What is OOP?

- **Paradigm**: Programming based on "`objects`“
- **Objects**: Contain data (attributes) and code (behaviors)
- **Contrast**: Procedural Programming (focus on functions)
- **Goal**: Model real-world entities and their interactions

Four Pillars of OOP

- **Encapsulation**: Bundling data and methods, hiding internal details.
- **Inheritance**: Creating new classes from existing ones.
- **Polymorphism**: Objects taking on many forms.
- **Abstraction**: Showing only essential information.

Encapsulation

- **Definition**: Wrapping data and methods into a single unit (class)
- **Purpose**: Protect data, reduce complexity
- **Example**

```
class BankAccount
{
    private decimal balance;
    public void Deposit(decimal amount) { balance += amount;}
    public decimal GetBalance() {return balance;}
}
```

Abstraction

- **Definition**: Hiding internal details and showing only essential features
- **Purpose**: Simplify interface, reduce complexity
- **Example:**

```
abstract class Animal
{
    public abstract void Speak();
}
class Dog : Animal
{
    public override void Speak() { Console.WriteLine("Bark");}
}
```

Inheritance

- **Definition**: Ability of one class to inherit from another class
- **Purpose**: Reuse code, hierarchical classification
- **Example**:

```
class Vehicle
{
    public void Start() { Console.WriteLine("Starting...");}
}
class Car : Vehicle
{
    public void Drive() { Console.WriteLine("Driving...");}
}
```

Polymorphism

- **Definition**: One interface, many implementations
- **Purpose**: Code flexibility and reusability
- **Example**:

```
class Animal
{
    public virtual void Speak() { Console.WriteLine("Animal sound");}
}
class Cat : Animal
{
    public override void Speak() { Console.WriteLine("Meow"); }
}
Animal pet = new Cat();
pet.Speak(); // Outputs: Meow
```

OOP Pillars

- Recap of Four Pillars:
  - **Encapsulation**: Protect data
  - **Abstraction**: Hide complexity
  - **Inheritance**: Reuse logic
  - **Polymorphism**: Reuse interface

Benefits of OOP

- **Modularity**: Code organized into self-contained units.
- **Reusability**: Classes can be reused in different parts of an application.
- **Maintainability**: Easier to debug and update.
- **Scalability**: Easier to extend and grow.
- **Real-World Modeling**: Intuitive representation of complex systems.

Class: The Blueprint

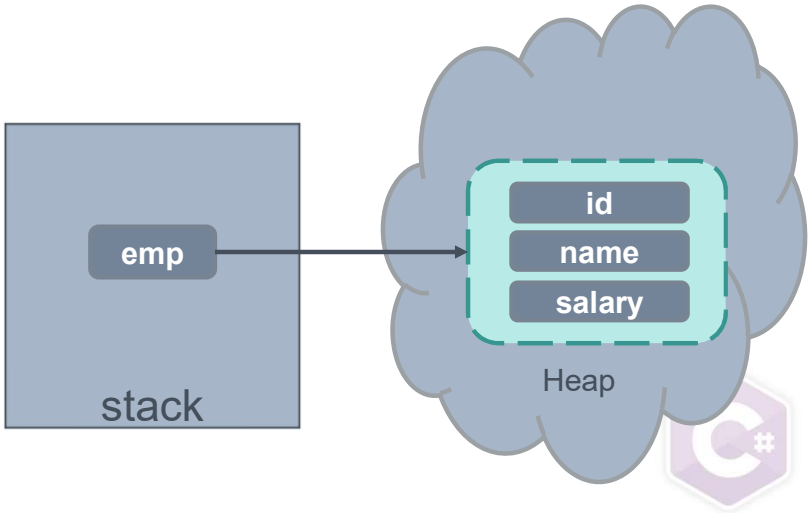
- **Definition**: A template or blueprint for creating objects.
- Defines structure (fields/data) and behavior (methods).
- **Example**: `Car` class defines what a car ***`is`*** (color, model) and ***`does`*** (start, stop).
- **Syntax**:

```
class ClassName
 {
     // Members (fields, properties, methods)
 }
```

Object: The Instance

- **Definition**: A concrete instance of a class.
- A real-world entity created from the class blueprint.
- **Example**: Your specific `red Honda Civic` is an object of the `Car` class.
- **Creation**: Using the `new` keyword.
- **Syntax**:

```
ClassName objectName = new ClassName();
```



Fields (Instance Variables)

- **Definition**: Variables declared directly within a class.
- Represent the state or data of an object.
- **Example**: `private string _model;`
- **Best Practice**: Keep fields `private` (encapsulation).

Methods (Behaviors)

- **Definition**: Functions defined within a class.
- Represent the actions an object can perform.
- **Example**: `public void Accelerate() { ... }`
- **Review**: Access modifiers, return types, parameters (from Day 1).
- `set` , `get` methods

Access Modifiers: Controlling Visibility

- **Purpose**: Control accessibility of class members.
- **`public`**: Accessible from anywhere.
- **`private`**: Accessible only within the defining class. (Default for members)
- **`protected`**: Accessible within defining class and by derived classes.

Access Modifiers: Controlling Visibility

- **`internal`**: Accessible only within the same assembly. (Default for classes)
- **`protected internal`**: Accessible within same assembly OR by derived classes.
- **`private protected`**: Accessible within defining class AND by derived classes in same assembly.

Properties: Encapsulated Access

- **Purpose**: Safe and flexible access to private fields.
- **Encapsulation**: Primary mechanism in C# for controlled data access.
- **Syntax**:

```
public DataType PropertyName
  {
      get { /* return field; */ }
      set { /* field = value; */ }
  }
```

- `get` accessor: Reads value.
- `set` accessor: Writes value (`value` keyword).

Properties: Types and Examples

- **Read-only Properties**: Only `get` accessor.
- **Auto-Implemented Properties (Auto-Properties):**
  - Shorthand when no custom logic needed.
  - Compiler creates private backing field.
  - Example: `public string Name { get; set; }`
  - Could be initialized `public float Salary { get; set; }=1000;`
  - Why??
- **Property Validation**: Add logic in `set`

  (e.g., `if (value < 0) throw ...`).
- **Caution**: properties ***can't*** be passed to method as `ref` or `out`

Constructors: Object Initialization

- **Definition**: Special methods called when an object is created.
- **Characteristics**:
  - Same name as the class.
  - No return type (not even `void`).
  - Automatically called by `new` .
- **Default Constructor**: Provided if no explicit constructor.
  - Auto initialize fields to default values
- **Parameterized Constructors**: Take arguments to initialize fields.

Constructors: Examples

```
public class Person
{
  public string Name { get; set; }
  public int Age { get; set; }
  // Parameterized Constructor
  public Person(string name, int age)
  {
   Name = name;
   Age = age;
  }
}
```

- **Constructor Overloading**: Multiple constructors with different parameters.
- **`this` keyword**: Refers to current instance; used for constructor chaining.

Instance Methods vs. Static Methods

- **Instance Methods**:
  - Operate on a specific object instance.
  - Access instance fields (`this`).
  - Example: `person.Walk();`
- **Static Methods**:
  - Belong to the class itself, not an object.
  - Called using class name: `Math.Sqrt()`, `Console.WriteLine()`.
  - Cannot access instance members directly.
  - Used for utility functions.

Method Overloading (Review)

- **Concept**: Multiple methods with the same name but different parameter lists.
- **Purpose**: Provide similar functionality for different inputs.
- **Example**:

```
 public int Add(int a, int b) { ... }
 public double Add(double a, double b)
{ ... }
```

Object Initializer

- Instantiate An Object (create an Object)
  - Through Constructor

```
Employee emp = new Employee();
Employee emp2 = new Employee(20,"Ahmed", 20000);
```

  - Through Object Initializer
    - Default constructor Called first then setting member variable

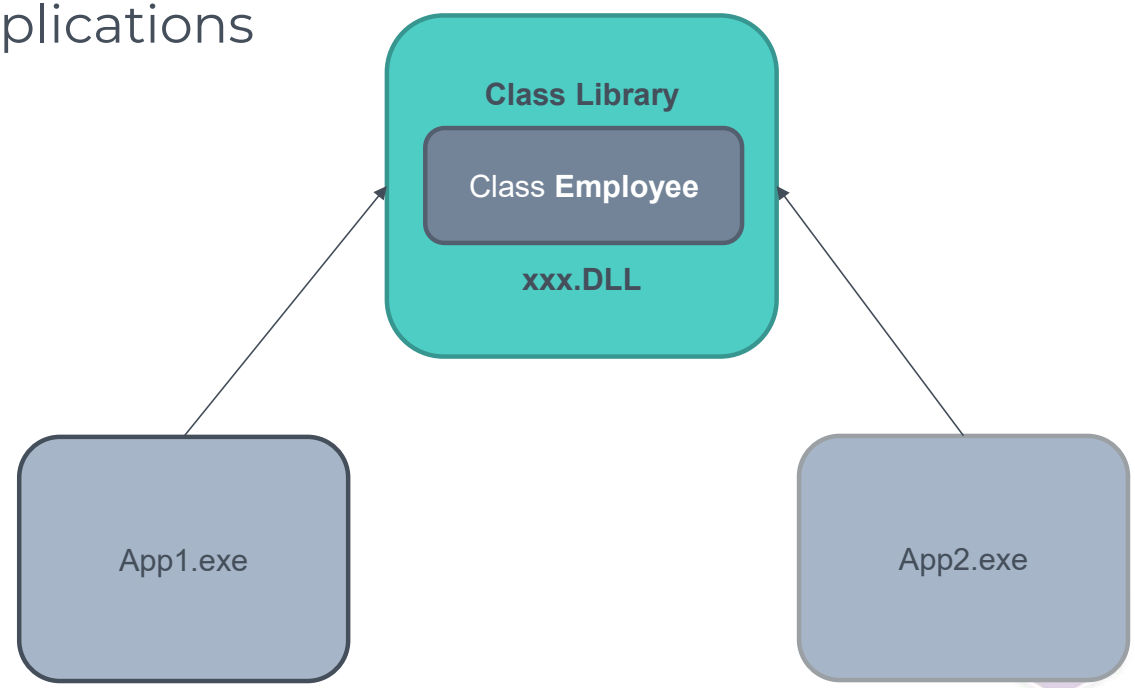
```
Employee emp = new Employee{id=20 ,name="Ahmed", salary=20000};
```

Namespace & Class Access Modifier

- Container for related data types
- Assembly (exe or DLL) could contain One namespace (at least) or more
- Namespace could contain namespace(s)
- For use a data type contained in namespace other than current namespace
  - Full name of data type *namespace* **.** *DatatypeName*
  - Using namespace;

Class Library

- A class library is used to enable sharing Data Types (ex: classes)among applications
- Demo
  - Creating class library
  - Using class library



Class Diagram

- Demo

Assignment

- Design a class represents Employee
  - Name
  - ID
  - Salary
  - DispalyData() method
  - Age as a property (18=< age <=60)
- Adding the employee class to class library and used in menu program

Assignment

- Design a class that represent a Stack Data Structure that contain
  - Data
    - Array of integers (to store values)
    - Size (init property)
    - Top_of_Stack
  - Actions
    - Push
      - full
    - Pop
      - empty