OOPS with C#:

Classes and Objects:

Classes create objects

and objects use methods to communicate between them as shown in Fig. 12.1. This is object-oriented programming (OOP).

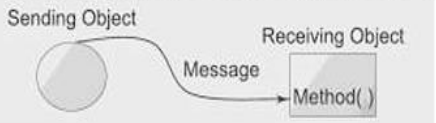


Fig. 12.1 *Measage passing between objects*

Classes provide a convenient approach for packing together a group of logically related data items and functions that work on them. In C#, the data items are called *fields* and the functions are called *methods*. Calling a specific method in an object is described as sending the object a message.

Basic principles of OOP:

All object-oriented languages employ three core principles, namely,

- encapsulation,
- inheritance, and
- polymorphism.

Encapsulation provides the ability to hide the internal details of an object from its users. The outside user may not be able to change the state of an object directly. However, the state of an object may be altered indirectly using what are known *accessor* and *mutator* methods. In C#, encapsulation is implemented using the access modifier keywords ***public, private***, and ***protected***

Inheritance is the concept we use to build new classes using the existing class definitions. Through inheritance we can modify a class the way we want to create new objects. The original class is known as *base* or *parent* class and the modified one is known as *derived class* or *subclass* or *child class*.

Polymorphism is the third concept of OOP. It is the ability to take more than one form. For example, an operation may exhibit different behaviour in different situations. The behaviour depends upon the types of data used in the operation.

Defining a Class:

Once the class type has been defined, we can create 'variables' of that type using declarations that are similar to the basic type declarations. In C#, these variables are termed as *instances* of classes, which are the actual *objects*. The basic form of a class definition is :

```
class classname
{
        [ variables declaration; ]
        [ methods declaration; ]
}
```

class is a keyword and *classname* is any valid C# identifier. Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class Empty        //class name is Empty
{
}
```

Member access modifiers:

One of the goals of object-oriented programming is 'data hiding'. That is, a class may be designed to hide its members from outside accessibility. C# provides a set of 'access modifiers' that can be used with the members of a class to control their visibility to outside users. Table 12.1 lists various access modifiers provided by C# and their visibility control. These modifiers are a part of C# keywords.

Table 12.1 *C# access modifiers*

Modifier	*Accessibility Control*
private	Member is accessible only within the class containing the member.
public	Member is accessible from anywhere outside the class as well. It is also accessible in derived classes.
protected	Member is visible only to its own class and its derived classes.
internal	Member is available within the assembly or component that is being created but not to the clients of that component.
protected internal	Available in the containing program or assembly and in the derived classes.

In C#, all members have **private** access by default. If we want a member to have any other visibility range, then we must specify a suitable access modifier to it individually. *Example:*

```
class Visibility
{
    public int x;
    internal int y;
    protected double d;
     float p;        //private by default
}
```

Creating Objects:

Objects in C# are created using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type **Rectangle**.

```
Rectangle rect1 ;          // declare
rect1 = new Rectangle( ); // instantiate
```

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable **rect1** is now an object of the **Rectangle** class. (See Fig. 12.3).

Both statements can be combined into one as shown below:

```
Rectangle rect1 = new Rectangle( );
```

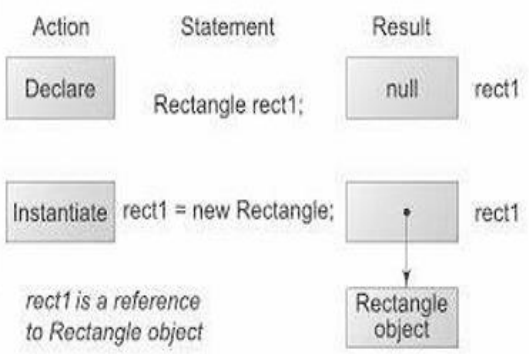


Fig. 12.3 *Creating object references*

Constructors:

Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even **void**. This is because they do not return any value.

Program 12.2 | APPLICATION OF CONSTRUCTORS

```
using System;
class Rectangle
{
    public int length, width ;
    public Rectangle(int x, int y) // Defining constructor
    {
        length = x ;
        width = y ;
    }
    public int RectArea( )
    {
        return (length * width);
    }
}
class RectangleArea
{
    public static void Main( )
    {
        Rectangle rect1 = new Rectangle(15,10); // Calling constructor
      int area1 = rect1.RectArea( ) ;
      Console.WriteLine("Area1 = "+ area1) ;
   }
}
```

Overloaded Constructors:

To create an overloaded constructor method, all we have to do is to provide several different constructor definitions with different parameter lists.

```
class Room
{
    public double length ;
    public double breadth ;
    public Room(double x, double y)        // constructor1
    {
        length = x ;
        breadth = y ;
    }
    public Room(double x)                  // constructor2
    {
        length = breadth = x ;
    }
    public int Area( )
    {
        return (length * breadth) ;
    }
}
```

Destructors:

A destructor is opposite to a constructor. It is a method called when an object is no more required. The name of the destructor is the same as the class name and is preceded by a tilde (~). Like constructors, a destructor has no return type. *Example:*

```
class Fun
{
    . . . .
    . . . .
    ~ Fun ( ) //No arguments
    {
    . . . .
    }
    }
```

Inheritance and Polymorphism:

C# classes can be reused in several ways. Reusability is achieved by designing new classes, reusing all or some of the properties of existing ones. The mechanism of designing or constructing one class from another is called *inheritance*. This may be achieved in two different forms.

- Classical form
- Containment form

Classical Inheritance:

Inheritance represents a kind of relationship between two classes. Let us consider two classes **A** and **B**. We can create a class hierarchy such that **B** is derived from **A** as shown in Fig. 13.1.

Class **A**, the initial class that is used as the basis for the derived class is referred to as the *base class*, *parent class* or *superclass*. Class **B**, the derived class, is referred to as *derived class, child class* or *sub class*. A derived class is a completely new class that incorporates all the data and methods of its base class. It can also have its own data and method members that are unique to itself. That is, it can enhance the content and behaviour of the base class.

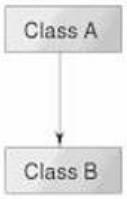


Fig.13.1 *Simple inheritance*

We can now create objects of classes A and B independently. *Example:*

```
A a; // a is object of A
B b; // b is object of B
```

In such cases, we say that the object **b** is a type of **a.** Such relationship between **a** and **b** is referred to as 'is-a' relationship. Examples of is-a relationship are:

- Dog is-a type of animal
- Manager is-a type of employee
- Ford is-a type of car

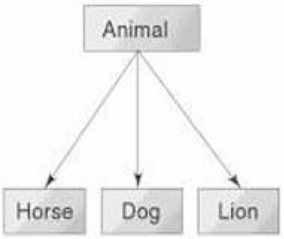
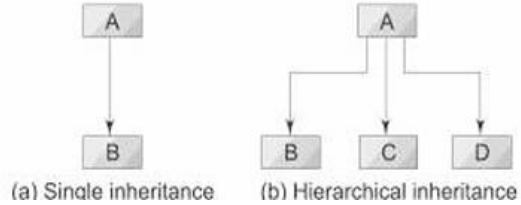


Fig.13.2 *The is-a resistance*

The is-a relationship is illustrated in Fig. 13.2.

The classical inheritance may be implemented in different combinations as illustrated in Fig.13.3.



(a) Single inheritance (b) Hierarchical inheritance

They include:

- Single inheritance (only one base class)
- Multiple inheritance (several base classes)
- Hierarchical inheritance (one base class, many subclasses)
- Multilevel inheritance (derived from a derived class)

(c) Multilevel inheritance (d) Multiple inheritance

Containment Inheritance:

We can also define another form of inheritance relationship known as *containership* between class A and B. *Example:*

```
class A
{
    . . . .
}
class B
{
    . . . .
    A a; // a is contained in b
}
B b;
. . . .
```

In such cases, we say that the object **a** is contained in the object **b.** This relationship between **a** and **b** is referred to as 'has–a'relationship. The outer class **B** which contains the inner class **A** is termed the 'parent' class and the contained class A is termed a 'child' class. Examples are:

- Car has-a radio
- House has-a store room
- City has-a road

The has-a relationship is illustrated in Fig.13.4.

Fig.13.4 *The has-a relationship*

Mutilevel Inheritance:

A common requirement in object-oriented programming is the use of a derived class as a superclass. C# supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes as shown in Fig. 13.5.

The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C**. The chain **ABC** is known as *inheritance path.*

A derived class with multilevel base classes is declared as follows:

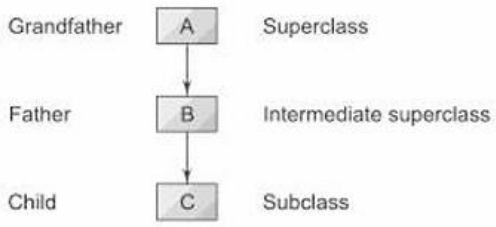


Fig. 13.5 *Multilevel inheritance*

```
class A
{
    . . . .
    . . . .
}
class B : A // First level derivation
{
    . . . .
    . . . .
}
class C : B // Second level derivation
{
    . . . .
```

```
    . . . .
}
```

This process may be extended to any number of levels. The class **C** can inherit the members of both **A** and **B** as shown in Fig. 13.6.

As discussed earlier, the constructors are executed from the top downwards, with the bottom most class constructor being executed last. *Example:*

```
class A
{
    protected int a;
    public A (int x)
{
        a = x;
    }
}
class B : A
{
    protected int b;
    public B (int x, int y ) : base (x)
    {
        b = y;
    }
}
class C : B
{
    int c ;
    public C (int x, int y, int z) : base (x, y)
      {
          c = z;
      }
  }
```

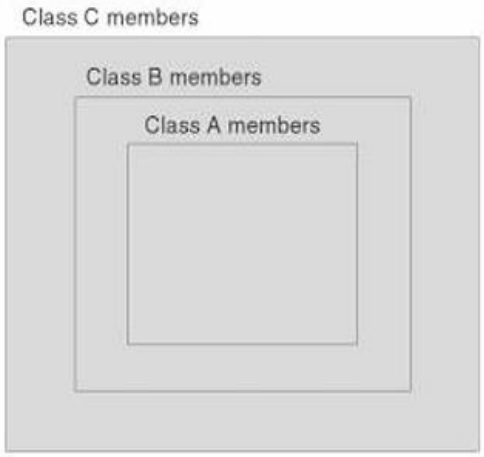


Fig. 13.6 C contains B which contains A

Sealed Classes: Preventing inheritance

Sometimes, we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called a *sealed class*. This is achieved in C# using the modifier **sealed** as follows:

```
sealed class Aclass
{
    . . . .
    . . . .
}
sealed class Bclass: Someclass
{
    . . . .
    . . . .
}
```

Sealed Methods:

When an instance method declaration includes the **sealed** modifier, the method is said to be a *sealed method.* It means a derived class cannot override this method.

A sealed method is used to override an inherited virtual method with the same signature. That means, the **sealed** modifier is always used in combination with the **override** modifier. *Example:*

```
class A
{
    public virtual void Fun( )
    {
    . . . .
    }
}
class B : A
{
    public sealed override void Fun ( )
    {
    . . . .
    }
}
```

Polymorphism:

polymorphism means 'one name, many forms'.

13.15.1 Operation Polymorphism

Operation polymorphism is implemented using overloaded methods and operators. We have already used the concept of overloading while discussing methods and constructors. The overloaded methods are 'selected' for invoking by matching arguments, in terms of number, type and order. This information is known to the compiler at the time of compilation and, therefore, the compiler is able to select and bind the appropriate method to the object for a particular call at *compile time* itself. This process is called *early binding,* or *static binding*, or *static linking*. It is also known as *compile time polymorphism.*

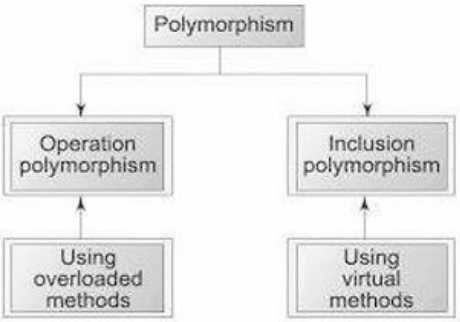


Fig. 13.8 *Achieving polymorphism*

Early binding simply means that an object is bound to its

Program 13.6 | **OPERATION POLYMORPHISM**

```
using System;
class Dog
{
}
class Cat
{
}
class Operation
{
    static void Call (Dog d)
    {
        Console.WriteLine ("Dog is called");
    }
    static void Call (Cat c)
    {
        Console.WriteLine (" Cat is called ");
    }
```

```
    public static void Main( )
    {
        Dog dog = new Dog( );
        Cat cat = new Cat ( );
        Call(dog); //invoking Call( )
        Call(cat); //again invoking Call( )
    }
}
```

Inclusion Polymorphism:

Inclusion polymorphism is achieved through the use of virtual functions. Assume that the class **A** implements a **virtual** method **M** and classes **B** and **C** that are derived from **A** override the virtual method **M.** When **B** is cast to **A**, a call to the method **M** from **A** is dispatched to **B**. Similarly, when **C** is cast to **A**, a call to **M** is dispatched to **C**.

Program 13.7 | **INCLUSION POLYMORPHISM**

```
using System;
class Maruthi
{
    public virtual void Display ( ) //virtual method
    {
        Console.WritLine("Maruthi car");
    }
}
class Esteem : Maruthi
{
    public override void Display( )
    {
        Console.WriteLine("Maruthi Esteem");
    }
}
class Zen : Maruthi
{
    public override void Display ( )
    {
        Console.WriteLine("Maruthi Zen");
    }
}
class Inclusion
{
    public static void Main( )
    {
        Maruthi m = new Maruthi ( );
        m = new Esteem ( );       //upcasting
        m.Display ( );
        m = new Zen ( );          //upcasting
        m.Display ( )
    }
}
```

Interfaces:

An interface in C# is a reference type. It is basically a kind of class with some differences. Major differences include:

- All the members of an interface are implicitly **public** and **abstract.**
- An interface cannot contain constant fields, constructors and destructors.
- Its members cannot be declared **static.**
- Since the methods in an interface are abstract, they do not include implementation code.
- An interface can inherit multiple interfaces.

Defining an Interface:

The syntax for defining an interface is very similar to that used for defining a class. The general form of an interface definition is:

```
interface        InterfaceName
{
    Member declarations;
}
```

Extending an Interface:

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved as follows:

```
interface    name2 : name1
{

    Members of name2
}
```

Consider the code below:

```
interace Addition
{
    int Add (int x, int y) ;
}

interface Compute : Addition
{
    int Sub (int x, int y);
}
```

The interface **Compute** will have both the methods and any class implementing the interface **Compute** should implement both of them; otherwise, it is an error.

We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface I1
{
    ....
}
```

```
interface I2
{
    . . . .
}
interface I3 : I1, I2 //multiple inheritance
{
    . . . .
}
```

Implementing an Interfaces:

Interfaces are used as 'superclasses' whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname : interfacename
{
    body of classname
}
```

Here the class **classname** 'implements' the interface **interfacename.** A more general form of implementation may look like this:

```
class classname : superclass, interface1, interface2. . . .
{
    body of classname
}
```

Program 14.1 | IMPLEMENTATION OF MULTIPLE INTERFACES

```
using System;
interface Addition
{
    int Add ( );
}
interface Multiplication
{
    int Mul ( );
}
class Computation  : Addition, Multiplication
{
    int x, y;
    public Computation (int x, int y )              //Constructor
    {
            this.x = x;
            this.y = y;
    }
    public int Add ( )                              //Implement Add ( )
    {
            return ( x + y );
    }
    public int Mul ( )                              //Implement Mul ( )
    {
            return ( x * y );
    }
}
class InterfaceTest1
{
    public static void Main( )
    {
            Computation com = new Computation (10,20);
            Addition add = (Addition ) com;                         // casting
            Console.WriteLine ("Sum = " + add.Add ( ));
            Multiplication mul = (Multiplication) com;              // casting
            Console.WriteLine("Product = " + mul.Mul ( ) );
```

}}

Interfaces and Inheritance:

Program 14.3 | INHERITING A CLASS THAT IMPLEMENTS AN INTERFACE

```
using System;
interface Display
{
    void Print ( );
}
class B : Display    // implements Display
{
    public void Print ( )
    {
        Console.WriteLine("Base Display");
    }
}
class D : B        // inherits B class
{
    public new void Print ( )
    {
        Console.WriteLine("Derived Display");
    }
}
class InterfaceTest3
{
    public static void Main( )
    {
        D d = new D ( );
        d.Print ( );
        Display dis = (Display) d;
        dis.Print ( );
    }
}
```

Program 14.3 would produce the following output:

```
Derived Display
Base Display
```

Operator Overloading:

Operator overloading is one of the many exciting features of object-oriented programming. C# supports the idea of operator overloading. It means that C# operators can be defined to work with the user-defined data types such as structs and classes in much the same way as the built-in types.

Overloadable Operators:

Table 15.1 *Overloadable operators*

Category	*Operators*
Binary arithmetic	+, *, /, -, %
Unary arithmetic	+, -, ++, --
Binary bitwise	&, |, ^, <<, >>
Unary bitwise	!, ~, true, false
Logical operators	==, !=, >=, <, <=, >

Table 15.2 *Operators that cannot be overloaded*

CATEGORY	OPERATORS
Conditional operators	& &, ||
Compound assignment	+=, -=, *=, /=, %=,
Other operators	[ ], ( ), =, ? :, ->, new, sizeof, typeof, is, as

Need for operator Overloading:

Although operator overloading gives us syntactical convenience, it also help us greatly to generate more readable and intuitive code in a number of situations. These include:

- Mathematical or physical modeling where we use classes to represent objects such as co-ordinates, vectors, matrices, tensors, complex numbers and so on.
- Graphical programs where co-ordinate-related objects are used to represent positions on the screen.
- Financial programs where a class represents an amount of money.
- Text manipulations where classes are used to represent strings and sentences.

Defining Operator Overloading:

The operator is defined in much the same way as a method, except that we tell the compiler it is actually an operator we are defining by the **operator** keyword, followed by the operator symbol *op*. The key features of operator methods are:

- They must be defined as **public** and **static.**
- The *retval* (return value) type is the type that we get when we use this operator. But, technically, it can be of any type.
- The *arglist* is the list of arguments passed. The number of arguments will be one for the unary operators and two for the binary operators.
- In the case of unary operators, the argument must be the same type as that of the enclosing class or struct.
- In the case of binary operators, the first argument must be of the same type as that of the enclosing class or struct and the second may be of any type.

Overloading Unary Operators:

The method **operator –( )** takes one argument of type **Space** and changes the sign of data members of the object **s.** Since it is a member method of the same class, it can directly access the members of the object which activated it. Remember, a statement like

```
s2 = -s1;
```

will not work, because the operator method does not return any value. It can work if the method is modified to return an object.

Program 15.1 | OVERLOADING UNARY MINUS

```
using System;
class Space
{

    int x, y, z;
    public Space ( int a, int b, int c )
    {

        x = a;
        y = b;
```

```
            z = c;
        }
        public void Display ( )
        {
            Console.Write(" " + x);
            Console.Write(" " + y);
            Console.Write(" " + z);
            Console.WriteLine( );

        }
        public static Space operator - (Space s)
        {
            s.x = -s.x;
            s.y = -s.y;
            s.z = -s.z;
        }
}

class SpaceTest
{
        public static void Main( )
        {
            Space s = new Space ( 10, -20, 30 );
            Console.Write(" s : ");
            s.Display( );

            -s;                 //activates opeator -( ) method

            Console.Write(" s : ");
            s.Display ( );
        }
}
```

Program 15.1 will produce the following output:

```
S : 10 -20 30
S : -10 20 -30
```

Overloading Binary Operators:

We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator. We can always use methods to add two objects. A statement like

```
C = sum ( A, B); //functional notation
```

is possible. The functional notation can be replaced by a natural looking expression

```
C = A + B; //arithmetic notation.
```

by overloading the + operator using an **operator + ( )** method. Here is the syntax used to define the **operator + ( )** method.

```
public static Vector operator + (Vector u1, Vector u2)
    {
        //Create a new Vector object

        //Add the contents of u1 and u2
        //to the new Vector object
        //Return the new vector object
    }
```

Overloading Comparison Operators:

C# supports six comparison operators that can be considered in three pairs:

- = = and ! =
- > and > =
- < and < =

The significance of pairing is two-fold.

1. Within each pair, the second operator should always give exactly the opposite result to the first. That is, whenever the first returns **true**, the second returns **false** and vice versa.
2. C# always requires us to overload the comparison operators in pairs. That is, if we overload = =, then we must overload ! = also, otherwise it is an error.

Delegates and Events:

A delegate object is a special type of object that contains the details of a method rather than data. Delegates in C# are used for two purposes:

- Callback
- Event handling

Delegates:

The dictionary meaning of **delegate** is "a person acting for another person". In C#, it really means a method acting for another method. As pointed out earlier, a delegate in C# is a class type object and is used to invoke a method that has been encapsulated into it at the time of its creation. Creating and using delegates involve four steps. They include:

- Delegate declaration
- Delegate methods definition
- Delegate instantiation
- Delegate invocation

A delegate declaration defines a class using the class **System.Delegate** as a base class. Delegate

Delegate Declaration:

A delegate declaration is a type declaration and takes the following general form:

modifier **delegate** *return-type delegate-name (parameters*);

delegate is the keyword that signifies that the declaration represents a class type derived from **System. Delegate**. The *return-type* indicates the return type of the delegate. *Parameters* identifies the signature of the delegate. The *delegate-name* is any valid C# identifier and is the name of the delegate that will be used to instantiate delegate objects.

That is, a delegate may be defined in the following places:

- Inside a class
- Outside all classes
- As the top level object in a namespace

Delegate Methods:

The methods whose references are encapsulated into a delegate instance are known as *delegate methods* or *callable entities*. The signature and return type of delegate methods must exactly match the signature and return type of the delegate.

One feature of delegates, as pointed out earlier, is that they are type-safe to the extent that they ensure the matching of signatures of the delegate methods. However, they do not care

- what type of object the method is being called against, and
- whether the method is a static or an instance method.

Delegate Instantiation:

Although delegates are of class types and behave like classes, C# provides a special syntax for instantiating their instances. A *delegate-creation-expression* is used to create a new instance of a delegate.

```
new delegate-type (expression)
```

Delegate Invocation:

C# uses a special syntax for invoking a delegate. When a delegate is invoked, it in turn invokes the method whose reference has been encapsulated into the delegate, (only if their signatures match). Invocation takes the following form:

```
delegate_object (parameters list )
```

EVENTS:

An *event* is a delegate type class member that is used by the object or class to provide a notification to other objects that an event has occurred. The client object can act on an event by adding an *event handler* to the event.

Events are declared using the simple *event declaration* format as follows:

```
modifier event type event-name;
```

Errors and Exceptions:

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible errors and error conditions in the program so that they do not terminate or cause the system to crash during execution.

Types of Errors:

Errors may be broadly classified into two categories:

- Compile-time errors
- Run-time errors

Compile time errors:

All syntax errors will be detected and displayed by the C# compiler and therefore these errors are known as *compile-time errors*.

Program 18.1 | ILLUSTRATION OF COMPILE-TIME ERRORS

```
/* This program contains an error */
using Sytem;
class Error1
{
    public static void Main()
    {
        Console.WriteLine("Hello C#!");
    }
}
```

Run time errors:

Most common run-time errors are:

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Passing a parameter that is not in a valid range or value for a method.
- Attempting to use a negative size for an array.
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting an invalid string to a number or vice versa.
- Accessing a character that is out of bounds of a string, and so on.

Program 18.2 | ILLUSTRATION OF RUN-TIME ERRORS

```
using System;
class Error2
{
    public static void Main( )
    {
        int a = 10;
        int b = 5;
        int c = 5;

        int x = a/(b-c);
        Console.WriteLine("x = " + x);
        int y = a/(b+c);
        Console.WriteLine("y = " + y);
    }
}
```

Exceptions:

An *exception* is a condition that is caused by a run-time error in the program. When the C# compiler encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred).

The purpose of the exception handling mechanism is to provide a means to detect and report an 'exceptional circumstance' so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

- Find the problem (***Hit*** the exception)
- Inform that an error has occurred (***Throw*** the exception)
- Receive the error information (***Catch*** the exception)
- Take corrective actions (***Handle*** the exception)

Table 18.1 *Common C# exceptions*

Exception Class	*Cause of Exception*
SystemException	A failed run-time check; used as a base class for other exceptions
AccessException	Failure to access a type member, such as a method or field
ArgumentException	An argument to a method was invalid
ArgumentNullException	A null argument was passed to a method that does not accept it
ArgumentOutofRangeException	Argument value is out of range
ArithmeticException	Arithmetic over-or underflow has occurred
ArrayTypeMismatchException	Attempt to store the wrong type of object in an array
BadImageFormatException	Image is in the wrong format
CoreException	Base class for exceptions thrown by the runtime
DivideByZeroException	An attempt was made to divide by zero
FormatException	The format of an argument is wrong
IndexOutofRangeException	An array index is out of bounds
InvalidCastException	An attempt was made to cast to an invalid class
InvalidOperationException	A method was called at an invalid time
MissingMemberException	An invalid version of a DLL was accessed
NotFiniteNumberException	A number is not valid
NotSupportedException	Indicates that a method is not implemented by a class
NullReferenceException	Attempt to use an unassigned reference
OutofMemoryException	Not enough memory to continue execution
StackOverflowException	A stack has overflowed

Syntax of Exception Handling:

The basic concepts of exception handling are *throwing* an exception and *catching* it. This is illustrated in Fig. 18.1.

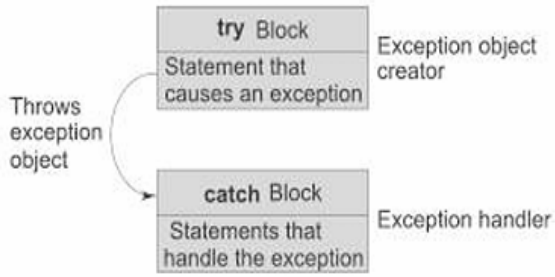


Fig. 18.1 *Exception handling mechanism*

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every **try** statement should be followed by at least one **catch** statement; otherwise compilation error will occur.

Program 18.3 | USING TRY AND CATCH FOR EXCEPTION HANDLING

```
using System;
class Error3
{
    public static void Main( )
    {
        int a = 10;
        int b = 5;
        int c = 5;
        int x, y ;
        try
        {
            x = a / (b-c);          / / Exception here
        }
            catch (Exception e)
        {
            Console.WriteLine("Division by zero");
        }
        y = a / (b+c);
        Console.WriteLine("y = " + y);
    }
}
```

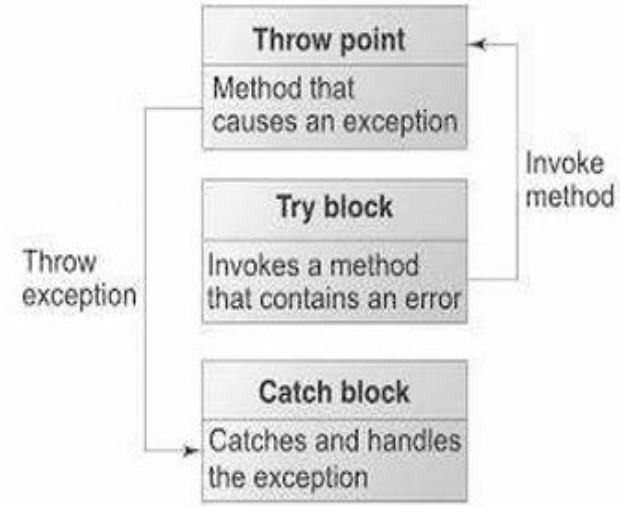


Fig. 18.2 *Invoking a method that contain exceptions*

Multiple catch statements:

It is possible to have more than one catch statement in the catch block as illustrated below:

```
.....
.....
try
{
    statement ;          / / generates an exception
}
catch (Exception-Type-1 e)
{
    statement;           / / processes exception type 1
}
catch (Exception-Type-2 e)
{
    statement;           / / processes exception type 2
}
.
.
.
catch (Exception-type-N e)
{
    statement ;          / / processes exception type N
}
```

When an exception in a **try** block is generated, the C# treats the multiple **catch** statements like cases in a **switch** statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

Note that C# does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example:

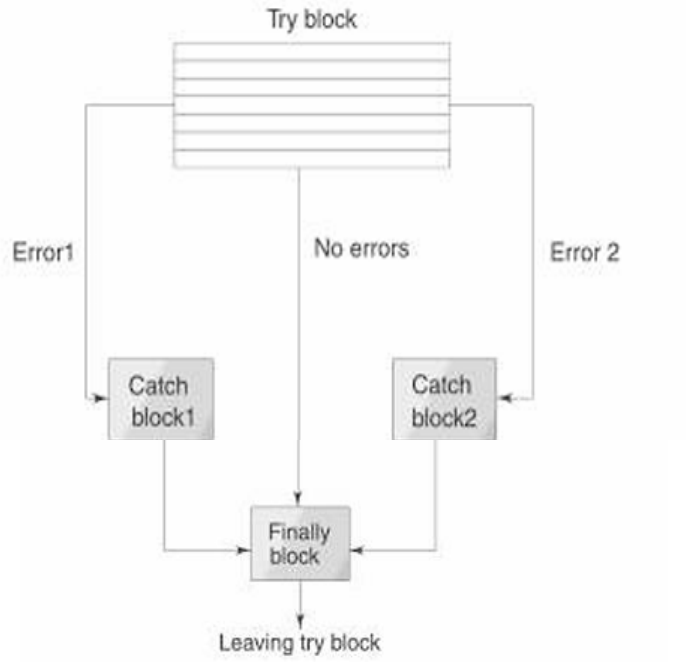
```
catch (Exception e){ }
```

Using finally statement:

C# supports another statement known as a **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements. A **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try                          try
{                            {
    .....                        .....
    .....                        .....
}                            }
finally                      catch (....)
{                            {
    .....                        .....
    .....                        .....
}                            }
```

```
.
.
.
.
finally
{
    .....
    .....
}
```



Throwing our own exception:

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

```
throw new Throwable_subclass;
```

Examples:

```
throw new ArithmeticException( );
throw new FormatException( );
```

Program 18.6 demonstrates the use of a user-defined subclass of Exception class. An object of a class that extends **Throwable** can be thrown and caught.

Introduction to VB.NET:

VB.NET is a simple, multi-paradigm object-oriented programming language designed to create a wide range of Windows, Web, and mobile applications built on the **.NET Framework**. The VB.NET stands for Visual Basic. Network Enabled Technologies. Furthermore, it supports the OOPs concept, such as abstraction, encapsulation, inheritance, and polymorphism. Therefore, everything in the VB.NET language is an object, including all primitive data types

VB.NET Features

These are the following features that make it the most popular programming language.

- It is an object-oriented programming language that follows various oops concepts such as abstraction, encapsulation, inheritance, and many more.
- This language is used to design user interfaces for window, mobile, and web-based applications.
- It supports a rapid application development tool kit.
- It is not a case sensitive language like other languages such as C++, java, etc.
- It supports Boolean condition for decision making in programming.
- It also supports the multithreading concept
- It provides simple events management in .NET application.
- A Window Form enables us to inherit all existing functionality of form
- It uses an external object as a **reference** that can be used in a VB.NET application.
- Automatic initialized a garbage collection.
- It follows a structured and extensible programming language for error detection and recovery.
- Conditional compilation and easy to use generic classes.
- It is useful to develop web, window, and mobile applications.

Advantages of VB.NET

- The VB.NET executes a program in such a way that runs under CLR (Common Language Runtime), creating a robust, stable, and secure application.
- It is a pure object-oriented programming language based on objects and classes.
- Using the Visual Studio IDE, you can develop a small program that works faster, with a large desktop and web application.
- The .NET Framework is a software framework that has a large collection of libraries
- t uses drop and drag elements to create web forms in .NET applications.
- However, a Visual Basic .NET allows to connect one application to another application that created in the same language to run on the .NET framework.

- A VB.NET can automatically structure your code.
- The Visual Basic .NET language is also used to transfer data between different layers of the .NET architecture

Disadvantages of VB.NET

1. The VB.NET programming language is unable to handle pointers directly
2. The VB.NET programming is easy to learn, that increases a large competition between the programmers to apply the same employment or project in VB.NET. Thus, it reduces a secure job in the programming field as a VB.NET developer.
3. It uses an Intermediate Language (IL) compilation that can be easily decompiled (reverse engineered), but there is nothing that can prevent an application from disintegrating.

VB.NET IDE:

Visual Basic.NET IDE is built out of a collection of different windows. Some windows are used for writing code, some for designing interfaces, and others for getting a general overview of files or classes in your application.

The frequently using programming tools in Visual Studio IDE.

- 1. Menu Bar
- 2. Standard Toolbar
- 3. ToolBox
- 4. Forms Designer
- 5. Output Window
- 6. Solution Explorer
- 7. Properties Window

Creating Shortcut to start VB.NET :

In the following section we are going to create a shortcut for starting the VB.NET (i.e. Visual Basic .NET). We can start the Visual Basic by choosing Start | All Programs | Microsoft Visual Studio .NET | Microsoft Visual Studio.NET. Using this shortcut will create us to start Visual Basic more quickly. This shortcut is nothing but an icon that will be placed on the wallpaper/home screen.

Auto-Hide:

Once the initial VB.NET screen appears, you'll find the controls stored inside of the toolbox on the left side of the screen. These are the objects you can place on a form. Users interact with these objects. You can run your mouse over each object and a tool tip will appear informing you of the object's purpose.

The direction of the push pin on the toolbox's title bar dictates whether or not auto hide is in effect. If the push pin is vertical, auto hide is turned off. This means that the toolbox is always visible. If you click on the push pin, it will be displayed vertically. At this point, auto hide is activated and moving your cursor off the toolbox will cause it to collapse. It will only expand when your cursor is over the toolbox icon that appears docked against the left corner of the window. Other windows also have autohide capabilities activated and deactivated by clicking the window's respective push pin.

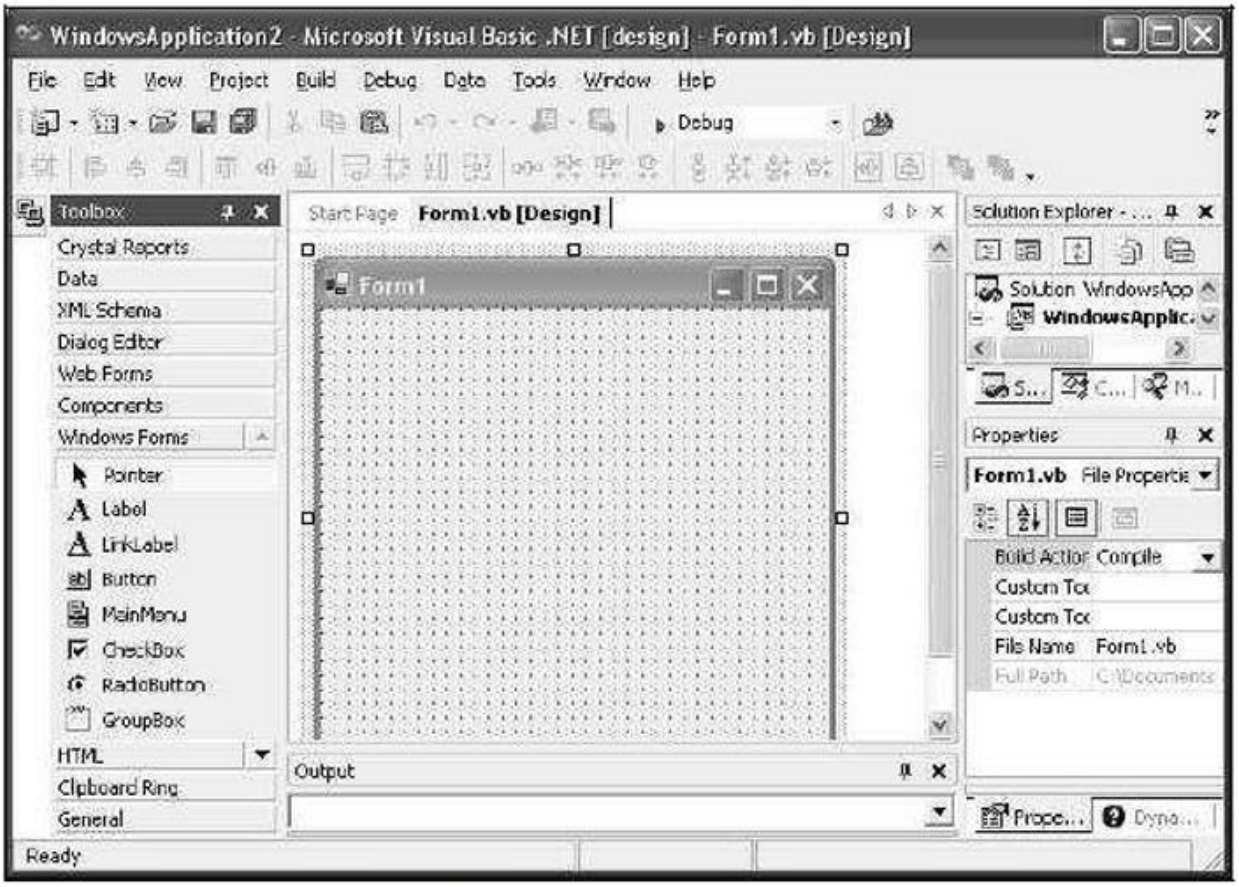


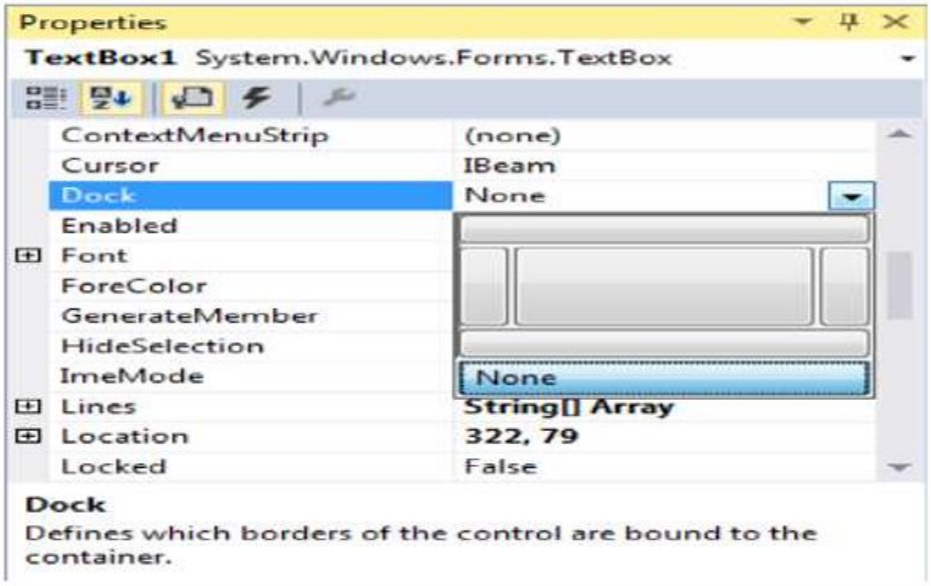
Figure 2.10: A window showing the push pin

Docking and Undocking:

Docking refers to how much space you want the control to take up on the form. If you dock a control to the left of the form, it will stretch itself to the height of the form, but its width will stay the same. To see how it works, click on one of your textboxes and locate the Dock property. Click the arrow to reveal a drop down box:

This time, all the rectangles are like buttons. You can only dock to one side at a time, and the default is None. Click a button to see what it does to your textbox. Click the middle one, and the textbox will Fill the whole form.

Docking is quite useful when used with the splitter control and panels, allowing you to create a Windows-style interface.



We can dock the tool windows we use the most at the location we prefer. Sometimes, we want to undock it for instance to move it to another display. Sometimes, we also undock a tool window by error.

Placing and Resizing the Windows:

If you visually add a control to a form (at design time), in order to perform any type of configuration on the control, you must first select it. Sometimes you will need to select a group of controls.

1. Single Control Selection

To select a control, if you know its name, you can click the arrow of the combo box in the top section of the Properties window and select it.

To select a control on the form, you can simply click it. A control that is selected indicates this by displaying 8 small squares, also called handles, around it. Between these handles, the control is surrounded by dotted rectangles. In the following picture, the selected rectangle displays 8 small squares around its shape:

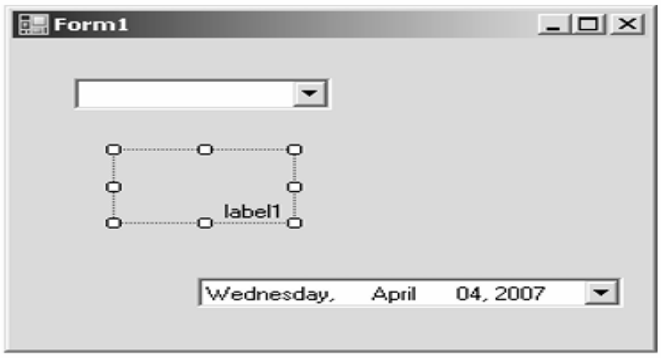


Figure : A single control selection

2.6.2 Multiple Control Selection

To select more than one control on the form, click the first. Press and hold either Shift or Ctrl, then click each of the desired controls on the form. If you click a control that should not be selected, click it again. After selecting the group of controls, release either Shift or Ctrl that you were holding. When a group of controls is selected, the last selected control displays 8 handles too but its handles are white while the others are black. In the following picture, a form contains four controls, three controls are selected:

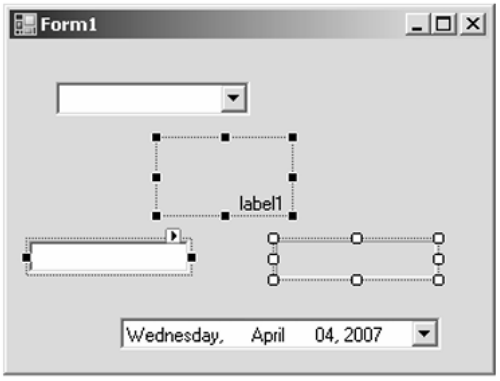


Figure: Multiple control selection

Another technique you can use to select various controls consists of clicking on an unoccupied area on the form, holding the mouse down, drawing a fake rectangle, and releasing the mouse:

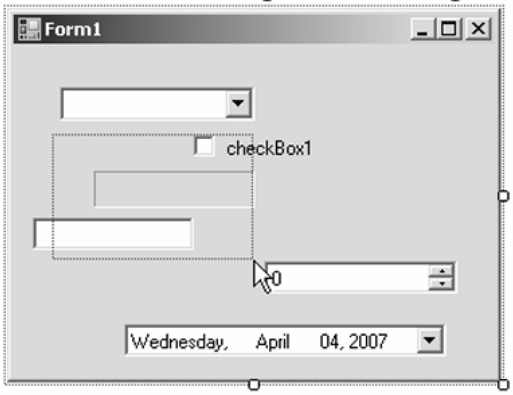


Figure :More than one control selection

Every control touched by the fake rectangle or included in it would be selected:

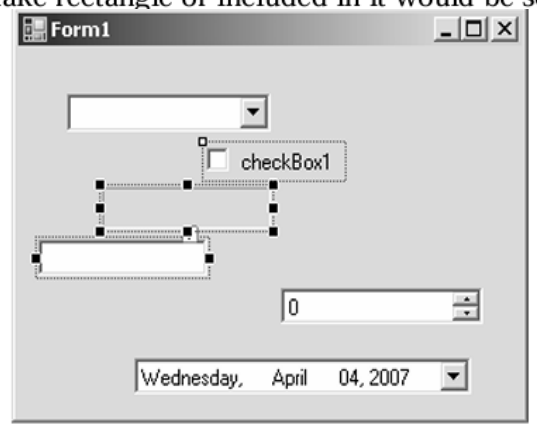


Figure : Multiple control selection

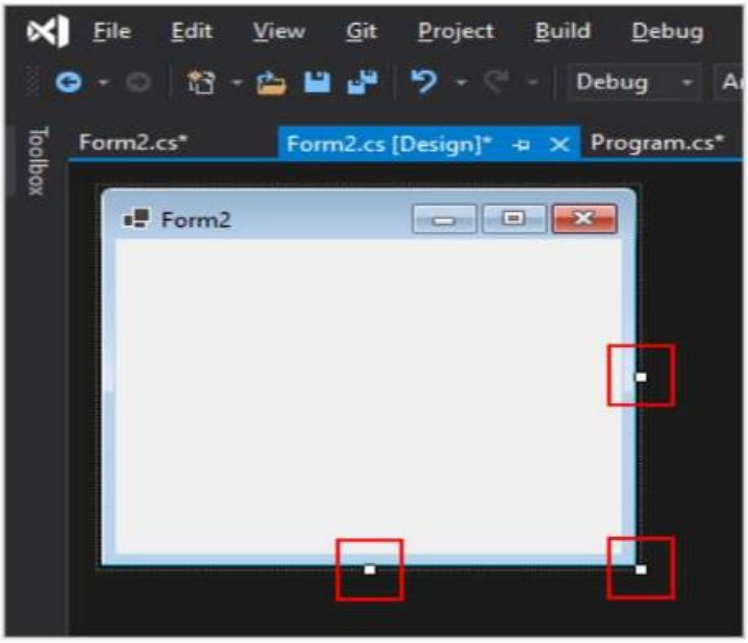
When various controls have been selected, you can resize them by click and drag proces

Placing and resizing the Forms:

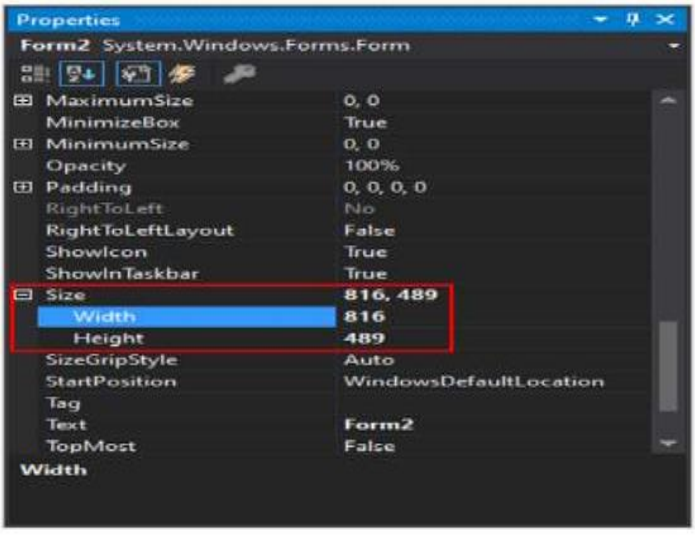
When a form is created, the size and location is initially set to a default value. The default size of a form is generally a width and height of *800x500* pixels. The initial location, when the form is displayed, depends on a few different settings.

Resize with the designer

After adding a new form to the project, the size of a form is set in two different ways. First, you can set it is with the size grips in the designer. By dragging either the right edge, bottom edge, or the corner, you can resize the form.



The second way you can resize the form while the designer is open, is through the properties pane. Select the form, then find the **Properties** pane in Visual Studio. Scroll down to **size** and expand it. You can set the **Width** and **Height** manually.



Properties window and Solution explorer:

The box on the right side of each property name represents the value of the property that you can set for an object. There are various kinds of fields you will use to set the properties. To know what particular kind a field is, you can click its name. But to set or change a property, you use the box on the right side of the property's name: the property's value, also referred to as the field's value.

Text Fields: There are fields that expect you to type a value. Most of these fields have a default value. To change the value of the property, click the name of the property, type the desired value, and press Enter. While some properties, such as the Text, would allow anything, some other fields expect a specific type of text, such as a numeric value.

Expandable Fields: Some fields have a + button. This indicates that the property has a set of sub-properties that actually belong to the same property and are set together. To expand such a field, click its + button and a – button will appear:

To collapse the field, click the – button. Some of the properties are numeric based, such as the above Size. With such a property, you can click its name and type two numeric values separated by a comma. Some other properties are created an enumerator or a class. If you expand such a field, it would display various options. Here is an example:

Boolean Fields: Some fields can have only a true or false value. To change their setting, you can either select from the combo box or double-click the property to toggle to the other value.
Action Fields: Some fields would require a value or item that needs to be set through an intermediary action. Such fields display an ellipsis button. When you click the button, a dialog box would come up and you can set the value for the field.
Boolean Fields: Some fields can have only a true or false value. After clicking the arrow, a list would display:
There are various types of selection fields. Some of them display just two items. To change their value, you can just double-click the field. Some other fields have more than two values in the field. To change them, you can click their arrow and select from the list. You can also double-click a few times until the desired value is selected.

Execution of Basic keywords:

A **keyword** is a reserved word with special meanings in the compiler, whose meaning cannot be changed. Therefore, these keywords cannot be used as an identifier in **VB.NET** programming such as class name, variable, function, module, etc.
In visual basic, Keywords are differentiated into two types those are

- Reserved Keywords
- Unreserved Keywords

Generally, the **reserved keywords** will not allow us to use them as names for programming elements such as variables, class, etc. In case, if we want to bypass this restriction, we need to define the variable name in brackets ([]).
he following table lists the available reserved keywords in the Visual Basic programming language.

AddHandler	AddressOf	Alias	And	AndAlso
As	Boolean	ByRef	Byte	ByVal
Call	Case	Catch	CBool	CByte
CChar	CDate	CDbl	CDec	Char
CInt	Class	CLng	CObj	Const

In visual basic, **unreserved keywords** are not reserved keywords which means we can use them as names for programming elements such as variables, class, etc. However, it's not recommended to use the keywords as programming elements because it will make your code hard to read and lead to subtle errors that can be difficult to find.

The following table lists unserved keywords in a visual basic programming language.

Aggregate	Ansi	Assembly	Async	Auto
Await	Binary	Compare	Custom	Distinct
Equals	Explicit	Group By	Group Join	Into
IsFalse	IsTrue	Iterator	Join	Key

Visual basic Data Types:

When declaring a variable an indication can be given about what type of data will be stored. There are many data types permissible under Visual Basic. The following are common for declaring the usual types of data involved in computer processing.

- Short. A non-decimal (whole) number with a value in the range of -32,768 to +32,767.
- Integer. A non-decimal (whole) number with a value in the range of -2,147,483,648 to +2,147,483,647.
- Long. A non-decimal (whole) number with a value in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.
- Single. A floating-point (decimal) positive or negative number with a value in the range of up to 3.402823E38 (scientific notation).
- Double. A floating-point (decimal) positive or negative number with a value in the range of up to 1.79769313486232E308.
- Decimal. A decimal number with a value in the range of up to +79,228,162,514,264,337,593,950,335. without decimal precision and up to +7.9228162514264337593543950335 with 28 places of decimal precision.

- String. Any number of alphabetic, numerical, and special characters.
- Date. Dates in the range between January 1, 0001 to December 31, 9999 and times in the range between 0:00:00 to 23:59:59.
- Boolean. The value True or False.

Variables, then, are declared by assigning a name and a data type. The following are examples of valid variable declarations that can be used to store data of the identified types.

Dim My_Counter As Integer

Dim My_Salary As Decimal

Dim My_Great-Big-Number As Double

Dim My_Name As String

Dim My_Birthday As Date

Dim My_Current_Time As Date

Dim My_Decision As Boolean

The Data types available in VB .NET, their size, type, description are summarized in the table below...

Data Type	Size in Bytes	Description	Type
Byte	1	8-bit unsigned integer	System.Byte
Char	2	16-bit Unicode characters	System.Char
Integer	4	32-bit signed integer	System.Int32
Double	8	64-bit floating point variable	System.Double
Long	8	64-bit signed integer	System.Int64
Short	2	16-bit signed integer	System.Int16
Single	4	32-bit floating point variable	System.Single
String	Varies	Non-Numeric Type	System.String
Date	8	Date Type	System.Date
Boolean	2	Non-Numeric Type	System.Boolean
Object	4	Non-Numeric Type	System.Object
Decimal	16	128-bit floating point variable	System.Decimal

VB.NET Statements:

A **statement** is a complete instruction in Visual Basic programs. It may contain keywords, operators, variables, literal values, constants and expressions.

Statements could be categorized as −

- **Declaration statements** − these are the statements where you name a variable, constant, or procedure, and can also specify a data type.
- **Executable statements** − these are the statements, which initiate actions. These statements can call a method or function, loop or branch through blocks of code or assign values or expression to a variable or constant. In the last case, it is called an Assignment statement.

Conditional Statements:
If…Else statement

An **If** statement can be followed by an optional **Else** statement, which executes when the Boolean expression is false.

Syntax

The syntax of an If...Then... Else statement in VB.Net is as follows

```
If(boolean_expression)Then
  'statement(s) will execute if the Boolean expression is true
Else
  'statement(s) will execute if the Boolean expression is false
End If
```

The If...Else If...Else Statement

An **If** statement can be followed by an optional **Else if...Else** statement, which is very useful to test various conditions using single If...Else If statement.

When using If... Else If... Else statements, there are few points to keep in mind.

- An If can have zero or one Else's and it must come after an Else If's.
- An If can have zero to many Else If's and they must come before the Else.
- Once an Else if succeeds, none of the remaining Else If's or Else's will be tested.

Syntax

The syntax of an if...else if...else statement in VB.Net is as follows −

```
If(boolean_expression 1)Then
   ' Executes when the boolean expression 1 is true
ElseIf( boolean_expression 2)Then
   ' Executes when the boolean expression 2 is true
ElseIf( boolean_expression 3)Then
   ' Executes when the boolean expression 3 is true
Else
   ' executes when the none of the above condition is true
End If
```

Select case:

A **Select Case** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each select case.

Syntax

The syntax for a Select Case statement in VB.Net is as follows −

```
Select [ Case ] expression
   [ Case expressionlist
      [ statements ] ]
   [ Case Else
      [ elsestatements ] ]
End Select
```

Where,

- ***expression*** − is an expression that must evaluate to any of the elementary data type in VB.Net, i.e., Boolean, Byte, Char, Date, Double, Decimal, Integer, Long, Object, SByte, Short, Single, String, UInteger, ULong, and UShort.
- ***expressionlist*** − List of expression clauses representing match values for *expression*. Multiple expression clauses are separated by commas.
- ***statements*** − statements following Case that run if the select expression matches any clause in *expressionlist*.
- ***elsestatements*** − statements following Case Else that run if the select expression does not match any clause in the *expressionlist* of any of the Case statements.

Switch statement:

Evaluates a list of expressions and, on finding the first expression to evaluate to True, returns an associated value or expression

```
Switch(expr-1, value-1[, expr-2, value-2 ... [, _
    expr-n,value-n]])
```

expr (required; Object)

A number of expressions to be evaluated

value (required; Object)

An expression or value to return if the associated expression evaluates to True

Important points to be remembered:

- A minimum of two expression/value pairs is required; additional pairs are optional.
- Expressions are evaluated from left to right.
- If none of the expressions is True, the *Switch* function returns Nothing.
- If multiple expressions are True, *Switch* returns the value that corresponds to the first True expression.
- *value* can be a constant, variable, or expression.

VB.NET Do loop:

A **Loop** is used to repeat the same process multiple times until it meets the specified condition in a program. By using a loop in a program, a programmer can repeat any number of statements up to the desired number of repetitions. A loop also provides the suitability to a programmer to repeat the statement in a program according to the requirement. A loop is also used to reduce the program **complexity, easy to understand**, and easy to **debug**.

Advantages of VB.NET loop

- It provides code iteration functionality in a program.
- It executes the statement until the specified condition is true.
- It helps in reducing the size of the code.
- It reduces compile time.

Types of Loops

There are five types of loops available in VB.NET:

- Do While Loop

- For Next Loop
- For Each Loop
- While End Loop
- With End Loop

Do While Loop

In VB.NET, Do While loop is used to execute blocks of statements in the program, as long as the condition remains true. It is similar to the While End Loop, but there is slight difference between them. The **while** loop **initially checks** the defined condition, if the condition becomes true, the while loop's statement is executed. Whereas in the **Do** loop, is opposite of the while loop, it means that it executes the Do statements, and then it checks the condition.

Syntax:

```
1. Do
2. [ Statements to be executed]
3. Loop While Boolean_expression
4. // or
5. Do
6. [Statement to be executed]
7. Loop Until Boolean_expression
```

For Next loop:

A **For Next loop** is used to repeatedly execute a sequence of code or a block of code until a given condition is satisfied. A For loop is useful in such a case when we know how many times a block of code has to be executed. In VB.NET, the For loop is also known as For Next Loop.

Syntax

```
1. For variable_name As [ DataType ] = start To end [ Step step ]
2. [ Statements to be executed ]
3. Next
```

Let's understand the For Next loop in detail.

- **For**: It is the keyword that is present at the beginning of the definition.
- **variable_name**: It is a variable name, which is required in the For loop Statement. The value of the variable determines when to exit from the **For-Next loop,** and the value should only be a numeric.
- **[Data Type]**: It represents the Data Type of the **variable_name**.
- **start To end**: The **start** and **end** are the two important parameters representing the initial and final values of the **variable_name**. These parameters are helpful while the execution begins, the initial value of the variable is set by the start. Before the completion of each repetition, the variable's current value is compared with the end value. And if the value of the variable is less than the end value, the execution continues until the variable's current value is greater than the end value. And if the value is exceeded, the loop is terminated.

- **Step**: A step parameter is used to determine by which the **counter** value of a variable is increased or decreased after each iteration in a program. If the counter value is not specified; It uses 1 as the default value.
- **Statements**: A statement can be a single statement or group of statements that execute during the completion of each iteration in a loop.
- **Next**: In VB.NET a **Next** is a keyword that represents the end of the **For loop's**

For Each Next loop:

Repeats a group of statements for each element in a collection.

Syntax

```
For Each element [ As datatype ] In group
   [ statements ]
   [ Continue For ]
   [ statements ]
   [ Exit For ]
   [ statements ]
Next [ element ]
```

Parts

Term	Definition
element	Required in the For Each statement. Optional in the Next statement. Variable. Used to iterate through the elements of the collection.
datatype	Optional if Option Infer is on (the default) or element is already declared; required if Option Infer is off and element isn't already declared. The data type of element.
group	Required. A variable with a type that's a collection type or Object. Refers to the collection over which the statements are to be repeated.
statements	Optional. One or more statements between For Each and Next that run on each item in group.
Continue For	Optional. Transfers control to the start of the For Each loop.
Exit For	Optional. Transfers control out of the For Each loop.
Term	**Definition**
Next	Required. Terminates the definition of the For Each loop.

While loop in VB.NET:

It executes a series of statements as long as a given condition is True.

The syntax for this loop construct is –

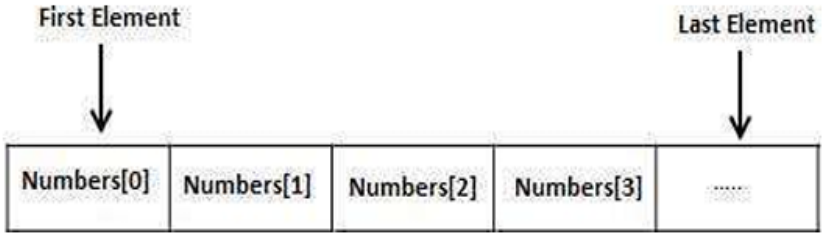
```
While condition
   [ statements ]
   [ Continue While ]
   [ statements ]
   [ Exit While ]
   [ statements ]
End While
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is logical true. The loop iterates while the condition is true.

Arrays in VB.NET:

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Creating Arrays in VB.Net

To declare an array in VB.Net, you use the Dim statement. For example,

```
Dim intData(30)      ' an array of 31 elements
Dim strData(20) As String   ' an array of 21 strings
Dim twoDarray(10, 20) As Integer     'a two dimensional array of integers
Dim ranges(10, 100)           'a two dimensional array
```

You can also initialize the array elements while declaring the array. For example,

```
Dim intData() As Integer = {12, 16, 20, 24, 28, 32}
Dim names() As String = {"Karthik", "Sandhya", _
"Shivangi", "Ashwitha", "Somnath"}
Dim miscData() As Object = {"Hello World", 12d, 16ui, "A"c}
```

The elements in an array can be stored and accessed by using the index of the array.

The following program demonstrates this –

```
Module arrayApl
  Sub Main()
    Dim n(10) As Integer ' n is an array of 11 integers '
    Dim i, j As Integer
    ' initialize elements of array n '

    For i = 0 To 10
      n(i) = i + 100 ' set element at location i to i + 100
    Next i
    ' output each array element's value '

    For j = 0 To 10
      Console.WriteLine("Element({0}) = {1}", j, n(j))
    Next j
    Console.ReadKey()
  End Sub
End Module
```

Dynamic Arrays

Dynamic arrays are arrays that can be dimensioned and re-dimensioned as par the need of the program. You can declare a dynamic array using the **ReDim** statement.

Syntax for ReDim statement −

ReDim [Preserve] arrayname(subscripts)

Where,

- The **Preserve** keyword helps to preserve the data in an existing array, when you resize it.
- **arrayname** is the name of the array to re-dimension.
- **subscripts** specifies the new dimension.

Multi-Dimensional Arrays

- VB.Net allows multidimensional arrays. Multidimensional arrays are also called rectangular arrays.
- You can declare a 2-dimensional array of strings as −
- Dim twoDStringArray(10, 20) As String
- or, a 3-dimensional array of Integer variables −
- Dim threeDIntArray(10, 10, 10) As Integer