JAVA INTERVIEW PREPARATION - 2025

SOLID PRINCIPLES JAVA INTERVIEWS

JAVA INTERVIEW PREPARATION - 2025

Single Responsibility Principle

A class should have only one responsibility.

```java
class Report {
    public void GenerateReport()
    {
        //Code
    }
    public void SendEmail()
    {
        //Code
    }
}
```

```java
class ReportGenerator {
    public void Generate()
    {
        //Code
    }
}
class EmailSender {
    public void Send()
    {
        //Code
    }
}
```

Case 1: Wrong - The `Report` class is doing two jobs: generating reports and sending emails, which makes it harder to maintain.

Case 2: Right - Separating into `ReportGenerator` and `EmailSender` ensures each class handles only one responsibility.

Use: This makes the code cleaner, reusable, and easier to debug or modify.

JAVA INTERVIEW PREPARATION - 2025

Open-Closed Principle

A class should be open for extension but closed for modification.

```
class PaymentProcessor {
    public void ProcessPayment(string paymentType)
    {
        if (paymentType == "CreditCard")
        {
            // Credit card logic
        }
        else if (paymentType == "PayPal")
        {
            // PayPal logic
        }
    }
}
```

```
interface IPaymentMethod
{
    void ProcessPayment(decimal amount);
}

class CreditCardPayment : IPaymentMethod
{
    public void ProcessPayment(decimal amount)
    {
        // Credit card logic
    }
}

class PayPalPayment : IPaymentMethod
{
    public void ProcessPayment(decimal amount)
    {
        // PayPal logic
    }
}
```

Case 1: Wrong - The `PaymentProcessor` class checks payment types with `if-else`, making it hard to add new types without modifying the existing code.

Case 2: Right - By using an `IPaymentMethod` interface, new payment types can be added by creating new classes without changing the existing code.

Use: Makes the code extensible, avoids breaking existing functionality, and simplifies maintenance.

JAVA INTERVIEW PREPARATION - 2025

Liskov Substitution Principle

Derived classes must be substitutable for their base classes.

```
class Rectangle {
    public virtual double GetArea()
    {
        return Width * Height;
    }
    public double Width { get; set; }
    public double Height { get; set; }
}

class Square : Rectangle {
    public override double GetArea()
    { return Width * Width; } // Violates LSP
}
```

```
interface IShape {
    double GetArea();
}

class Rectangle : IShape {
    public double Width { get; set; }
    public double Height { get; set; }
    public double GetArea() => Width * Height;
}

class Square : IShape {
    public double Side { get; set; }
    public double GetArea() => Side * Side;
}
```

Case 1: Wrong - The `Square` class inherits from `Rectangle`, but their behaviors conflict because a square doesn't have separate width and height, violating the substitution principle.

Case 2: Right - Using a common `IShape` interface separates their behaviors, ensuring each class (e.g., `Rectangle`, `Square`) works independently and correctly.

Use: Ensures derived classes can replace base classes without breaking functionality or logic.

JAVA INTERVIEW PREPARATION - 2025

Interface Segregation Principle

Clients should not be forced to depend on interfaces they don't use.

```
interface IUser
{
    void Register(string username, string password);
    void Login(string username, string password);
    void SendResetPasswordEmail(string email);
}
```

```
interface IRegistration
{
    void Register(string username, string password);
}

interface IAuthentication
{
    void Login(string username, string password);
}

interface IPasswordReset
{
    void SendResetPasswordEmail(string email);
}
```

Case 1: Wrong - The `IUser` interface forces classes to implement methods like `SendResetPasswordEmail` even if they don't need them, leading to unnecessary dependencies.

Case 2: Right - Splitting into smaller interfaces (`IRegistration`, `IAuthentication`, `IPasswordReset`) ensures classes only implement what they actually use.

Use: Improves flexibility, reduces redundant code, and ensures interfaces remain focused and modular.

JAVA INTERVIEW PREPARATION - 2025

Dependency Inversion Principle

High-level modules should depend on abstractions, not on concretions.

```
class ProductService
{
    private readonly ProductRepository repository = new
    ProductRepository();

    public List<Product> GetProducts()
    {
        return repository.GetAllProducts();
    }
}
```

```
interface IProductRepository
{
    List<Product> GetAllProducts();
}

class ProductRepository : IProductRepository
{
    public List<Product> GetAllProducts()
    {
        // Code
    }
}

class ProductService
{
    private readonly IProductRepository repository;
    public ProductService(IProductRepository repository)
    {
        this.repository = repository;
    }

    public List<Product> GetProducts()
    {
        return repository.GetAllProducts();
    }
}
```

Case 1: Wrong - `ProductService` directly depends on `ProductRepository`, making the code tightly coupled and hard to test or extend.

Case 2: Right - `ProductService` depends on an abstraction (`IProductRepository`), allowing flexibility to swap implementations without changing the high-level module.

Use: Makes the code scalable, testable, and easier to maintain by reducing dependency on concrete classes.