

C# Language

1

Agenda

- Variables and Data Types
- Operators
- Conditionals
- Loops
- Arrays and Collections
- Fields, Properties, Methods
- Class, Abstract Class, Interfaces
- Inheritance
- Enum and Constants
- Exception Handling
- Struct, Record, Tuple
- Delegates and Events
- Generics
- Asynchronous Methods
- Top-level Statements
- Extension Methods
- Lambda Expressions
- Pattern Matching

2



3

Test your knowledge and skills

- Q1. Correct the method below so that it returns the average of two integers a and b.


```
double Average(int a, int b)
{
    return a + b / 2;
}
```
- Q2. Write a method that takes a positive integer as input and returns 1 if it is odd. Otherwise, returns 0.
- Q3. Given an array of positive integers. Write a method to compute sum of odd numbers.
 - Case 1: No constraints/restrictions
 - Case 2: Do not use if statement, ternary operator, modulus operator

3



4

Variables and Data Types

- Simple data types:
 - Signed: `sbyte`, `short`, `int`, `long`
 - Unsigned: `byte`, `ushort`, `uint`, `ulong`
 - Floating: `float`, `double`, `decimal`
 - Others: `char`, `bool`
- Declaration


```
DataTypeName variableName;
```
- Assignment


```
myAge = 38;
int myAge = 38;
float temperature = 22.5f, humidity = 70.5f;
```

4



5

Operators

- Arithmetic Operators

```
float x = 3 + 2; // addition (5)
x = 3 - 2; // subtraction (1)
x = 3 * 2; // multiplication (6)
x = 3 / 2; // division (1)
x = 3 % 2; // modulus (1)
```

Assignment Operators

```
i += 5; // i = i+5;
i -= 5; // i = i-5;
i *= 5; // i = i*5;
i /= 5; // i = i/5;
i %= 5; // i = i%5;
```

- Increment & Decrement Operators

```
x++; // x = x+1;      ++x; // pre-increment
x--; // x = x-1;      --x; // pre-decrement
```

5



6

Operators

- Comparison Operators

```
bool b = (2 == 3); // equal to (false)
b = (2 != 3); // not equal to (true)
b = (2 > 3); // greater than (false)
b = (2 < 3); // less than (true)
b = (2 >= 3); // greater than or equal to (false)
b = (2 <= 3); // less than or equal to (true)
```

Bitwise Operators

```
int x = 5 & 4; // and (0b101 & 0b100 = 0b100 = 4)
x = 5 | 4; // or (0b101 | 0b100 = 0b101 = 5)
x = 5 ^ 4; // xor (0b101 ^ 0b100 = 0b001 = 1)
x = 4 << 1; // left shift (0b100 << 1 = 0b1000 = 8)
x = 4 >> 1; // right shift (0b100 >> 1 = 0b10 = 2)
x = ~4; // invert (~0b00000100 = 0b111111011 = -5)
```

- Logical Operators

```
bool b = (true && false); // logical and (false)
b = (true || false); // logical or (true)
b = !(true); // logical not (false)
```

6



7

Strings

- Example: `string hi = "Hello", name = "Everyone";`
- String concatenation:
 - `hi + " " + name`
 - `string.Format("{0} {1}", hi, name)`
- String interpolation (C# 6):
 - `($"{hi} {name}")`
- String compare:
 - `"Hello" == "hello"`
 - `"Hello".CompareTo("hello")`
 - `string.Compare("Hello", "hello", StringComparison.OrdinalIgnoreCase)`

7



8

Arrays

- Declaration: `int[] numbers;`
- Allocation: `int[] numbers = new int[3];`
- Initialization:
 - `int[] numbers = new int[] { 1, 3, 5};`
 - `int[] numbers = { 1, 3, 5};`
- Accession: `numbers[1]`
- Rectangular Array:
 - `double[,] matrix = new double[3, 5];`
- Jagged Array:
 - `string[][] namesMap = new string[2][];`

8

Exercises

- What type would you choose for the following “numbers”?
 1. A person’s telephone number
 2. A person’s height
 3. A person’s age
 4. A person’s salary
 5. A book’s ISBN
 6. A book’s price
 7. A book’s shipping weight
 8. A country’s population
 9. The number of stars in the universe
 10. The number of employees in each of the small or medium businesses in the United Kingdom (up to about 50,000 employees per business)

9

Conditionals

- If statement:

```
if (x < 1) {
    System.Console.Write(x + " < 1");
}
```

```
if (x < 1)
    System.Console.Write(x + " < 1");
else if (x > 1)
    System.Console.Write(x + " > 1");
```

- Switch statement:

```
switch (x)
{
    case 0: System.Console.Write(x + " is 0"); break;
    case 1: System.Console.Write(x + " is 1"); break;
    default: System.Console.Write(x + " is >1"); break;
}
```

Switch Expression: (C# 8)

```
string result = x switch {
    0 => "zero",
    1 => "one",
    _ => "more than one"
};
```

10



11

Loops

- While loop:

```
int i = 0;
while (i < 10) {
    System.Console.Write(i++);
}
```

- Do-While loop:

```
int j = 0;
do {
    System.Console.Write(j++);
} while (j < 10);
```

- For loop:

```
for (int k = 0; k < 10; k++) {
    System.Console.Write(k); // 0-9
}
```

- Foreach loop:

```
int[] a = { 1, 2, 3 };
foreach (int m in a) {
    System.Console.Write(m);
}
```

11



12

Methods

- Simple data types:

```
access-modifier return-type method-name(parameters)
{
    ...
    [return value;]
}
```

- Example:

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

12



13

Methods

- Params keyword

```
void Print(params string[] s)
{
    foreach (string x in s)
        System.Console.WriteLine(x);
}
```

- Optional parameters (C# 4)

```
void Sum(int i, int j = 0, int k = 0)
{
    System.Console.WriteLine(1*i + 2*j + 3*k);
}
```

13



14

Methods

- Named arguments (C# 4)

```
static void Main()
{
    new MyApp().Sum(1, k: 2); // 7
}
static void Main()
{
    new MyApp().Sum(i: 2, 1); // 4
}
```

14



15

Local Methods (C# 7)

```
static bool IsWordPalindrome(string word)
{
    if (word == null)
        throw new ArgumentNullException(nameof(word));

    if (word.Length < 2) return true;

    return IsPalidrome(0, word.Length - 1);

    bool IsPalidrome(int lo, int hi) {
        if (lo >= hi) return true;

        if (char.ToUpperInvariant(word[lo]) != char.ToUpperInvariant(word[hi]))
            return false;

        return IsPalidrome(lo + 1, hi - 1);
    }
}
```

15



16

Exercises

- Q1. Correct the method below so that it returns the average of two integers a and b.


```
static double Average(int a, int b)
{
    return a + b / 2;
}
```
- Q2. Write a method that takes a positive integer as input and returns 1 if it is odd. Otherwise, returns 0.
- Q3. Given an array of positive integers. Write a method to compute sum of odd numbers.
 - Case 1: No constraints
 - Case 2: Do not use if statement, ternary operator, modulus operator

16

Classes and Objects

17

Class

- A class is a template used to create objects.
- Classes are made up by two main members: *fields* and *methods*.

```
class Rectangle
{
    // Make members accessible for instances of the class
    public int x, y;
    public int GetArea() { return x * y; }
}
```

- Object creation:
 - `Rectangle r = new Rectangle();`
 - `Rectangle r = new ();` // C# 9

18

Class

- Constructors:

```
class Rectangle
{
    public int x, y;
    public Rectangle() { x = 10; y = 5; }
    public Rectangle(int a) { x = a; y = a; }
    public Rectangle(int a, int b) { x = a; y = b; }
}

class Rectangle
{
    public int x, y;
    public Rectangle() : this(10, 5) {}
    public Rectangle(int a) : this(a, a) {}
    public Rectangle(int a, int b) { x = a; y = b; }
}
```

Default constructor

Constructor overloading

Constructor chaining

19

Object

- Object initializers

```
Rectangle r1 = new Rectangle { x = 0, y = 0 };
Rectangle r2 = new() { x = 0, y = 0 };
```

- Access object members:

```
Rectangle r = new Rectangle();
r.x = 10;
r.y = 5;
int a = r.GetArea(); // 50
```

20

Properties

- Provide the ability to protect a field.
- Allow developers to change the internal implementation of the property without breaking any programs that are using it.
- Allow the data to be validated before permitting a change.

```
class Time
{
    private int sec;

    public int Seconds
    {
        get { return sec; }
        set
        {
            if (value > 0) sec = value;
            else sec = 0;
        }
    }
}
```

accessors

21

Properties

- Read-only and Write-only properties

```
// Read-only property
public int Seconds
{
    get { return sec; }
}
```

```
// Write-only property
public int Minutes
{
    set { min = value; }
}
```

- Auto-implemented properties (C# 3)

```
class Time
{
    private int sec;

    public int Seconds
    {
        get { return sec; }
        set { sec = value; }
    }
}
```

```
class Time
{
    public int Seconds { get; set; }
}
```

22

Properties

- C# 6: An initial value can be set as part of the declaration

```
class Time
{
    // Read-only auto-property with initializer
    public DateTime CreatedTime { get; } = DateTime.Now;
}
```

- C# 9: The `set` accessor can be replaced by an `init` accessor

```
class Time
{
    public int Seconds { get; init; }
}

Time t = new Time() { Seconds = 5 }
t.Seconds = 5; // Error: Seconds not settable
```

23

Type Inference

- `var` keyword: (C# 3)

```
class Example {}
var o = new Example(); // implicit type
Example o = new Example(); // explicit type
```

- Target-typed new expression (C# 9)

```
Example a = new();
var b = new(); // error: no target type
```

24



25

Anonymous Types

- A type created without an explicitly defined class.
- Provide a concise way to form a temporary object that is only needed within the local scope and therefore should not be visible elsewhere.

```
var v = new { first = 1, second = true };
System.Console.WriteLine(v.first); // "1"
```

- `var` keyword is required
- Field types are automatically determined by the compiler
- Fields are read-only

25



26

Null and Nullable Types

- `null` keyword represents a null reference
- C# 7 and earlier: null was used for reference types only.

```
string s = null; // warning as of C# 8
int length = s.Length; // error: NullReferenceException
```

- C# 8 introduces nullable types which is created by appending a question mark (?) to the original non-nullable type.

```
string? s1 = null; // nullable reference type
string s2 = ""; // non-nullable reference type
```

26



27

Null and Nullable Types

- Nullable value types: allow the simple types and struct types to indicate an undefined value.

- `bool?` : true, false, null
- `DateTime?`: normal date time value and null

- Null-coalescing Operator (??)

```
int? i = null;           int? i = null;
int j = i ?? 0; // 0      int j = (int)i; // error
```

- Null-coalescing Assignment Operator (C# 8)

```
int? i = null;
i ??= 3; // assign i=3 if i==null
// same as i = i ?? 3;
```

27



28

Null and Nullable Types

- Null-Conditional Operator (?.) (C# 6)

```
string s = null;
int? length = s?.Length; // null

string[] s = null;
string s3 = s?[3]; // null
```

- Combining with the null-coalescing operator

```
string s = null;
int length = s?.Length ?? 0; // 0
```

28

Null and Nullable Types

- Null-Forgiving Operator (!) (C# 8)

```
string s1 = null; // warning: non-nullable type  
string s2 = null!; // warning suppressed
```

29

Inheritance

30



31

Inheritance

- Inheritance allows a class to acquire the members of another class.
- A class in C# may only inherit from one base class.
- System.Object (object) class is the root class of all other classes.

```
// Base class (parent class)
class Rectangle
{
    public int w = 1, h = 1;
    public int GetArea() { return w * h; }
}

// Derived class (child class)
class Square : Rectangle { }
```

31



32

Downcast and Upcast

```
Square s = new Square(); s: Square
Rectangle r = s;          // upcast r: Square
Square s2 = (Square)r;    // downcast s2: Square
```

```
Rectangle r = new Rectangle(); r: Rectangle
Square s = (Square)r; InvalidCastException // <== Error
```

32



33

Boxing and Unboxing

- **Boxing**: a variable of value type to be **implicitly** converted into a reference type of the Object class.

```
int number = 10;    number: 10
object ob = number;    // boxing ob: 10
```

- **Unboxing**: converts the boxed value back into a variable of its value type. The unboxing operation must be **explicit**.

```
int num = (int)ob;    // unboxing num: 10
Square square = (Square)ob;    // <== error InvalidCastException
```

33



34

is and as keywords

- **is operator**: returns true if the left side object can be cast to the right side type without causing an exception.

```
Rectangle q = new Square();    q: Square
if (q is Square)
{
    Square o = (Square)q;    o: Square
} // condition is true
```

- **as operator**: provides an alternative way of writing an explicit cast to avoid object casting exceptions.

```
Rectangle r = new Rectangle();
Square o = r as Square;
// invalid cast, returns null
```

34



35

Pattern Matching (C# 7)

- Test a variable's type and, upon validation, assign it to a new variable of that type.
- Pros:
 - Safely casting variables between types
 - More convenient syntax
 - Replaces the use of the as operator

```
Rectangle q = new Square();
if (q is Square mySquare)
{
    /* use mySquare here */
} q: Square mySquare: Square
```

```
void DrawShape(Shape shape)
{
    switch (shape)
    {
        case Square s: /* ... */ break;
        case Rectangle r: /* ... */ break;
        case Circle c: /* ... */ break;
    }
}
```

35



36

Pattern Matching

- As of C# 9, patterns can include logical operators – and, or, and not – as well as relational operators.

```
bool IsLetter(char c)
{
    return c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
}

string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(
        nameof(date),
        $"Date with unexpected month: {date.Month}."),
};
```

36



37

Redefining Members

- Often used to give instance methods new implementation.
- To give a method a new implementation, the method is redefined in the child class with the same signature as it has in the base class.
- The signature includes:
 - the method name,
 - parameters, and
 - return type of the method

37



38

Hiding Members

- Default behavior
- Use the `new` keyword
- Allows access to the previously hidden method defined in the base class by upcasting.

```
Square s = new Square(); s: Square
Rectangle r = s; r: Square
int p = r.GetPerimeter(); p: 22
```

```
class Rectangle
{
    public int x = 1, y = 10;
    public int GetPerimeter()
    {
        return (x + y) * 2;
    }
}
class Square : Rectangle
{
    public new int GetPerimeter()
    {
        return x * 4;
    }
}
```

38



39

Overriding Members

- Must add the keyword **virtual** to the method in the base class.
- Must add the keyword **override** to the method in the derived class.
- The upcast will still call the version defined in the child class

```
Square s = new Square(); s: Square
Rectangle r = s; r: Square
int p = r.GetPerimeter(); p: 4
```

```
class Rectangle
{
    public int x = 1, y = 10;
    public virtual int GetPerimeter()
    {
        return (x + y) * 2;
    }
}
class Square : Rectangle
{
    public override int GetPerimeter()
    {
        return x * 4;
    }
}
```

39



40

Sealed and Base Keywords

- **sealed:**
 - To stop an overridden method from being further overridden
 - To prevent any class from inheriting a given class
- **base:**
 - To access a parent class's members
 - To call a base class constructor

```
class Square : Rectangle
{
    public sealed override int GetPerimeter()
    {
        return x * 4;
    }
}

sealed class NonInheritable {}

class Square : Rectangle
{
    public Square() : base() { }
    public override int GetPerimeter()
    {
        return base.x * 4;
    }
}
```

40



41

Inner Classes

```
class MyClass
{
    // Inner classes (nested classes)
    public class PublicClass {}

    protected internal class ProtIntClass {}

    private protected class PrivProtClass {}

    internal class InternalClass {}

    protected class ProtectedClass {}

    private class PrivateClass {}
}
```

- Classes may contain inner classes (or nested classes)
- By default, inner classes are private
- If the class is inaccessible, it cannot be instantiated or inherited.

41



42

Static Classes

- Only contains static members and constant fields.
- Cannot be inherited or instantiated into an object.
- Extension methods must be defined in static classes

```
public static class Math
{
    private static double doubleRoundLimit = 1e16d;

    public const double PI = 3.14159265358979323846;
    public const double E = 2.7182818284590452354;

    public static Decimal Floor(Decimal d)
    {
        return Decimal.Floor(d);
    }

    // Other static methods
}
```

42



43

Static Members

- Static members can be used by other classes without having to create an instance of the class.
- Static fields:
 - When only a single instance of the variable is needed.
 - Field value is persisted throughout the life of the application.
 - Often used to implement Singleton pattern
- Static methods:
 - Perform a generic function that is independent of any instance variables.
 - Extension methods must be static
- Static constructor:
 - Perform any actions needed to initialize a class
 - Will only be run once (occurs automatically), cannot be call directly

43



44

Extension Methods (C# 3)

- Provide a way to seemingly add new instance methods to an existing class outside its definition.
- An extension method must be defined as `static` in a `static` class and the keyword `this` is used on the first parameter.

```
public static class NumberExtensions
{
    public static bool IsEmpty(this string s)
    {
        return string.IsNullOrEmpty(s);
    }
    public static bool BeforeSunrise(this DateTime dt)
    {
        var sunrise = TimeOnly.MinValue.AddHours(6);
        return TimeOnly.FromDateTime(dt) < sunrise;
    }
}
```

44

Abstract Classes

45

What is abstract class?

- An abstract class provides a partial implementation that other classes can build on.
- When a class is declared as abstract, it can contain incomplete members.
- Incomplete members must be implemented in derived classes.
- Any member that requires a body can be declared abstract

```
abstract class Shape
{
    // Abstract method
    public abstract int GetArea();

    // Abstract property
    public abstract int Color { get; set; }

    // Abstract indexer
    public abstract int this[int index] { get; set; }

    // Abstract event
    public delegate void ShapeEventHandler();
    public abstract event ShapeEventHandler Drawn;
}
```

46



47

Abstract example

```

abstract class Shape
{
    protected int x, y;

    public Point Location
    {
        get { return new Point(x, y); }
    }

    public Shape(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract int GetArea();
}

class Rectangle : Shape
{
    private int w, h;

    public Rectangle(int x, int y, int w, int h)
        : base(x, y)
    {
        this.w = w;
        this.h = h;
    }

    public override int GetArea()
    {
        return w * h;
    }
}

```

47



48

More about abstract classes

- Can we instantiate an object of an abstract class?
- Can an abstract class have constructors?
- Can the deriving class be declared abstract as well?
- Can an abstract class inherit from a non-abstract class?
- Can virtual members be overridden as abstract?
- What are differences between virtual methods and abstract methods?

48

Interfaces

49

What is interface?

- An interface is used to specify members that deriving classes must implement.
- Naming convention: Starts with a capital "I" followed by initially capitalized words.
- The interface code block can only contain:
 - Signatures for methods
 - Properties
 - Indexers
 - Events
 - Default implementations (C# 8).

```
interface IShape
{
    event ShapeEventHandler Drawn;

    Point Location { get; }

    double GetArea();
}

// Built-in interfaces
interface IComparable
{
    int CompareTo(object o);
}

interface ICloneable
{
    object Clone ();
}
```

50



51

Implement interfaces

```
interface IShape
{
    double GetArea();
}
class Circle : IShape, IComparable
{
    private int r;

    public double GetArea() { return r * r * 3.14; }

    public int CompareTo(Object? o)
    {
        if (o != null && o is Circle c)
            return r - c.r;
        else
            return 0;
    }
}
```

51



52

Default implementation (C# 8)

```
interface ILogger
{
    void Log(LogLevel level, string message);

    void Log(Exception ex)
    {
        Log(LogLevel.Error, ex.ToString());
    }
}

class ConsoleLogger : ILogger
{
    public void Log(LogLevel level, string message)
    {
        Console.WriteLine($"{level}: {message}");
    }
}

// Error???
var logger = new ConsoleLogger();
logger.Log(new Exception());

// No error!!!
void LogException(
    ConsoleLogger logger,
    Exception ex)
{
    // Converting to interface
    ILogger ilogger = logger;

    // Calling new Log overload
    ilogger.Log(ex);
}
```

52

Abstract Classes vs. Interfaces

- Similarities:
 - Both can define member signatures that deriving classes must implement
 - Neither one of them can be instantiated
- Differences:
 - Declaration
 - Inheritance
 - Constructor
 - Access level
 - Static members
- **Exercise:** When you should use interface? Abstract class?

53

Constants

54



55

Compile-time constants

- Use `const` modifier
 - Only be used together with the simple types.
 - Must assign a value at the same time as the variable is declared.

```
void Compute()
{
    // Local constant
    const int MAX_POINTS = 10;

    // ...
}

// Use constant field
var edges = 3 * Box.FACES;

class Box
{
    // Compile-time constant field
    public const int FACES = 6;
}
```

- Naming convention: UPPER_CASE
- Constant fields cannot have the static modifier.

55



56

Runtime constants

- Use `readonly` modifier
 - Applied to fields, makes the field unchangeable.
 - Value is assigned at runtime when variable is declared or in the constructor.
 - Can be applied to any data type.

```
class Box
{
    // Runtime constant field
    public readonly int FACES = 6;

    public readonly int[] ORDERS
        = { 1, 2, 3, 4, 5, 6 };
}

class Box
{
    // Runtime constant field
    public readonly string LOGO;

    public Box()
    {
        LOGO = "aWeSoMe";
    }
}
```

56



57

Enum

- An enumeration is a special kind of value type consisting of a list of named constants.

- Definition

```
enum State
{
    Running, // 0
    Waiting, // 1
    Stopped  // 2
};

enum State : byte
{
    Running = 1,
    Waiting = 4,
    Stopped = Waiting * 2
};
```

- Usage:

```
State state = State.Running; state: Running

int i = (int)state; i: 0
string t = state.ToString(); t: Running

switch (state)
{
    case State.Running: /* ... */ break;
    case State.Waiting: /* ... */ break;
    case State.Stopped: /* ... */ break;
}
```

57



58

Enum usage

```
// Get the names of the enum constants
string[] names = Enum.GetNames<State>();

// Get all enum constants
State[] items = Enum.GetValues<State>();
```

```
// Bitwise enum
[Flags]
enum Position
{
    Center = 0,
    Left = 1,
    Right = 2,
    Top = 4,
    Bottom = 8,
    TopLeft = Top | Left,
    TopRight = Top | Right,
    BottomLeft = Bottom | Left,
    BottomRight = Bottom | Right
}
```

- **Exercise:** Learn this library <https://github.com/ardalis/SmartEnum>.

58

Exception Handling

59

Exception Handling

- Exception handling allows programmers to deal with unexpected situations that may occur in programs.

```
static void Main()
{
    // Run-time error
    var sr = new StreamReader("missing.txt");
}
```

- When an exception occurs, the program will terminate with an error message.

60



61

Try-Catch Statement

```

try {
    var sr = new StreamReader("missing.txt");
}
//Same as catch (Exception) {
catch {
    Console.WriteLine("Exception is cached");
}

```

Catch all exceptions

```

try {
    var sr = new StreamReader("missing.txt");
}
catch (FileNotFoundException) {
    Console.WriteLine("File not found");
}
catch (Exception e) {
    Console.WriteLine("Exception is cached."
        + $"Details: {e.Message}");
}

```

Catch specific exceptions

Optional exception object

61



62

Try-Catch-Finally Statement

- Finally block
 - Used to clean up certain resources allocated in the try block.
 - The code in the finally block will always execute even if the try block ends with a jump statement such as return.

```

StreamReader sr = null; sr: null
try {
    sr = new StreamReader("missing.txt");
}
catch (FileNotFoundException) {
    Console.WriteLine("File not found");
}
finally {
    if (sr != null) sr.Close();
}

```

62

Try-Finally Statement

- This statement will not catch any exceptions.
- To ensure the proper disposal of any resources allocated in the try block.

```

Bitmap b = null;
try {
    b = new Bitmap(100, 50);
    Console.WriteLine(b.Width); // "100"
}
finally {
    if (b != null) b.Dispose();
}

```



```

using (Bitmap b = new Bitmap(100, 50)) {
    Console.WriteLine(b.Width); // "100"
} // disposed

```

63

Throwing Exceptions

- To signal the caller that the method has failed.

```

throw new LoginFailedException(
    "Could not login with user: user***");

class LoginFailedException : Exception
{
    public LoginFailedException(string message)
        : base(message)
    {
        LoginFailedException
    }
}

```

```

try {
    ActionMakesError();
}
catch {
    throw; // rethrow error
}

```

64


Throwing Exceptions

- C# 7 allows “throw” to be used as an expression

```
class Student
{
    public string Name { get; set; }

    public Student(string name)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
    }
}
```

nameof expression (C# 6)



65

Struct, Record, Tuple

66



67

Struct

- The **struct** keyword in C# is used to create value types.

Create a struct variable
using default constructor
Fields are set to default value

```
// Point o = default(Point);
Point o = new Point();
int x = o.x; x: 0
int y = o.y; y: 0

struct Point
{
    public int x, y;
}
```

Unlike classes, structs
can also be instantiated
without using the new
operator.

```
Point p;
p.x = 5;
p.y = 15;
```

```
Console.WriteLine($"{p.x}, {p.y}");
```

```
struct Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

When a constructor is defined, all
struct fields must be assigned

67



68

Struct

- Parameterless constructor (C# 10)

```
Box b1 = default(Box); // b1.Size = 0
Box b2 = new Box(); // b2.Size = 1

int b1Size = b1.Size; b1Size: 0
int b2Size = b2.Size; b2Size: 1

struct Box
{
    public int Size;

    public Box()
    {
        this.Size = 1;
    }
}
```

Default keyword can be used to
initialize a struct with all fields
set to their default values.

The parameterless constructor may
be user defined and it can behave
differently from the default
initialization

68



69

Struct

- A struct:
 - Cannot inherit from another struct or class,
 - Cannot be a base class.
- Struct members cannot be declared as protected, private protected or protected internal
- Struct methods cannot be marked as virtual or abstract
- Structs implicitly inherit from System.ValueType.
- A struct can implements interfaces in the same way as classes.

69



70

Struct Guidline

- The struct type is typically used to encapsulate small groups of related variables.
- Two reasons for using struct:
 - To get value type semantics (natural assignment)
 - Increase performance (in terms of memory)
- Assignment and parameter passing by value are typically more expensive with structs than with reference types, because the entire struct needs to be copied for such operations.

70

Struct vs. Class

```
Box b1 = new Box();
Box b2 = b1; b2: Box

b2.Size = 5;

int b1Size = b1.Size; b1Size: 1
int b2Size = b2.Size; b2Size: 5

struct Box
{
    public int Size;

    public Box()
    {
        this.Size = 1;
    }
}
```

```
Box b1 = new Box(); b1: Box
Box b2 = b1; b2: Box

b2.Size = 5;

int b1Size = b1.Size; b1Size: 5
int b2Size = b2.Size; b2Size: 5

class Box
{
    public int Size;

    public Box()
    {
        this.Size = 1;
    }
}
```

71

Struct vs. Class

- Similarities:

- Represents a structure with mainly field and method members.
- Share most of the same syntax as classes
- Can implement interfaces

- Differences:

- A struct is a value type
- A struct variable directly stores the data of the struct (one memory space)
- A struct cannot contain destructors
- A class is a reference type
- A class variable only stores a reference to an object allocated in memory (2 memory spaces: one for the variable and one for the object)

72



73

Record (C# 9)

- The record type defines a reference type with value-based equality behavior.
- Once the object has been created, the data cannot be changed (immutable data).
- A record can contain anything that a class can contain.
- Different ways to create record type:
 - Regular declaration form
 - Positional parameter form
 - Combine both forms

73



74

Record

- Regular declaration form

```
var aboy = new Person
{
    Name = "Nicky",
    Age = 25
};

public record Person
{
    public string Name { get; init; }
    public int Age { get; init; }
}
```

init-only properties maintains the
immutability of the record.

74



75

Record

- Positional parameter form
 - More concise syntax
 - Compiler automatically generates the init-only properties as well as a constructor for those properties

```
var agirl = new Person("Nicole", 20);  
var name = agirl.Name;  
  
public record Person(string Name, int Age);
```

75



76

Record

- Combine two forms

```
var eric = new Person("Eric", 30);  
  
var nicole = new Person("Nicole", 21)  
{  
    Country = "Russia"  
};  
  
public record Person(string Name, int Age)  
{  
    public string? Country { get; init; }  
}
```

76



77

Record Inheritance

- Records support inheritance, allowing a new record type to add properties to an existing record type.
- A record can only inherit from another record or from the `System.Object` class.

```
var daryn = new Student("Daryn", "Math");

public record Person(string name);

public record Student(string name, string subject)
    : Person(name);
```

77



78

Record Behavior

- Although record is a reference type, the compiler automatically implements methods to enforce value-based equality.
- Two record instances are equal if the values of all of their fields and properties are equal.

```
var a = new Person("Nicole", 20);
var b = new Person("Nicole", 20);

bool equal = a.Equals(b);
bool same = a == b;
bool diff = a != b;

var info = a.ToString();

public record Person(string Name, int Age);
```

The compiler-generated `ToString()` method returns the names and values of all public fields and properties.

78

Record Behavior

- Properties of an immutable record instance cannot be modified
- However, they can be copied over to a new record by using a **with** expression.

```
var s = new Student("Jay", "Bio"); s: Student {...}

// copy record
var a = s with {}; a: Student {...}

// copy and alter record
var b = s with { Name = "Sara" }; b: Student {...}

var aName = a.Name; aName: Jay
var bName = b.Name; bName: Sara

var equal = s == a; equal: True
```

79

Tuples (C# 7)

- Tuples are an efficient way to combine two or more (loosely related) values into a single unit.

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple: {t1.Item1}, {t1.Item2}.");
// Output: Tuple: 4.5, 3.

(double Sum, int Count) t2 = (4.5, 3); t2: (4.5, 3)
Console.WriteLine($"Count: {t2.Count}, Sum: {t2.Sum}.");
// Output: Count: 3, Sum: 4.5.

var sum = 4.5; sum: 4.5
var count = 3; count: 3
var t = (sum, count);
Console.WriteLine($"Count: {t.count}, Sum: {t.sum}.");
// Output: Count: 3, Sum: 4.5.
```

80

Tuple Usage

- One of the most common use cases of tuples is as a method return type.

```
(int Quotient, int Remainder) Divide(int dividen, int divisor)
{
    int q = dividen / divisor;
    int r = dividen % divisor;

    return (q, r);
}

var result = Divide(10, 3); result: (3, 1)
var (Quotient, Remainder) = Divide(15, 4); Quotient: 3, Remainder: 3
var (q, r) = Divide(15, 4); q: 3, r: 3
```

- Use tuple types instead of anonymous types in LINQ queries.

81

Delegates and Events

82



83

Delegates

- A delegate is a type used to reference a method.
- It allows methods to be assigned to variables and passed as arguments.
- The delegate's declaration specifies the method signature to which objects of the delegate type can refer.

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

delegate void PrintDelegate(string input);

PrintDelegate d1 = new PrintDelegate(Print);
PrintDelegate d2 = Print; d2: PrintDele...
d1("Hello");
d2("World!");
```

83



84

Anonymous Methods (C# 2)

- An anonymous (unnamed) method is specified by using the delegate keyword followed by a method parameter list and body.
- Anonymous methods can be assigned to delegate objects.

```
PrintDelegate f = delegate(string s) f: PrintDele...
{
    Console.WriteLine(s);
};

f("I have no name");
```

84



85

Lambda Expressions (C# 3)

- Same goal as anonymous methods but with a more concise syntax.
- A lambda expression is written as a parameter list followed by the lambda operator (`=>`) and an expression.

```
// Create delegate
delegate int NumberDelegate(int i);

// Use delegate
a(5); // 25
b(5); // 25

// Anonymous method
NumberDelegate a = delegate(int x)
{
    return x * x;
};

// Lambda expression
NumberDelegate b = (int x) => x * x;
```

85



86

Lambda Expressions (C# 3)

- The lambda must match the signature of the delegate.
- Typically, the compiler can determine the data type of the parameters from the context, so they do not need to be specified.
- The parentheses may also be left out if the lambda has only one input parameter.

```
// Lambda expression
NumberDelegate b = x => x * x;
```

- If no input parameters are needed, an empty set of parentheses must be specified.

```
EmptyDelegate d = () => Console.WriteLine("Hello");

// Delegate without input
delegate void EmptyDelegate();
```

86

Lambda Expressions (C# 3)

- Expression lambda:

```
// Executes a single statement
NumberDelegate b = x => x * x;
```


- Statement lambda

```
// Contains multiple statements
NumberDelegate sum = n => { sum: NumberDel...
    var s = 0;
    for (var i=1; i<=n; i++)
        s += i;
    return s;
};
```

87

Lambda Expressions

- Expression body definition (C# 6)
 - Applied to methods and get properties that consists of only a single expression

<pre>class Person { public string Name { get; } = "John"; public void PrintName() { Console.WriteLine(Name); } }</pre>		<pre>class Person { public string Name => "John"; public void PrintName() => Console.WriteLine(Name); }</pre>
---	---	--

- C# 7: Extended list of allowed members to constructors, destructors, set properties and indexers.

88

Lambda Expressions

- Type inference (C# 10):
 - The compiler automatically infer a delegate type for a lambda.
 - Allow storing a reference to a lambda without declaring a delegate type or explicitly using built-in delegate types

```
var pow = (int x) => x * x;
```

The parameter type has to be specified since the compiler don't know the type of input.

```
var select = object (bool b) => b ? 0 : "one";
```

If the return type is ambiguous, it has to be specified explicitly

89

Multicast Delegates

- Multicast delegate: A delegate object refers to more than one method.
- The methods it refers to are contained in a so-called invocation list.

```
void Hi(string name) { Console.WriteLine($"Hi {name}, "); }
void Hello(string name) { Console.WriteLine($"Hello {name}, "); }
void Bye(string name) { Console.WriteLine($"Bye {name}, "); }
```

```
delegate void SayDelegate(string name);
```

```
SayDelegate say = Hello;
```

```
// add a method from the invocation list
say = say + Hi;    // addition operator
say += Bye;       // addition assignment op
```

```
say("John");      // -> Hello John, Hi John, Bye John
```

90



91

Delegates as Parameters

- Delegates can be passed as method parameters.

```

delegate void ProcessPerson(string name);

class PersonDB
{
    string[] list = { "John", "Sam", "Dave" };

    public void Process(ProcessPerson action)
    {
        foreach(string s in list)
            action(s);
    }
}

class Client
{
    public void Run()
    {
        PersonDB p = new PersonDB();
        p.Process(PrintName);
    }

    public void PrintName(string name)
    {
        Console.WriteLine(name);
    }
}

```

91



92

Events

- Events enable an object to notify other objects when something of interest occurs.
- Publisher: The object that raises the event
- Subscribers: The objects that handle the event.
- A delegate is needed to hold the subscribers.

```

// Standard pattern for event delegate
public delegate void EventHandlerDelegate(
    object sender, EventArgs e);

```

The first parameter specifies the source object of the event

The second parameter is a type that either is or inherits from the EventArgs class. It contains the details of the event.

Return type should be void

92



93

Events

- Example

```
// Represents event handler
delegate void NewspaperHandler(
    object sender, NewspaperEventArgs e);

// Represents data/details of the event
class Newspaper
{
    public string Title { get; init; }
}

// Represents event argument
class NewspaperEventArgs : EventArgs
{
    public Newspaper Newspaper { get; init; }
}
```

93



94

Events

- Example

```
class Publisher
{
    // Create event
    public event NewspaperHandler NewspaperPublished;

    // Event caller: Invokes the event
    protected virtual void OnNewspaperPublished(
        NewspaperEventArgs eventArgs)
    {
        if (NewspaperPublished != null)
            NewspaperPublished(this, eventArgs);
    }

    public void Publish(Newspaper newspaper)
    {
        // Do something ...

        // Raise event or dispatch event
        OnNewspaperPublished(new NewspaperEventArgs
        {
            Newspaper = newspaper
        });

        // Do other things ...
    }
}
```

94



95

Events

- Example

```
// Represents the subscriber
abstract class Customer
{
    public string Name { get; }

    public Customer(string name)
    {
        Name = name;
    }

    public virtual void Register(Publisher publisher)
    {
        // Do something ...

        // Subscribes to the event
        publisher.NewspaperPublished += ReceiveNewspaper;
    }

    // Event handler method
    public abstract void ReceiveNewspaper(
        object sender, NewspaperEventArgs ne);
}
```

95



96

Events

- Example

```
class Reader : Customer
{
    public Reader(string name) : base(name) {}

    public override void ReceiveNewspaper(
        object sender, NewspaperEventArgs ne)
    {
        Console.WriteLine(
            $"Reader {Name} is reading {ne.Newspaper.Title}");
    }
}

class Agency : Customer
{
    public Agency(string name) : base(name) {}

    public override void ReceiveNewspaper(
        object sender, NewspaperEventArgs ne)
    {
        Console.WriteLine(
            $"Agent {Name} sells {ne.Newspaper.Title}");
    }
}
```

96



97

Events

- Example

```
,var laodong = new Publisher(); laodong: Publisher
,var readerA = new Reader("Reader A"); readerA: Reader
,var agencyB = new Agency("Agency B"); agencyB: Agency

,readerA.Register(laodong);
,agencyB.Register(laodong);

,laodong.Publish(new Newspaper
{
    Title = "This is a hot news"
});
```

- Output

```
Reader Reader A is reading This is a hot news
Agent Agency B sells This is a hot news
```

97



98

Built-in Generic Delegates (C# 3.5)

- `public delegate void EventHandler(object sender, EventArgs e);`
- `public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);`
- `public delegate void Action<in T>(T obj);`
- `public delegate bool Predicate<in T>(T obj);`
- `public delegate TResult Func<in T, out TResult>(T arg);`

`TResult` : Type of the return value

```
public delegate TResult Func<in T, out TResult>(T arg);
```

`T` : Type of the input parameter

98

Generics

99

Generics

- Generics refer to the use of type parameters.
- Generics provide a way to design code templates that can operate with different data types.
- We can create generic methods, classes, interfaces, delegates, and events.

100

Generic Methods

```
void Swap(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
// Call generic method
int a = 0, b = 1; a: 0, b: 1
char x = 'm', y = 'n'; x: m, y: n
Swap<int>(ref a, ref b);
Swap<char>(ref x, ref y);
```

Type parameter: should start with a capital T followed by initially capitalized words.

```
void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Replace the original data type with the type parameter

101

Generic Type Parameters

- A generic can be defined to accept more than one type parameter just by adding more of them between the angle brackets.

```
Add<int, Reader>(1, new Reader());
Add<Guid, Agency>(Guid.NewGuid(), new Agency());

void Add<TKey, TValue>(TKey key, TValue value)
{
    // Do something with key and value
}
```

102



103

Default Value

```

void Reset<T>(ref T a)
{
    a = default(T);
}

void Reset<T>(ref T a)
{
    a = default;    // C# 7.1: Same as default(T)
}

int n = 10;
Reset<int>(ref n);    // n = 0

bool b = true;
Reset<bool>(ref b);    // b = False

```

103



104

Generic Classes

```

Point<int> pi = new Point<int>
{
    X = 10,
    Y = 15
};

Point<float> pf = new Point<float>
{
    X = 10.25f,
    Y = 11.31f
};

class Point<T>
{
    public T X {get; set;}
    public T Y {get; set;}
}

Customer<int> ci = new Customer<int>
{
    Id = 1,
    Name = "Tony"
};

Customer<Guid> cg = new Customer<Guid>
{
    Id = Guid.NewGuid(),
    Name = "Tony"
};

class Customer<TId>
{
    public TId Id { get; set;}
    public string Name { get; set; }
}

```

104

Generic Interfaces

```

interface IGenericCollection<T>
{
    void Store(T item);
}

class LegoBox : IGenericCollection<Lego>
{
    public void Store(Lego item)
    {
        // ...
    }
}

class Lego
{
    //...
}

class GenericBox<T> : IGenericCollection<T>
{
    public void Store(T item)
    {
        // ...
    }
}

```

105

Generic Delegates

```

PrintDelegate<string> d = Print; d: PrintDele...
d("Hello");
void Print(string s) s: Hello
{
    Console.Write(s);
}

public delegate void PrintDelegate<T>(T arg);

```

Type parameter matches method's parameter type.

Type parameter specifies the referable method's parameter.

106

Generic Events

```

delegate void EventDelegate<T, U>(T sender, U eventArgs);

class Publisher
{
    event EventDelegate<Publisher, BookEventArgs> BookPublished;
}

class BookEventArgs : EventArgs
{
}

```

107

Constraints

- Compile-time-enforced restrictions can be applied on the kinds of type arguments that may be used when the class or method is instantiated.
- Constraints are specified using the **where** keyword.
- 6 kinds of constraints:
 - `class C<T> where T : struct {}` // value type
 - `class D<T> where T : class {}` // reference type
 - `class E<T> where T : B {}` // be/derive from base class B
 - `class F<T, U> where T : U {}` // be/derive from U
 - `class G<T> where T : I {}` // be/implement interface I
 - `class H<T> where T : new() {}` // parameterless constructor

108



109

Multiple Constraints

- Type parameters are specified in a comma-separated list.
- To constrain more than one type parameter, additional where clauses can be added.
- The class or the struct constraint must appear first in the list.
- The parameterless constructor constraint must be the last one in the list.

```
var jo = new J<X, Y>(); jo: J`2[X,Y]

class J<T, U>
    where T : class
    where U : Y, new()
{
    // ...
}

class Y
{
    public Y()
    {
    }
}

class X
{
}
```

109



110

Example

```
interface ICustomer<TKey>
{
    public TKey Id { get; set; }
    public string Name { get; set; }
}

class CustomerMapper<TKey, TCustomer>
    where TCustomer : ICustomer<TKey>
{
    public void Add(TKey key, TCustomer customer)
    {
        // ...
    }

    public TCustomer GetOrDefault(TKey key)
    {
        // ...
        return default;
    }
}
```

110

Asynchronous Methods

111

What is Asynchronous Method?

- An asynchronous method is a method that can return before it has finished executing.
- Asynchronous methods improve the responsiveness of the program.
- `async` & `await` (C# 5):
 - The `async` modifier specifies that the method is asynchronous and that it can therefore contain one or more `await` expressions.
 - An `await` expression consists of the `await` keyword followed by an awaitable method call.
- Naming convention: `DoSomethingAsync()`

112



113

Example

```
WriteTextAsync();
Console.Write("B");

// prevent the program from exiting
// before the async method has finished.
Console.ReadKey();

// Output: ABC

async void WriteTextAsync()
{
    Console.Write("A");
    await Task.Delay(5000);
    Console.Write("C");
}
```

- Task.Delay(5000): A timed delay that will complete after 5 seconds

113



114

Async Return Types (C# 5)

- Three built-in return types: Task<T>, Task, and void.
 - Task or void denotes that the method does not return a value.
 - Task<T> means it will return a value of type T.
 - The Task and Task<T> types are awaitable, a caller can use the await keyword to suspend itself until after the task has finished.
 - The void type is mainly used to define async event handlers, as event handlers require a void return type.

```
async Task WriteTextAsync()
{
    Console.Write("A");
    await Task.Delay(5000);
    Console.Write("C");
}
```



```
await WriteTextAsync();
Console.Write("B");

// Output: ACB
```

114

Custom Asynchronous Methods

```

string LongRunningAction()
{
    // Do something that takes time
    Thread.Sleep(5000);
    return "Y";
}

Task<string> LongRunningTask()
{
    return Task.Run<string>(() => {
        Thread.Sleep(5000);
        return "Y";
    });
}

RunTaskAsync();
Console.Write("X");
Console.ReadKey();

// Output: XY

```

The diagram illustrates the process of creating custom asynchronous methods. It shows a synchronous method `LongRunningAction()` being converted into a `Task<string>` `LongRunningTask()` using `Task.Run`. This task is then used within an `async void RunTaskAsync()` method, which `await`s the task and writes the result. A final call to `RunTaskAsync()` in the main thread demonstrates how the asynchronous operation can be initiated without blocking the main thread, as evidenced by the output "XY" where 'X' is printed before 'Y'.

115

Top-Level Statements

116



117

Top-Level Statements (C# 9)

- Allows the Main method and its surrounding class to be omitted.
- A compiler feature that automatically generates the omitted class and Main method.
- Any statements typed at the top level will be moved into the Main method.
- Usage directives can appear before the top-level statements.
- Type definitions or namespaces have to be placed below the top-level statements.
- A project may only have one file with top-level statements.

117



118

Top-Level Statements (C# 9)

```
using System; // Not needed in .NET 6 or later.

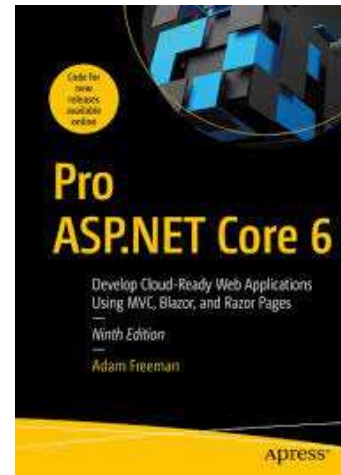
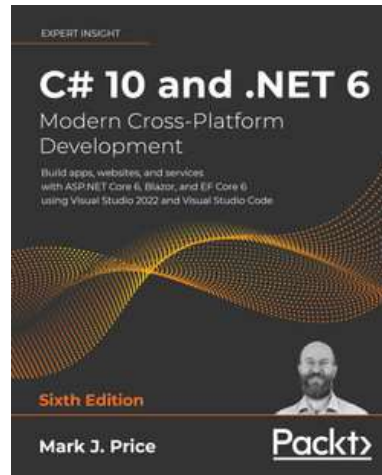
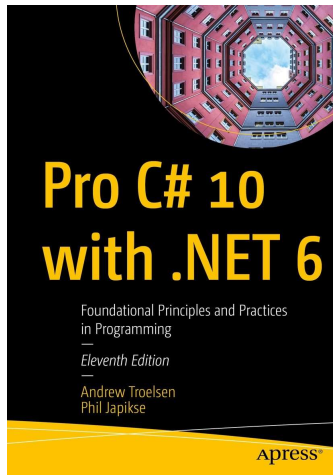
namespace HelloCSharp
{
    class Program
    {
        static void Main(string[] args) args: { }
        {
            Console.WriteLine("Hello, World!");
        }
    }
}

↓

Console.WriteLine("Hello, World!");
```

118

References



119

END

120