# 01\_Introduction

### **Introduction to Design Patterns**

Design patterns are a key concept in software engineering and often come up in interviews, especially alongside questions about interfaces, abstract classes, delegates, and other object-oriented programming (OOP) features.

### **Why Learn Design Patterns?**

* **Improves Confidence:** Understanding design patterns can boost your confidence in interviews and when applying these concepts in your projects.
* **Proven Solutions:** Design patterns offer time-tested solutions to common problems, making your code more reliable, scalable, and maintainable.

### **What are Design Patterns?**

Design patterns are reusable solutions to common problems encountered in day-to-day programming. They provide a template that can be applied to solve real-world programming challenges, particularly those related to object generation and integration.

### **History and Evolution of Design Patterns**

The concept of design patterns was popularized by the "Gang of Four" (GoF), authors of the book *"Design Patterns: Elements of Reusable Object-Oriented Software."* This book is divided into two parts:

1. **OOP Concepts:** The first part discusses the advantages and limitations of object-oriented programming.
2. **Design Patterns:** The second part introduces 23 classic software design patterns that have since become foundational in software development.

For more details, refer to the [Wikipedia article on Design Patterns](https://en.wikipedia.org/wiki/Design_Patterns).

### 

### 

### 

### **Types of Design Patterns**

The Gang of Four categorized design patterns into three types, based on the problems they address:

1. **Creational Design Patterns:**
   * **Purpose:** Deal with object creation and initialization.
   * **Benefit:** Provides flexibility in deciding which objects to create in a given situation.
   * **Examples:** Singleton, Factory, Abstract Factory.
2. **Structural Design Patterns:**
   * **Purpose:** Deal with class and object composition.
   * **Benefit:** Focuses on decoupling the interface from the implementation of classes and objects.
   * **Examples:** Adapter, Facade, Bridge.
3. **Behavioral Design Patterns:**
   * **Purpose:** Deal with communication between classes and objects.
   * **Examples:** Chain of Responsibility, Command, Interpreter.

This structured overview should help you better understand the role of design patterns in software development and their categorization. As you dive deeper into each pattern, try to understand the problems they solve and how you can apply them in your projects.

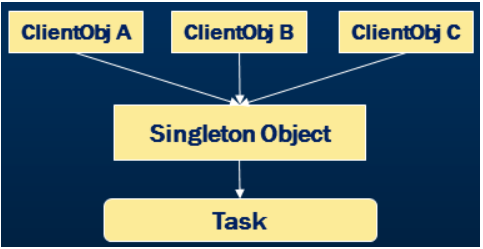
# 02\_Singleton Pattern

### **Singleton Pattern**

The Singleton Pattern is a part of the Creational design patterns category. As discussed, the Gang of Four (GoF) defined five creational design patterns: Singleton, Factory, Abstract Factory, Builder, and Prototype. Creational design patterns focus on object creation mechanisms, allowing objects to be created in a manner suitable to the given situation.

## What is the Singleton Design Pattern?

The Singleton Pattern ensures that only one instance of a particular class is created throughout the application. This single instance is responsible for coordinating actions across the application.



### **Diagram Overview**

Imagine multiple objects in an application trying to access a shared resource or service. By using a Singleton, we ensure that all these objects access the same instance, which manages the underlying methods or events.

### **Advantages of Singleton Design Pattern**

* **Controlled Access:** Ensures that concurrent access to a resource is well managed, preventing conflicts or inconsistent states.
* **Global Access Point:** Provides a single point of access to a resource across the application.

### **Implementation Guidelines for Singleton**

1. **Private Constructor:** All constructors should be private to prevent direct instantiation of the class.
2. **Static Instance Property:** Provide a static property that returns the single instance of the class, ensuring that only one instance is created.

### **Singleton Class Implementation Example**

Here's a C# implementation of the Singleton Pattern:

**Program.cs**

using System;

namespace SingletonDemo

{

class Program

{

static void Main(string[] args)

{

// Singleton created from an employee context

Singleton fromEmployee = Singleton.GetInstance;

fromEmployee.PrintDetails("From Employee");

// Singleton created from a student context

Singleton fromStudent = Singleton.GetInstance;

fromStudent.PrintDetails("From Student");

Console.ReadLine();

}

}

}

**Singleton.cs**

using System;

namespace SingletonDemo

{

public sealed class Singleton

{

private static int counter = 0;

// Private static instance initialized to null

private static Singleton instance = null;

// Public static property to return the single instance

public static Singleton GetInstance

{

get

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

// Private constructor to prevent direct instantiation

private Singleton()

{

counter++;

Console.WriteLine("Counter Value: " + counter.ToString());

}

// Public method that can be invoked through the Singleton instance

public void PrintDetails(string message)

{

Console.WriteLine(message);

}

}

}

### **Explanation:**

* **Singleton Class:** The class is marked as sealed to prevent inheritance, ensuring that the instance can't be extended or modified through inheritance.
* **Private Constructor:** The constructor is private, meaning the class cannot be instantiated directly.
* **Static Property GetInstance:** This property checks if the instance is null. If it is, a new instance is created. Otherwise, the existing instance is returned.
* **Usage in Main Program:** In Program.cs, two different references (fromEmployee and fromStudent) are both pointing to the same Singleton instance, demonstrating that only one instance is shared across the application.

## Why Seal the Singleton Class?

You might wonder why we need to seal the Singleton class when it already has a private constructor that prevents external instantiation. Let's explore this concept further.

### **Understanding the Role of the sealed Keyword**

* **Private Constructor:** A private constructor in the Singleton class restricts the creation of instances from outside the class. This means that the class cannot be instantiated from another class directly.
* **Inheritance and Private Constructor:** If you attempt to inherit the Singleton class in another class (e.g., DerivedSingleton), the compiler will throw an error, stating that the Singleton constructor is inaccessible due to its protection level. This error occurs because the private constructor prevents inheritance from outside the class.

### **Exploring the Need for sealed**

Given that the private constructor already restricts instantiation, why do we need to seal the class?

* **Nested Class Scenario:** Let's remove the sealed keyword from the Singleton class and create a nested class inside it. When a class is nested within another, it has access to the outer class's private members, including the private constructor.

**Creating a Nested Derived Class:**  
public class Singleton

{

private static int counter = 0;

private static Singleton instance = null;

// Private constructor

private Singleton()

{

counter++;

Console.WriteLine("Counter Value: " + counter.ToString());

}

public static Singleton GetInstance

{

get

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

// Nested derived class

public class DerivedSingleton : Singleton

{

}

}

* In this example, the DerivedSingleton class is nested within the Singleton class. Because it's nested, it has access to the private constructor of the Singleton class.

**Compiling and Running the Program:** When you compile and run this code, you'll find that it compiles successfully and allows the creation of a DerivedSingleton instance, which violates the Singleton pattern principles.  
  
class Program

{

static void Main(string[] args)

{

Singleton fromStudent = Singleton.GetInstance;

fromStudent.PrintDetails("From Student");

Singleton fromEmployee = Singleton.GetInstance;

fromEmployee.PrintDetails("From Employee");

Console.WriteLine("-------------------------------------");

// Instantiating Singleton from a derived class (violates Singleton principle)

Singleton.DerivedSingleton derivedObj = new Singleton.DerivedSingleton();

derivedObj.PrintDetails("From Derived");

Console.ReadLine();

}

}

Output:  
  
Counter Value: 1

From Student

From Employee

-------------------------------------

Counter Value: 2

From Derived

* The counter value incrementing to 2 shows that multiple instances of the Singleton are being created, which violates the Singleton principle.

### **Enforcing Singleton with sealed**

To prevent this violation, we must use the sealed keyword:

* **Effect of Sealing:** When the Singleton class is sealed, it can no longer be inherited by any class, including nested classes. This effectively prevents any other class from creating a new instance of the Singleton, preserving the Singleton principle.

### **Conclusion**

* **Private Constructor:** Prevents external instantiation of the class.
* **sealed Keyword:** Prevents the class from being inherited, ensuring the integrity of the Singleton pattern.

By sealing the Singleton class, you ensure that the Singleton instance is truly the only instance that can ever be created, even in more complex scenarios involving nested classes.

## Lazy Initialization in Singleton

Lazy Initialization is a design pattern used to delay the creation of an object until it is first needed. In the context of the Singleton pattern, the instance of the Singleton class is not created until the GetInstance property is accessed for the first time. This delay in instantiation is what we refer to as Lazy Initialization.

### **Lazy Initialization in Singleton:**

* **What it is:** The Singleton instance is created only when the GetInstance property is invoked for the first time.
* **Benefit:** Resources are not allocated until necessary, improving the efficiency of the program.

### **Multi-Threading in Singleton:**

While Lazy Initialization works well in a single-threaded environment, it can cause issues in a multi-threaded scenario. If multiple threads access the GetInstance property simultaneously, there is a risk of creating multiple instances of the Singleton, which violates the Singleton pattern's principle of having only one instance.

### **Thread Safety Issues:**

* **Thread Race Condition:** If two threads try to access the GetInstance property at the same time, they might both find that the Singleton instance has not yet been created and proceed to create their own instances, leading to multiple instances in memory.
* **Thread Safety Concerns:** This scenario creates thread safety issues, as the Singleton pattern's integrity is compromised.

### **Simulating Multi-Threading Issues:**

To demonstrate this issue, you can modify the Main program using the Parallel.Invoke method, which allows multiple threads to invoke GetInstance simultaneously:

using System;

using System.Threading.Tasks;

namespace SingletonDemo

{

class Program

{

static void Main(string[] args)

{

Parallel.Invoke(

() => PrintStudentDetails(),

() => PrintEmployeeDetails()

);

Console.ReadLine();

}

private static void PrintEmployeeDetails()

{

Singleton fromEmployee = Singleton.GetInstance;

fromEmployee.PrintDetails("From Employee");

}

private static void PrintStudentDetails()

{

Singleton fromStudent = Singleton.GetInstance;

fromStudent.PrintDetails("From Student");

}

}

}

### **Implementing a Thread-Safe Singleton Class:**

To address the thread safety issue, you can implement a thread-safe Singleton by using locks to control access to the instance creation process. This ensures that only one thread can enter the critical section of code that creates the instance.

### **Double-Checked Locking:**

The technique used here is called Double-Checked Locking. This approach checks whether the instance is null before and after acquiring the lock, ensuring that only one instance of the Singleton is created, even when multiple threads are involved.

using System;

namespace SingletonDemo

{

public sealed class Singleton

{

private static int counter = 0;

private static readonly object obj = new object();

private static Singleton instance = null;

// Private constructor to prevent external instantiation

private Singleton()

{

counter++;

Console.WriteLine("Counter Value: " + counter.ToString());

}

public static Singleton GetInstance

{

get

{

if (instance == null)

{

lock (obj)

{

if (instance == null)

{

instance = new Singleton();

}

}

}

return instance;

}

}

public void PrintDetails(string message)

{

Console.WriteLine(message);

}

}

}

### 

### **Key Points:**

* **Locks:** Ensure that only one thread can access the critical section where the Singleton instance is created.
* **Double-Checked Locking:** This pattern minimizes the overhead of acquiring a lock by first checking if the instance is null outside the lock.

### **Conclusion:**

By implementing locks and using the double-checked locking pattern, you can ensure that your Singleton class is thread-safe. This approach prevents multiple instances from being created in a multi-threaded environment, maintaining the integrity of the Singleton pattern.

For more details on Double-Checked Locking, refer to [this article on Wikipedia](https://en.wikipedia.org/wiki/Double-checked_locking).

## Non-Lazy or Eager Loading in Singleton

**Eager Loading** refers to the practice of initializing an object before it is accessed. This means the object is instantiated when the application starts and is kept ready for use whenever needed.

### **Key Characteristics of Eager Loading:**

* **Immediate Initialization:** The object is created during the application's startup or at the point of declaration.
* **Lower Memory Footprint:** Suitable for scenarios where the overhead of checking whether the instance is initialized can be avoided, and the memory footprint is minimal.
* **Thread Safety:** In eager loading, the Common Language Runtime (CLR) manages the initialization and ensures thread safety, so there's no need for explicit thread-safety code.

### **Example of Eager Loading:**

In an eager loading scenario, the Singleton instance would be created immediately when the class is loaded:

public sealed class EagerSingleton

{

private static readonly EagerSingleton instance = new EagerSingleton();

// Private constructor to prevent external instantiation

private EagerSingleton()

{

// Initialization code here

}

// Public property to return the already created instance

public static EagerSingleton Instance

{

get

{

return instance;

}

}

}

**Singleton with Lazy Keyword in .NET 4.0**

The Lazy<T> keyword in .NET 4.0 introduces a convenient and thread-safe way to implement Lazy Initialization. This allows the Singleton instance to be created only when it is first accessed, ensuring that resources are allocated only when necessary.

### **Key Characteristics of Lazy<T>:**

* **Thread Safety:** By default, Lazy<T> objects are thread-safe. The first thread that accesses the Value property of the Lazy object initializes it, taking care of any thread safety concerns.
* **No Need for Explicit Locking:** Since Lazy<T> manages thread safety internally, there’s no need to write explicit code for handling thread race conditions.

### **Example Using Lazy<T>:**

Here’s an example showing how to implement a Singleton pattern using the Lazy<T> keyword:

using System;

namespace SingletonDemo

{

public sealed class Singleton

{

private static int counter = 0;

// Private constructor to prevent external instantiation

private Singleton()

{

counter++;

Console.WriteLine("Counter Value " + counter.ToString());

}

// Lazy initialization with thread safety

private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new Singleton());

// Public property to access the singleton instance

public static Singleton GetInstance

{

get

{

return instance.Value;

}

}

// Public method to demonstrate functionality

public void PrintDetails(string message)

{

Console.WriteLine(message);

}

}

}

### **How It Works:**

* **Lazy Initialization:** The instance variable is of type Lazy<Singleton>, which defers the creation of the Singleton instance until GetInstance is called.
* **Thread Safety:** The Lazy<T> keyword ensures that the Singleton instance is created only once, even in a multi-threaded environment, without the need for explicit locking mechanisms.

### **Conclusion:**

Using the Lazy<T> keyword simplifies the implementation of a thread-safe Singleton by handling the lazy initialization and thread safety internally. This approach is especially useful in scenarios where the Singleton instance may be resource-intensive to create and is not always required at startup.

# 03\_Factory Pattern

#### **Gang of Four Definition**

The Gang of Four (GoF) defines the Factory Design Pattern as:

"Define an interface for creating an object, but let sub-classes decide which class to instantiate. The Factory method lets a class defer instantiation it uses to sub-classes."

#### **Overview**

The Factory Pattern is one of the most widely used design patterns in real-world applications. It provides a way to create objects without exposing the creation logic to the client and uses a common interface to refer to newly created objects.

#### **When to Use the Factory Pattern**

* **Object Extension:** When the object needs to be extended to subclasses.
* **Unknown Sub-Classes:** When the classes don’t know what exact sub-classes they need to create.
* **Changing Product Implementation:** When the product implementation tends to change over time, but the client code should remain unchanged.

### **Example: Differentiating Employees by Type**

Let’s consider a business requirement where employees need to be differentiated as permanent and contract employees. Their pay scales and bonuses vary based on their employee types.

### 

## 

## 

## 

## 

## 

## 

## Simple Factory Design

### Without Factory Pattern

In a basic implementation, you might directly check the employee type in the controller and assign pay and bonus accordingly.

public class EmployeeController

{

public void Create(Employee employee)

{

if (employee.EmployeeTypeID == 1)

{

employee.HourlyPay = 8;

employee.Bonus = 10;

}

else if (employee.EmployeeTypeID == 2)

{

employee.HourlyPay = 12;

employee.Bonus = 5;

}

// Save employee to database

}

}

#### **Tight Coupling**

* **Definition:** Tight coupling occurs when a class is heavily dependent on the details of another class. This means changes in one class will likely require changes in the other.
* **Creation:** In the example above, the controller is tightly coupled with the logic of determining pay and bonus. If a new employee type is added, the controller must be modified, leading to extra development and testing efforts.
* **Drawbacks:** This coupling makes the system rigid and difficult to extend. Every time a new type of employee is introduced, the controller code needs to be updated.

### 

### 

### With Factory Pattern

Using the Factory Pattern, you can remove the tight coupling between the controller and the employee type determination logic.

#### **Step 1: Define an Interface**

public interface IEmployeeManager

{

decimal GetBonus();

decimal GetPay();

}

#### **Step 2: Implement the Interface in Subclasses**

public class PermanentEmployeeManager : IEmployeeManager

{

public decimal GetBonus() => 10;

public decimal GetPay() => 8;

}

public class ContractEmployeeManager : IEmployeeManager

{

public decimal GetBonus() => 5;

public decimal GetPay() => 12;

}

#### **Step 3: Create a Factory Class**

public class EmployeeManagerFactory

{

public IEmployeeManager GetEmployeeManager(int employeeTypeID)

{

return employeeTypeID switch

{

1 => new PermanentEmployeeManager(),

2 => new ContractEmployeeManager(),

\_ => throw new NotImplementedException()

};

}

}

#### **Step 4: Use the Factory in the Controller**

public class EmployeeController

{

public void Create(Employee employee)

{

EmployeeManagerFactory empFactory = new EmployeeManagerFactory();

IEmployeeManager empManager = empFactory.GetEmployeeManager(employee.EmployeeTypeID);

employee.Bonus = empManager.GetBonus();

employee.HourlyPay = empManager.GetPay();

// Save employee to database

}

}

### **Benefits of Using Factory Pattern**

* **Reduced Coupling:** The controller no longer needs to know the details of pay and bonus calculation. This logic is encapsulated in specific employee manager classes.
* **Easy to Extend:** If a new type of employee is introduced, you just need to add a new class implementing IEmployeeManager and update the factory method without changing the controller.
* **Improved Maintainability:** The code becomes easier to maintain as changes in business logic don't require changes in multiple places.

In summary, the Factory Pattern helps in achieving loose coupling, making your codebase more flexible and easier to extend or modify.

## 

## Factory Method Design Pattern

### **Factory Method Design Pattern Example**

The Factory Method Pattern is a more advanced version of the Simple Factory Pattern, providing another level of abstraction. It allows subclasses to alter the type of objects that will be created.

### **Business Requirement**

We need to differentiate between permanent and contract employees. Both types have specific pay scales and bonuses, and now we also need to calculate additional allowances:

* **Permanent Employee:** Calculate House Rent Allowance.
* **Contract Employee:** Calculate Medical Allowance.

### **Step-by-Step Implementation Using Factory Method Pattern**

Let's implement this without involving controllers or databases, focusing purely on the logic with simple methods and classes.

### **Step 1: Define the Employee Model**

First, we create a simple Employee class to represent the employee data.

public class Employee

{

public int EmployeeTypeID { get; set; }

public decimal HourlyPay { get; set; }

public decimal Bonus { get; set; }

public decimal HouseAllowance { get; set; }

public decimal MedicalAllowance { get; set; }

}

### 

### 

### 

### 

### **Step 2: Create the Employee Manager Interface**

Define an interface IEmployeeManager that will provide methods to calculate pay and allowances.

public interface IEmployeeManager

{

decimal GetBonus();

decimal GetPay();

decimal GetHouseAllowance(); // Only for permanent employees

decimal GetMedicalAllowance(); // Only for contract employees

}

### 

### **Step 3: Implement Concrete Employee Managers**

Now, let's implement the interface for both types of employees:

public class PermanentEmployeeManager : IEmployeeManager

{

public decimal GetBonus() => 10;

public decimal GetPay() => 8;

public decimal GetHouseAllowance() => 1500; // Specific to permanent employees

public decimal GetMedicalAllowance() => 0; // Not applicable for permanent employees

}

public class ContractEmployeeManager : IEmployeeManager

{

public decimal GetBonus() => 5;

public decimal GetPay() => 12;

public decimal GetHouseAllowance() => 0; // Not applicable for contract employees

public decimal GetMedicalAllowance() => 500; // Specific to contract employees

}

**Step 4: Create an Abstract Base Factory**

The BaseEmployeeFactory is an abstract class that defines the template for creating an employee manager. It also applies the salary calculations.

public abstract class BaseEmployeeFactory

{

protected Employee \_employee;

public BaseEmployeeFactory(Employee employee)

{

\_employee = employee;

}

public Employee ApplySalary()

{

IEmployeeManager manager = this.Create();

\_employee.Bonus = manager.GetBonus();

\_employee.HourlyPay = manager.GetPay();

return \_employee;

}

public abstract IEmployeeManager Create();

}

### 

### 

### 

### 

### 

### 

### 

### **Step 5: Implement Concrete Factories**

Implement concrete factory classes for each employee type:

public class PermanentEmployeeFactory : BaseEmployeeFactory

{

public PermanentEmployeeFactory(Employee emp) : base(emp)

{

}

public override IEmployeeManager Create()

{

PermanentEmployeeManager manager = new PermanentEmployeeManager();

\_employee.HouseAllowance = manager.GetHouseAllowance();

return manager;

}

}

public class ContractEmployeeFactory : BaseEmployeeFactory

{

public ContractEmployeeFactory(Employee emp) : base(emp)

{

}

public override IEmployeeManager Create()

{

ContractEmployeeManager manager = new ContractEmployeeManager();

\_employee.MedicalAllowance = manager.GetMedicalAllowance();

return manager;

}

}

### 

### 

### **Step 6: Implement the Factory Method to Choose the Right Factory**

Create a factory class that returns the appropriate factory based on the employee type.

public class EmployeeManagerFactory

{

public BaseEmployeeFactory CreateFactory(Employee employee)

{

if (employee.EmployeeTypeID == 1)

{

return new PermanentEmployeeFactory(employee);

}

else if (employee.EmployeeTypeID == 2)

{

return new ContractEmployeeFactory(employee);

}

throw new ArgumentException("Invalid Employee Type");

}

}

### **Step 7: Use the Factory Method in the Main Program**

Finally, let's use the factory in a simple Main method to create an employee and apply the salary logic.

class Program

{

static void Main(string[] args)

{

Employee permanentEmployee = new Employee { EmployeeTypeID = 1 };

Employee contractEmployee = new Employee { EmployeeTypeID = 2 };

EmployeeManagerFactory factory = new EmployeeManagerFactory();

// For Permanent Employee

BaseEmployeeFactory permanentFactory = factory.CreateFactory(permanentEmployee);

permanentFactory.ApplySalary();

Console.WriteLine($"Permanent Employee - Hourly Pay: {permanentEmployee.HourlyPay}, Bonus: {permanentEmployee.Bonus}, House Allowance: {permanentEmployee.HouseAllowance}");

// For Contract Employee

BaseEmployeeFactory contractFactory = factory.CreateFactory(contractEmployee);

contractFactory.ApplySalary();

Console.WriteLine($"Contract Employee - Hourly Pay: {contractEmployee.HourlyPay}, Bonus: {contractEmployee.Bonus}, Medical Allowance: {contractEmployee.MedicalAllowance}");

}

}

### **Summary**

* **Tight Coupling Issue:** Before using the Factory Method Pattern, the logic to determine employee types and calculate their pay was directly inside the client code, leading to tight coupling. Any change in the employee types would require changes in the client code, making it less maintainable.
* **Factory Method Pattern Solution:** By using the Factory Method Pattern, we've abstracted the creation of employee managers, allowing the client code to remain unchanged even if new employee types are introduced. This makes the code more modular, maintainable, and easier to extend.

The Factory Method Pattern provides a way to encapsulate object creation, allowing for more flexibility and scalability in your application.