

Chapter 2

Piyush Pant

Introduction to C#

C# is a modern, object-oriented programming language developed by Microsoft as part of its .NET initiative. It is designed to be simple, powerful, and versatile, making it suitable for a wide range of applications.

Basic Features :

Object-Oriented: Supports abstraction, encapsulation, inheritance, and polymorphism.

Type-Safe: Prevents type errors by enforcing strict type rules.

Rich Standard Library: Provides a robust library for various functionalities.


Interoperable: Works seamlessly with other .NET languages.

Platform-Independent: Enables cross-platform development through .NET Core.

Memory Management: Automatic garbage collection and efficient memory handling.



Comments in c#



Comments are non-executable lines in code used to describe or explain the code's logic.

Purpose:

- Improve code readability.
- Help developers understand the code.
- Serve as documentation for future reference.



Types of Comments

Single-line Comments: Use `//` to comment a single line.

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

Multi-line Comments: Use `/*` to start and `*/` to end a comment block.

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

XML Documentation Comments: Use `///` to generate documentation.

```
/// <summary>
```

```
/// This method adds two numbers.
```

```
/// </summary>
```

Variables

What are Variables?

- Containers for storing data.
- Each variable has a type that determines what kind of data it can hold.

Syntax

`dataType variableName = value ;`

Example :

```
int age = 25;
```

```
string name = "John" ;
```



Data Types

What are Data Types?

- Define the type of data a variable can store.

Value Types: Store data directly (e.g., int, double, char, bool).

Reference Types: Store references to data (e.g., string, arrays, objects).

Common Value Types:

int: Whole numbers.

double: Decimal numbers.

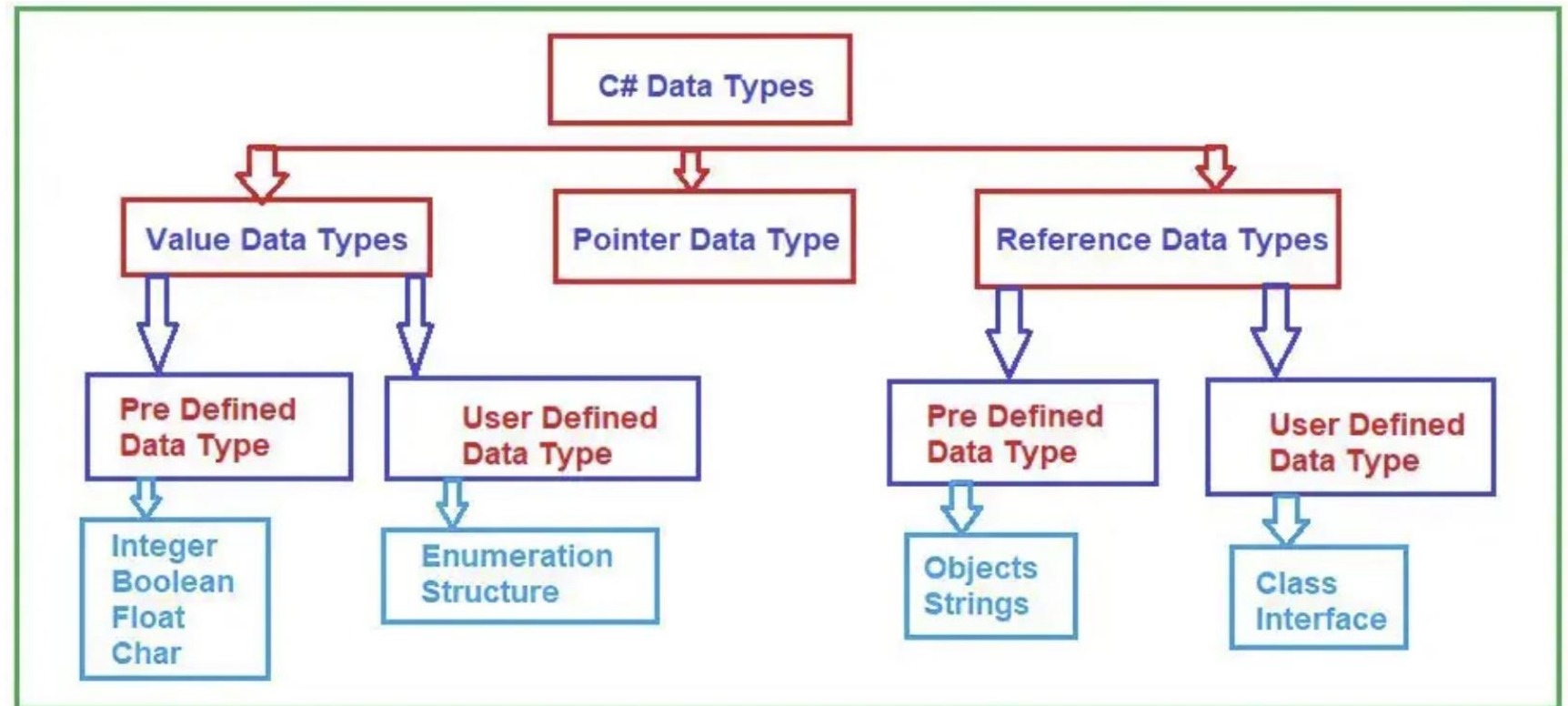
char: Single characters.

bool: True/false values.

Code Example

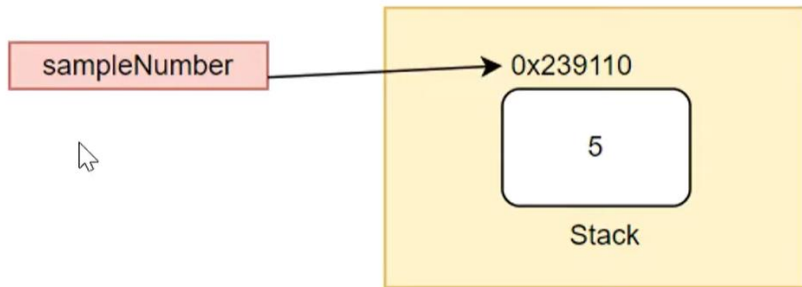
```
int number = 10;    // Integer type
double price = 99.99; // Floating-point type
char grade = 'A';   // Character type
bool isPassed = true; // Boolean type
string message = "Hello!"; // String type
```

Data Types



Value Type vs Reference Type

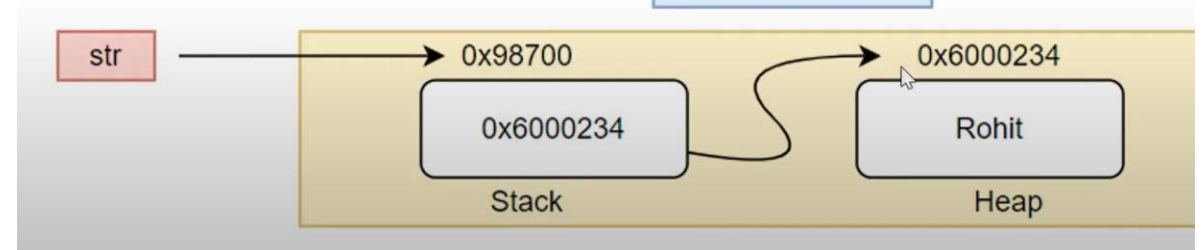
Memory Allocation



```
int sampleNumber = 5;
```

```
string str = "Rohit"; //HEAP
```

Memory Allocation



Data Types

Short Name	.NET Class	Type	Width	Range (bits)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65535
long	Int64	Signed integer	64	-9223372036854775808 to 9223372036854775807
ulong	UInt64	Unsigned integer	64	0 to 18446744073709551615
float	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	True or false
object	Object	Base type of all other types		
string	String	A sequence of characters		
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$

Implicit Conversion

Implicit conversion is automatically handled by C# when a smaller data type is converted to a larger data type (e.g., int to double).

```
int marks = 85;  
double gradePoint = marks / 10.0; // Implicit conversion from int to double
```

When dividing by 10.0 (a double), C# automatically converts the integer result to a double without requiring explicit casting.

Explicit Conversion

Explicit conversion is needed when converting from a larger data type to a smaller one (e.g., double to int), and it requires a cast.

```
double gradePoint = 3.6;  
int truncatedGradePoint = (int)gradePoint; // Explicit conversion from double to int
```

The decimal part of the gradePoint is truncated when explicitly converted to int.

Operators

Symbols or keywords used to perform operations on variables and values.

	Operator	Type
Binary Operator →	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator →	++, --	Unary Operators
Ternary Operator →	?:	Ternary Operator or Conditional Operator

Operators

Arithmetic Operators

```
int a = 10;  
int b = 5;  
int sum = a + b; //15
```

Relational Operator

```
bool isEqual = (a == b); // false
```

Logical

```
bool result = (a > b) && (b > 0); // true
```

Bitwise:

```
int x = 5; // 0101 in binary  
int y = 3; // 0011 in binary  
int andResult = x & y; // 0001 (1 in decimal)  
int orResult = x | y; // 0111 (7 in decimal)
```

Operators

Assignment:

```
int num = 10;  
num += 5; // num = 15
```

Unary

```
int count = 10;  
count++; // Increment by 1, count = 11  
count--;
```

Ternary

```
int age = 18;  
string result = (age >= 18) ? "Adult" : "Minor";
```



Variable Scope

What is Variable Scope?

The area in code where a variable is accessible.

Types:

Local Variables: Declared inside methods; accessible only within those methods.

Global Variables: Declared outside methods; accessible throughout the class.

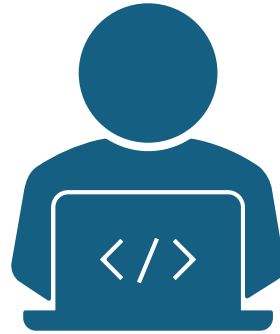
```
public class Program {  
    int globalVar = 10; // Global Variable  
  
    public void Method() {  
        int localVar = 5; // Local Variable  
    }  
}
```

Best Practices

- Use meaningful variable names.
- Declare variables with the smallest scope possible.
- Initialize variables before using them.
- Choose the appropriate data type for the variable.



C# Exercise



Write a C# program that asks the user to input two integer numbers. Perform the following operations and display the results:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)
- Increment the first number (++) and display the result.
- Decrement the second number (--) and display the result.

Example Input/Output:

Input:

Enter the first number: 10

Enter the second number: 3

Output:

Addition: 13

Subtraction: 7

Multiplication: 30

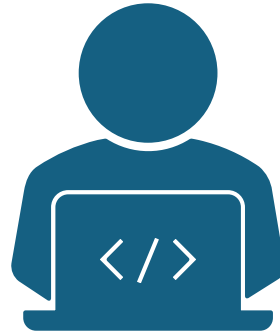
Division: 3

Modulus: 1

After Increment, First Number: 11

After Decrement, Second Number: 2

C# Exercise



Write a C# program to compare two numbers entered by the user. Perform the following:

- Use relational operators (>, <, >=, <=, ==, !=) to compare the two numbers and display the results.
- Use logical operators (&&, ||, !) to check the following conditions:
 - Both numbers are greater than 10.
 - At least one of the numbers is even.
 - Neither number is negative.

Input:

Enter the first number: 15

Enter the second number: 8

Output:

First number > Second number: True

First number < Second number: False

First number >= Second number: True

First number <= Second number: False

First number == Second number: False

First number != Second number: True

Both numbers are greater than 10:

False

At least one number is even: True

Neither number is negative: True

Condition Statements

Condition statements help control the flow of the program based on whether a condition is true or false.

They are crucial for decision-making in programming.

If-Else, Switch, and Ternary Operators

If-Else Statement

- The if statement checks whether a condition is true.
- The else block runs when the condition is false.

Syntax

```
if (condition) {  
    // code block if condition is true  
}  
else {  
    // code block if condition is false  
}
```

Code :

```
int number = 10;  
if (number > 5)  
{  
    Console.WriteLine("Number is greater than 5");  
}  
else  
{  
    Console.WriteLine("Number is not greater than 5"); }  
}
```

Else-If Ladder

- An else if ladder allows multiple conditions to be tested sequentially.
- If the first condition fails, the next condition is checked, and so on.

Syntax

```
if (condition1) {  
    // code block if condition is true  
}  
else if (condition2)  
{  
    // code block if condition2 is true  
}  
else {  
    // code block if condition is false  
}
```

Code :

```
int number = 15;  
if (number > 20)  
{  
    Console.WriteLine("Number is greater than  
20");  
}  
else if (number > 10)  
{  
    Console.WriteLine("Number is greater than 10  
but less than or equal to 20");  
}  
else  
{  
    Console.WriteLine("Number is 10 or less");  
}
```

Switch Statement

- The switch statement is used when there are multiple possible values for a variable.
- It is often more readable than multiple if-else statements.

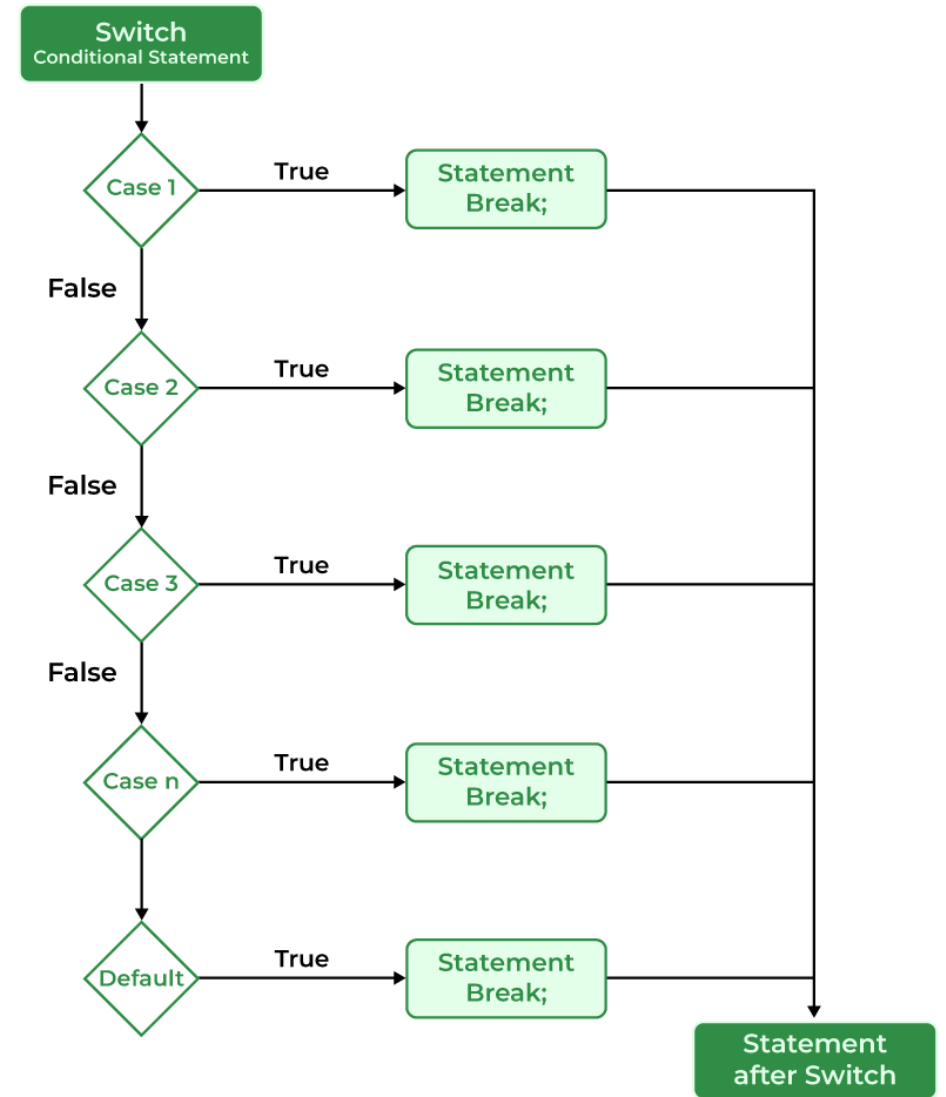
Syntax

```
switch (variable)
{
    case value1:
        // code block if variable == value1
        break;
    case value2:
        // code block if variable == value2
        break;
    default:
        // code block if no case matches
        break;
}
```

Switch Statement

Code

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



Ternary Operator

- The ternary operator is a shorthand for `if-else` statements.
- It is useful for assigning values based on a condition.

Syntax

`condition ? expression_if_true : expression_if_false;`

Code :

```
int age = 20;  
string result = age >= 18 ? "Adult" : "Minor";  
Console.WriteLine(result);
```


Nested Condition Statements

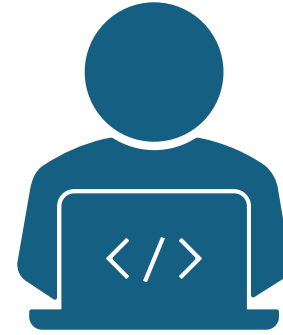
Condition statements can be nested inside one another to handle more complex conditions.

Code :

```
int age = 25;
bool hasTicket = true;

if (age >= 18)
{
    if (hasTicket)
    {
        Console.WriteLine("You can enter the event.");
    }
    else
    {
        Console.WriteLine("You need a ticket to enter.");
    }
}
else
{
    Console.WriteLine("You must be 18 or older to enter.");
}
```

C# Exercise



Write a C# program that asks the user to enter their **marks** (an integer between 0 and 100). Based on the marks, the program should assign a grade and grade point according to the Nepali grading system:

- **Above 90:** Grade **A+**, Grade Point **4.0**, Comment **Outstanding**.
- **80 to 90:** Grade **A**, Grade Point **3.6**, Comment **Excellent**.
- **70 to 80:** Grade **B+**, Grade Point **3.2**, Comment **Very Good**.
- **60 to 70:** Grade **B**, Grade Point **2.8**, Comment **Good**.
- **50 to 60:** Grade **C+**, Grade Point **2.4**, Comment **Satisfactory**.
- **40 to 50:** Grade **C**, Grade Point **2.0**, Comment **Acceptable**.
- **30 to 40:** Grade **D+**, Grade Point **1.6**, Comment **Partially Acceptable**.
- **20 to 30:** Grade **D**, Grade Point **1.2**, Comment **Insufficient**.
- **0 to 20:** Grade **E**, Grade Point **0.8**, Comment **Very Insufficient**.

If the marks are outside the valid range (less than 0 or greater than 100), print:

"Invalid input! Please enter marks between 0 and 100."

Input:

Enter your marks: 92

Output:

Grade: A+

Grade Point: 4.0

Comment: Outstanding

Loops in C# (Repeat Operations)

- The **while loop** checks a condition and executes the statement or statement block following the while. It repeatedly checks the condition, executing those statements until the condition is false.
- The while loop checks the condition before executing the block of code. If the condition is true, the block of code is executed, and then the condition is checked again. If the condition is false initially, the block of code will not be executed at all.
- The **do-while loop** executes the block of code first, and then checks the condition. If the condition is true, the block of code is executed again. This guarantees that the code inside the do-while loop runs at least once, regardless

Code :

```
int counter = 0;
while (counter < 10) {
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Use Cases

while Loop Use Cases:

Pre-Condition Checks: Ideal for scenarios where the condition should be checked before executing the code block. For example, iterating through a list until a specific condition is met.

Polling or Monitoring: Useful in tasks that involve continually checking a condition, such as monitoring a system process or waiting for an event.

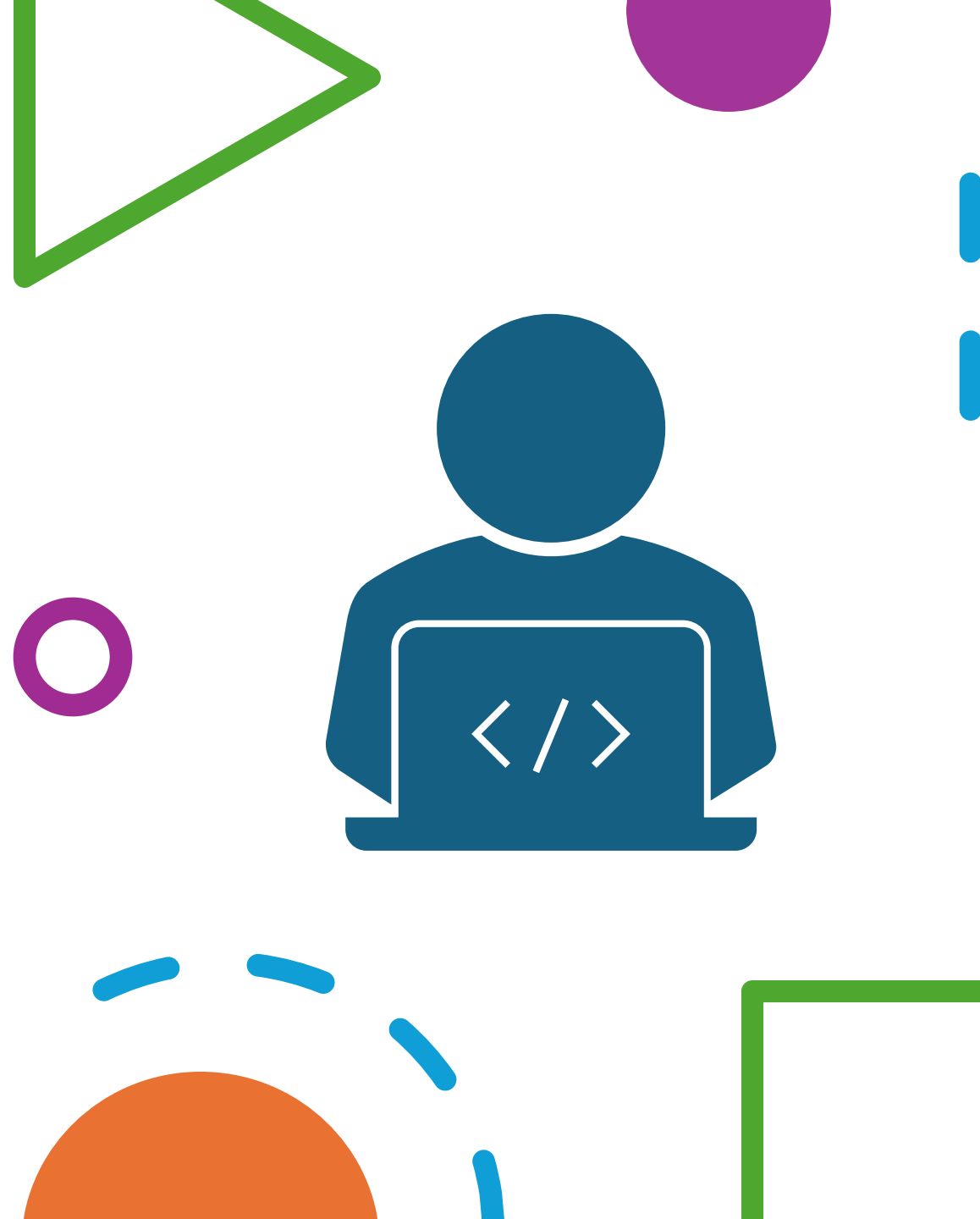
do-while Loop Use Cases:

User Prompting: Perfect for scenarios where you need to execute the code at least once, such as prompting a user for input.

Initialization and Execution: Handy in cases where you want to perform an action at least once and then continue based on a condition, like retrying an operation until it succeeds.

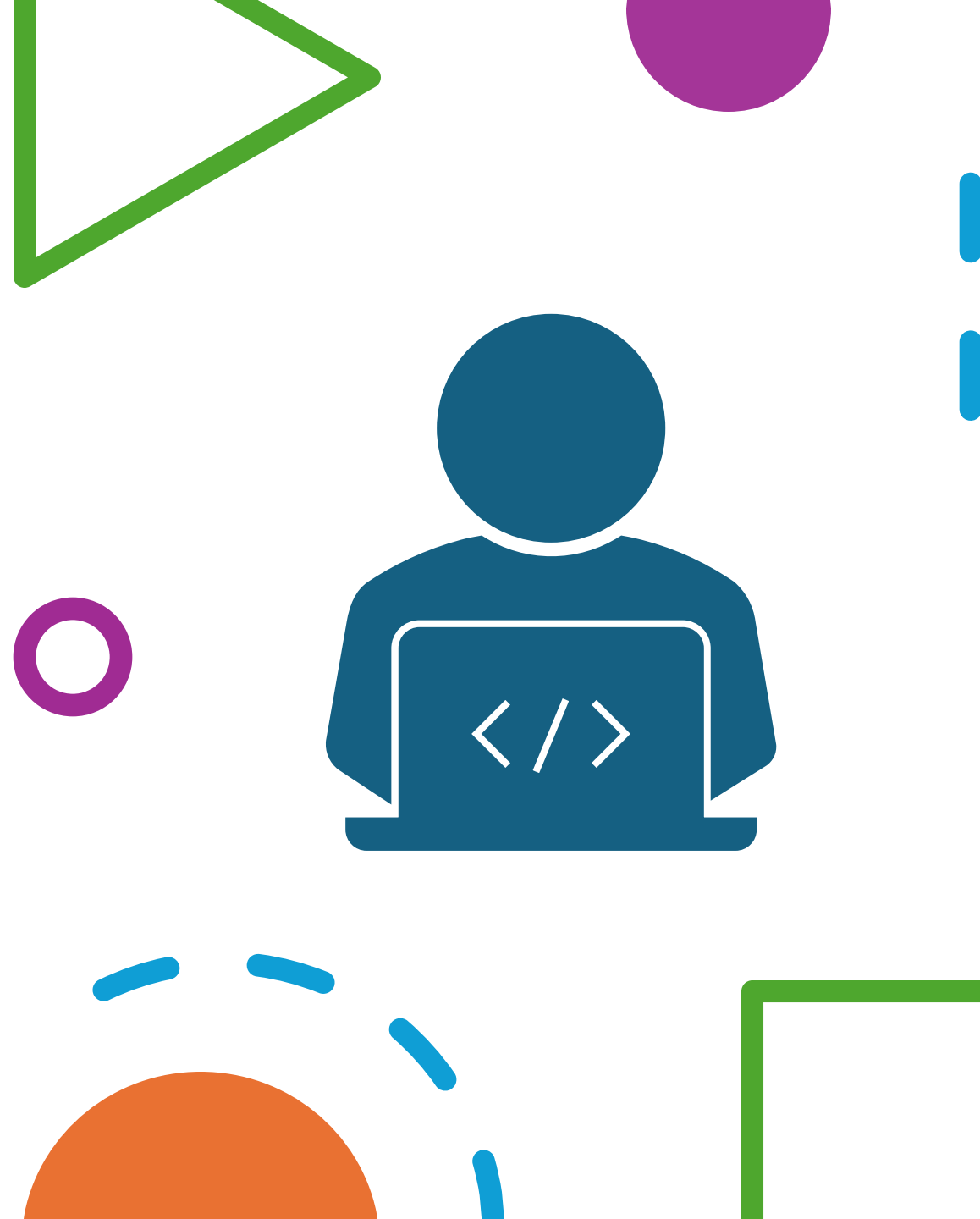
C# Exercise while loop

- Write a program that continuously asks the user for a positive integer and calculates the sum of all even numbers up to that integer.
- Implement a program that asks the user for a password and keeps prompting them until the correct password is entered.
- Create a program that checks if a number is a prime number by using a while loop.



C# Exercise do while loop

- Design a simple game where the user has to guess a number between 1 and 100, and the program keeps prompting them until they guess correctly.
- Write a menu-driven program where the user can choose between several options and the menu reappears until the user chooses "Exit".
- Create a program that simulates rolling a die, and the user is asked if they want to roll again until they choose to stop.



for loop vs forEach loop



FOR LOOP:

GIVES YOU FULL CONTROL OVER
ITERATION, INCLUDING CUSTOM
STEP SIZES AND INDEX
MANIPULATION.



FOREACH LOOP:

SIMPLIFIES ITERATION BY
FOCUSING ONLY ON THE ELEMENTS
OF THE COLLECTION



For loop

for statement has three parts that control how it works.

- The first part is the for initializer: **int index = 0**; declares that index is the loop variable, and sets its initial value to 0.
- The middle part is the for condition: **index < 10** declares that this for loop continues to execute as long as the value of counter is less than 10.
- The final part is the for iterator: **index++** specifies how to modify the loop variable after executing the block following the for statement. Here, it specifies that index should be incremented by 1 each time the block executes.

Code :

```
for (int index = 0; index < 10; index++)  
{  
    Console.WriteLine($"Hello World! The  
    Index is {index}");  
}
```


ForEach loop

for statement has three parts that control how it works.

- The first part is the for initializer: **int index = 0**; declares that index is the loop variable, and sets its initial value to 0.
- The middle part is the for condition: **index < 10** declares that this for loop continues to execute as long as the value of counter is less than 10.
- The final part is the for iterator: **index++** specifies how to modify the loop variable after executing the block following the for statement. Here, it specifies that index should be incremented by 1 each time the block executes.

Code :

```
for (int index = 0; index < 10; index++)  
{  
    Console.WriteLine($"Hello World! The  
    Index is {index}");  
}
```

When to use for

- When you need index-based access.
- When you want custom iteration (e.g., step size).
- When modifying the collection during iteration.

Example: Accessing in reverse order

```
for (int i = numbers.Length - 1; i >= 0; i--)  
{  
    Console.WriteLine(numbers[i]);  
}
```



When to use foreach

- When simplicity and readability are priorities.
- When you only need to access (not modify) elements.
- When iterating over non-indexed collections like Dictionary or HashSet.

```
int[] numbers = { 10, 20, 30, 40, 50 };  
foreach (int num in numbers) {  
    Console.WriteLine($"Number: {num}");  
}
```



Arrays

An array is a collection of elements of the same data type stored in contiguous memory locations.

Fixed Size: Once defined, the size of an array cannot be changed.

Declaration and Initialization

Declaration:

```
int[] numbers;
```

Initialization:

Inline Initialization:

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

Separate Steps:

```
int[] numbers = new int[5];
```

```
numbers[0] = 1;
```

```
numbers[1] = 2;
```



Arrays Key features

Zero-Based Indexing:

First element at index 0.

Last element at index Length - 1.

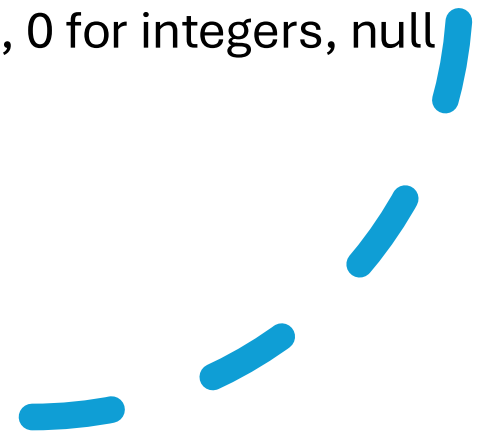
Fixed Length: Defined at the time of creation.

```
int[] arr = new int[3];
```

```
Console.WriteLine(arr.Length); // Output: 3
```

Default Values:

Elements are initialized to default values (e.g., 0 for integers, null for reference types).



Types of Arrays

Single-Dimensional Arrays:

Example:

```
int[] numbers = new int[5];
```

Multi-Dimensional Arrays:

Example:

```
int[,] matrix = new int[3, 3];
```

Jagged Arrays: (Arrays of arrays)

Example:

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[5];
```

```
jaggedArray[1] = new int[3];
```



Common Array Operations

Access Elements:

```
int first = numbers[0];
```

Modify Elements:

```
numbers[1] = 10;
```

Iterate Through an Array:

Using for:

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

Using foreach:

```
foreach (int num in numbers)  
{  
    Console.WriteLine(num);  
}
```



Advantages and Limitations of Arrays

Advantage

- Easy to use.
- Efficient memory access due to contiguous allocation.
- Allows random access using indices.

Limitations

- Fixed size.
- Elements must be of the same type.
- Limited functionality compared to collections like `List<T>`.

Best Practice

- Use arrays for static and fixed-size data.
- Prefer `List<T>` or other collections for dynamic or flexible-size requirements.



Exception Handling

Exception handling is a programming construct that allows you to detect, manage, and recover from runtime errors in a controlled way.

Purpose:

- Prevent the application from crashing due to unexpected errors.
- Provide clear and meaningful error messages to users.
- Ensure proper cleanup of resources (e.g., closing files, releasing memory)

Exception Handling

Errors in programming are typically classified into three categories:

- **Syntax Errors:**
 - Detected by the compiler when the code violates the language's syntax rules.
 - Example: Missing semicolon or mismatched brackets.
- **Logical Errors:**
 - Flaws in program logic that lead to incorrect behavior.
 - Example: Using the wrong formula for a calculation.
- **Runtime Errors:**
 - Errors that occur while the program is running.
 - Example: Dividing by zero, accessing a null object, or reading from a nonexistent file.



Exception Handling

Without proper exception handling, a runtime error can cause the application to terminate abruptly.

Exception handling allows developers to:

- Gracefully recover from unexpected situations.
- Provide a better user experience with meaningful feedback.
- Maintain application stability even when errors occur.

Exception Handling

C# provides the following keywords for handling exceptions:

try Block

- Code that may throw an exception is placed inside a try block.
- If an exception occurs, the remaining code in the try block is skipped, and the control moves to the appropriate catch block.

```
try{  
    int result = 10 / 0; // Exception occurs here  
}
```

catch Block

- Captures and handles the exception thrown by the try block.
- Each catch block handles a specific type of exception.
- Multiple catch blocks can be used to handle different exception types.

```
catch (DivideByZeroException ex) {  
    Console.WriteLine("Division by zero is not allowed.");  
}
```

finally Block

- The finally block is always executed, regardless of whether an exception occurs or not.
- Commonly used for resource cleanup, such as closing files or releasing memory.

```
finally {  
    Console.WriteLine("Execution completed.");  
}
```



Key words Exception Handling

throw Statement

- Used to manually raise an exception

```
if (number < 0) {  
    throw new ArgumentOutOfRangeException("Number must be positive");  
}
```

System.Exception: Base class for all exceptions.

Below are some frequently encountered exceptions in C#:

- **DivideByZeroException:** Raised when attempting to divide by zero.
- **FormatException:** Occurs when input data has an invalid format, such as converting a string to an integer.
- **NullReferenceException:** Raised when attempting to use an object reference that is null.
- **IndexOutOfRangeException:** Occurs when accessing an array with an invalid index.
- **FileNotFoundException:** Raised when attempting to access a file that does not exist.



Enumerations

An enum (short for *enumeration*) is a value type in C# used to define a set of named constant values.

Purpose:

- Improve code readability by replacing numeric or literal constants with descriptive names.
- Group related constants together for better organization and maintainability.

Enumerations

- Each constant in the enum is assigned an underlying integer value by default, starting from 0.

```
enum DaysOfWeek
```

```
{
```

```
    Sunday,    // 0
```

```
    Monday,    // 1
```

```
    Tuesday,   // 2
```

```
    Wednesday, // 3
```

```
    Thursday,  // 4
```

```
    Friday,    // 5
```

```
    Saturday   // 6
```

```
}
```

Enumerations

Advantages of Enums

Improves Code Readability:

- Replaces numeric constants with descriptive names.

Ensures Type Safety:

- Prevents invalid values from being assigned.

Facilitates Maintenance:

- Easier to update or add constants.

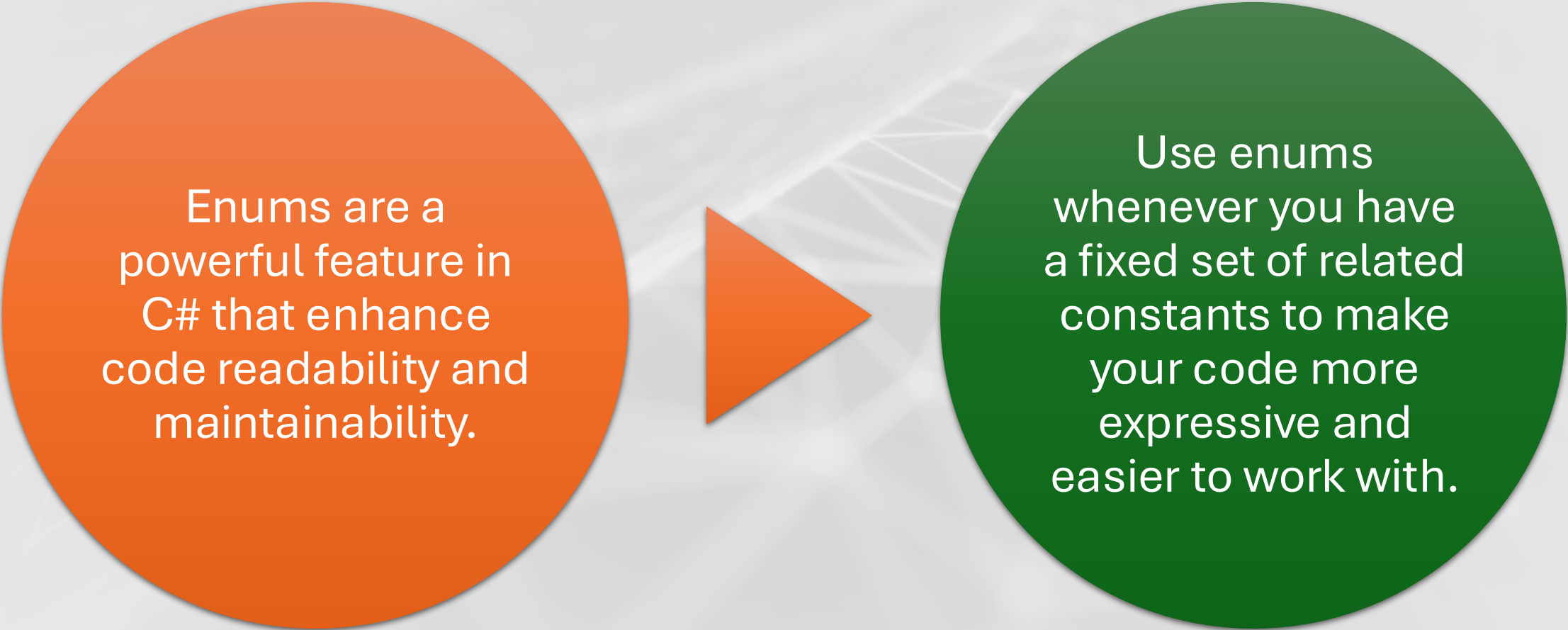
Reduces Errors:

- Reduces the likelihood of hardcoding numeric constants.

Limitations of Enum

- **No Behavior:**
 - Enums can only store values, not methods or logic.
- **Limited Extensibility:**
 - Cannot be extended or derived like classes.
- **No Support for Strings:**
 - Enums only support numeric or integral values.

Enum



Enums are a powerful feature in C# that enhance code readability and maintainability.

The diagram consists of two circles connected by a right-pointing arrow. The left circle is orange and contains text about the benefits of enums. The right circle is green and contains text about when to use enums. The background features a faint, abstract geometric pattern.

Use enums whenever you have a fixed set of related constants to make your code more expressive and easier to work with.

Class

A class is a blueprint for creating objects. It defines a datatype by bundling data and methods that work on the data into one single unit. In C#, a class is defined using the `class` keyword

Fields: Variables inside the class. These can hold data relevant to the object.

Properties: Methods that provide a flexible mechanism to read, write, or compute the values of private fields.

Methods: Functions defined inside the class that can perform actions on the data.

Constructors: Special methods invoked when an object is created to initialize the object.

```
public class Person
{
    // Fields
    private string name;
    private int age;

    // Constructor
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Method
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}, Age: {age}");
    }
}
```

Object

An object is an instance of a class. When a class is defined, no memory is allocated until an object is created. Objects can be created using the new keyword.

When you create an object, you are creating an instance of the class with its own set of values for the fields defined in the class.

```
// Creating an object of the Person class
    Person person1 = new Person("John Doe", 30);
    Person person2 = new Person("Jane Smith", 25);

// Using the object
    person1.DisplayInfo();
    person2.DisplayInfo();
```

Constructor

Constructors and destructors are special types of methods in C# used to initialize and clean up objects when they're created and destroyed.

A constructor is a special method that is called when an object is instantiated. It allows you to set default values and perform any required setup or initialization.

Key Features of Constructors:

Same name as the class: A constructor must have the same name as the class in which it resides.

No return type: Constructors do not have a return type.

Can be overloaded: A class can have multiple constructors with different parameter

Destructor

A destructor is a special method that is called when an object is about to be destroyed. It's used to clean up resources like closing files, releasing memory, etc. Destructors are less common in C# due to the presence of the garbage collector, which automatically handles most resource cleanup.

Key Features of Destructors:

Same name as the class, preceded by a tilde (~).

No parameters and no return type.

Cannot be called directly: They're triggered by the garbage collector when the object is no longer needed.

Partial Class

A **partial class** in C# allows the definition of a class to be split into multiple files. This feature is particularly useful in large projects or situations where multiple developers need to work on different parts of the same class simultaneously.

Key Features of Partial Classes

Multiple Files: A class can be split into multiple files, making it easier to manage and organize code.

Unified Definition: At compile time, all parts are combined into a single class.

Maintainability: Improves maintainability by segregating functionality into separate files.

Collaborative Development: Facilitate teamwork by allowing multiple developers to work on different parts of the same class without conflicts.

Sealed Class

A **sealed class** in C# is a class that cannot be inherited by other classes. This means no other class can derive from a sealed class. Sealed classes are often used to prevent further specialization of classes and to lock down the behavior defined within them.

Key Features of Sealed Classes:

Inheritance Restriction: Classes marked as sealed cannot be extended.

Final Implementation: Ensures that the class's implementation cannot be altered by inheriting classes.

Optimization: In some cases, sealing a class can enable certain compiler optimizations, as the compiler knows that the class cannot be subclassed.

When to Use Sealed Classes:

Security: To prevent inheritance and modification of critical classes.

Performance: In some scenarios, to optimize performance by avoiding polymorphic behavior.

Final Class Design: When the class design is complete and should not be extended.

Static Class

A **static class** is a class that cannot be instantiated. It can only contain static members, such as static fields, methods, properties, and events. This means you cannot create an object of a static class using the new keyword.

Key Characteristics:

Instantiation: Cannot be instantiated.

Members: Contains only static members.

Inheritance: Cannot be inherited or used as a base class.

Access: Members are accessed using the class name itself.

Usage:

Static classes are used when you want to create utility classes that contain methods which do not need to be tied to a specific object instance. They are commonly used for:

- Helper methods
- Extension methods
- Utility functions that operate on data passed as parameters

Static Class

```
public static class MathHelper
{
    public static double PI = 3.14159;

    public static double Square(double number)
    {
        return number * number;
    }

    public static double Add(double a, double b)
    {
        return a + b;
    }
}

// Accessing static members
double result = MathHelper.Add(2, 3);
double area = MathHelper.PI * MathHelper.Square(radius);
```

Structure

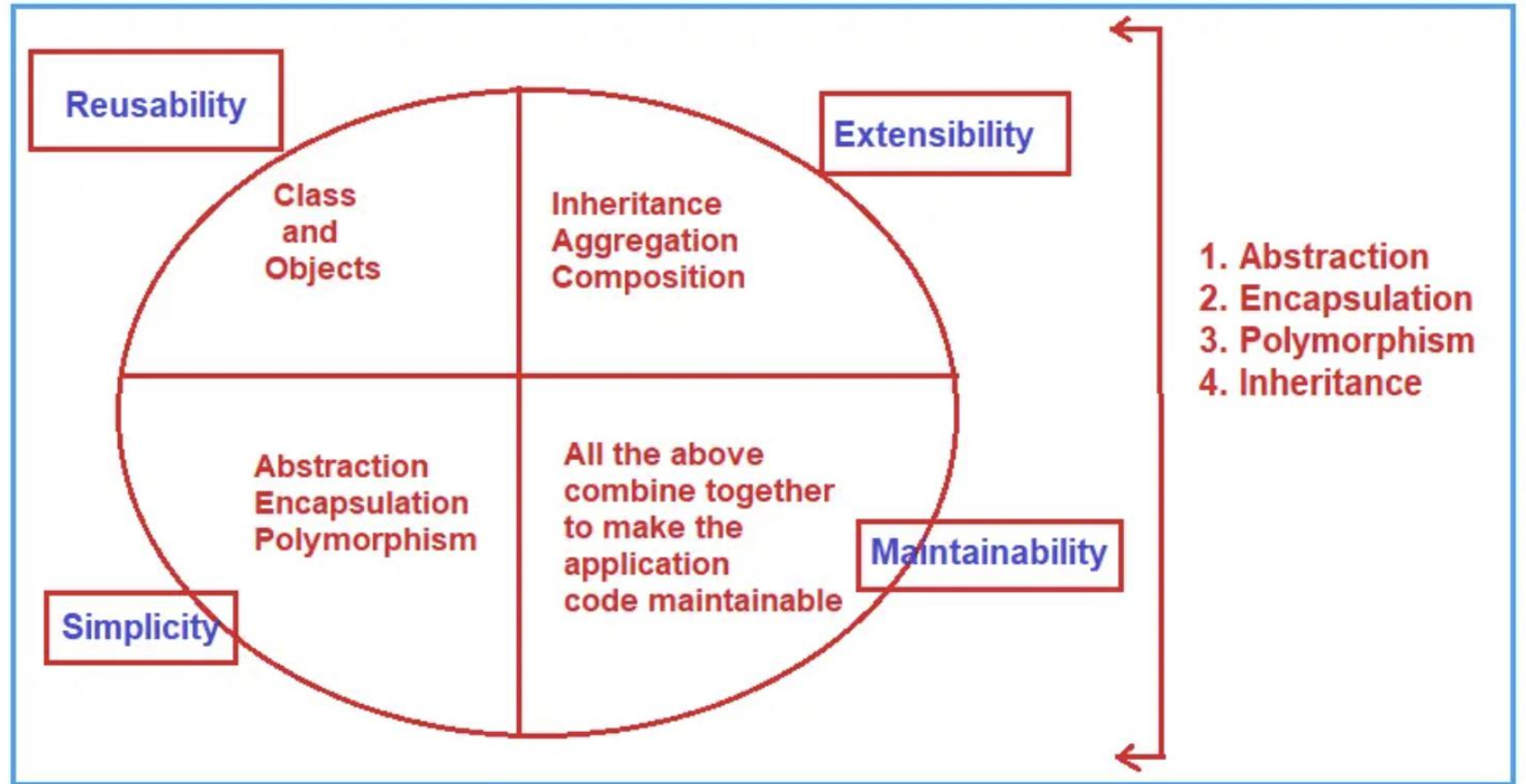
What is a Struct?

A **struct** (structure) in C# is a value type that is useful for creating small data structures that do not require the overhead of classes. Structs are typically used for lightweight objects such as coordinates, complex numbers, or key-value pairs.

Characteristics:

- **Value Type:** Structs are value types, which means they are allocated on the stack and deallocated when they go out of scope. This makes them more efficient in terms of memory usage and garbage collection compared to classes, which are reference types and allocated on the heap.
- **Immutability:** While structs themselves can be mutable, it's a common practice to make them immutable (readonly) to avoid unintended side-effects. This practice is especially useful for ensuring thread safety.
- **No Inheritance:** Structs cannot inherit from other classes or structs. However, they can implement interfaces. This lack of inheritance keeps structs lightweight and straightforward.
- **Default Constructor:** Structs cannot have explicit parameterless constructors. They always have an implicit default constructor that initializes the struct's fields to their default values (e.g., zero for integers, null for objects).

What OOP Solves?



A large orange circle on the left side of the slide.

Pillars of OOP

Inheritance

Abstraction

Polymorphism

Encapsulation



Pillars of OOP

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Inheritance

- Inheritance is a fundamental concept in object-oriented programming that allows you to define a new class based on an existing class. The new class, called the **derived class**, inherits the properties and behaviors (methods) of the existing class, called the **base class**. This promotes code reusability and establishes a relationship between classes.
- Key Concepts:
- **Base Class (Parent Class)**: The class whose properties and methods are inherited by another class.
- **Derived Class (Child Class)**: The class that inherits the properties and methods of the base class.
- **class Keyword**: Used to define a new class.
- **: Symbol**: Used to denote inheritance.

Syntax

// Base Class

```
public class BaseClass
```

```
{
```

```
    // Members (fields, properties, methods, etc.)
```

```
}
```

// Derived Class

```
public class DerivedClass : BaseClass
```

```
{
```

```
    // Additional members (fields, properties, methods, etc.)
```

```
}
```


Types of Inheritance

1. Single Inheritance

Single inheritance is when a class inherits from just one base class. This means that the derived class (child) inherits features from one parent class only. This type of inheritance helps in maintaining simple and easy-to-understand relationships between classes and promotes code reusability.

2. Multilevel Inheritance

Multilevel inheritance occurs when a class is derived from another derived class, creating a chain of inheritance. For example, if Class C inherits from Class B and Class B inherits from Class A, then Class C has properties and methods of both Class B and Class A. This type of inheritance facilitates deeper hierarchical relationships and helps in creating a more structured and organized class system.

3. Hierarchical Inheritance

Hierarchical inheritance happens when multiple classes inherit from a single base class. This means that more than one derived class share the features of a common base class. It promotes code reusability and consistency across related classes and helps in maintaining a common structure among them.

Types of Inheritance

4. Multiple Inheritance (via Interfaces)

Although C# does not support multiple inheritance directly with classes, it achieves it through interfaces. A class can implement multiple interfaces and hence inherit the behavior outlined by those interfaces. This allows classes to have the functionality of multiple inheritances without the complexities associated with it, such as the diamond problem.

Limitations in C#:

No Direct Multiple Inheritance with Classes: To avoid confusion and potential conflicts (like the diamond problem), C# doesn't support direct multiple inheritance with classes.

Hybrid Inheritance: Combining two or more types of inheritance may introduce complications and typically isn't explicitly supported in C#, but can often be achieved through careful design utilizing interfaces and single/multilevel inheritance.

These types of inheritance help in creating structured, modular, and reusable code in object-oriented programming.

Polymorphism

Polymorphism is a fundamental principle in object-oriented programming that allows objects to be treated as instances of their parent class rather than their actual derived class. The term polymorphism means "many shapes," and it empowers a single function, method, or operator to operate in different ways. In C#, polymorphism promotes flexibility and integration within systems, making code more maintainable and extensible.

Types of Polymorphism:

Compile-Time Polymorphism (Static Binding):

Method Overloading: This occurs when multiple methods in the same class have the same name but different parameters (types or number of parameters). It allows a class to have multiple methods with the same name but perform different tasks.

Operator Overloading: This allows an operator to have different implementations depending on the operands.

Run-Time Polymorphism (Dynamic Binding):

Method Overriding: This occurs when a derived class has a method with the same name as a method in its base class but provides a specific implementation. This is typically achieved using the `virtual` and `override` keywords in C#.

Polymorphism

Method Overloading (Compile-Time Polymorphism):

Method overloading allows multiple methods in the same class to have the same name but different signatures (parameter lists). It is resolved at compile-time.

Method Overriding (Run-Time Polymorphism):

Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class. It is resolved at run-time.

The base class method must be marked with the `virtual` keyword, and the derived class method must be marked with the `override` keyword.

Key Concepts:

Virtual Method: A method in a base class that can be overridden in derived classes. It is declared using the `virtual` keyword.

Override Method: A method in a derived class that overrides a base class method. It is declared with the `override` keyword.

Base Keyword: Used to call a base class method from within an overridden method in a derived class.

.

Benefits of Polymorphism

Code Reusability: Allows for the reuse of methods without rewriting them for different data types or classes.

Flexibility: Provides a mechanism to define one interface and have multiple implementations.

Maintainability: Makes the code easier to manage and maintain, as changes in method implementation can be confined to the derived classes.

Polymorphism enables designers to build systems that are more modular, scalable, and easier to troubleshoot and extend.

Abstraction

Abstraction is a core principle of object-oriented programming (OOP) in C#. It helps manage complexity by focusing on the essential features of an object while hiding unnecessary details. Abstraction allows you to define the interface or contract for a type, specifying what it does, without requiring the details of how it accomplishes its tasks. This makes your code easier to understand, maintain, and extend.

Key Concepts:

Abstract Classes:

Definition: An abstract class is a class that cannot be instantiated and is intended to be subclassed. Abstract classes can include abstract methods, which are methods declared without implementation.

Purpose: Abstract classes are used to provide a common base class for derived classes. They allow you to define a template for future classes that derive from the base class.

Usage: Use the `abstract` keyword to declare an abstract class and its methods. Derived classes must implement the abstract methods.

Interfaces

Definition: An interface defines a contract that implementing classes must adhere to. It includes method signatures without implementations.

Purpose: Interfaces are used to specify a set of methods that a class must implement, promoting a consistent design across different classes.

Usage: Use the `interface` keyword to declare an interface. Classes implement interfaces using the `implements` keyword, providing the method definitions outlined in the interface.

Abstract Classes: Use an abstract class when you want to provide a common base with shared functionality and leave some methods to be implemented by derived classes.

Interfaces: Use interfaces when you want to define a contract that different classes can implement, allowing for flexible code design without enforcing shared functionality.

Benefits of Abstraction

Simplifies Complexity: By focusing on the essential features of an object and hiding the details, abstraction simplifies the design and implementation of complex systems.

Enhances Reusability: Abstract classes and interfaces provide a blueprint for creating reusable components that can be extended and customized in derived classes or implementing classes.

Improves Maintainability: Changes made to abstract classes or interfaces automatically propagate to all derived classes or implementing classes, making the code easier to maintain.

Facilitates Flexibility: Abstraction allows for flexible and scalable code design by defining what an object does rather than how it does it.

Encapsulation:

Encapsulation is the mechanism of bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object's components, defining a clear boundary between the object's internal state and its external interface.

Purpose: Encapsulation helps protect the internal state of an object from unintended interference and misuse. It also promotes modularity and reduces code complexity by hiding the implementation details from the users of the object.

Usage: Use access modifiers like `private`, `protected`, and `public` to control the visibility of class members. Provide public methods (getters and setters) to allow controlled access to the class's attributes.

Access Modifiers:

- **Private:** The member is accessible only within the class.
- **Protected:** The member is accessible within the class and by derived class instances.
- **Public:** The member is accessible from any other code.
- **Internal:** The member is accessible within the same assembly, but not from another assembly.
- **Protected Internal:** The member is accessible within its own assembly or to derived types.

Encapsulation:

Properties:

Getters and Setters: Provide controlled access to class attributes. By using properties, you can define how values are read from or assigned to class members, adding an extra layer of validation or logic.

Benefits of Encapsulation:

Data Hiding: Encapsulation hides the internal state of an object from external modification. This protects the object's integrity by preventing unintended interference.

Simplified Interface: Encapsulation provides a simplified interface to interact with the object, making it easier for developers to use without needing to understand the internal details.

Maintainability and Security: By controlling how data is accessed and modified, encapsulation makes the code more maintainable and secure. Changes can be made internally in the class without affecting the external code that interacts with it.

Modularity: Encapsulation helps in breaking down complex systems into simpler, manageable modules. Each class can be developed and tested independently.