**Spring Singleton Vs Java Singleton**

The Java singleton is scoped by the Java class loader, the spring singleton is scoped by the container context.

In Java, you can be sure a singleton is a truly a singleton only within the context of the **class loader which loaded it**. Other class loaders should be capable of creating another instance of it (provided the class loaders are not in the same class loader hierarchy), despite of all your efforts in code to try to prevent it.

In Spring, if you could load your singleton class in two different contexts and then again we can break the singleton concept.

So, in summary, Java considers something a singleton if it cannot create more than one instance of that class within a given class loader, whereas Spring would consider something a singleton if it cannot create more than one instance of a class within a given container/context.

**Third normal form** (**3NF**) is a [database schema](https://en.wikipedia.org/wiki/Database_schema) design approach for [relational databases](https://en.wikipedia.org/wiki/Relational_database) which uses [normalizing](https://en.wikipedia.org/wiki/Database_normalization) principles to reduce the duplication of data, **avoid**[**data anomalies**](https://en.wikipedia.org/wiki/Software_bug)**, ensure**[**referential integrity**](https://en.wikipedia.org/wiki/Referential_integrity), and simplify data management

A functional interface can extends another interface only when it does not have any abstract method.

Map<String, Long> map = str.toLowerCase()  
 .replace(" ", "")  
 .codePoints()  
 .mapToObj(Character::*toString*)  
 .collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*()));  
System.*out*.println(map);

**What is a race condition with respect to hashmaps?**

When two or more threads see the need for resizing the same HashMap, they might end up adding the elements of the old bucket to the new bucket simultaneously. As a result, this might lead to infinite loops. In case of collision, i.e, when there are different keys with the same hashcode, internally, we use a single linked list to store elements. We store every new element at the head of the linked list to avoid tail traversing. At the time of resizing, the entire sequence of objects in the linked list gets reversed, during which there are chances of infinite loops.

**How linked list is replaced with binary tree?**

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

**Singleton Design Pattern –** The volatile prevents memory writes from being re-ordered, making it impossible for other threads to read uninitialized fields of your singleton through the singleton's pointer.

Consider this situation: thread A discovers that **uniqueInstance == null**, locks, confirms that it's still null, and calls singleton's constructor. The constructor makes a write into member XYZ inside Singleton, and returns. Thread A now writes the reference to the newly created singleton into uniqueInstance, and gets ready to release its lock.

**How Concurrent HashMap works internally in java?**

It allows concurrent access to the map. Part of the map called *Segment (internal data structure)*is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance. The implementation performs internal sizing to accommodate these many elements.

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on each segment during high concurrency and at-most 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

/\*\* Inner Segment class plays a significant role \*\*/

protected static final class Segment {

protected int count;

protected synchronized int getCount() {

return this.count;

}

protected synchronized void synch() {}

}

/\*\* Segment Array declaration \*\*/

public final Segment[] segments = new Segment[32];

**Redis** is an in-memory datastore hence its primary use-case is in-memory caching. Since it is a Key-value store, it has generally limited query ability, only allowing queries by primary key.

While **CosmosDB** is globally distributed, horizontally scalable, multi-model database service. It becomes handy in scenarios where you need the ability to query over heterogeneous data.

Those two are totally for different purposes; even Microsoft has redis cache as a service apart from CosmosDB only to serve this purpose.

The **ApplicationContext**is the central interface within a spring application that is used for providing configuration information to the application. It implements the **BeanFactory** interface. Hence, the **ApplicationContext** includes all functionality of the **BeanFactory** and much more. Its main function is to support the creation of big business applications. Features:

* Bean instantiation/wiring
* Automatic **BeanPostProcessor** registration
* Automatic **BeanFactoryPostProcessor** registration
* Convenient **MessageSource** access (for i18n)
* **ApplicationEvent** publication

It uses eager loading, so every bean instantiate after the **ApplicationContext** is started up.

**public** **class** FindDupInArray {

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

**int** numRay[] = { 0, 4, 3, 2, 7, 8, 2, 3, 1 };

**for** (**int** i = 0; i < numRay.length; i++) {

                          numRay[numRay[i] % numRay.length] = numRay[numRay[i] % numRay.length] + numRay.length;

                 }

                 System.***out***.println("The repeating elements are : ");

**for** (**int** i = 0; i < numRay.length; i++) {

**if** (numRay[i] >= numRay.length \* 2) {

                                  System.***out***.println(i + " ");

                          }

                 }

         }

}

**Parallel Stream -** very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divided into multiple streams which can be executed parallel on separate cores of the system and the result is shown as the combination of all the individual core’s outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming; they are complex and error prone.

The Java stream library provides a couple of ways to do it. Easily, and in a reliable manner.

* One of the simple ways to obtain a parallel stream is by invoking the [parallelStream()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/) method of **Collection**interface.
* Another way is to invoke the[parallel()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/)method of **BaseStream** interface on a sequential stream.

It is important to ensure that the result of the parallel stream is the same as is obtained through the sequential stream, so the parallel streams must be stateless, non-interfering, and associative.

How to read the file

BufferedReader reader = new BufferedReader(new FileReader(new File(“abc.txt”)));

Java Lock API

1. Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. Synchronized block or method can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
4. Synchronized keyword doesn’t provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
5. We can create different conditions for Lock and different thread can await() for different conditions.

 One functional interface extends another functional interface then it’s no longer the functional interface because one come from parent and other its own, so two abstract methods.

**public** **class** Employee {

**public** **static** **void** m1() {

                 System.***out***.println("ABC");

         }

**public** **void** m2() {

                 System.***out***.println("ABC");

         }

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

                 Employee e;

                 e.*m1*(); //The static method m1() from the type Employee should be accessed in a static way.

                 e.m2(); //The local variable e may not have been initialized.

         }

}

**Technically**

**Abstraction**: Abstraction is the act of representing essential information without including background details and explanations.

**Encapsulation**: Encapsulation is the act of wrapping up of attributes (represented by data members) and operations (represented by functions) under one single unit (represented by class).

**Taking Real world example**

Suppose you go to an *automatic cola vending machine* and request for a *cola*. The machine processes your request and gives the cola.

* Here automatic cola vending machine is a class. It contains both data i.e. Cola can and operations i.e. service mechanism and they are wrapped/integrated under a single unit Cola Vending Machine. This is called Encapsulation.
* You need not know how the machine is working. This is called Abstraction.
* You can interact with cola cans only through service mechanism. You can’t access the details about internal data like how much it cans contains mechanism etc. This is Data Hiding.
* You cannot pick the cans directly. You request for cola through proper instructions and request mechanism (i.e. by paying amount and filling request) and get that cola only through specified channel. This is message passing.

The working and data are hidden from you. This is possible because that Vending machine is made (or Encapsulated or integrated) so. Thus, **we can say Encapsulation is a way to implement Abstraction.**

**public** **class** ReverseString {

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

                 String input = "GeeksForGeeks";

**char**[] try1 = input.toCharArray();

**for** (**int** i = try1.length-1; i >= 0; i--) {

                          System.***out***.println(try1[i]);

                 }

         }

}

**Pair of vowels**

**public** **class** PairOfVowels {

**static** **boolean** isVowel(**char** ch) {

**switch** (ch) {

**case** 'a':

**case** 'e':

**case** 'i':

**case** 'o':

**case** 'u':

**return** **true**;

**default**:

**return** **false**;

                 }

         }

**static** **int** vowelPairs(String s, **int** n) {

**int** cnt = 0;

**for** (**int** i = 0; i < n - 1; i++) {

**if** (*isVowel*(s.charAt(i)) && *isVowel*(s.charAt(i + 1)))

                                  cnt++;

                 }

**return** cnt;

         }

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

                 String s = "abaebio";

**int** n = s.length();

                 System.***out***.print(*vowelPairs*(s, n));

         }

}

**Synchronized Block in Java**

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

o   Synchronized block is used to lock an object for any shared resource.

o   Scope of synchronized block is smaller than the method.

[Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> findFirst() – Returns an [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.

[Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> findAny()

Returns an [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) describing some element of the stream, or an empty Optional if the stream is empty.

The behaviour of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (**If a stable result is desired, use**[**findFirst()**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#findFirst--)**instead**)

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction solves the problem in the design level | Encapsulation solves the problem in the implementation level |
| Abstraction is used for hiding the unwanted data and giving relevant data | Encapsulation means hiding the code and data into a single unit to protect the data from outside world |
| Abstraction lets you focus on what the object does instead of how it does it. | Encapsulation means hiding the internal details or mechanism of how an object does something. |
| Abstraction – Outer layout used in terms of design  Ex: Outer look of a mobile phone like it has a display screen and keypad buttons to dial a no. | Encapsulation – Inner layout used in terms of implementations. Ex: Inner implementation details of a mobile phone and how keypad buttons and display screens relate to each other using circuits |

**When to Use an Interface**

Let's look at some scenarios when one should go with an interface:

* If you are creating functionality that will be useful across a wide range of objects, then you must use an interface. Abstract classes, at the end of the day, should be used for objects that are closely related. But the interfaces are best suited for providing common functionality to unrelated cases.
* Interfaces are a great choice if you think that the API won’t be changing for a while.
* Interfaces are also a great choice. If you want to have something like the multiple inheritances, then you can implement various interfaces.
* If we are going to design the small, concise bits of functionality, then you must use interfaces. But if you are designing the large functional units, then you must use an abstract class.

Employee Manager Hierarchy

* Find name of immediate manager

select Emp.firstName+' '+Emp.lastName, mgr.firstName+ ' '+mgr.lastName

from Employee emp

inner join Employee mgr

on emp.ManagerId = mgr.EmployeeId

Find Max Salary belongs to each department

select e.emp\_name, max(e.emp\_salary)

from employee e

inner join department d

on e.dept\_id = d.dept\_id

group by d.dept\_id;

**Real life examples of queue are:**

* A queue of people at ticket-window: The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
* Vehicles on toll-tax bridge: The vehicle that comes first to the toll tax booth leaves the booth first. The vehicle that comes last leaves last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
* Phone answering system: The person who calls first gets a response first from the phone answering system. The person who calls last gets the response last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.
* Luggage checking machine: Luggage checking machine checks the luggage first that comes first. Therefore, it follows FIFO principle of queue.
* Patients waiting outside the doctor's clinic: The patient who comes first visits the doctor first, and the patient who comes last visits the doctor last. Therefore, it follows the first-in-first-out (FIFO) strategy of queue.

**Real life examples of stack are:**

* To ***reverse a word***. You push a given word to stack - letter by letter - and then pop letters from the stack.
* An ***"undo"*** mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
  + *Undo/Redo*stacks in Excel or Word.
* ***Language processing***:
  + *space for parameters and local variables* is created internally using a stack.
  + compiler's*syntax check for matching braces* is implemented by using stack.
* A ***stack of plates/books*** in a cupboard.
* Wearing/Removing ***Bangles***.
* Support for***recursion***
  + Activation*records of method calls.*

**Can a class have a static inner class?** – A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of the outer class, including private.
* The static nested class cannot access non-static (instance) data members.

Spring - @Primary annotation example

In Spring framework, the @Primary annotation is used to give higher preference to a bean, when there are multiple beans of same type.

The @Primary annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

String s1 = "Hello";

String s2 = s1;

String s3 = **new** String("Hello");

System.***out***.println(s1 == s2);  //true

System.***out***.println(s1 == s3);  //false

System.***out***.println(s2 == s3);  //false

**public** **class** Demo {

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

                 String s1 = "Hello";

                 String s2 = **new** StringBuilder("Hello").toString();

                 System.***out***.println(s1 == s2);  //false

                 System.***out***.println(s1.equals(s2)); //true

         }

}

**-  HashSet** is Implemented using a HashMap. Elements are not ordered. The add, remove, and contains methods have constant time **complexity O(1)**.

**-  TreeSet** is implemented using a tree structure (**red-black** **tree** in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods has time **complexity O(log (n)**). It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.

o First major difference between HashSet and TreeSet is performance. HashSet is faster than TreeSet and should be preferred choice if sorting of element is not required.

o HashSet allows one null object but TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw java.lang.NullPointerException.

o HashSet is backed by HashMap while TreeSet is backed by NavigableMap in Java.

o HashSet uses equals() method to compare two object in Set and for detecting duplicates, while TreeSet uses compareTo() method for same purpose. If equals() and compareTo() are not consistent, i.e. for two equal object equals should return true while compareTo() should return zero, than it will break contract of Set interface and will allow duplicates in Set implementations like TreeSet

o HashSet doesn't guaranteed any order while TreeSet maintains objects in Sorted order defined by either Comparable or Comparator method in Java.

o TreeSet does not allow to insert Heterogeneous objects. It will throw ClassCastException at Runtime if trying to add heterogeneous objects, whereas HashSet allows heterogeneous objects.

**Combining vs Composing**

Combining is a process where a BiFunction/BiConsumer is used for combining the results from two stages into one, whereas composing is the process where the result of the first stage is fed into the second stage to get a composite result.

**Producer Consumer Problems –** In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

**Problem**   
To make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

**Solution**  
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.   
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.   
**Implementation of Producer Consumer Class**

* A **LinkedList list** – to store list of jobs in queue.
* **A Variable Capacity** – to check for if the list is full or not
* A mechanism to control the insertion and extraction from this list so that we do not insert into list if it is full or remove from it if it is empty.

**Prefer composition over Inheritance?**

With all the undeniable benefits provided by inheritance, here are some of its disadvantages.

Disadvantages of Inheritance:

1. You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).
2. Inheritance exposes a subclass to details of its parent class implementation, that's why it's often said that **inheritance breaks encapsulation** (in a sense that you really need to focus on interfaces only not implementation, so reusing by sub classing is not always preferred).
3. The tight coupling provided by inheritance makes the implementation of a subclass very bound up with the implementation of a super class that any change in the parent implementation will force the sub class to change.
4. Excessive reusing by sub-classing can make the inheritance stack very deep and very confusing too.

On the other hand Object composition is defined at runtime through objects acquiring references to other objects. In such a case these objects will never be able to reach each-other's protected data (no encapsulation break) and will be forced to respect each other's interface. And in this case also, implementation dependencies will be a lot less than in case of inheritance.

**Comparing Composition and Inheritance**

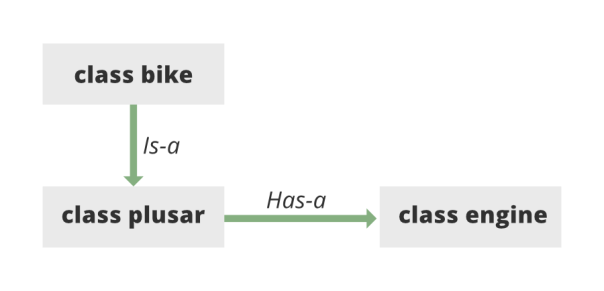
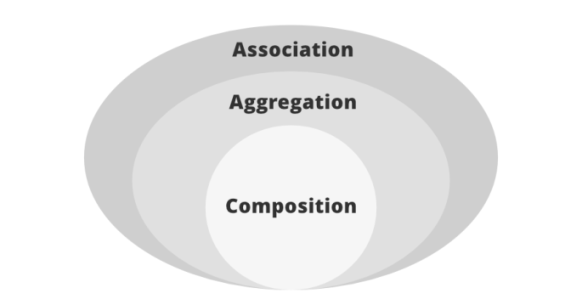
* It is easier to change the class implementing composition than inheritance. The change of a super-class impacts the inheritance hierarchy to subclasses.
* You can't add to a subclass a method with the same signature but a different return type as a method inherited from a super class. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding).
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
  + **Don't use inheritance just to get code reuse** If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
  + **Don't use inheritance just to get at polymorphism** If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

**Summary**

* IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
* Has-a relationship is composition relationship which is a productive way of code reuse.

**What is Has-A-Relation in Java?**

[Association](https://www.geeksforgeeks.org/association-composition-aggregation-java/) is the relation between two separate classes which establishes through their Objects. [Composition](https://www.geeksforgeeks.org/association-composition-aggregation-java/) and [Aggregation](https://www.geeksforgeeks.org/association-composition-aggregation-java/) are the two forms of association. In Java, a [Has-A](https://www.geeksforgeeks.org/association-composition-aggregation-java/) relationship is otherwise called composition. It is additionally utilized for code reusability in Java. In Java, a Has-A relationship essentially implies that an example of one class has a reference to an occasion of another class or another occurrence of a similar class. For instance, a vehicle has a motor; a canine has a tail, etc. In Java, there is no such watchword that executes a Has-A relationship. Yet, we generally utilize new catchphrases to actualize a Has-A relationship in Java.



[Has-a](https://www.geeksforgeeks.org/association-composition-aggregation-java/) is a special form of Association where:

* It represents the Has-A relationship.
* It is a unidirectional association i.e. a one-way relationship. For example, here above as shown pulsar motorcycle has an engine but vice-versa is not possible and thus unidirectional in nature.
* **In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.**

**Illustration:**

This shows that class Pulsar Has- an engine. By having a different class for the engine, we don’t need to put the whole code that has a place with speed inside the Van class, which makes it conceivable to reuse the Speed class in numerous applications.

In an Object-Oriented element, the clients don’t have to make a big deal about which article is accomplishing the genuine work. To accomplish this, the Van class conceals the execution subtleties from the clients of the Van class. Thus, essentially what happens is the clients would ask the Van class to do a specific activity and the Van class will either accomplish the work without help from anyone else or request that another class play out the activity.

**Implementation:**Here is the implementation of the same which is as follows:

1. Car class has a couple of instance variable and few methods
2. Maserati is a type of car that extends the Car class that shows Maserati is a Car. Maserati also uses an Engine’s method, stop, using composition. So it shows that a Maserati has an Engine.
3. The Engine class has the two methods *start()* and *stop()* that are used by the Maserati class.

**Why Java 8 Interface has static method?**

1. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
2. Java interface static method helps us in providing security by not allowing implementation classes to override them.
3. We can’t define interface static method for Object class methods; we will get compiler error as “This static method cannot hide the instance method from Object”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.
4. We can use java interface static methods to remove utility classes such as Collections and move all of its static methods to the corresponding interface, that would be easy to find and use.

**Why Java 8 interface has default methods?**

1. A Java interface default method has bridge down the differences between interfaces and abstract classes.
2. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
3. Java interface default methods will help us in removing base implementation classes; we can provide default implementation and the implementation classes can chose which one to override.
4. One of the major reasons for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
5. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.

PS. The volatile keyword ensures that multiple threads handle the singleton instance correctly.

**Static methods and static blocks**

Static methods belong to the class and they will be loaded into the memory along with the class, you can invoke them without creating an object. (Using the class name as reference).

Whereas a static block is a block of code with a static keyword, used to initialize the static members. JVM executes static blocks before the main method at the time of class loading.

Accessing a non-final variable inside lambda expressions will cause a compile-time error, **but that doesn’t mean that we should mark every target variable as *final.***

According to the “[**effectively final**](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html)” concept, a compiler treats every variable as *final* as long as it is assigned only once.

It's safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.

**Can a class have a static inner class?** – A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of the outer class, including private.
* The static nested class cannot access non-static (instance) data members.

Spring - @Primary annotation example

In spring framework, the **@Primary** annotation is used to give higher preference to a bean, when there are multiple beans of same type.

The **@Primary** annotation may be used on any class directly or indirectly annotated with **@Component** or on methods annotated with **@Bean**.

**What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?**

- Java.util.concurrent.BlockingQueue is a queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

- BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.

- BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.

- BlockingQueue interface is part of the Java collections framework and it’s primarily used for implementing the producer-consumer problem.

**Q6) Can two threads call two different static synchronized methods of the same class?**

Ans) No. The static synchronized methods of the same class always block each other as only one lock per class exists. So, no two static synchronized methods can execute at the same time.

As far as I know, there are basically three different synchronization features in Java:

* Monitor-Objects (used with synchronize keyword)
* [Locks](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html) (e.g. ReentrantLock)
* [Semaphores](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html) (Quite like Locks, but they provide a pool of permits which can be claimed to enter a critical section a Semaphore with a single available Token works equivalent to a Lock)

class A {

public synchronized void methodA() {

//method A

}

public synchronized void methodB() {

// method B

}

}

Both methods lock the same monitor. Therefore, you can't simultaneously execute them on the same object from different threads (one of the two methods will block until the other is finished).

[**Can two threads access a synchronized method at the same time?**](https://stackoverflow.com/questions/9382015/can-two-threads-access-a-synchronized-method-at-the-same-time)

* Yes, if the method is called on ***different*** instances of the class.
* The two threads can't simultaneously call synchronized methods on *the same* instance of the class. This is the case even if the two threads call different methods (if the instance is the same).

**Does a static synchronized method block a non-static synchronized method?**

No, the thread executing the static synchronized method holds a lock on the **class** and the thread executing the non-static synchronized method holds the lock on the **object** on which the method has been called, these two locks are different and these threads do not block each other.

**Filter**: - A filter as the name suggests is a Java class executed by the servlet container for each incoming HTTP request and for each http response. This way is possible to manage HTTP incoming requests before them reach the resource, such as a JSP page, a servlet, or a simple static page; in the same way is possible to manage HTTP outbound response after resource execution.

**Interceptor**: - Spring Interceptors are like Servlet Filters but they act in Spring Context so are many powerful to manage HTTP Request and Response, but they can implement more sophisticated behaviour because can access to all spring context. The preHandle(), postHandle() and afterCompletion() are the methods present in Handler Interceptors.

Request Scopes a single bean definition to the lifecycle of a single HTTP request; **that is each HTTP request will have its own instance of a bean created off the back of a single bean definition.**

Prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made.

🡪Maven plug-in are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin.

Plugins are the central feature of Maven that allow for the reuse of common build logic across multiple projects

**Java Generics advantages** – 1) Type Safety 2) Type casting is not required 3) **compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.🡪Enabling programmers to implement generic algorithms.  
By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

**Public class** Parent {

**Public void** run() {

System.***out***.println("Run-Parent");

}

**Public void** walk() {

System.***out***.println("Walk-Parent");

}

}

**class** Child **extends** Parent {

**public void** run() {

**super**.run();

System.***out***.println("Run Child");

}

**publicvoid** walk() {

**super**.run();

System.***out***.println("Walk Child");

}

}

**class** Main {

**publicstaticvoid** main(String[] args) {

Parent p = **new** Child();

p.run();

p.walk();

}

}

Run-Parent🡪Run Child🡪Run-Parent🡪Walk Child

Microservices Design Patterns

1. Decomposition Patterns

Domain based micro services 🡪 Business Process based micro services 🡪 Atomic Transaction based 🡪Strangler pattern 🡪 Sidecar pattern

1. Integration Patterns

Gateway Pattern 🡪 Process Aggregation Pattern 🡪 Edge Pattern

1. Data Pattern

Single Service database 🡪 Shared Service database 🡪 CQRS 🡪 Synchronous eventing

1. Operational Pattern

Log aggregation patterns 🡪 Metrics Aggregation Pattern 🡪Tracing Pattern 🡪 External Configuration 🡪 Service Discovery

Frameworks to develop microservices – Spring Boot, Drop-wizard, Micronaut, Eclipse Micro profile.

**Program to find top 3 salaried employees named from list of employees using Java stream API.**

@Builder

@Data

@AllArgsConstructor

@NoArgsConstructor

**Public class** Employee {

**private** String name;

**private** Integer age;

**private** Double salary;

**public** String toString() {

DecimalFormatdformat = **new**DecimalFormat(".##");

**return**"Employee Name:" + **this**.name + " Age:" + **this**.age + " Salary:" + dformat.format(**this**.salary);

}

// Standard equals() and hashcode() implementations go here

}

**Class** MaxMinWithCollectors {

**static** List<Employee>*employeeList* = Arrays.*asList*(**new** Employee("Tom Jones", 45, 15000.00),

**new** Employee("Tom Jones", 45, 7000.00), **new** Employee("Ethan Hardy", 65, 8000.00),

**new** Employee("Nancy Smith", 22, 10000.00), **new** Employee("Deborah Sprightly", 29, 9000.00));

**publicstaticvoid** main(String[] args) {

Optional<Employee>maxSalaryEmp = *employeeList*.stream()

.collect(Collectors.*maxBy*(Comparator.***comparing***(Employee::getSalary)));

System.***out***.println("Employee with max salary:" + (maxSalaryEmp.isPresent() ? maxSalaryEmp.get() : "Not Applicable"));

Optional<Employee>minAgeEmp = *employeeList*.stream()

.collect(Collectors.*minBy*(Comparator.*comparing*(Employee::getAge)));

System.***out***.println("Employee with min age:" + (minAgeEmp.isPresent() ? minAgeEmp.get() : "Not Applicable"));

List<String>values = *employeeList*.stream()

.sorted(Comparator.*comparingDouble*(Employee::getSalary))

.map(e->e.getName())

.limit(3)

.collect(Collectors.*toList*());

System.***out***.println(values);

List<Employee>sortedEmployees = employees.stream()

.sorted(Comparator.*comparing*(Employee::getFirstName).thenComparing(Employee::getLastName))

.collect(Collectors.*toList*());

Map<Department, Long> result = employeeList.stream().collect(Collectors.*groupingBy*(MyEmployee::getDepartment,

Collectors.*counting*()));

System.***out***.println(result);

}

}

1) Atomic variables in Java provide the **same memory semantics as a volatile variable**, but with an added feature of making **compound action** atomic.  
2) It provides a convenient method to perform atomic increment, decrement, CAS operations. See [Javadoc](https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/atomic/package-summary.html) for java.util.concurrent.atomic package and individual classes.  
  
3) Useful methods are addAndGet(int delta), compareAndSet(int expect, int update), incrementAndGet() and decrementAndGet()

@RestController is the combination of @Controller and @ResponseBody.  
  
Read more: <https://javarevisited.blogspot.com/2020/04/difference-between-atomic-volatile-and-synchronized-in-java-multi-threading.html#ixzz6pgJhUSwt>

**Program to group employees based on departments using Java stream**

<https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/>

@Builder

@Data

@AllArgsConstructor

@NoArgsConstructor

**publicclass**MyEmployee {

**private** String name;

**private** Integer age;

**private** Double salary;

**private** Department department;

}

**publicenum** Department {

***MARKETING***, ***LEGAL***, ***HR***, ***OPERATIONS***;

}

**publicclass**GroupEmpDemo {

**publicstaticvoid** main(String[] args) {

List<MyEmployee>employeeList = Arrays.*asList*(

**new**MyEmployee("Tom Jones", 45, 12000.00,Department.***MARKETING***),

**new**MyEmployee("Harry Major", 26, 20000.00, Department.***LEGAL***),

**new**MyEmployee("Ethan Hardy", 65, 30000.00, Department.***LEGAL***),

**new**MyEmployee("Nancy Smith", 22, 15000.00, Department.***MARKETING***),

**new**MyEmployee("Catherine Jones", 21, 18000.00, Department.***HR***),

**new**MyEmployee("James Elliot", 58, 24000.00, Department.***OPERATIONS***),

**new**MyEmployee("Frank Anthony", 55, 32000.00, Department.***MARKETING***),

**new**MyEmployee("Michael Reeves", 40, 45000.00, Department.***OPERATIONS***));

Map<Department, List<MyEmployee>>collect = employeeList.stream()

.collect(Collectors.*groupingBy*(MyEmployee::getDepartment));

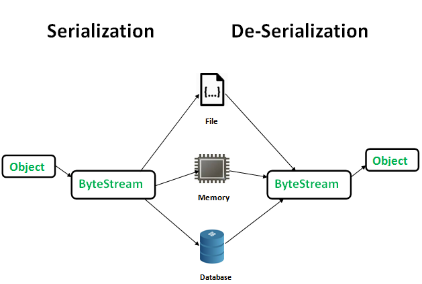
System.***out***.println(collect);

}

}

**What is serialization and de-serialization?**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface.  
The **ObjectOutputStream** class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj) throws IOException

The **ObjectInputStream** class contains **readObject()** method for deserializing an object.

public final Object readObject() throws IOException, ClassNotFoundException

**Advantages of Serialization**  
1. To save/persist state of an object. 2. To travel an object across a network.

Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface.  
Serializable is a **marker interface** (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are: - Cloneable and Remote.

Points to remember  
1. If a parent class has implemented Serializable interface, then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static Data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is de-serialized.  
5. Associated objects must be implementing Serializable interface.

**SerialVersionUID –** The serialization runtime associates with each serializable class a version number, called a **serialVersionUID**, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an **InvalidClassException**. A serializable class can declare its own **serialVersionUID** explicitly by declaring a field named **serialVersionUID** that must be static, final, and of type long:

**ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;**

🡪If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations and can thus result in unexpected **InvalidClassExceptions** during deserialization. Therefore, to guarantee a consistent **serialVersionUID** value across different java compiler implementations, a serializable class must declare an explicit **serialVersionUID** value. It is also strongly advised that explicit **serialVersionUID** declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class — **serialVersionUID** fields are not useful as inherited members.

How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

There are mainly 3 concepts which can break singleton property of a class. Let’s discuss them one by one.

**Reflection:** [Reflection](https://www.geeksforgeeks.org/reflection-in-java/) can be caused to destroy singleton property of singleton class, as shown in following example:

// Singleton class

**class** Singleton {

// public instance initialized when loading the class

**publicstatic** Singleton *instance* = **new** Singleton();

**private** Singleton() {

// private constructor

}

}

**publicclass**GFG {

**publicstaticvoid** main(String[] args) {

Singleton instance1 = Singleton.*instance*;

Singleton instance2 = **null**;

**try** {

Constructor[] constructors = Singleton.**class**.getDeclaredConstructors();

**for** (Constructorconstructor : constructors) {

// Below code will destroy the singleton pattern

constructor.setAccessible(**true**);

instance2 = (Singleton) constructor.newInstance();

**break**;

}

} **catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println("instance1.hashCode():- " + instance1.hashCode());

System.***out***.println("instance2.hashCode():- " + instance2.hashCode());

}

}

instance1.hashCode():- 118352462

instance2.hashCode():- 1550089733

After running this class, you will see that hashCodes are different that means, 2 objects of same class are created, and singleton pattern has been destroyed.

**Overcome reflection issue:** To overcome issue raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e., it does not allow lazy initialization.

//Java program for Enum type singleton

public enum Singleton{

INSTANCE;

}

As Enum don’t have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can’t invoke them by ourselves.**JVM handles the creation and invocation of Enum constructors internally.** As Enums don’t give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can’t break singleton property in case of Enums.

**Serialization:-** [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

**class**Singleton**implements** Serializable {

// public instance initialized when loading the class

**publicstatic** Singleton *instance* = **new** Singleton();

**private** Singleton() {

// private constructor

}

}

**publicclass**GFG {

**publicstaticvoid** main(String[] args) {

**try** {

Singleton instance1 = Singleton.*instance*;

ObjectOutputout = **new**ObjectOutputStream(**new**FileOutputStream("file.text"));

out.writeObject(instance1);

out.close();

// deserailize from file to object

ObjectInputin = **new**ObjectInputStream(**new**FileInputStream("file.text"));

Singleton instance2 = (Singleton) in.readObject();

in.close();

System.***out***.println("instance1 hashCode:- " + instance1.hashCode());

System.***out***.println("instance2 hashCode:- " + instance2.hashCode());

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

instance1 hashCode:- 1118140819

instance2 hashCode:- 1078694789

As you can see, hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

**Overcome serialization issue: -** To overcome this issue, we have to implement method readResolve() method.

// implement readResolve method

protected Object readResolve(){

return instance;

}

instance1 hashCode:- 1118140819

instance2 hashCode:- 1118140819

Above both hashcodes are same hence no other instance is created.

readResolve() is used for *replacing* the object read from the stream. The only use I've ever seen for this is enforcing singletons; when an object is read, replace it with the singleton instance. This ensures that nobody can create another instance by serializing and deserializing the singleton.

**Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is a concept to create duplicate objects. Using clone, we can create copy of object. Suppose we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

**class**SuperClass**implements** Cloneable {

**int**i = 10;

@Override

**protected** Object clone() **throws**CloneNotSupportedException {

**returnsuper**.clone();

}

}

// Singleton class

**class** Singleton **extends** SuperClass {

// public instance initialized when loading the class

**publicstatic** Singleton *instance* = **new** Singleton();

**private** Singleton() {

// private constructor

}

@Override

**protected** Object clone() **throws**CloneNotSupportedException {

**throw new** CloneNotSupportedException();

}

}

**Public class**GFG {

**Public static void** main(String[] args) **throws**CloneNotSupportedException {

Singleton instance1 = Singleton.*instance*;

Singleton instance2 = (Singleton) instance1.clone();

System.***out***.println("instance1 hashCode:- " + instance1.hashCode());

System.***out***.println("instance2 hashCode:- " + instance2.hashCode());

}

}

Exception in thread "main" java.lang.CloneNotSupportedException

at com.mastercard.customer.data.management.batch.Singleton.clone(GFG.java:25)

at com.mastercard.customer.data.management.batch.GFG.main(GFG.java:32)

Now we have stopped user to create clone of singleton class. If you don’t want to throw exception you can also return the same instance from clone method.

**class**SuperClass**implements** Cloneable {

**int**i = 10;

@Override

**protected** Object clone() **throws**CloneNotSupportedException {

**returnsuper**.clone();

}

}

// Singleton class

**class** Singleton **extends**SuperClass {

// public instance initialized when loading the class

**publicstatic** Singleton *instance* = **new** Singleton();

**private** Singleton() {

// private constructor

}

@Override

**protected** Object clone() **throws**CloneNotSupportedException {

**return***instance*;

}

}

**publicclass**GFG {

**publicstaticvoid** main(String[] args) **throws**CloneNotSupportedException {

Singleton instance1 = Singleton.*instance*;

Singleton instance2 = (Singleton) instance1.clone();

System.***out***.println("instance1 hashCode:- " + instance1.hashCode());

System.***out***.println("instance2 hashCode:- " + instance2.hashCode());

}

}

instance1 hashCode:- 118352462

instance2 hashCode:- 118352462

**Association, Composition and Aggregation in Java?**

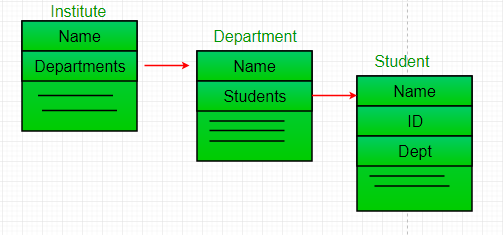
Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.  
In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

Aggregation –

It is a special form of Association where:

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one-way relationship. For ex, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not affect the other entity

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make an Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).  
It represents a **Has-A** relationship.

[](https://www.geeksforgeeks.org/media.geeksforgeeks.org/wp-content/uploads/Reference.png)

**When do we use Aggregation ??**  
Code reuse is best achieved by aggregation.

Composition –

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

***Aggregation vs Composition***

1. ***Dependency:****Aggregation implies a relationship where the child****can exist independently****of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child****cannot exist independent****of the parent. Example: Human and heart, heart don’t exist separate to a Human*
2. ***Type of Relationship:****Aggregation relation is****“has-a”****and composition is****“part-of”****relation.*
3. ***Type of association:****Aggregation is a****weak****Association whereas Composition is strong Association.*

The main difference between the PUT and PATCH method is that >>PUT method uses the request URI to supply a modified version of the requested resource which replaces the original version of the resource, whereas the PATCH method supplies a set of instructions to modify the resource. If the PATCH document is larger than the size of the new version of the resource sent by the PUT method, then the PUT method is preferred.

**Create immutable class in java and advantages?**

**Immutable object** is instance whose state doesn’t change after it has been initialized. For example, [String](https://www.journaldev.com/16928/java-string) is an immutable class and once instantiated its value never changes.

* An immutable class is good for caching purposes because you don’t have to worry about the value changes.
* It is inherently [**thread-safe**](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t have to worry about thread safety in case of multi-threaded environment.
* Ease of caching, Protection against null reference error, better encapsulation.

One area where immutability makes an especially big difference is concurrency: **immutable objects can safely be shared among multiple threads**, whereas mutable objects must be made thread-safe via careful design and implementation - usually this is far from a trivial task.

**How to Create an immutable class in Java?**

To create an immutable class in Java, you must do the following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables.
4. Make all mutable fields final so that its value can be assigned only once.
5. Initialize all the fields via a constructor performing **deep copy**.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.
7. To understand points 4 and 5, let’s run the sample Final class that works well, and values don’t get altered after instantiation.

@Getter

**publicfinalclass**FinalClassExample {

**privatefinalint**id;

**privatefinal** String name;

**privatefinal** HashMap<String, String>testMap;

// Accessor function for mutable object

@SuppressWarnings("unchecked")

**public**HashMap<String, String>getTestMap() {

**return** (HashMap<String, String>) testMap.clone();

}

**public**FinalClassExample(**int**i, String n, Map<String, String>hm) {

System.***out***.println("Performing Deepfor Object initialization");

**this**.id = i;

**this**.name = n;

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

String key;

Iterator<String>it = hm.keySet().iterator();

**while** (it.hasNext()) {

key = it.next();

tempMap.put(key, hm.get(key));

}

**this**.testMap = tempMap;

}

**publicstaticvoid** main(String[] args) {

HashMap<String, String>h1 = **new** HashMap<>();

h1.put("1", "first");

h1.put("2", "second");

String s = "original Value";

**int**i = 10;

FinalClassExamplece = **new**FinalClassExample(i, s, h1);

// Lets see whether its copy by field or reference

System.***out***.println(s == ce.getName());🡪 true

System.***out***.println(h1 == ce.getTestMap());🡪 false

// print the ce values

System.***out***.println("ce id:" + ce.getId()+", ce name:" + ce.getName()+", cetestMap:" + ce.getTestMap());

// change the local variable values

i= 20;

s= "modified";

h1.put("3", "third");

// print the values again

System.***out***.println("ce id after local variable change:" + ce.getId());

System.***out***.println("ce name after local variable change:" + ce.getName());

System.***out***.println("cetestMap after local variable change:" + ce.getTestMap());

HashMap<String, String>hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.***out***.println("cetestMap after changing variable from accessor methods:" + ce.getTestMap());

}

}

Performing Deep for Object initialization

true

false

ce id:10, cename:original Value, cetestMap:{1=first, 2=second}

ce id after local variable change:10

ce name after local variable change:original Value

cetestMap after local variable change:{1=first, 2=second}

cetestMap after changing variable from accessor methods:{1=first, 2=second}

|  |  |
| --- | --- |
| MongoDB | Postgres |
| **Schema evolves as your application evolves:** MongoDB is great to use when you need to have a database whose schema can evolve as your application evolves. Sometimes, we just don’t know what the entire scope of our application will be like and having the flexibility to adjust the schema to fit your needs is desirable. | **Data depends on reliability of ACID:** Because transactions in this model follow ACID properties, it’s a good choice for any sort of fintech business. When you absolutely need to control the state of your data, use a relational database like Postgres. |
| **Horizontal scaling means that MongoDB is fast:** Because MongoDB doesn’t depend on more processing power as it scales, but instead combines power with additional machines, it can be a bit faster. If you plan to have tens or hundreds of thousands of documents of data, it might be a good idea to use the horizontal scaling approach by using MongoDB. | **Schemas have an identified relationship:** If the structure of your data can be identified with a 1:1, 1:many, or many:1 relationship, having tables that identify those schemas and having foreign keys will be useful. |
| ACID not a priority: MongoDB is great if ACID is not your priority, even with the advent of Mongo’s ACID transaction paradigm. It’s up to what the higher priority is for you and your business. | Schemas have an identified relationship: If the structure of your data can be identified with a 1:1, 1:many, or many:1 relationship, having tables that identify those schemas and having foreign keys will be useful. |

**>>Synchronized in Java?**

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So, it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

// A Class used to send a message

**class**Sender {

**publicvoid** send(String msg) {

System.***out***.println("Sending " + msg);

**try** {

Thread.*sleep*(1000);

} **catch** (Exception e) {

System.***out***.println("Thread interrupted.");

}

System.***out***.println(msg + "Sent");

}

}

// Class for send a message using Threads

**class**ThreadedSend**extends** Thread {

**private** String msg;

Sendersender;

// Recieves a message object and a string message to be sent

ThreadedSend(String m, Senderobj) {

msg = m;

sender = obj;

}

**publicvoid**run() {

// Only one thread can send a message at a time.

**synchronized** (sender) {

// synchronizing the snd object

sender.send(msg);

}

}

}

// Driver class

**class**SyncDemo {

**publicstaticvoid** main(String args[]) {

Sendersnd = **new**Sender();

ThreadedSendS1 = **new**ThreadedSend(" Hi ", snd);

ThreadedSendS2 = **new**ThreadedSend(" Bye ", snd);

// Start two threads of ThreadedSend type

S1.start();

S2.start();

// wait for threads to end

**try** {

S1.join();

S2.join();

} **catch** (Exception e) {

System.***out***.println("Interrupted");

}

}

}

Sending Hi

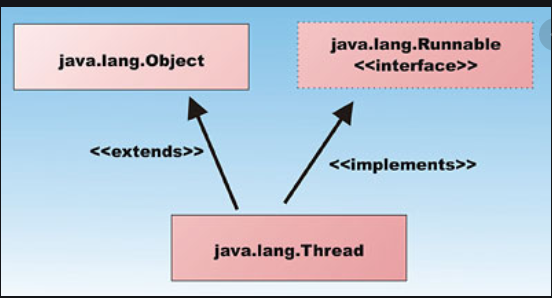
Hi Sent

Sending Bye

Bye Sent

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the whole send() block as synchronized and it would produce the same result. Then we don’t have to synchronize the Message object inside the run() method in ThreadedSend class.

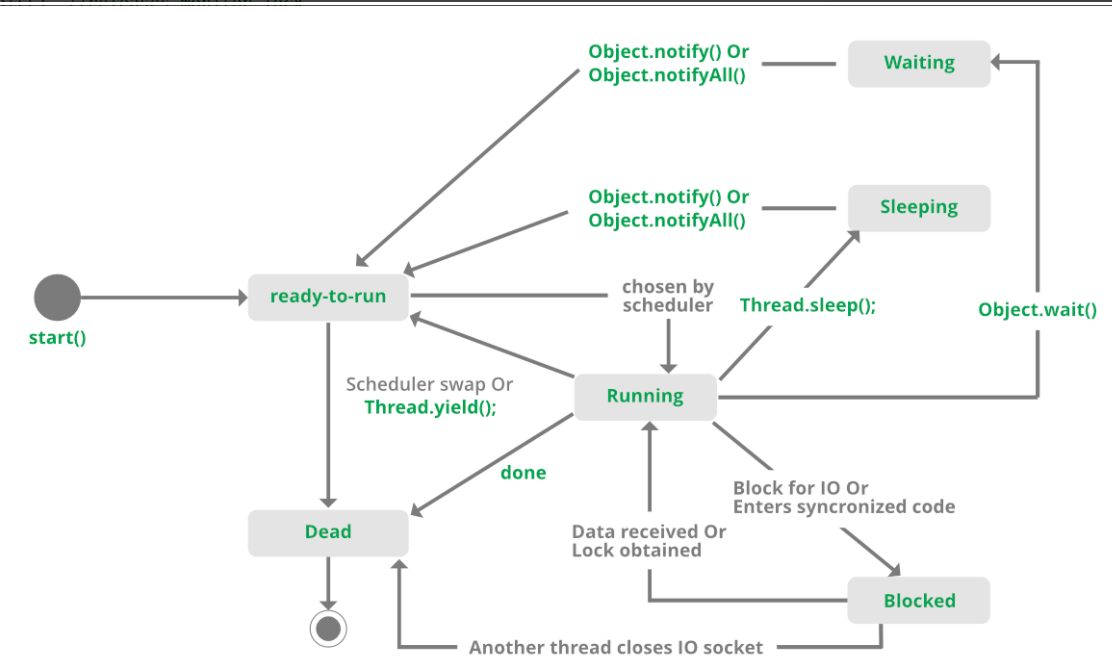


*- In the JVM, every object and class are logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock (sometimes called a mutex) is associated with each object and class. This is called a semaphore in operating systems books, mutex is a binary semaphore.*

*- If one thread owns a lock on some data, then no others can obtain that lock until the thread that owns the lock releases it. It would be not convenient if we need to write a semaphore all the time when we do multi-threading programming. Luckily, we don't need to since JVM does that for us automatically.*

*- To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods. Once the code is embedded with synchronized keyword, it is a monitor region. The locks are implemented in the background automatically by JVM.*

|  |  |
| --- | --- |
| **Wait** | **Sleep** |
| Wait() method belongs to Object class | Sleep() method belongs to Thread class. |
| Wait() method releases lock during Synchronization. | Sleep() method does not release the lock on object during Synchronization. |
| Wait() should be called only from Synchronized context. | There is no need to call sleep() from Synchronized context. |
| Wait() is not a static method. | Sleep() is a static method. |
| Sleep() Has Two Overloaded Methods:   * sleep(long millis)millis: milliseconds * sleep(long millis,int nanos) nanos: Nanoseconds | Wait() Has Three Overloaded Methods:   * wait() * wait(long timeout) * wait(long timeout, int nanos) |
| public final void wait(long timeout) | public static void sleep(long millis) throws Interrupted\_Execption |



**class**GfG {

**private static** Object *LOCK* = **new** Object();

**public static void** main(String[] args) **throws** InterruptedException {

Thread.*sleep*(1000);

System.***out***.println("Thread '" + Thread.*currentThread*().getName() + "' is woken after sleeping for 1 second");

**synchronized** (*LOCK*) {

*LOCK*.wait(1000);

System.***out***.println("Object '" + *LOCK* + "' is woken after" + " waiting for 1 second");

}

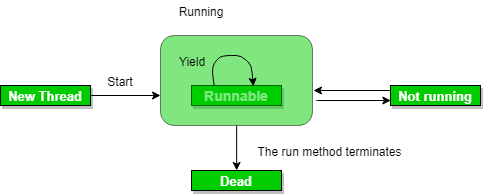
}

}

**Java Concurrency – yield(), sleep() and join() methods?**

We can prevent the execution of a thread by using one of the following methods of Thread class.

1. **yield():** Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 must have to wait for 5 hours to just finish 5 minutes job. **In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yeild() helps us in doing so.  
   yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



**Use of yield method:**

1. Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
2. If any thread executes yield method, thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority, then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.

public static native void yield()

// MyThread extending Thread

**class**MyThread**extends** Thread {

**publicvoid**run() {

**for** (**int**i = 0; i< 5; i++)

System.***out***.println(Thread.*currentThread*().getName() + " in control");

}

}

// Driver Class

**publicclass**yieldDemo {

**publicstaticvoid** main(String[] args) {

MyThreadt = **new**MyThread();

t.start();

**for** (**int**i = 0; i< 5; i++) {

// Control passes to child thread

Thread.*yield*();

// After execution of child Thread main thread takes over

System.***out***.println(Thread.*currentThread*().getName() + " in control");

}

}

}

main in control

main in control

main in control

main in control

main in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

The yield() thread first is higher than the other thread because main thread is always pausing its execution and giving chance to child thread(with same priority).

**Note:**

* Once a thread has executed yield method and there are many threads with same priority is waiting for processor, then we can't specify which thread will get execution chance first.
* The thread which executes the yield method will enter in the Runnable state from Running state.
* Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.
* Underlying platform must provide support for pre-emptive scheduling if we are using yield method.

sleep(): This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.  
**publicclass**SleepDemo**implements** Runnable {

Thread t;

**publicvoid**run() {

**for** (**int**i = 0; i< 4; i++) {

System.***out***.println(Thread.*currentThread*().getName() + " " + i);

**try** {

// thread to sleep for 1000 milliseconds

Thread.*sleep*(1000);

}**catch** (Exception e) {

System.***out***.println(e);

}

}

}

**publicstaticvoid** main(String[] args) **throws** Exception {

Thread t = **new** Thread(**new**SleepDemo());

// call run() function

t.start();

Thread t2 = **new** Thread(**new**SleepDemo());

// call run() function

t2.start();

}

}

Thread-0 0

Thread-1 0

Thread-1 1

Thread-0 1

Thread-0 2

Thread-1 2

Thread-0 3

Thread-1 3

Note:

* we can make a thread to be in sleeping state for a specified period.
* Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

**yield() vs sleep()**

yield:() indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the current thread will continue to run.

sleep(): causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

[join():](https://www.geeksforgeeks.org/joining-threads-in-java/) The join() method of a Thread instance is used to join the start of a thread’s execution to end of other thread’s execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.  
The join() method waits at most this much milliseconds for this thread to die. A timeout of 0 means to wait forever.

// waits for this thread to die.

**public final void join() throws InterruptedException**

// waits at most this much milliseconds for this thread to die

**public final void join(long millis) throws InterruptedException**

// waits at most milliseconds plus nanoseconds for this thread to die.

The **java.lang.Thread.join(long millis, int nanos)**

**publicclass**JoinDemo**implements** Runnable {

**publicvoid**run() {

Thread t = Thread.*currentThread*();

System.***out***.println("Current thread: " + t.getName());

// checks if current thread is alive

System.***out***.println("Is Alive? " + t.isAlive());

}

**publicstaticvoid** main(String args[]) **throws** Exception {

Thread t = **new** Thread(**new** JoinDemo());

t.start();

// Waits for 1000ms this thread to die.

t.join(1000);

System.***out***.println("\nJoining after 1000" + " mili seconds: \n");

System.***out***.println("Current thread: " + t.getName());

// Checks if this thread is alive

System.***out***.println("Is alive? " + t.isAlive());

}

}

Current thread: Thread-0

Is Alive? True

Joining after 1000 mili seconds:

Current thread: Thread-0

Is alive? False

* If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
* Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.

**Advantages of microservices:**

* It is easy to manage as it is relatively smaller in size.
* If there’s any update in one of the microservices, then we need to redeploy only that microservice.
* Microservices are self-contained and hence, deployed independently. Their start-up and deployment time are relatively less.
* It is very easy for a new developer to on-board the project as he needs to understand only a particular microservice providing the functionality he will be working on and not the whole system.
* If a particular microservice is facing a large load because of the users using that functionality in excess, then we need to scale out that microservice only. Hence, microservices architecture supports horizontal scaling.
* Each microservice can use different technology based on the business requirements.
* If a particular microservice goes down due to some bug, then it doesn’t affect other microservices and the whole system remains intact, continues providing other functionalities to the users.

**Disadvantages of microservices:**

* Being a distributed system, it is much more complex than the monolithic applications. Its complexity increases with the increase in number of microservices.
* Skilled developers are required to work with microservices architecture which can identify the microservices and manage their inter-communications.
* Independent deployment of microservices is complicated.
* Microservices are costly in terms of network usage as they need to interact with each other, and all these remote calls results into network latency.
* Microservices are less secure relative to monolithic applications due to the inter-services communication over the network.
* Debugging is difficult as the control flows over many microservices and to point out why and where exactly the error occurred is a difficult task.

**Find the 3rd highest element –**

**int** value = IntStream.*of*(number).sorted().skip(number.length- 3).findFirst().getAsInt();

List<Integer> collect = IntStream.*of*(number).distinct().boxed().collect(Collectors.*toList*());

Spring Bean is not thread safe.

**Find 2nd highest salary**

SELECT name, MAX(salary) AS salary FROM employee WHERE salary < (SELECT MAX(salary) FROM employee);

Multiply each element by 5.

Integer[] arr = **new** Integer[] { 1, 2, 3, 4, 5 };

List<Integer> asList = Arrays.*asList*(arr);

List<Integer> values = asList.stream().mapToInt(n -> n \* 5).boxed().collect(Collectors.*toList*());

System.***out***.println(values);

IntStream::**boxed** turns an IntStream into a Stream<Integer>, which you can then collect into a List:

Integer i1 = 1000;

Integer i2 = 1000;

if(i1 != i2)

System.out.println("different objects");

The key to the answer is called *object interning*. Java interns’ small numbers (less than 128), so all instances of Integer(n) with n in the interned range are the same. Numbers greater than or equal to 128 are not interned, hence Integer(1000) objects are not equal to each other.i1.intValue() == i2.intValue()

**How to rebase local branch onto remote master**

git fetch origin # Updates origin/master

git rebase origin/master # Rebases current branch onto origin/master

**public static** <T> Optional<T> ofNullable(T value) {

**return** value == **null** ? *empty*() : *of*(value);

}

**No**, we cannot override main method of java because a static method cannot be overridden.  
The static method in java is associated with class whereas the non-static method is associated with an object.

**Kafka consumers** belonging to the same **consumer group** share a **group** id. The **consumers** in a **group** then divides the topic partitions as fairly amongst themselves as possible by establishing that each partition is only consumed by a single **consumer** from the **group**.

**What is a high-level design document?**

A high-level design document (HLDD) describes the architecture used in the development & development of a particular software product. It usually includes a diagram that depicts the envisioned structure of the software system. Since this is a high-level document, non-technical language is often used.

Cross-origin resource sharing (**CORS**) is a mechanism that allows JavaScript on a web page to make [AJAX](https://howtodoinjava.com/jquery/jquery-ajax-tutorial/) requests to another domain, different from the domain from where it originated.

What are types of pools –

1. FixedThreadPool 2) CacheThreadPool 3) ScheduledThreadPool 4) SingleThrededExecutor

**class** Task **implements** Runnable {

@Override

**publicvoid** run() {

System.***out***.println("Thread Name >> "+ Thread.*currentThread*().getName());

}

}

**publicclass**FixedThreadPool {

**publicstaticvoid** main(String[] args) {

ExecutorService service = Executors.*newFixedThreadPool*(5);

**for** (**int**i = 0; i< 10; i++) {

service.execute(**new** Task());

}

System.***out***.println("Thread Name >> "+ Thread.*currentThread*().getName());

}

}

**Reverse a linked list?**

**publicclass** LinkedList {

**static** Node *head*;

**staticclass** Node {

**int**data;

Node next;

Node(**int**d) {

data = d;

next = **null**;

}

}

// Function to reverse a linkedlist

Node reverse(Node node) {

Node prev = **null**;

Node current = node;

Node next= **null**;

**while** (current != **null**) {

next = current.next;

current.next = prev;

prev = current;

current = next;

}

node = prev;

**return**node;

}

// prints content of double linked list

**void**printList(Node node) {

**while** (node != **null**) {

System.***out***.print(node.data + " ");

node = node.next;

}

}

// Driver Code

**publicstaticvoid** main(String[] args) {

LinkedList list = **new** LinkedList();

list.*head* = **new** Node(85);

list.*head*.next = **new** Node(15);

list.*head*.next.next = **new** Node(4);

list.*head*.next.next.next = **new** Node(20);

System.***out***.println("Given Linked list");

list.printList(*head*);

*head* = list.reverse(*head*);

System.***out***.println("");

System.***out***.println("Reversed linked list ");

list.printList(*head*);

}

}

What is deadlock and how to avoid it?

The **deadlock** is a situation when two or more threads try to access the same object that is acquired by another thread. Since the threads wait for releasing the object, the condition is known as **deadlock**. The situation arises with more than two threads.

What is deadlock in Java?

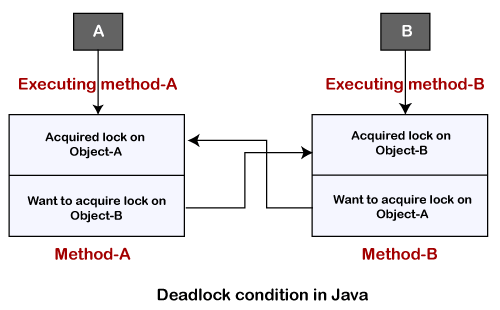
In the thread, each object has a lock. To acquire a lock, [Java](https://www.javatpoint.com/java-tutorial) provides synchronization to lock a method or code block. It allows that at a time only one thread can access that method.

Nevertheless, if a thread wants to execute a synchronized method it first tries to acquire a lock. It is possible that another thread has already acquired that lock then the thread (that wants to acquire the lock) will have to wait until the previous thread does not release the lock.

Let's understand it through an example.

Ex – Suppose there are two threads A and B. The thread A and B acquired the lock of Object-A and Object-B, respectively. Assume that thread A executing method A and wants to acquire the lock on Object-B, while thread B is already acquired a lock on Object-B.

On the other hand, thread B also tries to acquire a lock on Object-A, while thread A is acquired a lock on Object-A. In such a situation both threads will not complete their execution and wait for releasing the lock. The situation is known as, [**deadlock**](https://www.javatpoint.com/deadlock-in-java).



<https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

Difference between QueryParam and PathVariable –

**@QueryParam** is used to access key/value pairs in the query string of the URL (the part after the ?). For example in the url <http://example.com?q=searchterm>, you can use**@QueryParam("q")** to get the value of q.

**@PathParam** is used to match a part of the URL as a parameter. For example in an url of the form [http://example.com/books/{bookid}](http://example.com/books/%7bbookid%7d), you can use **@PathParam("bookid")** to get the id of a book.

|  |  |  |
| --- | --- | --- |
| # | Factor | Description |
| I | Codebase | There should be exactly one codebase for a deployed service with the codebase being used for many deployments. |
| II | Dependencies | All dependencies should be declared, with no implicit reliance on system tools or libraries. |
| III | Config | Configuration that varies between deployments should be stored in the environmen |
| IV | Backing services | All backing services are treated as attached resources and attached and detached by the execution environment. |
| V | Build, release, run | The delivery pipeline should strictly consist of build, release, run. |
| VI | Processes | Applications should be deployed as one or more stateless processes with persisted data stored on a backing service. |
| VII | Port binding | Self-contained services should make themselves available to other services by specified ports. |
| VIII | Concurrency | Concurrency is advocated by scaling individual processes. |
| IX | Disposability | Fast start-up and shutdown are advocated for a more robust and resilient system. |
| X | Dev/Prod parity | All environments should be as similar as possible. |
| XI | Logs | Applications should produce logs as event streams and leave the execution environment to aggregate. |
| XII | Admin Processes | Any needed admin tasks should be kept in source control and packaged with the application. |

|  |  |
| --- | --- |
| **Principal Name** | **What it says?** |
| Single Responsibility Principle | One class should have one and only one responsibility |
| Open Closed Principles | Software components should be open for extensions, but closed for modifications |
| Liskov’s Substitution Principle | Derived types must be completely substitutable for their base types. |
| Interface Segregation Principle | Clients should not be forced to implement the unnecessary methods which they will not use. |
| Dependency Inversion Principle | Depends on abstractions and on concretions. |

**Introduction to SOLID Principles**

*SOLID* is the acronym for a set of practices that, when implemented together, makes the code more adaptive to change.

SOLID principles form the fundamental guidelines for building object-oriented applications that are robust, extensible, and maintainable.

**Example**

We can find plenty of classes in all popular Java libraries which follow single responsibility principle. For example, in [Log4j2](https://howtodoinjava.com/log4j2/log4j-2-json-configuration-example/), we have different classes with logging methods, different classes are logging levels and so on.

In our application level code, we define model classes to represent real time entities such as Person, Employee, Account etc. Most of these classes are examples of SRP principle because when we need to change the state of a person, only then we will modify the Person class, and so on.

In the given example, we have two classes Person and Account. Both have single responsibility to store their specific information. If we want to change the state of Person, then we do not need to modify the class Account and vice-versa.

|  |
| --- |
| Person.java |
| publicclassPerson {      privateLong personId;      privateString firstName;      privateString lastName;      privateString age;      privateList<Account> accounts;  } |
| Account.java |
| publicclassAccount{      privateLong guid;      privateString accountNumber;      privateString accountName;      privateString status;      privateString type;  } |

**Open Closed Principle**

OCP is the second principle which we should keep in mind while designing our application. It states:

“Software components should be open for extension but closed for modification”. It means that the application classes should be designed in such a way that whenever fellow developers want to change the flow of control in specific conditions in application, all they need to extend the class and override some functions and that’s it.

If other developers are not able to write the desired behaviour due to constraints put by the class, then we should reconsider refactoring the class.

I do not mean here that anybody can change the whole logic of the class, but one should be able to override the options provided by software in a non-harmful way permitted by the software.

**Example**

If we investigate any good framework like struts or spring, we will see that we cannot change their core logic and request processing, but we modify the desired application flow just by extending some classes and plugin them in configuration files.

For example, spring framework has class [DispatcherServlet](https://howtodoinjava.com/spring5/webmvc/spring-dispatcherservlet-tutorial/). This class acts as a **front controller** for String based web applications. To use this class, we are not required to modify this class. All we need is to pass initialization parameters and we can extend its functionality the way we want.

Please note that apart from passing initialization parameters during application startup, we can override methods as well to modify the behaviour of target class by extending the classes. For example, struts Action classes are extended to override the request processing logic.

|  |
| --- |
| Extending Struts Action |
| publicclassHelloWorldAction extendsAction{      @Override      publicActionForward execute(ActionMapping mapping,              ActionForm form,              HttpServletRequest request,              HttpServletResponse response)              throwsException {          //Process the request      }  } |

**Liskov’s Substitution Principle**

LSP is a variation of previously discussed open closed principle. It says:“Derived types must be completely substitutable for their base types”

LSP means that the classes, fellow developers created by extending our class, should be able to fit in application without failure. This is important when we resort to [polymorphic behaviour](https://howtodoinjava.com/oops/what-is-polymorphism-in-java/) through [inheritance](https://howtodoinjava.com/oops/java-inheritance/).

This requires the objects of the subclasses to behave in the same way as the objects of the superclass. This is mostly seen in places where we do runtime type identification and then cast it to appropriate reference type.

**Example**

An example of LSP can be [custom property editors](https://howtodoinjava.com/spring-boot/custom-property-editor-example/) in Spring framework. Spring provides property editors to represent properties in a different way than the object itself e.g. parsing human readable inputs from HTTP request parameters or displaying human readable values of pure java objects in view layer e.g. Currency or URL.

Spring can register one property editor for one data type, and it is required to follow the constraint mandated by **base class** PropertyEditorSupport. So if any class extends PropertyEditorSupport class, then it can be substituted by everywhere the base class is required.

For example, every book has an ISBN number which is always in a fixed display format. You can have separate representations of ISBN in database and UI. For this requirement, we may write property editor in such a way –

|  |
| --- |
| IsbnEditor.java |
| importjava.beans.PropertyEditorSupport;  importorg.springframework.util.StringUtils;  importcom.howtodoinjava.app.model.Isbn;    publicclassIsbnEditor extendsPropertyEditorSupport {      @Override      publicvoidsetAsText(String text) throwsIllegalArgumentException {          if(StringUtils.hasText(text)) {              setValue(newIsbn(text.trim()));          } else{              setValue(null);          }      }        @Override      publicString getAsText() {          Isbnisbn = (Isbn) getValue();          if(isbn != null) {              returnisbn.getIsbn();          } else{              return"";          }      }  } |

**Interface Segregation Principle**

ISP is applicable to interfaces as a single responsibility principle holds to classes. ISP says:

“Clients should not be forced to implement unnecessary methods which they will not use”

Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf(). Now client ‘A’ wants to use this interface, but he intends to use reports only in PDF format and not in excel. Will he be able to use the functionality easily?

NO. He will have to implement both the methods, out of which one is an extra burden put on him by the designer of the software. Either he will implement another method or leave it blank. This is not a good design.

So what is the solution? Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to users to use only the required functionality only.

**Example**

The best place to look for IPS examples is Java AWT event handlers for handling GUI events fired from keyboard and mouse. It has different listener classes for each kind of event. We only need to write handlers for events, we wish to handle. Nothing is mandatory.

Some of the listeners are –

* FocusListener
* KeyListener
* MouseMotionListener
* MouseWheelListener
* TextListener
* WindowFocusListener

Anytime, we wish to handle any event, just find out a corresponding listener and implement it.

|  |
| --- |
| MouseMotionListenerImpl.java |
| publicclassMouseMotionListenerImpl implementsMouseMotionListener {      @Override      publicvoidmouseDragged(MouseEvent e) {          //handler code      }        @Override      publicvoidmouseMoved(MouseEvent e) {          //handler code      }  } |

**Dependency Inversion Principle**

Most of us are already familiar with the words used in principle’s name. [DI principle](https://howtodoinjava.com/spring-core/spring-ioc-vs-di/) says:

“Depend on abstractions, not on concretions”

In other words. we should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

**Java equals and hashcode –**

The contract between [equals()](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)) and [hashCode()](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Object.html#hashCode()) is basically: *if equals() is true, hashCode() should return the same value.*

From the Javadoc – If two objects are equal according to the **equals(Object)** method, then calling the **hashCode** method of the two objects must produce the same integer result.

🡪But the reason that this hashCode() method is a problem is because it returns the same hash for both equal and unequal instances. For best performance, developers should try to write hashCode() methods that generate distinct int values for unequal instances. This is not required by the contract for hashCode() but it is essential if good performance is important.

For ex, with the method above, every instance that is added to a HashMap will be distributed to the same bucket. The result is that performance degrades. The best hash function will do a good job of distributing unequal instances into separate buckets. For this reason, your case #2 could result in serious performance problems if those instances were used in a HashMap.

<https://www.javamadesoeasy.com/2015/10/exception-handling-quiz-in-java.html>

**Problem 1: If SuperClass doesn’t declare an exception:**

In this problem, two cases arise:

* Case 1: If SuperClass doesn’t declare any exception and subclass declare checked exception

Example:

**class**SuperClass {

// SuperClass doesn't declare any exception

**void** method() {

System.***out***.println("SuperClass");

}

}

// SuperClass inherited by the SubClass

**class**SubClass**extends**SuperClass {

// method() declaring Checked Exception IOException

**void**method() **throws**IOException {

// IOException is of type Checked Exception so the compiler will give Error

System.***out***.println("SubClass");

}

// Driver code

**publicstaticvoid** main(String args[]) {

SuperClasss = **new**SubClass();

s.method();

}

}

Compilation error.

**Conclusion for Handling such Exceptions:** Hence, following conclusions can be derived from the above examples:

1. If SuperClass does not declare an exception, then the SubClass can only declare unchecked exceptions, but not the checked exceptions.
2. If SuperClass declares an exception, then the SubClass can only declare the child exceptions of the exception declared by the SuperClass, but not any other exception.
3. If SuperClass declares an exception, then the SubClass can declare without exception.

**Public class**

MinimumNoOfPlatform {

// Returns minimum number of platforms reqquired

**publicstaticint**findPlatform(**int**arr[], **int**dep[], **int**n) {

// plat\_needed indicates number of platforms needed at a time

**int**plat\_needed= 1, result = 1;

**int**i= 1, j= 0;

// run a nested loop to find overlap

**for** (i = 0; i<n; i++) {

// minimum platform

plat\_needed = 1;

**for** (j = i + 1; j<n; j++) {

// check for overlap

**if** ((arr[i] >= arr[j] &&arr[i] <= dep[j]) || (arr[j] >= arr[i] &&arr[j] <= dep[i]))

plat\_needed++;

}

// update result

result = Math.*max*(result, plat\_needed);

}

**return**result;

}

// Driver Code

**publicstaticvoid** main(String[] args) {

**int**arr[] = { 900, 940, 950, 1100, 1500, 1800 };

**int**dep[] = { 910, 1200, 1120, 1130, 1900, 2000 };

**int**n = 6;

System.***out***.println("Minimum Number of Platforms Required = " + *findPlatform*(arr, dep, n));

}

}

The minimum no. of platform required = 3

**What is deadlock?**

When synchronized keyword in java causes the executing thread to block while waiting for the monitor (lock) associated with the specified object, then Deadlock may occur. Since the executing thread might already holds the object which is required by another thread (also in waiting state). In such a case, two threads end up waiting forever for each other object lock.

**publicclass**DeadLockProgram {

// Creating Object Locks

**static** Object *ObjectLock1* = **new** Object();

**static** Object *ObjectLock2* = **new** Object();

**private staticclass** ThreadName1 **extends** Thread {

**publicvoid**run() {

**synchronized** (*ObjectLock1*) {

System.***out***.println("Thread 1: Has ObjectLock1");

/\*

\* Adding sleep() method so that Thread 2 can lock ObjectLock2

\*/

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedExceptione) {

e.printStackTrace();

}

System.***out***.println("Thread 1: Waiting for ObjectLock 2");

/\*

\* Thread 1 has ObjectLock1 but waiting for ObjectLock2

\*/

**synchronized** (*ObjectLock2*) {

System.***out***.println("Thread 1: No DeadLock");

}

}

}

}

**privatestaticclass** ThreadName2 **extends** Thread {

**publicvoid**run() {

**synchronized** (*ObjectLock1*) {

System.***out***.println("Thread 2: Has ObjectLock2");

/\*

\* Adding sleep() method so that Thread 1 can lock ObjectLock1

\*/

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedExceptione) {

e.printStackTrace();

}

System.***out***.println("Thread 2: Waiting for ObjectLock 1");

/\*

\* Thread 2 has ObjectLock2 but waiting for ObjectLock1

\*/

**synchronized** (*ObjectLock2*) {

System.***out***.println("Thread 2: No DeadLock");

}

}

}

}

**publicstaticvoid** main(String args[]) {

ThreadName1 thread1 = **new** ThreadName1();

ThreadName2 thread2 = **new** ThreadName2();

thread1.start();

thread2.start();

}

}

Thread 1: Has ObjectLock1

Thread 1: Waiting for ObjectLock 2

Thread 1: No DeadLock

Thread 2: Has ObjectLock2

Thread 2: Waiting for ObjectLock 1

Thread 2: No DeadLock

**Best Practices to Avoid Deadlock**

**1. Avoiding Nested Locks:**

This is one of the most common reason to get the Deadlock. If you share the locks with multiple threads, then Deadlock might occur. Try to avoid sharing locks to multiple threads.

**2. Avoiding Unnecessary Locks:**

We should use lock when required. Unnecessary and frequent use of locks can lead to Deadlock which is fatal for the application.

**class** A1 **extends** Thread {

**int**t = 0;

@Override

**publicvoid** run() {

**synchronized** (**this**) {

**for** (**int**i = 0; i<= 10; i++) {

t = t + i;

}

**this**.notify();

}

}

}

**class**Main {

**publicstaticvoid** main(String[] args) **throws** InterruptedException {

A1 a1 = **new** A1();

a1.start();

**synchronized** (a1) {

System.***out***.println("Main Thread");

a1.wait();

System.***out***.println("Main Thread got notified...");

System.***out***.println(a1.t);

}

}

}

Main Thread

Main Thread got notified...

55

**publicclass** Customer {

**int**amount;

Object obj;

**public** Customer(**int**amount) {

**this**.amount = amount;

}

//

**Public synchronized void** withdraw(**int** amount) {

System.***out***.println("Checking Balance to withdraw "+amount);

**if**(**this**.amount < amount) {

System.***out***.println("The current balance is "+**this**.amount+" which is less than withdraw amount, waiting for deposite ...");

**try** {

wait();

} **catch** (InterruptedExceptione) {

e.printStackTrace();

}

}

**this**.amount = **this**.amount - amount;

System.***out***.println("Amount : "+amount+" withdrawn from your account. Available balance :"+**this**.amount);

}

**Public synchronized void** deposits(**int**amount) {

System.***out***.println("Depositing amount "+amount + " ....");

**this**.amount = **this**.amount + amount;

System.***out***.println("Amount : "+amount+" credited your account. Available balance :"+**this**.amount);

}

**publicstaticvoid** main(String[] args) {

Customer customer = **new** Customer(1000);

**new** Thread(() ->customer.withdraw(1500)).start();

**new** Thread(() ->customer.deposits(300)).start();

}

}

**What Is a Feign Client?**

Feign provides an abstraction over REST-based calls via annotation, by which micro-services can use to communicate with each other without writing detailed REST client code.

@SpringBootConfiguration

@EnableAutoConfiguration - The **@EnableAutoConfiguration** annotation enables the auto-configuration of Spring ApplicationContext by scanning the classpath components and registers the beans that are matching various Conditions. SpringBoot provides various AutoConfiguration classes in spring-boot-autoconfigure-{version}.jar, which are responsible for registering various components.

@ComponentScan

@SpringBootApplication

HTTP Status code

|  |  |
| --- | --- |
| 204 | No content |
| 400 | Bad Request |
| 403 | Forbidden |
| 404 | Not Found |
| 409 | Conflict |
| 502 | Bad gateway |
| 405 | Method Not Allowed |
| 401 | Unauthorized |
| 406 | Not Acceptable |

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments.

- Spring Boot supports Tomcat, Jetty, Undertow.

**- Builder Pattern**is one of the **Design Pattern** in Java, it reduces the number of parameters required for a constructor or method invocation via custom types and parameter objects.

**What is Cloning?**

**Cloning** is a process of creating an exact copy of an existing object in the memory. In Java, **clone()** method of **java.lang.Object** class is used for cloning process. This method creates an exact copy of an object on which it is called through **field-by-field assignment** and returns the reference of that object. Not all the objects in java are eligible for cloning process. The objects which implement **Cloneable interface** are only eligible for cloning process. Cloneable interface is a [marker interface](https://javaconceptoftheday.com/marker-interface-java/) which is used to provide the marker to cloning process.

Both shallow copy and deep copy are related to this cloning process. The default version of clone() method creates the shallow copy of an object. To create the deep copy of an object, you have to override the clone() method. Let’s see how these shallow copy and deep copy work.

**Shallow Copy in Java:**

The shallow copy of an object will have exact copy of all the fields of original object. If original object has any references to other objects as fields, then only references of those objects are copied into clone object, copy of those objects are not created. That means any changes made to those objects through clone object will be reflected in original object or vice-versa. Shallow copy is not 100% disjoint from original object. Shallow copy is not 100% independent of original object.

Below is the example which creates the shallow copy of an object ‘**student1**‘.

**class** Course {

String subject1;

String subject2;

String subject3;

**public** Course(String sub1, String sub2, String sub3) {

**this**.subject1 = sub1;

**this**.subject2 = sub2;

**this**.subject3 = sub3;

}

}

**class** Student **implements** Cloneable {

**int**id;

String name;

Course course;

**public** Student(**int**id, String name, Course course) {

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

// Default version of clone() method. It creates shallow copy of an object.

**protected** Object clone() **throws**CloneNotSupportedException {

**returnsuper**.clone();

}

}

**Public class** ShallowCopyInJava {

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

Course science = **new** Course("Physics", "Chemistry", "Biology");

Student student1 = **new** Student(111, "John", science);

Student student2 = **null**;

**try** {

// Creating a clone of student1 and assigning it to student2

student2 = (Student) student1.clone();

} **catch** (CloneNotSupportedExceptione) {

e.printStackTrace();

}

// Printing the subject3 of 'student1'

System.***out***.println(student1.course.subject3); // Output : Biology

// Changing the subject3 of 'student2'

student2.course.subject3 = "Maths";

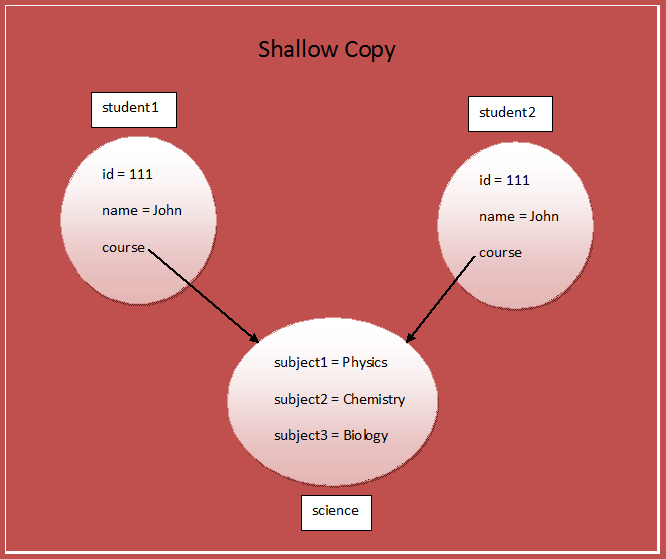
// This change will be reflected in original student 'student1'

System.***out***.println(student1.course.subject3); // Output : Maths

}

}

**student1** is an object of ‘**Student**‘ class which has three fields – **id**, **name** and **course**. The **course** is a reference variable pointing to a ‘**Course**‘ type object. Clone of ‘**student1**‘ is created by calling clone method on it and assigned it to ‘**student2**‘. As default version of clone method creates the shallow copy, the ‘**course**‘ field of both ‘**student1**‘ and ‘**student2**‘ will be pointing to same ‘**Course**‘ object. So, any changes made to this object through **student2**will be reflected in ‘**student1**‘ or vice-versa.



**Deep Copy In Java:**

Deep copy of an object will have exact copy of all the fields of original object just like shallow copy. But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone() method on them. That means clone object and original object will be 100% disjoint. They will be 100% independent of each other. Any changes made to clone object will not be reflected in original object or vice-versa.

To create a deep copy of an object, you have to override the clone() method as demonstrated in the below example.

**class** Course **implements** Cloneable {

String subject1;

String subject2;

String subject3;

**public** Course(String sub1, String sub2, String sub3) {

**this**.subject1 = sub1;

**this**.subject2 = sub2;

**this**.subject3 = sub3;

}

**protected** Object clone() **throws**CloneNotSupportedException {

**returnsuper**.clone();

}

}

**class** Student **implements** Cloneable {

**int**id;

String name;

Course course;

**public** Student(**int**id, String name, Course course) {

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

// Overriding clone() method to create a deep copy of an object.

**protected** Object clone() **throws**CloneNotSupportedException {

Student student = (Student) **super**.clone();

student.course = (Course) course.clone();

**return**student;

}

}

**publicclass**DeepCopyInJava {

**publicstaticvoid** main(String[] args) {

Course science = **new** Course("Physics", "Chemistry", "Biology");

Student student1 = **new** Student(111, "John", science);

Student student2 = **null**;

**try** {

// Creating a clone of student1 and assigning it to student2

student2 = (Student) student1.clone();

} **catch** (CloneNotSupportedExceptione) {

e.printStackTrace();

}

// Printing the subject3 of 'student1'

System.***out***.println(student1.course.subject3); // Output : Biology

// Changing the subject3 of 'student2'

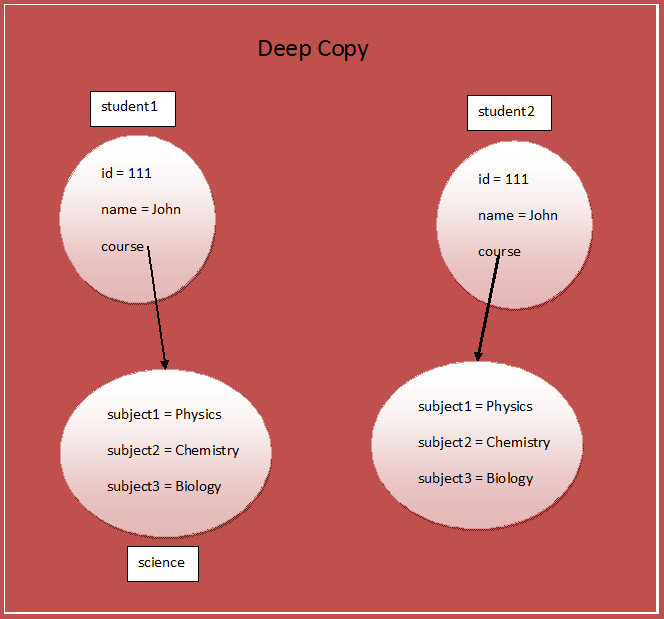
student2.course.subject3 = "Maths";

// This change will not be reflected in original student 'student1'

System.***out***.println(student1.course.subject3); // Output : Biology

}

}



|  |  |
| --- | --- |
| **Shallow Copy** | **Deep Copy** |
| Cloned Object and original object are **not** 100% disjoint. | Cloned Object and original object are 100% disjoint. |
| Any changes made to cloned object will be reflected in original object or vice versa. | Any changes made to cloned object will not be reflected in original object or vice versa. |
| Default version of clone method creates the shallow copy of an object. | To create the deep copy of an object, you must override clone method. |
| Shallow copy is preferred if an object has only primitive fields. | Deep copy is preferred if an object has references to other objects as fields. |
| Shallow copy is fast and less expensive. | Deep copy is slow and very expensive. |

[JAX-WS](http://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services) - is Java API for the XML-Based Web Services - a standard way to develop a Web- Services in [SOAP](http://en.wikipedia.org/wiki/SOAP) notation (Simple Object Access Protocol).

[JAX-RS](http://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services) - Java API for RESTful Web Services. RESTful Web Services are represented as resources and can be identified by Uniform Resource Identifiers ([URI](http://en.wikipedia.org/wiki/Uniform_resource_identifier)).

CDC - Consumer-Driven Contracts is an approach to ensure service communication compatibility, in which Consumer and Provider make an agreement about the format of the data they transfer between each other. Sharing those contracts and communicating the test results is important part of implementing proper CDC tests.

**How to process large file size in java?**

The most likely efficient way to do this is:

* Have a single thread that reads the input file. Harddisks are at their fastest when reading sequentially.
* Do **not** read it into memory all at once! That is a huge waste of memory which could be used much better to speed the processing!
* Instead, have this single thread read a bundle of entries (maybe 100, maybe 1000, this is a tuning parameter) at once and submit them to a thread to process. If each line represents a record, the reading thread can defer all the parsing (other than looking for newlines) to the processing threads. But even if not, it is very unlikely that the parsing of records is your bottleneck.
* Do the thread handlings through a [fixed size thread pool](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool-int-), choose the size to be the number of CPU cores on the machine, or maybe a bit more.
* If your database is an SQL database, make sure the individual threads access the database through a connection pool and do all their DB updates for one bundle of entries in a single transaction and using batch inserts.

You might want to use [Spring Batch](http://projects.spring.io/spring-batch/) for this, as it will guide you towards doing the right thing. But it is somewhat over engineered and hard to use.

[**Choose between ExecutorService's submit and ExecutorService's execute**](https://stackoverflow.com/questions/3929342/choose-between-executorservices-submit-and-executorservices-execute)**?**

* There is a difference concerning exception/error handling.
* A task queued with **execute()** that generates some **Throwable** will cause the **UncaughtExceptionHandler** for the **Thread** running the task to be invoked. The default **UncaughtExceptionHandler**, which typically prints the **Throwable** stack trace to System.err, will be invoked if no custom handler has been installed.
* On the other hand, a **Throwable** generated by a task queued with **submit()** will bind the **Throwable** to the Future that was produced from the call to **submit().** Calling **get()** on that Future will throw an **ExecutionException** with the original **Throwable** as its cause (accessible by calling **getCause()** on the **ExecutionException**).

**From the book Java concurency essentials:**

**CorePoolSize**: The **ThreadPoolExecutor** has an attribute **corePoolSize** that determines how many threads it will start until new threads are only started when the queue is full

**MaximumPoolSize**: This attribute determines how many threads are started at the maximum. You can set this to **Integer.MAX\_VALUE** in order to have no upper boundary

**Handling Exception for thread pool executor.**

When you submit a task to the executor, it returns you a [FutureTask](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html) instance. [FutureTask.get()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html#get%28%29) will re-throw any exception thrown by the task as an **ExecutorException**. So, when you iterate through the **List<Future>** and call get on each, catch **ExecutorException** and invoke an orderly shutdown.

Create Custom HashMap

**class** Entry<K, V> {

K key;

V val;

**public** K getKey() {

**return**key;

}

**publicvoid**setKey(K key) {

**this**.key = key;

}

**public** V getVal() {

**return**val;

}

**publicvoid**setVal(V val) {

**this**.val = val;

}

@Override

**publicint**hashCode() {

**int**prime = 13;

**int**mul = 11;

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

**int**hashCode = prime \* mul + key.hashCode();

**return**hashCode;

}

**return** 0;

}

@Override

**publicboolean**equals(Object o) {

**if** (**this** == o) {

**returntrue**;

}

**if** (o == **null** || **this**.getClass().getName() != o.getClass().getName()) {

**returnfalse**;

}

Entrye = (Entry) o;

**if** (**this**.key == e.key) {

**returntrue**;

}

**returnfalse**;

}

}

**publicclass**HashMapImpl<K, V> {

**privatefloat**loadfactor = 0.75f;

**privateint**capacity = 100;

**privateint**size = 0;

**private** Entry<K, V>table[] = **new** Entry[capacity];

**privateint**Hashing(**int**hashCode) {

**int**location = hashCode % capacity;

System.***out***.println(">>> Location:" + location);

**return**location;

}

**publicint** size() {

**returnthis**.size;

}

**publicboolean**isEmpty() {

**if** (**this**.size == 0) {

**returntrue**;

}

**returnfalse**;

}

**publicboolean**containsKey(Object key) {

**if** (key == **null**) {

**if** (table[0].getKey() == **null**) {

**returntrue**;

}

}

**int**location = Hashing(key.hashCode());

Entrye = **null**;

**try** {

e = table[location];

} **catch** (NullPointerExceptionex) {

}

**if** (e != **null**&&e.getKey() == key) {

**returntrue**;

}

**returnfalse**;

}

**publicboolean**containsValue(Object value) {

**for** (**int**i = 0; i<table.length; i++) {

**if** (table[i] != **null**&&table[i].getVal() == value) {

**returntrue**;

}

}

**returnfalse**;

}

**public** V get(K key) {

V ret = **null**;

Entry<K, V>e = **null**;

**try** {**if** (key == **null**) {

e = table[0];

} **catch** (NullPointerExceptionex) {

}

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

**return** (V) e.getVal();

}

} **else** {

**int**location = Hashing(key.hashCode());

Entry<K, V>e = **null**;

**try** {

e = table[location];

} **catch** (NullPointerExceptionex) {

}

**if** (e != **null**&&e.getKey() == key) {

**return** (V) e.getVal();

}

}

**return**ret;

}

**public** V put(K key, V val) {

V ret = **null**;

**if** (key == **null**) {

ret = putForNullKey(val);

**return**ret;

} **else** {

**int**location = Hashing(key.hashCode());

**if** (location>= capacity) {

System.***out***.println("Rehashing required");

**returnnull**;

}

Entry<K, V>e = **null**;

**try** {

e = table[location];

} **catch** (NullPointerExceptionex) {

}

**if** (e != **null**&&e.getKey() == key) {

ret = (V) e.getVal();

} **else** {

Entry<K, V>eNew = **new** Entry<K, V>();

eNew.setKey(key);

eNew.setVal(val);

table[location] = eNew;

size++;

}

}

**return**ret;

}

**private** V putForNullKey(V val) {

Entry<K, V>e = **null**;

**try** {

e = table[0];

} **catch** (NullPointerExceptionex) {

}

V ret = **null**;

**if** (e != **null**&&e.getKey() == **null**) {

ret = (V) e.getVal();

e.setVal(val);

**return**ret;

} **else** {

Entry<K, V>eNew = **new** Entry<K, V>();

eNew.setKey(**null**);

eNew.setVal(val);

table[0] = eNew;

size++;

}

**return**ret;

}

**public** V remove(K key) {

**int**location = Hashing(key.hashCode());

V ret = **null**;

**if** (table[location].getKey() == key) {

**for** (**int**i = location; i<table.length; i++) {

table[i] = table[i + 1];

}

}

**return**ret;

}

**publicstaticvoid** main(String[] args) {

HashMapImpl<Integer, String>hashMap = **new**HashMapImpl<Integer, String>();

hashMap.put(10, "Apple");

hashMap.put(1, "Orange");

hashMap.put(79, "Grape");

System.***out***.println("Val at 79 " + hashMap.get(79));

System.***out***.println("Val at 1 " + hashMap.get(1));

System.***out***.println("Val at 10 " + hashMap.get(10));

System.***out***.println("Val at 2 " + hashMap.get(2));

hashMap.put(**null**, "Pear");

System.***out***.println("Val at null " + hashMap.get(**null**));

System.***out***.println("Hashmap has key at null:" + hashMap.containsKey(**null**));

System.***out***.println("Hashmap has value at null:" + hashMap.containsValue("Pear"));

System.***out***.println("Size of Map:" + hashMap.size());

}

}

>>> Location:10

>>> Location:1

>>> Location:79

>>> Location:79

Val at 79 Grape

>>> Location:1

Val at 1 Orange

>>> Location:10

Val at 10 Apple

>>> Location:2

Val at 2 null

Val at null Pear

Hashmap has key at null:true

Hashmap has value at null:true

Size of Map:4

[**Overriding**](https://www.geeksforgeeks.org/overriding-in-java/): Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in the superclass (or base class).   
The implementation to be executed is decided at run-time and a decision is made according to the object used for the call. Note that signatures of both methods must be the same. Refer [Overriding in Java](https://www.geeksforgeeks.org/overriding-in-java/) for details.  
[**Overloading**](https://www.geeksforgeeks.org/overloading-in-java/): Overloading is also a feature of OOP languages like Java that is related to compile-time (or static) polymorphism. This feature allows different methods to have the same name, but different signatures, especially the number of input parameters and type of input parameters. Note that in both C++ and Java, [methods cannot be overloaded according to](https://www.geeksforgeeks.org/g-fact-75/)the [return type.](https://www.geeksforgeeks.org/g-fact-75/)

|  |
| --- |
| **Can we overload static methods?**  Yes, we can have two or more static methods with the same name, but differences in input parameters. For example, consider the following Java program.  // filename Test.java  **public class Test {**  **public static void foo() {**  **System.out.println("Test.foo() called ");**  **}**  **public static void foo(int a) {**  **System.out.println("Test.foo(int) called ");**  **}**  **public static void main(String args[]){**  **Test.foo();**  **Test.foo(10);**  **}**  **}** |

**Output**

Test.foo() called

Test.foo(int) called

**What is CompletableFuture?**

A CompltableFuture is used for asynchronous programming. Asynchronous programming means writing non-blocking code. It runs a task on a separate thread than the main application thread and notifies the main thread about its progress, completion or failure.

In this way, the main thread does not block or wait for the completion of the task. Other tasks execute in parallel. Parallelism improves the performance of the program.

A CompletableFuture is a class in Java. It belongs to java.util.cocurrent package. It implements CompletionStage and Future interface.

**CompletionStage**

* It performs an action and returns a value when another completion stage completes.
* A model for a task that may trigger other tasks.

Hence, it is an element of a chain. When more than one thread attempt to complete - complete exceptionally or cancel a CompletableFuture, only one of them succeeds.

**Future vs. CompletableFuture**

A CompletableFuture is an extension to Java's Future API which was introduced in Java 8. A Future is used for asynchronous Programming. It provides two methods, **isDone()** and **get()**. The methods retrieve the result of the computation when it completes.

Limitations of the Future

* A Future cannot be mutually complete.
* We cannot perform further action on a Future's result without blocking.
* Future has not any exception handling.
* We cannot combine multiple futures.

Future has so many limitations, that's why we have CompletableFuture. CompletableFuture provides a broad set of methods for creating multiple Futures, chaining, and combining. It also has comprehensive exception handling support.