C# - Data Types

The variables in C#, are categorized into the following types −

- Value types
- Reference types
- Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010 −

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0}$ to 28	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0

long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine −

```
using System;

namespace DataTypeApplication {
  class Program {
    static void Main(string[] args) {
      Console.WriteLine("Size of int: {0}", sizeof(int));
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Size of int: 4
```

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic,** and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; // this is boxing
```

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is −

dynamic <variable_name> = value;

For example,

```
dynamic d = 20;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

```
String str = "Tutorials Point";
```

A @quoted string literal looks as follows −

```
@"Tutorials Point";
```

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is −

```
type* identifier;
```

For example,

```
char* cptr;
int* iptr;
```

C# - Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

- **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
- **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion −

```
using System;

namespace TypeConversionApplication {
  class ExplicitConversion {
    static void Main(string[] args) {
      double d = 5673.74;
      int i;

      // cast double to int.
      i = (int)d;
      Console.WriteLine(i);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

5673

C# Type Conversion Methods

C# provides the following built-in type conversion methods −

Sr.No.	Methods & Description
1	**ToBoolean**

	Converts a type to a Boolean value, where possible.
2	**ToByte** Converts a type to a byte.
3	**ToChar** Converts a type to a single Unicode character, where possible.
4	**ToDateTime** Converts a type (integer or string type) to date-time structures.
5	**ToDecimal** Converts a floating point or integer type to a decimal type.
6	**ToDouble** Converts a type to a double type.
7	**ToInt16** Converts a type to a 16-bit integer.
8	**ToInt32** Converts a type to a 32-bit integer.
9	**ToInt64** Converts a type to a 64-bit integer.
10	**ToSbyte** Converts a type to a signed byte type.
11	**ToSingle** Converts a type to a small floating point number.

12	**ToString** Converts a type to a string.
13	**ToType** Converts a type to a specified type.
14	**ToUInt16** Converts a type to an unsigned int type.
15	**ToUInt32** Converts a type to an unsigned long type.
16	**ToUInt64** Converts a type to an unsigned big integer.

The following example converts various value types to string type −

```
using System;

namespace TypeConversionApplication {
  class StringConversion {
    static void Main(string[] args) {
      int i = 75;
      float f = 53.005f;
      double d = 2345.7652;
      bool b = true;

      Console.WriteLine(i.ToString());
      Console.WriteLine(f.ToString());
      Console.WriteLine(d.ToString());
      Console.WriteLine(b.ToString());
      Console.ReadKey();

    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

75
53.005
2345.7652
True

C# - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as −

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class.**

Defining Variables

Syntax for variable definition in C# is −

<data_type> <variable_list>;

Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here −

int i, j, k;

```
char c, ch;
float f, salary;
double d;
```

You can initialize a variable at the time of definition as −

```
int i = 100;
```

Initializing Variables

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is −

```
variable_name = value;
```

Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as −

```
<data_type> <variable_name> = value;
```

Some examples are −

```
int d = 3, f = 5;    /* initializing d and f. */
byte z = 22;         /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';        /* the variable x has the value 'x'. */
```

It is a good programming practice to initialize variables properly, otherwise sometimes program may produce unexpected result.

The following example uses various types of variables −

```
using System;

namespace VariableDefinition {
  class Program {
    static void Main(string[] args) {
      short a;
      int b ;
      double c;

      /* actual initialization */
      a = 10;
      b = 20;
      c = a + b;
      Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
      Console.ReadLine();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

a = 10, b = 20, c = 30

Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console.ReadLine()** accepts the data in string format.

Lvalue and Rvalue Expressions in C#

There are two kinds of expressions in C# −

- **lvalue** − An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** − An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side. Following is a valid C# statement −

int g = 20;

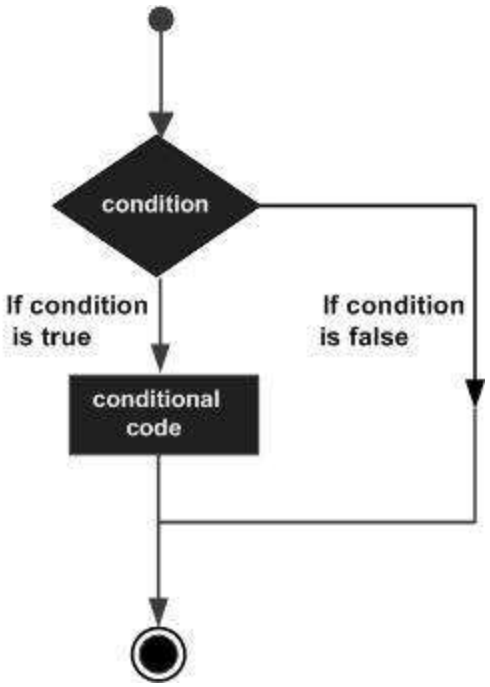
But following is not a valid statement and would generate compile-time error −

10 = 20;

C# - Decision Making

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C# provides following types of decision making statements.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements

	You can use one **switch** statement inside another **switch** statement(s).

The ? : Operator

We have covered **conditional operator ? :** which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

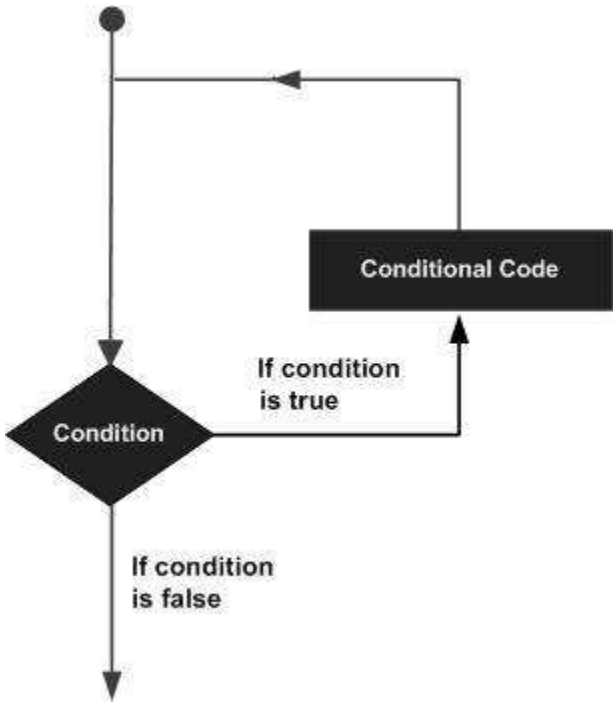
The value of a ? expression is determined as follows: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C# - Loops

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C# provides following types of loop to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is similar to a while statement, except that it tests the condition at the end of the loop body
4	nested loops You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements.

Sr.No.	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

Example

```
using System;

namespace Loops {
  class Program {
    static void Main(string[] args) {
      for (; ; ) {
        Console.WriteLine("Hey! I am Trapped");
      }
    }
  }
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

C# - Inheritance

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows −

```
<acess-specifier> class <base_class> {
  ...
}
```

```
class <derived_class> : <base_class> {
   ...
}
```

Consider a base class Shape and its derived class Rectangle −

```
using System;

namespace InheritanceApplication {
  class Shape {
    public void setWidth(int w) {
      width = w;
    }
    public void setHeight(int h) {
      height = h;
    }
    protected int width;
    protected int height;
  }

  // Derived class
  class Rectangle: Shape {
    public int getArea() {
      return (width * height);
    }
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle Rect = new Rectangle();

      Rect.setWidth(5);
      Rect.setHeight(7);

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35

Initializing Base Class
```

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this −

```
using System;

namespace RectangleApplication {
  class Rectangle {

    //member variables
    protected double length;
    protected double width;

    public Rectangle(double l, double w) {
      length = l;
      width = w;
    }
    public double GetArea() {
      return length * width;
    }
    public void Display() {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
    }
  }//end class Rectangle
  class Tabletop : Rectangle {
    private double cost;
    public Tabletop(double l, double w) : base(l, w) { }

    public double GetCost() {
      double cost;
      cost = GetArea() * 70;
      return cost;
    }
    public void Display() {
      base.Display();
      Console.WriteLine("Cost: {0}", GetCost());
    }
  }
  class ExecuteRectangle {
    static void Main(string[] args) {
      Tabletop t = new Tabletop(4.5, 7.5);
      t.Display();
      Console.ReadLine();
```

```
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

Multiple Inheritance in C#

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this −

```
using System;

namespace InheritanceApplication {
  class Shape {
    public void setWidth(int w) {
      width = w;
    }
    public void setHeight(int h) {
      height = h;
    }
    protected int width;
    protected int height;
  }

  // Base class PaintCost
  public interface PaintCost {
    int getCost(int area);
  }

  // Derived class
  class Rectangle : Shape, PaintCost {
    public int getArea() {
      return (width * height);
    }
    public int getCost(int area) {
      return area * 70;
    }
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle Rect = new Rectangle();
```

```
      int area;

      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
      Console.ReadKey();
   }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
Total paint cost: $2450
```

C# - Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

- Function overloading
- Operator overloading

We discuss operator overloading in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types −

```
using System;

namespace PolymorphismApplication {
  class Printdata {
```

```
    void print(int i) {
      Console.WriteLine("Printing int: {0}", i );
    }
    void print(double f) {
      Console.WriteLine("Printing float: {0}" , f);
    }
    void print(string s) {
      Console.WriteLine("Printing string: {0}", s);
    }
    static void Main(string[] args) {
      Printdata p = new Printdata();

      // Call print to print integer
      p.print(5);

      // Call print to print float
      p.print(500.263);

      // Call print to print string
      p.print("Hello C++");
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes −

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class −

```
using System;
```

```
namespace PolymorphismApplication {
  abstract class Shape {
    public abstract int area();
  }

  class Rectangle:  Shape {
    private int length;
    private int width;

    public Rectangle( int a = 0, int b = 0) {
      length = a;
      width = b;
    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this −

```
using System;

namespace PolymorphismApplication {
  class Shape {
    protected int width, height;

    public Shape( int a = 0, int b = 0) {
```

```
      width = a;
      height = b;
    }
    public virtual int area() {
      Console.WriteLine("Parent class area :");
      return 0;
    }
  }
  class Rectangle: Shape {
    public Rectangle( int a = 0, int b = 0): base(a, b) {

    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
  class Triangle: Shape {
    public Triangle(int a = 0, int b = 0): base(a, b) {
    }
    public override int area() {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller {
    public void CallArea(Shape sh) {
      int a;
      a = sh.area();
      Console.WriteLine("Area: {0}", a);
    }
  }
  class Tester {
    static void Main(string[] args) {
      Caller c = new Caller();
      Rectangle r = new Rectangle(10, 7);
      Triangle t = new Triangle(10, 5);

      c.CallArea(r);
      c.CallArea(t);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

Rectangle class area:
Area: 70
Triangle class area:
Area: 25

C# - Regular Expressions

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching. A pattern consists of one or more character literals, operators, or constructs.

Constructs for Defining Regular Expressions

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the following links to find these constructs.

- Character escapes
- Character classes
- Anchors
- Grouping constructs
- Quantifiers
- Backreference constructs
- Alternation constructs
- Substitutions
- Miscellaneous constructs

The Regex Class

The Regex class is used for representing a regular expression. It has the following commonly used methods −

Sr.No.	Methods & Description
1	**public bool IsMatch(string input)** Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	**public bool IsMatch(string input, int startat)** Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.

3	**public static bool IsMatch(string input, string pattern)** Indicates whether the specified regular expression finds a match in the specified input string.
4	**public MatchCollection Matches(string input)** Searches the specified input string for all occurrences of a regular expression.
5	**public string Replace(string input, string replacement)** In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	**public string[] Split(string input)** Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

For the complete list of methods and properties, please read the Microsoft documentation on C#.

Example 1

The following example matches words that start with 'S' −

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {
  class Program {
    private static void showMatch(string text, string expr) {
      Console.WriteLine("The Expression: " + expr);
      MatchCollection mc = Regex.Matches(text, expr);

      foreach (Match m in mc) {
        Console.WriteLine(m);
      }
    }
    static void Main(string[] args) {
      string str = "A Thousand Splendid Suns";

      Console.WriteLine("Matching words that start with 'S': ");
      showMatch(str, @"\bS\S*");
      Console.ReadKey();
```

```
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

Example 2

The following example matches words that start with 'm' and ends with 'e' −

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {
  class Program {
    private static void showMatch(string text, string expr) {
      Console.WriteLine("The Expression: " + expr);
      MatchCollection mc = Regex.Matches(text, expr);

      foreach (Match m in mc) {
        Console.WriteLine(m);
      }
    }
    static void Main(string[] args) {
      string str = "make maze and manage to measure it";

      Console.WriteLine("Matching words start with 'm' and ends with 'e':");
      showMatch(str, @"\bm\S*e\b");
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

Example 3

This example replaces extra white space −

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {
   class Program {
      static void Main(string[] args) {
         string input = "Hello   World   ";
         string pattern = "\\s+";
         string replacement = " ";

         Regex rgx = new Regex(pattern);
         string result = rgx.Replace(input, replacement);

         Console.WriteLine("Original String: {0}", input);
         Console.WriteLine("Replacement String: {0}", result);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Original String: Hello World
Replacement String: Hello World
```