Object Oriented Programming and Concepts I Methodology	420-JV4-AS Section 07242 Winter 2023	Instructor: Mike Poullas

Week 1 - Theory

Fast, Good or Cheap

Pick any two, you can't have it all.

- When writing code professionally, there will always be priorities.
- You will need to deliver based on hard choices that will affect the outcome of any project.

Structured versus Object Oriented Code

Top-Bottom versus Bottom-Up

- Structured programming follows a procedural method of building software using a linear approach with execution following the writing order of the code going from top to bottom. This

method is used to allow for easy readability of code with no abstractions. The emphasis is on Code blocks, sequences, conditional, control structures, iteration, and nested structures. It was created to avoid “spaghetti code” scenarios which make code unreliable and difficult to manage.

- Object Oriented programming is a paradigm used in software development that relies on the concept of classes and objects. Its primary intent is to allow developers to group attributes and behaviors of various entities as an abstraction to simplify and better manage complex software development. OOP permits code reuse and extensibility using Inheritance, Abstraction, Encapsulation and Polymorphism as its mechanism. Software development using OOP allows for a bottom-up approach to development by allowing for smaller programs at the start to be more easily extendable as complexity grows.
- https://www.geeksforgeeks.org/difference-between-bottom-up-model-and-top-down-model/

C# is derived from C/C++

C# is a strictly typed language like C/C++

- C# is an Objected Oriented language that use a Run-time (CLR) to execute code.
- C# is a static language and provides default signed types (short, int, double, char, string, bool) as well as UNISGNED types (ushort, uint, ulong).
- We use type declarations to assign variables to values
- We can declare variables as CONST to make them immutable i.e. Pi
- This course will only cover Windows Console apps using the .NET 4.5 Framework.

Week 1 – Lab

Visual Studio 2019

- We will only be using Visual Studio 2019 Community. Download and install it along with the .NET Desktop Development option.

OneDrive and Folder

- ALWAYS back up your source files to your school OneDrive cloud account.
- Create a Parent folder labeled “**OOP_420_JA4-AS_07242**” and start adding folders inside it labeled “Week_n” where n will be the course week number.
- Save your code to the corresponding weekly folder for each lab.
- Also add a “Final Project” folder.

W3School

- We will be using the C# tutorial from W3School as a review of C#.

- https://www.w3schools.com/cs/index.php
- You should also have as a reference the Microsoft Docs site for C#
- https://learn.microsoft.com/en-us/dotnet/csharp/

Comments

- Always use comments at the start of a **Class** or **Code Block** (if/while/for) to describe the behavior of your code. Indicate revisions as well. Rarely use comments for a single line of code unless it's a Lambda function (to be discussed)
- Typing a triple slash /// above a Class or Code Block will auto generate a comment template for you to fill. Use it.
- ALL your source files should include your full name and student ID at the top and the assignment number when you're handing in code.

Naming Conventions

- Various naming conventions exist for labeling variables, functions, classes, etc. such as **PascalCase**, **camelCase**, and **snake_case**.
- For this course, variables will always be in **snake_case**, Methods and Classes and Structs will always be in PascalCase.
- Use very descriptive names but keep it under 20 characters.

The general rules for naming variables are:

- Names can contain letters, digits and the underscore character (_)
- Names must begin with a letter (a-z).
- Names should start with a lowercase letter, and it cannot contain whitespace.
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like C# keywords, such as int or double) cannot be used as names.

(Source: W3School.com)

Week 2 – Theory

C# keywords and types

- Review of most of the C# keywords as presented on W3School.com.
- Explored Types and the size in Bytes used by the most common types such as:

  short = 2 bytes = -32768 to +32768

int = 4 Bytes = -2,147,482,648 to +2,147,482,648

a '**u**' in front of a numeric type makes it **UNSIGNED** so it cannot have a negative value.

i.e., **uint** = 0 to +4,294,965,296

- A good resource for C# types is here:

  https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types

C# Output Stream Formatting

- We can display values from our variables using placeholders in the WriteLine() method call.
- A placeholder **without** an index will place the variable value in the order it is placed in the method parameter list: Console WriteLine ("This is the result of {} and {}", a_var, b_var); Will first display a_var then b_var.
- A placeholder **with** an index will place the variable value in the order it is placed in the method parameter list: Console WriteLine ("This is the result of {0} and {1}", a_var, b_var); Will first display a_var then b_var but by switching the index values. ConsoleWriteLine ("This is the result of {1} and {0}", a_var, b_var); Will display b_var then a_var.
- You can also use the names of the variables as placeholders: Console WriteLine ("This is the result of {a_var} and {b_var}", a_var, b_var);
- Place holder can also use format specifiers and precision specifiers to convert a string (or other types) to a specific format and precision (if required). For example:

```
decimal value = 123.456m;
Console.WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

  You can find more details here:
  https://learn.microsoft.com/en-us/dotnet/standard/base-types/formatting-types

- Escape codes can be used within the WriteLine() method to print control codes such as Carriage-Return and Linefeed or tab.
- Escape codes start with a back slash:
  Console.WriteLine("The main heading is: C# **\n \t** And the subheading is: Objects");
  This will generate the following output:

```
The main heading is:  C#
        And the subheading is: Objects
```

- You can find more details here:
  https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-escapes-in-regular-expressions

- Using the **String.Format** method allows for additional string formatted outputs as follows:

```
string s = String.Format("It is now {0:d} at {0:t}", DateTime.Now);
Console.WriteLine(s);
// Output similar to: 'It is now 4/10/2015 at 10:04 AM'
```

You can find more details here:
https://learn.microsoft.com/en-us/dotnet/api/system.string.format?view=net-7.0

Week 2 – Lab

Capturing Command Line Arguments

- A console application built using Visual Studio typically resides in the following folder path:

```
C:\Users\{%userprofile%}\source\repos\{projectnamefolder}\{projectname}\bin\debug
```

- Using the **CD** command will allow you to change the current directory on the condole window to
- **DIR** will allow you to list the contents of a directory.
- You run your console application by using the CMD console and then changing to the ..\debug directory (see above) and execute the .EXE file created by Visual Studio simple by typing the %program% name.
- Every console application has a single **namespace** (project name), **class** (program name) and **Main** method as the entry point of the application:

```
static void Main(string[] args)
```

- The args parameter has a special purpose. It captures any string(s) added to the end of console application when running it.
- Console applications can capture arguments from the command line such as:

```
Program.exe always be coding
```

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");
for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = always
```

```
        Arg[1] = be
        Arg[2] = coding
    */
```

- It's important to remember that while args.Length returns the actual number of arguments (3 in the example), the index always starts with zero (0). Your loop should range from 0 to 2!

Week 3 – Theory

Method overloading

- Method overloading allows developers to use the same name to define a function or method while using different parameter input types:

```
int MyMethod(int x)
float MyMethod(float x)
double MyMethod(double x, double y)
```

- Overloading offers several advantages to developers but the main one is flexibility. Allowing methods to handle different types increases code readability without having to rename similar functions in code. Overloading is important in OOP as we will see later in the course when we discuss constructors.

Classes and Objects

- (Most) everything in C# is an Object. Objects are extendable and have multiple actions (methods)
- Classes are used in Object Oriented Programming to define both data (properties/attributes) and operations (operations/methods) as one contained entity with common characteristics and behaviors.
- A class defines a template or blueprint for creating objects that are functional in a program.
- Classes do not occupy memory. They are only definitions and used as an **object constructor**.
- Objects constructed from a class do occupy memory in an application and are thus an **instance** of a class.
- C# uses the class keyword to define a class as follows:

```
class Car {
   string color = "red";
   static void Main(string[] args)
   {
      Car myObj = new Car();
      Console.WriteLine(myObj.color);
   }
```

```
}
```

- In the example above, the Car class defines the **color** variable as a string and the **Main**() function as the method within the class.
- The color variable is a field (or data) member in the class and the Main() function is the member method.
- An object, **myObj** is derived or **instantiated** from the class by using the new keyword in the line:

```
Car myObj = new Car();
```

`myObj` is defined as a Car type (class) and new allocates the space in memory to be used by the application of size Car. We will go into more detail about the sizes of classes later in the course.

-
- Every object instantiated has its own unique space in memory. One cannot override another.
- One important note is that the scope (access of data and methods) of everything inside an object is local. Methods within a class must be defined using the public access modifier.

```
class Car
  {
        string color;                // field
        int maxSpeed;                // field
        public void fullThrottle()   // method
        {
            Console.WriteLine("The car is going as fast as it can!");
        }
        static void Main(string[] args)
        {
            Car myObj = new Car();
            myObj.fullThrottle();  // Call the method
        }
  }
```

- a `static` method can be accessed without creating an object of the class, while `public` methods can only be accessed by objects.

- A class's attributes or operations/methods can be accessed by another class if the field or method has been made public. In addition, they must belong to the same namespace.

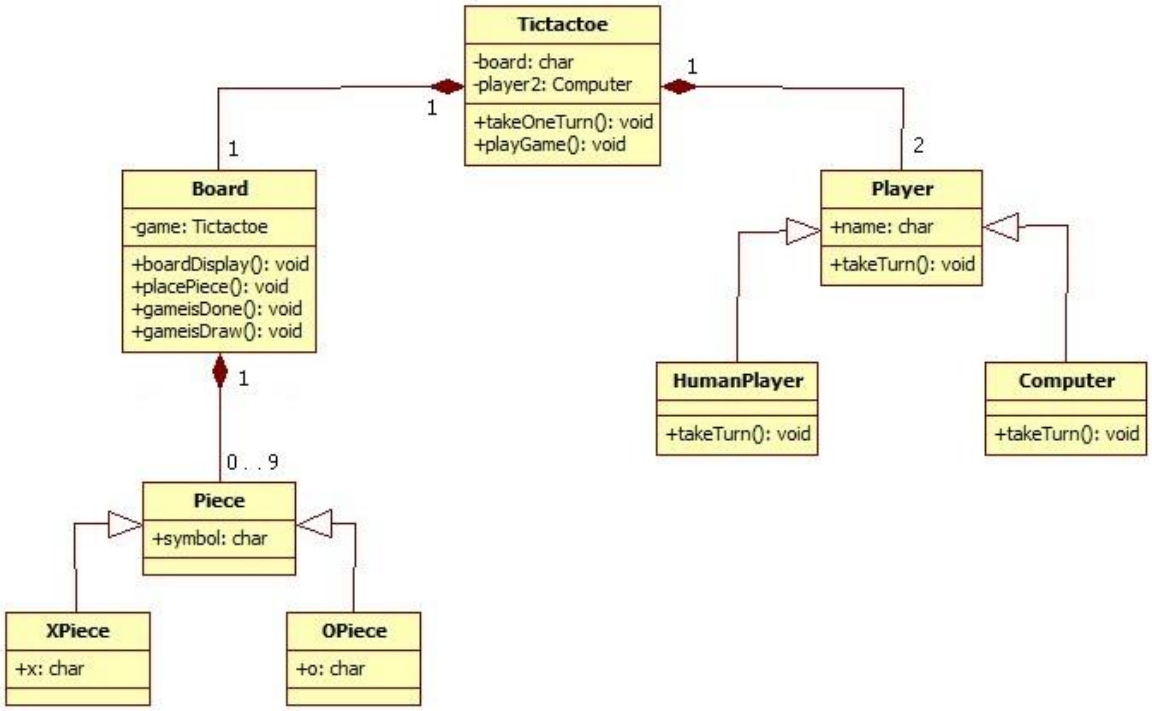
  See the following code on W3School:
  C# Multiple Classes and Objects (w3schools.com)

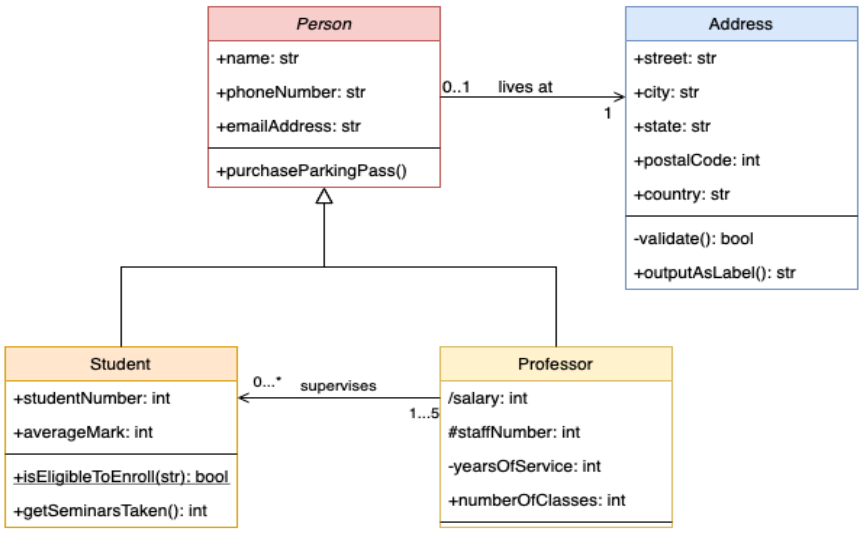
UML

Unified Modeling Language

- UML is a visual description and definition consisting of an integrated set of diagrams.
- UML uses mostly graphical notations to express the design of software projects.
- Using UML helps project teams communicate, explore potential designs, and validate the architectural design of the software,
- UML is a very important part of developing object-oriented software and the software development process by allowing developers to "see" code before its constructed.
- UML provides a very rich, complex and sophisticated set of tools to help design and architect software however for the purpose of this course, we will use a very basic and generalized approach to diagramming with UML to assist in building C# classes or what is referred to as a Class Diagram.

  Here is an example of a UML defining classes to be used for a tic-tac-toe game:





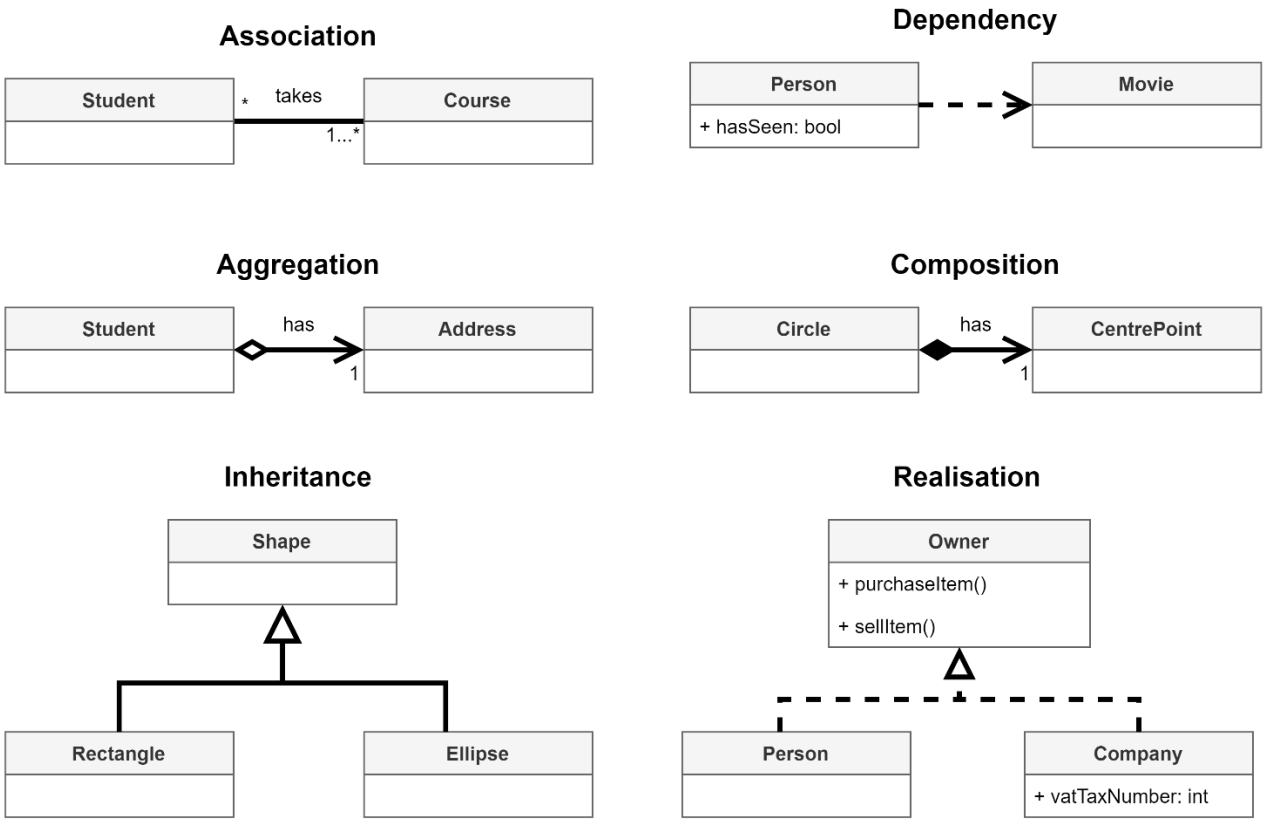
- The UML block consists of three sections:
    1. The name of the class at the top third.
    2. The fields used in the class in the middle third.
    3. The methods used in the bottom third.
- Dark arrows pointed from one class to another represent an **IsA** relationship. Technically it is referred to as a **Generalization** or **Inheritance**.
- White arrows pointed from one class to another represent a **HasA** relationship. Technically it is referred to as a **Basic Relationship** or **Association**
- The plus (+) symbol next to a field or method name represents a public access modifier.
- The minus (-) symbol next to a field or method name represents a private access modifier.

Java	UML

`public class Employee {`
`  private int empID;`
 	
`  public double calcSalary() {`
`  …`
`  }`
`}`
	Employee
	-empID:int
	+calcSalary():double

- It's important to understand how many items on each side of a relationship can exist, which we call **cardinality**. The cardinality is expressed in terms of: "**one to one**", "**one to many**" and "**many to many**". In UML it's notated as follows:

Notation	Meaning
0	No instances
0..1	No instances, or one instance
1	Exactly one instance
*	Zero to many instances
1..*	One or more instances



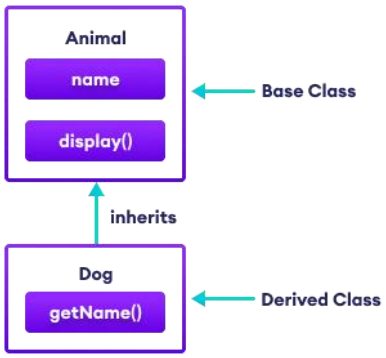
https://www.diagrams.net/blog/uml-class-diagrams

https://www.w3schools.com/cs/cs_class_members.php

– Part 1https://www.w3schools.com/cs/cs_constructors.phphttps://www.w3schools.com/cs/cs_enums.php

- Part 1 https://www.w3schools.com/cs/cs_inheritance.php

Part 2



- In C#, inheritance is an **Is-A** relationship. We use inheritance only if there is an I**s-A** relationship between two classes. For example,

  Dog is an Animal

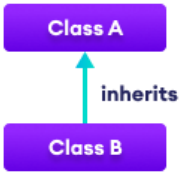
  Apple is a Fruit

  Car is a Vehicle

- There are a variety of inheritances in OOP.

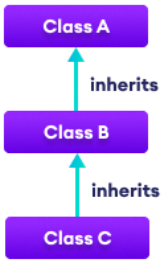
    1. Single Inheritance

        In single inheritance, a single derived class inherits from a single base class:



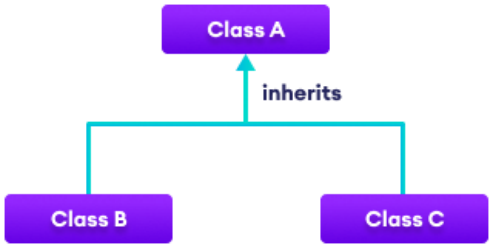
    2. Multiple Inheritance

        In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class:



    3. Hierarchical Inheritance

        In hierarchical inheritance, multiple derived classes inherit from a single base class.



- Importance of Inheritance. Example:

```
using System;

namespace Inheritance {

  class RegularPolygon {
      public void CalculatePerimeter(int length, int sides) {
        int result = length * sides;
        Console.WriteLine("Perimeter: " + result);
      }
  }

  class Square : RegularPolygon {
    public int  length = 200;
    public int sides = 4;
    public void CalculateArea() {
      int area = length * length;
      Console.WriteLine("Area of Square: " + area);
    }
  }

 class Rectangle : RegularPolygon {
    public int length = 100;
    public int breadth = 200;
    public int sides = 4;

    public void CalculateArea() {
      int area = length * breadth;
      Console.WriteLine("Area of Rectangle: " + area);
    }
  }

  class Program {
    static void Main(string[] args) {
      Square s1 = new Square();
      s1.CalculateArea();
      s1.CalculatePerimeter(s1.length, s1.sides);
      Rectangle t1 = new Rectangle();
      t1.CalculateArea();
      t1.CalculatePerimeter(t1.length, t1.sides);
    }
  }
}
```

Output

```
Area of Square: 40000
Perimeter: 800
Area of Rectangle: 20000
Perimeter: 400
```

- In the above example, we have created a **RegularPolygon** class that has a method to calculate the perimeter of the regular polygon.
- Here, the **Square** and **Rectangle** inherit from **RegularPolygon**.
- The formula to calculate the perimeter is common for all, so we have reused the **CalculatePerimeter**() method of the base class.
- And since the formula to calculate the area is different for different shapes, we have created a separate method inside the derived class to calculate the area.

C# Encapsulation - Access Modifiers

- In C#, access modifiers specify the accessibility of types (classes, interfaces, etc.) and type members (fields, methods, etc.). For example:

```
class Student {

  public string name;

  private int num;

  protected int student_id;

  }
```

name - **public** field that can be accessed from anywhere.
num - **private** field can only be accessed within the **Student** class.
student_id - **protected** field can only be accessed within the **Student** class **or** by a derived class.

Types of Access Modifiers

- In C#, there are 4 basic types of access modifiers:
    1. public
    2. private
    3. protected
    4. internal

1. Public access modifier

- When we declare a type or type member **public**, it can be accessed from anywhere. For example:

```
using System;

namespace MyApplication {

  class Student {
    public string name = "Smith";

    public void Print() {
      Console.WriteLine("Hello from Student class!");
     }
  }

  class Program {
    static void Main(string[] args) {

      // creating object of Student class
      Student student1 = new Student();

      // accessing name field and printing it
      Console.WriteLine("Name: " + student1.name);

      // accessing print method from Student
      student1.Print();
      Console.ReadLine();
    }
  }
}
```

Output:

```
- Name: Smith
- Hello from Student class!
```

- In the above example, we have created a class named **Student** with a field **name** and a method **print().**

```
// accessing name field and printing it
Console.WriteLine("Name: " + student1.name);

// accessing print method from Student
student1.Print();
```

- Since the field and method are public, we can access them from the **Program** class.

Note: We have used the object **student1** of the **Student** class to access its members

2. private access modifier

- When we declare a type member with the **private** access modifier, it can only be accessed within the same **class** or **struct**. For example:

```
using System;

namespace MyApplication {

  class Student {
    private string name = "Smith";
    private void Print() {
      Console.WriteLine("Hello from Student class");
     }
  }

  class Program {
    static void Main(string[] args) {
      // creating object of Student class
      Student student1 = new Student();

      // accessing name field and printing it
      Console.WriteLine("Name: " + student1.name);

      // accessing print method from Student
      student1.print();
      Console.ReadLine();
    }
  }
}
```

- In the above example, we have created a class named **Student** with a field **name** and a method **print().**

```
// accessing name field and printing it
Console.WriteLine("Name: " + student1.name);

// accessing print method from Student
student1.Print();
```

- Since the field and method are private, we are not able to access them from the `Program` class. Here, the code will generate the following error.

```
• Error    CS0122    'Student.name' is inaccessible due to its protection level
• Error    CS0122    'Student.Print()' is inaccessible due to its protection level
```

3. protected access modifier

- When we declare a type member as **protected**, it can only be accessed from the same class and its derived classes. For example:

```
using System;

namespace MyApplication {

  class Student {
    protected string name = "Smith";
  }

  class Program {
    static void Main(string[] args) {

      // creating object of student class
      Student student1 = new Student();

      // accessing name field and printing it
      Console.WriteLine("Name: " + student1.name);
      Console.ReadLine();
    }
  }
}
```

- In the above example, we have created a class named **Student** with a field **name**. Since the field is protected, we are not able to access it from the **Program** class.
- Here, the code will generate the following error:

```
Error    CS0122    'Student.name' is inaccessible due to its protection level
```

- Access the **protected** member from a derived class:

```
using System;

namespace MyApplication {

  class Student {
    protected string name = "Smith";
  }

  // derived class
  class Program : Student {
    static void Main(string[] args) {

      // creating object of derived class
      Program program = new Program();

      // accessing name field and printing it
      Console.WriteLine("Name: " + program.name);
      Console.ReadLine();
    }
  }
}
```

Output:

```
Name: Smith
```

- In the above example, we have created a class **Student** with a protected field **name**. Notice that we have inherited the **Program** class from the **Student** class.

```
• // accessing name field and printing it
• Console.WriteLine("Name: " + program.name);
```

- Since the **protected** member can be accessed from derived classes, we are able to access **name** from the **Program** class.

Method Overriding in C# Inheritance

- If the same method is present in both the base class and the derived class, the method in the derived class **overrides** the method in the base class. This is called **method overriding** in C#:

```
using System;

namespace Inheritance {

  // base class
  class Animal {
    public virtual void Eat() {
      Console.WriteLine("I eat food");
    }
  }

  // derived class of Animal
  class Dog : Animal {

    // overriding method from Animal
    public override void Eat() {
      Console.WriteLine("I eat Dog food");
    }
  }

  class Program {
    static void Main(string[] args) {
      // object of derived class
      Dog labrador = new Dog();

      // accesses overridden method
      labrador.Eat();
    }
  }
}
```

Output:

```
I eat Dog food
```

- In the above example, the **eat()** method is present in both the **base** class and **derived** class.
- When we call **eat()** using the **Dog** object labrador: **labrador.Eat()**

- the method inside **Dog** is called. This is because the method inside **Dog** **overrides** the same method inside **Animal**.
- Notice, we have used **virtual** and **override** with methods of the **base** class and **derived** class respectively:

virtual - allows the method to be overridden by the derived class.

override - indicates the method is **overriding** the method from the base class.

base Keyword in C# Inheritance

- In the previous example, we saw that the method in the derived class overrides the method in the base class.
- However, what if we want to call the method of the **base** class as well?
- In that case, we use the **base** keyword to call the method of the base class from the derived class.

Example:

```
using System;

namespace Inheritance {

  // base class
  class Animal {
    public virtual void Eat() {
      Console.WriteLine("Animals eat food.");
    }
  }

  // derived class of Animal
  class Dog : Animal {
    // overriding method from Animal
    public override void Eat() {
      // call method from Animal class
      base.Eat();
      Console.WriteLine("Dogs eat Dog food.");
    }
  }
  class Program {
    static void Main(string[] args) {
      Dog labrador = new Dog();
      labrador.Eat();
    }
  }
}
```

Output:

```
Animals eat food.
Dogs eat Dog food.
```

- In the above example, the **eat()** method is present in both the **base** class Animal and the **derived** class Dog. Notice the statement, **base.Eat();**
- We have used the **base** keyword to access the method of **Animal** class from the **Dog** class.

C# Encapsulation – Properties (get/set)

- In the previous section, we learned about controlling access to a class's fields or operations using public, private or protected.
- In particular, **private** provides a mechanism to keep class properties (data) secure and prevent direct access to any object's data.
- A property in C# is a class member that exposes the class's **private** fields.
- Internally, C# properties are special methods called **accessors** and **mutators**.
- One C# property is an **accessor**, a **get** property (or a **getter)** and is configured for **READ-ONLY** operation.
- Another C# property is a **mutator**, a **set** property (or a **setter**) and is configured for **WRITE-ONLY** operations.
- A **get** accessor **returns** a property value, and a **set** mutator **assigns** a new value.
- The value keyword represents the value of a property**.**

Example:

```
Using System;

class MyClass {
    private int number;
    public int Number {
       get { return number; }
       set { number = value; }
   }
}
class Program {
   static void Main(string[] args) {
      MyClass mc = new MyClass();
      mc.Number = 10;
      Console.WriteLine(mc.Number);
    }
  }
}
```

Output:

```
10
```

- The **Number** property is associated with the **number** field.
- It is **required** that you use **the same name** for both the property and the private field, but with an **uppercase** first letter.
- The get method **returns** the value of the variable **number**.
- The set method **assigns** a value to the **number** variable.
- The value keyword represents the value we **assign** to the property.
- Additional code could be added within the scope of the get or set to perform other functions.

Note: It I important to understand that **only** the **accessor** and **mutator** can have direct access to any class data. Any class or method outside the original class must use a property to access or alter data.

- Any given property can have both a **getter** and a **setter** or just a **getter** or just a **setter**.

Property Shorthand

- In many cases, we do not need to make the getter or setter perform any other function other than to read or write class data for the property.
- We can apply a shorthand inside a property without having to use the value or return keyword.

```
class MyClass {
    private int number;
    public int Number {
       { get; set; }
   }
}
```

Why Encapsulation?

- Better control of class members and reduce errors.
- Fields can be made read-only (if you only use the **get** method), or write-only (if you only use the **set** method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Constructors – Part 2

Types of Constructors

- There are the following types of constructors:
  - Default Constructor
  - Parameterless Constructor
  - Parameterized Constructor

1. Default Constructor

- If we have not defined a constructor in our class, then the C# will automatically create a default constructor with an empty code and no parameters. For example,

```
using System;

namespace Constructor {

  class Program {

    int a;
    static void Main(string[] args) {
      // call default constructor
      Program p1 = new Program();
      Console.WriteLine("Default value of a: " + p1.a);
      Console.ReadLine();
    }
  }
}
```

Output

```
Default value of a: 0
```

- In the above example, we have not created any constructor in the Program class.
- However, while creating an object, we are calling the constructor.

  Program p1 = new Program();

- Here, C# automatically creates a default constructor. The default constructor initializes any uninitialized variable with the default value.
- Hence, we get 0 as the value of the **int** variable **a**.

Note: In the default constructor, all the numeric fields are initialized to 0. **String** and **object** are initialized as **null**.

2. Parameterless Constructor

- When we create a constructor without parameters, it is known as a parameterless constructor. For example:

```
using System;

namespace Constructor {
  class Car {
    // parameterless constructor which overrides the default constructor!
    Car() {
      Console.WriteLine("Car Constructor");
    }
    static void Main(string[] args) {
      // call constructor
      new Car();
      Console.ReadLine();
    }
  }
}
```

Output:

```
Car Constructor
```

- In the above example, we have created a constructor named **Car().**

```
new Car();
```

- We can call a constructor by adding a **new** keyword to the constructor name.

3. C# Parameterized Constructor

- In C#, a constructor can also accept **parameters**. It is called a **parameterized constructor**.

Example:

```
using System;

namespace Constructor {

  class Car {
    string brand;
    int price;
    // parameterized constructor
    Car(string the_brand, int the_Price) {
      brand = the_brand;
      price = the_price;
    }

    static void Main(string[] args) {
      // call parameterized constructor
      Car car1 = new Car("Bugatti", 50000);
      Console.WriteLine("Brand: " + car1.brand);
      Console.WriteLine("Price: " + car1.price);
      Console.ReadLine();
    }
  }
}
```

Output

```
Brand: Bugatti
Price: 50000
```

- In the above example, we have created a constructor named **Car()**.
- The constructor takes two parameters: **theBrand** and **thePrice**.
- Notice the statement:

```
Car car1 = new Car("Bugatti", 50000);
```

- Here, we are passing the two values to the constructor.
- The values passed to the constructor are called arguments.
- We must pass the same number and type of values as parameters.

4. Copy Constructor in C#

- To be discussed depending on time.

5. Private Constructor in C#

- To be discussed depending on time.

C# Constructor Overloading

- In C#, we can create two or more constructors in a class.
- It is known as constructor **overloading**. For example:

```
using System;

namespace ConstructorOverload {

  class Car {

    // constructor with no parameter
    Car() {
      Console.WriteLine("Car constructor");
    }

    // constructor with one parameter
    Car(string brand) {
      Console.WriteLine("Car constructor with one parameter");
      Console.WriteLine("Brand: " + brand);
    }

    static void Main(string[] args) {

      // call constructor with no parameter
      Car car = new Car();

      Console.WriteLine();

      // call constructor with parameter
      Car car2 =  new Car("Bugatti");

      Console.ReadLine();
    }
  }
}
```

Output:

```
Car constructor

Car constructor with one parameter
Brand: Bugatti
```

- In the above example, we have overloaded the **Car** constructor:

  - One constructor no parameter
  - Another has one parameter

- Based on the number of the argument passed during the constructor call, the corresponding constructor is called

  - Object **car** - calls constructor no parameter
  - Object **car2** - calls constructor with one parameter

C# Constructor Overloading

- In C#, like **method overloading**, we can also overload constructors. For constructor overloading, there must be two or more constructors with the same name but different:

  1. number of parameters
  2. types of parameters
  3. order of parameters

- We can perform constructor overloading in the following ways:

1. Different number of parameters

- We can overload the constructor if the number of parameters in a constructor are different.

```
class Car {

  Car() {
    ...
  }

  Car(string brand) {
    ...
  }

  Car(string brand, int price) {
    ...
  }

}
```

- Here, we have three constructors in class Car. It is possible to have more than one constructor because the number of parameters in constructors is different.
- Notice that:
    1. **Car()** { } - has no parameter
    2. **Car(string brand) { }** - has one parameter
    3. **Car(string brand, int price) { }** - has two parameters

- Constructor Overloading with different number of parameters. Example:

```
using System;

namespace ConstructorOverload {

  class Car {

    // constructor with no parameter
    Car() {
      Console.WriteLine("Car constructor");
    }

    // constructor with one parameter
    Car(string brand) {
      Console.WriteLine("Car constructor with one parameter");
      Console.WriteLine("Brand: " + brand);
    }

    static void Main(string[] args) {

      // call with no parameter
      Car car = new Car();

      Console.WriteLine();

      // call with one parameter
      Car car2 =  new Car("Bugatti");

      Console.ReadLine();
    }
  }
}
```

Output:

```
Car constructor

Car constructor with one parameter
Brand: Bugatti
```

- In the above example, we have overloaded the Car constructor:

  1. One constructor has no parameter
  2. Another has one parameter

- Based on the number of the argument passed during the constructor call, the corresponding constructor is called:
  - Object **car** - calls constructor with no parameter
  - Object **car2** - calls constructor with one parameter

2. Different types of parameters

```
class Car {

  Car(string brand) {
    ...
   }

  Car(int price) {
    ...
  }
}
```

- Here, we have two Car constructors with the same number of parameters. We are able to create constructors with the same parameters because the data type inside the parameters is different. Notice that:

  - **Car(string brand) { }** - has parameter of **string** type
  - **Car(int price) { }** - has parameter of **int** type

- Constructor overloading with different types of parameters. Example:

```
using System;

namespace ConstructorOverload {

  class Car {

    // constructor with string parameter
    Car(string brand) {
      Console.WriteLine("Brand: " + brand);
    }

    // constructor  with int parameter
    Car(int price) {
      Console.WriteLine("Price: " + price);
    }

    static void Main(string[] args) {

      // call constructor  with string parameter
      Car car = new Car("Lamborghini");

      Console.WriteLine();

      // call constructor  with int parameter
      Car car2 =new Car(50000);

      Console.ReadLine();
    }
  }
}
```

Output

```
Brand: Lamborghini

Price: 50000
```

- In the above program, we have overloaded the constructor with different types of parameters:

  1. Object **car** - calls constructor with **string** type parameter
  2. Object **car2** - calls constructor with **int** type parameter

3. Different order of parameters

```
Car {
  Car(string brand, int price) {
    ...
  }
  Car(int speed, string color) {
    ...
  }
}
```

- Here, we have two constructors with the same number of parameters. This is possible because the order of data type in parameters is different. Notice that:
  - **Car(string brand, int price) { }** - **string** data type comes before **int**
  - **Car(int speed, string color) { }** - **int** data type comes before **string**

- Constructor overloading with different order of parameters. Example:

```
using System;

namespace ConstructorOverload {

  class Car {
    // constructor with string and int parameter
    Car(string brand, int price) {
      Console.WriteLine("Brand: " + brand);
      Console.WriteLine("Price: " + price);
    }

    // constructor with int and string parameter
    Car(int speed, string color) {
      Console.WriteLine("Speed: " + speed + " km/hr");
      Console.WriteLine("Color: " + color);
    }
    static void Main(string[] args) {
      // call constructor  with string and int parameter
      Car car = new Car("Bugatti", 50000);
      Console.WriteLine();

      // call constructor with int and string parameter
      Car car2 = new Car(60, "Red");
```

```
      Console.ReadLine();
    }
  }
}
```

Output

```
Brand: Bugatti
Price: 50000

Speed: 60 km/hr
Color: Red
```

- In the above program, we have overloaded the constructors with different orders of parameters:

    - Object **car** - calls constructor with **string** and **int** parameter respectively
    - Object **car2** - calls constructor with **int** and **string** parameter respectively

C# Polymorphism

- Polymorphism simply means occurring in **more than one form**.
- The same entity (method or operator or object) can perform different operations in different scenarios.

Example:

```
using System;

class Program
{
    // method does not take any parameter
    public void Greet()
    {
        Console.WriteLine("Hello");
    }


    // method takes one string parameter
    public void Greet(string name)
    {
        Console.WriteLine("Hello " + name);
    }


    static void Main(string[] args)
    {
        Program p1 = new Program();

        // calls method without any argument
        p1.Greet();

        //calls method with an argument
        p1.Greet("Tim");

    }
}
```

Output

```
Hello
Hello Tim
```

- In the above example, we have created a class **Program** inside which we have two methods of the same name **greet().**
- One of the **greet()** methods takes no parameters and displays "Hello". While the other **greet()** method takes a parameter and displays "Hello Tim".
- Hence, the **greet()** method behaves differently in different scenarios.
- Or, we can say greet() is **polymorphic**.

Types of Polymorphism

- There are two types of polymorphism:

  - Compile Time Polymorphism / Static Polymorphism
  - Run-Time Polymorphism / Dynamic Polymorphism

1. Compile Time Polymorphism

- At **compile time** polymorphism, the compiler identifies which method is being called before executing the program.
- In C#, we achieve compile time polymorphism in 2 ways:
  - Method overloading
  - Operator overloading

C# Method Overloading

- In a C# class, we can create methods with the same name in a class if they have:
  1. Different number of parameter
  2. Types of parameters

For example:

```
void TotalSum() {...}
void TotalSum(int a) {...}
void TotalSum(int a, int b) {...}
void TotalSum(float a, float b) {...}
```

- Here we have different types and numbers of parameters in **totalSum()**. This is known as method overloading in C#. The same method will perform different operations based on the parameter.

Look at the example below:

```
using System;
class Program
{

    // method adds two integer numbers
    void TotalSum(int a, int b)
    {
        Console.WriteLine("The sum of numbers is " + (a + b));
    }

    // method adds two double-type numbers
    // totalSum() method is overloaded
    void TotalSum(double a, double b)
    {
        Console.WriteLine("The sum of numbers is " + (a + b));
    }

    static void Main(string[] args)
    {

        Program sum1 = new Program();
        sum1.TotalSum(5, 7);
        sum1.TotalSum(53.5, 8.7);
    }
}
```

Output

```
The sum of numbers is 12
The sum of numbers is 62.2
```

- In the above example, the class **Program** contains a method named **TotalSum()** that is overloaded. The **TotalSum()** method prints:
  - sum of integers if two integers are passed as an argument
  - sum of doubles if two doubles are passed as an argument

C# Operator Overloading

- Some operators in C# behave differently with different operands. For example,

+ operator is overloaded to perform numeric addition as well as string concatenation and

- Now let's see how we can achieve polymorphism using operator overloading.

The **+ operator** is used to add two entities. However, in C#, the **+ operator** performs two operations:

1. Adding two numbers,

```
int x = 7;
int y = 5;
int sum = x + y;
Console.WriteLine(sum);
// Output: 12
```

2. Concatenating two strings,

```
string first_string = "Harry";
string second_string = "Styles";

string concatenated_string = first_string + second_string;
Console.WriteLine(concatenated_string);
// Output: HarryStyles
```

- Here, we can see that the **+ operator** is **overloaded** in C# to perform two operations: addition and concatenation.

2. Runtime Polymorphism

- In runtime polymorphism, the method that is called is determined at the runtime not at compile time.
- The runtime polymorphism is achieved by: **Method Overriding**

Polymorphism

Method Overriding in C#

- Using polymorphism in C#, if the same method is present in both the superclass (base) and the subclass (derived) then, the method in the subclass **overrides** the same method in the superclass. This is called **method overriding**.
- In this case, the same method will perform one operation in the superclass and another operation in the subclass.
- We can use **virtual** and **override** keywords to achieve method overriding.
- Without using **virtual** or **override**, the base class method is always used.

Look at this example:

```
using System;
class Polygon
{
    // method to Render a shape
    public virtual void Render()
    {
        Console.WriteLine("Rendering Polygon...");
    }

}

class Square : Polygon
{
    // overriding Render() method
    public override void Render()
    {
        Console.WriteLine("Rendering Square...");
    }

}
class MyProgram
{
    public static void Main()
    {
        // obj1 is the object of Polygon class
        Polygon obj1 = new Polygon();

        // calls render() method of Polygon Superclass
        obj1.Render();

        // here, obj1 is the object of derived class Square
        obj1 = new Square();

        // calls render() method of derived class Square
        obj1.Render();
    }
}
```

Output

```
Rendering Polygon...
Rendering Square…
```

- In the above example, we have created a superclass: Polygon and a subclass: **Square**.
- We must use **virtual** and **override** with methods of the base class and derived class respectively. So:
  - **virtual** - allows the method to be overridden by the derived class
  - **override** - indicates the method is overriding the method from the base class
- In this way, we achieve **method overriding** in C#.

Note: A method in derived class overrides the method in base class if the method in derived class has the **same name**, **same return type** and **same parameters** as that of the base class.

Why Polymorphism?

- Polymorphism allows us to create consistent code. In the previous example, we can also create a different method: **RenderSquare()** to render Square.
- This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.
- To solve this, polymorphism in C# allows us to create a single method **Render()** that will **behave differently for different shapes** while still using the **same** calling method name.

C# Scope

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

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

C# Class Level Variable 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 scoped variable can be accessed by the non-static methods of the class in which it is declared.
- Access modifier of class level variables doesn't affect their scope within a class.
- Member variables can also be accessed outside the class by using the access modifiers.
- Class level variables are known as fields, and they are declared outside of methods, constructors, and blocks of the class. For example:

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

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

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

    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.

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

- 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 Variable Scope

- Variables that are declared inside a method have **method level scope**. These **are not** accessible outside the method.
- However, these variables can be accessed by the nested code blocks inside a method.
- These variables are termed as the local variables.
- There will be a compile-time error if these variables are declared twice with the same name in the same scope.
- **These variables don't exist after method's execution is over.**
  For example:

```
using System;

namespace VariableScope {
  class Program {

    public void Method1() {
      // display variable inside method
      string str = "method level";
    }

    public void Method2() {

      // accessing str from 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
```

- we get an error.

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

- This is because method level variables have scope inside the method where they are created. For example:

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

    public void Display() {
     string str = "inside method";

      // accessing method level variable
      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 errors.

Block Level Variable Scope in C#

- 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**
- These variables are also termed as the loop variables or statements variable as they have limited their scope up to the body of the statement in which it declared.
- Generally, a loop inside a method has three level of nested code blocks(i.e., class level, method level, loop level).
- The variable which is declared outside the loop is also accessible within the nested loops. It means a class level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.
- A variable which is declared inside a loop body will not be visible to the outside of loop body.

For 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
```

C# static Keyword

- In C#, if we use a **static** keyword with **class** members, then there will be a **single copy** of the type member.
- All objects of the class **share a single copy** instead of creating individual copies.

C# Static Variables

- If a variable is declared static, we can access the variable using the class name. For example,:

```
using System;

namespace StaticKeyword {

  class Student {

    // static variable
    public static string department = "Computer Science";
  }

  class Program {
    static void Main(string[] args) {

      // access static variable
      Console.WriteLine("Department: " + Student.department);

      Console.ReadLine();
    }
  }
}
```

Output

```
Department: Computer Science
```

- In the above example, we have created a static variable named **department**. Since the variable is **static**, we have used the class name **Student** to access the variable.

Static Variables Vs Instance Variables

- In C#, every object of a class will have its own copy of instance variables. For example,

```
class Student {

  // instance variable
  public string student_name;
 }

class Program {
  static void Main(string[] args) {

    Student s1 = new Student();
    Student s2 = new Student();
  }
}
```

- Here, both the objects **s1** and **s2** will have separate copies of the variable **student_name**. And they are different from each other.
- However, if we declare a variable **static**, all objects of the class share the same static variable. And we don't need to create objects of the class to access the static variables.

Example: C# Static Variable Vs. Instance Variable

```
using System;

namespace StaticKeyword {

  class Student {
    public static string school_name = "CS School";
    public string student_name;
  }

    class Program {
    static void Main(string[] args) {

      Student s1 = new Student();
      s1.student_name = "Julie";

      // calls instance variable
      Console.WriteLine("Name: " + s1.student_name);
```

```
      // calls static variable
      Console.WriteLine("School: " + Student.school_name);

      Student s2 = new Student();
      s2.student_name = "Sam";

       // calls instance variable
      Console.WriteLine("Name: " + s2.student_name);
      // calls static variable
      Console.WriteLine("School: " + Student.school_name);

      Console.ReadLine();
    }
  }
}
```

Output

```
Name: Julie
School: CS School
Name: Sam
School: CS School
```

- In the above program, the **Student** class has a non-static variable named **student_name** and a static variable named **school**_name.
- Inside the Program class,
  - **s1.student_name** / **s2.student_name** - calls the non-static variable using objects s1 and s2 respectively
  - **Student.school_name** - calls the static variable by using the class name
- Since the **school_name** is the same for all students, it is good to make the school_name static. It saves memory and makes the program more efficient.

C# Static Methods

- Just like static variables, we can call the static methods using the class name.

```
class Test {
  public static void display() {....}
}

class Program {
  static void Main(string[] args) {
    Test.display();
  }
}
```

- Here, we have accessed the static method directly from **Program** classes using the class name.
- When we declare a method **static**, all objects of the class share **the same static method**.

Example: C# Static and Non-static Methods

```
using System;

namespace StaticKeyword {

  class Test {

    public void Display1() {

      Console.WriteLine("Non static method");
    }
    public static void Display2() {

      Console.WriteLine("Static method");
    }
  }

  class Program {
    static void Main(string[] args) {

      Test t1 = new Test();
      t1.Display1();
      Test.Display2();
      Console.ReadLine();
    }
  }
}
```

Output

```
Non static method
Static method
```

- In the above program, we have declared a non-static method named **Display1()** and a static method named **Display2()** inside the class Test.
- Inside the Program class,
  - **t1.Display1()** - access **the non-static** method using **s1** object
  - **Test.Display2()** - access **the static method** using the class name **Test**

Note: In C#, the Main method is **static**. So, we can call it without creating the object.

C# Static Class

- In C#, when we declare a class as static, we cannot create objects of the class. For example,

```
using System;

namespace StaticKeyword {

  static class Test {
    static int a = 5;
    static void Display() {

      Console.WriteLine("Static method");
    }

    static void Main(string[] args) {

      // creating object of Test
      Test t1 = new Test();
      Console.WriteLine(a);
      Display();
    }
  }
}
```

- In the above example, we have a static class **Test**. We have created an object **t1** of the class **Test**.
- Since we cannot make an object of the static class, we get the following error:

```
error CS0723: Cannot declare a variable of static type 'Test'
error CS0712: Cannot create an instance of the static class
```

- Notice the field and method of the static class are also static because we can only have static members inside the static class.
- **Note**: We cannot inherit a static class in C#. For example,

```
static class A {
  ...
}
// Error Code
class B : A {
  ...
}
```

Access static Members within the Class

- If we are accessing the static variables and methods inside the same class, we can directly access them without using the class name. For example,

```
using System;

namespace StaticKeyword {

  class Test {

    static int age = 25;
    public static void Display() {

      Console.WriteLine("Static method");
    }

    static void Main(string[] args) {

      Console.WriteLine(age);
      Display();
      Console.ReadLine();
    }
  }
}
```

Output

```
25
Static method
```

- Here, we are accessing the static field **age** and static method **Display()** without using the class name.

C# Preprocessor directives

- Preprocessor directives are a block of statements that gets processed before the actual compilation starts. C# preprocessor directives are the commands for the compiler that affects the compilation process.
- These commands specifies which sections of the code to compile or how to handle specific errors and warnings.
- C# preprocessor directive begins with **a** `# (hash)` symbol and all preprocessor directives last for one line. Preprocessor directives are terminated by `new line` rather than `semicolon`.

`#region`	Allows us to create a region that can be expanded or collapsed when using a Visual Studio Code Editor	`#region region-description`
		`codes`
		`#endregion`

`#endregion`	Indicates the end of a region	`#region region-description`
		`codes`
		`#endregion`

#region and #endregion directive

- The `#region` directive allows us to create a region that can be expanded or collapsed when using a Visual Studio Code Editor.
- This directive is simply used to organize the code.
- The #region block cannot overlap with a `#if` block. However, a `#region` block can be included within a `#if` block and a `#if` block can overlap with a `#region` block.
- `#endregion` directive indicates the end of a `#region` block.

Example:

```
class Program
{
    #region Application entry point
    0 references
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    1 reference
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

```
class Program
{
    Application entry point

    My method
}
```

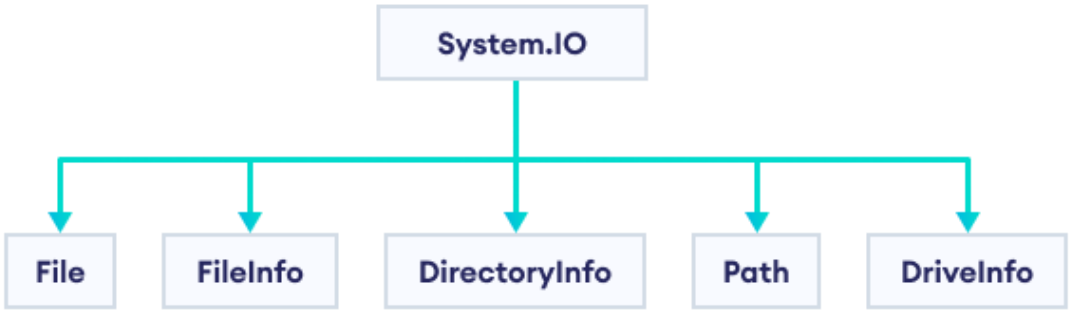
C# File I/O

What are Files and Directory ?

- A file is a named location that can be used to store related information.
- For example, **program.cs** is a C# file that contains information about the C# program.
- A **directory** is a collection of files and subdirectories. A directory inside a directory is known as a **subdirectory**.

Working with Files in C#

- C# provides a **System.IO** namespace that contains several classes that are used to perform operations on files and dictionaries.



Some classes under System.IO namespace

- The above image shows some of the classes under the **System.IO** namespace. Among these classes, we will learn about the **File** class and it's methods to work with files in C#.

C# File Class and Its Methods

- The **File** class provides us built-in methods that allow us to perform input / output operations on files. Some of the commonly used methods are:

Methods	Use
`Create()`	Create or overwrite a file in the specified path.
`Open()`	Opens a `FileStream` on the specified path with read / write access
`WriteAllText()`	Create a new file, writes the specified string to the file, and then closes the file
`ReadAllText()`	Opens a text file, reads all lines of the file, and then closes the file.
`Copy()`	Copies an existing file to a new file. Overwriting a file of the same name is not allowed.
`AppendAllText()`	Opens a file, appends the specified string to the file, and then closes the file. If the file does not exist, this method creates a file, writes the specified string to the file, then closes the file.

Create a File in C#

- We use the **Create()** method of the **File** class to create a new file in C#. For example,

```
// create a file at path_name
FileStream fs = File.Create(path_name);
```

- Here, the **File** class creates a file at **path_name**.

Note: If the file already exists, the Create() method overwrites the file.

Example: Create a File

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        // path of the file that we want to create
        string path_name = @"C:\Program\myFile.txt";

        // Create() creates a file at path_name
        FileStream fs = File.Create(path_name);

        // check if myFile.txt file is created at the specified path
        if (File.Exists(path_name))
        {
            Console.WriteLine("File is created.");
        }
        else
        {
            Console.WriteLine("File is not created.");
        }
    }
}
```

Output

```
File is created.
```

- In the above example, we have created a file `myFile.txt` at `C:\Program` directory using the `Create()` method.
- After creating the file, notice that we have used the `Exists()` method to check whether the file `myFile.txt` exists or not.

Note: The "@" in front of "C:\Program\myFile.txt" indicates this as a verbatim string. We use verbatim string to tell the compiler to ignore escape character \.

Open a File

- We use the **Open()** method of the **File** class to open an existing file in C#. The method opens a **FileStream** on the specified file. For example,

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        string path_name = @"C:\Program\myFile.txt";
       // open a file at path_name
        FileStream fs = File.Open(path_name, FileMode.Open);
    }
}
```

- In the above example, notice the code,

```
// opens the file at path_name
FileStream fs = File.Open(path_name, FileMode.Open);
```

Here, the `Open()` method opens `myFile.txt` file. Here, `FileMode.Open` specifies - **open the existing file**.

Note: A file stream is a sequence of bytes used to hold file data. Every file contains at least one file stream.

Write to a File

- We use the **WriteAllText()** method of the **File** class to write to a file. The method creates a new file and writes content to that file.

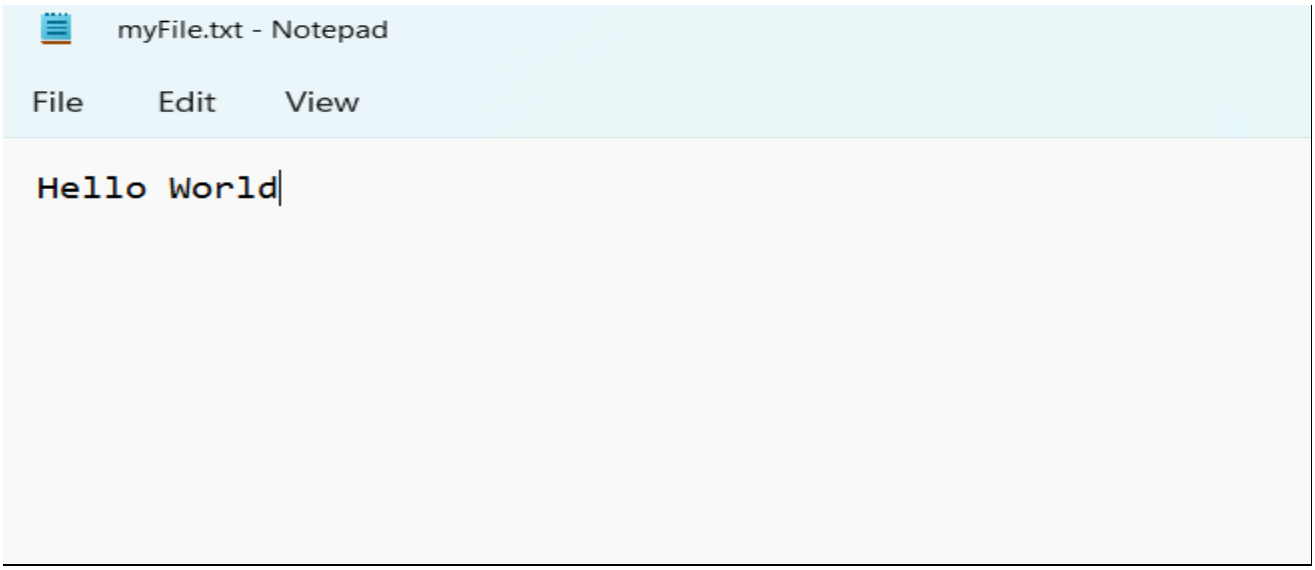
Example:

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        string path_name = @"C:\Program\myFile.txt";
        // create a file at path_name and write "Hello World" to the file
        File.WriteAllText(path_name, "Hello World");
    }
}
```

- In the above example, notice the code,

```
File.WriteAllText(path_name, "Hello World");
```

- Here, the `WriteAllText()` method creates `myFile.txt` at `C:\Program` directory and writes `"Hello World"` to the file.



Writing text in a File

- The above image shows the `myFile.txt` file that contains the text `"Hello World"`.

Note: If the file already exists, the WriteAllText() method overwrites the file.

Read a File in C#

- We use the `ReadAllText()` method of the `File` class to read contents of the file. The method returns a string containing all the text in the specified file.
- Let's read the content of the file `myFile.txt` where we had written `"Hello World"`.

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        string path_name = @"C:\Program\myFile.txt";

        // read the content of myFile.txt file
        string read_txt = File.ReadAllText(path_name);
        Console.WriteLine(read_text);
    }
}
```

Output

```
Hello World
```

- In the above example, notice the code,

```
// read the content of myFile.txt file
string read_text = File.ReadAllText(path_name);
```

- The `ReadAllText()` method reads the file `myFile.txt` and returns `"Hello World"`.

C# Exception Handling

- An exception is an unexpected event that occurs during program execution. For example,

```
int divide_by_zero = 7 / 0;
```

- The above code causes an exception as it is not possible to divide a number by **0**.
- Exceptions abnormally terminate the flow of the program instructions; we need to handle those exceptions. Responding to or handling exceptions is called **Exception Handling**.

C# Exception Handling Blocks

- C# provides built-in blocks to handle exceptions. They are `try..catch` and `finally`.

Using try…catch block

- The `try..catch` block is used to handle exceptions in C#. Here's the syntax of `try...catch` block:

```
try
{
    // code that may raise an exception
}
catch (Exception e)
{
    // code that handles the exception
}
```

- Here, we place the code that might generate an exception inside the `try` block. The `try` block then throws the exception to the `catch` block which handles the raised exception.

Example:

```
try
{
    int num = 0;
    // code that may raise an exception
    int divide_by_zero = 7 / num;
}
catch (Exception e)
{
    Console.WriteLine("Exception has occurred");
}
```

- Here, we are trying to divide a number by **zero**. In this case, an exception occurs. Hence, we have enclosed this code inside the `try` block. The `catch` block notifies the user about the occurred exception.

Example: Exception Handling Using try…catch

```
using System;
class Program
{
    static void Main()
    {
        string[] colors = { "Red", "Blue", "Green" };

        try
        {
            // code that may raise an exception
            Console.WriteLine(colors[5]);

        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine("An exception occurred: " + e.Message);
        }
    }
}
```

Output

```
An exception occurred: Index was outside the bounds of the array.
```

- In the above example, notice the code,

```
Console.WriteLine(colors[5]);
```

- Since there is no element at index **5** of the `colors` array, the above code raises an exception. So we have enclosed this code in the `try` block.
- When the program encounters this code, `IndexOutOfRangeException` occurs. And the exception is caught by the `catch` block and executes the code inside the `catch` block.
- Look at the code,

```
catch (IndexOutOfRangeException e)
{
    Console.WriteLine("An exception occurred: " + e.Message);
}
```

- Here, `catch` is taking an instance of the `IndexOutOfRangeException` class. And inside the block we have used the `Message` property of this class to display a message.

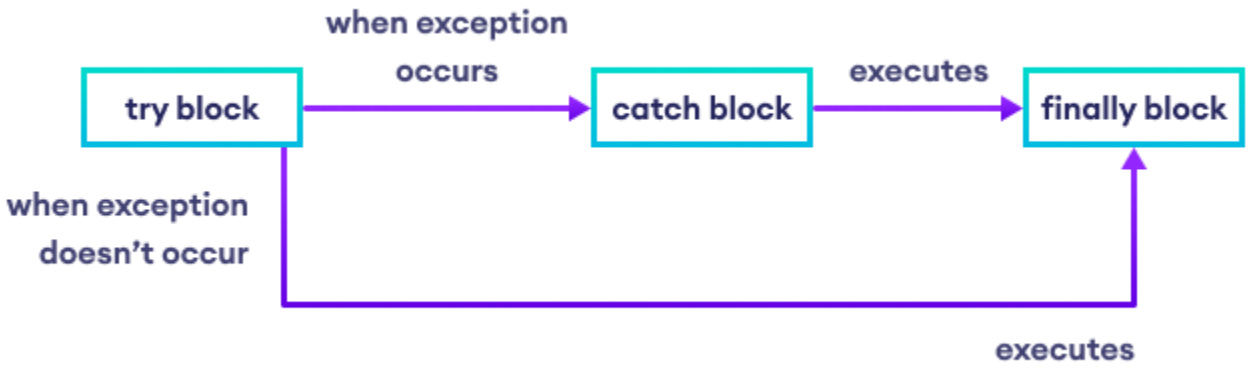
Note: If you want to learn more about the Exception class methods and properties visit here:

https://learn.microsoft.com/en-us/dotnet/standard/exceptions/exception-class-and-properties

Using try…catch…finally block

- We can also use `finally` block with `try` and `catch` block. The `finally` block is always executed whether there is an exception or not.
- The syntax of the `try…catch…finally` block is:

```
try
{
    // code that may raise an exception
}
catch (Exception e)
{
    // code that handles the exception
}
finally
{
    // this code is always executed
}
```



C# finally block

- We can see in the above image that the `finally` block is executed in both cases. The `finally` block is executed:
- after `try` and `catch` block - when exception has occurred
- after `try` block - when exception doesn't occur
- The `try..catch..finally` block can be collectively used to handle exceptions.
- Now let's look at an example of exception handling using `try…catch…finally`.

Example: Exception Handling Using try…catch…finally block

```
using System;
public class Program
{
    static void Main()
    {
        // take first int input from user
        Console.WriteLine("Enter first number:");
        int first_number = int.Parse(Console.ReadLine());

        // take second int input from user
        Console.WriteLine("Enter second number:");
        int second_number = int.Parse(Console.ReadLine());

        try
        {
            // code that may raise raise an exception
            int division_result = first_number / second_number;
            Console.WriteLine("Division of two numbers is: " + division_result);
        }
```

```
        // this catch block gets executed only when an exception is raised
        catch (Exception e)
        {
            Console.WriteLine("An exception occurred: " + e.Message);
        }

        finally
        {
            // this code is always executed whether of exception occurred or not
            Console.WriteLine("Sum of two numbers is: " + (first_number + second_number));
        }
    }
}
```

Output

```
Enter first number:
8
Enter second number:
0
An exception occurred: Attempted to divide by zero.
Sum of two numbers is: 8
```

- In the above example, we have tried to perform division and addition operations to two `int` input values using `try...catch…finally`.
- Look at the code,

```
try
{
    // code that may raise raise an exception
    int division_result = first_number / second_number;
}
```

- Here, we have enclosed the code that performs division operation inside try because this code may raise the **DivideByZeroException** exception.
- There are two cases in this program:

  Case I - When exception occurs in `try`, the `catch` block is executed followed by the `finally` block.
  Case II - The `finally` block is directly executed after the `try` block if an exception doesn't occur. For example, if we enter **9** and **2**, exception doesn't occur in the `try` block and we get the following output:

```
// Output when exception doesn't occur
Enter first number:
9
Enter second number:
2
Division of two numbers is: 4
Sum of two numbers is: 11
```

C# Nested try-catch

- In C#, a `try…catch` block inside another `try…catch` block is called nested `try…catch`.

Example:

```
using System;
class Program
{
    static void Main()
    {
        int divisor = 0;
        try
        {
            // nested try block
            try
            {
                int divide_by_zero = 6 / divisor;
            }
            // inner catch block
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("Inner catch is executed. " + e.Message);
            }
        }
        // outer catch block
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Outer catch block is executed. " + e.Message);
        }
    }
}
```

Output

```
Outer catch block is executed. Attempted to divide by zero.
```

- In the above example, we have used a try-catch block **inside** another try-catch block.
- The inner `catch` block gets executed when the raised exception type is `IndexOutOfRangeException`. However, in our program, `DivideByZeroException` is raised in the outer `try` block, so the outer `catch` block gets executed.

C# Generic catch block

- The `catch` block without exception class is called generic `catch` block.

Example:

```
using System;
class Program
{
    static void Main()
    {
        int[] numbers = { 1, 2, 3 };
        try
        {
            // access elements present at 5th index position
            Console.WriteLine(numbers[5]);
        }

        // generic catch block
        catch
        {
            Console.WriteLine("Some exception occurred");
        }

    }
}
```

Output

```
Some exception occurred
```

Here, we have used a generic `catch` block with the `try` block.
Note: The generic catch block can catch and handle any type of exception thrown by the try block.

Exception Handling Using Two catch Blocks

- In C#, we can use multiple `catch` blocks to handle exceptions. If non-generic `catch` blocks don't handle the exception, then generic `catch` blocks are executed.

Example:

```
using System;
class Program
{
    static void Main()
    {
        int[] numbers = { 1, 0, 3, 4 };
        try
        {
            // code that may raise an exception
            int divide = numbers[2] / numbers[1];
        }

        // if IndexOutofRangeException occurs, the following block is executed
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine(e.Message);
        }

        // if the above catch block doesn't handle the exception,
        // this block is executed
        catch(Exception e)
        {
            Console.WriteLine("Some exception occurred");
        }

    }
}
```

Output

```
Some exception occurred
```

- Here, inside the `try` block, the `IndexOutofRangeException` exception is not raised hence the corresponding catch block doesn't handle the exception.

C# this Keyword

- In C#, **this** keyword refers to the **current** instance of a class. For example:

```
using System;

namespace ThisKeyword {
  class Test {

    // num will be modified using this as per below
    int num;

    Test(int num) {
      // this.num refers to the instance field
      this.num = num;

      Console.WriteLine("object of this: " + this);
    }

    static void Main(string[] args) {

      Test t1 = new Test(4);
      Console.WriteLine("object of t1: " + t1);
      Console.ReadKey();
    }
  }
}
```

Output

```
object of this: ThisKeyword.Test
object of t1: ThisKeyword.Test
```

- In the above example, we have created an object named `t1` of the class `Test`. We have printed the name of the object `t1` and `this` keyword of the class.
- Here, we can see the name of both `t1` and `this` is the same. This is because `this` keyword refers to the current instance of the class which is `t1`.

Here are some of the major uses of this keyword in C#.

C# with same named variables

- We cannot declare two or more variables with the same name inside a scope (class or method). However, instance variables and parameters may have the same name.

```
using System;

namespace ThisKeyword {
  class Test {

    int num;
    Test(int num) {
      num = num;
    }

    static void Main(string[] args) {
      Test t1 = new Test(4);
      Console.WriteLine("value of num: " + t1.num);
      Console.ReadKey();
    }
  }
}
```

Output

```
0
```

- In the above program, the instance variable and the parameter have the same name: `num`. We have passed `4` as a value to the constructor.
- However, we are getting **0** as an output. This is because C# does not differentiate the names of the instance variable and the parameter are the same.
- We can solve this issue by using `this`.

C# using this with same named variables

```
using System;

namespace ThisKeyword {
  class Test {

    int num;
    Test(int num) {

      // this.num refers to the instance field
      this.num = num;
    }

    static void Main(string[] args) {
      Test t1 = new Test(4);
      Console.WriteLine("value of num: " +t1.num);
      Console.ReadLine();
    }
  }
}
```

Output

```
value of num: 4
```

- Now, we are getting the expected output that is **4**. It is because `this.num` refers to the instance variable of the class.
- Now C# can differentiate between the names of the instance variable and the parameter.

Invoke Constructor of the Same Class Using this

- While working with constructor overloading, we might have to invoke one constructor from another constructor. In this case, we can use `this` keyword.

```
using System;

namespace ThisKeyword {
  class Test {

    Test(int num1, int num2) {
      Console.WriteLine("Constructor with two parameter");
    }

    // invokes the constructor with 2 parameters
    Test(int num) : this(33, 22) {
      Console.WriteLine("Constructor with one parameter");
    }

    public static void Main(String[] args) {
      Test t1 = new Test(11);
      Console.ReadLine();
    }
  }
}
```

Output

```
Constructor with two parameter
Constructor with one parameter
```

- In the above example, we have used `:` followed by `this` keyword to call constructor `Test(int num1, num2)` from the constructor `Test(int num)`.
- When we call the `Test(int num)` constructor the `Test(int num1, int num2)` constructor executes first.

Note: Calling one constructor from another constructor is known as **constructor chaining**.

C# this as an object argument

- We can use `this` keyword to pass the current object as an argument to a method.

```
using System;

namespace ThisKeyword {

  class Test {
    int num1;
    int num2;

    Test() {
      num1 = 22;
      num2 = 33;
    }
    // method that accepts this as argument
    void PassParameter(Test t1) {
      Console.WriteLine("num1: " + num1);
      Console.WriteLine("num2: " + num2);
    }
    void display() {
      // passing this as a parameter
      PassParameter(this);
    }
    public static void Main(String[] args) {
      Test t1 = new Test();
      t1.display();
      Console.ReadLine();
    }
  }
}
```

Output

```
num1: 22
num2: 33
```

- In the above program, we have a method `PassParameter()`. It accepts the object of the class as an argument.

```
PassParameter(this);
```

- Here, we have passed `this` to the `PassParameter()` method. As `this` refers to the instance of the class, we are able to access the value of `num1` and `num2`.

C# generics

- C# Generics allow us to create a single class or method that can be used with **different types of data**. This helps us to reuse our code.

C# generics Class

- A generics class is used to create an instance of any data type. To define a generics class, we use angle brackets (`<>`) as,

```
class Student<T>
{
  // Block of code
}
```

- Here, we have created a generics class named `Student`. `T` used inside the angle bracket is called the **type parameter**.
- While creating an instance of the class, we specify the data type of the object which replaces the type parameter.

Create an Instance of Generics Class

- Here are two instances of the generics class.

```
// Create an instance with data type string
Student<string> student_name = new Student<string>();
// create an instance with data type int
Student<int> student_id = new Student<int>();
```

- Here, we have created two instances named `student_name` and `student_id` with data types `string` and `int`, respectively.
- During the time of compilation, the type parameter `T` of the `Student` class is replaced by,
- `string` - for instance `studentName`
- `int` - for instance `studentId`

C# generic Class

```
using System;

// Define a generics class named Student
public class Student<T>
{
    // Define a variable of type T
    public T data;

    // Define a constructor of the Student class
    public Student(T data){
        this.data = data;
        Console.WriteLine("Data passed: " + this.data);
    }
}

class Program
{
    static void Main()
    {
        // create an instance with data type string
        Student<string> student_name = new Student<string>("John");
        // Create an instance with data type int
        Student<int> student_id = new Student<int>(219003);
    }
}
```

Output

```
Data passed: John
Data passed: 219003
```

- In the above example, we have created a generic class named `Student`.
- We have also defined a constructor that prints `this` value.
- Inside the `Main` class, we have created two instances of the `Student` classes: `student_name` and `student_id`.
- The type parameter `T` of `Student<T>` is replaced by:
  - `string` - in `student_name`
  - `int` - in `student_id`
- Here, the `Student` class works with both the `int` and the `string` data type.

C# generics Method

- Similar to the generic class, we can also create a method that can be used with any type of data. Such a class is known as the **generic Method**.

```
public void DisplayData(T data) {
    Console.WriteLine("Data Passed: " + data);
}
```

- Above, `DisplayData` - name of the generic method
- `T` - type parameter to specify the function can accept any type of data
- `data` - function parameter
- Now we can use this function to work with any type of data.

```
// Calling function with integer data
Obj_name.DisplayData(34);
// Calling function with string data
Obj_name.displayData("Tim");
```

C# generics Method

- Below, we use generic Methods.

```
using System;

// Define a generic class named Employee
class Employee<T>
{
    // Define a generics method that displays the passed data
    public void DisplayData(T data1, T data2)
    {
        Console.WriteLine("The data passed is: " + data1 + "The data passed is: " + data2);
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of Employee class by specifying T as string
        Employee<string> employee_name = new Employee<string>();

        // Call DisplayData() generics method and pass a string value - "Jack"
        Employee_name.DisplayData("Jack","Smith");

        // Create an instance of Employee class by specifying T as int
        Employee<int> employee_id = new Employee<int>();

        // Call DisplayData() generics method and pass an integer value
        Employee_id.DisplayData(123);
    }
}
```

Output

```
The data passed is: Jack
The data passed is: 123
```

- In the above example, we have defined a generics method named `DisplayData()` inside the `Employee<T>` generics class.

C# Generic Method with Return Type

- Earlier we defined a generic method without a return type. However, we can also define a generics method with a return type.

```
using System;

// Define a generics class named Employee
class Movie<T>
{
    // Define a generic method that returns T type value
    public T DisplayData(T data)
    {
        return data;
    }
}

class Program
{
    static void Main()
    {
        // Create an instance with data type string
        Movie<string> movie_name = new Movie<string>();
        Console.WriteLine("Generics Method returns: " + movie_name.DisplayData("Inception"));

        // Create an instance with data type int
        Movie<int> movie_rating = new Movie<int>();
        Console.WriteLine("Generics Method returns: " + movie_rating.DisplayData(9));

    }
}
```

Output

```
Generics Method returns: Inception
Generics Method returns: 9
```

- In the above example, we have created a generics method named `DisplayData()`.
- Notice that we have used `T` as a return type instead of void or any other default type like **int**.
- This means the method can return a value of any type.

```
public T DisplayData(T data) {...}
```

- In the example above, the method is returning,
  - `string` data - `"Inception"`
  - `int` data - **9**

Advantages of Generics

1.Code Reusability

- With the help of generics in C#, we can write code that will work with different types of data.

```
public void DisplayData(T data) {...}
```

- Here, we have created a generics method.
- This same method can be used to perform operations on integer data, string data, and so on.

2.Compile-time Type Checking

- The type parameter of generics provides information about the type of data used in the generics code.

```
// Int type instance of GenericsClass
GenericsClass<int> list = new GenericsClass<>();
```

- Here, we know that `GenericsClass` is working with `int` data only.
- Now, if we try to pass data other than `int` to this class, the program will generate an error at compile time.

3.Used with Collections

- The collections framework uses the concept of generics in C#. For example:

```
// Create a string type List
List<string> course_name = new List<string>();

// Create an int type List
List<int> course_id = new List<int>();
```

- In the above example, we have used the same `List` class to work with different types of data.
- Similar to `List`, other collections (`Queue`, `Stack`, and so on) are also generic in C#

C# Generics Method with Non-Generics Class

- We can also define a generics method inside a **non-generics** class. For example:

```
using System;

// Create a non-generic class named Gadget
public class Gadget
{
    // Define a generic method
    public T DisplayData<T>(T data)
    {
        return data;
    }
}

class Program
{
    static void Main()
    {
        Gadget gadget_name = new Gadget();

        // Call generic method displayData() and pass string data
        Console.WriteLine("Generic Method returns: " + gadget_name.DisplayData("Laptop"));

        Gadget gadget_model = new Gadget();

        // Call generic method displayData() and pass integer data
        Console.WriteLine("Generic Method returns: " + gadget_model.displayData(513));
    }
}
```

Output

```
Generic Method returns: Laptop
Generic Method returns: 513
```

- In the above example, we have created a generic method named `DisplayData()` inside the non-generic class named `Gadget`. Notice the code,

```
public T DisplayData<T>(T data) {...}
```

- Here, we need to use `<T>` to define a generics method `DisplayData()` without a generics class.

C# Generics Property

- In C#, we can also create generic properties. For example,

```
using System;

public class Sport<T>
{
    // Define a generic field
    public T data;

    // Define a generic property
    public T data
    {
        get { return data; }
        set { data = value; }
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of Sport with data type string
        Sport<string> sport_name = new Sport<string>();

        // access the generic property, Data
        sport_name.Data = "Football";
        Console.WriteLine("Name of the sport is: " + sport_name.Data);
    }
}
```

Output

```
Name of the sport is: Football
```

- Here, we have defined a generics property named `Data`.

Note: Not only for property, class, or methods, we can also use generics with abstract class, interface, events, delegates, and so on.

C# Collections - List

- `List<T>` is a class that contains multiple objects of the same data type that can be accessed using an index. For example:

```
// list containing integer values
List<int> number = new List<int>() { 1, 2, 3 };
```

- Here, `number` is a `List` containing integer values (**1**, **2** and **3**).

Create a List

- To create `List<T>` in C#, we need to use the `System.Collections.Generic` namespace.

```
using System;
using System.Collections.Generic;
class Program
{
    public static void Main()
    {
        // Create a list named subjects that contain 2 elements
        List<string> subjects = new List<string>() { "English", "Math" };
    }
}
```

Access the List Elements

- We can access `List` using index notation `[]`.For example,

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a list
        List<string> languages = new List<string>() { "Python", "Java" };

        // Access the first and second elements of languages list
        Console.WriteLine("The first element of the list is " + languages[0]);
        Console.WriteLine("The second element of the list is " + languages[1]);

    }
}
```

Output

```
The first element of the list is Python
The second element of the list is Java
```

Since the index of the list starts from **0**:

- `language[0]` - accesses the first element
- `language[5]` - accesses the fourth element

Iterate the List

- In C#, we can also loop through each element of `List<T>` using a `for` loop.

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a list
        List<string> names = new List<string>() { "Joe", "Julie", "Nick" };

        // Iterate through the names list
        for (int i = 0; i < names.Count; i++)
            Console.WriteLine(names[i]);
    }
}
```

Output

```
Joe
Julie
Nick
```

- In the above example, we have looped through the `names` list using a `for` loop.

Note: The Count property returns the total number of elements inside the list.

Basic Operations on List

- The `List<T>` class provides various **methods** to perform different operations on `List`.
- The following are commonly used `List` operations:
  - **Add** Elements
  - **Insert** Elements
  - **Remove** Elements

Add Elements to List

- To add a single element to the `List`, we use the `Add()` method of the `List<T>` class.

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a list
        List<string> country = new List<string>() { "Canada" };

        // Add "Canada" to the country list
        country.Add("USA");

        // Add "USA" to the country list
        country.Add("China");

        // Iterate through the country list
        for (int i = 0; i < country.Count; i++)
            Console.WriteLine(country[i]);
    }
}
```

Output

```
Canada
USA
China
```

- In the above example, at first, we have created a `country` list that contains `"Canada"`.
- Then we added `"USA"` and `"China"` to the list using the `Add()` method.

Insert Element in a List

- To insert an element in a specified index in `List`, we use the `Insert()` method of the `List<T>` class.

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a list
        List<string> languages = new List<string>() { "Python", "Java", "C" };

        // Insert "JavaScript" at index 2
        languages.Insert(2, "JavaScript");

        // Display element at index position 2
        Console.WriteLine(languages[2]);
    }
}
```

Output

```
JavaScript
```

- `languages.Insert(2, "JavaScript")` inserts `"JavaScript"` at the 2nd index position

Remove elements from a List

- We can delete one or more items from `List<T>` using **2** methods:
  - `Remove()` - removes the first occurrence of an element from the given list
  - `RemoveAt()` - removes the elements at the specified position in the list

Remove() Method

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        var car = new List<string>() { "BMW", "Tesla", "Suzuki", "Tesla" };

        // Remove the first occurrence of "Tesla" from the list
        car.Remove("Tesla");

        // Remove the first occurrence of "Suzuki"
        car.Remove("Suzuki");

        // Print the updated list after removing
        for (int i = 0; i < car.Count; i++)
        {
            Console.WriteLine(car[i]);
        }
    }
}
```

Output

```
BMW
Tesla
```

- **car.Remove("Tesla")** - removes the **first** occurrence of "Tesla"
- **car.Remove("Suzuki)** - removes the **first** occurrence of "Suzuki"
- The original list: **{ "BMW", "Tesla", "Suzuki", "Tesla" }**
- The modified list: **{"BMW", "Tesla"}**

Example: RemoveAt() Method

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        var car = new List<string>() { "BMW", "Tesla", "Suzuki", "Tesla" };

        // Remove the element present at the 2nd index position
        car.RemoveAt(2);

        // Print the updated list after removing the element
        for (int i = 0; i < car.Count; i++)
        {
            Console.WriteLine(car[i]);

        }
    }
}
```

Output

```
BMW
Tesla
Tesla
```

- In the above example, we have removed the element of `List<T>` using the `RemoveAt()` method.
- Here, `car.RemoveAt(2)` removes `"Suzuki"` from the list, in position 2.

Another way to create a List

- We can also create a List using `var` keyword. For example,

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a list named color
        var color = new List<string>() {
            "Red",
            "Blue",
            "Pink"
        };
        Console.WriteLine(color[2]);
    }
}
```

Output

```
Pink
```

Implicitly typed local variables (source: learn.microsoft.com)

- Variables that are declared at method scope can have an implicit "type" **var**.
- An implicitly typed local variable is **strongly typed** as if you had declared the type yourself, but the compiler determines the type.
- The following two declarations of `a` and `b` are functionally equivalent:

```
var a = 10; // Implicitly typed as an int by c# compiler.
int b = 10; // Explicitly typed.
```

- A common use of the **var** keyword is with constructor invocation expressions.
- The use of **var** allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

```
var xs = new List<int>();
```

C# foreach loop

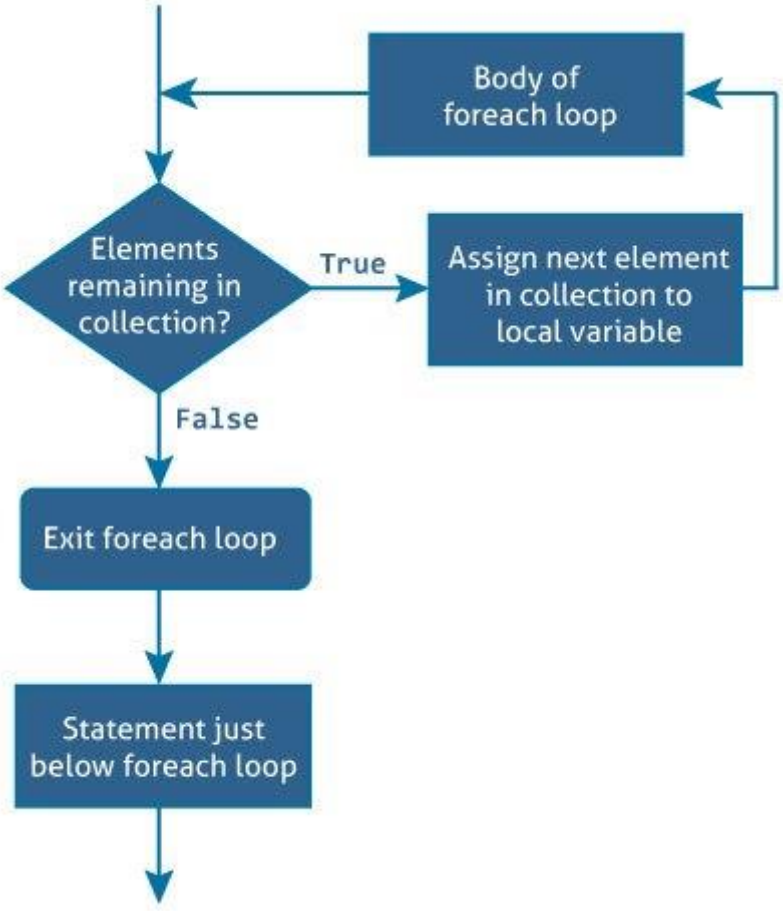
- C# provides a more readable alternative to the **for loop** using a **foreach** loop when working with arrays and collections to **iterate** through the items of arrays or collections.
- The **foreach** loop iterates through each item based on the size and definition of each type.

Syntax of foreach loop

```
foreach (element in iterable-item)
{
    // Body of foreach loop
}
```

- From above, iterable-item can be an array or a class of collection.

How foreach loop works?



- The `in` keyword used along with **foreach** loop is used to iterate over the `iterable-item`. The **in** keyword selects an item from the `iterable-item` on each iteration and store it in the variable `element`.
- On first iteration, the first item of `iterable-item` is stored in element. On second iteration, the second element is selected and so on.
- The number of times the foreach loop will execute **is equal to the number of elements in the array or collection**.

Printing array using for loop

```
using System;

namespace Loop
{
    class ForLoop
    {
        public static void Main(string[] args)
        {
            char[] my_array = {'H','e','l','l','o'};

            for(int i = 0; i < my_array.Length; i++)
                Console.WriteLine(my_array[i]);
        }
    }
}
```

- The same task can be done more easily using a **foreach** loop.

Printing an array using foreach loop

```
using System;

namespace Loop
{
    class ForEachLoop
    {
        public static void Main(string[] args)
        {
            char[] my_array = {'H','e','l','l','o'};
             foreach(char ch in my_array) {            {
                Console.WriteLine(ch);
            }
        }
    }
}
```

- When we run both programs, the output will be the same:

```
H
e
l
l
o
```

- In the above program, the **foreach** loop iterates over the array, `myArray`.
- On first iteration, the first element i.e., `myArray[0]` is selected and stored in `ch`.
- Similarly on the last iteration, the last element i.e., `myArray[4]` is selected.
- Inside the body of loop, the value of `ch` is printed.
- When we look at both programs, the program that uses **foreach** loop is **more readable and easier to understand**. The syntax is more expressive for a group of similar data.
- **A foreach loop is preferred over a for loop when working with arrays and collections.**

Traversing an array of gender using foreach loop

```
using System;

namespace Loop
{
    class ForEachLoop
    {
        public static void Main(string[] args)
        {
            char[] gender = {'m','f','m','m','m','f','f','m','m','f'};
            int male = 0, female = 0;
            foreach (char g in gender) {
                if (g == 'm')
                   male++;
                else if (g =='f')
                   female++;
            }
            Console.WriteLine("Number of male = {0}", male);
            Console.WriteLine("Number of female = {0}", female);
        }
    }
}
```

When we run the program, the output will be:

```
Number of male = 6
Number of female = 4
```

A foreach loop with List (Collection)

```
using System;
using System.Collections.Generic;
namespace Loop {
  class ForEachLoop {
    public static void Main(string[] args) {
      var numbers = new List<int>() { 5, -8, 3, 14, 9, 17, 0, 4 };
      int sum = 0;
      foreach (int number in numbers) {
        sum += number;
      }
      Console.WriteLine("Sum = {0}", sum);
      Console.ReadLine();
    }
  }
}
```

- When we run the program, the output will be:

```
Sum = 44
```

- In this program, **foreach** loop is used to traverse through a **collection**.
- Traversing a collection is similar to traversing through an array.
- The first element of collection is selected on the first iteration, second element on second iteration and so on till the last element.

C# ArrayList – Non-Generic

- In C#, an `ArrayList` stores elements of **multiple data types** whose size can be changed dynamically.

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        ArrayList student = new ArrayList();

        // Add elements to ArrayList
        student.Add("Jackson");
        student.Add(5);

        // display every element of myList
        for (int i = 0; i < student.Count; i++)
        {
            Console.WriteLine(student[i]);
        }
    }
}
```

Output

```
Jackson
5
```

- Here, `student` is an `ArrayList` that contains elements (`"Jackson"` and **5**) of different data types.
- We will learn about `ArrayList` in detail.

Create an ArrayList

- To create `ArrayList` in C#, we need to use the `System.Collections` namespace.

```
// Create an ArrayList
ArrayList myList = new ArrayList();
```

- Here, we have created an ArrayList named `myList`.

Basic Operations on ArrayList

- In C#, we can perform different operations on **ArrayList**. Some commonly used **ArrayList** operations in this tutorial:
  - Add Elements
  - Access Elements
  - Change Elements
  - Remove Elements

Add Elements in ArrayList

- C# provides a method `Add()` using which we can add elements in `ArrayList`. For example,

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        ArrayList student = new ArrayList();

        // Add elements to ArrayList
        student.Add("Tina");
        student.Add(5);
    }
}
```

- In the above example, we have created an **ArrayList** named student.
- Then we added "**Tina**" and **5** to the **ArrayList** using the **Add()** method.

Note: ArrayList stores elements with different data types. However, if you want to store elements of the same data type use List<T> class instead.

Other way to add Elements to ArrayList

- Add Elements in an ArrayList using Object Initializer Syntax
- Add Elements in an ArrayList at specified index

Access ArrayList Elements

- We use indexes to access elements in ArrayList. The indexing starts from 0. For example,

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        ArrayList school_details = new ArrayList();
        school_details.Add("Mary's");
        school_details.Add("France");
        school_details.Add(23);

        // Access the first element
        Console.WriteLine("First element: " + school_details[0]);

        // Access the second element
        Console.WriteLine("Second element: " + school_details[1]);
    }
}
```

Output

```
First element: Mary's
Second element: France
```

- Since the index of the `ArrayList` starts from **0**:
  - `schoolDetails[0]` - accesses the first element
  - `schoolDetails[1]` - accesses the second element

Iterate ArrayList

- We can also, as previously with a List collection, loop through each element of ArrayList using a for loop.

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList containing 3 elements
        ArrayList myList = new ArrayList();

        my_list.Add("Science");
        my_list.Add(true);
        my_list.Add(5);

        // Display every element of myList
        for (int i = 0; i < my_list.Count; i++)
        {
            Console.WriteLine(myList[i]);
        }

    }
}
```

Output

```
Science
True
5
```

- In the above example, we have looped through `my_list` using a `for` loop.
- Here, `my_list.Count` gives the number of elements in `my_list`.

Change ArrayList Elements

- We can change the value of elements in `ArrayList` as:

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        ArrayList myList = new ArrayList();

        my_list.Add("Harry");
        my_list.Add("Miller");

        Console.WriteLine("Original Second element: " + my_list[1]);

        // Change the value of second element
        my_list[1] = "Styles";

        Console.WriteLine("Updated second element: " + my_list[1]);
    }
}
```

Output

```
Original Second element: Miller
Updated second element: Styles
```

- Here, we have changed the value of the second element in `my_list`.

Remove ArrayList Elements

- C# provides methods like `Remove()`, `RemoveAt()`, `RemoveRange()` to remove elements from `ArrayList`.
- We will see an example below using `Remove()` to a remove element:

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        ArrayList myList = new ArrayList();
        my_List.Add("Jack");
        my_list.Add(4);
        my_list.Add("Jimmy");

        // Remove "Jack" from myList
        my_list.Remove("Jack");

        // Iterate through myList after removing "Jack" using foreach
        foreach (int i in my_list)
        {
            Console.WriteLine(i);
        }
    }
}
```

Output

```
4
Jimmy
```

- In the above example, we have removed `"Jack"` from `my_list` using the `Remove()` method.
- We also used a **foreach** loop to iterate through each element

How can you check whether an element is inside ArrayList?

- C# provides a method `Contains()` using which we can determine whether an element is inside `ArrayList`. For example,

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // Create an ArrayList
        var my_list = new ArrayList() { "Small", "Medium", "Large", 3 };

        // Check whether myList contains "Willow"
        var result = my_list.Contains("Medium");

        Console.WriteLine(result);
    }
}
```

Output

```
True
```

- Here, **myList** contains **"Willow"** so **my_list.Contains("Willow")** returns **True**.

Week 10 – Theory

Midterm

Week 10 – Lab

Review of Midterm and work on final project

Week 11 – Theory

Continuation of Collections: Dictionary

C# Dictionary

- A Dictionary<TKey, TValue> is a **generic collection** that consists of elements as **key/value** pairs that are not sorted in an order. For example,

```
Dictionary<int, string> country = new Dictionary<int, string>();
```

- Here, `country` is a dictionary that contains `int` type keys and `string` type values.

Create a Dictionary

To create a dictionary in C#, we need to use the `System.Collections.Generic` namespace. Here is how we can create a dictionary in C#.

```
// create a dictionary
```

- **Dictionary<dataType1, dataType2> dictionary_name = new Dictionary<dataType1, dataType2>();**
- Dictionary_name - **name of the dictionary**
- dataType1 - **datatype of keys**
- dataType2 - **datatype of values**

Create a Dictionary

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // Create a dictionary
        Dictionary<int, string> country = new Dictionary<int, string>();
        // Add items to dictionary
        country.Add(5, "France");
        country.Add(3, "Canada");
        country.Add(4, "USA");

        // Print value having key is 3
        Console.WriteLine("Value having key 3: " + country[3]);
    }
}
```

Output

```
Value having key 3: Canada
```

- In the above example, we have created a dictionary named `country`.
- The keys are of `int` type and values are of `string` type.

Basic Operations on Dictionary

- In C#, we can perform different operations on a dictionary.
- We will look at some commonly used Dictionary<TKey, TValue> operations:
  - Add Elements
  - Access Elements
  - Change Elements
  - Remove Elements

Add Elements in Dictionary

- C# provides the `Add()` method using which we can add elements in the dictionary.

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // Create a dictionary
        Dictionary<string, string> my_songs = new Dictionary<string, string>();

        // add items to dictionary
        my_songs.Add("Queen", "Break Free");
        my_songs.Add("Free", "All right now");
        my_songs.Add("Pink Floyd", "The Wall");



    }
}
```

- In the above example, we have created a `Dictionary<TKey, TValue>` named `my_songs`.
- Here we have added key/value pairs using the `Add()` method where,
  - **keys** - `"Queen"`, `"Free"` and `"Pink Floyd"`
  - **values** - `"Break Free"`, `"All right now"` and `"The Wall"`

Another way to add Elements to Dictionary

Add Elements in a dictionary without using `Add()` method

Access Dictionary Elements

- We can access the elements inside the dictionary using it's keys. For example,

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // create a dictionary
        Dictionary<string, string> student = new Dictionary<string, string>();

        // add items to dictionary
        student.Add("Name", "Susan");
        student.Add("Faculty", "History");

        // access the value having key "Name"
        Console.WriteLine(student["Name"]);


        // access the value having key "Faculty"
        Console.WriteLine(student["Faculty"]);

    }
}
```

Output

```
Susan
History
```

- In the above example, we have accessed the values of the dictionary using their keys:
    - `student["Name"]` - accesses the value whose key is `"Name"`
    - `student["Faculty"]` - accesses the value whose key is `"Faculty"`

Iterate through Dictionary

- In C#, we can also loop through each element of the dictionary using a `foreach` loop. For example,

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // Create a dictionary
        Dictionary<string, string> car = new Dictionary<string, string>();

        // Add items to dictionary
        car.Add("Model", "Hyundai");
        car.Add("Price", "36K");

        // Iterate through the car dictionary
        foreach (KeyValuePair<string, string> items in car)
        {
            Console.WriteLine("{0} : {1}", items.Key, items.Value);
        }

    }
}
```

Output

```
Model : Hyundai
Price : 36K
```

- In the above example, we have looped through `car` using a `foreach` loop.
- Here, the `Key` and `Value` property returns a collection containing keys and values in the dictionary.

Change Dictionary Elements

- We can change the value of elements in dictionary as:

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // Create a dictionary
        Dictionary<string, string> car = new Dictionary<string, string>();

        // Add items to dictionary
        car.Add("Model", "Hyundai");
        car.Add("Price", "36K");

        // Print the original value
        Console.WriteLine("Value of Model before changing: " + car["Model"]);

        // Change the value of "Model" key to "BMW"
        car["Model"] = "BMW";

        // Print new updated value of "Model"
        Console.WriteLine("Value of Model after changing: " + car["Model"]);
    }
}
```

Output

```
Value of Model before changing: Hyundai
Value of Model after changing: BMW
```

- Here, we have changed the value of the `"Model"` key in the `car` dictionary.

Remove Dictionary Elements

- To remove the elements inside the dictionary we use:
  - `Remove()` - removes the key/value pair from the dictionary.

```
using System;
using System.Collections;
class Program
{
    public static void Main()
    {
        // Create a dictionary
        Dictionary<string, string> employee = new Dictionary<string, string>();

        // Add items to dictionary
        employee.Add("Name", "Marry");
        employee.Add("Role", "Manager");
        employee.Add("Address", "California");

        Console.WriteLine("Original Dictionary :");

        // Iterate through the modified dictionary
        foreach (KeyValuePair<string, string> items in employee)
        {
            Console.WriteLine("{0} : {1}", items.Key, items.Value);
        }

        // remove value with key "Role"
        employee.Remove("Role");

        Console.WriteLine("\nModified Dictionary :");

        // Iterate through the modified dictionary
        foreach (KeyValuePair<string, string> items in employee)
        {
            Console.WriteLine("{0} : {1}", items.Key, items.Value);
        }
    }
}
```

Output

```
Original Dictionary :
Name : Marry
Role : Manager
Address : California

Modified Dictionary :
Name : Marry
Address : California
```

- In the above example, we have removed the element whose key is "**Role**".
- Here, **employee.Remove("Role")** removes the key/value pair "Role" : "Manager" from the employee dictionary.
- So, when we iterate through employee we get a modified dictionary.

Note: If you want to remove all the elements of the dictionary, use the Clear() method.

Another way to create a Dictionary

- We can also create a dictionary using `var` keyword. For example:

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main()
    {
        // Create a dictionary named fruits
        var fruits = new Dictionary<int, string>() {
            {1, "Apple"},
            {2, "Orange"},
        };
        // Print value having key 2
        Console.WriteLine(fruits[2]);
    }
}
```

Output

```
Orange
```

Further reading: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-7.0

C# Dictionary and KeyValuePair.

- While the C# Dictionary collection has tremendous benefits, it also has restrictions due to their structure and purpose.
- As such, it is best to convert a Dictionary to a List collection using the KeyValuePair structure.
- Doing so allows you to use the important methods available in a List Collection such as **Sort** that is not available in a Dictionary.
- The following also highlights some key differences:

List<KeyValuePair>

- Lighter
- Insertion and iteration is faster using a List.
- Searching is slower than Dictionary.
- This can be serialized to XMLSerializer.
- Changing the key, value is not possible.
- **KeyValuePair can be assigned value only during creation.**
- If you want to change then remove and add new item in same place.

Dictionary<T Key, T Value>

- Heavy
- **Insertion is slower. Has to compute Hash.**
- Searching is faster because of Hash.
- **Can't be serialized**. Custom code is required.
- You can change and update dictionary.

- Various updates to the C# language and .NET Framework do provide alternatives such as **SortedDictionary** or by using the **System.Linq** library you can “query” data inside a Dictionary. Example:

```
Dictionary<string, int> dict = new Dictionary<string, int>() {
    {"A", 15}, {"B", 10}, {"C", 12}, {"D", 20}, {"E", 14}
};

List<KeyValuePair<string, int>> mappings = dict.ToList();

mappings.Sort((x, y) => x.Value.CompareTo(y.Value));
```

Constructor and base() keyword.

- One page 68, we saw how to use the **this** with a constructor to call another constructor in the same class for initialization to reduce code.
- The following code segment demonstrates how it works

```
public Test(bool a, int b, string c) : this(a, b)
{
    this.m_C = c;
}

public Test(bool a, int b, float d) : this(a, b)
{
    this.m_D = d;
}

private Test(bool a, int b)
{
    this.m_A = a;
    this.m_B = b;
}
```

- When any of the overloaded constructers is called, the constructor with matching parameters in the this keyword is called first to do the initialization. Parameters **a** and **b** are passed using **this**.
- Using constructor chaining so that you have only one constructor calling the base constructor will eliminate duplicate code.
- Similar to this(), using base() invokes a base class constructor first.
- It could be parameterized or parameter less as follows:

```
public class MyBaseClass
{
    // Invoke the parameterless constructor in object
    public MyBaseClass(int x) : base()
    {
        Console.WriteLine("In the base class constructor taking an int, which is " + x);
    }
}

public class MyDerivedClass : MyBaseClass
{
    // Invoke the MyDerivedClass constructor taking an int
    public MyDerivedClass() : this(5)
    {
        Console.WriteLine("In the derived class parameterless constructor.");
    }

    // Invoke the MyBaseClass constructor taking an int
    public MyDerivedClass(int y) : base(y + 1)
    {
        Console.WriteLine("In the derived class constructor taking an int parameter.");
    }

    // Invoke the MyBaseClass constructor taking an int
    public MyDerivedClass(string x) : base(10)
    {
        Console.WriteLine("In the derived class constructor taking a string parameter.");
    }
}
```

- There must always be a "chain" of constructors which runs constructors all the way up the class hierarchy.
- Every class in the hierarchy will have a constructor invoked, although some of those constructors may not explicitly appear in the code. (See the section on default constructors, later.)
- The parameters (if any) within the brackets of base(...) or this(...) are passed as the parameters to the invoked constructors.
- They can be the parameters given in the constructor declaration, but don't have to be.
- With the code above, if we call **new MyDerivedClass();** It would start a chain as follows:
    - The MyDerivedClass parameterless constructor get called first
    - Which would in turn invoke the MyDerivedClass constructor
    - Which takes an int parameter (with 5 as that parameter value)
    - Which would in turn invoke the MyBaseClass constructor
    - Which takes an int parameter (with 6 as that parameter value).
    - Note that the specified constructor is run before the constructor body, so the result of **new MyDerivedClass()** is:

```
In the base class constructor taking an int, which is 6
In the derived class constructor taking an int parameter.
In the derived class parameterless constructor.
```

Further reading: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/base

C# abstract class and method

Abstract Class

- In C#, abstract classes have a similar function to Polymorphism but with some distinct differences.
- First, we use the **abstract** keyword to create an abstract class. Example:

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

- Second, we **cannot** create objects using an abstract class. More on this later.
- An abstract class can have both abstract methods (method without body) and non-abstract methods (method with the body). Example:

```
abstract class Language {

  // Abstract method
  public abstract void display1();

  // Non-abstract method
  public void display2() {
    Console.WriteLine("Non abstract method");
  }
}
```

Inheriting Abstract Class

- As we cannot create objects of an abstract class, we must create **a derived class from it** so that we can access members of the abstract class using the object of the derived class.

```
using System;
namespace AbstractClass {

  abstract class Language {

    // Non-abstract method
    public void display() {
      Console.WriteLine("Non abstract method");
    }
  }

  // Inheriting from abstract class
  class Program : Language {

    static void Main (string [] args) {

      // Object of Program class
      Program obj = new Program();

      // Access method of an abstract class
      obj.display();
      Console.ReadKey();
    }
  }
}
```

Output

```
Non abstract method
```

- In the above example, we have created an abstract class named `Language`.
- The class contains a non-abstract method `display()`.
- We have created the `Program` class that inherits the abstract class. Notice the statement:

```
obj.display();
```

- Here, `obj` is the object of the derived class `Program`. We are calling the method of the abstract class using the object `obj`.

Note: We can use abstract class only as a base class.

C# Abstract Method

- A method that does not have a body is known as an abstract method. We use the `abstract` keyword to create abstract methods. Example:

```
public abstract void display();
```

- Here, `display()` is an abstract method.
- An abstract method can only be present inside an abstract class.
- When a non-abstract class inherits an abstract class, it should provide an implementation of the abstract methods.

Implementation of the abstract method

```
using System;
namespace AbstractClass {

  abstract class Animal {
    // Abstract method
    public abstract void MakeSound();
  }

  // Inheriting from abstract class
  class Dog : Animal {
    // Provide implementation of abstract method
    public override void MakeSound() {
      Console.WriteLine("Bark Bark");
    }
  }

  class Program  {
    static void Main (string [] args) {
      // Create an object of Dog class
      Dog obj = new Dog();
      obj.MakeSound();
      Console.ReadKey();
    }
  }
}
```

Output

```
Bark Bark
```

- In the above example, we have created an abstract class named `Animal`. We have an abstract method `MakeSound()` inside the class.
- We have a `Dog` class that inherits from the `Animal` class. `Dog` class provides the implementation of the abstract method `MakeSound()`.

```
// provide implementation of abstract method
public override void MakeSound() {

  Console.WriteLine("Bark Bark");

}
```

- Notice, we have used `override` with the `MakeSound()` method.
- This indicates the method is overriding the method from the base class.
- We then used the object of the `Dog` class to access `MakeSound()`.
- If the `Dog` class had not provided the implementation of the abstract method `MakeSound()`, `Dog` class should have been marked abstract as well.

Note: Unlike the C# inheritance, we cannot use virtual with the abstract methods of the base class. This is because an abstract class is implicitly virtual.

Abstract class with get and set accessors

- We can mark **get** and **set** accessors as abstract. Example:

```
using System;
namespace AbstractClass {
  abstract class Animal {
    protected string name;
    // Abstract method
    public abstract string Name {
      get; set;
    }
  }

  // Inheriting from abstract class
  class Dog : Animal {
    // provide implementation of abstract method
    public override string Name {
      get { return name; }
      set { name = value;  }
    }
  }
  class Program  {
    static void Main (string [] args) {
      // create an object of Dog class
      Dog obj = new Dog();
      obj.Name = "Tom";
      Console.WriteLine("Name: " + obj.Name);
      Console.ReadKey();
    }
  }
}
```

Output

```
Name: Tom
```

- In the above example, we have marked the get and set accessor as abstract.

```
obj.Name = "Tom";
Console.WriteLine("Name: " + obj.Name);
```

- We are setting and getting the value of the `name` field of the abstract class `Animal` using the object of the derived class `Dog`.

Access Constructor of Abstract Classes

- An abstract class can have constructors as well. Example:

```
using System;
namespace AbstractClass {
  abstract class Animal {

    public Animal() {
      Console.WriteLine("Animal Constructor");
    }
  }

  class Dog : Animal {
    public Dog() {
      Console.WriteLine("Dog Constructor");
    }
  }

  class Program  {
    static void Main (string [] args) {
      // Create an object of Dog class
      Dog d1 = new Dog();
      Console.ReadKey();
    }
  }
}
```

Output

```
Animal Constructor
Dog Constructor
```

- In the above example, we have created a constructor inside the abstract class `Animal`.

```
Dog d1 = new Dog();
```

- Here, when we create an object of the derived class `Dog` the constructor of the abstract class `Animal` gets called as well.

Note: We can also use destructors inside the abstract class.

C# Abstraction

- Abstract classes are used to achieve abstraction in C#.
- It allows us to hide unnecessary details and only show the needed information.
- This helps us to manage complexity by hiding details with a simpler, higher-level idea.
- A practical example of abstraction can be motorbike brakes. We know what a brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us.
- The major advantage of hiding the working of the brake is that now the manufacturer can implement brakes differently for different motorbikes. However, what brake does will be the same.

Example: C# Abstraction

```
using System;

namespace AbstractClass {
  abstract class MotorBike {
    public abstract void brake();
  }

  class SportsBike : MotorBike {
    // Provide implementation of abstract method
    public override void brake() {
      Console.WriteLine("Sports Bike Brake");
    }
  }

  class MountainBike : MotorBike {
    // Provide implementation of abstract method
    public override void brake() {
      Console.WriteLine("Mountain Bike Brake");
    }
  }

  class Program  {
    static void Main (string [] args) {
      // create an object of SportsBike class
      SportsBike s1 = new SportsBike();
      s1.brake();

      // create an object of MountainBike class
      MountainBike m1 = new MountainBike();
      m1.brake();

      Console.ReadLine();
    }
  }
}
```

Output

```
Sports Bike Brake
Mountain Bike Brake
```

- In the above example, we have created an abstract class `MotorBike`. It has an abstract method `brake()`.
- As `brake()` is an abstract method the implementation of `brake()` in `MotorBike` is kept hidden.
- Every motorbike has a different implementation of the brake. This is why `SportsBike` makes its own implementation of `brake()` and `MountainBike` makes its own implementation of `brake()`.

Note: We use interfaces to achieve complete abstraction in C#. To learn more, visit C# Interface.

Further reading: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract

C# interface

- In C#, an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract (method without body).
- We use the `interface` keyword to create an interface. Example:

```
interface IPolygon {

  // Method without body
  void calculateArea();
}
```

- `IPolygon` is the name of the interface.
- By convention, interface starts with `I` so that we can identify it just by seeing its name.
- We cannot use access modifiers inside an interface.
- All members of an interface are public by default.
- An interface doesn't allow fields.

Implementing an Interface

- **We cannot create objects of an interface**. To use an interface, other classes must **implement** it. Same as in C# Inheritance, we use `:` symbol to implement an interface.

```
using System;
namespace CsharpInterface {

  interface IPolygon {
    // Method without body
    void CalculateArea(int l, int b);
  }

  class Rectangle : IPolygon {
    // Implementation of methods inside interface
    public void CalculateArea(int l, int b) {

      int area = l * b;
      Console.WriteLine("Area of Rectangle: " + area);
    }
  }

  class Program {
    static void Main (string [] args) {
      Rectangle r1 = new Rectangle();
      r1.calculateArea(100, 200);
       Console.ReadKey();
    }
  }
}
```

Output

```
Area of Rectangle: 20000
```

- In the above example, we have created an interface named `IPolygon`. The interface contains a method `calculateArea(int a, int b)` without implementation.
- Here, the `Rectangle` class implements `IPolygon` and provides the implementation of the `calculateArea(int a, int b)` method.

Note: We must provide the implementation of all the methods of interface inside the class that implements it.

Implementing Multiple Interfaces

- Unlike inheritance, a class can implement multiple interfaces. For example,

```
using System;
namespace CsharpInterface {

  interface IPolygon {
    // method without body
    void CalculateArea(int a, int b);
  }

  interface IColor {
    void GetColor();
  }

  // Implements two interface
  class Rectangle : IPolygon, IColor {


    // Implementation of IPolygon interface
    public void CalculateArea(int a, int b) {
      int area = a * b;
      Console.WriteLine("Area of Rectangle: " + area);
    }

    // Implementation of IColor interface
    public void GetColor() {
      Console.WriteLine("Red Rectangle");
    }
  }

  class Program {
    static void Main (string [] args) {

      Rectangle r1 = new Rectangle();

      r1.calculateArea(100, 200);
      r1.getColor();
      Console.ReadKey();
    }
  }
}
```

Output

```
Area of Rectangle: 20000
Red Rectangle
```

- In the above example, we have two interfaces, `IPolygon` and `IColor`.

```
class Rectangle : IPolygon, IColor {
  …
}
```

- We have implemented both interfaces in the Rectangle class separated by a comma.
- Rectangle has to implement the method of both interfaces.

Using reference variable of an interface

- We can use the reference variable of an interface. Example,

```
using System;
namespace CsharpInterface {

  interface IPolygon {
    // Method without body
    void CalculateArea(int l, int b);
  }

  class Rectangle : IPolygon {
    // Implementation of methods inside interface
    public void CalculateArea(int l, int b) {

      int area = l * b;
      Console.WriteLine("Area of Rectangle: " + area);
    }
  }

  class Program {
    static void Main (string [] args) {

      // using reference variable of interface
      IPolygon r1 = new Rectangle();
      r1.calculateArea(100, 200);
    }
  }
}
```

Output

```
Area of Rectangle: 20000
```

- In the above example, we have created an interface named `IPolygon`. The interface contains a method `calculateArea(int l, int b)` without implementation.

```
IPolygon r1 = new Rectangle();
```

- Notice, we have used the reference variable of interface `IPolygon`. It points to the class `Rectangle` that implements it.
- Though we cannot create objects of an interface, we can still use the **reference variable** of the interface that points to its implemented class.

Practical Example of Interface

```
using System;
namespace CsharpInterface {

  interface IPolygon {
    // method without body
    void CalculateArea();
  }

  // Implements interface
  class Rectangle : IPolygon {

    // Implementation of IPolygon interface
    public void CalculateArea() {
      int l = 30;
      int b = 90;
      int area = l * b;
      Console.WriteLine("Area of Rectangle: " + area);
    }
  }

  class Square : IPolygon {

    // Implementation of IPolygon interface
    public void CalculateArea() {
      int l = 30;
      int area = l * l;
      Console.WriteLine("Area of Square: " + area);
    }
  }

  class Program {
    static void Main (string [] args) {

      Rectangle r1 = new Rectangle();
      r1.calculateArea();
      Square s1 = new Square();
      s1.calculateArea();
    }
  }
}
```

Output

```
Area of Rectangle: 2700
Area of Square: 900
```

- In the above program, we have created an interface named `IPolygon`. It has an abstract method `calculateArea()`.
- We have two classes `Square` and `Rectangle` that implement the `IPolygon` interface.
- The rule for calculating the area is different for each polygon. Hence, `calculateArea()` is included without implementation.
- Any class that implements `IPolygon` must provide an implementation of `calculateArea()`. Hence, implementation of the method in class `Rectangle` is independent of the method in class `Square`.

Advantages of C# interface

- Here, the method `calculateArea()` inside the interface, does not have a body. Thus, it hides the implementation details of the method.
- Interfaces provide specifications that a class (which implements it) must follow.
- In our previous example, we have used `calculateArea()` as a specification inside the interface `IPolygon`. This is like setting a rule that we should calculate the area of every polygon.
- Now any class that implements the `IPolygon` interface must provide an implementation for the `calculateArea()` method.
- Interfaces are used to achieve multiple inheritance in C#.
- Interfaces provide **loose coupling** (having no or least effect on other parts of code when we change one part of a code).
- In our previous example, if we change the implementation of `calculateArea()` in the `Square` class it does not affect the `Rectangle` class.

Further reading: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces

From W3School (active link to site):

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default `abstract` and `public`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

How to choose between Polymorphism, Abstract Classes, and Interfaces?

The choice between polymorphism, abstract classes and interfaces depends on the specific requirements of the project and requires experience with Object Oriented Analysis and an understanding of Design Patterns. For now, just understand how to implement them. Here are some general guidelines:

Use **polymorphism** when:

- When you have a group of related classes that share common behavior but have different implementations.
- When you want to allow for more flexibility and extensibility in the code.
- When you want to implement design patterns such as the **Strategy pattern**, the **Template Method pattern**, or the **Factory Method pattern**.
- When you want to use **dependency injection**.

Use **abstract classes** when:

- You want to provide a common base implementation for a group of related classes.
- You want to **share code among multiple subclasses**.
- You want to define non-abstract methods or fields that are common to all subclasses.
- You want to define constructors that **enforce certain initialization** steps.

Use **interfaces** when:

- You want to define a common interface but not an implementation.
- You want to enforce a contract that certain methods, properties, or events must be implemented by a class.
- You want to provide a way for **unrelated classes to share a common interface**.

- In general, it's a good practice to **start with an interface** and only use an abstract class when you need to provide a base implementation or when you want to share code among multiple subclasses.
- **This allows for more flexibility and allows different classes to implement the interface in different ways**.
- However, if you need to define **a common base implementation**, an **abstract class may be a better choice.**
- Polymorphism is useful when you have **a group of related classes that share common behavior** but have different implementations.
- In practice, it's common to use a combination of these concepts to achieve the desired functionality in your code.

Week 11 – Lab

- Practice Abstract classes and Interfaces.
- Work on Final project.

Week 12 – Theory

Lambdas

Delegates