**Java interview questions:**

1. **How to prevent java singleton class using reflection API**

**public class Singleton {**

**// Volatile variable to ensure thread-safety**

**private static volatile Singleton instance;**

**// Private constructor to prevent instantiation**

**private Singleton() {**

**if (instance != null) {**

**throw new IllegalStateException("Cannot create instance via reflection.");**

**}**

**}**

**// Static method to provide global access to the instance**

**public static Singleton getInstance() {**

**if (instance == null) {**

**synchronized (Singleton.class) {**

**if (instance == null) {**

**instance = new Singleton();**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**2. Use Enum for Singleton:**

**Java enum is inherently singleton, and reflection cannot be used to instantiate an enum, making it the most robust solution.**

**Code Example:**

**java**

**public enum Singleton {**

**INSTANCE;**

**public void someMethod() {**

**// Singleton logic**

**}**

**}**

**Why It Works:**

* Reflection cannot instantiate an enum, as it enforces its singleton property at the JVM level.
* This approach is also thread-safe and serialization-safe by default.

**3. Secure against Reflection via SecurityManager (Advanced Approach):**

You can prevent access to private constructors by configuring a **SecurityManager**. This approach is more suitable for application environments with strict security requirements.

java

Copy code

SecurityManager securityManager = new SecurityManager();

System.setSecurityManager(securityManager);

**Additional Thoughts:**

* While the Enum approach is the most foolproof, the method of throwing an exception in the constructor works well for most practical singleton designs.

**Java coding Questions:**

**arr= {“aa”, ” bb”, ”cc”, “aa”, “aa”, “dd”, “ee”, “bb”};**

**Process all msg parallelly but duplicate msg has to wait completed other same task**

**Java serialization how it work explore this internal working**

Java serialization is a mechanism that allows you to convert an object into a byte stream, making it easier to save the object's state to a file, send it over a network, or store it in a database. This process is essential for persisting the state of an object or for communication between Java Virtual Machines (JVMs). Here’s an exploration of how Java serialization works internally:

**How Java Serialization Works**

1. **Serialization Process**:
   * When an object is serialized, its state (the values of its instance variables) is converted into a byte stream. This byte stream can then be saved to a file or sent over a network.
   * The Java ObjectOutputStream class is typically used for serialization. You call its writeObject() method, passing the object you want to serialize.
2. **Deserialization Process**:
   * Deserialization is the reverse process, where a byte stream is converted back into a copy of the original object. This is done using the ObjectInputStream class and its readObject() method.
   * During deserialization, the JVM reconstructs the object and restores its state.
3. **Serializable Interface**:
   * To enable serialization for a class, it must implement the java.io.Serializable interface. This interface does not contain any methods; its presence signals to the Java serialization mechanism that the class can be serialized.
   * Classes that do not implement this interface will throw a java.io.NotSerializableException at runtime if serialization is attempted.
4. **SerialVersionUID**:
   * It’s a good practice to declare a serialVersionUID field in your class. This unique identifier is used during deserialization to verify that the sender and receiver of a serialized object maintain compatibility regarding serialization.
   * If no serialVersionUID is declared, Java will generate one at runtime, which can lead to InvalidClassException if the class definition changes between serialization and deserialization.
5. **Handling Non-Serializable Fields**:
   * If a class contains fields that are not serializable (e.g., database connections, file streams), you can mark these fields as transient. The transient keyword tells the serialization mechanism to skip these fields during serialization.
   * Example:

java

Copy code

public class MyClass implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private transient Connection dbConnection; // Non-serializable field

}

1. **Custom Serialization**:
   * You can customize the serialization process by implementing the methods writeObject and readObject. This allows you to control how the object is serialized and deserialized.
   * Example:

java

Copy code

private void writeObject(ObjectOutputStream out) throws IOException {

out.defaultWriteObject(); // Serializes default fields

// Custom serialization logic

}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {

in.defaultReadObject(); // Deserializes default fields

// Custom deserialization logic

}

**Internals of Serialization**

* **Stream of Bytes**: The object’s state is written as a sequence of bytes. This byte representation includes information about the object’s class and the values of its fields.
* **Reflection**: Java uses reflection to examine the fields of the object, allowing it to read and write their values during serialization and deserialization.
* **Hierarchy Handling**: If the class is a subclass, the serialization mechanism handles parent classes as well, ensuring the entire object hierarchy is serialized.
* **Performance Considerations**: Serialization can be expensive in terms of performance and memory. Using transient fields wisely and optimizing the serialization logic can help mitigate these costs.

**Example Code**

Here’s a simple example demonstrating serialization and deserialization:

java

Copy code

import java.io.\*;

// Serializable class

class User implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private transient String password; // Not to be serialized

public User(String name, String password) {

this.name = name;

this.password = password;

}

@Override

public String toString() {

return "User{name='" + name + "', password='" + password + "'}";

}

}

public class SerializationExample {

public static void main(String[] args) {

User user = new User("Alice", "secret");

// Serialization

try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("user.ser"))) {

out.writeObject(user);

} catch (IOException e) {

e.printStackTrace();

}

// Deserialization

User deserializedUser = null;

try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("user.ser"))) {

deserializedUser = (User) in.readObject();

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

System.out.println("Deserialized User: " + deserializedUser);

}

}

**Conclusion**

Java serialization is a powerful feature that enables the conversion of objects to byte streams for storage or transmission. Understanding its internals, including the requirements for serializability and the use of transient fields, is crucial for effective implementation. For more detailed information on serialization, you can check the [Java documentation](https://docs.oracle.com/javase/8/docs/platform/serialization/spec/).

Using serialVersionUID as 1L in Java is a common practice, especially in early development phases or simple applications. Here are some reasons why you might not have encountered issues when using 1L for serialVersionUID:

**Reasons for Not Encountering Issues**

1. **Minimal Changes to Class Structure**:
   * If you have not made significant changes to the structure of your class (e.g., adding or removing fields), the automatically generated serialVersionUID will likely remain the same. As a result, serialization and deserialization will continue to work without problems.
2. **Same Application Context**:
   * If you are consistently using the same version of your class in the same application context (e.g., development, testing, or a production environment), you might not run into issues related to class compatibility.
3. **Deserialization of Compatible Classes**:
   * If the objects being deserialized were created using the same version of the class (with the same serialVersionUID), you won’t experience any problems. The serialized object will successfully match the class definition.
4. **Controlled Development Environment**:
   * In smaller projects or controlled environments, you might not have multiple developers modifying the same class concurrently, reducing the chances of introducing incompatible changes.

**Java, Design Pattern Important Interview Questions**

| map() | flatMap() |
| --- | --- |
| The function passed to map() operation returns a single value for a single input. | The function you pass to flatmap() operation returns an arbitrary number of values as the output. |
| One-to-one mapping occurs in map(). | One too many mapping occurs in flatMap(). |
| Only perform the mapping. | Perform mapping as well as flattening. |
| Produce a stream of value. | Produce a stream of stream value. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

**Q) difference between executorservice.submit and execute**

The submit() can accept both Runnable and Callable task but execute() can only accept the Runnable task.

The submit() method is declared in ExecutorService interface while execute() method is declared in the Executor interface.

The return type of submit() method is a Future object but return type of execute() method is void.

public <T> Future<T> submit(Callable<T> task) {

if (task == null) throw new NullPointerException();

RunnableFuture<T> ftask = newTaskFor(task);

execute(ftask);

return ftask;

}

public <T> Future<T> submit(Runnable task, T result) {

if (task == null) throw new NullPointerException();

RunnableFuture<T> ftask = newTaskFor(task, result);

execute(ftask);

return ftask;

}

# How to print even and odd numbers using threads in java

package org.arpit.code2master;

public class PrintEvenOdd implements Runnable{

public int MAX=20;

static int num=1;

int remainder;

static Object lock=new Object();

PrintEvenOdd(int remainder)

{

this.remainder=remainder;

}

@Override

public void run() {

while (num < MAX) {

synchronized (lock) {

while (num % 2 != remainder) { // wait for numbers other than remainder

try {

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(Thread.currentThread().getName() + " --> " + num);

num++;

lock.notifyAll();

}

}

}

}

Create main program as below to execute.

package org.arpit.code2master;

public class PrintEvenOddThreadMain {

public static void main(String[] args) {

PrintEvenOdd runnable1=new PrintEvenOdd(1);

PrintEvenOdd runnable2=new PrintEvenOdd(0);

Thread t1=new Thread(runnable1,"Thread 1");

Thread t2=new Thread(runnable2,"Thread 2");

t1.start();

t2.start();

}

}

https://www.geeksforgeeks.org/print-even-and-odd-numbers-in-increasing-order-using-two-threads-in-java/

**Write code to solve the produce consumer problem in Java?**

The **Producer-Consumer** problem is a classic example of a multi-thread synchronization problem. It involves two types of threads, a **Producer** thread that produces data and a **Consumer** thread that consumes it. The key challenge is to make sure that:

1. The **Producer** thread should wait if the buffer is full (no overwriting data).
2. The **Consumer** thread should wait if the buffer is empty (no consuming data that doesn't exist).
3. Correct synchronization must be maintained so that no data is lost or consumed more than once.

Here's an implementation of the **Producer-Consumer** problem in Java using a shared buffer and wait() and notifyAll() methods for synchronization.

1. **Producer-Consumer Problem in Java:**

java

Copy code

import java.util.LinkedList;

import java.util.Queue;

// The shared buffer between Producer and Consumer

class Buffer {

private Queue<Integer> buffer = new LinkedList<>();

private int capacity;

public Buffer(int capacity) {

this.capacity = capacity;

}

// Producer adds items to the buffer

public synchronized void produce(int value) throws InterruptedException {

while (buffer.size() == capacity) {

System.out.println("Buffer is full. Producer is waiting...");

wait(); // Wait if the buffer is full

}

buffer.offer(value);

System.out.println("Produced: " + value);

notifyAll(); // Notify Consumers that they can consume

}

// Consumer removes items from the buffer

public synchronized int consume() throws InterruptedException {

while (buffer.isEmpty()) {

System.out.println("Buffer is empty. Consumer is waiting...");

wait(); // Wait if the buffer is empty

}

int value = buffer.poll();

System.out.println("Consumed: " + value);

notifyAll(); // Notify Producers that they can produce

return value;

}

}

// Producer thread

class Producer implements Runnable {

private Buffer buffer;

public Producer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

int value = 0;

try {

while (true) {

buffer.produce(value++);

Thread.sleep(100); // Simulate time taken to produce

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

// Consumer thread

class Consumer implements Runnable {

private Buffer buffer;

public Consumer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

try {

while (true) {

buffer.consume();

Thread.sleep(200); // Simulate time taken to consume

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

// Main class to run the Producer-Consumer problem

public class ProducerConsumerExample {

public static void main(String[] args) {

Buffer buffer = new Buffer(5); // Create a buffer with capacity of 5

// Create Producer and Consumer threads

Thread producerThread = new Thread(new Producer(buffer));

Thread consumerThread = new Thread(new Consumer(buffer));

// Start the threads

producerThread.start();

consumerThread.start();

}

}

**Write code to implement blocking queue in Java?**

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

// Producer class

class Producer implements Runnable {

private BlockingQueue<Integer> queue;

public Producer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

int value = 0;

try {

while (true) {

System.out.println("Producing: " + value);

queue.put(value++); // Automatically waits if the queue is full

Thread.sleep(100); // Simulate time taken to produce

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

// Consumer class

class Consumer implements Runnable {

private BlockingQueue<Integer> queue;

public Consumer(BlockingQueue<Integer> queue) {

this.queue = queue;

}

@Override

public void run() {

try {

while (true) {

int value = queue.take(); // Automatically waits if the queue is empty

System.out.println("Consuming: " + value);

Thread.sleep(200); // Simulate time taken to consume

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

// Main class to demonstrate Producer-Consumer problem using BlockingQueue

public class BlockingQueueExample {

public static void main(String[] args) {

// Create a BlockingQueue with a capacity of 5

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

// Create Producer and Consumer threads

Thread producerThread = new Thread(new Producer(queue));

Thread consumerThread = new Thread(new Consumer(queue));

// Start the threads

producerThread.start();

consumerThread.start();

}

}

**Externalizable vs Serializable:**

**Parameter**

**Serializable**

**Externalizable**

**Marker interface:**

**It is marker interface. You don’t have to provide implementation of any method.**

**Externalizable is not marker interface, you have to override writeExternal and readExternal method.**

**Control:**

**Serializable interface has less control over serialization process and it is optional to override readObject and writeObject.**

**Externalizable interface has more control over serialization process and it is mandatory to override writeExternal and readExternal.**

**Performance:**

**JVM uses reflection to perform serialization in the case of Serializable interface which is quite slow.**

**Programmer have to implement readExternal and writeExternal methods but it relatively results in better performance**

**Supersedes:**

**NA**

**If you implement Externalizable interface and provide implementation of readExternal and writeExternal then it supersedes readObject and writeObject methods in that class. It is due to the fact that Externalizable extends Serializable interface.**

**Constructor called during Deserialization:**

**Default constructor is not called during Deserialization process.**

**Default constructor is called during Deserialization process.**

<https://java2blog.com/how-hashmap-works-in-java/>

<https://java2blog.com/java-string-interview-questions-and-answers/>

<https://java2blog.com/java-multithreading-interview-questions-and-answers/>

<https://java2blog.com/java-exception-handling-interview-questions/>

<https://java2blog.com/java-serialization-interview-questions-and-answers/>

<https://java2blog.com/restful-web-services-interview-questions/>

<https://java2blog.com/java-coding-interview-questions/>

<https://java2blog.com/spring-interview-questions-and-answers/>

<https://java2blog.com/hibernate-interview-questions-and-answers/>

## **Why String is declared final or immutable in java? =>https://www.journaldev.com/802/string-immutable-final-java**

In Java, the String class is declared as final and is immutable for several important reasons. Let's break them down:

* **1. Security:**
* Immutability ensures that once a String object is created, it cannot be modified. This property is crucial for security purposes, especially when dealing with sensitive data like usernames, passwords, and network connections.
* If String were mutable, malicious code could potentially alter the contents of a String after its creation. For example, if a password String could be changed, it could lead to unauthorized access.

**Example:**

String url = "https://secure.com";

**// If mutable, malicious code could alter the URL**

* **2. String Pooling (Memory Efficiency):**
* String immutability allows Java to optimize memory usage by using the String Pool. The String Pool is a special area in the heap where literal strings are stored. When you create a String using a literal, Java checks the pool to see if an equivalent String already exists. If so, it returns a reference to the existing object rather than creating a new one.
* This reduces the need for creating multiple copies of the same string, saving memory.

**Example:**

String str1 = "Hello";

String str2 = "Hello"; // Reuses the same object from the String Pool

If String were mutable, changing the contents of str1 would also affect str2, which could lead to unpredictable behavior.

* **3. Thread Safety:**
* Immutability makes String inherently thread-safe. Since its value cannot be changed after creation, multiple threads can access the same String object without needing synchronization. This helps avoid concurrency issues and makes String a good candidate for sharing between threads.
* If String were mutable, you would have to introduce synchronization mechanisms to prevent race conditions, adding complexity and reducing performance.

**Example:**

String message = "Hello";

Thread t1 = new Thread(() -> System.out.println(message));

Thread t2 = new Thread(() -> System.out.println(message));

* **4. Performance (Caching):**
* The String class benefits from internal optimizations such as hash code caching. Since the value of a String cannot change, its hash code only needs to be calculated once and then cached. This boosts performance, especially when using String as keys in data structures like hash maps.
* If String were mutable, the hash code would have to be recalculated each time the value changed, leading to inefficiency.

**Example:**

String key = "name";

int hashCode = key.hashCode(); // Calculated once and reused

* **5. Design Simplicity:**
* Making String immutable simplifies its design. Once created, String objects remain consistent throughout their lifetime. This ensures predictable behavior and avoids potential bugs or side effects that could arise from changing the internal state of String objects.
* 6. Keys in Data Structures:
* String is frequently used as a key in hash-based data structures such as HashMap or HashSet. Since String objects are immutable, their hash codes do not change after creation. This ensures the integrity of the hash-based structures, allowing them to work efficiently.
* If String were mutable, changing its value after it was used as a key in a HashMap would corrupt the map, as the key’s hash code would no longer match its original position in the map.

**Example:**

HashMap<String, Integer> map = new HashMap<>();

String key = "John";

map.put(key, 1);

// If key changes after insertion, retrieving the value would fail

* **7. Ease of Use:**
* Strings are ubiquitous in programming. Making String immutable allows developers to pass them around in code without worrying about whether their content might be altered unexpectedly. This leads to safer and cleaner code.
* Conclusion:

String in Java is declared final to prevent subclasses from overriding its behavior and to ensure immutability. The immutability of String provides several benefits, including security, memory efficiency (through String Pooling), thread safety, performance optimizations (like hash code caching), and design simplicity. These features make String a critical component in Java's architecture.

1. **StringBuilder vs StringBuffer in Java**

| **StringBuffer Class** | **StringBuilder Class** |
| --- | --- |
| StringBuffer is present in Java. | StringBuilder was introduced in Java 5. |
| StringBuffer is synchronized. This means that multiple threads cannot call the methods of StringBuffer simultaneously. | StringBuilder is asynchronized. This means that multiple threads can call the methods of StringBuilder simultaneously. |
| Due to synchronization, StringBuffer is called a thread safe class. | Due to its asynchronous nature, StringBuilder is not a thread safe class. |
| Due to synchronization, StringBuffer is lot slower than StringBuilder. | Since there is no preliminary check for multiple threads, StringBuilder is a lot faster than StringBuffer. |

The StringBuffer class is a mutable sequence of characters. Unlike String, StringBuffer does not override the equals() method to compare contents, and you cannot directly compare StringBuffer objects like you can with String.

**how to create custom immutable class in java**

Creating a **custom immutable class** in Java involves carefully designing the class so that its state cannot be modified after it is constructed. This includes ensuring that all fields are final and unchangeable, and that no methods allow modification of the class's internal state.

Here’s how you can create an immutable class in Java:

1. **Key Steps to Create an Immutable Class:**
2. **Declare the class as final** so that it cannot be subclassed.
3. **Make all fields private and final** to prevent modification after initialization.
4. **Initialize all fields through a constructor** and do not provide any "setter" methods.
5. **Do not allow direct access to mutable objects** or collections. If your class contains references to mutable objects, return deep copies or defensive copies from any getter methods.
6. **Ensure the class does not provide any methods that modify its state**.
7. **Example of a Custom Immutable Class**

Let’s create a simple immutable Person class with fields name and age.

java

Copy code

public final class Person {

private final String name;

private final int age;

// Constructor that initializes the fields

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter method for name (String is immutable)

public String getName() {

return name;

}

// Getter method for age (primitive types are immutable)

public int getAge() {

return age;

}

// No setters or any methods that modify the fields

}

1. **Explanation:**
2. **final class**: The class is declared as final so that it cannot be subclassed. This is important because if a subclass can be created, it could modify the state, violating immutability.
3. **private and final fields**: The fields name and age are private and final. This ensures that they can only be set once (in the constructor) and never changed afterward.
4. **No setters**: There are no setter methods (setName(), setAge(), etc.), ensuring that the object's state cannot be modified after it is created.
5. **Immutable fields**: The field name is of type String, which is immutable in Java, and age is a primitive type (int), which is also immutable. This makes the class easy to keep immutable because neither field can be modified directly.
6. **Handling Mutable Fields in Immutable Classes**

If your class has fields that are **mutable** objects (like Date, List, Map, or custom objects), you need to take extra care to ensure that the internal state cannot be modified.

Here’s an example with a mutable Date field:

java

Copy code

import java.util.Date;

public final class ImmutableEvent {

private final String eventName;

private final Date eventDate; // Date is mutable

// Constructor

public ImmutableEvent(String eventName, Date eventDate) {

this.eventName = eventName;

// Defensive copy of the mutable object to prevent external modifications

this.eventDate = new Date(eventDate.getTime());

}

// Getter method that returns a defensive copy of the mutable field

public Date getEventDate() {

return new Date(eventDate.getTime());

}

public String getEventName() {

return eventName;

}

}

1. **Explanation:**
2. **Mutable field (Date)**: The eventDate field is of type Date, which is mutable.
3. **Defensive copying**: In the constructor, a **defensive copy** of the Date object is made. This ensures that the original object passed to the constructor cannot be modified externally. Similarly, the getEventDate() method returns a copy of the Date object, not the original.
   * This prevents external code from modifying the internal state of the object through reference.
4. **Example of a Custom Immutable Class with a Collection Field**

java

Copy code

import java.util.Collections;

import java.util.List;

public final class ImmutableEmployee {

private final String name;

private final List<String> skills; // List is mutable

public ImmutableEmployee(String name, List<String> skills) {

this.name = name;

// Defensive copy of the mutable list

this.skills = List.copyOf(skills); // or Collections.unmodifiableList(new ArrayList<>(skills));

}

public String getName() {

return name;

}

// Return an unmodifiable view of the list

public List<String> getSkills() {

return skills;

}

}

1. **Explanation:**
2. **Mutable field (List)**: The skills field is a List, which is mutable.
3. **Immutable List**: The constructor uses List.copyOf() to create an immutable copy of the list passed into the constructor. This ensures that any modification to the original list does not affect the ImmutableEmployee object.
4. **Return unmodifiable list**: The getSkills() method returns the list without exposing internal mutability. The list is already immutable, so there is no need for defensive copying in this case.
5. **Summary of Best Practices for Creating an Immutable Class:**
6. **Make the class final** to prevent inheritance.
7. **Make all fields private and final**, initialized only through the constructor.
8. **Avoid providing setter methods** or any other methods that modify the object's state.
9. For fields that refer to **mutable objects**, ensure that:
   * You create defensive copies in the constructor.
   * You return defensive copies from any getter methods to prevent external modification.
10. **Immutable classes are inherently thread-safe** because their state cannot change after they are created.

Immutable classes are widely used in Java, especially for representing **value objects** (e.g., String, Integer, BigDecimal), and they can help ensure thread safety, simplify your code, and avoid bugs related to unintended modifications

**Different ways to create objects in Java**

As you all know, in Java, a class provides the blueprint for objects, you create an object from a class. There are many different ways to create objects in Java.  
**Following are some ways in which you can create objects in Java:**

1. **Using new Keyword :** Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).
2. **Using**[**New Instance**](https://www.geeksforgeeks.org/new-operator-vs-newinstance-method-java/)**:** If we know the name of the class & if it has a public default constructor we can create an object –**Class.forName**. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn’t create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

**// Java program to illustrate creation of Object**

**// using new Instance**

**public class NewInstanceExample**

**{**

**String name = "GeeksForGeeks";**

**public static void main(String[] args)**

**{**

**try**

**{**

**Class cls = Class.forName("NewInstanceExample");**

**NewInstanceExample obj =**

**(NewInstanceExample) cls.newInstance();**

**System.out.println(obj.name);**

**}**

**catch (ClassNotFoundException e)**

**{**

**e.printStackTrace();**

**}**

**catch (InstantiationException e)**

**{**

**e.printStackTrace();**

**}**

**catch (IllegalAccessException e)**

**{**

**e.printStackTrace();**

**}**

**}**

**}**

1. **Using**[**clone() method:**](https://www.geeksforgeeks.org/clone-method-in-java-2/) Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
   To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

**// Java program to illustrate creation of Object**

**// using clone() method**

**public class CloneExample implements Cloneable**

**{**

**@Override**

**protected Object clone() throws CloneNotSupportedException**

**{**

**return super.clone();**

**}**

**String name = "GeeksForGeeks";**

**public static void main(String[] args)**

**{**

**CloneExample obj1 = new CloneExample();**

**try**

**{**

**CloneExample obj2 = (CloneExample) obj1.clone();**

**System.out.println(obj2.name);**

**}**

**catch (CloneNotSupportedException e)**

**{**

**e.printStackTrace();**

**}**

**}**

**}**

1. **Using**[deserialization](https://www.geeksforgeeks.org/serialization-in-java/)**:** Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn’t use any constructor to create the object.  
   To deserialize an object we need to implement the Serializable interface in the class.
2. **Using newInstance() method of Constructor class :** This is similar to the newInstance() method of a class. There is one newInstance() method in the **java.lang.reflect.Constructor** class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.

Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.

// Java program to illustrate creation of Object

// using newInstance() method of Constructor class

import java.lang.reflect.\*;

public class ReflectionExample

{

private String name;

ReflectionExample()

{

}

public void setName(String name)

{

this.name = name;

}

public static void main(String[] args)

{

try

{

Constructor<ReflectionExample> constructor

= ReflectionExample.class.getDeclaredConstructor();

ReflectionExample r = constructor.newInstance();

r.setName("GeeksForGeeks");

System.out.println(r.name);

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

**volatile keyword in Java**

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

**class** SharedObj

{

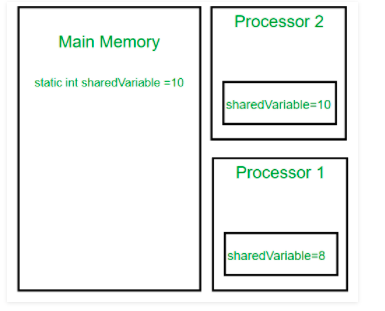
// Changes made to sharedVar in one thread

// may not immediately reflect in other thread

**static int** sharedVar = 6;

}

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](https://en.wikipedia.org/wiki/CPU_cache#Write_policies) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.



class SharedObj

{

// volatile keyword here makes sure that

// the changes made in one thread are

// immediately reflect in other thread

static **volatile** int sharedVar = 6;

}

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

### Atomicity

Atomic operations are those operations that ALWAYS execute together. Either all of them execute together, or none of them executes. If an operation is atomic, then it cannot be partially complete, either it will be complete, or not start at all, but will not be incomplete.

Consider the following example:

1

public class Atomicity {

2

​

3

int i;

4

​

5

public void incrementNumber() {

6

i = 1;

7

}

8

}

In the above example, we are writing value 1 to an int i. This is an atomic operation since this will either happen all together, assigning 1 to i, or it won’t happen at all. This property of an operation, or a set of operations, being atomic is known as atomicity.

Now, consider the following example:

1

public class Atomicity {

2

​

3

int i;

4

​

5

public void incrementNumber() {

6

i = i + 1;

7

}

8

}

At first glance, this might seem like an atomic operation, but it is actually not. This one-liner actually consists of three operations.

* Read operation, where the value of i is read.
* Modify operation, where a new value is being calculated (i + 1).
* Write operation, where the new value is written to the variable i.

Now, since these operations are separate, these set of operations are currently not atomic, since it is possible for them to partially execute. This might be a little hard to understand in a single-threaded environment, but just imagine running this piece of code on multiple threads. If we call this method two times on a single thread, it will result in i being equal to 2. But imagine calling it on two threads. The result should ideally remain the same, but if we call this method on two threads at the same time, then thread A and B will both read the value at the same time and update the value at the same time as well, resulting i to be equal to 1. This is happening because the set of operations here are not atomic. Let’s see how we can make these set of operations atomic.

### Atomic Variables

Atomic variables come to the rescue in these types of situations. Atomic variables allow us to perform atomic operations on the variables.

Consider the following example:

1

import java.util.concurrent.atomic.AtomicInteger;

2

​

3

public class Atomicity {

4

​

5

AtomicInteger i;

6

​

7

public void incrementNumber() {

8

i.incrementAndGet();

9

}

10

}

The above example shows how an AtomicInt can be used to update the value atomically. What incrementAndGet()  does is atomically increment the value by 1, and then returns the updated value. If the program is now run in a multi-threaded environment, supposing with 2 threads, then the end result will always be i being equal to 2. This is because no matter which thread gets to the incrementAndGet() method first, since it is an atomic operation, the thread will update the value of i to 1, and only then another thread will be able to access or update it, which will make the value of i to 2, thus giving us the correct result.

The most commonly used atomic classes are – AtomicInt, AtomicLong, AtomicBoolean, and AtomicReference. All of these provide atomic operations for the respective classes. AtomicReference can be used for just any type of object.

How to create Immutable class in Java?

Immutable class means that once an object is created, we cannot change its content. In Java, all the [wrapper classes](https://www.geeksforgeeks.org/wrapper-classes-java/) (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:

* The class must be declared as final (So that child classes can’t be created)
* Data members in the class must be declared as private (So that direct access is not allowed)
* Data members in the class must be declared as final (So that we can’t change the value of it after object creation)
* A parameterized constructor should initialize all the fields performing a deep copy (So that data members can’t be modified with object reference)
* Deep Copy of objects should be performed in the getter methods (To return a copy rather than returning the actual object reference)
* No setters (To not have the option to change the value of the instance variable)

import java.util.HashMap;

import java.util.Map;

// An immutable class

public final class Student {

private final String name;

private final int regNo;

private final Map<String, String> metadata;

public Student(String name, int regNo,

Map<String, String> metadata)

{

this.name = name;

this.regNo = regNo;

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

this.metadata = tempMap;

}

public String getName() { return name; }

public int getRegNo() { return regNo; }

public Map<String, String> getMetadata()

{

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

this.metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

return tempMap;

}

}

// Driver class

class Test {

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("1", "first");

map.put("2", "second");

Student s = new Student("ABC", 101, map);

System.out.println(s.getName());

System.out.println(s.getRegNo());

System.out.println(s.getMetadata());

// Uncommenting below line causes error

// s.regNo = 102;

map.put("3", "third");

System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in constructor

s.getMetadata().put("4", "fourth");

System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in getter

}

}

## Advantages of Copy Constructors Over Object.clone()

Copy constructors are better than Object.clone() because they:

* Don’t force us to implement any interface or throw an exception, but we can surely do it if it is required.
* Don’t require any type casting.
* Don’t require us to depend on an unknown object creation mechanism.
* Don’t require parent classes to follow any contract or implement anything.
* Allow us to modify final fields.
* Allow us to have complete control over object creation, meaning we can write our initialization logic in it.

### Is there Constructor class in Java?

Yes.

### What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

### Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Coupling in Java

In object oriented design, Coupling refers to the degree of direct knowledge that one element has of another. In other words, how often do changes in class A force related changes in class B.  
**There are two types of coupling:**

1. **Tight coupling :**In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.  
   **Example :**If you want to change the skin, you would also have to change the design of your body as well because the two are joined together – they are tightly coupled. The best example of tight coupling is RMI(Remote Method Invocation).

|  |
| --- |
| // Java program to illustrate  // tight coupling concept  class Subject {      Topic t = new Topic();      public void startReading()      {          t.understand();      }  }  class Topic {      public void understand()      {          System.out.println("Tight coupling concept");      }  } |

**Loose coupling :**In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled. In order to over come from the problems of tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling.  
**Example :**If you change your shirt, then you are not forced to change your body – when you can do that, then you have loose coupling. When you can’t do that, then you have tight coupling. The examples of Loose coupling are Interface, JMS.

|  |
| --- |
| // Java program to illustrate  // loose coupling concept  public interface Topic  {      void understand();  }  class Topic1 implements Topic {  public void understand()      {          System.out.println("Got it");      }  } class Topic2 implements Topic {  public void understand()      {          System.out.println("understand");      }  } public class Subject {  public static void main(String[] args)      {          Topic t = new Topic1();          t.understand();      }  } |

Cohesion in Java

In object oriented design, cohesion refers all about how a single class is designed. Cohesion is the Object Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose.  
The more focused a class is, the cohesiveness of that class is more. The advantages of high cohesion is that such classes are much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

**Example :**Suppose we have a class that multiply two numbers, but the same class creates a pop up window displaying the result. This is the example of low cohesive class because the window and the multiplication operation don’t have much in common.  
To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply’s method to get the result and display it. This way to develop a high cohesive solution.

**Lets understand the structure of high cohesive program :**

|  |
| --- |
| // Java program to illustrate  // high cohesive behavior  class Multiply {      int a = 5;      int b = 5;      public int mul(int a, int b)      {          this.a = a;          this.b = b;          return a \* b;      }  }    class Display {      public static void main(String[] args)      {          Multiply m = new Multiply();          System.out.println(m.mul(5, 5));      }  } |

Output:

25

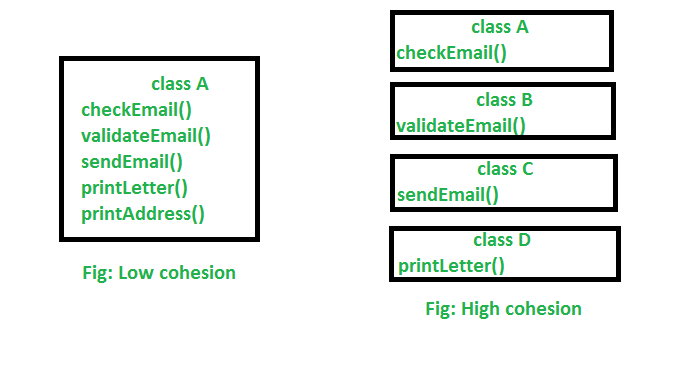
|  |
| --- |
| // Java program to illustrate  // high cohesive behavior  class Name {      String name;      public String getName(String name)      {          this.name = name;          return name;      }  }    class Age {      int age;      public int getAge(int age)      {          this.age = age;          return age;      }  }    class Number {      int mobileno;      public int getNumber(int mobileno)      {          this.mobileno = mobileno;          return mobileno;      }  }    class Display {      public static void main(String[] args)      {          Name n = new Name();          System.out.println(n.getName("Geeksforgeeks"));          Age a = new Age();          System.out.println(a.getAge(10));          Number no = new Number();          System.out.println(no.getNumber(1234567891));      }  } |

Output:

Geeksforgeeks

10

1234567891

**Pictorial view of high cohesion and low cohesion:**  
  
**Explanation :**In the above image, we can see that in low cohesion only one class is responsible to execute lots of job which are not in common which reduces the chance of re-usability and maintenance. But in high cohesion there is a separate class for all the jobs to execute a specific job, which result better usability and maintenance.

**Difference between high cohesion and low cohesion:**

* High cohesion is when you have a class that does a well defined job. Low cohesion is when a class does a lot of jobs that don’t have much in common.
* High cohesion gives us better maintaining facility and Low cohesion results in monolithic classes that are difficult to maintain, understand and reduces re-usability

# Types Of Relationship

Based on reusing the data members from one class to another class in JAVA we have three types of relationships. They are is-a relationship, has-a relationship and uses-a relationship.

1. Is-a relationship is one in which data members of one class is obtained into another class through the concept of inheritance.
2. Has-a relationship is one in which an object of one class is created as a data member in another class.
3. Uses-a relationship is one in which a method of one class is using an object of another class.

Inheritance is the technique which allows us to inherit the data members and methods from base class to derived class.

1. Base class is one which always gives its features to derived classes.
2. Derived class is one which always takes features from base class.

A Derived class is one which contains some of features of its own plus some of the data members from base class

#### Syntax for Inheriting the features from base class to derived class:

class <clsname-2> extends <clsname-1>

{

Variable declaration;

Method definition;

};

Here, clsname-1 and clsname-2 represents derived class and base class respectively.

Extends is a keyword which is used for inheriting the data members and methods from base class to the derived class and it also improves functionality of derived class.

**Note:**

1. Final classes cannot be inherited.
2. If the base class contains private data members then that type of data members will not be inherited into derived class.

Whenever we develop any inheritance application, it is always recommended to create an object of bottom most derived class. Since, bottom most derived class contains all the features from its super classes.

1. One class can extend only one class at a time. Since, JAVA does not support multiple inheritance.

Whenever we inherit the base class members into derived class, when we creates an object of derived class, JVM always creates the memory space for base class members first and later memory space will be created for derived class members.

**For example:**

**class** **c1**;

{

**int** a;

**void** **f1**()

{

.......;

}

};

**class** **c2** **extends** c1

{

**int** b;

**void** **f2**()

{

.......;

}

};

**Note:**

1. Whatever the data members are coming from base class to the derived class, the base class members are logically declared in derived class, the base class methods are logically defined in derived class.
2. Private data members and private methods of the base class will not be inherited at all.

**Write a JAVA program computes sum of two numbers using inheritance?**

**Answer:**

**class** **Bc**

{

**int** a;

};

**class** **Dc** **extends** Bc

{

**int** b;

**void** **set** (**int** x, **int** y)

{

a=x;

b=y;

}

**void** **sum** ()

{

System.out.println ("SUM = "+(a+b));

}

};

**class** **InDemo**

{

**public** **static** **void** **main** (String k [])

{

**int** n1=Integer.parseInt (k [**0**]);

**int** n2=Integer.parseInt (k [**1**]);

Dc do1=**new** Dc ();

do1.set (n1, n2);

do1.sum ();

}

};

For every class in JAVA we have a super class called object class. The purpose of object class is that it provides garbage collector for collecting unreferenced memory locations from the derived classes.

**Design Pattern :**

# What are the SOLID principles in Java?

**SOLID** principles are object-oriented design concepts relevant to software development. SOLID is an acronym for five other class-design principles: **S**ingle Responsibility Principle, **O**pen-Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, and **D**ependency Inversion Principle.

| Principle | Description |
| --- | --- |
| **Single Responsibility Principle** | Each class should be responsible for a single part or functionality of the system. |
| **Open-Closed Principle** | Software components should be open for extension, but not for modification. |
| **Liskov Substitution Principle** | Objects of a superclass should be replaceable with objects of its subclasses without breaking the system. |
| **Interface Segregation Principle** | No client should be forced to depend on methods that it does not use. |
| **Dependency Inversion Principle** | High-level modules should not depend on low-level modules, both should depend on abstractions. |

SOLID is a structured design approach that ensures your software is modular and easy to maintain, understand, debug, and refactor. Following SOLID also helps save time and effort in both development and maintenance. SOLID prevents your code from becoming rigid and fragile, which helps you build long-lasting software.

## Examples

### 1. Single responsibility principle

Every class in Java should have a single job to do. To be precise, there should only be one reason to change a class. Here’s an example of a Java class that does not follow the single responsibility principle (SRP):

public class Vehicle {  
    public void printDetails() {}  
    public double calculateValue() {}  
    public void addVehicleToDB() {}  
}

The Vehicle class has three separate responsibilities: reporting, calculation, and database. By applying SRP, we can separate the above class into three classes with separate responsibilities.

### 2. Open-closed principle

Software entities (e.g., classes, modules, functions) should be open for an extension, but closed for modification.

Consider the below method of the class VehicleCalculations:

public class VehicleCalculations {  
    public double calculateValue(Vehicle v) {  
        if (v instanceof Car) {  
            return v.getValue() \* 0.8;  
        if (v instanceof Bike) {  
            return v.getValue() \* 0.5;  
  
    }  
}

Suppose we now want to add another subclass called Truck. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle.  
A better approach would be for the subclasses Car and Truck to override the calculateValue method:

public class Vehicle {  
    public double calculateValue() {...}  
}  
public class Car extends Vehicle {  
    public double calculateValue() {  
        return this.getValue() \* 0.8;  
}  
public class Truck extends Vehicle{  
    public double calculateValue() {  
        return this.getValue() \* 0.9;  
}

Adding another Vehicle type is as simple as making another subclass and extending from the Vehicle class.

### 3. Liskov substitution principle

The **Liskov Substitution Principle (LSP)** applies to inheritance hierarchies such that derived classes must be completely substitutable for their base classes.

Consider a typical example of a Square derived class and Rectangle base class:

public class Rectangle {  
    private double height;  
    private double width;  
    public void setHeight(double h) { height = h; }  
    public void setWidht(double w) { width = w; }  
    ...  
}  
public class Square extends Rectangle {  
    public void setHeight(double h) {  
        super.setHeight(h);  
        super.setWidth(w);  
    }  
    public void setWidth(double h) {  
        super.setHeight(h);  
        super.setWidth(w);  
    }  
}

The above classes do not obey LSP because you cannot replace the Rectangle base class with its derived class Square. The Square class has extra constraints, i.e., the height and width must be the same. Therefore, substituting Rectangle with Square class may result in unexpected behavior.

### 4. Interface segregation principle

The **Interface Segregation Principle (ISP)** states that clients should not be forced to depend upon interface members they do not use. In other words, do not force any client to implement an interface that is irrelevant to them.

Suppose there’s an interface for vehicle and a Bike class:

public interface Vehicle {  
    public void drive();  
    public void stop();  
    public void refuel();  
    public void openDoors();  
}  
public class Bike implements Vehicle {  
  
    // Can be implemented  
    public void drive() {...}  
    public void stop() {...}  
    public void refuel() {...}  
      
    // Can not be implemented  
    public void openDoors() {...}  
}

As you can see, it does not make sense for a Bike class to implement the openDoors() method as a bike does not have any doors! To fix this, ISP proposes that the interfaces be broken down into multiple, small cohesive interfaces so that no class is forced to implement any interface, and therefore methods, that it does not need.

### 5. Dependency inversion principle

The **Dependency Inversion Principle (DIP)** states that we should depend on abstractions (interfaces and abstract classes) instead of concrete implementations (classes). The abstractions should not depend on details; instead, the details should depend on abstractions.

Consider the example below. We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.

public class Car {  
    private Engine engine;  
    public Car(Engine e) {  
        engine = e;  
    }  
    public void start() {  
        engine.start();  
    }  
}  
public class Engine {  
   public void start() {...}  
}

The code will work, for now, but what if we wanted to add another engine type, let’s say a diesel engine? This will require refactoring the Car class.  
However, we can solve this by introducing a layer of abstraction. Instead of Car depending directly on Engine, let’s add an interface:

public interface EngineInterface {  
    public void start();  
}

Now we can connect any type of Engine that implements the Engine interface to the Car class:

public class Car {  
    private EngineInterface engine;  
    public Car(EngineInterface e) {  
        engine = e;  
    }  
    public void start() {  
        engine.start();  
    }  
}  
public class PetrolEngine implements EngineInterface {  
   public void start() {...}  
}  
public class DieselEngine implements EngineInterface {  
   public void start() {...}  
}

# Performance Analysis of ArrayList and LinkedList in Java

ArrayListuses the Array data structure, and LinkedList uses the DoublyLinkedList data structure. Here, we are going to discuss how the underlying data structure affects the performance of insert, search, and delete operation on ArrayList and LinkedList.

| **Operation** | **LinkedList time complexity** | **ArrayList time complexity** | **Preferred** |
| --- | --- | --- | --- |
| Insert at last index | O(1) | O(1)  (If array copy operation is Considered then O(N)) | LinkedList |
| Insert at given index | O(N) | O(N) | LinkedList |
| Search by value | O(N) | O(N) | ArrayList |
| Get by index | O(N) | O(1) | ArrayList |
| Remove by value | O(N) | O(N) | LinkedList |
| Remove by index | O(N) | O(N) | LinkedList |

## Vector time complexity in Java

Good for

Retrieving elements from a specific position – O (1).

Adding and removing elements from the end. Constant time complexity – O(1).

**Note :** Adding and removing elements from any other position is expensive — Lenear: O(n-i), where n is the number of elements and i is the index of the element added or removed. These operations are more expensive because you have to shift all elements at index i and higher over by one element.

**Uses:**

# Below are the Big O performance of common functions of different Java Collections.

# List | Add | Remove | Get | Contains | Next | Data Structure

# ---------------------|------|--------|------|----------|------|---------------

# ArrayList | O(1) | O(n) | O(1) | O(n) | O(1) | Array

# LinkedList | O(1) | O(1) | O(n) | O(n) | O(1) | Linked List

# CopyOnWriteArrayList | O(n) | O(n) | O(1) | O(n) | O(1) | Array

# Set | Add | Remove | Contains | Next | Size | Data Structure

# ----------------------|----------|----------|----------|----------|------|-------------------------

# HashSet | O(1) | O(1) | O(1) | O(h/n) | O(1) | Hash Table

# LinkedHashSet | O(1) | O(1) | O(1) | O(1) | O(1) | Hash Table + Linked List

# EnumSet | O(1) | O(1) | O(1) | O(1) | O(1) | Bit Vector

# TreeSet | O(log n) | O(log n) | O(log n) | O(log n) | O(1) | Red-black tree

# CopyOnWriteArraySet | O(n) | O(n) | O(n) | O(1) | O(1) | Array

# ConcurrentSkipListSet | O(log n) | O(log n) | O(log n) | O(1) | O(n) | Skip List

# Queue | Offer | Peak | Poll | Remove | Size | Data Structure

# ------------------------|----------|------|----------|--------|------|---------------

# PriorityQueue | O(log n) | O(1) | O(log n) | O(n) | O(1) | Priority Heap

# LinkedList | O(1) | O(1) | O(1) | O(1) | O(1) | Array

# ArrayDequeue | O(1) | O(1) | O(1) | O(n) | O(1) | Linked List

# ConcurrentLinkedQueue | O(1) | O(1) | O(1) | O(n) | O(n) | Linked List

# ArrayBlockingQueue | O(1) | O(1) | O(1) | O(n) | O(1) | Array

# PriorirityBlockingQueue | O(log n) | O(1) | O(log n) | O(n) | O(1) | Priority Heap

# SynchronousQueue | O(1) | O(1) | O(1) | O(n) | O(1) | None!

# DelayQueue | O(log n) | O(1) | O(log n) | O(n) | O(1) | Priority Heap

# LinkedBlockingQueue | O(1) | O(1) | O(1) | O(n) | O(1) | Linked List

# Map | Get | ContainsKey | Next | Data Structure

# ----------------------|----------|-------------|----------|-------------------------

# HashMap | O(1) | O(1) | O(h / n) | Hash Table

# LinkedHashMap | O(1) | O(1) | O(1) | Hash Table + Linked List

# IdentityHashMap | O(1) | O(1) | O(h / n) | Array

# WeakHashMap | O(1) | O(1) | O(h / n) | Hash Table

# EnumMap | O(1) | O(1) | O(1) | Array

# TreeMap | O(log n) | O(log n) | O(log n) | Red-black tree

# ConcurrentHashMap | O(1) | O(1) | O(h / n) | Hash Tables

# ConcurrentSkipListMap | O(log n) | O(log n) | O(1) | Skip List