

# Object Oriented Programming

---



# CHAPTER 1

## Introduction to Object Oriented Programming



# Programming paradigm



- A programming paradigm is a set of concepts, ideas, and practices that programmers use to design and implement computer programs.
  - **Imperative programming** is the most common paradigm, and it is based on the idea of giving instructions to the computer. In imperative programming, the programmer tells the computer what to do, step by step.  
C, C++, Java, Python
  - **Procedural programming** is a subset of imperative programming that breaks down programs into smaller units called procedures. Procedures are reusable blocks of code that can be called from different parts of the program.  
Pascal, Fortran
  - **Object-oriented programming (OOP)** is a paradigm that views software as a collection of objects that interact with each other. Objects have data and methods, and they can be used to model real-world entities.  
C#, Java, C++, Python, Ruby
  - **Functional programming** is a paradigm that emphasizes the use of functions to represent computation. Functions are pure, meaning that they do not have any side effects. This makes functional programs easier to understand and debug.  
F#, Haskell, Lisp, Scala
  - **Logic programming** is a paradigm that is based on formal logic. Logic programs are made up of facts and rules, and they can be used to solve problems by reasoning about the facts and rules.  
Prolog

# Introduction to OOPS



---

- Object-oriented programming (OOP) is a programming paradigm that treats software design concepts as objects, which are data structures consisting of data fields and methods together with their interactions.
- OOP is one of the most widely used programming paradigms in the world. It is used to create large and complex software systems, such as operating systems, web browsers, and video games.

# Advantages of OOPS

---





## CHAPTER 2

Introduction to C#

# Introduction to C#



- C# is a general-purpose, multi-paradigm programming language developed by Microsoft. It was first released in 2002 as part of the .NET Framework, and it has since become one of the most popular programming languages in the world.
- C# was designed by Anders Hejlsberg, who also designed the Turbo Pascal and Delphi programming languages. Hejlsberg wanted to create a language that was both powerful and easy to use, and he drew inspiration from a variety of other languages, including C++, Java, and Visual Basic.  
2000: C# is first announced by Anders Hejlsberg at the Professional Developers Conference.
- C# is a general-purpose, modern, object-oriented programming language developed by Microsoft.
- C# is a compiled language, which means that it is converted into machine code before it is executed.
- C# is a versatile language that can be used to develop a wide variety of applications, including web applications, desktop applications, and mobile applications. It is also a popular language for game development and scientific computing. C# is a compiled language, which means that it is converted into machine code before it is executed.
- C# is also a type-safe language, which means that the compiler can check for errors at compile time, rather than at runtime.
- Latest version of C# is 11.
- Latest .Net Framework 4.8.
- Latest .Net 7.

# Features of C#

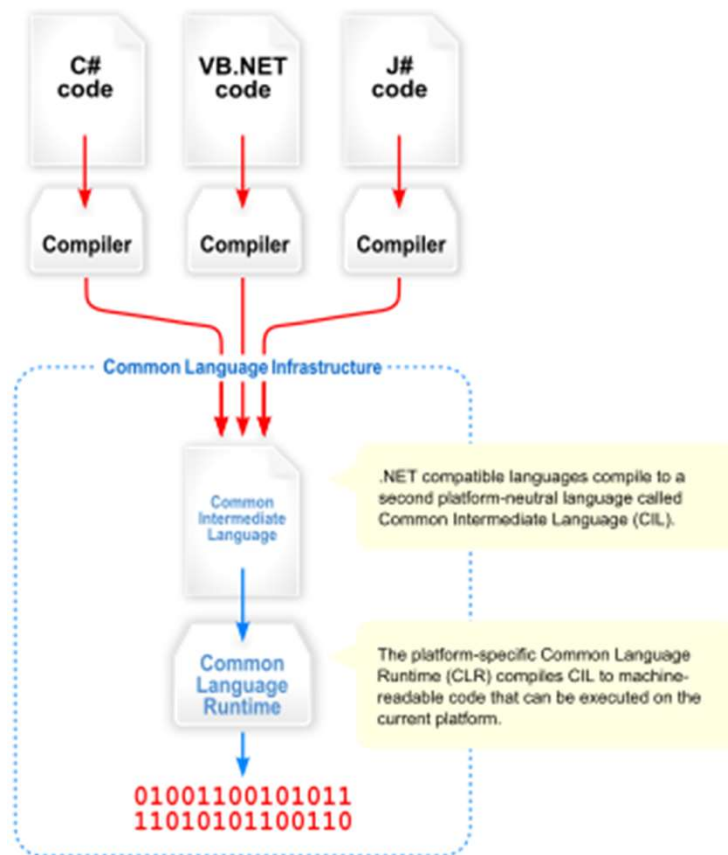


---

- Object-oriented: C# is an object-oriented language, which means that it uses objects to represent data and behavior. Objects are made up of classes, which are blueprints for creating objects.
- Compiled: C# is a compiled language, which means that it is converted into machine code before it is executed. This makes C# programs faster than interpreted languages, such as Python.
- Type-safe: C# is a type-safe language, which means that the compiler can check for errors at compile time, rather than at runtime. This makes C# programs more reliable.
- Secure: C# has a number of security features that help to protect applications from malicious code. These features include code access security, which allows developers to control which code can be executed in an application.
- Portable: C# code can be run on a variety of platforms, including Windows, macOS, and Linux. This makes C# a good choice for developing cross-platform applications.
- Extensible: C# is an extensible language, which means that developers can add new features and functionality to the language. This is done through the use of extensions, which are libraries of code that can be added to C# programs.



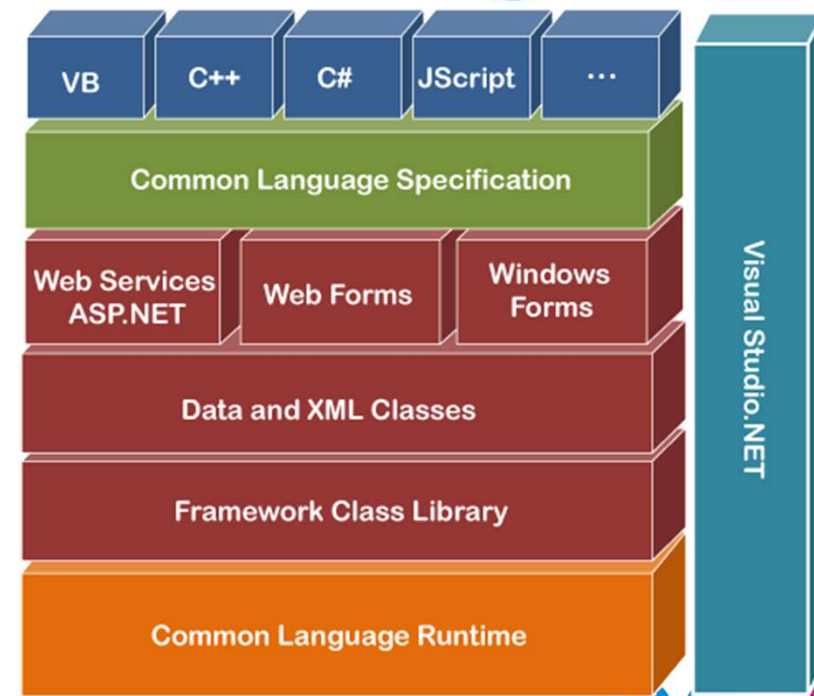
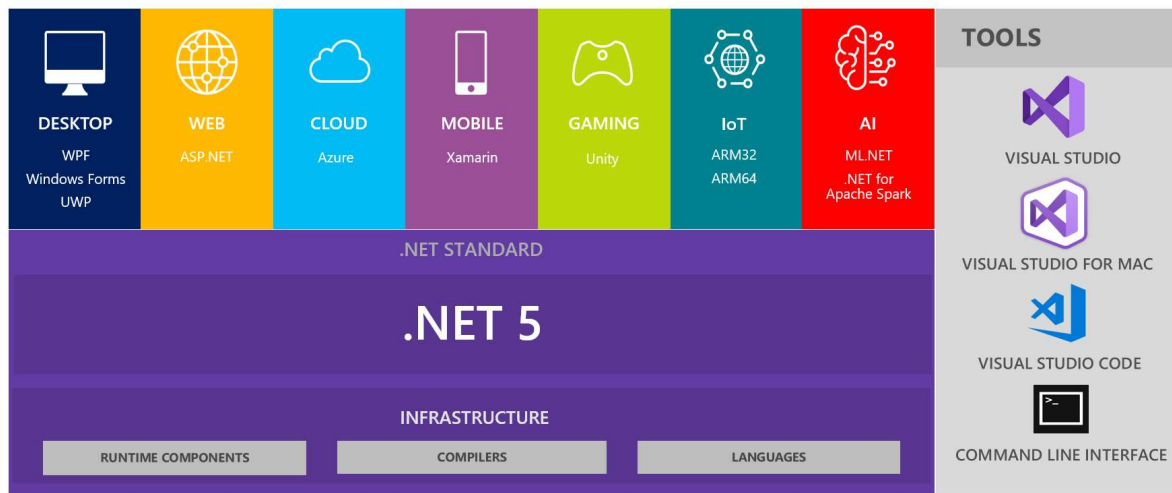
# C# Compilation Process



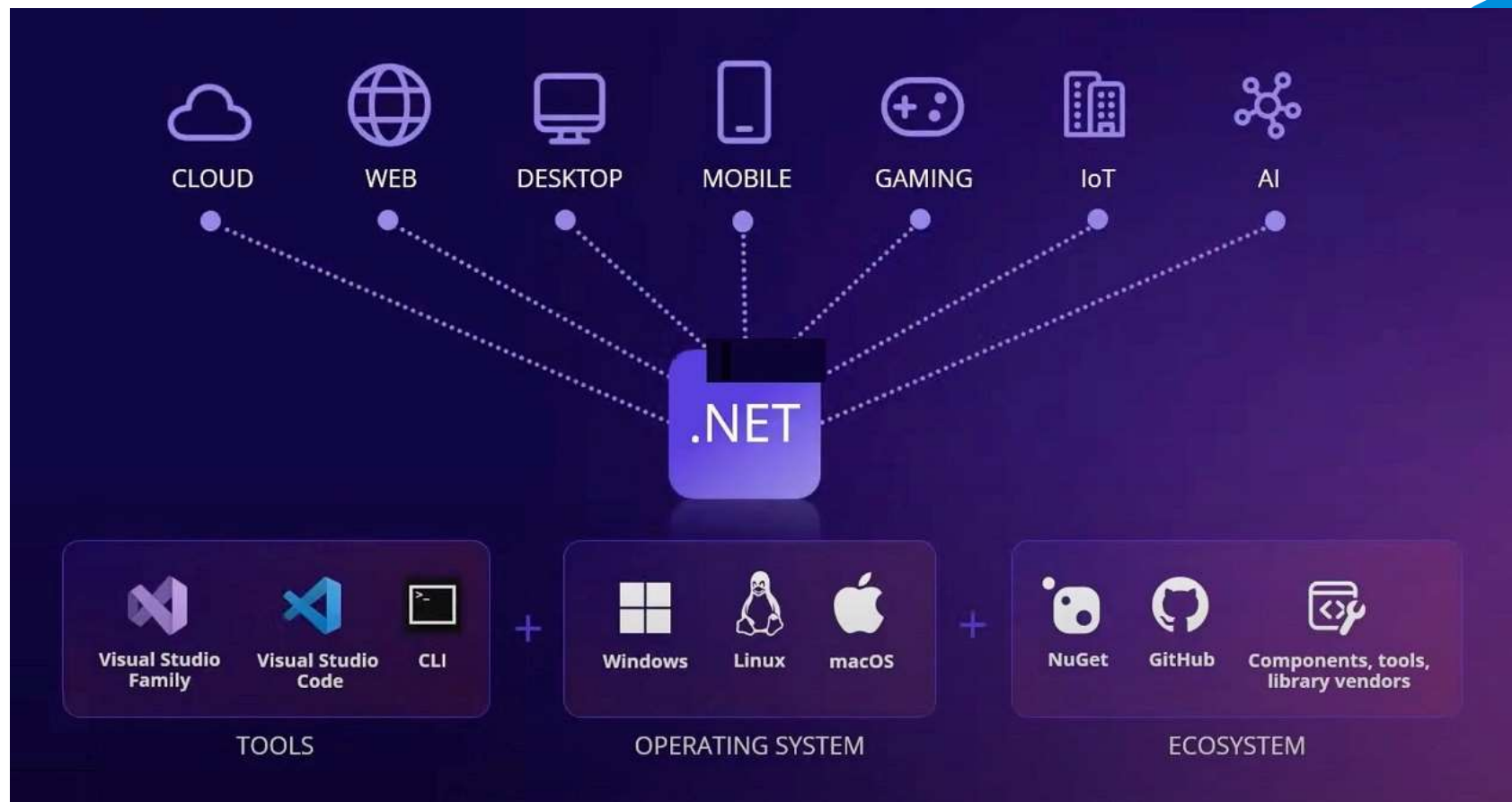
1. A developer writes a program in a .NET-supported language, such as C# or Visual Basic.
2. The program is compiled into an Intermediate Language (IL) file.
3. The IL file is then loaded into the Common Language Runtime (CLR).
4. The CLR executes the IL code, using a Just-in-Time (JIT) compiler to convert the IL code into native machine code.
5. The native machine code is then executed by the computer's processor.

# .NET and .NET Framework

.NET – A unified platform













# .NET and .NET Framework



# Popularity

- TIOBE Index – TIOBE

May 2023	May 2022	Change	Programming Language		Ratings	Change
1	1			Python	13.45%	+0.71%
2	2			C	13.35%	+1.76%
3	3			Java	12.22%	+1.22%
4	4			C++	11.96%	+3.13%
5	5			C#	7.43%	+1.04%
6	6			Visual Basic	3.84%	-2.02%
7	7			JavaScript	2.44%	+0.32%
8	10	▲		PHP	1.59%	+0.07%
9	9			SQL	1.48%	-0.39%
10	8	▼		Assembly language	1.20%	-0.72%

# CHAPTER 2

## Installation



Microsoft  
.NET

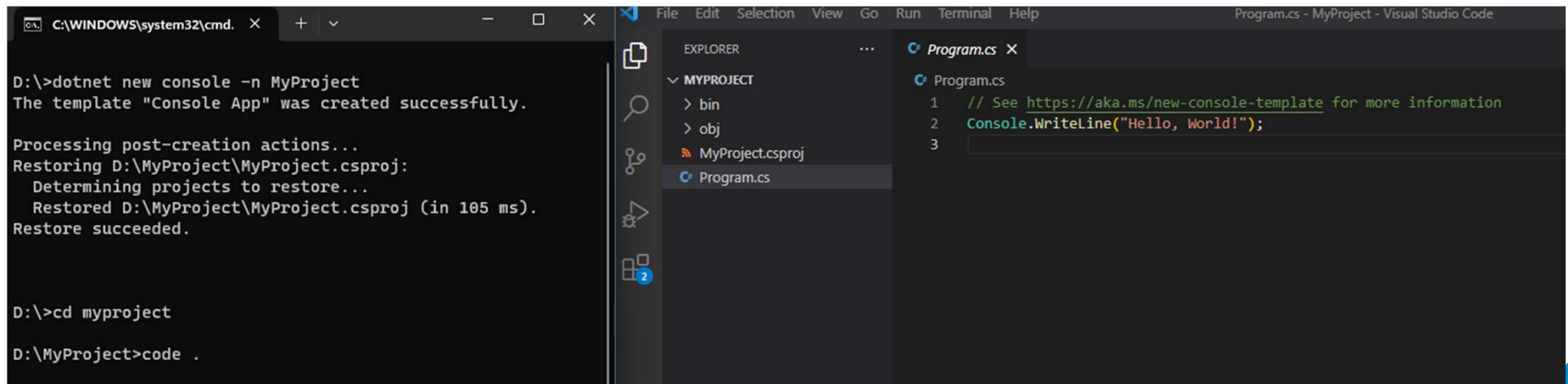
# Installation



- IDE - Visual Studio 2022 (Community Edition)  
<https://visualstudio.microsoft.com/>
- .NET Software Development Kit (SDK)
  - .NET Framework  
<https://dotnet.microsoft.com/en-us/download/dotnet-framework>
  - .NET  
<https://dotnet.microsoft.com/en-us/download>
- IDE - Visual Studio Code (Optional)
  - [Visual Studio Code - Code Editing. Redefined](#)
  - Extension - [C# - Visual Studio Marketplace](#)
- [C# Online Compiler | .NET Fiddle \(dotnetfiddle.net\)](#)

# Command Line Interface

- `dotnet --version`
- `dotnet new console -n MyProject`
- `dotnet run`



The screenshot displays the Visual Studio Code interface. On the left, a terminal window shows the following commands and output:

```
C:\WINDOWS\system32\cmd. x
D:\>dotnet new console -n MyProject
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring D:\MyProject\MyProject.csproj:
  Determining projects to restore...
  Restored D:\MyProject\MyProject.csproj (in 105 ms).
Restore succeeded.

D:\>cd myproject

D:\MyProject>code .
```

The Explorer pane in the center shows the project structure:

- MYPROJECT
  - bin
  - obj
  - MyProject.csproj
  - Program.cs

The right pane shows the content of `Program.cs`:

```
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
```

# Command Line Interface

- `dotnet new -h|--help`
- [dotnet new <TEMPLATE> - .NET CLI | Microsoft Learn](#)
- [.NET CLI | Microsoft Learn](#)

```
Command Prompt
D:\>dotnet new
The 'dotnet new' command creates a .NET project based on a template.

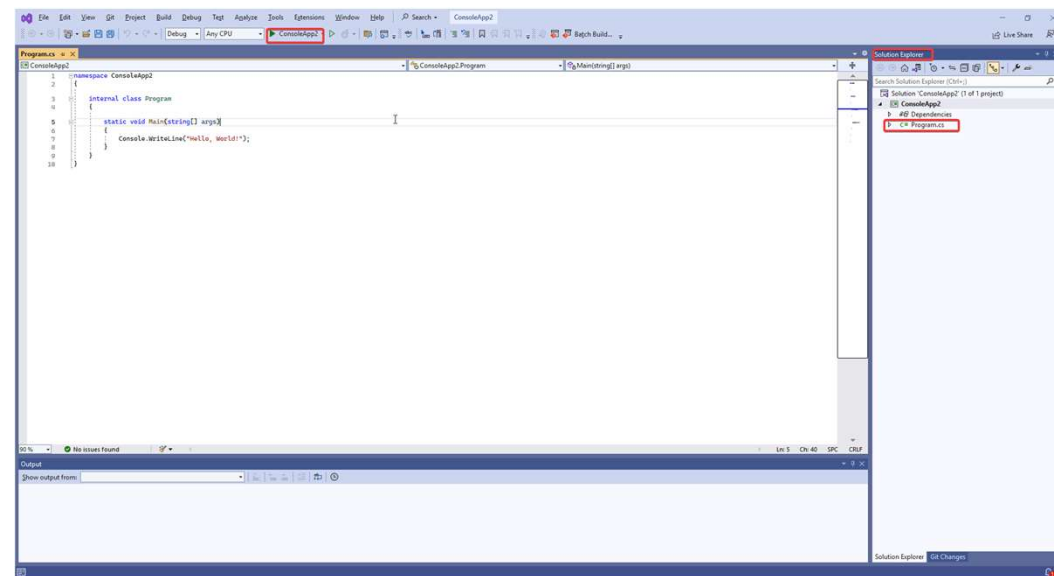
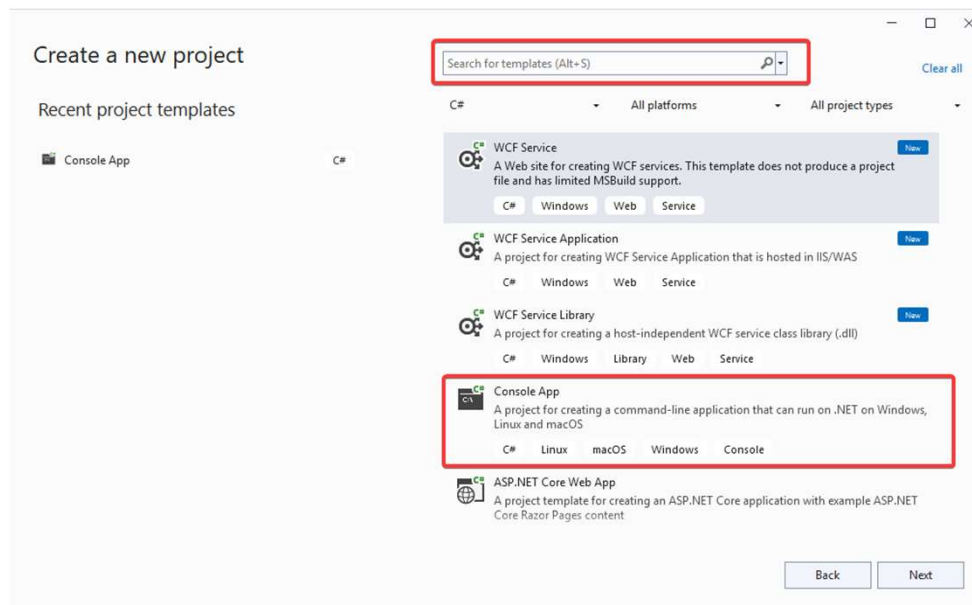
Common templates are:
Template Name      Short Name      Language      Tags
-----
ASP.NET Core Web App  webapp,razor    [C#]          Web/MVC/Razor Pages
Blazor Server App    blazorserver    [C#]          Web/Blazor
Class Library         classlib        [C#],F#,VB    Common/Library
Console App          console         [C#],F#,VB    Common/Console
Windows Forms App    winforms       [C#],VB       Common/WinForms
WPF Application      wpf            [C#],VB       Common/WPF

An example would be:
dotnet new console

Display template options with:
dotnet new console -h
Display all installed templates with:
dotnet new list
Display templates available on NuGet.org with:
dotnet new search web
```



# Visual Studio



# CHAPTER 3

Syntax



Microsoft  
.NET

# Syntax



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
//Namespaces are used to organize code into logical units
namespace MyFirstApp
{
    //Internal - An access modifier in C# is a keyword that is used to control the accessibility of a member or a type
    //Classes are a powerful tool that can be used to organize code and to create reusable objects
    internal class Program
    {
        //A function is a block of code that is used to perform a specific task.
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

# Commenting



- Comments are used for explaining code.
- Compilers ignore the comment entries and do not execute them.

```
//This is a single line comment
```

```
/* This is a multi-line comment.
```

```
It can be used to provide more detailed explanations of code.
```

```
*/
```

# Data types

Data Type	Size	Description
Boolean	1 byte	Can store a value of true or false.
Char	1 byte	Can store a single character.
Byte	1 byte	Can store an integer value from -128 to 127.
SByte	1 byte	Can store an integer value from -128 to 127.
Short	2 bytes	Can store an integer value from -32768 to 32767.
UShort	2 bytes	Can store an integer value from 0 to 65535.
Int	4 bytes	Can store an integer value from -2147483648 to 2147483647.
UInt	4 bytes	Can store an integer value from 0 to 4294967295.
Long	8 bytes	Can store an integer value from -9223372036854775808 to 9223372036854775807.
ULong	8 bytes	Can store an integer value from 0 to 18446744073709551615.
Float	4 bytes	Can store a floating-point value with a precision of 7 decimal places.
Double	8 bytes	Can store a floating-point value with a precision of 15 decimal places.
Decimal	12 bytes	Can store a floating-point value with a precision of 28 decimal places.
String	Variable	Can store a sequence of characters.
DateTime	8 bytes	Can store a date and time value.
Guid	16 bytes	Can store a globally unique identifier.
Nullable<T>	1 byte	Can store a value of type T or null.
Object	Variable	Can store any type of object.

# Variable – Value Type

Data Type	Size	Description
sbyte	1 byte	Signed byte, can store values from -128 to 127
byte	1 byte	Unsigned byte, can store values from 0 to 255
short	2 bytes	Signed short integer, can store values from -32,768 to 32,767
ushort	2 bytes	Unsigned short integer, can store values from 0 to 65,535
int	4 bytes	Signed integer, can store values from -2,147,483,648 to 2,147,483,647
uint	4 bytes	Unsigned integer, can store values from 0 to 4,294,967,295
long	8 bytes	Signed long integer, can store values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	8 bytes	Unsigned long integer, can store values from 0 to 18,446,744,073,709,551,615
char	2 bytes	Unicode character, can store any character in the Unicode character set
float	4 bytes	Single-precision floating-point number, can store values with a precision of 7 decimal places
double	8 bytes	Double-precision floating-point number, can store values with a precision of 15 decimal places
decimal	12 bytes	Decimal floating-point number, can store values with a precision of 28 decimal places
bool	1 byte	Boolean value, can store either true or false

# Variable – Reference Type

Data Type	Description
object	The base class of all reference types.
string	A sequence of characters.
class	A user-defined type that can be used to create objects.
interface	A contract that defines a set of methods that must be implemented by a class.
delegate	A type that represents a method.
array	A collection of elements of the same type.

# Operators

- **Arithmetic Operators:**

- Addition: +
- Subtraction: -
- Multiplication: \*
- Division: /
- Modulus (remainder): %
- Increment: ++
- Decrement: --

- **Assignment Operators:**

- Simple assignment: =
- Addition assignment: +=
- Subtraction assignment: -=
- Multiplication assignment: \*=
- Division assignment: /=
- Modulus assignment: %=

- **Comparison Operators:**

- Equal to: ==
- Not equal to: !=
- Greater than: >
- Less than: <
- Greater than or equal to: >=
- Less than or equal to: <=

- **Logical Operators:**

- Logical AND: &&
- Logical OR: ||
- Logical NOT: !

- **Bitwise**

- Bitwise AND: &
- Bitwise OR: |
- Bitwise XOR: ^
- Bitwise complement: ~
- Left shift: <<
- Right shift: >>



# Logical Expressions (IF)



```
int number = 10;
if (number > 0)
{
    Console.WriteLine("Number is positive.");
}
else if (number < 0)
{
    Console.WriteLine("Number is negative.");
}
else
{
    Console.WriteLine("Number is zero.");
}
```

# Logical Expressions (Switch)

```
• int dayOfWeek = 3;
• switch (dayOfWeek)
• {
•     case 1:
•         Console.WriteLine("Monday");
•         break;
•     case 2:
•         Console.WriteLine("Tuesday");
•         break;
•     case 3:
•         Console.WriteLine("Wednesday");
•         break;
•     default:
•         Console.WriteLine("Invalid day");
•         break;
• }
```

# Loops

- For loop

```
for (int i = 1; i <= 10; i++)  
{  
    Console.WriteLine(i);  
}
```

- While loop

```
int i = 1;  
while (i <= 10)  
{  
    Console.WriteLine(i);  
    i++;  
}
```

- Do-while loop

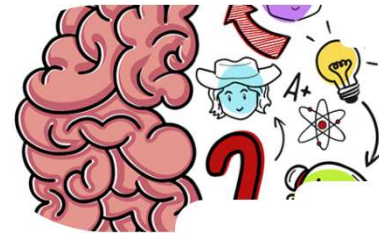
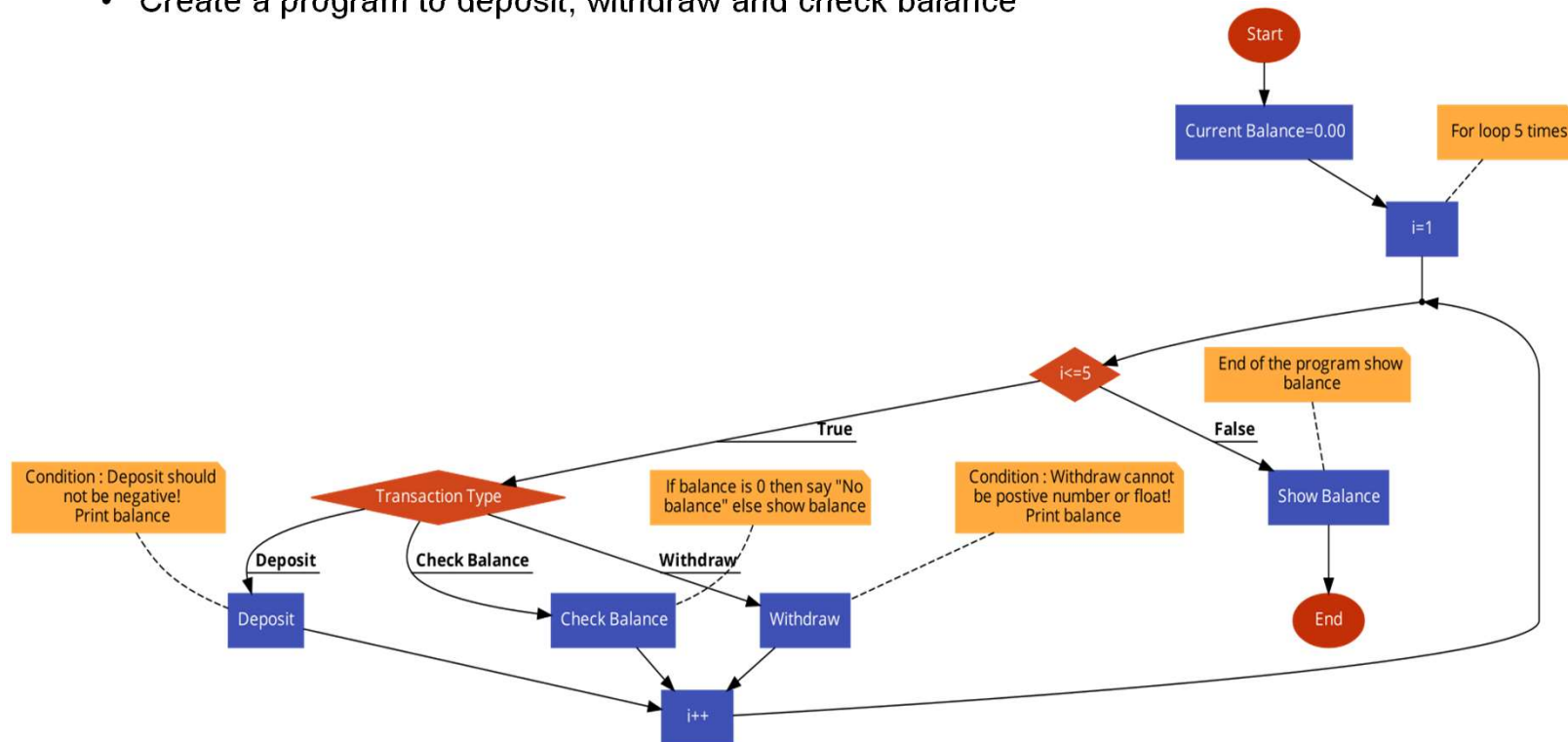
```
int i = 1;  
do  
{  
    Console.WriteLine(i);  
    i++;  
} while (i <= 10);
```

- For Each

```
string[] names = { "John", "Mary",  
    "Peter" };  
foreach (string name in names)  
{  
    Console.WriteLine(name);  
}
```

# Try it yourself

- Create a program to deposit, withdraw and check balance



# Scopes



```
static void Method() {  
    int a = 1;  
    for (int i = 0; i < 5; i++)  
    {  
        a++;  
    }  
    i = 0; //does not exist in the current context  
}
```

# String



- `string` firstString = "Hello";
- `string` secondString = "World";
- `string` thirdString = "!";
- `char` a='a'
- 
- `String` concatenatedString = firstString + secondString;
- `concatenatedString` = `String.Concat`(firstString, secondString, thirdString);
- `concatenatedString` = `$"Hello {firstString} {secondString} {thirdString}"`;
- `concatenatedString` = `string.Format`("{0} {1} {2}", firstString, secondString, thirdString);

# In-built Function



---

- String manipulation
  - Length - Returns the length of the string.
  - Contains - Returns a boolean value indicating whether the string contains a specified substring.
  - IndexOf - Returns the index of the first occurrence of a specified substring in the string.
  - Substring - Returns a substring of the string, starting at a specified index and ending at another specified index.
  - Replace - Replaces all occurrences of a specified substring in the string with another substring.
  - ToLower - Converts all uppercase characters in the string to lowercase.
  - ToUpper - Converts all lowercase characters in the string to uppercase.
  - Trim - Removes all leading and trailing whitespace characters from the string.
  - PadLeft - Pads the string with a specified number of spaces on the left.
  - PadRight - Pads the string with a specified number of spaces on the right.
  - Split - Splits the string into an array of strings, using a specified delimiter.
  - Join - Joins an array of strings into a single string, using a specified delimiter.

# In-built Function



- Number manipulation
  - Abs - Returns the absolute value of a number.
  - Ceiling - Returns the smallest integer that is greater than or equal to a number.
  - Floor - Returns the largest integer that is less than or equal to a number.
  - Pow - Returns the value of a number raised to a specified power.
  - Round - Returns the nearest integer to a number.
  - Sin - Returns the sine of a number.
  - Cos - Returns the cosine of a number.
  - Tan - Returns the tangent of a number.
  - Sqrt - Returns the square root of a number.
  - Log - Returns the logarithm of a number to a specified base.
  - Exp - Returns the exponential value of a number.
  - Atan2 - Returns the arctangent of a number, taking into account the sign of the second argument.
  - Max - Returns the maximum of two numbers.
  - Min - Returns the minimum of two numbers.
  - Pow - Returns the value of a number raised to a specified power.



# In-built Function

- Date Time manipulation
- Now - Returns the current date and time.
- Today - Returns the current date.
- Yesterday - Returns the date that is one day before the current date.
- Tomorrow - Returns the date that is one day after the current date.
- AddDays - Adds a specified number of days to a DateTime object.
- SubtractDays - Subtracts a specified number of days from a DateTime object.
- AddHours - Adds a specified number of hours to a DateTime object.
- SubtractHours - Subtracts a specified number of hours from a DateTime object.
- AddMinutes - Adds a specified number of minutes to a DateTime object.
- SubtractMinutes - Subtracts a specified number of minutes from a DateTime object.
- AddSeconds - Adds a specified number of seconds to a DateTime object.

- Date Time manipulation
- SubtractSeconds - Subtracts a specified number of seconds from a DateTime object.
- AddMilliseconds - Adds a specified number of milliseconds to a DateTime object.
- SubtractMilliseconds - Subtracts a specified number of milliseconds from a DateTime object.
- AddTicks - Adds a specified number of ticks to a DateTime object.
- SubtractTicks - Subtracts a specified number of ticks from a DateTime object.
- CompareTo - Compares two DateTime objects and returns an integer indicating which object is greater, less, or equal to the other object.
- Equals - Returns a Boolean value indicating whether two DateTime objects are equal.
- Many more...

# Function

- A function in C# is a block of code that is given a name and can be executed by calling its name. Functions are used to organize code and to make it reusable.

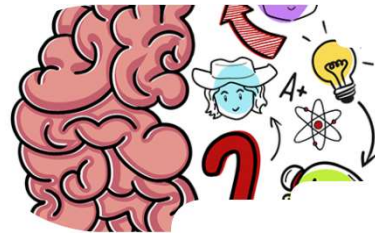
```
int Add(int x, int y)
{
    return x + y;
}
```

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Person: {Name.ToUpper()} ({Age})";
    }
}
```

# Try it yourself

- C# program that calculates age based on user-input date of birth
- make 3 variables of strings dd, mm, yyyy and convert them to datetime.
- Use try parse.
- Validate date if it's future date then say invalid date or it's a future date.
- Validate
  - “dd” must be greater or equal to 1 and less than 32,
  - “mm” must be greater or equals to 1 and less than 13 or equals to 12
  - “yyyy” must be greater than 1900 and till this year
- Output must in days, month and year
  - For eg:  
Your are 1000 days old.  
Your are 100 month/s old.  
Your are 1 year/s old.



# Collections



- C# collections are a set of classes that are used to store and manage data.
- They are designed to be efficient and easy to use, and they provide a variety of methods for adding, removing, searching, and sorting data.
- Some of the most common types of collections include:
  - Arrays: Arrays are the simplest type of collection. They are fixed in size, and they can only store data of a single type.
  - Lists: Lists are similar to arrays, but they can grow and shrink dynamically. They can also store data of any type.
  - Dictionaries: Dictionaries are a type of collection that stores data in key-value pairs. The key is used to identify the data, and the value is the data itself.
  - Stacks: Stacks are a type of collection that stores data in a last-in, first-out (LIFO) order. This means that the last item added to the stack is the first item to be removed.
  - Queues: Queues are a type of collection that stores data in a first-in, first-out (FIFO) order. This means that the first item added to the queue is the first item to be removed.

# Array Example

```
internal class Program
{
    static void Main(string[] args)
    {
        // Declare an array of integers
        int[] numbers = new int[5];

        // Initialize the array with values
        numbers[0] = 1;
        numbers[1] = 2;
        numbers[2] = 3;
        numbers[3] = 4;
        numbers[4] = 5;

        // Print the array elements
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

# List Example



```
public static void Main()
{
    // Create a list of strings.
    List<string> names = new List<string>();

    // Add some names to the list.
    names.Add("John");
    names.Add("Mary");
    names.Add("Peter");
    names.Remove("Mary");

    foreach (string name in names)
    {
        Console.WriteLine(name);
    }
}
```

# Dictionary Example

```
public static void Main()
{
    // Create a dictionary of strings.
    Dictionary<string, string> names = new Dictionary<string, string>();

    // Add some names to the dictionary.
    names.Add("John", "Doe");
    names.Add("Mary", "Smith");
    names.Add("Peter", "Jones");
    names.Remove("Mary");

    // Get the value for a particular key.
    string johnsName = names["John"];

    // Loop through the dictionary and print the names.
    foreach (KeyValuePair<string, string> name in names)
    {
        Console.WriteLine("{0}: {1}", name.Key, name.Value);
    }
}
```

# Stack Example



```
public static void Main()
{
    // Create a stack of strings.
    Stack<string> names = new Stack<string>();

    // Add some names to the stack.
    names.Push("John");
    names.Push("Mary");
    names.Push("Peter");

    // Pop the names off the stack and print them.
    while (names.Count > 0)
    {
        string name = names.Pop();
        Console.WriteLine(name);
    }
}
```



# Queue Example



```
public static void Main()
{
    // Create a queue of strings.
    Queue<string> names = new Queue<string>();

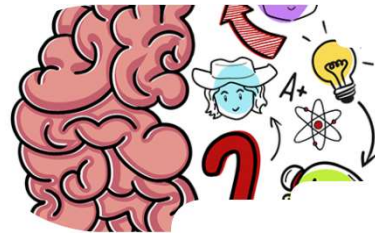
    // Add some names to the queue.
    names.Enqueue("John");
    names.Enqueue("Mary");
    names.Enqueue("Peter");

    // Dequeue the names from the queue and print them.
    while (names.Count > 0)
    {
        string name = names.Dequeue();
        Console.WriteLine(name);
    }
}
```

# Try it yourself

---

- Make classes
- Example: Book, Library, Student



# CHAPTER 4

## Building blocks of OOPS



# Building block of OOP

- **Classes**
- **Objects**
- **Methods**
- **Attributes**



# Class



---

- A class is a blueprint for creating objects. It defines the data and behavior of objects. Classes are used to organize code and data, and to make code reusable.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Person: {Name.ToUpper()} ({Age})";
    }
}
```

# Constructor



```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person()
    {
        // Initialize the data members to their
default values
        Name = "";
        Age = 0;
    }
    public Person(string name, int age)
    {
        // Initialize the data members with the
values passed as parameters
        Name = name;
        Age = age;
    }
}
```

# Destructor

The destructor is called implicitly by the garbage collector when the object is no longer needed. The programmer has no control over when the destructor is called.

- A destructor can only be defined in a class.
- A destructor cannot have any parameters.
- A destructor cannot return a value.
- A destructor cannot be overloaded.
- A destructor cannot be called explicitly.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    ~Person()
    {
        // Perform cleanup operations
        Console.WriteLine("Person object is being destroyed");
    }
}
```

# Object



- An object is an instance of a class. It has the data and behavior defined by its class. Objects are used to represent real-world entities, such as cars, people, and computers.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Person: {Name.ToUpper()} ({Age})";
    }
}
```

```
Person p1 = new Person();
p1.Name = "Ramesh";
p1.Age = 22;
```

```
Person p2 = new Person();
p2.Name = "Shyam";
p2.Age = 23;
```



# Methods

- Methods represent behaviors. Methods perform actions; methods might return information about an object, or update an object's data. The method's code is defined in the class definition.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Person: {Name.ToUpper()} ({Age})";
    }
}
```

```
Person p1 = new Person();
p1.Name = "Ramesh";
p1.Age = 22;
```

```
Person p2 = new Person();
p2.Name = "Shyam";
p2.Age = 23;
```

# Attributes



- Attributes are the information that is stored. Attributes are defined in the Class template. When objects are instantiated individual objects contain data stored in the Attributes field.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Person: {Name.ToUpper()} ({Age})";
    }
}

Person p1 = new Person();
p1.Name = "Ramesh";
p1.Age = 22;
```

# Principles of OOP

- **Inheritance:** child classes inherit data and behaviors from parent class
- **Encapsulation:** containing information in an object, exposing only selected information
- **Abstraction:** only exposing high level public methods for accessing an object
- **Polymorphism:** many methods can do the same task



# Inheritance

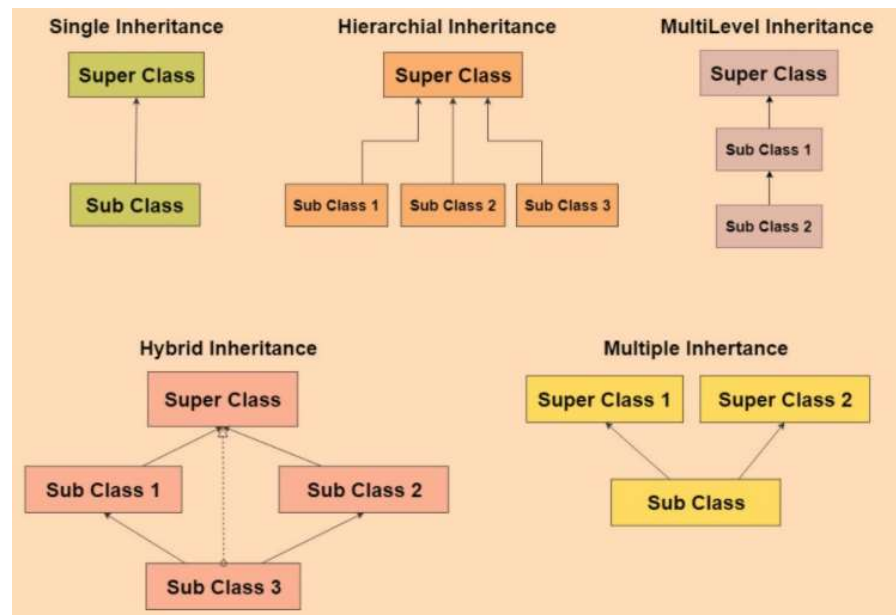


---

- process by which one class acquires the properties and functionalities of another class
- Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class

# Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- ~~Multiple Inheritance~~



# Inheritance Example

```
public class Dog
{
    public string Name { get; set; } = "";
    public virtual void Bark()
    {
        Console.WriteLine("Bark: Woof!");
    }
    public void Eat()
    {
        Console.WriteLine("Eat: Nom Nom Nom!");
    }
    public void Drink()
    {
        Console.WriteLine("Drink: Slurp Slurp!");
    }
    public void WagTail()
    {
        Console.WriteLine("Wag: Thump, thump, thump!");
    }
}
```

```
public class Poodle : Dog
{
    public string Size { get; set; } =
    "small";
    public override void Bark()
    {
        Console.WriteLine("High pitch bark");
    }
}

Poodle tommy = new Poodle();
tommy.Name = "Tommy";
tommy.Size = "Small";
tommy.Bark();
tommy.Eat();
```

# Encapsulation



---

- binding object state(fields) and behavior(methods) together
- If you are creating class, you are doing encapsulation.
- hides the internal software code implementation inside a class, and hides internal data of inside objects.

# Encapsulation Example

```
public class Car
{
    private int _speed;

    public int Speed
    {
        get { return _speed; }
        set { _speed = value; }
    }

    public void Drive()
    {
        // Code to make the car drive
    }

    public void Stop()
    {
        // Code to make the car stop
    }
}
```





# Abstraction



---

- process where you show only “relevant” data and “hide” unnecessary details of an object from the user
- Abstraction is an extension of encapsulation

# Abstraction Example

```
public class BankAccount
{
    private float balance { get; set; }

    public void Deposit(float amount)
    {
        balance += amount;
    }

    public void Withdraw(float amount)
    {
        if(balance <= 0)
        {
            throw new CustomerError("Cannot withdraw from negative balance");
        }
        balance -= amount;
    }

    public void ShowBalance(float amount)
    {
        Console.WriteLine("Balance :" + balance);
    }
}
```



# Polymorphism



- means designing objects to **share behaviors**
- one method with multiple implementation, for a certain class of action
- allows you define one interface and have multiple implementations
- Types of polymorphism
  - Static Polymorphism (compile time)
  - Dynamic Polymorphism (run time)

# Polymorphism Example

- Method overloading is a technique where you can have multiple methods with the same name, but with different parameters. The compiler will choose the correct method to call based on the number and types of parameters that are passed.

```
public void Print(string message)
{
    Console.WriteLine(message);
}

public void Print(int number)
{
    Console.WriteLine(number);
}
```



# Polymorphism Example

- Method overriding is a technique where a derived class can provide its own implementation for a method that is defined in a base class. The compiler will call the overridden method in the derived class, even if there is a method with the same name in the base class.

```
public class Car
{
    public void Drive()
    {
        Console.WriteLine("The car is driving");
    }
}

public class SportsCar : Car
{
    public override void Drive()
    {
        Console.WriteLine("The sports car is driving fast");
    }
}

public class Main()
{
    Car car = new Car();
    car.Drive(); // The car is driving

    SportsCar sportsCar = new SportsCar();
    sportsCar.Drive(); // The sports car is driving fast
}
```



# Abstract class and Methods



abstract class outlines the methods but not necessarily implements all the methods

Abstract method is declared but not defined

A class derived from the abstract base class must implement those methods that are not implemented

Abstract class cannot be instantiated which means you cannot create the object of abstract class

If a child does not implement all the abstract methods of parent class , then the child class must need to be declared abstract.

```
public abstract class Vehicle
{
    public abstract void Drive();
    public abstract void Stop();
}
```

# Interface



---

- An interface is a blueprint of a class
- Interfaces can contain only constants and abstract methods (methods with only signatures no body)
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.
- Interface is a common way to achieve full abstraction in C#
- What is difference between abstract class and Interface?

# Interface Example



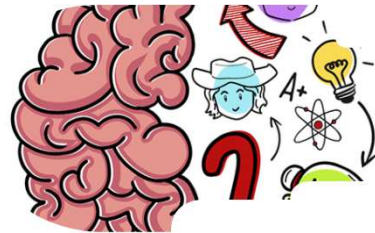
```
public interface Drawable
{
    void draw();
}
public class Rectangle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
public class Circle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing circle...");
    }
}
```



# Try it yourself

---

- Make classes
- Book
  - Add detail
  - Remove
  - List
- Library
  - Store books
  - Remove
  - List
  - Assign books to student
- Student
  - Add detail
  - Remove
  - Return book



# CHAPTER 5

## Error Handling



# Error Handling



- The try-catch-finally statement is used to handle exceptions.
- The try block contains the code that might throw an exception
- the catch block contains the code that is executed if an exception is thrown,
- and the finally block contains the code that is executed regardless of whether or not an exception is thrown.

# Error Handling



---

```
try
{
    // Code that might throw an exception
}
catch (Exception ex)
{
    // Handle the exception
}
finally
{
    // Code that is executed regardless of whether or not an exception is
    thrown
}
```

# Types of Exception

Exception Class	Description
ArgumentException	Raised when a non-null argument that is passed to a method is invalid.
ArgumentNullException	Raised when null argument is passed to a method.
ArgumentOutOfRangeException	Raised when the value of an argument is outside the range of valid values.
DivideByZeroException	Raised when an integer value is divide by zero.
FileNotFoundException	Raised when a physical file does not exist at the specified location.
FormatException	Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse.
IndexOutOfRangeException	Raised when an array index is outside the lower or upper bounds of an array or collection.
InvalidOperationException	Raised when a method call is invalid in an object's current state.
KeyNotFoundException	Raised when the specified key for accessing a member in a collection is not exists.
NotSupportedException	Raised when a method or operation is not supported.
NullReferenceException	Raised when program access members of null object.
OverflowException	Raised when an arithmetic, casting, or conversion operation results in an overflow.
OutOfMemoryException	Raised when a program does not get enough memory to execute the code.
StackOverflowException	Raised when a stack in memory overflows.
TimeoutException	The time interval allotted to an operation has expired.

# CHAPTER 6

## Generics



# Generics



---

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET class library contains several generic collection classes in the System.Collections.Generic namespace. The generic collections should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

# Generics



---

- The .NET class library contains several generic collection classes in the System.Collections.Generic namespace.
- ArrayList : Implements the IList interface using an array whose size is dynamically increased as required.
- ICollection : Defines size, enumerators, and synchronization methods for all nongeneric collections.
- IEnumerable : Exposes an enumerator, which supports a simple iteration over a non-generic collection.
- IList : Represents a non-generic collection of objects that can be individually accessed by index.
- IDictionary : Represents a nongeneric collection of key/value pairs.
- Hashtable : Represents a collection of key/value pairs that are organized based on the hash code of the key.
- Queue : Represents a first-in, first-out collection of objects.
- Stack: Represents a simple last-in-first-out (LIFO) non-generic collection of objects.



# Generics Example

```
public class Bag<T>
{
    private List<T> items = new List<T>();
    public void Add(T item)
    {
        items.Add(item);
    }

    public void Remove(T item)
    {
        items.Remove(item);
    }

    public bool Contains(T item)
    {
        return items.Contains(item);
    }

    public int Count()
    {
        return items.Count;
    }
}
```

```
Bag<string> myBag = new Bag<string>();
```