# 2. C# Data Types and Variables

## Understanding Data Types

In C#, every variable has a data type, specifying the kind of data it can hold. The common data types in C# include:

- **Integer Types:** int, long, short, byte
- **Floating-Point Types:** float, double
- **Decimal Type:** decimal
- **Character Type:** char
- **Boolean Type:** bool
- **String Type:** string

Choosing the right data type is crucial for efficient memory usage and accurate representation of data.

## Declaring Variables

Variables are used to store data in a program. In C#, you declare a variable by specifying its data type and a name.

**For example:**

```
int age;
float price;
string name;
```

## Initializing Variables

Variables can be initialized (assigned an initial value) at the time of declaration or later in the program.

**Here's an example of both:**

```
int quantity = 10; // Initialization at declaration
float temperature; // Declaration without initialization
temperature = 98.6f; // Initialization later in the program
```

## Type Inference (var keyword)

C# supports type inference using the var keyword, allowing the compiler to automatically determine the data type based on the assigned value.

**For example:**

```
var count = 5; // Compiler infers int
var price = 19.99; // Compiler infers double
```

## Constants

Constants are variables whose values cannot be changed once assigned. They are declared using the const keyword:

```
const double PI = 3.14159;
```

## Conversion between Data Types

C# supports implicit and explicit conversion between compatible data types. Implicit conversion is done automatically, while explicit conversion requires the use of casting.

**For example:**

```
int numInt = 10;
double numDouble = numInt; // Implicit conversion
int newInt = (int)numDouble; // Explicit conversion (casting)
```

## Nullable Types

In C#, value types cannot be assigned a value of null. However, by using nullable types, you can explicitly allow a value type to be null.

**For example:**

```
int? nullableInt = null;
```

Understanding C# data types and variables is fundamental for writing robust and efficient programs. As you progress, you'll encounter situations where choosing the right data type and managing variables effectively are critical for successful application development.

# 3. Operators and Expressions in C#

## Introduction to Operators

Operators in C# are symbols that perform operations on variables and values. Understanding these operators is essential for manipulating data and performing calculations in your programs.

## Arithmetic Operators

Arithmetic operators are used for basic mathematical operations. They include:

- **Addition (+):** Adds two operands.
- **Subtraction (-):** Subtracts the right operand from the left operand.
- **Multiplication (*):** Multiplies two operands.
- **Division (/):** Divides the left operand by the right operand.
- **Modulus (%):** Returns the remainder of the division.

```csharp
int a = 10;
int b = 3;
int sum = a + b; // 13
int difference = a - b; // 7
int product = a * b; // 30
int quotient = a / b; // 3
int remainder = a % b; // 1
```

## Comparison Operators

Comparison operators are used to compare values.

**They include:**

- **Equal to (==):** Checks if two operands are equal.

- **Not equal to (!=):** Checks if two operands are not equal.
- **Greater than (>):** Checks if the left operand is greater than the right operand.
- **Less than (<):** Checks if the left operand is less than the right operand.
- **Greater than or equal to (>=):** Checks if the left operand is greater than or equal to the right operand.
- **Less than or equal to (<=):** Checks if the left operand is less than or equal to the right operand.

```
int x = 5;
int y = 8;
bool isEqual = x == y; // false
bool isNotEqual = x != y; // true
bool isGreaterThan = x > y; // false
bool isLessThan = x < y; // true
bool isGreaterOrEqual = x >= y; // false
bool isLessOrEqual = x <= y; // true
```

## Logical Operators

Logical operators are used to perform logical operations.

They include:

**Logical AND (&&):** Returns true if both operands are true.
**Logical OR (||):** Returns true if at least one operand is true.
**Logical NOT (!):** Returns true if the operand is false, and vice versa.

```
bool condition1 = true;
bool condition2 = false;

bool resultAnd = condition1 && condition2; // false
bool resultOr = condition1 || condition2; // true
bool resultNot = !condition1; // false
```

## Assignment Operators

Assignment operators are used to assign values to variables.
**They include:**

**Assignment (=):** Assigns the value on the right to the variable on the left.
Addition and assignment (+=): Adds the right operand to the variable and assigns the result to the variable.

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

## Increment and Decrement Operators

Increment and decrement operators are used to increase or decrease the value of a variable by 1.

- **Increment (++):** Increases the value of the variable by 1.
- **Decrement (--):** Decreases the value of the variable by 1.

```
int counter = 5;
counter++; // counter is now 6
counter--; // counter is now 5
```

## Conditional Operator (Ternary Operator)

The conditional operator (? :) is a shorthand way of writing an if-else statement. It returns one of two values based on the evaluation of a condition.

```
int a = 10;
int b = 5;
int max = (a > b) ? a : b; // max is 10
```

Operators are fundamental building blocks for performing operations in C#. By mastering these operators, you gain the ability to create complex expressions, make decisions in your code, and manipulate variables effectively. As you proceed, you'll find these concepts crucial for creating efficient and functional programs.

# 4. Control Structures (if-else, loops)

Control structures in C# are used to manage the flow of execution in a program. They allow you to make decisions and repeat actions based on conditions. Two primary control structures are if-else statements and loops.

## If-Else Statements

If-else statements are used for conditional execution. They allow your program to take different paths based on whether a certain condition is true or false.

**If Statement**

```
int num = 10;

if (num > 0)
{
    Console.WriteLine("The number is positive.");
}
```

**If-Else Statement**

```
int num = -5;

if (num > 0)
{
    Console.WriteLine("The number is positive.");
}
else
{
    Console.WriteLine("The number is non-positive.");
}
```

**If-Else If-Else Statement**

```
int num = 0;

if (num > 0)
{
    Console.WriteLine("The number is positive.");
}
else if (num < 0)
{
    Console.WriteLine("The number is negative.");
}
else
{
    Console.WriteLine("The number is zero.");
}
```

## Loops

Loops allow you to execute a block of code repeatedly. There are several types of loops in C#.

**For Loop:** The for loop is used when the number of iterations is known.

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine($"Iteration {i}");
}
```

**While Loop:** The while loop is used when the number of iterations is not known in advance.

```
int count = 0;

while (count < 3)
{
    Console.WriteLine($"Count: {count}");
    count++;
}
```

**Do-While Loop:** The do-while loop is similar to the while loop but guarantees that the loop body is executed at least once.

```
int attempt = 0;

do
{
    Console.WriteLine($"Attempt: {attempt}");
    attempt++;
} while (attempt < 3);
```

**Foreach Loop:** The foreach loop is used to iterate over elements in a collection (e.g., arrays or lists).

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

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

**Nested Control Structures:** You can nest control structures within each other to create more complex logic.

```
int x = 10;
int y = 5;

if (x > 0)
{
    if (y > 0)
    {
        Console.WriteLine("Both x and y are positive.");
    }
    else
    {
        Console.WriteLine("x is positive, but y is non-positive.");
    }
}
else
{
    Console.WriteLine("x is non-positive.");
}
```

Control structures are crucial for determining the flow of your C# programs. Mastering if-else statements and loops empowers you to create dynamic and responsive applications.

# 5. Methods and Functions

In C#, methods are blocks of code that perform a specific task and can be called from other parts of the program. Methods promote code reuse, readability, and modularization.

## Declaring and Calling Methods

### Method Declaration

```
public class Calculator
{
    // Method declaration
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

### Calling a Method

```
Calculator calculator = new Calculator();
int result = calculator.Add(3, 5);
Console.WriteLine(result); // Output: 8
```

## Method Parameters

### Required Parameters

```
public class MathOperations
{
    public int Multiply(int x, int y)
    {
        return x * y;
    }
}
```

### Optional Parameters

```csharp
public class Printer
{
    // Optional parameter with a default value
    public void PrintMessage(string message, bool uppercase = false)
    {
        if (uppercase)
        {
            Console.WriteLine(message.ToUpper());
        }
        else
        {
            Console.WriteLine(message);
        }
    }
}
```

## Return Types

### Methods with Return Values

```csharp
public class Circle
{
    public double CalculateArea(double radius)
    {
        return Math.PI * radius * radius;
    }
}
```

### Void Methods

```csharp
public class Logger
{
    // Void method (no return value)
    public void LogError(string errorMessage)
    {
        Console.WriteLine($"Error: {errorMessage}");
    }
}
```

## Method Overloading

Method overloading allows you to define multiple methods with the same name but different parameter lists.

```
public class OverloadedMethods
{
    public int Add(int a, int b)
    {
        return a + b;
    }

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

## Recursive Methods

A method can call itself, creating a recursive function.

```
public class FactorialCalculator
{
    public int CalculateFactorial(int n)
    {
        if (n == 0 || n == 1)
        {
            return 1;
        }
        else
        {
            return n * CalculateFactorial(n - 1);
        }
    }
}
```

Methods and functions play a pivotal role in structuring C# programs. They allow you to encapsulate logic, promote reusability, and enhance the readability of your code. As you continue your C# journey, mastering the art of method declaration, parameter handling, and understanding return types will empower you to design robust and modular applications.

# 6. Classes and Objects

In C#, classes are the building blocks of object-oriented programming. A class is a blueprint for creating objects, and objects are instances of classes. This paradigm allows you to model real-world entities and their behaviors in your programs.

## Class Declaration

### Basic Class Structure

```csharp
public class Car
{
    // Fields (attributes)
    public string Model;
    public string Color;

    // Constructor
    public Car(string model, string color)
    {
        Model = model;
        Color = color;
    }

    // Methods (behaviors)
    public void Start()
    {
        Console.WriteLine($"{Color} {Model} is starting.");
    }

    public void Drive()
    {
        Console.WriteLine($"{Color} {Model} is on the move.");
    }
}
```

## Creating Objects

### Object Instantiation

```
class Program
{
    static void Main()
    {
        // Creating objects
        Car myCar = new Car("Toyota", "Blue");
        Car anotherCar = new Car("Honda", "Red");

        // Accessing fields
        Console.WriteLine($"My car is a {myCar.Color} {myCar.Model}.");

        // Calling methods
        myCar.Start();
        anotherCar.Drive();
    }
}
```

## Constructors

Constructors are special methods called when an object is created. They initialize the object's state.

### Default Constructor

```
public class Person
{
    public string Name;

    // Default constructor
    public Person()
    {
        Name = "John Doe";
    }
}
```

**Parameterized Constructor**

```
public class Book
{
    public string Title;
    public string Author;

    // Parameterized constructor
    public Book(string title, string author)
    {
        Title = title;
        Author = author;
    }
}
```

## Properties

Properties provide controlled access to the fields of a class.

```
public class Student
{
    // Private field
    private int _age;

    // Public property with get and set accessors
    public int Age
    {
        get { return _age; }
        set
        {
            if (value >= 0 && value <= 120)
            {
                _age = value;
            }
            else
            {
                Console.WriteLine("Invalid age value.");
            }
        }
    }
}
```

## Encapsulation

Encapsulation is the practice of bundling the data (fields) and methods that operate on the data within a single unit (class).

```
public class BankAccount
{
    private decimal _balance;

    public void Deposit(decimal amount)
    {
        _balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= _balance)
        {
            _balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }

    public decimal GetBalance()
    {
        return _balance;
    }
}
```

## Inheritance

Inheritance allows a class (subclass) to inherit the properties and behaviors of another class (base class).

```
 public class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

public class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}
```

Understanding classes and objects is fundamental to object-oriented programming in C#. Classes encapsulate data and behavior, providing a clean and modular way to structure your code. As you delve deeper into OOP concepts, you'll find that classes and objects form the basis for creating scalable, maintainable, and reusable software solutions.

# 7. Inheritance and Polymorphism in C#

Inheritance and polymorphism are fundamental concepts in object-oriented programming that enable code reuse, extensibility, and flexibility. In C#, classes can inherit from other classes, and objects can exhibit different forms through polymorphism.

## Inheritance

Inheritance allows a class (derived or child class) to inherit properties and behaviors from another class (base or parent class).

### Base Class (Parent Class)

```
public class Animal
{
    public string Species { get; set; }

    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}
```

### Derived Class (Child Class)

```
public class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}
```

### Accessing Base Class Members

```
Dog myDog = new Dog();
myDog.Species = "Canine";
myDog.Eat(); // Accessing the Eat method from the base class
myDog.Bark();
```

## Polymorphism

Polymorphism allows objects to be treated as instances of their base class, even if they are actually instances of derived classes.

### Method Overriding

```
public class Cat : Animal
{
    // Method overriding
    public override void Eat()
    {
        Console.WriteLine("Cat is eating.");
    }
}
```

### Using Polymorphism

```
Animal myAnimal = new Dog();
myAnimal.Eat(); // Calls the overridden Eat method in Dog class

myAnimal = new Cat();
myAnimal.Eat(); // Calls the overridden Eat method in Cat class
```

### Virtual and Override Keywords

```
public class Shape
{
    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape.");
    }
}

public class Circle : Shape
```

```
{
    // Override method
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }
}
```

## Abstract Classes and Methods

```
public abstract class Shape
{
    // Abstract method
    public abstract void Draw();
}

public class Circle : Shape
{
    // Implementation of the abstract method
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }
}
```

Inheritance and polymorphism are powerful concepts that enhance the structure and flexibility of your C# code. Leveraging these features allows you to build hierarchies of classes, create reusable code, and design systems that can adapt to changing requirements.

# 8. Interfaces and Abstract Classes

Interfaces and abstract classes are key components of object-oriented programming in C#. They provide a way to define contracts for classes, enforcing common behavior and ensuring consistency in your codebase.

## Interfaces

An interface is a contract that defines a set of methods and properties that a class must implement. It allows for multiple inheritance, enabling a class to implement multiple interfaces.

### Declaring an Interface

```
public interface ILogger
{
    void LogMessage(string message);
    void LogError(string errorMessage);
}
```

### Implementing an Interface

```
public class ConsoleLogger : ILogger
{
    public void LogMessage(string message)
    {
        Console.WriteLine($"Message: {message}");
    }

    public void LogError(string errorMessage)
    {
        Console.WriteLine($"Error: {errorMessage}");
    }
}
```

### Using Interfaces

```
ILogger logger = new ConsoleLogger();
```

```
logger.LogMessage("Informational message");
logger.LogError("Critical error");
```

## Abstract Classes

An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) that must be implemented by derived classes.

### Declaring an Abstract Class

```
public abstract class Shape
{
    // Abstract method
    public abstract void Draw();

    // Regular method
    public void Move()
    {
        Console.WriteLine("Moving the shape.");
    }
}
```

### Implementing an Abstract Class

```
public class Circle : Shape
{
    // Implementing the abstract method
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }
}
```

### Abstract Properties

```
public abstract class Animal
{
    // Abstract property
    public abstract string Species { get; }

    // Regular method
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}
```

**Using Abstract Classes**

```
Shape myShape = new Circle();
myShape.Draw(); // Calls the overridden Draw method in Circle class
myShape.Move(); // Calls the Move method from the abstract class

Animal myAnimal = new Dog();
Console.WriteLine($"Species: {myAnimal.Species}"); // Calls the overridden
property in Dog class
myAnimal.Eat(); // Calls the Eat method from the abstract class
```

## When to Use Interfaces and Abstract Classes

**Interfaces:** Use when you want to define a contract that multiple classes can implement, regardless of their inheritance hierarchy.

**Abstract Classes:** Use when you want to provide a common base class with shared functionality and enforce that derived classes implement certain methods.

Interfaces and abstract classes are crucial tools for designing flexible and maintainable C# code. They allow you to create contracts, establish common behaviors, and structure your code in a way that promotes extensibility and reusability. As you explore more advanced topics in C#, you'll find these concepts playing a vital role in creating well-organized and scalable software systems.

# 9. Exception Handling

Exception handling is a critical aspect of writing robust and reliable code. It allows you to gracefully handle errors and unexpected situations that may arise during the execution of your C# programs.

## Understanding Exceptions

In C#, an exception is a runtime error that occurs during the execution of a program. When an exception occurs, it disrupts the normal flow of the program and can be caught and handled by appropriate code.

## Try-Catch Blocks

The try-catch block is used to handle exceptions. Code that might throw an exception is placed inside the try block, and the corresponding exception-handling code is placed inside the catch block.

```
try
{
    // Code that may throw an exception
    int result = 10 / int.Parse("0");
}
catch (DivideByZeroException ex)
{
    // Handling specific exception
    Console.WriteLine($"Error: {ex.Message}");
}
catch (Exception ex)
{
    // Handling any other exception
    Console.WriteLine($"An unexpected error occurred: {ex.Message}");
}
finally
{
    // Code that will be executed regardless of whether an exception occurred
    Console.WriteLine("This code always executes, even if an exception
occurred.");
}
```

## Throwing Exceptions

You can manually throw exceptions using the throw keyword. This is useful when you want to indicate that a certain condition or error has occurred.

```
public class Calculator
{
    public int Divide(int dividend, int divisor)
    {
        if (divisor == 0)
        {
            throw new DivideByZeroException("Cannot divide by zero.");
        }

        return dividend / divisor;
    }
}
```

## Custom Exceptions

You can create your own custom exception classes by deriving from the Exception class. This allows you to define specific exception types for your application.

```
public class CustomException : Exception
{
    public CustomException(string message) : base(message)
    {
    }
}
```

## Exception Filters

Exception filters allow you to catch exceptions based on a specific condition. This feature was introduced in C# 6.0.

```
try
{
    // Code that may throw an exception
    int result = 10 / int.Parse("0");
}
catch (DivideByZeroException ex) when (ex.Message == "Attempted to divide by 
zero.")
{
    // Handling specific exception with a filter
    Console.WriteLine($"Error: {ex.Message}");
}
```

## Global Exception Handling

In larger applications, you may implement global exception handling to catch unhandled exceptions and log or display information about them.

```
static void Main()
{
    AppDomain.CurrentDomain.UnhandledException += GlobalExceptionHandler;

    // Rest of the application code
}

private static void GlobalExceptionHandler(object sender, 
UnhandledExceptionEventArgs e)
{
    Exception exception = (Exception)e.ExceptionObject;
    Console.WriteLine($"Global Exception Handler: {exception.Message}");
}
```

Exception handling is a crucial part of writing robust and reliable C# applications. By using try-catch blocks, throwing exceptions when necessary, and implementing appropriate error-handling strategies, you can ensure that your code gracefully handles unexpected situations, improving the overall stability of your software.

# 10. C# Arrays and Lists

Arrays and lists are fundamental data structures in C# that allow you to store and manipulate collections of elements. Understanding their usage is essential for working with structured data in your programs.

## Arrays

**Declaration and Initialization:** Arrays in C# are fixed-size collections of elements of the same data type.

```
// Declaration and Initialization
int[] numbers = new int[5] { 1, 2, 3, 4, 5 };
```

### Accessing Elements

```
// Accessing Elements
int firstElement = numbers[0]; // Accessing the first element (index 0)
```

### Iterating Through Arrays

```
// Iterating Through Arrays
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

### Multidimensional Arrays

```
// Multidimensional Arrays
int[,] matrix = new int[3, 3]
{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

## Lists

**Declaration and Initialization:** Lists provide dynamic arrays with additional functionality.

```
// Declaration and Initialization
List<string> names = new List<string>() { "Alice", "Bob", "Charlie" };
```

### Adding and Removing Elements

```
// Adding and Removing Elements
names.Add("David"); // Add an element
names.Remove("Bob"); // Remove an element by value
names.RemoveAt(0); // Remove an element by index
```

### Accessing Elements

```
// Accessing Elements
string firstElement = names[0]; // Accessing the first element (index 0)
```

### Iterating Through Lists

```
// Iterating Through Lists
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

## Array vs. List

**Arrays:** Fixed size, less flexibility, direct access by index.
**Lists:** Dynamic size, more functionality, slower access by index.

### LINQ with Lists

LINQ (Language Integrated Query) provides powerful querying capabilities for collections.

```
// LINQ with Lists
var result = names.Where(n => n.Length > 4).ToList();
```