Session: 2

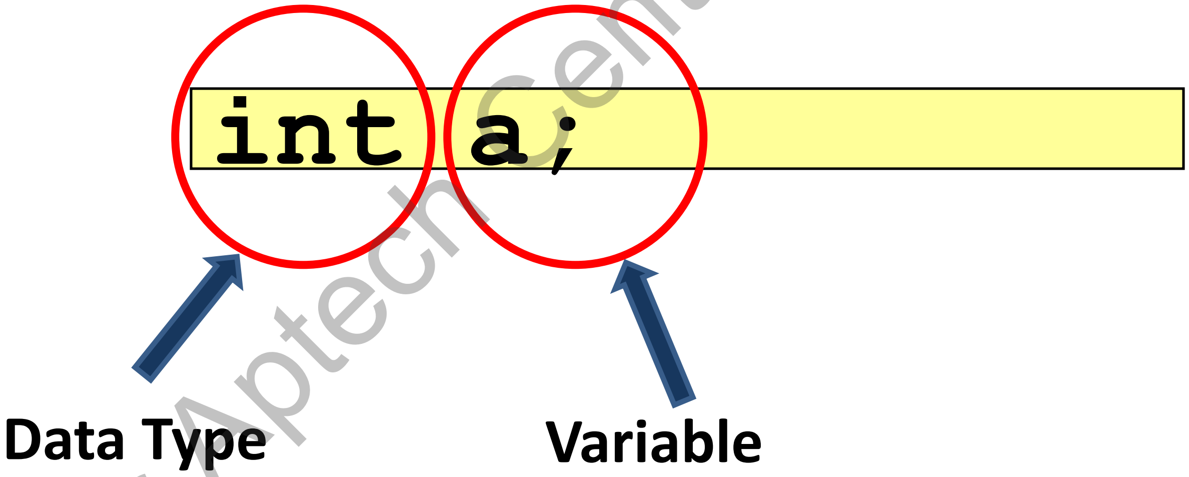
Variables and Data Types

Objectives

- Define and describe variables and data types in C#
- Explain comments and XML documentation
- Define and describe constants and literals
- List the keywords and escape sequences
- Explain input and output

Variables and Data Types in C#

- A variable is used to store data in a program and is declared with an associated data type.
- A variable has a name and may contain a value.
- A data type defines the type of data that can be stored in a variable.



```
int a;
```

Data Type

Variable

Definition

- A variable is an entity whose value can keep changing during the course of a program.

Example

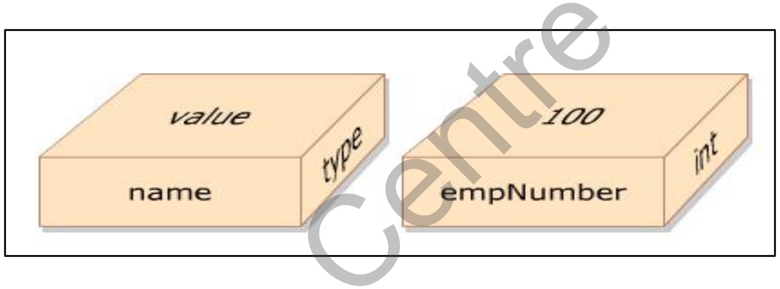
  - The age of a student, the address of a faculty member, and the salary of an employee are all examples of entities that can be represented by variables.

- In C#, a variable is a location in the computer's memory that is identified by a unique name and is used to store a value. The name of the variable is used to access and read the value stored in it.

- Different types of data such as a character, an integer, or a string can be stored in variables. Based on the type of data that needs to be stored in a variable, variables can be assigned different data types.

Using Variables 1-2

- In C#, memory is allocated to a variable at the time of its creation and a variable is given a name that uniquely identifies the variable within its scope.
- On initializing a variable, the value of a variable can be changed as required to keep track of data being used in a program. When referring to a variable, you are actually referring to the value stored in that variable.
- The following figure illustrates the concept of a variable:



- The following syntax is used to declare variables in C#:

Syntax

```
<datatype><variableName>;
```

where,

- `datatype`: Is a valid data type in C#.
- `variableName`: Is a valid variable name.

Using Variables 2-2

- The following syntax is used to initialize variables in C#:

Syntax

```
<variableName> = <value>;
```

- where,
  - =: Is the assignment operator used to assign values.
  - value: Is the data that is stored in the variable.
- The following code declares two variables, namely, **empNumber** and **empName**:

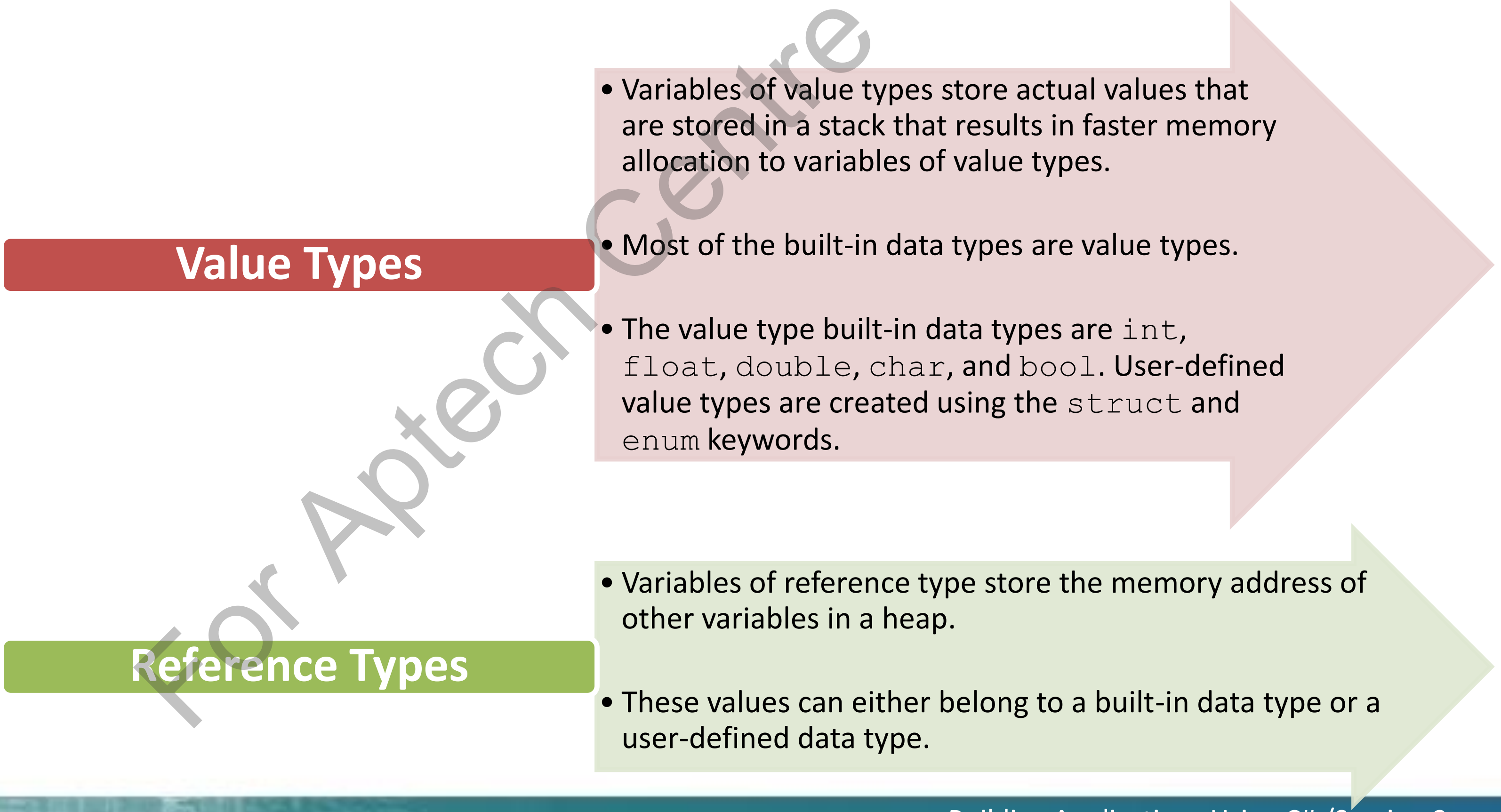
Snippet

```
int empNumber;
string empName;
```

- In the code:
  - an integer variable declares **empNumber**, and a string variable, **empName**. Memory is allocated to hold data in each variable.
  - Values can be assigned to variables by using the assignment operator (=), as follows:
    - empNumber = 100;
    - empName = “David Blake”;
  - You can also assign a value to a variable upon creation, as follows:
    - int empNumber = 100;

Data Types

- Different types of values such as numbers, characters, or strings can be stored in different variables. To identify the type of data that can be stored in a variable, C# provides different data types.
- When a variable is declared, a data type is assigned to the variable. This allows the variable to store values of the assigned data type.
- In C# programming language, data types are divided into two categories:



Value Types

- Variables of value types store actual values that are stored in a stack that results in faster memory allocation to variables of value types.
- Most of the built-in data types are value types.
- The value type built-in data types are `int`, `float`, `double`, `char`, and `bool`. User-defined value types are created using the `struct` and `enum` keywords.

Reference Types

- Variables of reference type store the memory address of other variables in a heap.
- These values can either belong to a built-in data type or a user-defined data type.

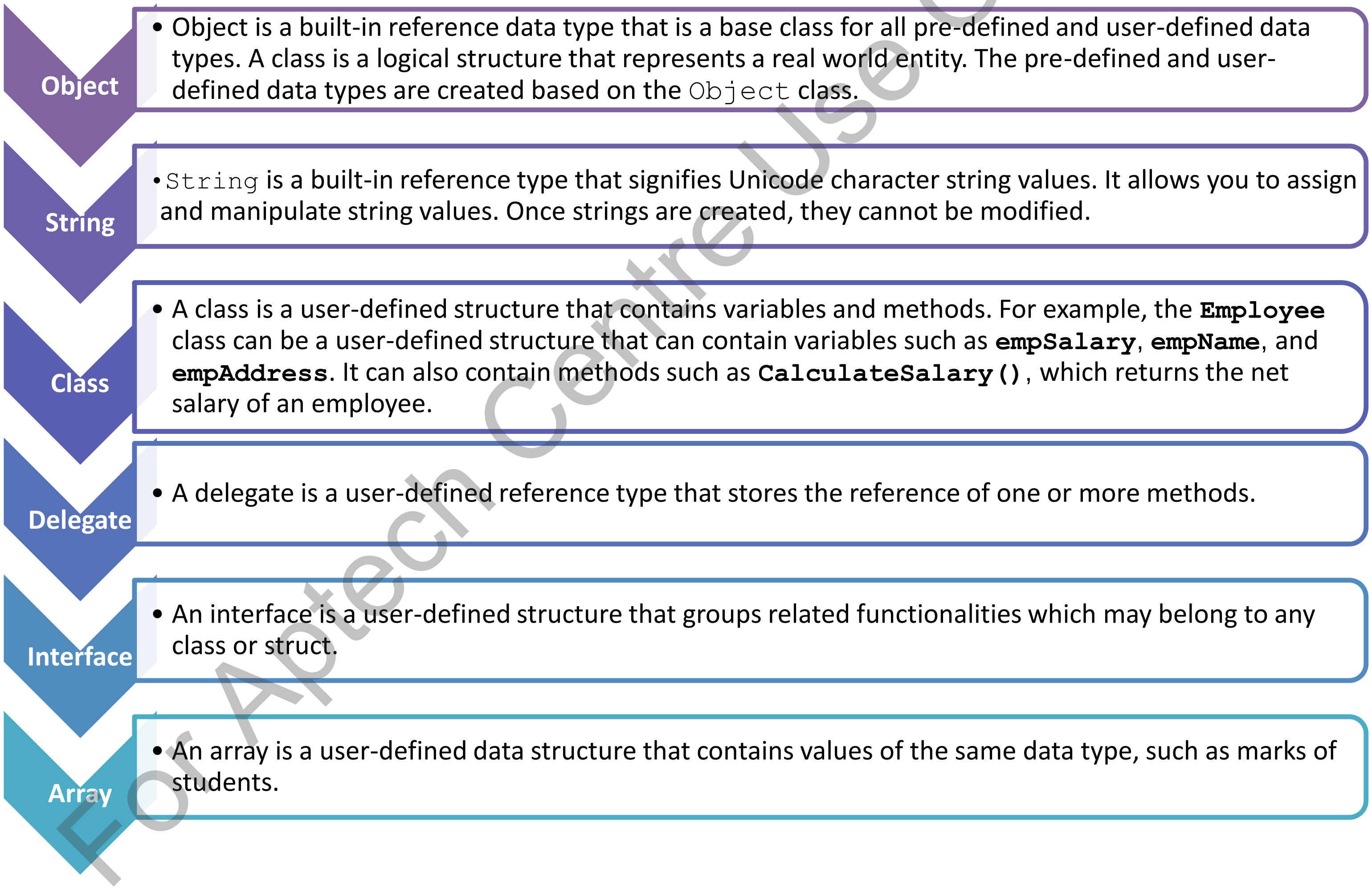
Pre-defined Data Types

- The pre-defined data types are referred to as basic data types in C# that have a pre-defined range and size.
- The size of the data type helps the compiler to allocate memory space and the range helps the compiler to ensure that the value assigned, is within the range of the variable's data type.
- The following table summarizes the pre-defined data types in C#:

Data Type	Size	Range
`byte`	Unsigned 8-bit integer	0 to 255
`sbyte`	Signed 8-bit integer	-128 to 127
`short`	Signed 16-bit integer	-32,768 to 32,767
`ushort`	Unsigned 16-bit integer	0 to 65,535
`int`	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
`uint`	Unsigned 32-bit integer	0 to 4,294,967,295
`long`	Signed 64-bit integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`ulong`	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
`float`	32-bit floating point with 7 digits precision	±1.5e−45 to ±3.4e38
`double`	64-bit floating point with 15-16 digits precision	±5.0e−324 to ±1.7e308
`decimal`	128-bit floating point with 28-29 digits precision	±1.0 × 10e−28 to ±7.9 × 10e28
`char`	Unicode 16-bit character	U+0000 to U+ffff
`bool`	Stores either true or false	true or false

Classification

- Reference data types store the memory reference of other variables that hold the actual values that can be classified into the following types:



Object

- Object is a built-in reference data type that is a base class for all pre-defined and user-defined data types. A class is a logical structure that represents a real world entity. The pre-defined and user-defined data types are created based on the `Object` class.

String

- `String` is a built-in reference type that signifies Unicode character string values. It allows you to assign and manipulate string values. Once strings are created, they cannot be modified.

Class

- A class is a user-defined structure that contains variables and methods. For example, the **`Employee`** class can be a user-defined structure that can contain variables such as **`empSalary`**, **`empName`**, and **`empAddress`**. It can also contain methods such as **`CalculateSalary()`**, which returns the net salary of an employee.

Delegate

- A delegate is a user-defined reference type that stores the reference of one or more methods.

Interface

- An interface is a user-defined structure that groups related functionalities which may belong to any class or struct.

Array

- An array is a user-defined data structure that contains values of the same data type, such as marks of students.

Rules

- A variable needs to be declared before it can be referenced by following certain rules as follows:

A variable name can begin with an uppercase or a lowercase letter. The name can contain letters, digits, and the underscore character (_).

The first character of the variable name must be a letter and not a digit. The underscore is also a legal first character, but it is not recommended at the beginning of a name.

C# is a case-sensitive language; hence, variable names count and Count refer to two different variables.

C# keywords cannot be used as variable names. If you still need to use a C# keyword, prefix it with the '@' symbol.

- It is always advisable to give meaningful names to variables such that the name gives an idea about the content that is stored in the variable.

Validity

- Mentioning a variable’s type and identifier (name) at the time of declaring a variable indicates to the compiler, the name of the variable and the type of data that will be stored.
- An undeclared variable generates an error message.
- The following table displays a list of valid and invalid variable names in C#:

Variable Name	Valid/Invalid
Employee	Valid
student	Valid
_Name	Valid
Emp_Name	Valid
@goto	Valid
static	Invalid as it is a keyword
4myclass	Invalid as a variable cannot start with a digit
Student&Faculty	Invalid as a variable cannot have the special character &

Declaration 1-3

- In C#, you can declare multiple variables at the same time in the same way you declare a single variable.
- After declaring variables, you need to assign values to them.
- Assigning a value to a variable is called initialization.
- You can assign a value to a variable while declaring it or at a later time.
- The following is the syntax to declare and initialize a single variable:

Syntax

```
<data type><variable name> = <value>;
```

where,

- `data type`: Is a valid variable type.
- `variable name`: Is a valid variable name or identifier.
- `value`: Is the value assigned to the variable.

Declaration 2-3

- The following is the syntax to declare multiple variables:

Syntax

```
<data type><variable name1>, <variable
name2>,..,<variable nameN>;
```

where,

- `data type`: Is a valid variable type.
- `variable name1,variable name2,variable nameN`: Are valid variable names or identifiers.

- The following is the syntax to declare and initialize multiple variables:

Syntax

```
<data <data type><variable name1> = <value1>,
<variable name2> = <value2>;
```

Declaration 3-3

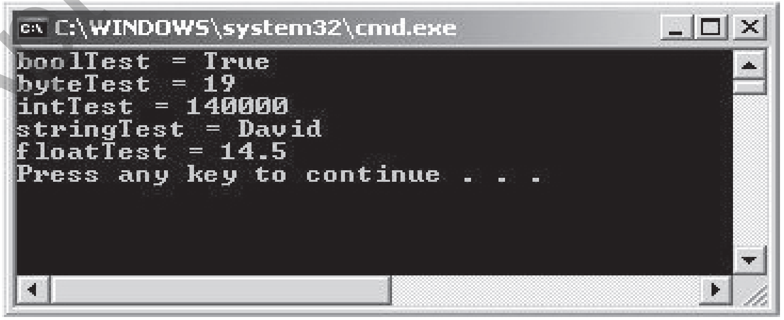
- The following code demonstrates how to declare and initialize variables in C#:

Snippet

```
bool boolTest = true;
short byteTest = 19;
int intTest;
string stringTest = “David”;
float floatTest;
int Test = 140000;
floatTest = 14.5f;
Console.WriteLine(“boolTest = {0}”, boolTest);
Console.WriteLine(“byteTest = ” + byteTest);
Console.WriteLine(“intTest = ” + intTest);
Console.WriteLine(“stringTest = ” + stringTest);
Console.WriteLine(“floatTest = ” + floatTest);
```

- In the code:
  - Variables of type `bool`, `byte`, `int`, `string`, and `float` are declared. Values are assigned to each of these variables and are displayed using the `WriteLine()` method of the `Console` class.
- The code displays the following output:

Output



```
boolTest = True
byteTest = 19
intTest = 140000
stringTest = David
floatTest = 14.5
Press any key to continue . . .
```

Implicitly Typed Variables

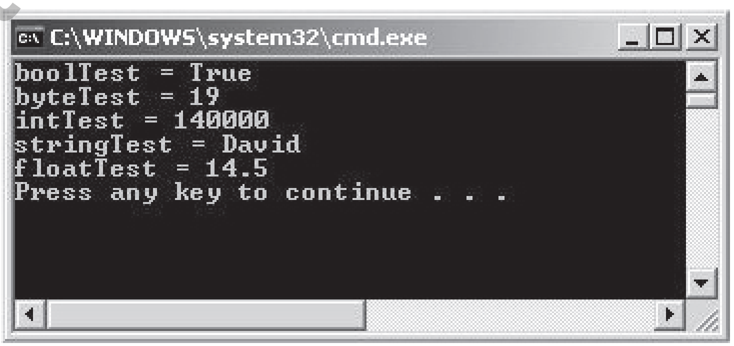
- When you declare and initialize a variable in a single step, you can use the `var` keyword in place of the type declaration. Variables declared using the `var` keyword are called implicitly typed variables. For implicitly typed variables, the compiler infers the type of the variable from the initialization expression. The following code demonstrates how to declare and initialize implicitly typed variables in C#:

Snippet

```
var boolTest = true;
var byteTest = 19;
var intTest =140000;
var stringTest = “David”;
var floatTest = 14.5f;
Console.WriteLine(“boolTest = {0}”, boolTest);
Console.WriteLine(“byteTest = ” + byteTest);
Console.WriteLine(“intTest = ” + intTest);
Console.WriteLine(“stringTest = ” + stringTest);
Console.WriteLine(“floatTest = ” + floatTest);
```

- In the code, four implicitly typed variables are declared and initialized with values. The values of each variable are displayed using the `WriteLine()` method of the `Console` class.

Output

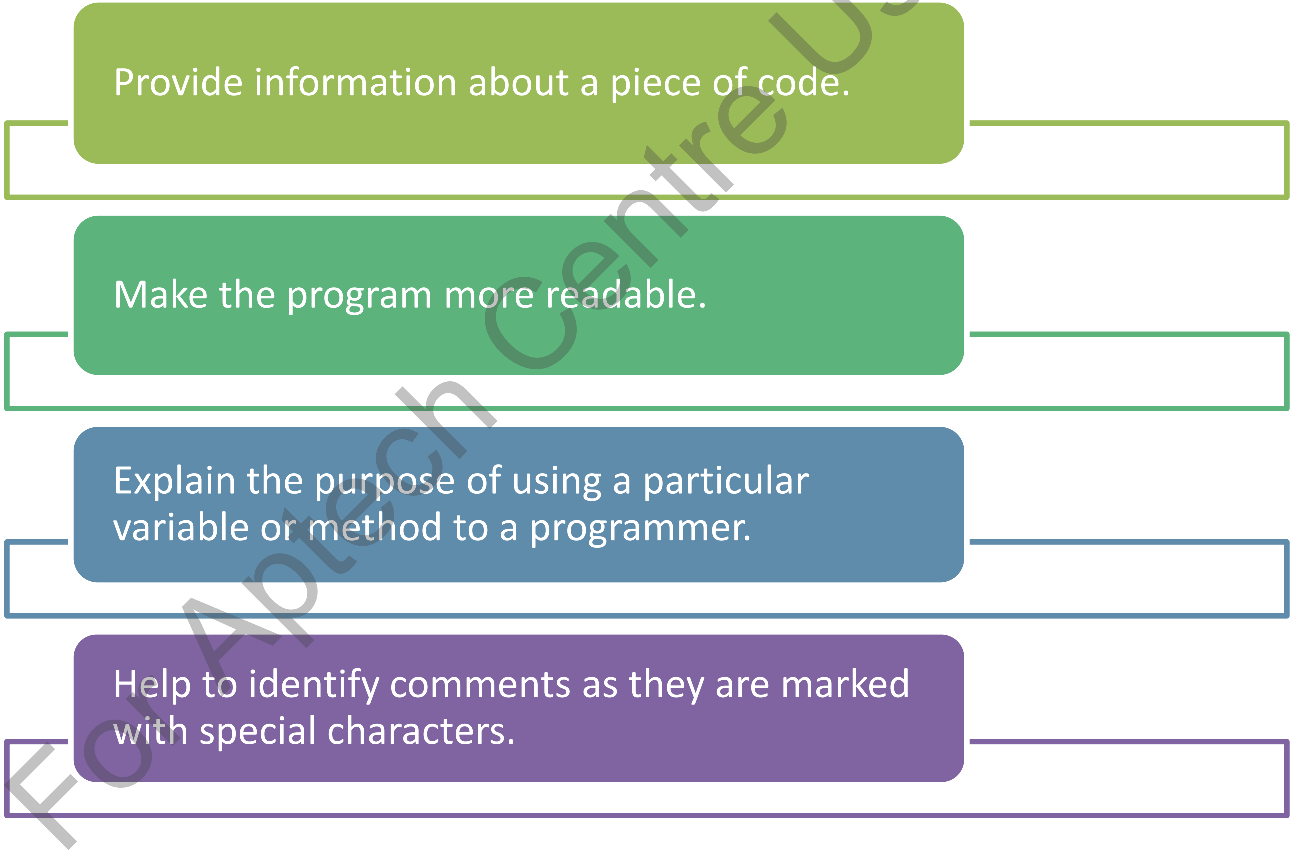


```
boolTest = True
byteTest = 19
intTest = 140000
stringTest = David
floatTest = 14.5
Press any key to continue . . .
```

Comments

- Comments help in reading the code of a program to understand the functionality of the program.

- In C#, comments are given by the programmer to:



Provide information about a piece of code.

Make the program more readable.

Explain the purpose of using a particular variable or method to a programmer.

Help to identify comments as they are marked with special characters.

- Comments are ignored by the compiler, during the execution of the program.

Types of Comments 1-2

- **Single-line Comments**: Begin with two forward slashes (//).

Snippet

```
// This block of code will add two numbers
int doSum = 4 + 3;
```

  - To write more than one line as comment, begin each line with the double slashes // characters, as shown in the following code:

```
 // This block of code will add two numbers and then put the result in the
// variable, doSum
 int doSum = 4 + 3;
```

  - You can also write the single-line comment in the same line as shown in the following code:

```
int doSum = 4 + 3; // Adding two numbers
```

- **Multi-line Comments**: Begin with a forward slash followed by an asterisk (/*) and end with an asterisk followed by a forward slash (*/).

Snippet

```
/* This is a block of code that will multiply two numbers,
divide the resultant value by 2 and display the quotient */
int doMult = 5 * 20;
int doDiv = doMult / 2;
Console.WriteLine(“Quotient is:” + doDiv)
```

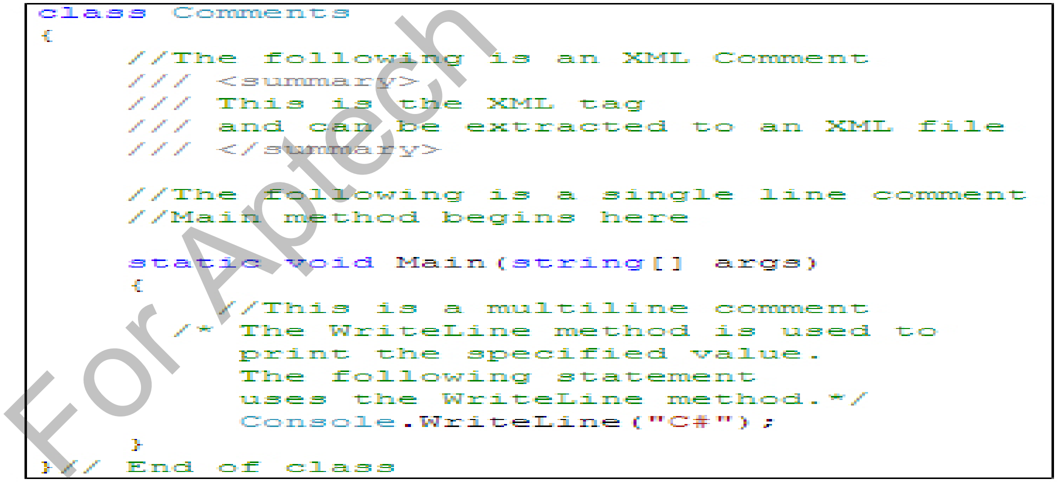
Types of Comments 2-2

- **XML Comments**: Begin with three forward slashes (///). Unlike single-line and multi-line comments, the XML comment must be enclosed in an XML tag. You need to create XML tags to insert XML comments. Both the XML tags and XML comments must be prefixed with three forward slashes.
  - You can insert an XML comment, as shown in the following code:

Snippet

```
/// <summary>
/// You are in the XML tag called summary.
/// </summary>
```

  - The following figure displays a complete example of using XML comments:



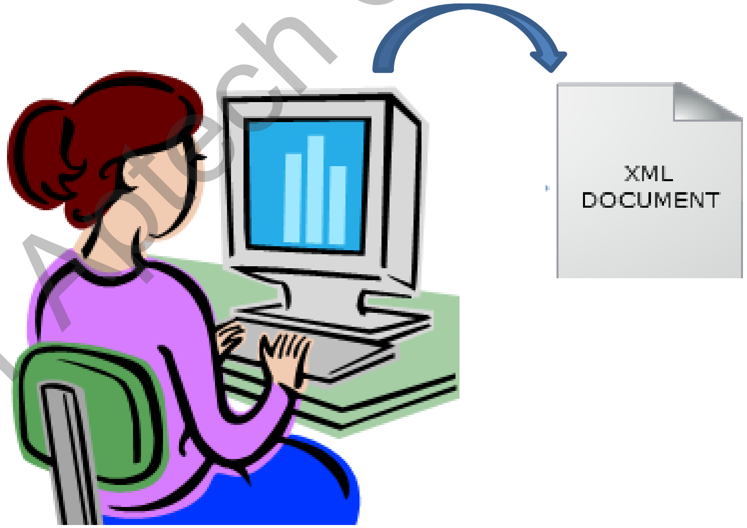
```
class Comments
{
    //The following is an XML Comment
    /// <summary>
    /// This is the XML tag
    /// and can be extracted to an XML file
    /// </summary>

    //The following is a single line comment
    //Main method begins here

    static void Main(string[] args)
    {
        //This is a multiline comment
      /* The WriteLine method is used to
         print the specified value.
         The following statement
         uses the WriteLine method.*/
         Console.WriteLine("C#");
    }
}// End of class
```

XML Documentation 1-2

- In C#, you can create an XML document that will contain all the XML comments.
- This document is useful when multiple programmers want to view information of the program.

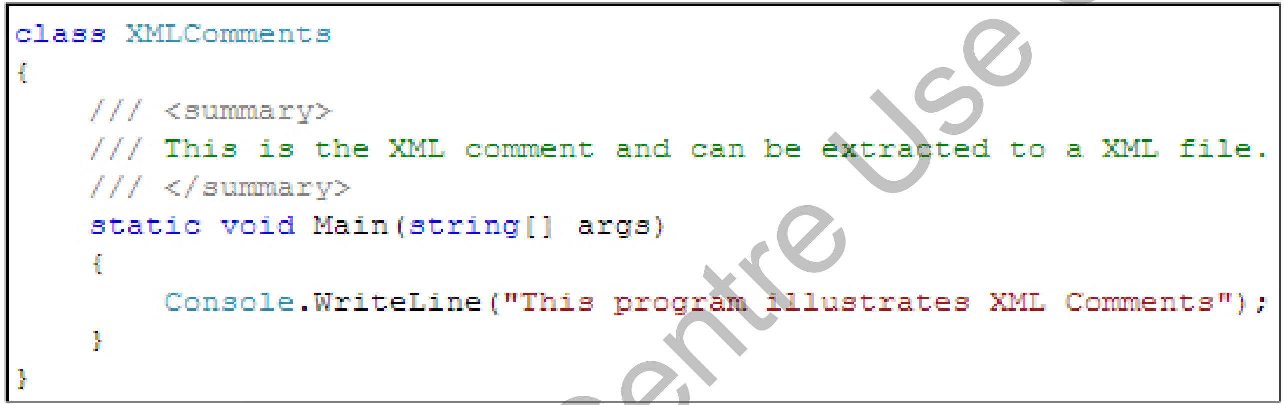
Example

- Consider a scenario, where one of the programmers wants to understand the technical details of the code and another programmer wants to see the total variables used in the code.
- In this case, you can create an XML document that will contain all the required information.



XML Documentation 2-2

- The following code displays the XML comments that can be extracted to an XML file:



```
class XMLComments
{
    /// <summary>
    /// This is the XML comment and can be extracted to a XML file.
    /// </summary>
    static void Main(string[] args)
    {
        Console.WriteLine("This program illustrates XML Comments");
    }
}
```

- The following syntax is used to create an XML document from the C# source file Visual Studio 2012 Command Prompt window:

Syntax

```
csc /doc: <XMLfilename.xml><CSharpfilename.cs>
```

where,

- **`XMLfilename.xml`**: Is the name of the XML file that is to be created.
- **`CSharpfilename.cs`**: Is the name of the C# file from where the XML comments will be extracted.

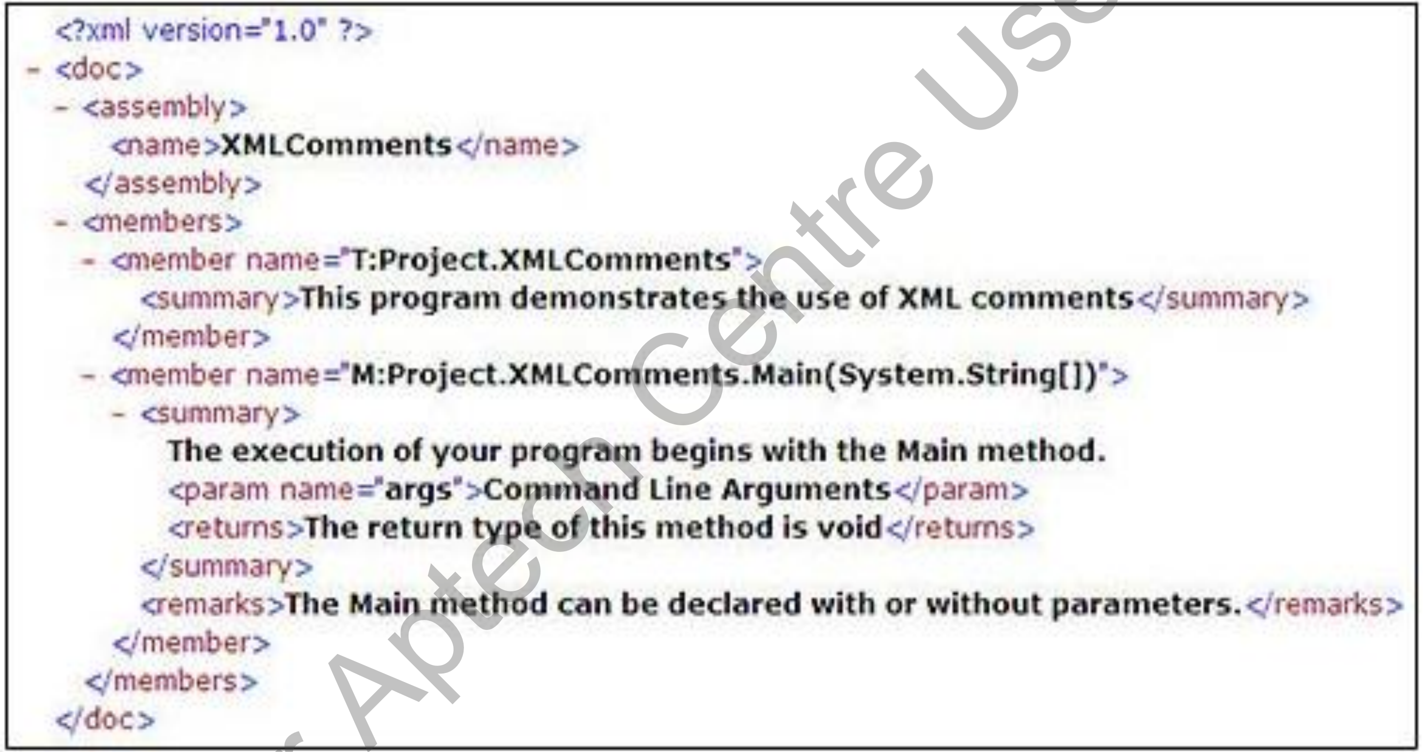
Pre-defined XML Tags 1-6

- XML comments are inserted in XML tags that can either be pre-defined or user-defined. The following table lists the widely used pre-defined XML tags and states their conventional use:

Pre-defined Tags	Descriptions
`<c>`	Sets text in a code-like font.
`<code>`	Sets one or more lines of source code or program output.
`<example>`	Indicates an example.
`<param>`	Describes a parameter for a method or a constructor.
`<returns>`	Specifies the return value of a method.
`<summary>`	Summarizes the general information of the code.
`<exception>`	Documents an exception class.
`<include>`	Refers to comments in another file using the XPath syntax, which describes the types and members in the source code.
`<list>`	Inserts a list into the documentation file.
`<para>`	Inserts a paragraph into the documentation file.
`<paramref>`	Indicates that a word is a parameter.
`<permission>`	Documents access permissions.
`<remarks>`	Specifies overview information about the type.
`<see>`	Specifies a link.
`<seealso>`	Specifies the text that might be required to appear in a See Also section.
`<value>`	Describes a property.

Pre-defined XML Tags 2-6

- The following figure displays an example of pre-defined XML tags:



```
  <?xml version="1.0" ?>
- <doc>
  - <assembly>
      <name>XMLComments</name>
    </assembly>
  - <members>
    - <member name="T:Project.XMLComments">
        <summary>This program demonstrates the use of XML comments</summary>
      </member>
    - <member name="M:Project.XMLComments.Main(System.String[])">
      - <summary>
          The execution of your program begins with the Main method.
          <param name="args">Command Line Arguments</param>
          <returns>The return type of this method is void</returns>
        </summary>
        <remarks>The Main method can be declared with or without parameters.</remarks>
      </member>
    </members>
  </doc>
```

Pre-defined XML Tags 3-6

- The following code demonstrates the use of XML comments:

Snippet

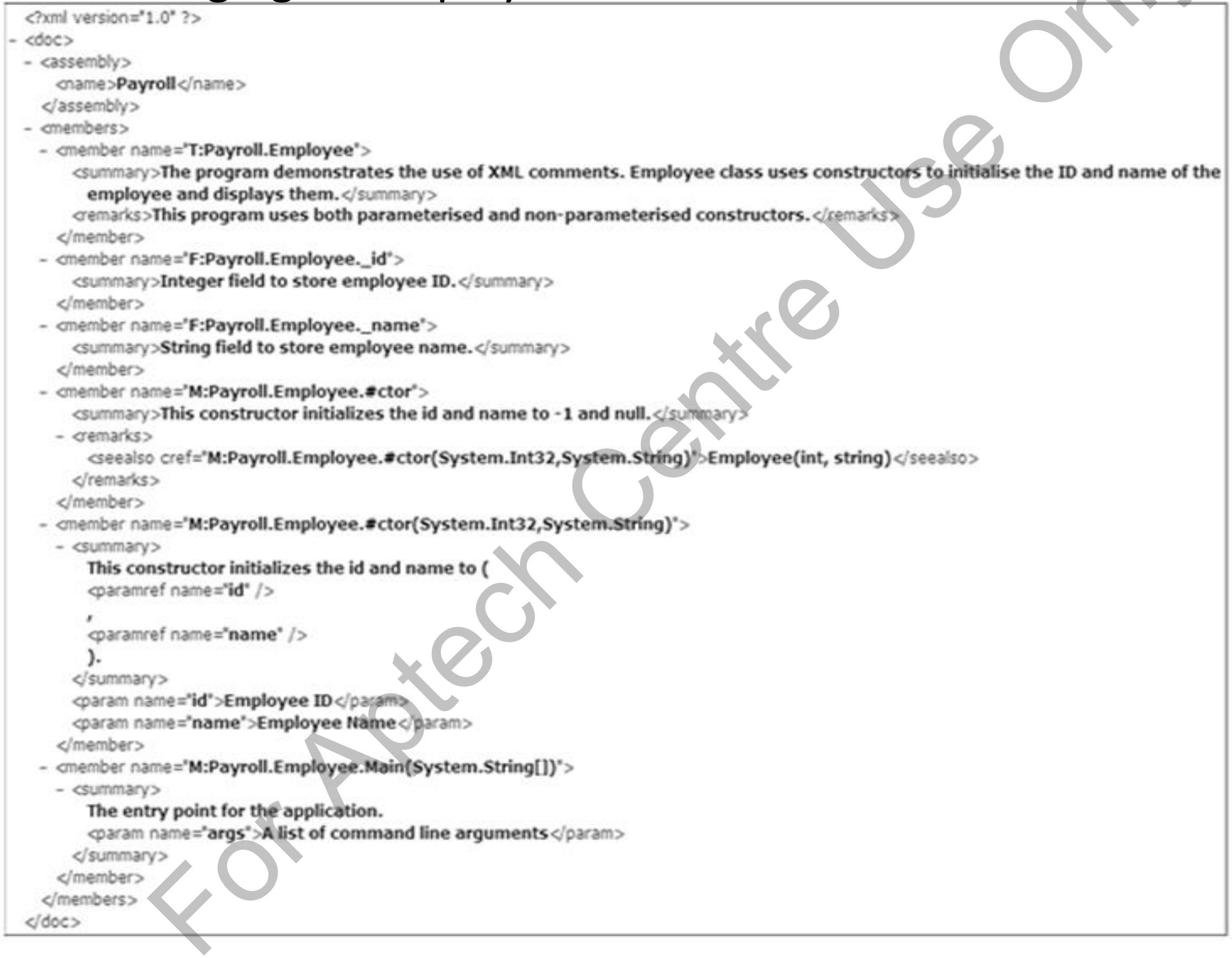
```
using System;
/// <summary>
/// The program demonstrates the use of XML comments.
///
/// Employee class uses constructors to initialize the ID and
/// name of the employee and displays them.
/// </summary>
/// <remarks>
/// This program uses both parameterized and
/// non-parameterized constructors.
/// </remarks>
class Employee
{
/// <summary>
/// Integer field to store employee ID.
/// </summary>
privateint _id;
/// <summary>
/// String field to store employee name.
/// </summary>
private string _name;
/// <summary>
/// This constructor initializes the id and name to -1 and null.
/// </summary>
/// <remarks>
/// <seealsocref="Employee(int,string)">Employee(int,string)</ seealso>
/// </remarks>
```

Pre-defined XML Tags 4-6

```
private string _name;
/// <summary>
/// This constructor initializes the id and name to -1 and null.
/// </summary>
/// <remarks>
/// <seealsocref="Employee(int,string)">Employee(int,string)</ seealso>
/// </remarks>
public Employee()
{
_id = -1;
_name = null;
}
/// <summary>
/// This constructor initializes the id and name to
/// (<paramref name="id"/>,<paramref name="name"/>).
/// </summary>
/// <param name="id">Employee ID</param>
/// <param name="name">Employee Name</param>
public Employee(int id, string name)
{
this._id = id;
this._name = name; }
/// <summary>
/// The entry point for the application.
/// <param name="args">A list of command line arguments</param>
/// </summary>
static void Main(string[] args)
{// Creating an object of Employee class and displaying the
// id and name of the employee
Employee objEmp = new Employee(101, "David Smith");
Console.WriteLine("Employee ID : {0} \nEmployee Name : {1}",
objEmp._name); }
```

Pre-defined XML Tags 5-6

- The following figure displays the XML document:



```
  <?xml version="1.0" ?>
- <doc>
  - <assembly>
      <name>Payroll</name>
    </assembly>
  - <members>
    - <member name="T:Payroll.Employee">
        <summary>The program demonstrates the use of XML comments. Employee class uses constructors to initialise the ID and name of the
          employee and displays them.</summary>
        <remarks>This program uses both parameterised and non-parameterised constructors.</remarks>
      </member>
    - <member name="F:Payroll.Employee._id">
        <summary>Integer field to store employee ID.</summary>
      </member>
    - <member name="F:Payroll.Employee._name">
        <summary>String field to store employee name.</summary>
      </member>
    - <member name="M:Payroll.Employee.#ctor">
        <summary>This constructor initializes the id and name to -1 and null.</summary>
      - <remarks>
          <seealso cref="M:Payroll.Employee.#ctor(System.Int32,System.String)">Employee(int, string)</seealso>
        </remarks>
      </member>
    - <member name="M:Payroll.Employee.#ctor(System.Int32,System.String)">
      - <summary>
          This constructor initializes the id and name to (
          <paramref name="id" />
          ,
          <paramref name="name" />
          ).
        </summary>
        <param name="id">Employee ID</param>
        <param name="name">Employee Name</param>
      </member>
    - <member name="M:Payroll.Employee.Main(System.String[])">
      - <summary>
          The entry point for the application.
          <param name="args">A list of command line arguments</param>
        </summary>
      </member>
    </members>
  </doc>
```

Pre-defined XML Tags 6-6

- In the code:
  - The `<remarks>`, `<seealso>`, and `<paramref>` XML documentation tags are used.
  - The `<remarks>` tag is used to provide information about a specific class.
  - The `<seealso>` tag is used to specify the text that should appear in the See Also section.
  - The `<paramref>` tag is used to indicate that the specified word is a parameter.

Constants 1-3

- A constant has a fixed value that remains unchanged throughout the program while a literal provides a mean of expressing specific values in a program.

Example

- Consider a code that calculates the area of the circle.
- To calculate the area of the circle, the value of pi and radius must be provided in the formula.
- The value of pi is a constant value.
- This value will remain unchanged irrespective of the value of the radius provided.
- Similarly, constants in C# are fixed values assigned to identifiers that are not modified throughout the execution of the code.
- They are defined when you want to preserve values to reuse them later or to prevent any modification to the values.

Constants 2-3

- In C#, you can declare constants for all data types.
- You have to initialize a constant at the time of its declaration.
- Constants are declared for value types rather than for reference types.
- To declare an identifier as a constant, the `const` keyword is used in the identifier declaration. The compiler can identify constants at the time of compilation, because of the `const` keyword.
- The following syntax is used to initialize a constant:

Syntax

```
const<data type><identifier name> = <value>;
```

where,

- **`const`**: Keyword denoting that the identifier is declared as constant.
- **`data type`**: Data type of constant.
- **`identifier name`**: Name of the identifier that will hold the constant.
- **`value`**: Fixed value that remains unchanged throughout the execution of the code.

Constants 3-3

- The following code declares a constant named `_pi` and a variable named `radius` to calculate the area of the circle:

Snippet

```
const float _pi = 3.14F;
float radius = 5;
float area = _pi * radius * radius;
Console.WriteLine(“Area of the circle is ” + area);
```

- In the code:
  - A constant called `_pi` is assigned the value 3.14, which is a fixed value. The variable, `radius`, stores the radius of the circle. The code calculates the area of the circle and displays it as the output.

Literals

- A literal is a static value assigned to variables and constants.
- Numeric literals might suffix with a letter of the alphabet to indicate the data type of the literal.
- This letter can be either in upper or lowercase.

Example

For example, in the following declaration,

```
string bookName = “Csharp”
```

`Csharp` is a literal assigned to the variable **`bookName`** of type string.

Types of Literals 1-2

- **Boolean Literal:** Boolean literals have two values, `true` or `false`. For example,

```
boolval = true;
```

where,

`true`: Is a Boolean literal assigned to the variable `val`.

- **Integer Literal:** An integer literal can be assigned to `int`, `uint`, `long`, or `ulongdata` types. Suffixes for integer literals include U, L, UL, or LU. U denotes `uint` or `ulong`, L denotes `long`. UL and LU denote `ulong`. For example,

```
longval = 53L;
```

where,

`53L`: Is an integer literal assigned to the variable `val`.

- **Real Literal:** A real literal is assigned to `float`, `double (default)`, and `decimal` data types. This is indicated by the suffix letter appearing after the assigned value. A real literal can be suffixed by F, D, or M. F denotes `float`, D denotes `double`, and M denotes `decimal`. For example,

```
floatval = 1.66F;
```

where,

`1.66F`: Is a real literal assigned to the variable `val`.

Types of Literals 2-2

- **Character Literal:** A character literal is assigned to a char data type. A character literal is always enclosed in single quotes. For example,

```
charval = ‘A’;
```

where,

A: Is a character literal assigned to the variable val.

- **String Literal:** There are two types of string literals in C#, regular and verbatim. A regular string literal is a standard string. A verbatim string literal is similar to a regular string literal but is prefixed by the ‘@’ character. A string literal is always enclosed in double quotes. For example,

```
stringmailDomain = “@gmail.com”;
```

where,

@gmail.com: Is a verbatim string literal.

- **Null Literal:** The null literal has only one value, null. For example,

```
string email = null;
```

where,

null: Specifies that e-mail does not refer to any objects (reference).

Keywords and Escape Sequences 1-3

- Keywords are reserved words that are separately compiled by the compiler and convey a pre-defined meaning to the compiler and hence, cannot be created or modified.

Example

`int` is a keyword that specifies that the variable is of data type integer.

- Escape sequence characters in C# are characters preceded by a backslash (\) and denote a special meaning to the compiler.

- You cannot use keywords as variable names, method names, or class names, unless you prefix the keywords with the ‘@’ character.

Keywords and Escape Sequences 2-3

- The following table lists the keywords used in C#:

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	implicitin	int	Interface
internal	is	lock	long	namespace	new	null
object	operator	out	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	virtual	void	volatile	while

Keywords and Escape Sequences 3-3

- C# provides contextual keywords that have special meaning in the context of the code, where they are used.
- The contextual keywords are not reserved and can be used as identifiers outside the context of the code.
- When new keywords are added to C#, they are added as contextual keywords.
- The following table lists the contextual keywords used in C#:

`add`	`alias`	`ascending`	`async`	`await`	`descending` `yield`	`dynamic`
`from`	`get`	`global`	`group`	`into`	`join`	`let`
`orderby`	`partial`	`remove`	`select`	`set`	`value`	`var`
`where`						

Need of Escape Sequence Characters

Example

- Consider a payroll system of an organization.
- One of its functions is to display the monthly salary as output with the salary displayed on the next line.
- The programmer wants to write the code in such a way that the salary is always printed on the next line irrespective of the length of string to be displayed with the salary amount.
- This is done using escape sequences.

Definition of Escape Sequences

- An escape sequence character is a special character that is prefixed by a backslash (\) and are used to implement special non-printing characters such as a new line, a single space, or a backspace.
- These non-printing characters are used while displaying formatted output to the user to maximize readability.
- The backslash character tells the compiler that the following character denotes a non-printing character.

Example

- `\n` is used to insert a new line similar to the `Enter` key of the keyboard.
- In C#, the escape sequence characters must always be enclosed in double quotes.

Escape Sequence Characters in C# 1-3

- There are multiple escape sequence characters in C# that are used for various kinds of formatting. The following table displays the escape sequence characters and their corresponding non-printing characters in C#:

Escape Sequence Characters	Non-Printing Characters
\’	Single quote, needed for character literals.
\”	Double quote, needed for string literals.
\\	Backslash, needed for string literals.
\0	Unicode character 0.
\a	Alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\v	Vertical tab.
\t	Horizontal tab.
\?	Literal question mark.
\ooo	Matches an ASCII character, using a three-digit octal character code.
\xhh	Matches an ASCII character, using hexadecimal representation (exactly two digits). For example, \x61 represents the character ‘a’.
\uhhhh	Matches a Unicode character, using hexadecimal representation (exactly four digits). For example, the character \u0020 represents a space.

Escape Sequence Characters in C# 2-3

- The following code demonstrates the use of Unicode characters:

Snippet

```
string str = “\u0048\u0065\u006C\u006C\u006F”;
Console.Write(“\t” + str + “!\n”);
Console.WriteLine(“David\u0020\”2007\” “);
```

Output

```
Hello!
David “2007”
```

- In the code:
  - The variable `str` is declared as type `string` and stores Unicode characters for the letters H, e, l, l, and o. The method uses the horizontal tab escape sequence character to display the output leaving one tab space.
  - The new line escape sequence character used in the string of the method displays the output of the next statement in the next line.
  - The next statement uses the `WriteLine()` method to display `David “2007”`. The string in the method specifies the Unicode character to display a space between `David` and `2007`.

Escape Sequence Characters in C# 3-3

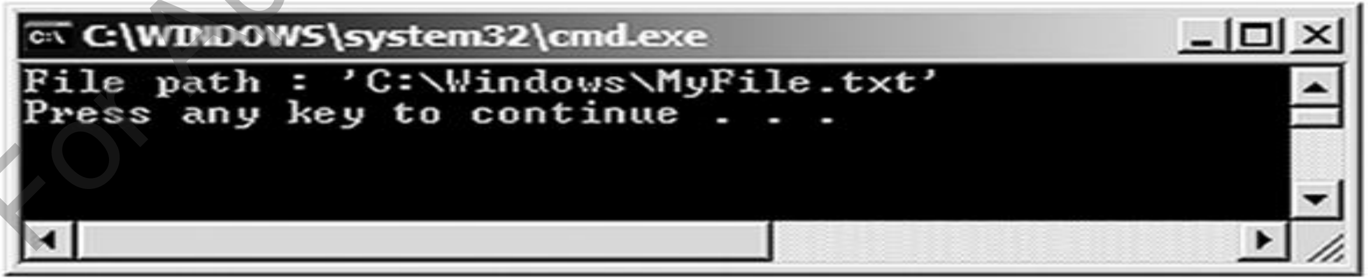
- The following code demonstrates the use of some of the commonly used escape sequences:

Snippet

```
using System;
classFileDemo
{
   static void Main(string[] args)
   {
        string path = "C:\\Windows\\MyFile.txt";
        bool found = true;
        if (found)
        {
            Console.WriteLine("File path : \'" + path + "\'");
        }
        else
        {
           Console.WriteLine("File Not Found!\a");
        }
   }
}
```

- In this code:
  - The \\, \', and \a escape sequences are used. The \\ escape sequence is used for printing a backslash.
  - The \' escape sequence is used for printing a single quote. The \a escape sequence is used for producing a beep.
- The following figure displays the output of the example of escape sequences:

Output



```
File path : 'C:\Windows\MyFile.txt'
Press any key to continue . . .
```

Input and Output

- Programmers often need to display the output of a C# program to users.
- The programmer can use the command line interface to display the output.
- The programmer can similarly accept inputs from a user through the command line interface.
- Such input and output operations are also known as console operations.

Console Operations 1-2

- Following are the features of the console operations:

Console operations are tasks performed on the command line interface using executable commands.

The console operations are used in software applications because these operations are easily controlled by the operating system.

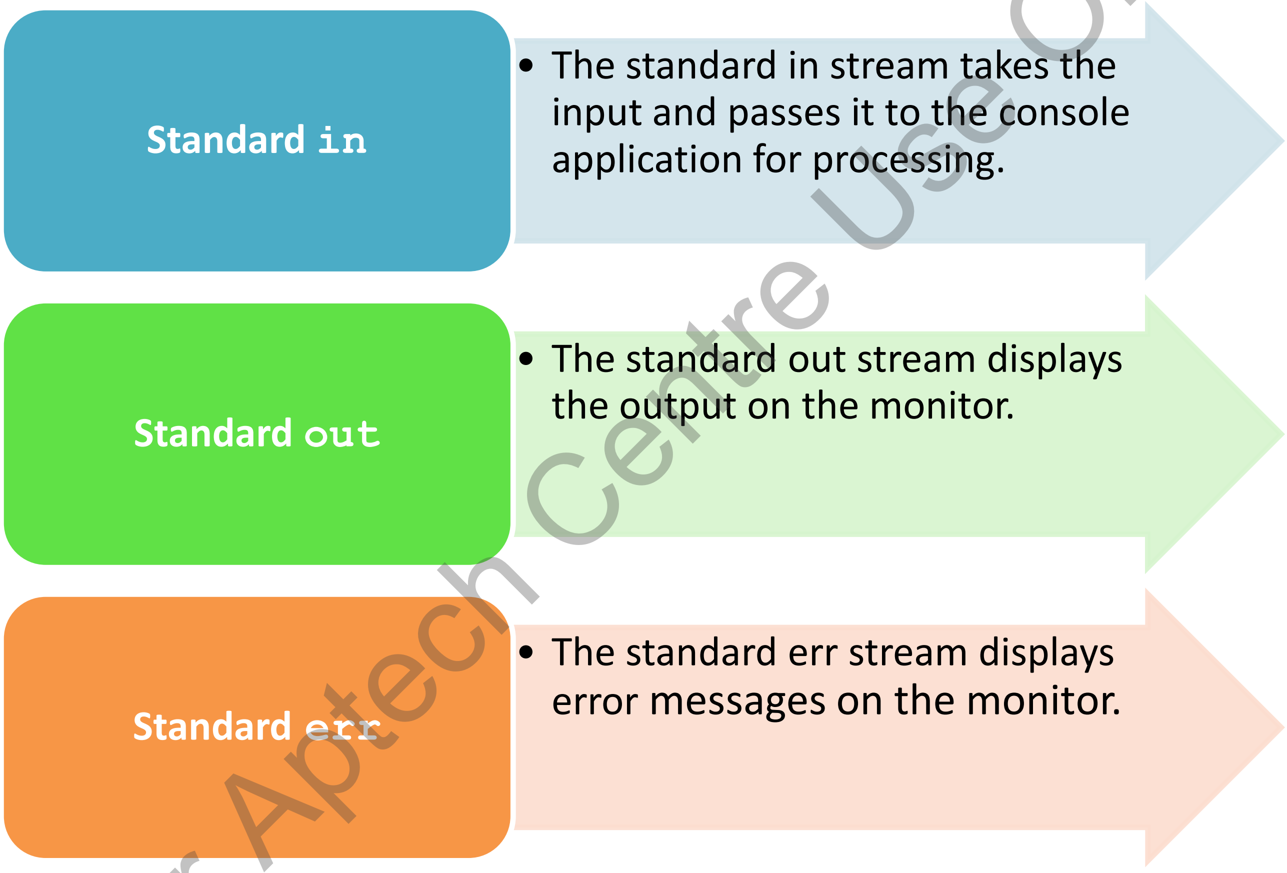
This is because console operations are dependent on the input and output devices of the computer system.

A console application is one that performs operations at the command prompt.

All console applications consist of three streams, which are a series of bytes. These streams are attached to the input and output devices of the computer system and they handle the input and output operations.

Console Operations 2-2

- The three streams are as follows:



Standard `in`

- The standard in stream takes the input and passes it to the console application for processing.

Standard `out`

- The standard out stream displays the output on the monitor.

Standard `err`

- The standard err stream displays error messages on the monitor.

Output Methods 1-3

- In C#, all console operations are handled by the `Console` class of the `System` namespace.

- A namespace is a collection of classes having similar functionalities.

- To write data on the console, you need the standard output stream, provided by the `Console` class.

- There are two output methods that write to the standard output stream as follows:
  - `Console.Write()`: Writes any type of data.
  - `Console.WriteLine()`: Writes any type of data and this data ends with a new line character in the standard output stream. This means any data after this line will appear on the new line.

Output Methods 2-3

- The following syntax is used for the `Console.Write()` method, which allows you to display the information on the console window:

Syntax

```
Console.Write(“<data>” + variables);
```

where,

- `data`: Specifies strings or escape sequence characters enclosed in double quotes.
- `variables`: Specify variable names whose value should be displayed on the console.

Output Methods 3-3

- The following syntax is used for the `Console.WriteLine()` method, which allows you to display the information on a new line in the console window:

Syntax

```
Console.WriteLine(“<data>” + variables);
```

- The following code shows the difference between the `Console.Write()` and `Console.WriteLine()` methods:

Snippet

```
Console.WriteLine(“C# is a powerful programming language”);
Console.WriteLine(“C# is a powerful”);
Console.WriteLine(“programming language”);
Console.Write(“C# is a powerful”);
Console.WriteLine(“ programming language”);
```

Output

```
C# is a powerful programming language
C# is a powerful
programming language
C# is a powerful programming language
```

Placeholders

- The `WriteLine()` and `Write()` methods accept a list of parameters to format text before displaying the output.

- The first parameter is a string containing markers in braces to indicate the position, where the values of the variables will be substituted. Each marker indicates a zero-based index based on the number of variables in the list.

- The following code uses placeholders in the `Console.WriteLine()` method to display the result of the multiplication operation:

Snippet

```
int number, result;
number = 5;
result = 100 * number;
Console.WriteLine(“Result is {0} when 100 is multiplied by {1}”,
result,number);
result = 150 / number;
Console.WriteLine(“Result is {0} when 150 is divided by {1}”, +result,
number);
```

Output

```
Result is 500 when 100 is multiplied by 5
Result is 30 when 150 is divided by 5
```

Here, `{0}` is replaced with the value in result and `{1}` is replaced with the value in number.

Input Methods 1-3

- In C#, to read data, you need the standard input stream. This stream is provided by the input methods of the `Console` class. There are two input methods that enable the software to take in the input from the standard input stream.
- These methods are as follows:
  - `Console.Read()`: Reads a single character.
  - `Console.ReadLine()`: Reads a line of strings.
- The following code reads the name using the `ReadLine()` method and displays the name on the console window:

Snippet

```
string name;
Console.Write(“Enter your name: “);
name = Console.ReadLine();
Console.WriteLine(“You are {0}”,name);
```

- In the code:
  - The `ReadLine()` method reads the name as a string and the string that is given is displayed as output using placeholders.

Output

```
Enter your name: David Blake
You are David Blake
```

Input Methods 2-3

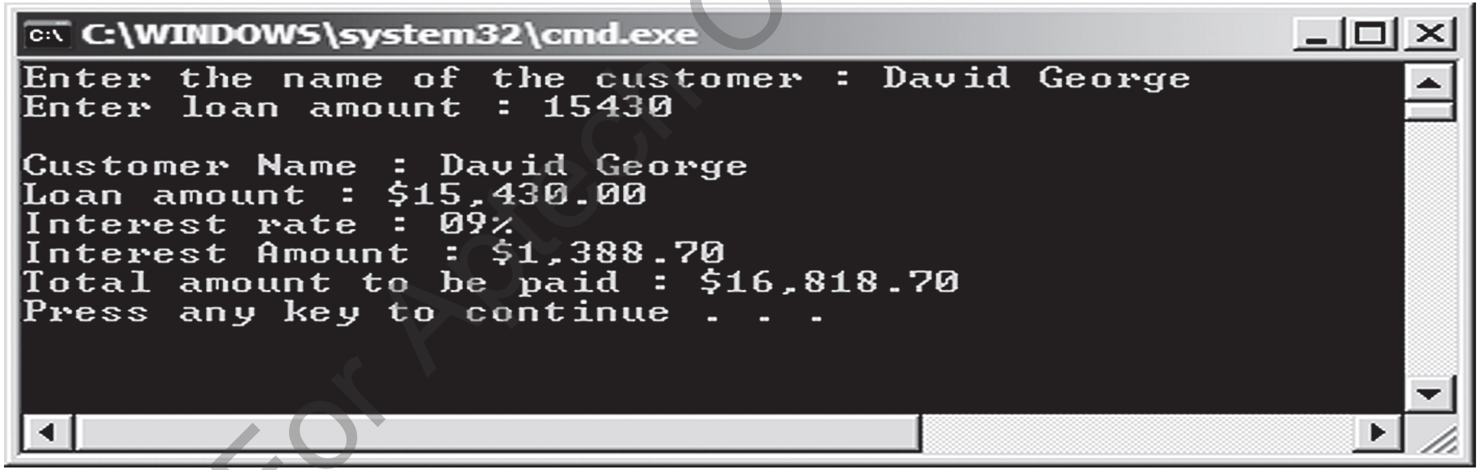
- The following code demonstrates the use of placeholders in the `Console.WriteLine()` method:

Snippet

```
using System;
class Loan
{
   static void Main(string[] args)
   {
      stringcustName;
      doubleloanAmount;
      float interest = 0.09F;
      doubleinterestAmount = 0;
      doubletotalAmount = 0;
      Console.Write("Enter the name of the customer : ");
      custName = Console.ReadLine();
      Console.Write("Enter loan amount : ");
      loanAmount = Convert.ToDouble(Console.ReadLine());
      interestAmount = loanAmount * interest;
      totalAmount = loanAmount + interestAmount;
      Console.WriteLine("\nCustomer Name : {0}", custName);
      Console.WriteLine("Loan amount : ${0:#,###.#0} \nInterest rate :
      {1:0#%} \nInterest Amount : ${2:#,###.#0}",
      loanAmount, interest, interestAmount );
      Console.WriteLine("Total amount to be paid : ${0:#,###.#0} ",
      totalAmount);
   }
}
```

Input Methods 3-3

- In the code:
  - The name and loan amount are accepted from the user using the `Console.ReadLine()` method. The details are displayed on the console using the `Console.Writeline()` method.
  - The placeholders {0}, {1}, and {2} indicate the position of the first, second, and third parameters respectively.
  - The 0 specified before # pads the single digit value with a 0. The #option specifies the digit position.
  - The % option multiplies the value by 100 and displays the value along with the percentage sign.
- The following figure displays the example of placeholders:



```
Enter the name of the customer : David George
Enter loan amount : 15430

Customer Name : David George
Loan amount : $15,430.00
Interest rate : 09%
Interest Amount : $1,388.70
Total amount to be paid : $16,818.70
Press any key to continue . . .
```

Convert Methods

- The ReadLine() method can also be used to accept integer values from the user. The data is accepted as a string and then converted into the int data type. C# provides a Convert class in the System namespace to convert one base data type to another base data type.
- The following code reads the name, age, and salary using the Console.ReadLine() method and converts the age and salary into int and double using the appropriate conversion methods of the Convert class:

Snippet

```
string userName;
int age;
double salary;
Console.Write(“Enter your name: “);
userName = Console.ReadLine();
Console.Write(“Enter your age: “);
age = Convert.ToInt32(Console.ReadLine());
Console.Write(“Enter the salary: “);
salary = Convert.ToDouble(Console.ReadLine());
Console.WriteLine(“Name: {0}, Age: {1}, Salary: {2} “, userName, age, salary);
```

Output

```
Enter your name: David Blake
Enter your age: 34
Enter the salary: 3450.50
Name: David Blake, Age: 34, Salary: 3450.50
```

Numeric Format Specifiers

- Format specifiers are special characters that are used to display values of variables in a particular format. For example, you can display an octal value as decimal using format specifiers.
- In C#, you can convert numeric values in different formats. For example, you can display a big number in an exponential form.
- To convert numeric values using numeric format specifiers, you should enclose the specifier in curly braces. These curly braces must be enclosed in double quotes. This is done in the output methods of the `Console` class.
- The following is the syntax for the numeric format specifier:

Syntax

```
Console.WriteLine(“{format specifier}”, <variable name>);
```

where,

- `formatspecifier`: Is the numeric format specifier.
- `variable name`: Is the name of the integer variable.

Using Numeric Format Specifiers

- Numeric format specifiers work only with numeric data that can be suffixed with digits. The digits specify the number of zeros to be inserted, after the decimal location.

Example

- If you use a specifier such as C3, three zeros will be suffixed, after the decimal location of the given number. The following table lists some of the numeric format specifiers in C#:

Format Specifier	Name	Description
`C or c`	Currency	The number is converted to a string that represents a currency amount.
`D or d`	Decimal	The number is converted to a string of decimal digits (0-9), prefixed by a minus sign in case the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. This format is supported for fundamental types only.
`E or e`	Scientific (Exponential)	The number is converted to a string of the form '-d.ddd...E+ddd' or '-d.ddd...e+ddd', where each 'd' indicates a digit (0-9).

Custom Numeric Format Strings 1-3

- Custom numeric format strings contain more than one custom numeric format specifiers and define how data is formatted.
- A custom numeric format string is defined as any string that is not a standard numeric format string. The following table lists the custom numeric format specifiers and their description:

Format Specifier	Description
0	If the value being formatted contains a digit where '0' appears, then it is copied to the result string
#	If the value being formatted contains a digit where '#' appears, then it is copied to the result string
.	The first '.' character verifies the location of the decimal separator
,	The ',' character serves as a thousand separator specifier and a number scaling specifier
%	The '%' character in a format string multiplies a number with 100 before it is formatted
E0, E+0,E-0, e0, e+0, e-0	If any of the given strings are present in the format string and they are followed by at least one '0' character, then the number is formatted using scientific notation
\	The backslash character causes the next character in the format string to be interpreted as an escape sequence
'ABC' "ABC"	The characters that are enclosed within single or double quotes are copied to the result string
;	The ';' character separates a section into positive, negative, and zero numbers
Other	Any of the other characters are copied to the result string

Custom Numeric Format Strings 2-3

- The following code demonstrates the conversion of a numeric value using C, D, and E format specifiers:

Snippet

```
int num = 456;
Console.WriteLine(“{0:C}”, num);
Console.WriteLine(“{0:D}”, num);
Console.WriteLine(“{0:E}”, num);
```

Output

```
$456.00
456
4.560000E+002
```

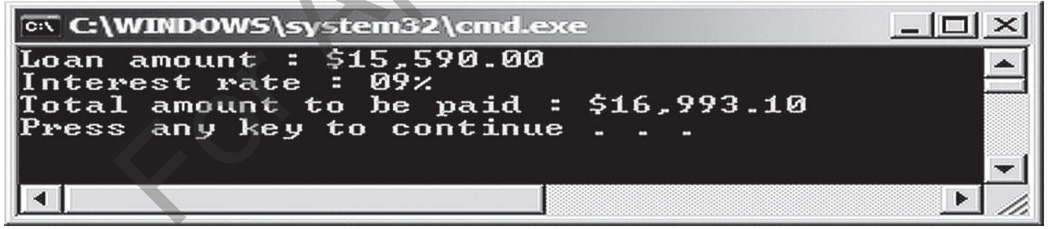
Custom Numeric Format Strings 3-3

- The following code demonstrates the use of custom numeric format specifiers:

Snippet

```
using System;
class Banking
{
static void Main(string[] args)
{
double loanAmount = 15590;
float interest = 0.09F;
double interestAmount = 0;
double totalAmount = 0;
interestAmount = loanAmount * interest ;
totalAmount = loanAmount + interestAmount;
Console.WriteLine(“Loan amount : ${0:#,###.#0} “, loanAmount);
Console.WriteLine(“Interest rate : {0:0#%} “, interest);
Console.WriteLine(“Total amount to be paid :${0:#,###.#0}”,totalAmount);
}
}
```

- In the code:
  - The #, %, ., and 0 custom numeric format specifiers are used to display the loan details of the customer in the desired format.
- The following figure displays the example of custom numeric format specifiers:



```
Loan amount : $15,590.00
Interest rate : 09%
Total amount to be paid : $16,993.10
Press any key to continue . . .
```

More Number Format Specifiers 1-2

- There are some additional number format specifiers that are described in the following table:

Format Specifier	Name	Description
F or f	Fixed-point	The number is converted to a string of the form '-ddd.ddd...' where each 'd' indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
N or n	Number	The number is converted to a string of the form '-d,ddd,ddd.ddd...', where each 'd' indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
X or x	Hexadecimal	The number is converted to a string of hexadecimal digits. Uses "X" to produce "ABCDEF", and "x" to produce "abcdef".

More Number Format Specifiers 2-2

- The following figure demonstrates the conversion of a numeric value using F, N, and X format specifiers:

Snippet

```
int num = 456;
Console.WriteLine(“{0:F}”, num);
Console.WriteLine(“{0:N}”, num);
Console.WriteLine(“{0:X}”, num);
```

Output

```
456.00
456.00
1C8
```

Standard Date and Time Format Specifiers

- A date and time format specifier is a special character that enables you to display the date and time values in different formats.

Example

- You can display a date in `mm-dd-yyyy` format and time in `hh:mm format`.
- If you are displaying GMT time as the output, you can display the GMT time along with the abbreviation GMT using date and time format specifiers.
- The date and time format specifiers allow you to display the date and time in 12-hour and 24-hour formats.
- The following is the syntax for date and time format specifiers:

Syntax

```
Console.WriteLine("{format specifier}",
<datetime object>);
```

- where,
  - `formatspecifier`: Is the date and time format specifier.
  - `datetime` object: Is the object of the `DateTime` class.

Using Standard Date and Time Format Specifiers 1-2

- Standard date and time format specifiers are used in the `Console.WriteLine()` method with the `datetime` object that can be created using an object of the `DateTimeclass` and initialize it.
- The formatted date and time are always displayed as strings in the console window. The following table displays some of the standard date and time format specifiers in C#:

Format Specifier	Name	Description
d	Short date	Displays date in short date pattern. The default format is ‘mm/dd/yyyy’.
D	Long date	Displays date in long date pattern. The default format is ‘dddd*, MMMM*, dd, yyyy’.
f	Full date/time (short time)	Displays date in long date and short time patterns, separated by a space. The default format is ‘dddd*, MMMM* dd, yyyy HH*:mm*’.
F	Full date/time (long time)	Displays date in long date and long time patterns, separated by a space. The default format is ‘dddd*, MMMM* dd, yyyy HH*: mm*: ss*’.
g	General date/time (short time)	Displays date in short date and short time patterns, separated by a space. The default format is ‘MM/dd/yyyy HH*:mm*’.

Using Standard Date and Time Format Specifiers 2-2

- The following code demonstrates the conversion of a specified date and time using the d, D, f, F, and g date and time format specifiers:

Snippet

```
DateTimedt = DateTime.Now;
// Returns short date (MM/DD/YYYY)
Console.WriteLine(“Short date format(d): {0:d}”, dt);
// Returns long date (Day, Month Date, Year)
Console.WriteLine(“Long date format (D): {0:D}”, dt);
// Returns full date with time without seconds
Console.WriteLine(“Full date with time without seconds (f):{0:f}”, dt);
// Returns full date with time with seconds
Console.WriteLine(“Full date with time with seconds (F):{0:F}”, dt);
// Returns short date and short time without seconds
Console.WriteLine(“Short date and short time without seconds (g):{0:g}”, dt);
```

Output

```
Short date format(d): 23/04/2007
Long date format (D): Monday, April 23, 2007
Full date with time without seconds (f):Monday, April 23, 2007
   12:58 PM
Full date with time with seconds (F):Monday, April 23, 2007
   12:58:43 PM
Short date and short time without seconds (g):23/04/2007 12:58 PM
```

Additional Standard Date and Time Format Specifiers 1-2

- The following table displays the standard date and time format specifiers in C#:

Format Specifier	Name	Description
G	General date/time (long time)	Displays date in short date and long time patterns, separated by a space. The default format is 'MM/dd/yyyy HH*:mm*:ss*'.
m or M	Month day	Displays only month and day of the date. The default format is 'MMMM* dd'.
T	Short time	Displays time in short time pattern. The default format is 'HH*: mm*'.
T	Long time	Displays time in long time pattern. The default format is 'HH*:mm*:ss*'.
y or Y	Year month pattern	Displays only month and year from the date. The default format is 'YYYY MMMM*'.

Additional Standard Date and Time Format Specifiers 2-2

- The following code demonstrates the conversion of a specified date and time using the G, m, t, T, and y date and time format specifiers:

Snippet

```
DateTimedt = DateTime.Now;
// Returns short date and short time with seconds
Console.WriteLine(“Short date and short time with seconds (G):{0:G}”, dt);
// Returns month and day - M can also be used
Console.WriteLine(“Month and day (m):{0:m}”, dt);
// Returns short time
Console.WriteLine(“Short time (t):{0:t}”, dt);
// Returns short time with seconds
Console.WriteLine(“Short time with seconds (T):{0:T}”, dt);
// Returns year and month - Y also can be used
Console.WriteLine(“Year and Month (y):{0:y}”, dt);
```

Output

```
Short date and short time with seconds (G):23/04/2007
   12:58:43 PM
Month and day (m):April 23
Short time (t):12:58 PM
Short time with seconds (T):12:58:43 PM
Year and Month (y):April, 2007
```

Custom DateTime Format Strings 1-2

- Any non-standard DateTime format string is referred to as a custom DateTime format string. Custom DateTime format strings consist of more than one custom DateTime format specifiers. The following table lists some of the custom DateTime format specifiers:

Format Specifier	Name
ddd	Represents the abbreviated name of the day of the week
dddd	Represents the full name of the day of the week
FF	Represents the two digits of the seconds fraction
H	Represents the hour from 0 to 23
HH	Represents the hour from 00 to 23
MM	Represents the month as a number from 01 to 12
MMM	Represents the abbreviated name of the month
s	Represents the seconds as a number from 0 to 59

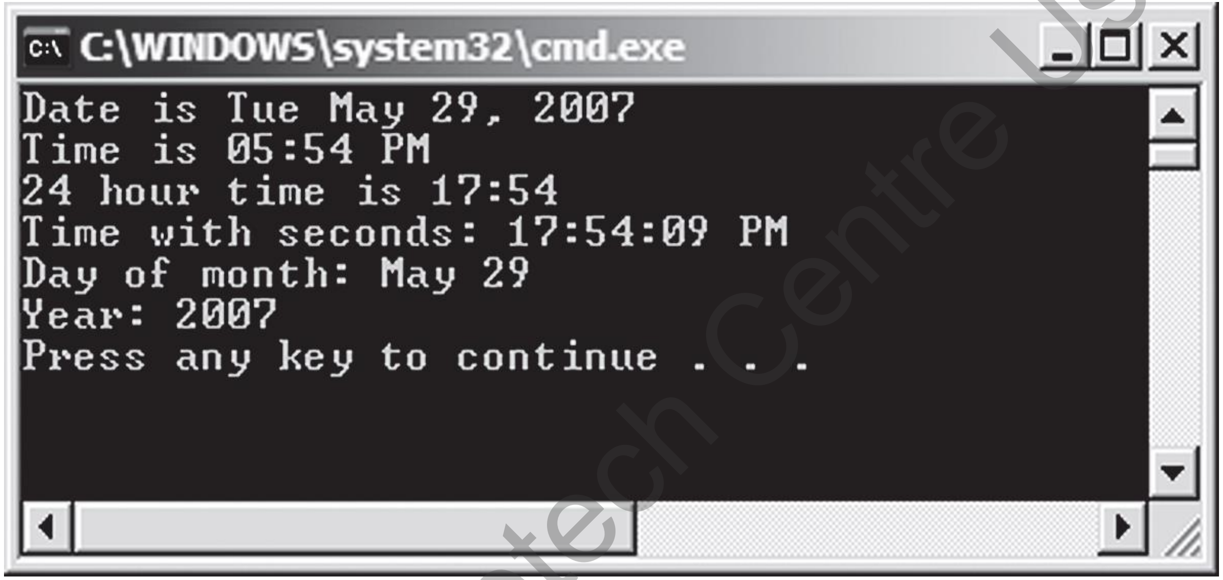
- The following code demonstrates the use of custom DateTime format specifiers:

```
using System;
class DateTimeFormat
{
   public static void Main(string[] args)
   {
        DateTime date = DateTime.Now;
        Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", date);
        Console.WriteLine("Time is {0:hh:mm tt}", date);
        Console.WriteLine("24 hour time is {0:HH:mm}", date);
        Console.WriteLine("Time with seconds: {0:HH:mm:ss tt}", date);
        Console.WriteLine("Day of month: {0:m}", date);
        Console.WriteLine("Year: {0:yyyy}", date);
   }
}
```

Custom DateTime Format Strings 2-2

- In the code:
  - The date and time is displayed using the different DateTime format specifiers.

- The following figure displays the output of the code:



```
Date is Tue May 29, 2007
Time is 05:54 PM
24 hour time is 17:54
Time with seconds: 17:54:09 PM
Day of month: May 29
Year: 2007
Press any key to continue . . .
```

Summary

- A variable is a named location in the computer's memory and stores values.
- Comments are used to provide detailed explanation about the various lines in a code.
- Constants are static values that you cannot change throughout the program execution.
- Keywords are special words pre-defined in C# and they cannot be used as variable names, method names, or class names.
- Escape sequences are special characters prefixed by a backslash that allow you to display non-printing characters.
- Console operations are tasks performed on the command line interface using executable commands.
- Format specifiers allow you to display customized output in the console window.