OOP with C#

C# - Classes

- **Class Definition:** A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces.

- Example?

Member Variables

- Variables are attributes or data members of a class, used for storing data.
- In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

- Functions are set of statements that perform a specific task.
- The member functions of a class are declared within the class.
- Our sample class Rectangle contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

Constructors in C#

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

- A constructor will have **exact same name as the class** and it does not have any **return type**.

- A **default constructor** does not have any parameter but if you need a constructor can have parameters.

- Such constructors are called **parameterized constructors**.

- This technique helps you to assign initial value to an object at the time of its creation.

- Example?

Destructors in C#

- A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope.

- A **destructor** will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

- Destructors cannot be inherited or overloaded.

- Example?

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // member variables
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

Implementation of a Rectangle class and discuss C# basic syntax

C# - Structures

- 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.

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

Class vs Structure

- Classes and Structures have the following basic differences:

✓Classes are reference types and structs are value types

✓Structures do not support inheritance

✓Structures cannot have default constructor

C# - Enums

- An enumeration is a set of named integer constants.

- An enumerated type is declared using the **enum** keyword.

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

- Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it.

- By default, the value of the first enumeration symbol is 0.

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

C# - Encapsulation

- Encapsulation, in the context of C#, refers to an object's ability to hide data and behavior that are not necessary to its user.

- Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member.
- C# supports the following access specifiers:
  - ✓Public
  - ✓Private
  - ✓Protected
  - ✓Internal
  - ✓Protected internal

Public Access Specifier

- Public access specifier allows a class to expose its member variables and member functions to other functions and objects.

- Any public member can be accessed from outside the class.

- Example?

Private Access Specifier

- Private access specifier allows a class to hide its member variables and member functions from other functions and objects.
- Only functions of the same class can access its private members.
- Even an instance of a class cannot access its private members.
- Example?

Protected Access Specifier

- Protected access specifier allows a child class to access the member variables and member functions of its base class.

- This way it helps in implementing inheritance.

- Example?

C# Inheritance

- **Inheritance** makes programs simpler and faster.
- With inheritance, build several types upon a common abstraction.
- **Base class:** The class another class inherits from. In the C# language, the ultimate base class is the object class.
- **Derived class:** The class with a base class. A derived class may have multiple levels of base classes.
- Example?

Multiple Inheritance in C#

- **C# does not support multiple inheritance**.
- However, can be used **interfaces** to implement multiple inheritance.
- Example?

C# - Interfaces

- An interface is defined as a syntactical contract that all the classes inheriting the interface should follow.
- Interfaces define properties, methods and events, which are the members of the interface.
- Interfaces contain only the declaration of the members.
- It is the responsibility of the deriving class to define the members.
- Example?

```
public interface ITransactions
{
    // interface members
    void showTransaction();
    double getAmount();
}
```

Abstract Classes in C#

A class that is declared by using the keyword abstract is called an abstract class. An abstract class is a partially implemented class used for developing some of the operations which are common for all next level subclasses. So it contains both abstract methods, concrete methods including variables, properties, and indexers.

It is always created as a superclass next to the interface in the object inheritance hierarchy for implementing common operations from the interface.

An abstract class may or may not have abstract methods. But if a class contains an abstract method then it must be declared as abstract. The abstract class cannot be instantiated directly. It's compulsory to create/derive a new class from an abstract class in order to provide the functionality to its abstract functions.

What is the abstract method?

A method that does not have the body is called an abstract method.

It is declared with the modifier abstract. It contains only the Declaration/signature and does not contain the implementation/ body of the method.

An abstract function should be terminated with a semicolon. Overriding of an abstract function is compulsory.

When to use the abstract method?

Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways.

What is a Sealed Class?

A sealed class is a class that cannot be inherited from.

That means if we have a class called Customer that is marked as sealed.

No other class can inherit from the Customer class.

ABSTRACT CLASS	SEALED CLASS
A class that contains one or more abstract methods is known as an abstract class.	A class from which it is not possible to derive a new class is known as a sealed class.
The abstract class can contain abstract and non-abstract methods.	A sealed class can contain non-abstract methods; it cannot contain abstract and virtual methods.
Creating a new class from an abstract class is compulsory to consume it.	It is not possible to create a new class from a sealed class.
An abstract class cannot be instantiated directly; we need to create the object for its child classes to consume an abstract class.	We should create an object for the sealed class to consume its members.
We need to use the keyword abstract to make any class as abstract.	We need to use the keyword sealed to make any class as sealed.
An abstract class cannot be the bottom-most class within the inheritance hierarchy.	The sealed class should be the bottommost class within the inheritance hierarchy.

Difference between abstract class and interface in C#?

- An Abstract class doesn't provide full abstraction but an interface does provide full abstraction

- Using Abstract we can not achieve multiple inheritance but using an Interface we can achieve multiple inheritance.

- We can not declare a member field in an Interface.

- We can not use any access modifier i.e. public , private , protected , internal etc. because within an interface by default everything is public.

- An Interface member cannot be defined using the keyword static, virtual, abstract or sealed

Identifiers

- An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:
- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \
- However an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Reserved 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	virtual	void
volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

Keywords are reserved words predefined to the C# compiler.

These keywords cannot be used as identifiers

If you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

C# - Data Types

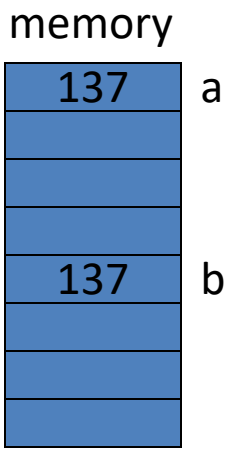
In C#, variables are categorized into the following types:

- Value types
- Reference types
- Pointer types

Value Types

- Value type variables can be assigned a value directly.
- Contain the actual value, not the location
- Inherited from **System.ValueType**
- Treated specially by the runtime

```
{
    int a = 137;
    int b = a;
}
```



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	-923,372,036,854,775,808 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

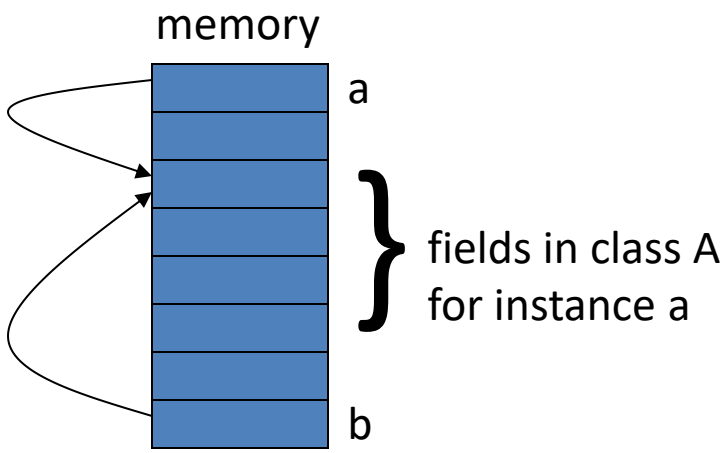
Reference Types

- 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 more than one variable, the reference types can refer to a memory location
- Example of **built in** reference types are: **object**, **dynamic** and **string**.

Reference Types

- Classes, including user-defined classes
  - Inherited from System.Object
  - Transparently refers to a memory location
  - Similar to pointers in other languages
  - Can be set to null

```
{
    A a = new A();
    A b = a;
}
```



Reference Types

- **Object Type**
  - The **Object Type** is the ultimate base class for all data types in C# Common Type System(CTS)
  - 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
```

Reference Types

- **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 runtime.

Syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

Reference Types

- **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.

For example,

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

Pointer Types

- Pointer type variables store the memory address of another type.
- Pointers in C# have the same capabilities as 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 basically type casting, or converting one type of data to another type.

- **Implicit type conversion** - these conversions are performed by C# in a type-safe manner.
- Examples:
  - conversions from smaller to larger integral types;
  - conversions from derived classes to base classes.

- **Explicit type conversion** - these conversions are done explicitly by users using the pre-defined functions.

C# - Type Conversion

Eg: Explicit Conversion

```
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();

        }
    }
}
```

C# - Type Conversion

- **ToChar:** Converts a type to a single Unicode character, where possible.
- **ToDateTime:** Converts a type (integer or string type) to date-time structures.
- **ToDecimal :** Converts a floating point or integer type to a decimal type.
- **ToDouble:** Converts a type to a double type.
- **ToInt16:** Converts a type to a 16-bit integer.
- **ToInt64:** Converts a type to a 64-bit integer.
- **ToSingle:** Converts a type to a small floating point number.
- **ToString:** Converts a type to a string.

C# - Variables

- A variable is nothing but a name given to a storage area that our programs can manipulate.

- 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

Variable Declaration in C#

- Syntax for variable declaration in C# is:

```
<data_type> <variable_list>;
```

- Some valid variable declarations along with their definition

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

- Variables are initialized (assigned a value) with an equal sign followed by a constant expression

```
int d = 3, f = 5;
byte z = 22;
double pi = 3.14159;
char x = 'x';
```

Nullable data types

- A nullable type can represent the correct range of values for its underlying value type, plus an additional null value.

- Nullable of Int32, can be assigned any value from -**2147483648** to **2147483647**, or it can be assigned the **null** value.
- A Nullable <bool> can be assigned the values **true, false**, or **null**.

```
int? zz = null;

//zz = 10;

if (zz == null)
{
    Console.WriteLine("zz=null");
}
else
{
    Console.WriteLine("zz={0}",zz);
}
```

Defining Constants

- Constants are defined using the **const** keyword.

```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // constant declaration
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

Escape Sequence Codes

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Defining Methods in C#

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

- Example?

```
class NumberManipulator
{
   public int FindMax(int num1, int num2)
   {
      /* local variable declaration */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

C# - Decision Making

Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one **swicth** statement inside another **switch** statement(s).

C# - nested switch Statements

```
switch(ch1)
{
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2)
      {
         case 'A':
            printf("This A is part of inner switch" );
            break;
         case 'B': /* inner B case code */
      }
      break;
   case 'B': /* outer B case code */
}
```

C# - Loops

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

C# - Arrays

- An array stores a fixed-size sequential collection of elements of the same type.

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

C# - Arrays

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n is an array of 10 integers */
         int i,j;


         /* initialize elements of array n */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }

         /* output each array element's value */
         for (j = 0; j < 10; j++ )
         {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

C# - Multidimensional Arrays

- The simplest form of the multidimensional array is the **two-dimensional array**.

- A two-dimensional array is, in essence, a list of one-dimensional arrays.

- A two dimensional array can be thought of as a table which will have x number of rows and y number of columns.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

C# - Jagged Arrays

- A Jagged array is an array of arrays.
- Can store efficiently many rows of varying lengths.
- Any type of data, reference or value, can be used.
- Example?

Two-Dimensional Array

1	2	3
4	5	6
7	8	9

Jagged Array

1	2				
3	4	5	6	7	8
9	10	11			

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.

- C# provides two techniques to implement static polymorphism.

  - ✓Function overloading

  - ✓Operator overloading

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.

- Example?

C# - Operator Overloading

❑Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined.

❑ Like any other function, an overloaded operator has a return type and a parameter list.

❑Example?

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

Overloadable and Non-Overloadable Operators

Operators	Description
+, -, !, ~, ++, --	These unary operators take one operand and can be overloaded.
+, -, *, /, %	These binary operators take one operand and can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators cannot be overloaded directly.
+=, -=, *=, /=, %=	The assignment operators cannot be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded.

Dynamic Polymorphism

- Run time polymorphism is also known as method overriding.

- In this mechanism by which a call to an overridden function is resolved at a Run-Time (not at Compile-time) if a base Class contains a method that is overridden.

- Method overriding means having two or more methods with the same name, same signature but with different implementation.

- In this process, an overridden method is called through the reference variable of a super class, the determination of the method to be called is based on the object being referred to by reference variable.

Dynamic Polymorphism

- C# allows you implement Dynamic Polymorphism by creating 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.

- Remember…
  - ✓You cannot create an instance of an abstract class
  - ✓You cannot declare an abstract method outside an abstract class

- Example?

C# - Exception Handling

- An exception is a problem that arises during the execution of a program.

- Exceptions provide a way to transfer control from one part of a program to another.

- C# exception handling is built upon four keywords: **try**, **catch**, **finally** and **throw**.

C# - Exception Handling

✓ **try**: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

✓**catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

✓**finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

✓**throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Classes in C#

- The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class.

- Some of the exception classes derived from the System.Exception class are;
  - **System.ApplicationException**
  - **System.SystemException**
  - **DivideByZeroException**
  - **DirectoryNotFoundException**
  - **FileNotFoundException**

Exception Classes in C#

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

C# - File I/O

- A **file** is a collection of data stored in a disk with a specific name and a directory path.

- When a file is opened for reading or writing, it becomes a **stream**.

- There are two main streams: the **input stream** and the **output stream**.

- The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

I/O Classes

- The **System.IO** namespace has various class that are used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.

- Example?

I/O Classes

I/O Class	Description
BinaryReader	Reads primitive data from a binary stream.
BinaryWriter	Writes primitive data in binary format.
BufferedStream	A temporary storage for a stream of bytes.
Directory	Helps in manipulating a directory structure.
DirectoryInfo	Used for performing operations on directories.
DriveInfo	Provides information for the drives.
File	Helps in manipulating files.
FileInfo	Used for performing operations on files.
FileStream	Used to read from and write to any location in a file.
MemoryStream	Used for random access to streamed data stored in memory.
Path	Performs operations on path information.
StreamReader	Used for reading characters from a byte stream.
StreamWriter	Is used for writing characters to a stream.
StringReader	Is used for reading from a string buffer.
StringWriter	Is used for writing into a string buffer.