1. Encapsulation

Definition: Restricting direct access to data and exposing it through controlled methods or properties.

Real-Life Example:
Think of an **ATM machine**. You enter a PIN (input), and the system securely processes your transaction without exposing internal logic or data.

Code Example:

```
public class BankAccount
{
   private decimal balance;  // Encapsulation: Balance is private
   public void Deposit(decimal amount) => balance += amount;
   public decimal GetBalance() => balance; // Controlled access
}

// Usage
var account = new BankAccount();
account.Deposit(1000);
Console.WriteLine(account.GetBalance()); // Output: 1000
```

---

2. Inheritance

Definition: A child class inherits properties or methods from a parent class, promoting code reuse.

Real-Life Example:
A **Car** inherits generic properties like wheels and engine from a parent **Vehicle** class while adding its own features.

Code Example:

```
public class Vehicle
{
   public void StartEngine() => Console.WriteLine("Engine started");
}

public class Car : Vehicle
{
   public void PlayMusic() => Console.WriteLine("Playing music");
}

// Usage
var car = new Car();
car.StartEngine();  // From Vehicle
car.PlayMusic();    // Specific to Car
```

---

3. Polymorphism

Definition: Methods behave differently based on the context, implemented through method overloading or overriding.

Real-Life Example:
A **printer** can print documents, images, or presentations, but the underlying processing varies depending on the input.

Code Example (Overloading):

```
public class Printer
{
   public void Print(string document) => Console.WriteLine($"Printing document: {document}");
   public void Print(int copies) => Console.WriteLine($"Printing {copies} copies");
}
```

Code Example (Overriding):

```
public class Printer
{
   public virtual void Print() => Console.WriteLine("Default printer");
}

public class LaserPrinter : Printer
{
   public override void Print() => Console.WriteLine("Laser printer");
}

// Usage
Printer printer = new LaserPrinter();
printer.Print(); // Output: Laser printer
```

---

4. Abstraction

Definition: Showing only essential details while hiding implementation. Achieved using abstract classes or interfaces.

Real-Life Example:
When you **book a cab** in an app, you only see "Car booked" and the driver's info; you don't know the algorithm behind it.

Code Example (Interface):

```
public interface ICab
{
   void BookCab();
}

public class Uber : ICab
{
   public void BookCab() => Console.WriteLine("Uber cab booked");
}

// Usage
ICab cab = new Uber();
cab.BookCab(); // Output: Uber cab booked
```

Aspect	Abstraction	Encapsulation

Definition	Hides implementation details, showing only essential features.	Restricts direct access to data by bundling it with methods.
Purpose	Focuses on "what" a class or method does.	Focuses on "how" data is protected and manipulated.
Implementation	Achieved using abstract classes, interfaces.	Achieved using access modifiers (private, public, etc.).
Real-Life Example	A car's dashboard showing the speedometer (abstracted functionality).	The engine components hidden inside the car (encapsulated logic).

Here's a **clear explanation** of **events and delegates** in C# with **real-life examples** suitable for an interview:

---

1. Delegates

Definition: A delegate is a type-safe function pointer that allows methods to be passed as parameters or assigned dynamically.

Real-Life Example:
Think of a **restaurant waiter** as a delegate. The waiter takes your order (method) and passes it to the chef (another method), regardless of what the order is.

Code Example:

```
public delegate void Notify(string message); // Define a delegate


public class Notifier
{
   public void NotifyCustomer(string message) => Console.WriteLine($"Notification: {message}");
}


public class Program
{
   public static void Main()
   {
      Notifier notifier = new Notifier();
      Notify notifyDelegate = notifier.NotifyCustomer; // Assign method to delegate
      notifyDelegate("Your order is ready!"); // Output: Notification: Your order is ready!
   }
}
```

---

2. Events

Definition: An event is a wrapper around a delegate that ensures only specific classes can invoke the delegate. It's used for notifying subscribers when something happens.

Real-Life Example:
Think of a **doorbell system**. The button acts as an event. When pressed, it notifies all the people inside the house (subscribers) that someone is at the door.

Code Example:

```
public delegate void DoorbellHandler(); // Define a delegate for the event


public class Door
{
   public event DoorbellHandler DoorbellPressed; // Declare the event
```

```
    public void PressDoorbell()
    {
        Console.WriteLine("Doorbell pressed!");
        DoorbellPressed?.Invoke(); // Notify all subscribers
    }
}

public class Resident
{
    public void AnswerDoor() => Console.WriteLine("Resident: Coming to open the door!");
}

public class Program
{
    public static void Main()
    {
        Door door = new Door();
        Resident resident = new Resident();

        door.DoorbellPressed += resident.AnswerDoor; // Subscribe to the event
        door.PressDoorbell(); // Output: Doorbell pressed! Resident: Coming to open the door!
    }
}
```

Key Differences

Feature	Delegate	Event
Definition	Type-safe function pointer.	Mechanism for notifying subscribers when triggered.
Invocation	Can be invoked directly from anywhere.	Only the class declaring it can invoke the event.
Use Case	Assign or execute a method dynamically.	Notify subscribers when something happens.

Combined Real-Life Analogy

- **Delegate:** The waiter taking orders and assigning them to the chef dynamically.
- **Event:** The restaurant bell (event) rings to notify all waiters that the food is ready.

CI/CD (Continuous Integration / Continuous Deployment)

- **CI**: Automates the integration of code (new features or fixes) frequently into a shared project.
- **CD**: Automatically deploys the integrated code to production after successful tests.

Real-Life Example:

- **CI**: Think of a **chef** preparing dishes one by one (code changes) and making sure each one is tested for taste (automated tests) before serving.
- **CD**: After a successful test, the dish (code) is directly served to customers (production environment).

Tools: Jenkins, AWS CodePipeline.

DevOps

- **Definition**: A culture where **development** and **operations** teams work together, automating manual tasks like deployment and testing.

Real-Life Example:

- In a **restaurant**, the **kitchen staff** (developers) and **waitstaff** (operations) work closely together. The waitstaff ensures the food is served to customers on time (deployment) while the kitchen continuously works on new orders (feature development).

Tools: Jenkins, Docker, Kubernetes.

AWS (Amazon Web Services)

- **Definition**: A cloud platform offering tools like computing, storage, and deployment services to help scale and automate processes.

Real-Life Example:

- **AWS** is like **a huge restaurant supply warehouse**, providing the restaurant (your app) with all the essential resources: ingredients (compute power), storage for leftovers (S3), and delivery trucks (networking).

Tools for CI/CD/DevOps on AWS: AWS CodePipeline, AWS CodeDeploy, EC2.

Together:

- **CI/CD** automates code integration and deployment.
- **DevOps** ensures developers and operations collaborate efficiently.
- **AWS** provides the infrastructure for automation and scaling.
