class within derived class. It is useful if base and derived classes have the same fields.

If derived class doesn't define same field, there is no need to use base keyword. Base class field can be directly accessed by the derived class.

```
using System;
public class Animal{
        public string color = "white";
        }
public class Dog: Animal {
```

Downloaded from Saral Notes **Learn something new today.** **Join Group**

```
        string color = "black";
        public void showColor() {
                Console.WriteLine(base.color); //displays white
                Console.WriteLine(color); //displays black
                }
        }
public class TestBase {
        public static void Main() {
                Dog d = new Dog();
                d.showColor();
        }
}
```

The base keyword has two uses:

- To call a base class constructor from a derived class constructor.
- To call a base class method which is overridden in the derived class.

Calling a base class constructor from a derived class constructor

```
class Base {
        public Base(int a, int b) {
                Console.WriteLine("Value of a={0} and b={1}",a,b);
                }
        }
class Derived : Base {
        public Derived(int x,int y):base(x,y) {
        Console.WriteLine("Value of x={0} and y={1}", x, y);
        }
}
class BaseEx {
        static void Main(){
                new Derived(10,5);
                Console.ReadKey();
        }
}
```

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

Output:

Value of a=10 and b=5

Value of x=10 and y=5

Calling a base class method which is overridden in derived class

```
class Base {
        public virtual void BaseMethod() {
                Console.WriteLine("I am inside base class");
                }
        }
class Derived : Base {
        public override void BaseMethod() {
                base.BaseMethod();
                Console.WriteLine("I am inside derived class");
                }
        }
class BaseEx {
        static void Main(){
                Derived obj = new Derived();
                obj.BaseMethod();
                Console.ReadKey();
        }
}
```

Output:

I am inside base class

I am inside derived class

The object Type

object (System.Object) is the ultimate base class for all types. Any type can be upcast to object.

To illustrate how this is useful, consider a general-purpose stack. A stack is a data structure based on the principle of LIFO—"Last-In First-Out." A stack has two operations: push an object on the stack, and pop an object off the stack.

saralnotes.com

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Here is a simple implementation that can hold up to 10 objects:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)   { data[position++] = obj;  }
  public object Pop()             { return data[--position]; }
}
```

Because Stack works with the object type, we can Push and Pop instances of *any type* to and from the Stack:

Boxing and Unboxing

Boxing is the act of converting a value-type instance to a reference-type instance. The reference type may be either the object class or an interface. In this example, we box an int into an object:

int x = 9;

object obj = x; // Box the int

Unboxing reverses the operation, by casting the object back to the original value type:

int y = (int)obj; // Unbox the int

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type, and throws an InvalidCastException if the check fails.

For instance, the following throws an exception, because long does not exactly match int:

```
object obj = 9;          // 9 is inferred to be of type int
long x = (long) obj;     // InvalidCastException
```

The following succeeds, however:

```
object obj = 9;
long x = (int) obj;
```

As does this:

```
object obj = 3.5;              // 3.5 is inferred to be of type double
int x = (int) (double) obj;    // x is now 3
```

The GetType Method and typeof Operator

All types in C# are represented at runtime with an instance of *System.Type*. There are two basic ways to get a *System.Type* object:

- Call GetType on the instance.
- Use the typeof operator on a type name.

Downloaded from Saral Notes Learn something new today. Join Group

GetType is evaluated at runtime; **typeof** is evaluated statically at compile time (when generic type parameters are involved, it's resolved by the Just-In-Time compiler).

System.Type has properties for such things as the type's name, assembly, base type, and so on. For example:

```
using System;

public class Point { public int X, Y; }

class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine (p.GetType().Name);             // Point
    Console.WriteLine (typeof (Point).Name);          // Point
    Console.WriteLine (p.GetType() == typeof(Point)); // True
    Console.WriteLine (p.X.GetType().Name);           // Int32
    Console.WriteLine (p.Y.GetType().FullName);       // System.Int32
  }
}
```

Structs

A struct is similar to a class, with the following key differences:

- A struct is a value type, whereas a class is a reference type.
- A struct does not support inheritance..

A struct can have all the members a class can, except the following:

- A parameter less constructor.
- Field initializers.
- A finalizer.
- Virtual or protected members.

Here is an example of declaring and calling struct:

Downloaded from Saral Notes **Learn something new today.** **Join Group**

```
public struct Point
{
  int x, y;
  public Point (int x, int y) { this.x = x; this.y = y; }
}

...
Point p1 = new Point ();       // p1.x and p1.y will be 0
Point p2 = new Point (1, 1);   // p1.x and p1.y will be 1
```

The next example generates three compile-time errors:

```
public struct Point
{
  int x = 1;                          // Illegal: field initializer
  int y;
  public Point() {}                   // Illegal: parameterless constructor
  public Point (int x) {this.x = x;}  // Illegal: must assign field y
}
```

Changing `struct` to `class` makes this example legal.

Access Modifiers

Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself. For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.

Access Modifiers	Inside Assembly		Outside Assembly	
	With Inheritance	With Type	With Inheritance	With Type
Public	✓	✓	✓	✓
Private	X	X	X	X
Protected	✓	X	✓	X
Internal	✓	✓	X	X
Protected Internal	✓	✓	✓	X

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

In C# there are 6 different types of Access Modifiers.

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Modifier	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified.
protected	Access is limited to within the class definition and any class that inherits from the class.
internal	Access is limited exclusively to classes defined within the current project assembly.
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                 // Class1 is internal (default)
public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private (default)
class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
{
  void Foo()           {}        // Foo is private (default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }       // Error - cannot access Foo
  void Test2() { Bar(); }       // OK
}
```

Downloaded from Saral Notes Learn something new today. Join Group

Restrictions on Access Modifiers

```
class BaseClass             { protected virtual  void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} }  // OK
class Subclass2 : BaseClass { public    override void Foo() {} }  // Error
```

(An exception is when overriding a `protected internal` method in another assembly, in which case the override must simply be `protected`.)

The compiler prevents any inconsistent use of access modifiers. For example, a subclass itself can be less accessible than a base class, but not more:

```
internal class A {}
public class B : A {}          // Error
```

Enums in C#

Enum in C# language is a value type with a set of related named constants often referred to as an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type, which is user-defined. Enums type can be an integer (float, int, byte, double etc.) but if you use beside int, it has to be cast.

Enum is used to create numeric constants in .NET framework. All member of the enum are of enum type. There must be a numeric value for each enum type.

The default underlying type of the enumeration elements is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

```
// making an enumerator 'month'
enum month
{

    // following are the data members
    jan,
    feb,
    mar,
    apr,
    may

}

class Program {

    // Main Method
    static void Main(string[] args)
    {

        // getting the integer values of data members..
        Console.WriteLine("The value of jan in month " +
                          "enum is " + (int)month.jan);
        Console.WriteLine("The value of feb in month " +
                          "enum is " + (int)month.feb);
        Console.WriteLine("The value of mar in month " +
                          "enum is " + (int)month.mar);
        Console.WriteLine("The value of apr in month " +
                          "enum is " + (int)month.apr);
        Console.WriteLine("The value of may in month " +
                          "enum is " + (int)month.may);
    }
}
```

Downloaded from Saral Notes Learn something new today. Join Group

Output

The value of jan in month enum is 0
The value of feb in month enum is 1
The value of mar in month enum is 2
The value of apr in month enum is 3
The value of may in month enum is 4

Generics

Generics in C# and .NET procedure many of the benefits of strongly-typed collections as well as provide a higher quality of and a performance boost for code.

Generics are very similar to C++ templates but having a slight difference in such a way that the source code of C++ templates is required when a template is instantiated with a specific type and .NET Generics are not limited to classes only. In fact, they can also be implemented with Interfaces, Delegates and Methods.

The detailed specification for each collection is found under the System.Collection.Generic **namespace.**

Generic Classes

The Generic class can be defined by putting the <T> sign after the class name. It isn't mandatory to put the "T" word in the Generic type definition. You can use any word in the TestClass<> class declaration.

```
public class TestClass<T> { }
```

The **System.Collection.Generic** namespace also defines a number of classes that implement many of these key interfaces. The following table describes the core class types of this namespace.

Generic class	Description
Collection<T>	The basis for a generic collection Comparer compares two generic objects for equality.
Dictionary<TKey, TValue>	A generic collection of name/value pairs.
List<T>	A dynamically resizable list of Items.
Queue<T>	A generic implementation of a first-in, first-out (FIFO) list.
Stack<T>	A generic implementation of a last in, first-out (LIFO) list.

Downloaded from Saral Notes Learn something new today. Join Group

```
using System.Collections.Generic;
class Test<T> {
        T[] t=new T[5];
        int count = 0;
        public void addItem(T item) {
                if (count < 5) {
                        t[count] = item;
                        count++;
                        } else {
                                Console.WriteLine("Overflow exists");
                        }
                }
        public void displayItem() {
                for (int i = 0; i < count; i++) {
                        Console.WriteLine("Item at index {0} is {1}",i,t[i]);
                        }
                }
  }
class GenericEx {
        static void Main() {
                Test<int> obj = new Test<int>();
                obj.addItem(10);
                obj.addItem(20);
                obj.addItem(30);
                obj.addItem(40);
                obj.addItem(50);
                //obj.addItem(60); //overflow exists
                obj.displayItem();
                Console.ReadKey();
        }
}
```

Output

Item at index 0 is 10

Item at index 1 is 20

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

Item at index 2 is 30
Item at index 3 is 40
Item at index 4 is 50

Generic Methods

The objective of this example is to build a swap method that can operate on any possible data type (value-based or reference-based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference via ref keyword.

using System.Collections.Generic;

```
class Program
{
    //Generic method
    static void Swap<T>(ref T a, ref T b)
    {
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
    static void Main(string[] args)
    {
        // Swap of two integers.
        int a = 40, b = 60;
        Console.WriteLine("Before swap: {0}, {1}", a, b);

        Swap<int>(ref a, ref b);

        Console.WriteLine("After swap: {0}, {1}", a, b);

        Console.ReadLine();
    }
}
```

Output

Before swap: 40, 60
After swap: 60, 40

Dictionary

Dictionaries are also known as maps or hash tables. It represents a data structure that allows you to access an element based on a key. One of the significant features of a dictionary is faster lookup; you can add or remove items without the performance overhead.

Downloaded from Saral Notes **Learn something new today.** **Join Group**

.Net offers several dictionary classes, for instance **Dictionary<TKey, TValue>.** The type parameters TKey and TValue represent the types of the keys and the values it can store, respectively.

```
using System.Collections.Generic;
public class Program {
        static void Main(string[] args) {
        //define Dictionary collection
        Dictionary<int, string> dObj = new Dictionary<int, string>(5);
        //add elements to Dictionary
        dObj.Add(1, 1, "Tom");
        dObj.Add(2, "John");
        dObj.Add(3, "Maria");
        dObj.Add(4, "Max");
        dObj.Add(5, "Ram");

        //print data
        for (int i = 1; i <= dObj.Count; i++) {
                Console.WriteLine(dObj[i]);
        }
        Console.ReadKey();
        }
}
```

Queues

Queues are a special type of container that ensures the items are being accessed in a FIFO (first in, first out) manner. Queue collections are most appropriate for implementing messaging components. We can define a Queue collection object using the following syntax:

Queue qObj = new Queue();

The Queue collection property, methods and other specification definitions are found under the System.Collection namespace. The following table defines the key members;

System.Collection.Queue Members	Definition
Enqueue()	Add an object to the end of the queue.
Dequeue()	Removes an object from the beginning of the queue.

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

Peek()	Return the object at the beginning of the queue without removing it.

```
using System.Collections.Generic;
class Program {
        static void Main(string[] args) {
        Queue < string > queue1 = new Queue < string > ();
        queue1.Enqueue("MCA");
        queue1.Enqueue("MBA");
        queue1.Enqueue("BCA");
        queue1.Enqueue("BBA");

        Console.WriteLine("The elements in the queue are:");
        foreach(string s in queue1) {
                Console.WriteLine(s);
        }
        queue1.Dequeue(); //Removes the first element that enter in the queue here the first element is MCA
        queue1.Dequeue(); //Removes MBA
        Console.WriteLine("After removal the elements in the queue are:");
        foreach(string s in queue1) {
                Console.WriteLine(s);
                }
        }
}
```

Stacks

A Stack collection is an abstraction of LIFO (last in, first out). We can define a Stack collection object using the following syntax:

Stack qObj = new Stack();

The following table illustrates the key members of a stack;

System.Collection.Stack Members	Definition
Contains()	Returns true if a specific element is found in the collection.

saralnotes.com

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Clear()	Removes all the elements of the collection.
Peek()	Previews the most recent element on the stack.
Push()	It pushes elements onto the stack.
Pop()	Return and remove the top elements of the stack.

```
class Program {
        static void Main(string[] args) {
        Stack < string > stack1 = newStack < string > ();
        stack1.Push("************");
        stack1.Push("MCA");
        stack1.Push("MBA");
        stack1.Push("BCA");
        stack1.Push("BBA");
        stack1.Push("************");
        stack1.Push("**Courses**");
        stack1.Push("************");
        Console.WriteLine("The elements in the stack1 are as:");
        foreach(string s in stack1) {
                Console.WriteLine(s);
        }

        //For remove/or pop the element pop() method is used
        stack1.Pop();
        stack1.Pop();
        stack1.Pop();
        Console.WriteLine("After removal/or pop the element the stack is as:");
        //the element that inserted in last is remove firstly.
        foreach(string s in stack1) {
                Console.WriteLine(s);
                }
            }
        }
```

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

List

List<T> class in C# represents a strongly typed list of objects. List<T> provides functionality to create a list of objects, find list items, sort list, search list, and manipulate list items. In List<T>, T is the type of objects.

Adding Elements

```
// Dynamic ArrayList with no size limit
List<int> numberList = new List<int>();
numberList.Add(32);
numberList.Add(21);
numberList.Add(45);
numberList.Add(11);
numberList.Add(89);

// List of string
List<string> authors = new List<string>(5);
authors.Add("Mahesh Chand");
authors.Add("Chris Love");
authors.Add("Allen O'neill");
authors.Add("Naveen Sharma");
authors.Add("Monica Rathbun");
authors.Add("David McCarter");

// Collection of string
string[] animals = { "Cow", "Camel", "Elephant" };

// Create a List and add a collection
List<string> animalsList = new List<string>();
animalsList.AddRange(animals);
foreach (string a in animalsList)
        Console.WriteLine(a);
```

saralnotes.com

Downloaded from Saral Notes **Learn something new today.** **Join Group**

Remove Elements

```
// Remove an item
authors.Remove("New Author1");

// Remove 3rd item
authors.RemoveAt(3);

// Remove all items
authors.Clear();
```

Sorting

```
authors.Sort();
```

Other Methods

```
authors.Insert(1,"Shaijal"); //insert item at index 1
authors.Count; //returns total items
```

Array List

C# ArrayList is a non-generic collection. The ArrayList class represents an array list and it can contain elements of any data types. The ArrayList class is defined in the System.Collections namespace. An ArrayList is dynamic array and grows automatically when new items are added to the collection.

ArrayList personList = new ArrayList();

Insertion

```
personList.Add("Sandeep");
```

Removal

```
// Remove an item
personList.Remove("New Author1");

// Remove 3rd item
personList.RemoveAt(3);
```

saralnotes.com

Downloaded from Saral Notes Learn something new today. Join Group

```
// Remove all items
personList.Clear();
```

Sorting

```
personList.Sort();
```

Other Methods

```
personList.Insert(1,"Dipendra"); //insert item at index 1
personList.Count; //returns total items
```

saralnotes.com

Downloaded from Saral Notes **Learn something new today.** **Join Group**