1.2 Literals, variables and Datatype

- A programming language is designed to manipulate certain kinds of data consisting of numbers, characters and strings and provide useful output known as ***information to the user.***
- The task of manipulating data is accomplished by executing a sequence of instructions constituting what is known as a ***program***
- A C# program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing instructions known as ***executable statements.***
- These instructions are formed using certain symbols and words according to some rigid rules known as syntax rules (or grammar).

- The smallest individual unit in a program are referred to as ***tokens.***
- The compiler recognizes them for building up expression and statements.
- White spaces and comments are not tokens, though they may act as separators for tokens
- A C# program is a collection of tokens, comments and white spaces.

C# includes the following five types of tokens.

- Keywords
- Identifiers
- Literals
- Operators
- Punctuators

Keywords

Keywords are predefined sets of reserved words that have special meaning in a program. The meaning of keywords cannot be changed, neither can they be directly used as identifiers in a program.

For example

```
int a;
```

Here, **int** is a keyword and **a** is a variable (identifier). **int** has a special meaning in C# i.e. it is used to declare variables of type **int** and this function cannot be changed.

C# has a total of 79 keywords. All these keywords are in lowercase. Here is a complete list of all C# keywords.

abstract	as	base	bool

break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in (generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out (generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	using static
void	volatile	while	

Although keywords are reserved words, they can be used as identifiers if `@` is added as prefix.

For example,

```
int @void;
```

The above statement will create a variable `@void` of type `int`.

Identifiers

Identifiers are the name given to entities such as variables, methods, classes, labels, namespaces etc. They are tokens in a program which uniquely identify an element.

For example

```
int value;
```

Here, `value` is the name of variable. Hence it is an identifier. Reserved keywords cannot be used as identifiers unless `@` is added as prefix.

Rules for Naming an Identifier

- An identifier cannot be a C# keyword.
- An identifier must begin with a letter, an underscore or `@` symbol. The remaining part of identifier can contain letters, digits and underscore symbol.
- Whitespaces are not allowed. Neither it can have symbols other than letter, digits and underscore.
- Identifiers are case-sensitive. So, `getName`, `GetName` and `getname` represents 3 different identifiers.

Literals

Literals are the values constants assigned to a variables (or result of an expression) in a program.

Operators

Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.

For example, in `2+3`, `+` is an operator that is used to carry out addition operation, while `2` and `3` are operands.

Punctuators

Punctuators are the symbols used for grouping and separating code. They define the shape and function of a program. Punctuators (also known as separators)

Parentheses ()	Colon :
Braces {}	Semicolon ;
Comma ,	Period .

Literals in C#

The fixed values are called as ***Literal***. Literal is a value that is used by the variables. Values can be either an integer, float or string, etc.

```
// Here 100 is a constant/literal.
int x = 100;
```

Literals can be of the following types:

- **Integer Literals**
- **Floating-point Literals**
- **Character Literals**
- **String Literals**
- **Null Literals**
- **Boolean Literals**

- **Integer Literals:**
  A literal of integer type is known as the integer literal. It can be octal, decimal, binary, or hexadecimal constant. No prefix is required for the decimal numbers. A suffix can also be used with the integer literals like *U* or *u* are used for unsigned numbers while *l* or *L* are used for long numbers. By default, every literal is of int type. For Integral data types (byte, short, int, long),

  Decimal literals (Base 10): In this form, the allowed digits are 0-9.
  int x = 101;

Octal literals (Base 8): In this form, the allowed digits are 0-7.

// The octal number should be prefix with 0.

int x = 0146;

Hexa-decimal literals (Base 16): In this form, the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that c# is a case-sensitive programming language but here c# is not case-sensitive.
// The hexa-decimal number should be prefix

// with 0X or 0x.

int x = 0X123Face;

Binary literals (Base 2): In this form, the allowed digits are only **1's** and **0's**.
// The binary number should be prefix with 0b.

int x = 0b101

- **Floating-point Literals:**
  The literal which has an integer part, a decimal point, a fractional part, and an exponent part is known as the floating-point literal. These can be represented either in decimal form or exponential form.
  Examples:
  Double d = 3.14145 // Valid
- **Character Literals:**
  For character data types we can specify literals in 3 ways:
  Single quote: We can specify literal to char data type as single character within single quote.
  char ch = 'a';
- **String Literals:**
  Literals which are enclosed in **double quotes("")** or starts with **@""** are known as the String literals.
  Examples:
  String s1 = "Hello Geeks!";

  String s2 = @"Hello Geeks!";
- **Boolean Literals:**
  Only two values are allowed for Boolean literals i.e. true and false.
  Example:
  bool b = true;

  bool c = false;

- **Backslash Character Literal**

  C# supports special backlash character constants that are used in output methods. For example ,the symbol '\n' stands for a new line character

Escape Sequence	Meaning
\b	Backspace
\a	Alert or Bell
\n	New Line
\f	Form Feed
\r	Carriage Return
\v	Vertical Tab

Variables

- A variables is an identifier that denotes a storage location used to store a data value.
- A variable may take different values at different time during the execution of the program.
- Every variable has a type that determines what value can be stored in the variable.
- A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program.

  Example: average, total_height, ClassStrenght

Rules for declaring a variables

- Variable names must be unique.
- Variable names can contain letters, digits, and the underscore _ only.
- Variable names must start with a letter.
- Variable names are case-sensitive, num and Num are considered different names.
- Variable names cannot contain reserved keywords. Must prefix @ before keyword if want reserve keywords as identifiers.
- White space is not allowed
- Variables names can be of any length

Declaring a variables

Variables are the name of storage locations. After designing suitable variable names, we must declare them to the computer.

Declaring does three things:

1. It tells the compiler about the variable name.
2. It specifies what type of data the variables will hold
3. The place of declaring (in the program) decides the scope of the variables.

Declaring a variables

Syntax:

```
datatype variable1, variable2, …….. variable N;
```

- A variable must be declared before it is used in the program.
- A variable can be used to store a value of any data type.
- Variables are separated by commas. A declaration statement must end with a semicolon

Example

```
int count;
float average;
double pi;
char c1, c2,c3;
```

Initialization of a variable

A variable must be given a value after it has been declared but before it is used in an expression. A simple method of giving value to a variable is through the assignment operator.

Syntax:

```
Variable = value;
```

Example:

```
a=10;
```

It is possible to assign a value to a variable at the time of its declaration.

Syntax:

```
datatype variableName = value;
```

Example:

int a=10;

We can also string assignment expression

x=y=z=10;

The process of initial value to variable is known as initialization

Constant variable

- The variable whose values do not change during the execution of a program is known as constant.
- **const** keyword is used to declare constant
- Constant must be declared and initialized simultaneously.

  Example:

  const int a=10;

  const double PI=3.14;

- A constant variable can be initialized using an expression

  const int m=10;

  const int n=m* 20;

- we cannot use non-constant values in expression.

  int m=10; // non-constant value

  int n=m*20; // error

Scope of a variable

A variable scope refers to the availability of variables in certain parts of the code.

In C#, a variable has three types of scope:

- Class Level Scope
- Method Level Scope
- Block Level Scope

Class Level Scope

In C#, when we declare a variable inside a class, the variable can be accessed within the class. This is known as **class level variable scope**.

Class level variables are known as fields and they are declared outside of methods, constructors, and blocks of the class.

Example

```
using System;
namespace VariableScope
{
  class Program
{

    // class level variable
    string str = "Class Level";

    public void display()
   {
      Console.WriteLine(str);
    }

   public  static void Main(string[] args)
   {
      Program ps = new Program();
      ps.display();
     Console.ReadLine();
    }
  }
}
```

Output

```
Class Level
```

In the above example, we have initialized a variable named `str` inside the `Program` class.

Since it is a class level variable, we can access it from a method present inside the class.

This is because the class level variable is accessible throughout the class.

Note: We cannot access the class level variable through static methods. For example, suppose we have a static method inside the `Program` class.

```
static void display2()
 {
```

```
 // Access class level variable
  // Cause an Error
  Console.WriteLine(str);
}
```

Method Level Scope

When we declare a variable inside a method, the variable cannot be accessed outside of the method. This is known as **method level variable scope**.

Example

```
using System;

namespace VariableScope
 {
  class Program
{
 public void method1()
  {
          string str = "method level";
   }

    public void method2()
    {
    Console.WriteLine(str);
    }
 static void Main(string[] args)
 {
     Program ps = new Program();
     ps.method2();
Console.ReadLine();
    }
  }
}
```

In the above example, we have created a variable named `str` inside `method1()`.

```
// Inside method1()
string str = "method level";
```

Here, `str` is a method level variable. So, it cannot be accessed outside `method1()`.

However, when we try to access the `str` variable from the `method2()`

```
// Inside method2
Console.WriteLine(str);  // Error code
```

This is because method level variables have scope inside the method where they are created.

Example

```
using System;
namespace VariableScope
{
  class Program
{
public void display()
   {
     string str = "inside method";
      Console.WriteLine(str);
    }
 static void Main(string[] args)
{
    Program ps = new Program();
    ps.display();
Console.ReadLine();
    }
  }
}
```

Output

```
inside method
```

Here, we have created the `str` variable and accessed it within the same method `display()`. Hence, the code runs without any error.

Block Level Scope

When we declare a variable inside a block (for loop, while loop, if..else), the variable can only be accessed within the block. This is known as **block level variable scope**.

Example

```
using System;
namespace VariableScope
 {
  class Program
 {
    public void display()
{

      for(int i=0;i<=3;i++)
     {

      }
    Console.WriteLine(i);
  }

    static void Main(string[] args)
 {
      Program ps = new Program();
      ps.display();
     Console.ReadLine();
    }
  }
}
```

In the above program, we have initialized a block level variable i inside the for loop.

```
for(int i=0;i<=3;i++)
{

}
```

Since i is a block level variable, when we try to access the variable outside the for loop,

```
// Outside for loop
Console.WriteLine(i);
```

we get an error.

```
Error    CS0103  The name 'i' does not exist in the current context
```

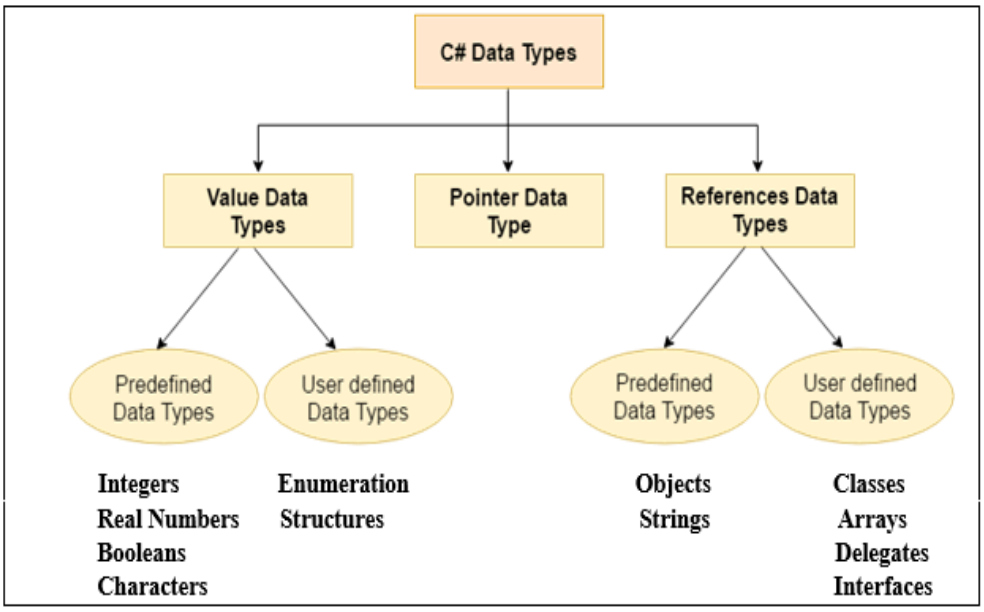
Data Types

Datatype specify the size the type of values that can be stored.
The C# type is primarily divided into two categories

- Value types
- Reference types
- Pointer types

Value types and reference type differ in two characteristics

- ✓ Where they are stored in the memory
- ✓ How they behave in the context of assignment statement



Value types

Value types are stored on the stack and when a value of a variable is assigned to another variable, the value is actually copied. This means that two identical copies of the value are available in the memory.

The value types of C# can be grouped into two categories

- Predefined types (primitive datatype)
- User-defined types (complex datatype)

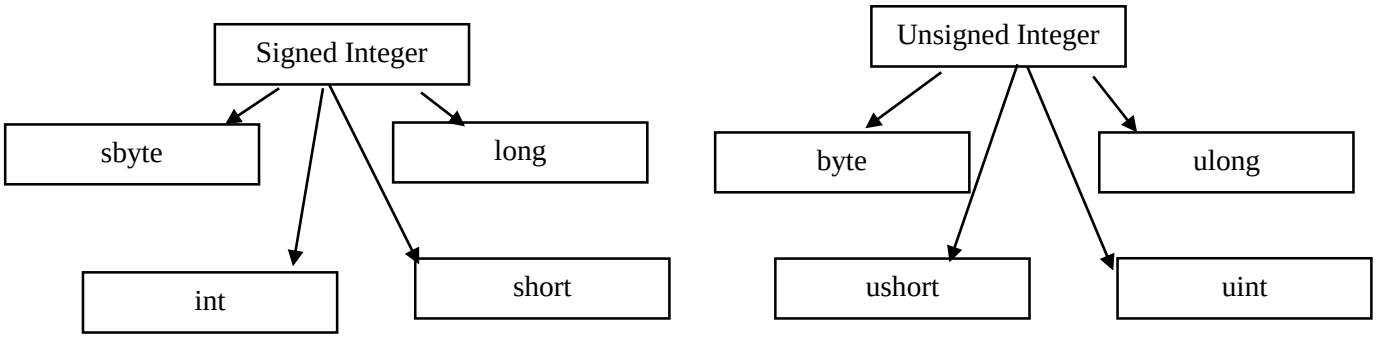
Predefined types (primitive datatype)

Predefined type are further subdivided into:

1. Integer
2. Floating Point number
3. Boolean
4. Character

1. Integer

Integer type numbers are whole numbers without decimal points like 20, 30. It can be negative or positive numbers.



- **Signed Integers**
  Signed integers types can hold both positive and negative numbers.
  Memory size and range of all four signed integer datatypes.

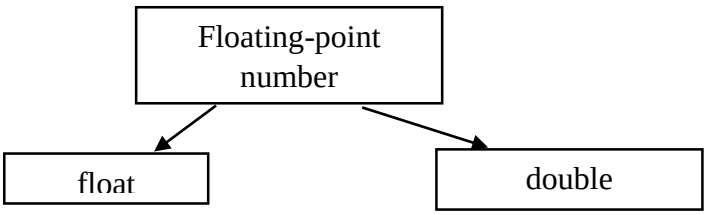
Type	Size
sbyte	1 byte
short	2 byte
int	4 byte
long	8 byte

- **Unsigned Integers**
  Unsigned integers types can hold both numbers.
  Memory size and range of all four signed integer datatypes.

Type	Size
byte	1 byte
ushort	2 byte
uint	4 byte
ulong	8 byte

2. <u>Floating Point number</u>

Floatig-point number holds numbers with fractional parts like 27.23. There are two kinds of floating-point number



- The float type values are single-precision numbers with a precision of seven digits.
- The double type values are double-precision numbers with a precision of 15/16 digits.

Type	Size
float	4 byte
double	8 byte

3. **Character type**

   The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c'. It is of 2 byte size

   Example

```
char myGrade = 'B';
```

4. **Boolean type**

   A boolean data type is declared with the bool keyword and can only take the values **true** or **false**. It uses 1 bit of storage.

   Example

```
bool a = true;

bool b  = false;
```

User-defined types

User-defined type are further subdivided into:

1. Enumeration
2. Structure

Enumeration

Enumeration (or enum) is a value data type in C#. It is mainly used to assign the names or string values to integral constants that make a program easy to read and maintain.

The general syntax for declaring an enumeration is −

```
enum <enum_name>
{
enumeration list
};
```

Where,

- The *enum_name* specifies the enumeration type name.
- The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value. By default, the value of the first enumeration symbol is 0.

Example

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Structure

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Example

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

Reference types

Reference types are stored on the heap and when an assignment of two reference variables occurs, only the reference is copied, the actual value remains in the same memory location. This means that there are two reference to a single value.

The Reference types of C# can be grouped into two categories

- Predefined types (primitive datatype)
- User-defined types (complex datatype)

Predefined types (primitive datatype)
Predefined type are further subdivided into:

1. Objects
2. Strings

Objects

- The **Object Type** is the base class for all data.
- The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.
- Before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing**

```
using System;
public class Program
 {
        public  static void Main()
        {
                int val = 2019;
                object o = val;     // Boxing
```

```
                val = 2000;

                Console.WriteLine("Value type of val is {0}", val);
                Console.WriteLine("Object type of val is {0}", o);
        }
}
```

Output:

```
Value type of val is 2000
Object type of val is 2019
```

When an object type is converted to a value type, it is called **unboxing**.

```
using System;
public class Program
{
public  static void Main()
        {
int val = 2019;
object o = val;                 // Boxing
int x = (int)o;                 // Unboxing
Console.WriteLine("Value of o is {0}", o);
Console.WriteLine("Value of x is {0}", x);
        }
}
```

Output:

```
Value of o is 2019
Value of x is 2019
```

String

The string data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
string greeting = "Hello World";
```

User-defined types

User-defined type are further subdivided into:

1. Class
2. Array
3. Delegates
4. Interfaces

Class

A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods into a single unit.\

Example:

```
public class Program
{


   // field variable
   public int a, b;


    // member function or method
    public void display()
    {
        Console.WriteLine("Class & Objects in C#");
    }
}
```

Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

```
int[] myNum;
```

We have now declared a variable that holds an array of integer

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
int[] myNum = {10, 20, 30, 40};
```

Delegates

A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods of any class. The only requirement is that its signature must match the signature of the method.

Syntax:

```
[access modifier] delegate [return type] [delegate name]([parameters])
```

Example:

```
public delegate void MyDelegate(string msg);
```

Interface

- Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods.
- It cannot have method body and cannot be instantiated.
- Its implementation must be provided by class or **struct.**

```
using System;
public interface Drawable
{
   void draw();
}
public class Rectangle : Drawable
{
   public void draw()
   {
 Console.WriteLine("drawing rectangle...");
   }
}
public class TestInterface
{
   public static void Main()
   {
      Drawable d;
      d = new Rectangle();
      d.draw();
      Console.ReadKey();
   }
}
```

Output:

```
drawing rectangle...
```