1. **Instance Methods**:

- Belong to an instance of a class (an object).
- Called using an object of the class.

```
class Dog {
    public void Bark() {
        Console.WriteLine("Woof!");
    }
}

Dog myDog = new Dog();
myDog.Bark(); // Calls the instance method
```

2. **Static Methods**:

- Belong to the class itself, not an instance.
- Called using the class name.

```
class MathUtils {
    public static int Square(int x) {
        return x * x;
    }
}

int result = MathUtils.Square(5); // result = 25
```

3. **Local Methods**

These are methods declared **inside** another method. They help organize helper logic that's only relevant to the outer method. , Encapsulate helper logic and improve readability without polluting the class's outer scope.

```
public void ProcessData() {
    Console.WriteLine("Processing started...");

    int Multiply(int a, int b) {
        return a * b;
```

```
    }

    int result = Multiply(4, 5);
    Console.WriteLine($"Result: {result}");
}
```

4. **Access Modifiers**:

- Control the visibility of methods (e.g., public, private, protected, internal).

```
public void Show() { }      // Accessible everywhere

private void Hide() { }     // Accessible only inside the class
```

5. **return Types**:

- A method can return a value or be void (return nothing).

```
public int Add(int a, int b) {

    return a + b;

}


public void Greet() {

    Console.WriteLine("Hello!");

}
```

6. **Overloading**:

- You can define multiple methods with the same name but different parameters.

```
public void Print(string text) { }
public void Print(int number) { }
```

7. **Virtual and Override**:

- Used in inheritance to allow method customization.

```
class Animal {
    public virtual void Speak() {
        Console.WriteLine("Animal sound");
    }
}

class Cat : Animal {
    public override void Speak() {
        Console.WriteLine("Meow");
    }
}
```

8. Expression-Bodied Methods

These are concise method definitions using the => syntax, ideal for simple methods that return a single expression. , When a method only contains one line of logic — keeps code clean and readable.

```
public class Calculator {
    public int Square(int x) => x * x;
}
```

9. Pass by Value (Default Behavior)

When you pass a variable to a method **by value**, a **copy** of the variable is sent. Changes inside the method **don't affect** the original variable.

```
void Increase(int number) {
    number += 10;
}

int x = 5;
Increase(x);
Console.WriteLine(x);  // Output: 5 (unchanged)
```

10. Pass by Reference (ref)

Using ref, you pass the **actual variable**, not just a copy. So changes inside the method **do affect** the original variable. , You **must initialize** the variable before passing it with ref

```
void Increase(ref int number) {
    number += 10;
}

int x = 5;
Increase(ref x);
Console.WriteLine(x);  // Output: 15 (changed)
```

11. Output Parameter (out)

Like ref, but designed for methods that **return multiple values**. With out, the variable **doesn't need to be initialized** before the method call, but **must be assigned** inside the method.

```
void GetStats(out int sum, out int product) {
    sum = 5 + 10;
    product = 5 * 10;
}

int a, b;
GetStats(out a, out b);
Console.WriteLine($"Sum: {a}, Product: {b}");  // Output: Sum: 15, Product: 50
```

---

1. Constructor (Basic)

A **constructor** is a special method that gets called **automatically** when you create an object. It's usually used to initialize values.

```
class Person {
    public string Name;

    // Constructor
    public Person(string name) {
        Name = name;
    }
}
```

2. Implicit Constructor

If you don't define any constructor, C# provides a **default (parameterless)** constructor implicitly.

```
class Animal {
    public string Type;
}

// Implicit parameterless constructor exists
Animal a = new Animal();
```

But if **you define any constructor**, the implicit one **won't be provided automatically** unless you explicitly add it.

3. Overloaded Constructor

You can define **multiple constructors** with different parameter lists — this is called **constructor overloading**.

```
class Car {
    public string Model;
    public int Year;

    public Car(string model) {
        Model = model;
        Year = 2020;
    }

    public Car(string model, int year) {
        Model = model;
        Year = year;
    }
}
```

4. This Keyword

Used to refer to the **current object's instance** — often to avoid name conflicts or to call **another constructor**.

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

    public Book(string title) : this(title, "Unknown") { }

    public Book(string title, string author) {
        this.Title = title;
        this.Author = author;
    }
}
```

5. Non-Public Constructor

You can define constructors with any **access modifier**. A private or protected constructor is often used when You want only one instance, validate or control object creation.

```
class Secret {
    private Secret() { }

    public static Secret Create() {
        return new Secret();  // Allowed inside the class
    }
}
```

6. Object Initializer

Lets you set property values **at the time of object creation**, even when using a parameterless constructor.

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

Student s = new Student {
    Name = "John",
    Age = 21
};
```

7. Readonly Field

A readonly field can only be assigned **once**, either at declaration or in the constructor. Unlike const, it's set at **runtime**, not compile-time. , Once set in constructor, readonly fields can’t be changed.

```
class Config {
    public readonly string AppName;

    public Config(string appName) {
        AppName = appName;
    }
}
```

---

1. What is a Property in C#?

A **property** in C# is a special kind of class member that provides a **controlled way to read, write, or compute** the value of a **private field**.

It acts like a combination of:

- A **field** (for storing data)
- And a **method** (for getting/setting the data safely)

2. Property & Encapsulation

Encapsulation is an OOP principle where object data is hidden and accessed only through **public members** (like properties). Properties allow **controlled access** to private fields.

```
class Person {
    private int age; // Encapsulated

    public int Age {
        get { return age; }
        set {
            if (value >= 0) age = value;
        }
    }
}
```

3. Why Property?

Instead of exposing fields directly (which is risky), properties give you:

- Validation (e.g., no negative age)
- Read-only/write-only access
- Flexibility (you can change logic later without changing the class interface)

```
// Bad practice
public int age; // anyone can directly set it to -100

// Better
public int Age {
    get { return age; }
    set {
        if (value >= 0) age = value;
    }
}
```

4. Get and Set Accessors

- get retrieves the value.
- set assigns the value.
- value is a **keyword** that represents what's being assigned.

```
private string name;

public string Name {
    get { return name; }
    set { name = value.Trim(); } // trimming whitespace
}
```

5. Property & Backing Field

A **backing field** is a private variable that holds the actual data behind a property.

```
private double _salary; // backing field

public double Salary {
    get { return _salary; }
    set {
        if (value >= 0) _salary = value;
    }
}
```

6. Property and Accessibility

You can control access to get and set individually:

```
public string Email { get; private set; } // Only readable outside

public int Age { private get; set; }      // Only writable outside
```

7. Automatic Property

No need to write a backing field manually — C# handles it internally:

```
public string FirstName { get; set; } // automatic property
```

8. Property Internally

Sometimes, you want properties to be accessible **only inside the class or assembly**:

```
public string Status { get; internal set; } // settable only inside the same assembly

public string Role { get; private set; }    // settable only inside the class
```

9. Read-Only Properties

You can make properties read-only by defining only get:

```
public string ID { get; } = Guid.NewGuid().ToString(); // assigned once
```

---

1. What is an Indexer?

An **indexer** allows an object to be **indexed like an array**. It looks like a property, but it takes parameters (usually the index) and is used with []. , They are useful when your class holds a collection or behaves like one.

```
object[index] = value;
value = object[index];
```

2. Scenarios When to Use

- Custom collection classes
- Wrapping a list, array, or dictionary
- Creating grid/matrix-like access
- Data models like **Sudoku**, **Chess boards**, **Tables**, etc.

3. Single-Dimensional Map Example

```
class NameMap {
    private string[] names = new string[5];

    public string this[int index] {
        get => names[index];
        set => names[index] = value;
    }
}
```

4. Multi-Dimensional Maps

You can have indexers that take multiple parameters (like a 2D array):

```
class Matrix {
    private int[,] grid = new int[3, 3];

    public int this[int row, int col] {
        get => grid[row, col];
        set => grid[row, col] = value;
    }
}
```

5. Sudoku Example Using Indexers

```
class Sudoku {
    private int[,] board = new int[9, 9];

    public int this[int row, int col] {
        get => board[row, col];
        set {
            if (value >= 0 && value <= 9)
                board[row, col] = value;
        }
    }

    public void PrintBoard() {
        for (int r = 0; r < 9; r++) {
            for (int c = 0; c < 9; c++) {
                Console.Write(board[r, c] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

---

1. What is a Delegate?

A **delegate** is a **type-safe object** that holds a reference to a **method** with a specific signature.

like a **function pointer**

```
public delegate void GreetDelegate(string name);
```

2. Scenarios When to Use Delegates

- Callbacks: pass logic as parameters
- Events: button clicks, timers, etc.
- Plugin systems or extensibility
- Replacing hard-coded logic with flexible behavior
- Filtering/sorting (e.g., LINQ, predicates)

3. When to Apply Delegates

Use delegates when:

- You want to pass a **method as data**
- You need to **trigger different logic dynamically**
- You want **loosely-coupled** design

✅ Delegates enable **decoupling** – one part defines “what to do,” another defines “when to do it.”

4. Anonymous Delegate

An **anonymous delegate** is an inline method with no name. , Useful for short, one-off actions without creating separate methods

```
GreetDelegate greet = delegate(string name) {
    Console.WriteLine("Hello, " + name);
};

greet("Alice");
```

5. Lambda Expression

A **lambda** is a modern, short syntax for anonymous methods.

, Lambdas are used heavily in LINQ and functional programming in C#.

```
GreetDelegate greet = (name) => Console.WriteLine("Hi " + name);
greet("Bob");
```

6. Multicast Delegate

A **multicast delegate** can point to **multiple methods**. All will be called in order.

```
public delegate void Notify();

void SendEmail() => Console.WriteLine("Email sent");
void SendSMS() => Console.WriteLine("SMS sent");

Notify notify = SendEmail;
notify += SendSMS;

notify(); // Calls both methods
// Return values are ignored except for the last method.
```

---

1. What is an Event?

An **event** is a way for a class (publisher) to notify other classes (subscribers) when **something happens**.

It acts as a **wrapper around a delegate**, allowing **only subscription/unsubscription**, not direct invocation from outside.

2. Event and Delegate

- Events are **based on delegates**.
- You define a delegate type → then define an event using that delegate.
- Only the **publisher** class can raise (invoke) the event.

```
public delegate void AlarmHandler();
public event AlarmHandler OnAlarm;
```

3. Event Publisher

The **publisher** is the class that declares and raises the event.

```
class Alarm {
    public event AlarmHandler OnRing;

    public void Trigger() {
        Console.WriteLine("Alarm Triggered!");
        OnRing?.Invoke();  // Raise the event
    }
}
```

4. Subscribe vs Unsubscribe

- **Subscribe (+=)** connects a method to the event.
- **Unsubscribe (-=)** removes the connection.
- You can subscribe multiple methods to one event.

```
void Notify() => Console.WriteLine("Notified!");

alarm.OnRing += Notify;  // Subscribe
alarm.OnRing -= Notify;  // Unsubscribe
```

5. Lambda Expression Handler

You can use a lambda instead of a named method for short, inline responses:

```
alarm.OnRing += () => Console.WriteLine("Lambda: Alarm is ringing!");
```

Or with the standard pattern:

```
alarm.AlarmRang += (sender, e) => Console.WriteLine("Handled with lambda");
```

6. What is an Event Handler?

An **Event Handler** is just a **method** that responds to an **event**.

It gets called **automatically** when the event is raised (invoked).

standard delegate for most events:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Using EventHandler

```
public class Alarm {
    public event EventHandler AlarmRang;

    public void Trigger() {
        Console.WriteLine("Alarm triggered!");
        AlarmRang?.Invoke(this, EventArgs.Empty);
    }
}
```

OPERATOR OVERLOADING

Operator overloading lets you redefine the behavior of built-in operators like (+, -) for user-defined types **like classes**

Unary vs. Binary Operators

- **Unary Operators**: Operate on **one operand**
  Examples: +, -, !, ++, --
- **Binary Operators**: Operate on **two operands**
  Examples: +, -, *, /, ==, !=

Supported Operators

- Arithmetic: +, -, *, /, %
- Comparison: ==, !=, <, >, <=, >=
- Logical: &, |, ^, !
- Increment/Decrement: ++, --

Must Be Overloaded in Pairs

comparison operators must be overloaded in pairs:

- == and !=
- < and >
- <= and >=