C# Programming – Day 3

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects**, which represent **real-world entities**.
Instead of focusing only on functions or logic, OOP focuses on **data and the operations performed on that data together**.

In OOP, a program is built using **objects that interact with each other**, just like objects in the real world interact.

Each object contains:

- **Data** (fields)
- **Behavior** (methods)

OOP allows developers to design software in a **modular and structured way**, where each object is responsible for its own data and behavior. This separation of responsibilities makes programs easier to understand, test, debug, and extend. When changes are required, developers can modify or enhance specific objects without affecting the entire system. Because of this, Object-Oriented Programming is widely used in large-scale applications such as banking systems, e-commerce platforms, enterprise software, and game development, where managing complexity is crucial.

Core Idea of OOP

OOP models software the same way we understand the real world:

- A **Car** has properties like color and speed
- A **Car** can perform actions like start, stop, and accelerate

In programming:

- Properties → **Data (Fields)**
- Actions → **Behavior (Methods)**

Components of an Object

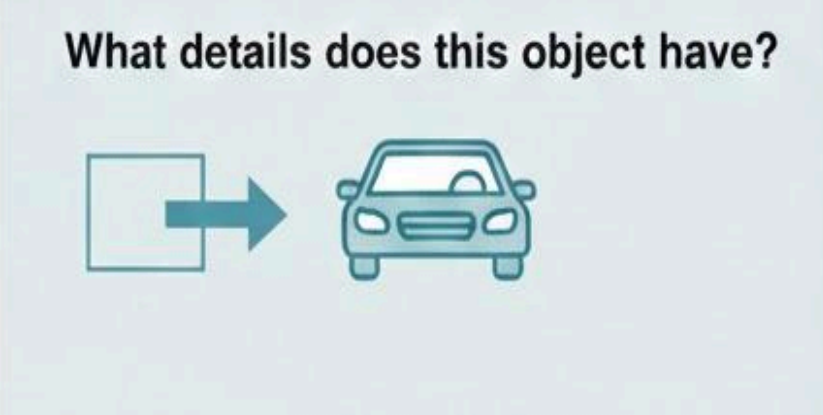
1. Data (Fields)

Fields represent the data or state of an object in Object-Oriented Programming.
They store information that describes what an object is at any given moment.

In simple words:

Fields answer the question: “What details does this object have?”

Each object of a class can have different values for the same fields.



Key Points about Fields

- Fields store **data**
- They define the **current state** of an object
- Also called **data members** or **attributes**
- Declared inside a **class**
- Can have access modifiers like `private`, `public`, etc.

Real-World Example : Bank Account

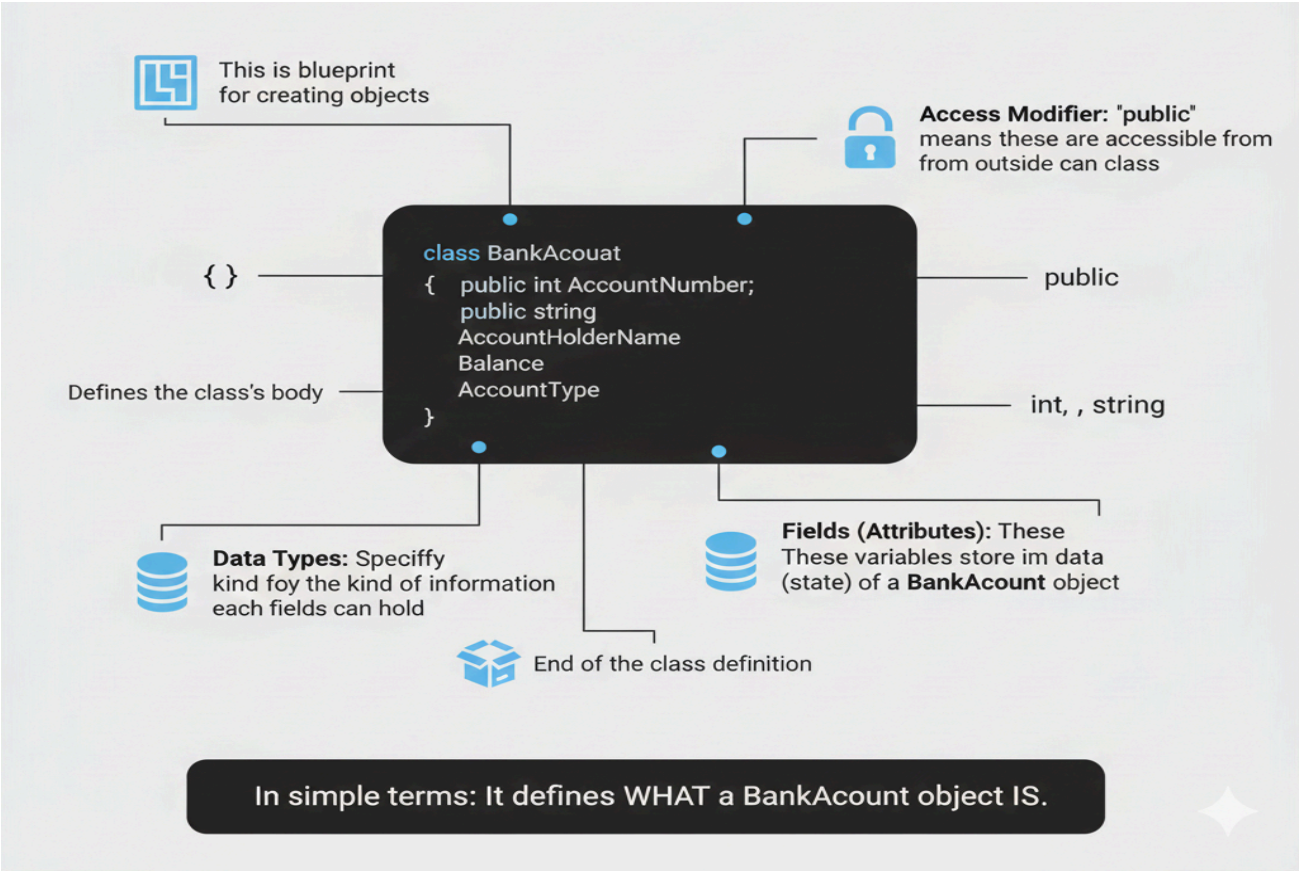
A Bank Account object contains the following fields:

- Account Number
- Account Holder Name
- Account Balance
- Account Type (Savings / Current)
- IFSC Code
- Branch Name
- Account Status (Active / Blocked)
- Opening Date

These fields describe the **state of a bank account** at any moment.

C# Example: Fields in a Class

```
class BankAccount
{
    public int AccountNumber;
    public string AccountHolderName;
    public double Balance;
    public string AccountType;
}
```



2. Behavior (Methods)

Methods represent the **behavior or actions** of an object.
They define **what an object can do** and **how it operates on its data (fields)**.

In Object-Oriented Programming, methods are used to:

- Manipulate object data
- Perform operations related to the object
- Control access to the object's internal state

In simple words:
Fields store information, methods use that information to perform actions.

Real-World Example : Bank Account

A Bank Account is not just data; it also performs actions.

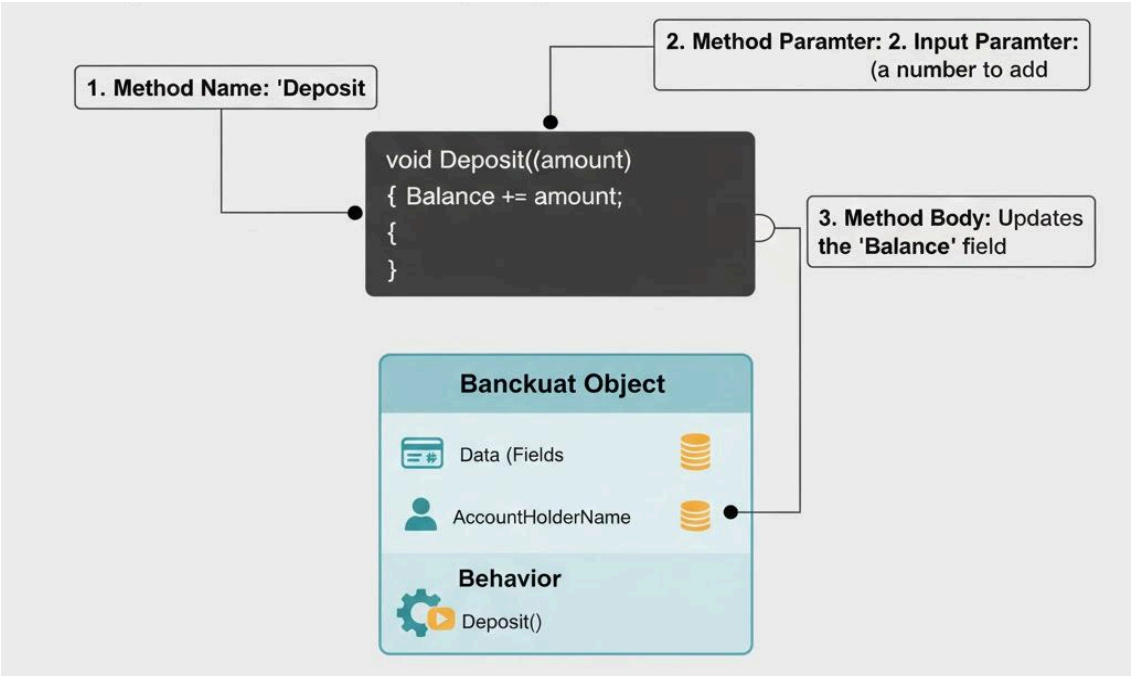
Behaviors of a Bank Account:

- Deposit money
- Withdraw money
- Check balance
- Transfer funds
- Generate statement

These actions change or use the account's data such as balance and account number.

Programming Example

```
void Deposit(double amount)
{
    Balance += amount;
}
```



Explanation:

- `Deposit` is a method
- `amount` is the input parameter
- The method updates the `Balance` field
- Logic and data work **together inside the object**

This ensures that balance changes only through valid operations.

Why Methods Are Important

- They protect data by controlling how it is modified
- They make code logical and readable
- They represent real-world actions
- They help enforce business rules (Example: preventing withdrawal if balance is insufficient)

Real-Life Example of OOP

Real World Entity	Programming Concept
Car	Class
BMW, Audi, Tesla	Objects
Speed, Color, Fuel	Fields
Drive(), Brake(), Accelerate()	Methods
Bank	Class
Customer Accounts	Objects
Balance, Account No	Fields
Deposit(), Withdraw()	Methods
Hospital	Class
Patients	Objects

1. Class and Object

Class

A class is a blueprint, template, or design used to create objects.It defines what data an object will have and what actions it can perform, but it does not hold actual values by itself.

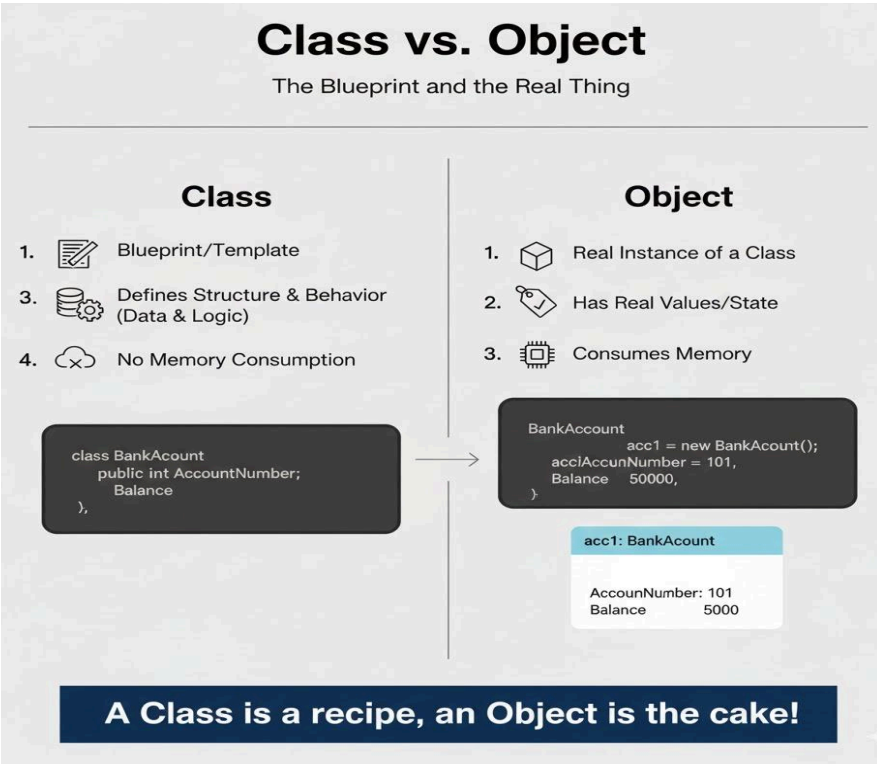
A class represents a general concept, not a real thing.

- Defines structure
- Defines behavior
- Does not consume memory until an object is created

Object

An object is a real instance of a class. It represents a real-world entity created using a class and holds actual data.

- Consumes memory
- Has real values
- Can call methods of the class



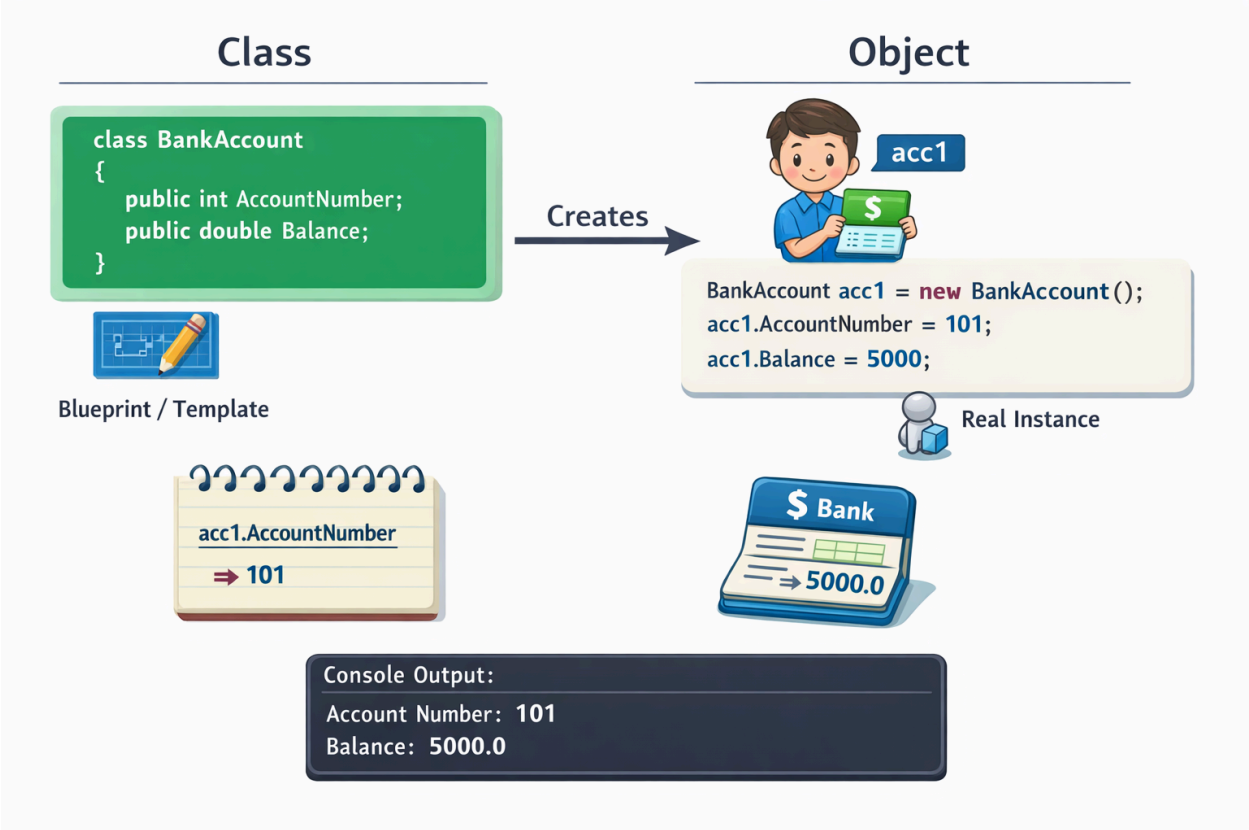
Code Example (C#)

```
class BankAccount
{
    public int AccountNumber;
    public double Balance;
}

BankAccount acc1 = new BankAccount();
acc1.AccountNumber = 101;
acc1.Balance = 5000;
```

Explanation:

- `BankAccount` → Class (blueprint)
- `acc1` → Object (real instance)
- `AccountNumber` and `Balance` → Data stored in the object

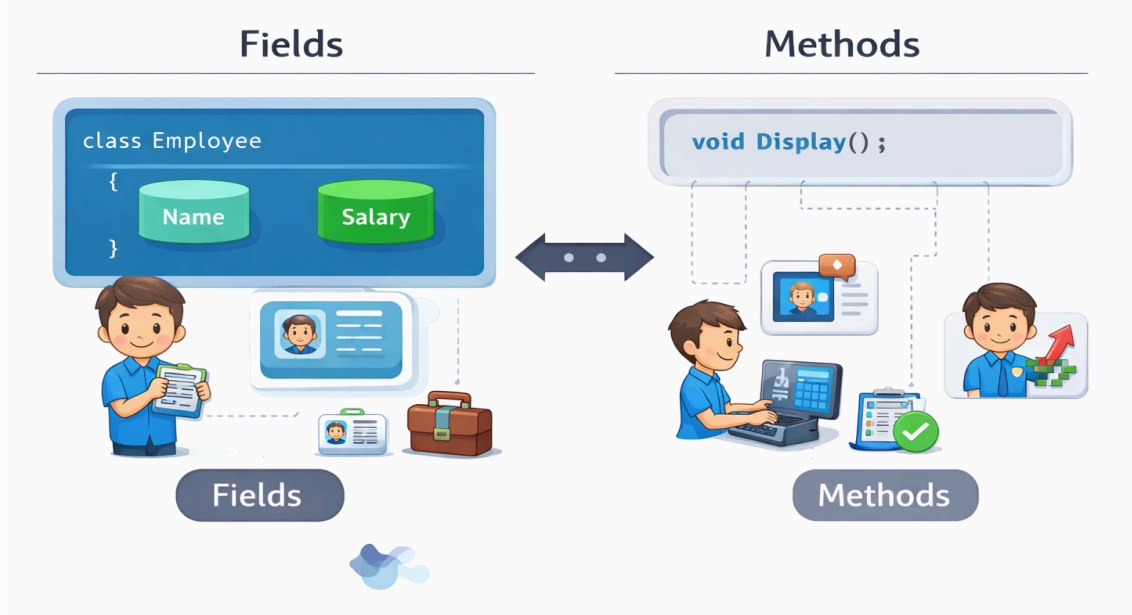


Real-World Explanation

Think of a class as a design and an object as the actual item created from that design.

2. Fields and Methods

In **Object-Oriented Programming (OOP)**, an object is defined by **what it has** and **what it can do**.
These two aspects are represented using **Fields** and **Methods**.



Fields (Data Members)

Fields are variables declared inside a class that **store the state or data of an object**.They represent the **properties or characteristics** of an object.

Each object created from a class has **its own copy of fields**, meaning field values can differ from one object to another.

Characteristics of Fields

- Store object-related information
- Represent the **current state** of an object
- Can be of any data type (int, string, double, etc.)
- Defined inside a class but outside methods
- Accessed using object reference

Real-Life Example (Employee)

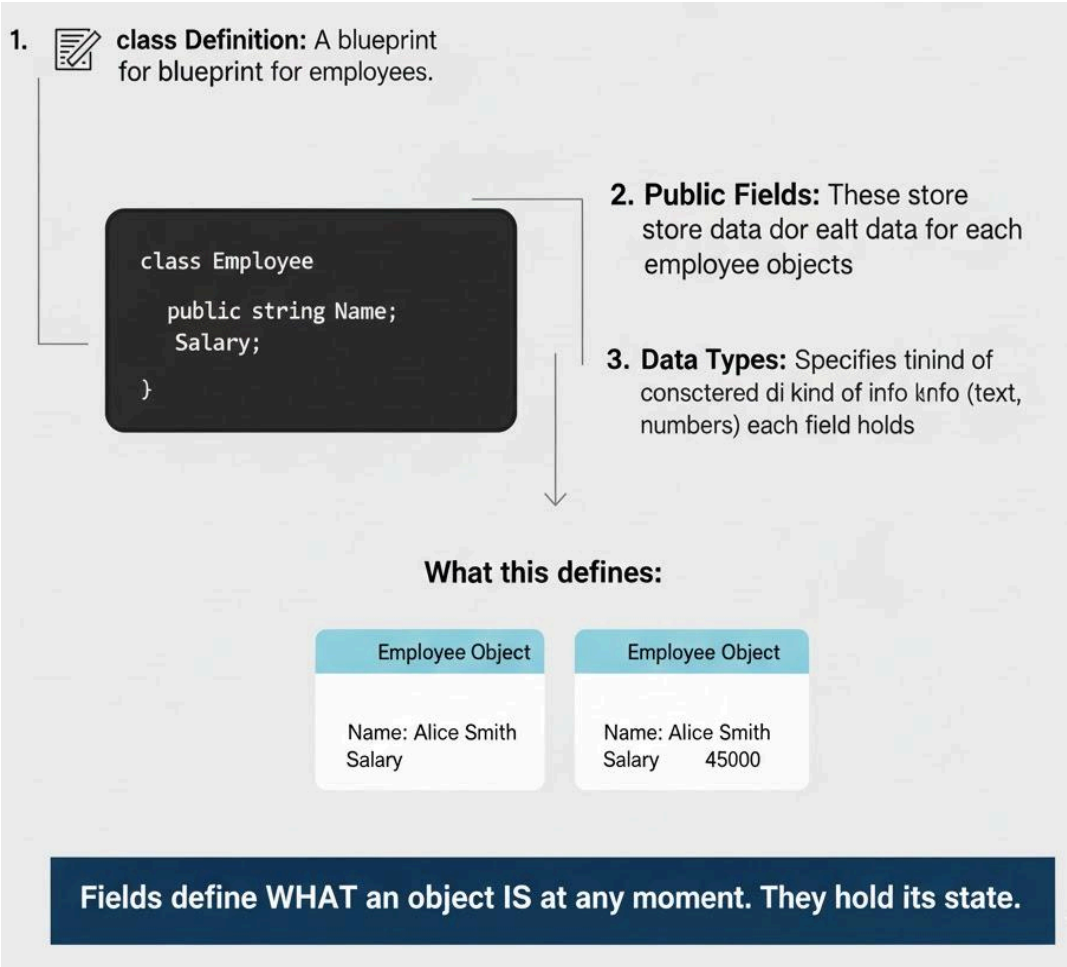
An **Employee** in real life has:

- Name
- Salary
- Employee ID
- Department
- Experience

These details describe the **state** of an employee.

Fields Example (Programming)

```
class Employee
{
    public string Name;
    public double Salary;
}
```



Here:

- `Name` and `Salary` are **fields**
- They store employee-related data

Methods (Member Functions)

Methods are blocks of code that define **what an object can do**. They operate on the object's fields and represent the **behavior or actions** of the object.

Characteristics of Methods

- Define object behavior
- Can access and modify fields
- Improve code organization
- Promote reusability
- Can accept parameters and return values

Real-Life Example (Employee Actions)

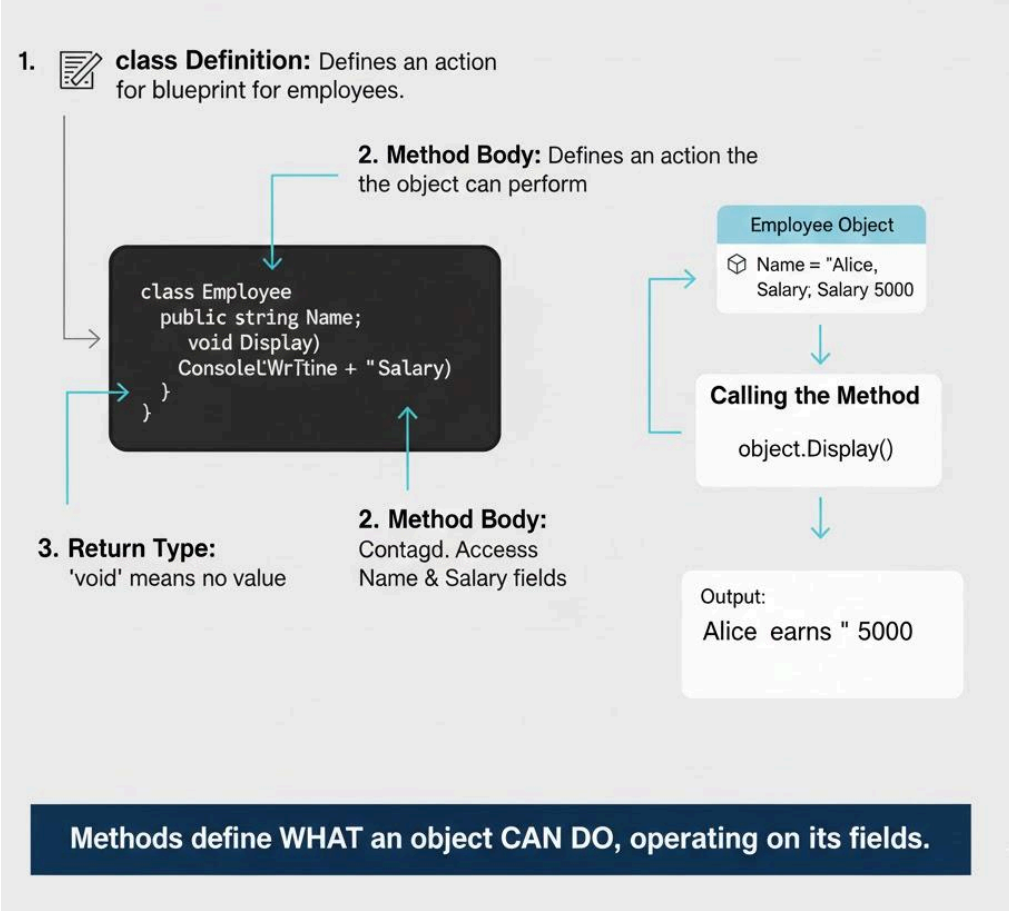
An **Employee** can:

- Display details
- Calculate salary
- Apply for leave
- Get promoted

These are **behaviors**, represented as methods in OOP.

Methods Example (Programming)

```
class Employee
{
    public string Name;
    public double Salary;
    public void Display()
    {
        Console.WriteLine(Name + " earns " + Salary);
    }
}
```



Here:

- `Display()` is a **method**
- It accesses fields `Name` and `Salary`
- It performs an action using object data

Fields + Methods Working Together

```
Employee emp1 = new Employee();
emp1.Name = "Rahul";
emp1.Salary = 50000;
emp1.Display();
```

Output:

```
Rahul earns 50000
```

Scenario-Based Question

Q: An employee system stores employee details and prints salary slips.
➡ **Fields:** Name, Salary
➡ **Methods:** DisplaySalarySlip()

---

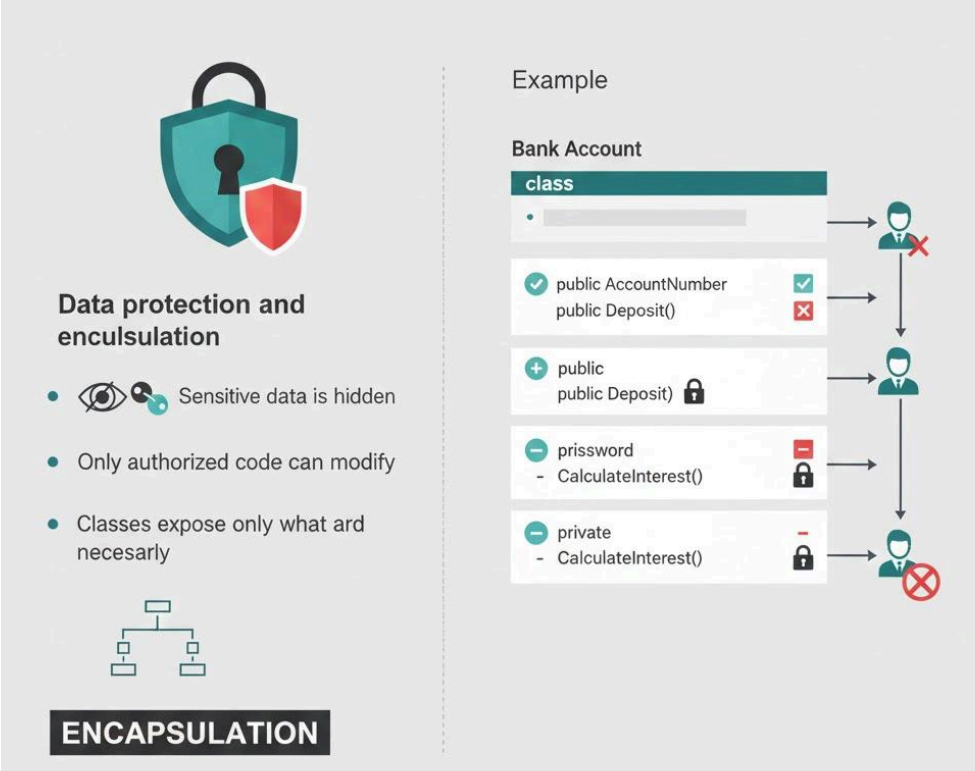
3. Access Modifiers

Access modifiers in Object-Oriented Programming control **who can access class members** such as fields and methods.
They help in **data protection**, **security**, and **proper encapsulation** by restricting unwanted access.

Using access modifiers ensures that:

- Sensitive data is hidden
- Only authorized code can modify important values
- Classes expose only what is necessary

This is a core principle of **Encapsulation** in OOP.



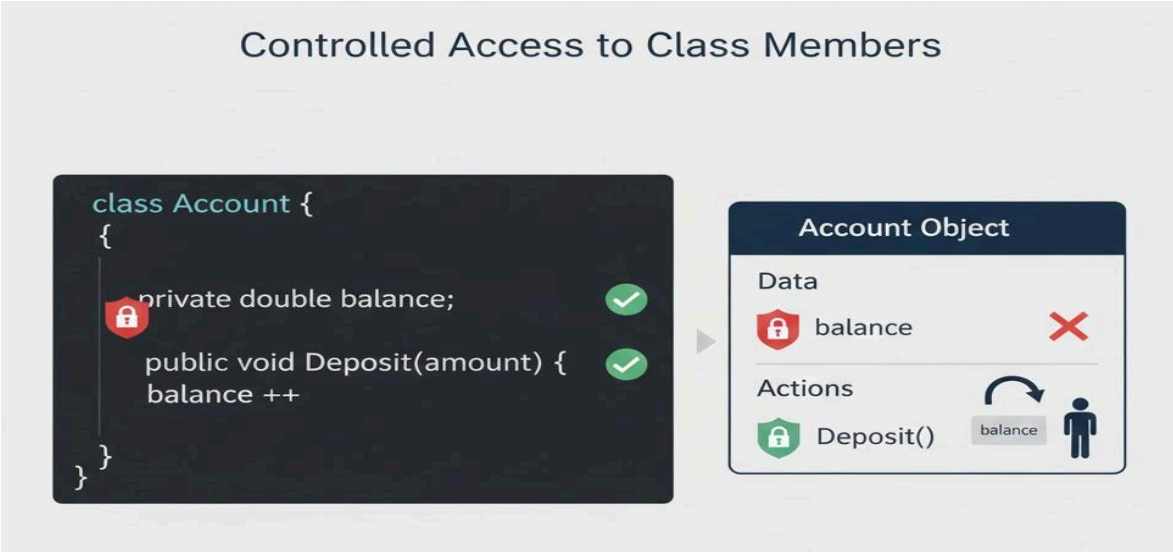
Common Access Modifiers in C#

Modifier	Accessibility
`public`	Accessible from anywhere
`private`	Accessible only within the same class
`protected`	Accessible within the class and derived (child) classes
`internal`	Accessible within the same assembly/project

Code Example

```
class Account
{
   private double balance;

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



Explanation

- `balance` is marked **private** → cannot be accessed directly from outside
- `Deposit()` is **public** → controlled access to modify balance
- This prevents unauthorized balance changes

Why Access Modifiers Are Important

- Prevent accidental misuse of data
- Improve security
- Make code easier to maintain
- Enforce business rules

Daily Life Examples (Easy to Remember)

Real Life Item	Access Modifier
ATM PIN	`private`
Bank account balance	`private`
Account number	`public`
Debit card	`public`
Bank internal policy	`internal`
Family inheritance rules	`protected`
Locker key	`private`
Customer service helpline	`public`

---

4. Static, Constant & Readonly Fields

In C#, data members can have different lifetimes and purposes. Static fields belong to the class itself and are shared by all objects, such as a bank name common to every account. Const fields represent values that never change and are fixed at compile time, like mathematical constants. Readonly fields allow values to be assigned only once, usually during object creation, ensuring data safety while supporting object-specific values.

Static Fields

A **static field** belongs to the **class itself**, not to any specific object.
This means:

- Only **one copy** of the static field exists
- All objects of the class **share the same value**
- Static fields are accessed using the **class name**

Static fields are used for **common data** that should be the same for all objects.

Syntax Example

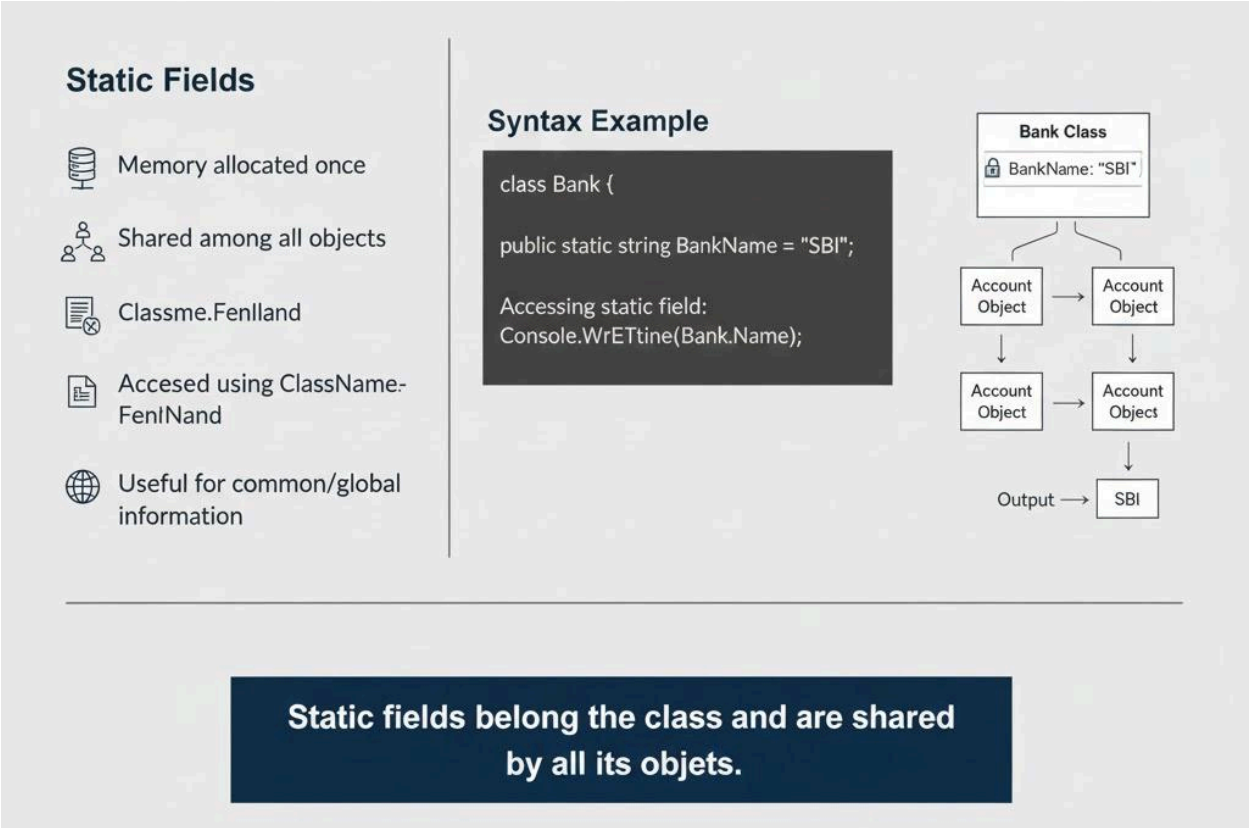
```
class Bank
{
    public static string BankName = "SBI";
}
```

Accessing static field:

```
Console.WriteLine(Bank.BankName);
```

Key Points (Static)

- Memory allocated **once**
- Shared among all objects
- Accessed using **ClassName.FieldName**
- Useful for common/global information



Real-Life Examples (Static)

- Country name → same for all citizens
- Bank name → same for all accounts
- Company name → same for all employees
- College name → same for all students
- GST percentage → same for all invoices

Constant Fields (`const`)

A **constant field** stores a value that **can never be changed**.
The value of a `const` field is:

- Fixed at **compile time**
- Must be initialized **at declaration**
- Automatically treated as **static**

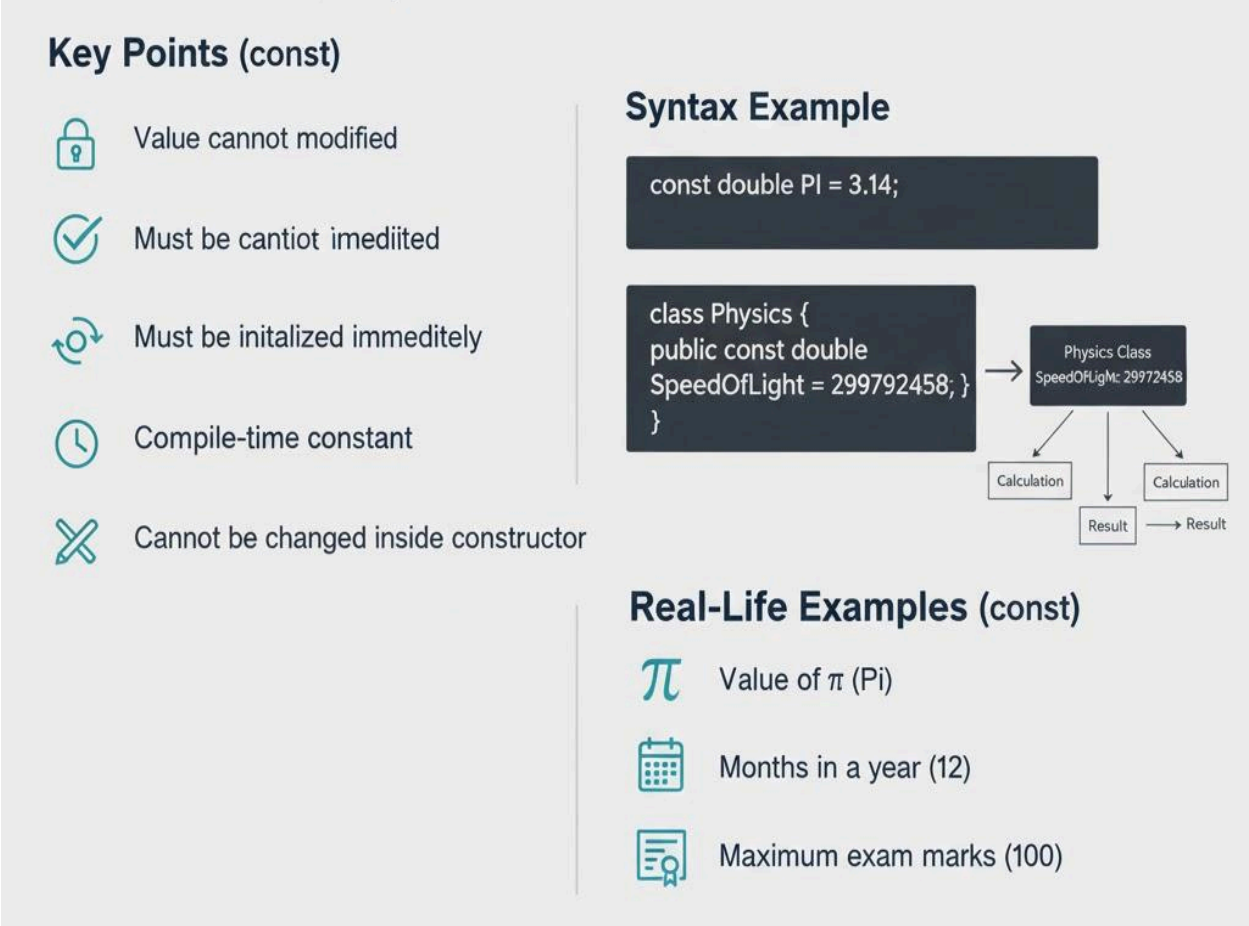
Constants are used for **universal values** that never change.

Syntax Example

```
const double PI = 3.14;
```

Key Points (const)

- Value cannot be modified
- Must be initialized immediately
- Compile-time constant
- Cannot be changed inside constructor



Real-Life Examples (const)

- Speed of light
- Value of π (Pi)
- Number of months in a year (12)
- Days in a week (7)
- Maximum allowed exam marks

Readonly Fields

A **readonly field** is a field whose value:

- Can be assigned **only once**
- Is usually set **inside a constructor**
- Cannot be changed later

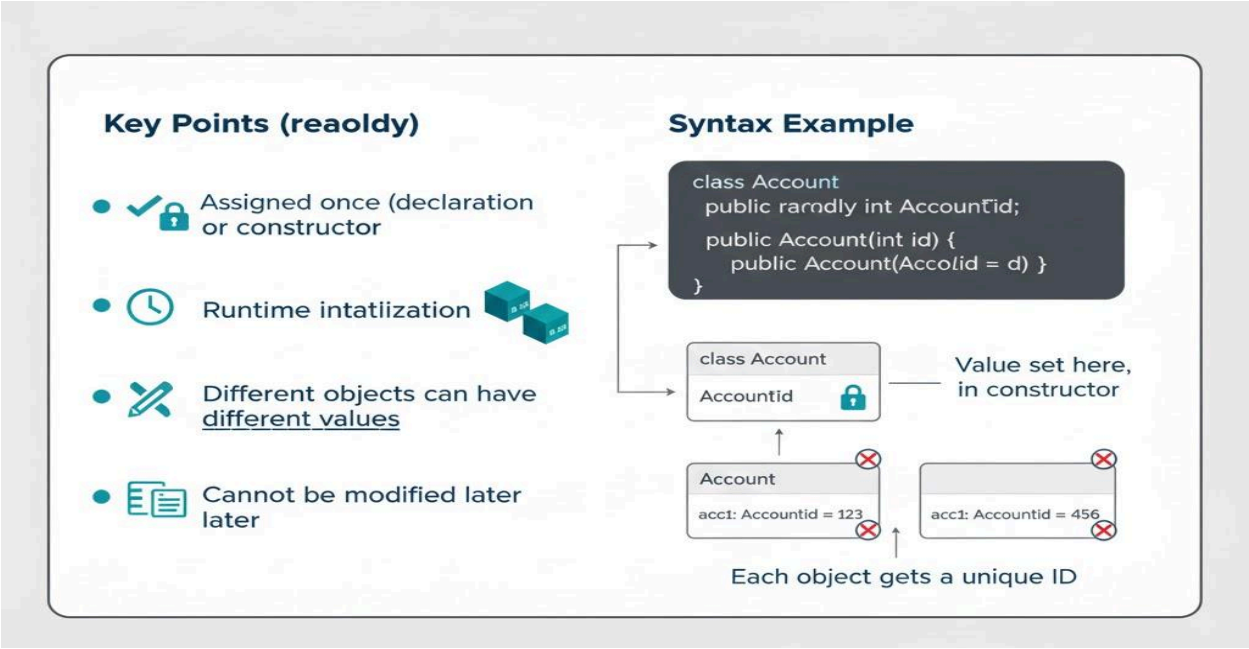
Readonly fields are used when:

- Value is **different for each object**
- But should **never change after initialization**

Syntax Example

```
class Account
{
    public readonly int AccountId;

    public Account(int id)
    {
        AccountId = id;
    }
}
```



Key Points (readonly)

- Assigned once (declaration or constructor)
- Runtime initialization
- Different objects can have different values
- Cannot be modified later

Real-Life Examples (readonly)

- Aadhaar number
- PAN card number
- Employee ID
- Roll number
- Vehicle chassis number

Comparison Table

Feature	static	const	readonly
Belongs to	Class	Class	Object
Value change	Yes	No	No (after init)
Initialization	Anytime	Declaration only	Constructor
Compile-time	No	Yes	No
Object-specific	No	No	Yes

When to Use What?

- Use **static** → when data is common for all objects
- Use **const** → when value should never change
- Use **readonly** → when value must be set once per object

---

5. Methods & Encapsulation

Encapsulation is one of the core principles of Object-Oriented Programming (OOP).
It means **wrapping data (fields) and methods (functions) together into a single unit (class)** and **restricting direct access to the data**.

In encapsulation:

- Data is **hidden** from outside access
- Access is provided only through **controlled methods**
- This improves **security, reliability, and maintainability**

Why Encapsulation Is Needed

Encapsulation is used to:

- Protect sensitive data
- Prevent accidental modification
- Enforce validation rules
- Control how data is accessed or changed

Example (Programming)

```
class Wallet
{
    private double money;

    public void AddMoney(double amount)
    {
        money += amount;
    }

    public double GetBalance()
    {
        return money;
    }
}
```

Explanation of Code

- `money` is marked as **private**
- It cannot be accessed directly from outside the class
- `AddMoney()` is used to modify the balance safely

- `GetBalance()` allows reading the balance without changing it

This ensures **controlled access** to data.

Scenario-Based Explanation

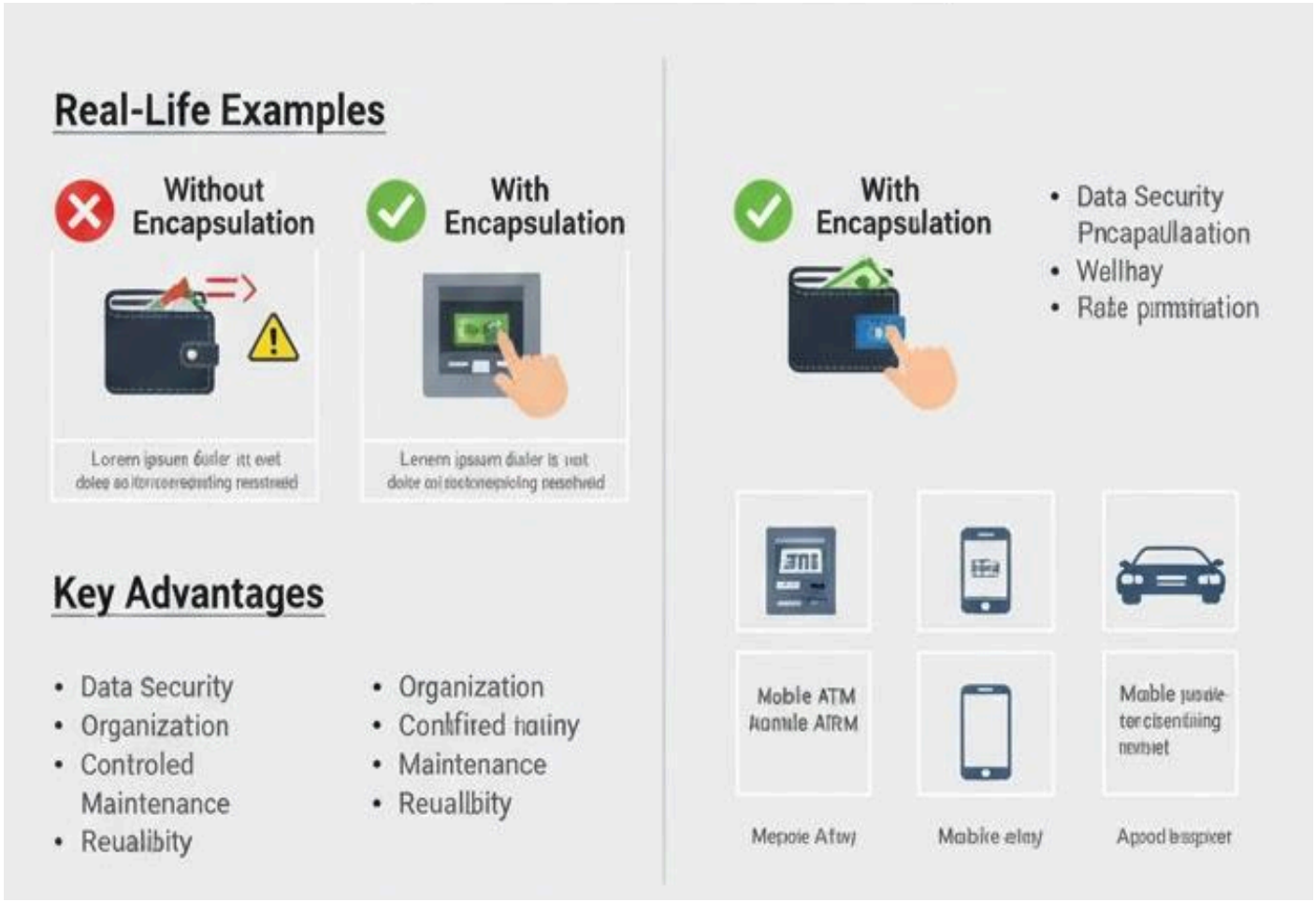
Without Encapsulation
Anyone could directly change the wallet balance:

```
wallet.money = -1000;
```

This would cause incorrect or unsafe behavior.

With Encapsulation
The user must use methods, allowing validation and control:

```
wallet.AddMoney(500);
```



Real-Life Examples

- **ATM Machine**
    - User cannot directly access account balance
    - Balance is shown only through secure system operations
- **Online Banking App**
    - User cannot edit balance manually
    - Transactions go through verified processes
- **Mobile Phone**
    - Battery is hidden inside
    - User interacts only through buttons and settings
- **Car Engine**
    - Driver does not directly control fuel injection
    - Uses accelerator and brake (methods)

Security Perspective

Encapsulation:

- Prevents unauthorized access
- Protects sensitive information
- Ensures data consistency
- Plays a key role in **secure software design**

Key Advantages of Encapsulation

- Data Security
- Better Code Organization
- Controlled Data Modification
- Easier Maintenance
- Improved Reusability

---

6. Static Methods

Static methods are methods that **belong to the class itself**, not to any specific object. They can be called **directly using the class name**, without creating an object.

Static methods are mainly used for **common functionality** that does not depend on object-specific data.

Key Characteristics of Static Methods

- Belong to the **class**, not an instance
- Can be called using **ClassName.MethodName()**
- Cannot access **non-static (instance) fields or methods directly**
- Commonly used for **utility and helper operations**
- Improve **memory efficiency** (no object creation needed)

Example (Programming)

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

int sum = Calculator.Add(5, 10);
```

Explanation:

- `Add()` is a static method
- No `Calculator` object is created
- Method is accessed using the class name

Why Use Static Methods?

Static methods are ideal when:

- The operation is **independent of object data**
- Logic is **common and reusable**
- You want to avoid unnecessary object creation

Common Use Cases of Static Methods

✔ Utility methods
✔ Mathematical calculations
✔ Validation logic
✔ Conversion functions
✔ Configuration helpers
✔ Logging utilities
✔ Authentication helpers
✔ Formatting operations

More Programming Examples

Validation Logic

```
class Validator
{
    public static bool IsValidAge(int age)
    {
        return age >= 18;
    }
}
```

// Output when used in a program:

// Example 1: Accessing the static method using the class name

```
bool result1 = Validator.IsValidAge(25);

Console.WriteLine(result1); // Output: True
```

// Example 2: Checking an invalid age

```
bool result2 = Validator.IsValidAge(15);

Console.WriteLine(result2); // Output: False
```

// Note: Since IsValidAge is static, you do not need to create an object of the Validator class.

Conversion Utility

```
class Converter
{
    public static double CelsiusToFahrenheit(double c)
    {
        return (c * 9 / 5) + 32;
    }
}
```

// Output when used in a program:

```
// Example 1: Converting 0 degrees Celsius
double f1 = Converter.CelsiusToFahrenheit(0);
Console.WriteLine($"0°C is {f1}°F"); // Output: 0°C is 32°F

// Example 2: Converting 100 degrees Celsius
double f2 = Converter.CelsiusToFahrenheit(100);
Console.WriteLine($"100°C is {f2}°F"); // Output: 100°C is 212°F

// Example 3: Converting room temperature (20 degrees Celsius)
double f3 = Converter.CelsiusToFahrenheit(20);
Console.WriteLine($"20°C is {f3}°F"); // Output: 20°C is 68°F

// Note: Since CelsiusToFahrenheit is static, you access it directly using the class name (Converter), without needing to create an object.
```

Logging

```
class Logger
{
    public static void Log(string message)
    {
        Console.WriteLine("LOG: " + message);
    }
}
```

// Output when used in a program:

```
// Example 1: Logging a simple event
Logger.Log("Application started successfully.");
// Output: LOG: Application started successfully.

// Example 2: Logging an error message
```

```
Logger.Log("ERROR: Database connection failed.");
// Output: LOG: ERROR: Database connection failed.

// Example 3: Logging a user action
Logger.Log("User 'Alice' logged in at 17:57.");
// Output: LOG: User 'Alice' logged in at 17:57.

// Note: Since Log is static, it provides a simple, centralized way to record messages
// across your application without creating a Logger object every time.
```

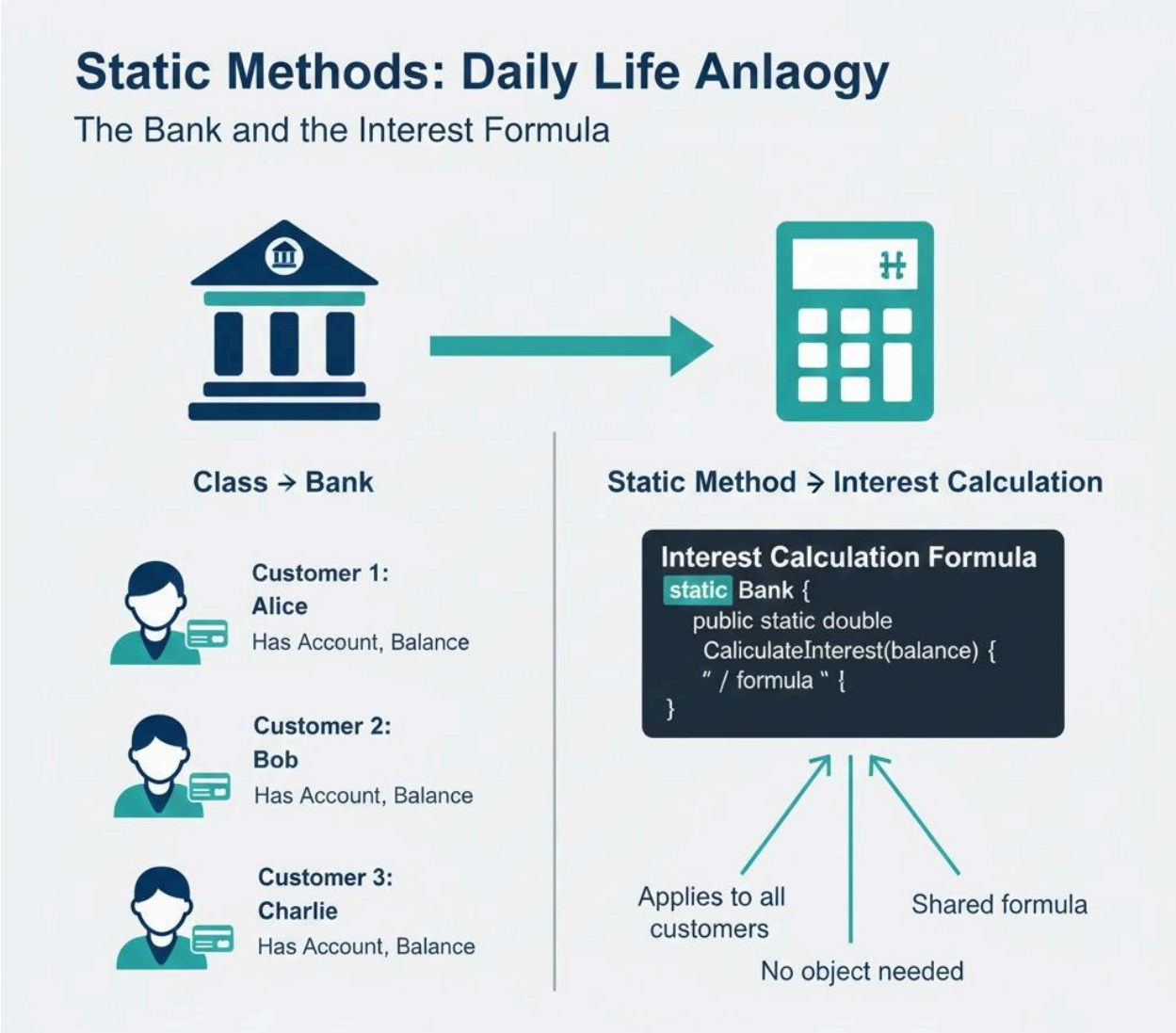
Real-Life Examples

Real Life	Static Method Meaning
Calculator in mobile	No personal data needed
Traffic signal rules	Same for everyone
ATM service charges	Fixed logic
Exam grading formula	Common calculation
Currency conversion rate	Universal logic

Daily Life Analogy

- **Class** → Bank
- **Static Method** → Interest calculation formula
- Same formula applies to **all customers**

No need to create a separate object for each customer just to calculate interest.



Static Method vs Instance Method

Feature	Static Method	Instance Method
Belongs to	Class	Object
Object required	No	Yes
Access instance data	No	Yes
Usage	Common logic	Object-specific behavior

---

7. Method Parameters & Overloading

Method Overloading

Method Overloading is a feature of **Object-Oriented Programming (OOP)** in which **multiple methods have the same name but different parameter lists** within the **same class**.

The difference in parameter list can be based on:

- Number of parameters
- Data type of parameters
- Order of parameters

The compiler decides **which method to execute at compile time** based on the arguments passed.

Same method name, different parameter list (type / count / order)
✔ Improves readability
✔ Makes code intuitive

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

    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }

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

Usage

```
MathOps m = new MathOps();

Console.WriteLine(m.Add(2, 3));          // 5
Console.WriteLine(m.Add(2, 3, 4));       // 9
Console.WriteLine(m.Add(2.5, 3.5));      // 6.0
```

Note:
Method overloading **cannot** be done by return type alone.

Why Method Overloading Cannot Be Done by Return Type Alone

Invalid Example

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

{

    return a + b;

}


double Add(int a, int b)

{

    return a + b;

}
```

Reason:

- Both methods have **same name**

- Same **parameter list**
- Only return type is different

The compiler cannot decide which method to call, because:

- Method calls do **not include return type**
- Return type is checked **after method selection**

Hence, **return type alone is not sufficient** for overloading.

Required Parameter in C#

Meaning

A **required parameter** is a method parameter **that must be given a value when the method is called**.

If a required parameter is **not provided**, the program will **not compile** and will show an error.

Why It Is Called “Required”

Because:

- The method **cannot work without that value**
- C# does **not provide any default value** for it
- The caller **must pass an argument**

Basic Example

```
void ShowName(string name)

{

    Console.WriteLine(name);

}
```

Here:

- `name` is a **required parameter**
- It has **no default value**

Default Arguments

Default values are used when arguments are not passed.

```
void PrintBill(string item, int qty = 1)
{
    Console.WriteLine(item + " x " + qty);
}
```

Usage

```
PrintBill("Pen");        // Pen x 1
PrintBill("Notebook", 5); // Notebook x 5
```

Default parameters must come **after required parameters**

Named Arguments

Allows passing parameters by **name**, not order.

```
void DisplayStudent(string name, int age, string city)
{
    Console.WriteLine($"{name}, {age}, {city}");
}
```

Usage

```
DisplayStudent(age: 20, city: "Delhi", name: "Amit");
```

✔ Order does not matter
✔ Improves readability

Default + Named Arguments (Combined)

```
void OrderFood(string food, int quantity = 1, bool takeAway = false)
{
    Console.WriteLine($"{food} x {quantity}, TakeAway: {takeAway}");
}
```

Usage

```
OrderFood("Burger");
OrderFood("Pizza", quantity: 2);
OrderFood(food: "Sandwich", takeAway: true);
```

`params` Keyword (Variable Arguments)

Allows passing **multiple values**.

```
int Sum(params int[] numbers)
{
    int total = 0;
    foreach (int n in numbers)
        total += n;
    return total;
}
```

Usage

```
Console.WriteLine(Sum(1, 2));
Console.WriteLine(Sum(1, 2, 3, 4));
```

---

8. ref, out, in, params

These keywords control **how data is passed to methods** in C#.

`ref` – Pass by Reference

Changes made inside the method **affect the original variable**.

Example

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

Usage

```
int a = 20;
Increase(ref a);
Console.WriteLine(a); // 30
```

Rules

- Variable **must be initialized** before passing
- Method **must use `ref`** in both declaration and call

Why `ref`?

Short Answer

We use `ref` **when we want a method to change the value of a variable permanently**.

Without `ref`, the change happens **only inside the method**, not outside.

First: What happens normally (WITHOUT `ref`)

By default, C# uses **pass by value**.

That means:

- A **copy** of the variable is sent to the method
- The original variable stays unchanged

Example (Without `ref`)

```
void Increase(int x)

{

    x = x + 10;

}


int a = 20;

Increase(a);

Console.WriteLine(a);   // 20
```

What really happens?

Think like this:

- `a = 20`
- Method receives **copy** → `x = 20`

- `x` becomes 30
- Method ends
- Copy is destroyed
- Original `a` is still 20

So the change is **temporary**.

Now: What happens WITH `ref`

When you use `ref`, you are saying:

> "Don't send a copy. Send the original variable itself."

Example (With `ref`)

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

int a = 20;
Increase(ref a);
Console.WriteLine(a);   // 30
```

What really happens?

- `a` and `x` point to the **same memory**

- Change `x` → change `a`
- Change is **permanent**

`out` – Output Parameters

Used to **return multiple values** from a method.

Example

```
void GetResult(out int total)
{
    total = 100;
}
```

Usage

```
int result;
GetResult(out result);
Console.WriteLine(result); // 100
```

Rules

- Variable **need not be initialized**
- Method **must assign a value** before returning

`ref` vs `out` (Quick Comparison)

Feature	ref	out
Initialization required	Yes	No

Value must be set inside method	No	Yes
Purpose	Modify existing value	Return value

`in` – Read-Only Reference

Passes variable by reference **without allowing modification**.

Example

```
void Print(in int x)
{
    Console.WriteLine(x);
}
```

Usage

```
int num = 50;
Print(in num);
```

Rules

- Cannot modify `x` inside the method
- Improves performance for large structs
- Introduced in **C# 7.2**

Invalid

```
x = x + 10; // Compile-time error
```

`params` – Variable Number of Arguments

Allows passing **any number of values** to a method.

Example

```
int Sum(params int[] numbers)
{
    int total = 0;
    foreach (int n in numbers)
        total += n;
    return total;
}
```

Usage

```
Console.WriteLine(Sum(1, 2));          // 3
Console.WriteLine(Sum(1, 2, 3, 4));    // 10
```

Rules

- Only **one** `params` allowed per method
- Must be the **last parameter**

Real-World Examples

Banking Example (`ref`)

```
void Deposit(ref int balance, int amount)
{
    balance += amount;
}
```

Student Result (`out`)

```
void CalculateMarks(out int total, out string result)
{
    total = 450;
    result = "Pass";
}
```

Configuration (`in`)

```
void ShowConfig(in string appName)
{
```

```
    Console.WriteLine(appName);
}
```

Shopping Cart (`params`)

```
double CalculateBill(params double[] prices)
{
    double sum = 0;
    foreach (double p in prices)
        sum += p;
    return sum;
}
```

---

9. Local & Static Local Functions

Local functions are **methods defined inside another method**.They help in **code organization, encapsulation, and readability**.

Local Functions

A local function can **access variables of the parent method**.

Example

```
void Process()
{
    string status = "Processing...";

    void PrintMsg()
    {
        Console.WriteLine(status);
    }

    PrintMsg();
}
```

Output

```
Processing...
```

Features

- Defined inside a method
- Can access local variables (closures)
- Improves readability
- Not accessible outside the parent method

Multiple Local Functions

```
void Calculator()
{
    int Add(int a, int b)
    {
        return a + b;
    }

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

    Console.WriteLine(Add(2, 3));      // 5
    Console.WriteLine(Multiply(2, 3)); // 6
}
```

Local Function vs Lambda Expression

Local functions are **better for complex logic**.

```
void Example()
{
    int Square(int x)
    {
        return x * x;
    }
    Func<int, int> squareLambda = x => x * x;

    Console.WriteLine(Square(4));
    Console.WriteLine(squareLambda(4));
```

```
}
```

✔ Local functions support:

- `ref`, `out`, `in`
- `yield return`
- Better debugging

Static Local Functions

Static local functions **cannot access outer variables**.

Example

```
void Calculate()
{
    int number = 5;

    static int Square(int x)
    {
        return x * x;
    }

    Console.WriteLine(Square(number));
}
```

Rules

- No access to parent variables
- No closures
- Better performance
- Introduced in **C# 8.0**

Invalid

```
static int Square()
{
    return number * number; // Compile-time error }
```

When to Use Static Local Functions

- Helper logic inside a method
- Performance-critical code
- Avoid accidental variable capture

Real-World Examples

Validation Logic

```
void RegisterUser(string email)
{
    bool IsValidEmail(string e)
    {
        return e.Contains("@");
    }

    Console.WriteLine(IsValidEmail(email));
}
```

Mathematical Utility

```
void Geometry()
{
    static double Area(double r)
    {
        return 3.14 * r * r;
    }

    Console.WriteLine(Area(5));
}
```

---

10. Recursion

Recursion is a technique where a method **calls itself** to solve a problem by breaking it into **smaller sub-problems**.

Basic Structure of Recursion

Every recursive method has **two parts**:

1. **Base Condition** – stops recursion
2. **Recursive Call** – function calls itself

```
int Factorial(int n)
{
    if (n == 1)        // Base case
        return 1;
    return n * Factorial(n - 1); // Recursive call
}
```

Usage

```
Console.WriteLine(Factorial(5)); // 120
```

How Recursion Works (Call Stack)

For `Factorial(4)`:

```
Factorial(4)
 → 4 * Factorial(3)
     → 3 * Factorial(2)
         → 2 * Factorial(1)
             → 1
```

Then values return back **one by one**.

Important Rules

Must have a **base condition**
Each call should move **closer to base case**
Missing base case → **StackOverflowException**

Common Recursion Examples

Fibonacci Series

```
int Fibonacci(int n)
{
    if (n <= 1)
        return n;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Sum of Numbers

```
int Sum(int n)
{
    if (n == 0)
        return 0;
    return n + Sum(n - 1);
}
```

Power Calculation

```
int Power(int baseNum, int exp)
{
    if (exp == 0)
        return 1;
    return baseNum * Power(baseNum, exp - 1);
}
```

Real-World / Daily Life Examples

Scenario	Explanation
Mirror reflection	Reflection inside reflection
File folder navigation	Folder → subfolder → file
Menu → submenu	Nested UI structure
Company hierarchy	Manager → employee

Recursion vs Loop

Feature	Recursion	Loop
Code simplicity	Cleaner	Slightly longer
Memory usage	Higher (stack)	Lower
Performance	Slower	Faster
Best for	Tree, DFS, backtracking	Repetition

Tail Recursion (Concept)

Recursive call is the **last statement**.

```
int FactorialTail(int n, int result = 1)
{
    if (n == 1)
        return result;
    return FactorialTail(n - 1, n * result);
}
```

Easier to optimize (conceptually).

When to Use Recursion

✔ Tree traversal
✔ Graph algorithms
✔ Divide & conquer (merge sort)
✔ Backtracking problems

When NOT to Use Recursion

Simple loops
Very deep calls
Performance-critical code

Type Conversion

Implicit Casting

Type conversion is the process of **changing one data type into another**.

Automatically done by C#
Small data type → Large data type. No data loss

```
int a = 10;
double b = a;
Console.WriteLine(b); // 10
```

Examples

```
int → long
float → double
char → int
```

Explicit Casting

Manually done by programmer
Large data type → Small data type. Possible data loss

```
double x = 10.5;
int y = (int)x;

Console.WriteLine(y); // 10
```

Decimal part is **removed**, not rounded.

`Parse()`

Converts **string to numeric type**. Throws **exception** if conversion fails.

```
int n = int.Parse("100");
Console.WriteLine(n); // 100
```

Invalid : `int n = int.Parse("abc"); // Runtime error`

`TryParse()`

Safe conversion method.
✔ Does not throw exception
✔ Returns `true` or `false`

```
int value;
bool success = int.TryParse("abc", out value);

Console.WriteLine(success); // False
Console.WriteLine(value);   // 0
```

Preferred in real applications

Convert Class

Another way to convert types.

```
int num = Convert.ToInt32("50");
double d = Convert.ToDouble("12.5");
```

Handles `null` better than `Parse()`.

Boxing & Unboxing (Intro)

```
int a = 10;
object obj = a;        // Boxing

int b = (int)obj;     // Unboxing
```

Real-Life Examples

User Input Conversion

```
Console.Write("Enter age: ");
int age = int.Parse(Console.ReadLine());
```

Safe Input Handling

```
int age;
if (int.TryParse(Console.ReadLine(), out age))
```

```
{
    Console.WriteLine("Valid age");
}
else
{
    Console.WriteLine("Invalid input");
}
```

Constructors

What is Constructor?

A special method that **initializes objects**.

Instance Constructor

```
class Student
{
    public string Name;

    public Student(string name)
    {
        Name = name;
    }
}
```

Static Constructor

```
class App
{
    static App()
    {
        Console.WriteLine("App started");
    }
}
```

Overloaded Constructors

```
class Product
{
    public string Name;
    public int Price;

    public Product() { }

    public Product(string name, int price)
    {
        Name = name;
        Price = price;
    }
}
```

Object Initializers

```
Product p = new Product
{
    Name = "Laptop",
    Price = 50000
};
```

Key Takeaways on Constructors

- Same name as class
- No return type
- Automatically called
- Used for initialization
- Can be overloaded
- Static constructor runs once

1-Hour Assignment

Requirements:

- Class: `BankAccount`
- Fields: AccountNo, Balance
- Methods: Deposit(), Withdraw(), Display()
- Use:
    - Encapsulation
    - Constructor
    - Static BankName
    - Method Overloading
    - ref / out
    - TryParse

Sample Tasks:

1. Create account
2. Deposit money
3. Withdraw money
4. Display balance
5. Prevent invalid inputs

Class Design

Class: `BankAccount`

Fields

- AccountNo
- Balance
- Static BankName

Methods

- Deposit()
- Withdraw()
- Display()

Complete Working Code

```
using System;

class BankAccount
{
    // Encapsulation (private fields)
    private int accountNo;
    private double balance;

    // Static Field
    public static string BankName = "ABC National Bank";

    // Constructor
    public BankAccount(int accNo, double initialBalance)
    {
        accountNo = accNo;
        balance = initialBalance;
    }

    // Method Overloading - Deposit
    public void Deposit(double amount)
```

```
{
    if (amount > 0)
    {
        balance += amount;
        Console.WriteLine("Amount deposited successfully.");
    }
    else
    {
        Console.WriteLine("Invalid deposit amount.");
    }
}

// Overloaded Deposit using ref
public void Deposit(ref double amount)
{
    if (amount > 0)
    {
        balance += amount;
        amount = 0; // cleared after deposit
        Console.WriteLine("Amount deposited using ref.");
    }
}

// Withdraw using out
public bool Withdraw(double amount, out string message)
{
    if (amount <= 0)
    {
        message = "Invalid withdrawal amount.";
        return false;
    }

    if (amount > balance)
    {
        message = "Insufficient balance.";
        return false;
    }
```

```
        balance -= amount;
        message = "Withdrawal successful.";
        return true;
    }

    // Display Account Details
    public void Display()
    {
        Console.WriteLine("---------------");
        Console.WriteLine($"Bank Name  : {BankName}");
        Console.WriteLine($"Account No : {accountNo}");
        Console.WriteLine($"Balance    : ₹{balance}");
        Console.WriteLine("---------------");
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine("Welcome to Banking System");

        // TryParse for safe input
        Console.Write("Enter Account Number: ");
        int accNo;
        while (!int.TryParse(Console.ReadLine(), out accNo))
        {
            Console.Write("Invalid input. Enter valid Account Number:
");
        }

        Console.Write("Enter Initial Balance: ");
        double initBalance;
        while (!double.TryParse(Console.ReadLine(), out initBalance))
        {
            Console.Write("Invalid input. Enter valid Balance: ");
        }
```

```
        // Create Account
        BankAccount account = new BankAccount(accNo, initBalance);

        int choice;
        do
        {
            Console.WriteLine("\n1. Deposit");
            Console.WriteLine("2. Withdraw");
            Console.WriteLine("3. Display Account");
            Console.WriteLine("4. Exit");
            Console.Write("Choose option: ");

            int.TryParse(Console.ReadLine(), out choice);

            switch (choice)
            {
                case 1:
                    Console.Write("Enter deposit amount: ");
                    double depAmount;
                    if (double.TryParse(Console.ReadLine(), out
depAmount))
                    {
                        account.Deposit(depAmount);
                    }
                    else
                    {
                        Console.WriteLine("Invalid amount.");
                    }
                    break;

                case 2:
                    Console.Write("Enter withdrawal amount: ");
                    double wAmount;
                    if (double.TryParse(Console.ReadLine(), out
wAmount))
                    {
                        if (account.Withdraw(wAmount, out string msg))
                            Console.WriteLine(msg);
```

```
                        else
                            Console.WriteLine(msg);
                    }
                    else
                    {
                        Console.WriteLine("Invalid amount.");
                    }
                    break;

                case 3:
                    account.Display();
                    break;

                case 4:
                    Console.WriteLine("Thank you for banking with
us!");
                    break;

                default:
                    Console.WriteLine("Invalid choice.");
                    break;
            }

        } while (choice != 4);
    }
}
```

Sample Output (Flow)

```
Welcome to Banking System
Enter Account Number: 101
Enter Initial Balance: 5000

1. Deposit
2. Withdraw
3. Display Account
4. Exit
```

```
Choose option: 1
Enter deposit amount: 2000
Amount deposited successfully.
```

How Requirements Are Met

Requirement	Covered
Encapsulation	Private fields + public methods
Constructor	Account initialization
Static	BankName
Overloading	Deposit()
ref	Deposit(ref amount)
out	Withdraw(out message)
TryParse	Safe user input
Validation	Negative / invalid prevention

Real-World Use Cases

✔ Banking software
✔ Employee payroll systems
✔ E-commerce wallet/cart
✔ Hospital billing
✔ Student fee management