Java Design pattern interview questions

codespaghetti.com/java-design-pattern-interview-questions/

Design Patterns

Java Design Pattern Interview Questions, Programs and Examples.

Table of Contents:

Top Five Java Design Pattern Interview Questions

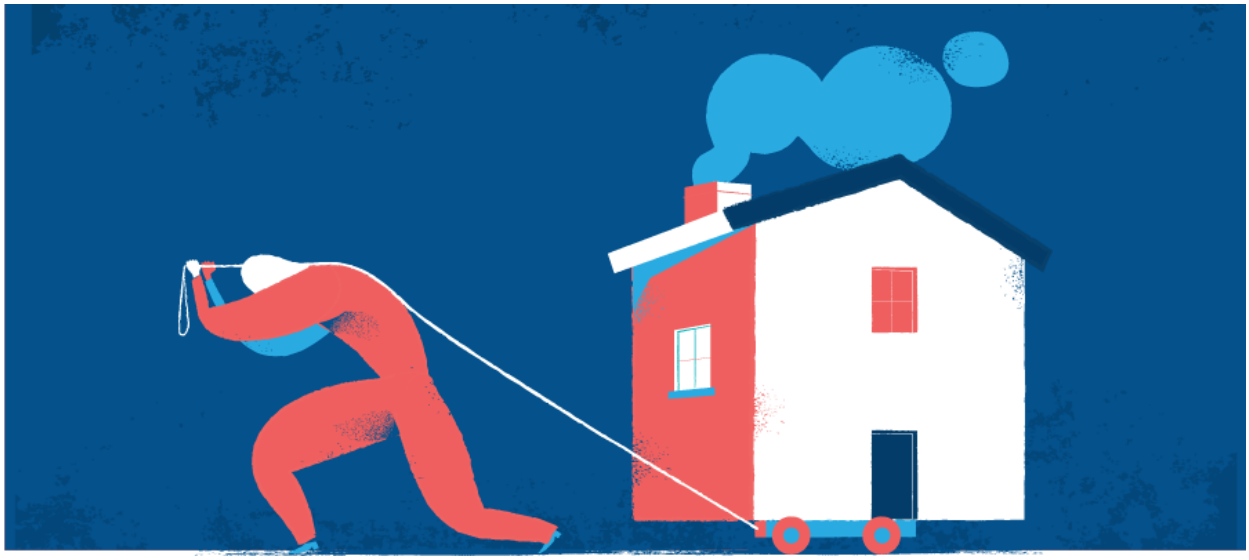
What Are Advantages of Design Patterns?

- A design patterns are **well-proved solution** for solving the specific problem.
- The benefit of design patterns lies in reusability and extensibility of the already developed applications.
- Design patterns use *object-oriented concepts* like decomposition, inheritance and polymorphism.
- They provide the solutions that help to define the system architecture.

What Are Disadvantages of Design Patterns?

- Design Patterns do not lead to direct code reuse.
- Design Patterns are deceptively simple.
- Teams may suffer from patterns overload.
- Integrating design patterns into a software development process is a human-intensive activity.

Singleton Design Pattern Interview Questions



What does singleton Design Pattern do

It ensures that a class has only one instance, and provides a global point of access to it.

When to use Singleton?

Use the Singleton when

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Typical Use Cases

- The logging class
- Managing a connection to a database
- File manager

Real world examples of singleton

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()

Disadvantages of singleton

- Violates Single Responsibility Principle (SRP) by controlling their own creation and lifecycle.
- Encourages using a global shared instance which prevents an object and resources used by this object from being deallocated.
- Creates tightly coupled code that is difficult to test.Makes it almost impossible to subclass a Singleton.

Question: What is a Singleton design pattern?

Singleton pattern is one of the simplest design pattern.It is used when an application needs one, and only one, instance of an object of class.

- Singleton ensures that a class has only one instance, and provide a global point of access to this instance.
- Singleton encapsulates "just-in-time initialization" of the object

Conditions:

Singleton should only be considered if following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

Implementation in Java:

Step 1:

Creating a Singleton Class.

```
Public class Singleton {

   //create an object of Singleton
   private static Singleton singletonInstance = new Singleton();

   //make the constructor private so that this class cannot be instantiated

   private Singleton(){}

   //Get the only object available
   public static Singleton getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello");
   }
}
```

Step 2

Get the only object from the singleton class

```
public class SingletonDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor Singleton() is not visible
      //Singleton object = new Singleton();

      //Get the only object available
      Singleton object = Singleton.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Check list

1. Define a private `static` attribute in the "single instance" class.
2. Define a public `static` accessor function in the class.
3. Do "lazy initialization" (creation on first use) in the accessor function.
4. Define all constructors to be `protected` or `private`.
5. Clients may only use the accessor function to manipulate the Singleton.

Question: In how many ways can you create singleton pattern in Java?

Singleton objects can be created in two ways.

- Lazy loading
- Eager loading

Question: If the singletons getInstance is not synchronized , how it would behave in multithreaded environment?

Answer: Threads can be unpredictable , we can't guarantee when the method is unsynchronized it would give max of two singleton instance.

Question: Can you write thread-safe Singleton in Java?

There are multiple ways to write thread-safe singleton in Java

- By writing singleton using double checked locking.
- By using static Singleton instance initialized during class loading.
- By the way using Java enum to create thread-safe singleton this is most simple way.

Question: If the singletons getInstance is not synchronized , how it would behave in multithreaded environment?

Answer: Threads can be unpredictable , we can't guarantee when the method is unsynchronized it would give max of two singleton instance.

Question: Can you write thread-safe Singleton in Java?

There are multiple ways to write thread-safe singleton in Java

- By writing singleton using double checked locking.
- By using static Singleton instance initialized during class loading.
- By the way using Java enum to create thread-safe singleton this is most simple way.

Question: Can we create a clone of a singleton object?

Answer: Yes, it is possible

Question: How to prevent cloning of a singleton object?

Answer: By throwing an exception within the body of the clone() method.

Question: What is eager initialization in singleton?

In eager initialization, the instance of Singleton Class is created at the time of class loading.

This is the easiest method but it has a drawback that instance is created even though it might not be using it by any client.

Here is the implementation in Java

```
package codespaghetti.com;

public class EagerInitialized{

    private static final EagerInitialized instance = new EagerInitialized();

    //private constructor to avoid client applications to use constructor
    private EagerInitialized(){}

    public static EagerInitialized getInstance(){
        return instance;
    }
}
```

If singleton class is not using a lot of resources, this is the approach to use.

But in most of the scenarios, Singleton classes are created for resources such as Database connections etc and we should avoid the instantiation until client calls the getInstance method.

Question: What is static block initialization?

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

```
package codespaghetti.com;

public class StaticBlock {

    private static StaticBlock instance;

    private StaticBlock(){}

    //static block initialization for exception handling
    static{
        try{
            instance = new StaticBlock();
        }catch(Exception e){
            throw new RuntimeException("Exception occurred in creating singleton
instance");
        }
    }

    public static StaticBlock getInstance(){
        return instance;
    }
}
```

Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use.

Question: What is Lazy Initialization in singleton?

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

```
package codespaghetti.com;

public class LazyInitialized {

    private static LazyInitialized instance;

    private LazyInitialized(){}

    public static LazyInitialized getInstance(){
        if(instance == null){
            instance = new LazyInitialized();
        }
        return instance;
    }
}
```

The above implementation works fine incase of single threaded environment but when it comes to multithreaded systems. It can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class.

Question: What is Thread Safe Singleton in Java?

The easier way to create a thread-safe singleton class is to make the global access method synchronized.

So that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```
package codespaghetti.com;

public class ThreadSafeSingleton {

    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance(){
        if(instance == null){
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }

}
```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method.

Although we need it only for the first few threads who might create the separate instances (Read: Java Synchronization).

To avoid this extra overhead every time, double checked locking principle is used. In this approach.

The synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Code below provides the double checked locking implementation.

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
        synchronized (ThreadSafeSingleton.class) {
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

Question: What is Enum Singleton in Java?

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program.

Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

```
package codespaghetti.com;

public enum EnumSingleton {

    INSTANCE;

    public static void doSomething(){
        //do something
    }
}
```

Question: How to implement singleton class with Serialization in Java?

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class.

So that we can store it’s state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

```
package codespaghetti.com;

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

    private static final long serialVersionUID = -1;

    private SerializedSingleton(){}

    private static class SingletonHelper{
        private static final SerializedSingleton instance = new
SerializedSingleton();
    }

    public static SerializedSingleton getInstance(){
        return SingletonHelper.instance;
    }

}
```

The problem with this serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let’s see it with a simple program.

```
package codespaghetti.com;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

    public static void main(String[] args) throws FileNotFoundException,
IOException, ClassNotFoundException {
        SerializedSingleton instanceOne = SerializedSingleton.getInstance();
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
                "filename.ser"));
        out.writeObject(instanceOne);
        out.close();

        //deserailize from file to object
        ObjectInput in = new ObjectInputStream(new FileInputStream(
                "filename.ser"));
        SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();
        in.close();

        System.out.println("instanceOne hashCode="+instanceOne.hashCode());
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());

    }

}
```

Output of the above program is;

```
instanceOne hashCode=2011117821
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

```
protected Object readResolve() {
    return getInstance();
}
```

After this you will notice that hashCode of both the instances are same in test program.

BONUS: Singleton Java Implementation Code With Junits

singleton_Java_Programme

Factory Design Pattern Interview Questions

What does Factory Pattern Do ?

It defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

When to use factory method pattern ?

Use the Factory pattern when

- a class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Typical real world use cases

- java.util.Calendar
- java.util.ResourceBundle
- java.text.NumberFormat
- java.nio.charset.Charset
- java.net.URLStreamHandlerFactory
- java.util.EnumSet
- javax.xml.bind.JAXBContext

Rules of thumb:

- Factory Methods are usually called within Template Methods.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
- The `new` operator considered harmful. There is a difference between requesting an object and creating one. The `new` operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and

allows an object to be requested without inextricable coupling to the act of creation.

Question: When will you prefer to use a Factory Pattern?

The factory pattern is preferred in the following cases:

- a class does not know which class of objects it must create

- factory pattern can be used where we need to create an object of any one of sub-classes depending on the data provided

Question: Difference between Factory and Abstract Factory Design Pattern?

Factory Pattern deals with creation of objects delegated to a separate factory class whereas Abstract Factory patterns works around a super-factory which creates other factories.

Question: Difference between Factory and Builder Design Pattern?

Builder pattern is the extension of Factory pattern wherein the Builder class builds a complex object in multiple steps.

Question: Difference between Factory and Strategy Design Pattern?

Factory is a creational design pattern whereas Strategy is behavioral design pattern. Factory revolves around the creation of object at runtime whereas Strategy or Policy revolves around the decision at runtime.

Question: What is main benefit of using factory pattern?

- Factory design pattern provides approach to code for interface rather than implementation.
- Factory pattern removes the instantiation of actual implementation classes from client code.
- Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance.

Bonus: Factory method java implementation with Junits

factory-method_code

Builder Design Pattern Interview Questions

What does builder pattern do?

Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.

When to use Builder pattern?

Use the Builder pattern when

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
- the construction process must allow different representations for the object that's constructed

Typical use cases

- java.lang.StringBuilder
- java.nio.ByteBuffer as well as similar buffers such as FloatBuffer, IntBuffer and so on.
- java.lang.StringBuffer
- All implementations of java.lang.Appendable
- Apache Camel builders

Rules of thumb

- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable,

subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Question: What are advantage of Builder Design Pattern?

The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects.

Question: What specific problems builder pattern solves?

This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns.

When the Object contains a lot of attributes. Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

Question: What do we need to consider when implementing builder pattern?

- Decide if a common input and many possible representations (or outputs) is the problem at hand.
- Encapsulate the parsing of the common input in a Reader class.
- Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
- Define a Builder derived class for each target representation.
- The client creates a Reader object and a Builder object, and registers the latter with the former.
- The client asks the Reader to "construct".
- The client asks the Builder to return the result.

Bonus: Builder pattern java implementation with Junits

builder_pattern_code

Proxy Design Pattern Interview Questions

What does proxy pattern do?

Proxy pattern provides a surrogate or placeholder for another object to control access to it.

When to use proxy pattern?

Proxy pattern is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable

- Remote proxy provides a local representative for an object in a different address space.
- Virtual proxy creates expensive objects on demand.
- Protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Typical Use Case

- Control access to another object
- Lazy initialization
- Implement logging
- Facilitate network connection
- Count references to an object

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

Question: What are different proxies?

There are multiple use cases where the proxy pattern is useful.

Protection proxy: limits access to the real subject. Based on some condition the proxy filters the calls and only some of them are let through to the real subject.

Virtual proxies: are used when an object is expensive to instantiate. In the implementation the proxy manages the lifetime of the real subject.

It decides when the instance creation is needed and when it can be reused. Virtual proxies are used to optimize performance.

Caching proxies: are used to cache expensive calls to the real subject. There are multiple caching strategies that the proxy can use.

Some examples are read-through, write-through, cache-aside and time-based. The caching proxies are used for enhancing performance.

Remote proxies: are used in distributed object communication. Invoking a local object method on the remote proxy causes execution on the remote object.

Smart proxies: are used to implement reference counting and log calls to the object.

Question: Difference between Proxy and Adapter?

Adapter object has a different input than the real subject whereas Proxy object has the same input as the real subject.

Proxy object is such that it should be placed as it is in place of the real subject.

Bonus: Proxy pattern java implementation with Junits

proxy_Pattern_code

Decorator Design Pattern Interview Questions

What does Decorator pattern do?

What does Decorator pattern do?

Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

When to use Decorator pattern?

Use Decorator pattern when

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects
- For responsibilities that can be withdrawn
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing

Real world examples

- java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer
- java.util.Collections#synchronizedXXX()
- java.util.Collections#unmodifiableXXX()
- java.util.Collections#checkedXXX()

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.

Summary:

Design Patterns are one of the most important topic of object oriented software development.

In todays world, systems are becoming more and more complex, and it is an essential quality of a good software developer to identify. which design pattern should be utilized to solve the problems in the most efficient way.

Design patterns , data structure and algorithms are essential components of a technical interviews. If you are unable to defend yourself in these areas, then you will keep rejected over and over again.

The world of design patterns is vast and complex. You need to make your self familiarize with the most important design patterns to increase your chances of success in interviews.

About The Author

References:

- Design Patterns: Elements of Reusable Object-Oriented Software
- http://java-design-patterns.com/
- http://www.vogella.com/tutorials/DesignPatterns/article.html
- https://www.quora.com/topic/Design-Patterns-Object-Oriented-Software
- https://en.wikipedia.org/wiki/Software_design_pattern
- https://sourcemaking.com/design_patterns/decorator