**CORE JAVA INTERVIEW QUESTION ANSWER**

**New features in Java SE 8**

1. Lambda Expressions
2. Pipelines and Streams
3. Date and Time API
4. Default Methods
5. Type Annotations
6. Nashhorn JavaScript Engine
7. Concurrent Accumulators
8. Parallel operations
9. PermGen Error Removed
10. TLS SNI

**New features in Java SE 7**

1. Strings in switch Statement
2. Type Inference for Generic Instance Creation
3. Multiple Exception Handling
4. Support for Dynamic Languages
5. Try with Resources
6. Java nio Package
7. Binary Literals, underscore in literals
8. Diamond Syntax
9. Automatic null Handling

**New features in Java SE 6**

1. **Collection Enhancement**

These new collection interfaces are provided:

[**Deque**](http://docs.oracle.com/javase/6/docs/api/java/util/Deque.html) - a *double ended queue*, supporting element insertion and removal at both ends. Extends the [Queue](http://docs.oracle.com/javase/6/docs/api/java/util/Queue.html) interface.

[**BlockingDeque**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingDeque.html) - a Deque with operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element. Extends both the [Deque](http://docs.oracle.com/javase/6/docs/api/java/util/Deque.html) and [BlockingQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html)interfaces. (This interface is part of [java.util.concurrent](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html).)

[**NavigableSet**](http://docs.oracle.com/javase/6/docs/api/java/util/NavigableSet.html) - a SortedSet extended with navigation methods reporting closest matches for given search targets. A NavigableSet may be accessed and traversed in either ascending or descending order. This interface is intended to supersede the SortedSet interface.

[**NavigableMap**](http://docs.oracle.com/javase/6/docs/api/java/util/NavigableMap.html) - a SortedMap extended with navigation methods returning the closest matches for given search targets. A NavigableMap may be accessed and traversed in either ascending or descending key order. This interface is intended to supersede the SortedMap interface.

[**ConcurrentNavigableMap**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) - a ConcurrentMap that is also a NavigableMap. (This interface is part of [java.util.concurrent](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html).)

The following concrete implementation classes have been added:

**ArrayDeque** - efficient resizable-array implementation of the Deque interface.

**ConcurrentSkipListSet** - concurrent scalable skip list implementation of the NavigableSet interface.

**ConcurrentSkipListMap** - concurrent scalable skip list implementation of the ConcurrentNavigableMap interface.

[**LinkedBlockingDeque**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/LinkedBlockingDeque.html) - concurrent scalable optionally bounded FIFO blocking deque backed by linked nodes.

**AbstractMap**.**SimpleEntry** - simple mutable implementation of [Map.Entry](http://docs.oracle.com/javase/6/docs/api/java/util/Map.Entry.html)

**AbstractMap**.**SimpleImmutableEntry** - simple immutable implementation of [Map.Entry](http://docs.oracle.com/javase/6/docs/api/java/util/Map.Entry.html)

These existing classes have been retrofitted to implement new interfaces:

**LinkedList** - retrofitted to implement the **Deque** interface.

**TreeSet** - retrofitted to implement the **NavigableSet** interface.

**TreeMap** - retrofitted to implement the **NavigableMap** interface.

Two new methods were added to the **Collections** utility class:

* [newSetFromMap(Map)](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#newSetFromMap(java.util.Map)) - creates a general purpose Set implementation from a general purpose Map implementation.

There is no IdentityHashSet class, but instead, just use

Set<Object> identityHashSet=Collections.newSetFromMap( new IdentityHashMap<Object, Boolean>());

* [asLifoQueue(Deque)](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#asLifoQueue(java.util.Deque)) - returns a view of a Deque as a Last-in-first-out (Lifo) Queue.

The Arrays utility class now has methods [copyOf](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html#copyOf(int[],%20int)) and [copyOfRange](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html#copyOfRange(int[],%20int,%20int)) that can efficiently resize, truncate, or copy subarrays for arrays of all types.

Before:

int[] newArray = new int[newLength];

System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);

After:

int[] newArray = Arrays.copyOf(a, newLength);

1. [Drag and Drop](http://www.oracle.com/technetwork/articles/javase/javase6-139238.html)
2. JDBC 4.0 API
3. Integrated Web Services.
4. Java nio package added.
5. [JConsole is Officially Supported in Java SE 6](http://download.oracle.com/javase/6/docs/technotes/guides/management/enhancements.html#jconsole)
6. Scripting framework.

**New features in J2SE 5.0**

1. **Generics**
2. **Enhanced for Loop**
3. **Autoboxing/Unboxing**
4. **Typesafe Enums**
5. **Varargs**
6. **Static Import**
7. **Metadata (Annotations)**

**Maven Vs Ant**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Maven** | **Ant** |
| 1 | Maven **has a convention** to place source code, compiled code etc. So we don't need to provide information about the project structure in pom.xml file. | Ant **doesn't has formal conventions**, so we need to provide information of the project structure in build.xml file. |
| 2 | Maven is **declarative**, everything you define in the pom.xml file. | Ant is **procedural**, you need to provide information about what to do and when to do through code. You need to provide order. |
| 3 | There is **life cycle** in Maven. | There is **no life cycle** in Ant. |
| 4 | There is a concept of repository | No such concept. |
| 5 | **Standard naming conventions** likegroupId, artifactId, version | No such standard. |
| 6 | Maven uses the concep to plugiin for reusable task. | Ant uses script for reusable task. |
| 7 | Easily work with JUnits | Not so easy. |

**DOM vs SAX vs STAX Parser**

|  |  |  |
| --- | --- | --- |
| **DOM Parser** | **SAX Parser** | **STAX Parser** |
| The DOM models involves creating IN-MEMORY objects representing an entire document tree and the complete infoset state for an XML document | The SAX Push Model  The SAX push model means that it is the SAX parser that calls your handler, not your handler that calls the SAX parser. The SAX parser thus "pushes" events into your handler. Here it is, summarized:  SAX Parser --> Handler  With a push model you have not control over how and when the parser iterates over the file. Once you start the parser, it iterates all the way until the end, calling your handler for each and every XML event in the input XML document. | The StAX Pull Model  The StAX pull model means that it is your "handler" class that calls the parser, not the other way around. Thus your handler class controls when the parser is to move on to the next event in the input. In other words, your handler "pulls" the XML events out of the parser. Additionally, you can stop the parsing at any point. The pull model is summarized like this:  Handler --> StAX Parser |
| It support Schema Validation | It support Schema Validation | It does not support Schema Validation |
| It support for writing XML | SAX does not have support for writing XML | StAX has Support for XML Writing |
| It loads entire file into memory | It loads part of file into memory | It load part of file into memory |
| It is faster than SAX and Stax as it access the entire file in memory | It is slower than DOM | It is slower than DOM |
| It is useful for small size file | It is useful for large file | It is also useful for large file but it is easier than SAX. |
|  |  |  |
| More chance of Java.lang.OutOfMemoryError:java heap space | Less chance | Less chance |

**http://javaconceptoftheday.com/types-of-references-in-java-strong-soft-weak-and-phantom/**

## Weak Reference

When there are one or more reference to an object it will not be garbage collected in Java. But this rule depends on what type of reference it is. If an object has only weak reference associated with other objects, then it is a valid candidate for garbage collection.

Let us take a sample scenario to understand it better. Let TextView be an object (recently programming in [Android](http://javapapers.com/android/android-hello-world/) and so using its class for example :-)) and we will have program generated ids used for its identification. These ids are used in some other object for referencing the TextViews.

Map textViewIdMap = new HashMap();

textViewIdMap.put(textView1, iD1);

textViewIdMap.put(textView2, iD2);

Key is TextView object and value is the Id. Now, during the execution of the program we have removed a TextView object say textView1. We do not require that view object so we have made it null. Now, what will happen to the key-value pair(textView1, iD1) stored in HashMap. This pair as of now makes no sense and it is not required as that textview itself is null.

So, programmatic we need to ensure that, when a textView is removed then its corresponding entry in the map should be removed. Only then, that object becomes a candidate for garbage collection. Otherwise, even though it is not used at run-time, this stale object will not be garbage collected.

## WeakHashMap

There is a predefined Map which uses weak reference. This is an implementation of Map interface and exactly same as HashMap but with only difference of weakrefernce. Key-value pairs extext WeakReference class.

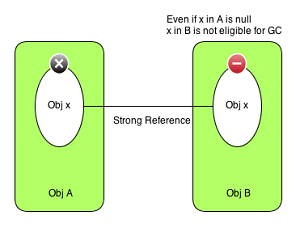
**private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V>**

In WeakHashMap there is a private method name expungeStaleEntires(), which is used to remove stale entries as given in above scenario. This method is internally used before get/getEntry/put/resize/size /…. all methods. Only after expunge done the regular operation is done. It just makes a stale entry ‘null’ and releases it for garbage collection. You may read its source code for more detail.

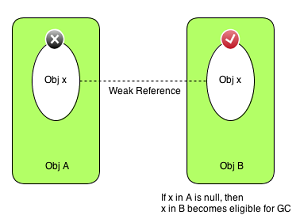
## Strong Reference vs Weak Reference

Strong reference is not something new, it is nothing but what we use in our daily programming. The default reference for objects. Strong reference is strongest of all references, if there there is a strong reference garbage collecter will not even come to this object :-)

**StringBuilder iD1 = new StringBuilder();**

[](http://javapapers.com/wp-content/uploads/2013/02/Strong-Reference.png)

WeakReference weakTextView1 =newWeakReference(textView1);

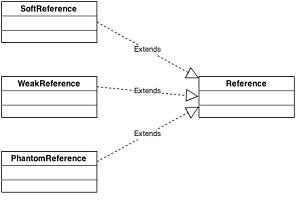
[](http://javapapers.com/wp-content/uploads/2013/02/Weak-Reference.png)

## Types of References in Java

There are four types of references in Java,

1. Strong Reference
2. Soft Reference
3. Weak Reference
4. Phantom Reference

above is listed in the same order as their strength.

[](http://javapapers.com/wp-content/uploads/2013/02/References.png)

### Soft Reference

Soft Reference is slightly stronger that weak reference. Soft reference allows for garbage collection, but begs the garbage collector to clear it only if there is no other option. That is, it is eligible for garbage collection, but garbage collector can remove it based on the memory crunch it has. If it is left with little memory and it is in a position to reclaim memory, then it will collect the soft references.

**Phantom Reference**

Phantom reference is different from soft/weak reference. This can never be reached before it is cleared in the program. Calling the get() on it return null always. When a phantom referenced object is garbage collected, it is notified via the ReferenceQueue. We can use this to find when it is garbage collected and no use other than that.

1. **Strong references.**
2. **Soft references.**
3. **Weak references.**
4. **Phantom references.**

These references are different solely because of the existence of a garbage collection mechanism in the JVM. That is because, the decision of reclaiming memory from the object heap depends not only on the fact that there are active references to an object, but also on the type of reference to the object.

Lets try to see how each of them differ from one another.

Strong references

As a normal programmer(that is, if you consider programmers to be 'normal'), we are most likely to encounter only the most ubiquitous form of references - Strong references. The name of this reference type should by itself give you an idea of the importance of the reference in the opinion of the garbage collector. Any object with an active strong reference to it, will never be garbage collected(except in rare circumstances, such as cyclical references).

Say, for example, when you create a class Employee, and you create a reference to a new employee object like this

|  |  |
| --- | --- |
| 1 | Employee emp = new Employee(); |

you are actually creating a strong reference. An object to which a reference is created in this way will be eligible for garbage collection only when the reference is to it is pointed to null.

So, if you write this

|  |  |
| --- | --- |
| 1 | emp=null; |

and there are no more references created to this object until the garbage collector runs again, this object will become eligible for garbage collection, and is most likely to be garbage collected.

So, strong references are pretty simple to understand, probably because most of the code that you commonly write mainly consists of strong references. And we let the smart garbage collector take care of cleaning up the memory for us.

However, there might be cases where you need more control over the lifetime of an object. What if you want to keep some kind of a pool of objects, but still want it to be garbage collected if your JVM is running out of memory? In such cases although you may want a reference to the object most of the time, you are willing to let go of a memory reference for a better JVM performance, when times are crucial. This is where the other kinds of references come into the picture.

The 3 remaining Reference types that I discuss here are subclasses of an abstract class Reference.

***Soft references***

The soft reference class is used to create a soft reference to an existing object that is already referred to by a strong reference. The actual object on the heap that is being referred to is called the *referent*.

In order to create a SoftReference to an object, you first need to create a strong reference to that object and then pass that object as a parameter to the constructor of the SoftReference object, as shown in the code below.

Employeeemp=newEmployee(); //1

SoftReference<Employee>softRef=newSoftReference<employee>(emp);

In the above code, a few interesting things have happened.

* In line #1,a (referent)object is created and allocated memory on the heap that represents the Employee object.
* Again, in line #1, a strong reference is created to the newly created employee object. We call this reference 'emp'.
* In line #2, a new SoftReference object is created and allocated memory on the heap. This is a special object because it contains an internal reference to the (referent) object passed to it in the constructor, i.e. in our case, the actual Employee object on the heap.
* Again in line #2, a strong reference is created to the SoftReference object on the heap. We call this reference 'softRef'.

So, in total, we have 2 strong references created. And the object that represents the soft reference, internally holds a reference to the actual employee object.

So what is it that makes soft references useful? Soft references are considered to be a special kind of reference by the garbage collector. Let us assume that at a later point of time, we nullify the 'emp' reference as follows and no new strong references were created to this emp object.

emp=null;

Now lets assume that the garbage collector decides to run at this point of time. What it will see is that the current employee object only has a soft reference pointing to it, and no strong references. The garbage collector may optionally choose to reclaim the memory occupied by the employee object. But what makes soft references even more special is the fact that the garbage collector will always reclaim the memory of objects that have only soft references pointing to them before it throws an OutOfMemory Error. This gives the garbage collector some sort of a last chance to save the life of the JVM.

At any given point of time, if the garbage collector has not yet reclaimed the memory of the actual referent object, you can get a strong reference to the referent via the get method.

EmployeeresurrectedEmp=softRef.get();

The above code resurrects the employee object in the sense that the garbage collector will not consider it a candidate for garbage collection because it now has a Strong Reference pointing to it.

This makes soft references highly useful in creating object pools, where the size of the pool needs to be dynamic. If you do not know the size of a pool when you begin, or choose not to set a minimum or a maximum size on the object pool, instead you want it to grow dynamically, and at the same time, you want to give the JVM a chance to cleanup unused objects, in that case SoftReferences are a perfect fit.

***Weak References***

Weak references are even more awesome. Thats because seemingly the garbage collector has no regard for an object that only has a week reference. What that means is that the garbage collector will make it eligible for garbage collection because object only has a week reference pointing to it. Not only is that awesome and useful, but desirable as well in some scenarios.

For example, let  us assume that you need to maintain some metadata related to a user per database connection. In such a case you will be tempted to use a hashmap where you can store the database connection as the key and the user metadata as the value. However, this approach has one drawback. When the database connection has been cleaned up by some other part of the application, then you need to ensure the removal of the connection from the hashmap as well. If you forget to do such a thing, a reference to the connection will remain in the hashmap thereby preventing it from being garbage collected. This means that over a period of time, you are bound to end up with a very large hashmap, a clear indication of a memory leak. And the JVM will eventually spit out a OutOfMemoryError.

So, what do you do in such cases? Oh, of course, Weak referencnes to the rescue!

You can simply create a weak reference to the object, in the same way that we created a soft reference.

DBConnectionemp=newEmployee(); //1

WeakReference<DBConnection>weakRef=newWeakReference<DBConnection>(emp);

This creates a weak reference to the DBConnection object. This means that if at any future point in of time during the execution of the program, if the garbage collector detects that the only reference to the actual DBConnection object is a Weak reference, then it will mark the object for garbage collection.

Weak references are primarily used in conjuction with a WeakHashMap. This is a special kind of hashmap where the keys are all made of weak references. So, in our database example, we could effectively create a weak reference of the Database connection and store it in the WeakHashMap and the metadata of the user as the value in the hashmap. In this way, when the application no longer holds a strong reference to the Database connection, the only reference to the database connection object will be the one that we have via the WeakReference entry in the WeakHashMap. When the garbage collector detects this, it will mark the object for garbage collection. When the object is garbage collected, the entry from the WeakHashMap will be removed. And then, finally, we can all go home and rest in peace.

So, colloquially speaking, this is what the Soft reference and the Weak reference tell us about themselves.

**Soft Reference** : Hey! I am a soft reference. Ill take your shit as long as the JVM has patience. When it begins running out of patience(i.e. about to throw an OutOfMemoryError), i take no more. Your object will be gone. And then you simply have to create a new object.

**Weak Reference** : Hey! You know what. I am even cooler than the WeakReference. Coz I wont take your shit at all. The moment you lose me, if the JVM detects that you're no longer around, am gonna punch you in the face and run away! (i.e. Marked for garbage collection). Can you dig it sucka!

As you see, our two awesome friends, Softy and Weaky, certainly have some ego there. But they are pretty useful, at crucial times.

Before I proceed any further, there is one more puny lil thing that you might need to know. Oh! Did i just say 'might'. Oh no.. I meant, you should know. And that is ReferenceQueues. ReferenceQueues are some sort of a queue where the JVM can store objects of type reference once it has decided to take some action on the objects to which they refer.

What I mean to say is, let us suppose you have a weak reference which points to an object in the heap. And that object has been left lonely and desolate by the rest of the application.i.e. No strong references to it. And the garbage collector detects this object during its garbage collection cycle. Since this object only has a weak reference to it, the garbage collector will mark it for garbage collection. But at the same time, it looks if there is a reference queue associated with the weak reference that points to this object. If yes, it puts this weak reference in the reference queue to indicate that the object has been marked for garbage collection. The subtle point to be noted here is that even though the object has been marked for garbage collection, garbage collection may not have happened yet. This may be because the object has a finalize method, which the JVM needs to execute before reclaiming memory.

This also means that you can, but should not,unless deemed necessary, resurrect an object in the finalize method and create a strong reference to it. But when you do that, the weak reference still remains en-queued in the ReferenceQueue. Overriding the finalize method to resurrect an object is a rare case, but since it is one of the options that the JVM supports, it is therefore something that needs to be considered. Nevertheless, when you do such things, its almost equivalent to artificially manipulating the life of an object. That's because the second time when the object becomes eligible for garbage collection, the finalize method wont run, which is a good thing because if you run it again, its simply gonna revive itself. So, practically speaking, an object in the JVM has only one spare life. You get just one medical kit at the max. And thats it. You screw up again, and ur doomed. Your object will be on mars, having a boss fight with the garbage collecting thread of the JVM, which will eventually win, and reclaim the memory of the object.

The same facts about the reference queue hold true for Soft references as well.

In order to associate a weak or a soft reference with a reference queue, you can use the 2 argument constructor as shown below.

Employeeemp1=newEmployee();

Employeeemp2=newEmployee();

ReferenceQueuesoftQueue=newReferenceQueue();

ReferenceQueueweakQueue=newReferenceQueue();

SoftReferencesoftRef=newSoftReference(emp1, softQueue);

WeakReferencesoftRef=newWeakReference(emp2, weakQueue)

Now, aint that simple? Yes of course it is.

Then again, you haven't met the Phantom yet, have ya?

The phantom reference is a place where it gets all the more interesting.

***Phantom References***

Phantom references tell a long tale themselves and its a topic that warrants a blog post of its own. However, in this blog post, ill give a brief idea about what they are. Phantom references are quite similar to Strong and Weak references in the sense that the garbage collector will collect the referent object if the only reference to it is the phantom reference. But that's where the similarity ends.

What makes Phantom references are unique is the way in which they are used along with a reference queue. A phantom reference is always associated with a references queue during construction time. This is because phantom references are enqueued in the queue only when the the object is about to be garbage collected, after the finalize method(if any) has been executed on it. Calling a get() on the Phantom reference always returns null. And that is quite appropriate because the finalize function has already run on the referent object. So, there should be no 'legal' way of resurrecting the object (resurrecting i.e. creating a strong reference). This may at first seem to make no sense, since, what use is a phantom reference if we cant extract the referent from it? But on giving it a deep thought, you would realize that this is not the reason why phantom references are meant to be useful in the first place. It is the time at which the JVM puts a phantom reference in the reference queue that makes its use so intuitively amazing.

**Utill Concurrent**

Semaphore

A semaphore controls access to shared resources. A semaphore maintains a counter to specify the number of resources that the semaphore controls. Access to the resource is allowed if the counter is greater than zero,while a zero value of the counter indicates that no resource is available at the moment and so the access is denied. The methods acquire() and release() are for acquiring and releasing resources from a semaphore. If a thread calls acquire() and the counter is zero (i.e., resources are unavailable), the thread waits until the counter is non-zero and then gets the resource for use. Once the thread is done using the resource, it calls release() to increment the resource availability counter. Note if the number of resources is 1, then at a given time only one thread can access the resource; in this case, using the semaphore is similar to using a lock.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Semaphore(int permits) | Constructor to create Semaphore objects with a given number of permits(the number of threads that can access the resource at a time). If thepermit’s value is negative, the given number of release() calls musthappen before acquire() calls can succeed. |
| Semaphore(int permits,boolean fair) | Same as the previous constructor, but this extra fair option indicates that the permits should be allotted on a first-come-first-served basis. |
| void acquire()  void acquire(int permits) | Acquires a permit if available; otherwise, it blocks until a permit becomesavailable. Can throw an InterruptedException if some other threadinterrupts it while waiting to acquire a permit. The overloaded versiontakes a number of permits as an argument. |
| Void acquireUninterruptibly() | Same as the acquire() method, but this thread cannot be interruptedwhile waiting to acquire a permit. |
| boolean tryAcquire()  boolean tryAcquire(long timeout, TimeUnit unit) | Acquires a permit from the semaphore if available at the time of the call and returns true; if unavailable, it returns false immediately (without blocking). The overloaded tryAcquire() method additionally takes a time-out argument—the thread blocks to acquire a permit from the semaphore until a given time-out period. |
| void release()  void release(int permits) | Releases a permit from the semaphore. The overloaded version specifies the number of permits to release. |
|  |  |

Let’s assume that there are two ATM machines available in a ATM machine room. Therefore, only two people are allowed at a time in the room. There are five people waiting outside to use the ATM machines. The situation can be simulated by the code in Listing 14-1, in which each ATM machine is treated as a resource controlled by semaphore.

ATMRoom.java

**package** com.test.semaphore;

**import** java.util.concurrent.Semaphore;

/\*\*

\* Semaphore A semaphore controls access to shared resources. A semaphore

\* maintains a counter to specify the number of resources that the semaphore

\* controls. Access to the resource is allowed if the counter is greater than

\* zero,while a zero value of the counter indicates that no resource is

\* available at the moment and so the access is denied. The methods acquire()

\* and release() are for acquiring and releasing resources from a semaphore. If

\* a thread calls acquire() and the counter is zero (i.e., resources are

\* unavailable), the thread waits until the counter is non-zero and then gets

\* the resource for use. Once the thread is done using the resource, it calls

\* release() to increment the resource availability counter.

\*

\* **@author**rakeshku

\*

\*/

// This class simulates a situation where an ATM room has only two ATM machines

// and five people are waiting to access the machine. Since only one person can

// access

// an ATM machine at a given time, others wait for their turn

**class** ATMRoom {

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

// assume that only two ATM machines are available in the ATM room

Semaphore machines = **new** Semaphore(2);

// list of people waiting to access the machine

**new** Person(machines, "Mickey");

**new** Person(machines, "Donald");

**new** Person(machines, "Tom");

**new** Person(machines, "Jerry");

**new** Person(machines, "Casper");

}

}

// Each Person is an independent thread; but their access to the common resource

// (two ATM machines in the ATM machine room in this case) needs to be

// synchronized.

**class** Person **extends** Thread {

**private** Semaphore machines;

**public** Person(Semaphore machines, String name) {

**this**.machines = machines;

**this**.setName(name);

**this**.start();

}

**publicvoid** run() {

**try** {

System.*out*.println(getName() + " waiting to access an ATM machine");

machines.acquire();

System.*out*.println(getName() + " is accessing an ATM machine");

Thread.*sleep*(1000); // simulate the time required for withdrawing

// amount

System.*out*.println(getName() + " is done using the ATM machine");

machines.release();

} **catch** (InterruptedException ie) {

System.*err*.println(ie);

}

}

}

Here is the output of the program in one sample run:

Donald waiting to access an ATM machine

Donald is accessing an ATM machine

Mickey waiting to access an ATM machine

Jerry waiting to access an ATM machine

Tom waiting to access an ATM machine

Mickey is accessing an ATM machine

Casper waiting to access an ATM machine

Mickey is done using the ATM machine

Donald is done using the ATM machine

Jerry is accessing an ATM machine

Tom is accessing an ATM machine

Tom is done using the ATM machine

Jerry is done using the ATM machine

Casper is accessing an ATM machine

Casper is done using the ATM machine

Now let’s analyze how this program works. People waiting to access an ATM machine are simulated by creating a Person class that extends Thread. The run() method in the Thread class acquires a semaphore, simulates withdrawing money from the ATM machine, and releases the semaphore. The main() method simulates an ATM room with two ATM machines by creating a Semaphore object with two permits. People waiting in the queue to access the ATM machine are implemented by just adding them to the Semaphore object. As you can see from the program output, the semaphore allows only two threads at a time and the other threads keep waiting. When a thread releases the semaphore, another thread acquires it. Cool, isn’t it?

**CountDownLatch**

**CountDownLatch:**

This synchronizer allows one or more threads to wait for a countdown to complete. This countdown could be for a set of events to happen or until a set of operations being performed in other threads completes.

This can be useful in scenario when you have Oracle and MySQL and Application server and you want to ensure if the both DB is up then only start the application server, so you can write a utility that will check if Oracle has started the decrement the counter using countdown method then check if the MySQL is started if yes decrement the counter using countdown and if the counter is 0 then start the application server.

*Important Methods in the CountDownLatch Class*

|  |  |
| --- | --- |
| **Method** | **Description** |
| CountDownLatch(int count) | Creates an instance of CountDownLatch with the number of times thecountDown() method must be called before the threads waiting with await()can continue execution. |
| void await() | If the current count in CountDownLatch object is zero, it immediately returns; otherwise, the thread blocks until the countdown reaches zero. Can throw an InterruptedException. |
| boolean await(long timeout, TimeUnit unit) | Same as the previous method, await(), but takes an additional time-out argument. If the thread returns successfully after the count reaches zero, this method returns true; if the thread returns because of time-out, it returns false. |
| void countDown() | Reduces the number of counts by one in this CountDownLatch object. If the count reaches zero, all the (a)waiting threads are released. If the current count is already zero, nothing happens. |
| long getCount() | Returns the pending counts in this CountDownLatch object. |

***Listing 14-2.*** RunningRaceStarter.java

**package** com.test.countdownlatch;

**import** java.util.concurrent.\*;

// this class simulates the start of a running race by counting down from 5. It holds

// three runner threads to be ready to start in the start line of the race and once the count down

// reaches zero, all the three runners start running...

**class** RunningRaceStarter {

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

CountDownLatch counter = **new** CountDownLatch(5);

// count from 5 to 0 and then start the race

// instantiate three runner threads

**new** Runner(counter, "Carl");

**new** Runner(counter, "Joe");

**new** Runner(counter, "Jack");

System.*out*.println("Starting the countdown ");

**long** countVal = counter.getCount();

**while** (countVal > 0) {

Thread.*sleep*(1000); // 1000 milliseconds = 1 second

System.*out*.println(countVal);

**if** (countVal == 1) {

// once counter.countDown(); in the next statement is called,

// Count down will reach zero; so shout "Start"

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

}

counter.countDown(); // count down by 1 for each second

countVal = counter.getCount();

}

}

}

// this Runner class simulates a track runner in a 100-meter dash race. The

// runner waits until the

// count down timer gets to zero and then starts running

**class** Runner **extends** Thread {

**private** CountDownLatch timer;

**public** Runner(CountDownLatch cdl, String name) {

timer = cdl;

**this**.setName(name);

System.*out*.println(**this**.getName()

+ " ready and waiting for the count down to start");

start();

}

**publicvoid** run() {

**try** {

// wait for the timer count down to reach 0

timer.await();

} **catch** (InterruptedException ie) {

System.*err*.println("interrupted -- can't start running the race");

}

System.*out*.println(**this**.getName() + " started running");

}

}

This program prints the following:

Carl ready and waiting for the count down to start

Joe ready and waiting for the count down to start

Jack ready and waiting for the count down to start

Starting the countdown

5

4

3

2

1

Start

Joe started running

Carl started running

Jack started running

Let’s consider how the program works. The class Runner simulates a runner in a running race waiting to start running. It waits for the race to start by calling the await() method on the CountDownLatch object passed through the constructor. The RunningRaceStarter class creates a CountDownLatch object. This counter object is initialized with the count value 5, which means the countdown is from 5 to 0. In the main() method, you create Runner objects; these three threads wait on the counter object. For each second, you call the countDown() method, which decrements count by 1. Once the count reaches zero, all three waiting threads are released and they automatically continue execution.

Note: In this program, the sequence in which Joe, Carl, or Jack is printed cannot be predicted since it depends on thread scheduling. So, if you run this program, you may get these three names printed in some other order.

**Exchanger**

**Exchanger**

The Exchanger class is meant for exchanging data between two threads. What Exchanger does is something very simple: it waits until both the threads have called the exchange() method. When both threads have called the exchange() method, the Exchanger object actually exchanges the data shared by the threads with each other. This class is useful when two threads need to synchronize between them and continuously exchange data. This class is a tiny class with only one method: exchange(). Note that this exchange() method has an overloaded form where it takes a time-out period as an argument. Listing 14-3 shows an example simulating silly talk between the Java Duke mascot and the coffee shop. The two threads DukeThread and CoffeeShop threads run independently. However, for a chat to happen, they need to listen when the other is talking. An Exchange object provides a means for them to talk to each other.

KnockKnock.java

**import** java.util.concurrent.Exchanger;

// The DukeThread class runs as an independent thread. It talks to the CoffeeShopThread that

// also runs independently. The chat is achieved by exchanging messages through a common

// Exchanger<String> object that synchronizes the chat between them.

// Note that the message printed are the "responses" received from CoffeeShopThread

**class** DukeThread **extends** Thread {

**private** Exchanger<String>sillyTalk;

**public** DukeThread(Exchanger<String> args) {

sillyTalk = args;

}

**publicvoid** run() {

String reply = **null**;

**try** {

// start the conversation with CoffeeShopThread

reply = sillyTalk.exchange("Knock knock!");

// Now, print the response received from CoffeeShopThread

System.*out*.println("CoffeeShop: " + reply);

// exchange another set of messages

reply = sillyTalk.exchange("Duke");

// Now, print the response received from CoffeeShopThread

System.*out*.println("CoffeeShop: " + reply);

// an exchange could happen only when both send and receive happens

// since this is the last sentence to speak, we close the chat by

// ignoring the "dummy" reply

reply = sillyTalk

.exchange("The one who was born in this coffee shop!");

// talk over, so ignore the reply!

} **catch** (InterruptedException ie) {

System.*err*.println("Got interrupted during my silly talk");

}

}

}

**class** CoffeeShopThread **extends** Thread {

**private** Exchanger<String>sillyTalk;

**public** CoffeeShopThread(Exchanger<String> args) {

sillyTalk = args;

}

**publicvoid** run() {

String reply = **null**;

**try** {

// exchange the first messages

reply = sillyTalk.exchange("Who's there?");

// print what Duke said

System.*out*.println("Duke: " + reply);

// exchange second message

reply = sillyTalk.exchange("Duke who?");

// print what Duke said

System.*out*.println("Duke: " + reply);

// there is no message to send, but to get a message from Duke

// thread,

// both ends should send a message; so send a "dummy" string

reply = sillyTalk.exchange("");

System.*out*.println("Duke: " + reply);

} **catch** (InterruptedException ie) {

System.*err*.println("Got interrupted during my silly talk");

}

}

}

// Coordinate the silly talk between Duke and CoffeeShop by instantitaing the

// Exchanger object

// and the CoffeeShop and Duke threads

**class** KnockKnock {

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

Exchanger<String> sillyTalk = **new** Exchanger<String>();

**new** CoffeeShopThread(sillyTalk).start();

**new** DukeThread(sillyTalk).start();

}

}

The program prints the following:

Duke: Knock knock!

CoffeeShop: Who's there?

Duke: Duke

CoffeeShop: Duke who?

Duke: The one who was born in this coffee shop!

The comments inside the program explain how the program works. The main concept to understand with thisexample is that Exchanger helps coordinate (i.e., synchronize) exchanging messages between two threads. Both thethreads wait for each other and use the exchange() method to exchange messages.

**CyclicBarrier**

**CyclicBarrier**:

There are many situations in concurrent programming where threads may need to wait at a predefined executionpoint until all other threads reach that point. CyclicBarrier helps provide such a synchronization point;

|  |  |
| --- | --- |
| Method | Description |
| CyclicBarrier(int numThreads) | Creates a CyclicBarrier object with the number of threads waiting on it specified. Throws IllegalArgumentException if numThreads is negative or zero. |
| CyclicBarrier(int parties,Runnable barrierAction) | Same as the previous constructor; this constructor additionally takes the thread to call when the barrier is reached. |
| int await()  int await(long timeout,TimeUnit unit) | Blocks until the specified number of threads have called await()on this barrier. The method returns the arrival index of thisthread. This method can throw an InterruptedException if  the thread is interrupted while waiting for other threads or a  BrokenBarrierException if the barrier was broken for some reason(for example, another thread was timed-out or interrupted). Theoverloaded method takes a time-out period as an additional option;this overloaded version throws a TimeoutException if all otherthreads aren’t reached within the time-out period. |
| boolean isBroken() | Returns true if the barrier is broken. A barrier is broken if at least one thread in that barrier was interrupted or timed-out, or if a barrier action failed throwing an exception. |
| void reset() | Resets the barrier to the initial state. If there are any threads waiting on that barrier, they will throw the BrokenBarrier exception. |

***Listing 14-4.*** CyclicBarrierTest.java

**import** java.util.concurrent.\*;

// The run() method in this thread should be called only when four players are ready to start the game

**class** MixedDoubleTennisGame **extends** Thread {

**publicvoid** run() {

System.*out*

.println("All four players ready, game starts \n Love all...");

}

}

// This thread simulates arrival of a player.

// Once a player arrives, he/she should wait for other players to arrive

**class** Player **extends** Thread {

CyclicBarrier waitPoint;

**public** Player(CyclicBarrier barrier, String name) {

**this**.setName(name);

waitPoint = barrier;

**this**.start();

}

**publicvoid** run() {

System.*out*.println("Player " + getName() + " is ready ");

**try** {

waitPoint.await(); // await for all four players to arrive

} **catch** (BrokenBarrierException | InterruptedException exception) {

System.*out*.println("An exception occurred while waiting... "

+ exception);

}

}

}

// Creates a CyclicBarrier object by passing the number of threads and the

// thread to run

// when all the threads reach the barrier

**class** CyclicBarrierTest {

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

// a mixed-double tennis game requires four players; so wait for four

// players

// (i.e., four threads) to join to start the game

System.*out*

.println("Reserving tennis court \n As soon as four players arrive,game will start");

CyclicBarrier barrier = **new** CyclicBarrier(4,

**new** MixedDoubleTennisGame());

**new** Player(barrier, "G I Joe");

**new** Player(barrier, "Dora");

**new** Player(barrier, "Tintin");

**new** Player(barrier, "Barbie");

}

}

The program prints the following:

Reserving tennis court

As soon as four players arrive, game will start

Player G I Joe is ready

Player Dora is ready

Player Tintin is ready

Player Barbie is ready

All four players ready, game starts

Love all...

Now let’s see how this program works. In the main() method you create a CyclicBarrier object. The constructor takes two arguments: the number of threads to wait for, and the thread to invoke when all the threads reach the barrier. In this case, you have four players to wait for, so you create four threads, with each thread representing a player. The second argument for the CyclicBarrier constructor is the MixedDoubleTennisGame object since this thread represents the game, which will start once all four players are ready. Inside the run() method for each Player thread, you call the await() method on the CyclicBarrier object. Once the number of awaiting threads for the CyclicBarrier object reaches four, the run() method in MixedDoubleTennisGame is called.

**Phaser**

**Phaser:**

Phaser is a useful feature when few independent threads have to work in phases to complete a task. So, a synchronization point is needed for the threads to work on a part of a task, wait for others to complete other part of the task, and do a sync-up before advancing to complete the next part of the task.

|  |  |
| --- | --- |
| Method | Description |
| Phaser() | Creates a Phaser object with no registered parties and no parents. The initial phase is set to 0. |
| Phaser(int numThreads) | Creates a Phaser object with a given number of threads (parties) to arrive to advance to the next stage; the initial phase is set to 0. |
| int register() | Adds a new thread (party) to this Phaser object. Returns the phase  current number. Throws an IllegalStateException if the maximum  supported parties are already registered |
| int bulkRegister(int numThreads) | Adds numThreads of unarrived parties to this Phaser object. Returns thephase current number. Throws an IllegalStateException if maximumsupported parties are already registered. |
| int arrive() | Arrives at this phase without waiting for other threads to arrive. Returns the arrival phase number. Can throw an IllegalStateException. |
| int arriveAndDeregister() | Same as the previous method, but also deregisters from the Phaser object. |
| int arriveAndAwaitAdvance() | Arrive at this phase and waits (i.e., blocks) until other threads arrive. |
| int getRegisteredParties() | Returns the number of threads (parties) registered with this Phaser object. |
| int getArrivedParties() | Returns the number of threads (parties) arrived at the current phase ofthe Phaser object. |
| int getUnarrivedParties() | Returns the number of threads (parties) that have not arrived when compared to the registered parties at the current phase of the Phaser object. |
| int awaitAdvance(int phase) | Waits (i.e., blocks) until this Phaser object advances to the givenphase value. |

Consider the example of processing a delivery order in a small coffee shop. Assume that there are only three workers: a cook, a helper, and an attendant. To simplify the program logic, assume that each delivery order consists of three food items. Completing a delivery order consists of preparing the three orders one after another. To complete preparing a food item, all three workers—the cook, the helper, and the attendant—should do their part of the work.

Listing 14-5 shows how this situation can be implemented using the Phaser class.

**package** com.test.phaser;

**import** java.util.concurrent.Phaser;

// ProcessOrder thread is the master thread overlooking to make sure that the Cook, Helper,

// and Attendant are doing their part of the work to complete preparing the food items

// and complete order delivery

// To simplify the logic, we assume that each delivery order consists of exactly three food items

**publicclass** ProcessOrder {

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

// the Phaser is the synchronizer to make food items one-by-one,

// and deliver it before moving to the next item

Phaser deliveryOrder = **new** Phaser(1);

System.*out*.println("Starting to process the delivery order ");

**new** Worker(deliveryOrder, "Cook");

**new** Worker(deliveryOrder, "Helper");

**new** Worker(deliveryOrder, "Attendant");

**for** (**int** i = 1; i <= 3; i++) {

// Prepare, mix and deliver this food item

deliveryOrder.arriveAndAwaitAdvance();

System.*out*.println("Deliver food item no. " + i);

}

// work completed for this delivery order, so deregister

deliveryOrder.arriveAndDeregister();

System.*out*

.println("Delivery order completed... give it to the customer");

}

}

// The work could be a Cook, Helper, or Attendant. Though the three work

// independently, the

// should all synchronize their work together to do their part and complete

// preparing a food item

**class** Worker **extends** Thread {

Phaser deliveryOrder;

Worker(Phaser order, String name) {

deliveryOrder = order;

**this**.setName(name);

deliveryOrder.register();

start();

}

**publicvoid** run() {

**for** (**int** i = 1; i <= 3; i++) {

System.*out*.println("\t" + getName()

+ " doing his work for order no. " + i);

**if** (i == 3) {

// work completed for this delivery order, so deregister

deliveryOrder.arriveAndDeregister();

} **else** {

deliveryOrder.arriveAndAwaitAdvance();

}

**try** {

Thread.*sleep*(3000); // simulate time for preparing the food item

} **catch** (InterruptedException ie) {

/\* ignore exception \*/

ie.printStackTrace();

}

}

}

}

The program prints the following:

Starting to process the delivery order

Cook doing his work for order no. 1

Attendant doing his work for order no. 1

Helper doing his work for order no. 1

Deliver food item no. 1

Helper doing his work for order no. 2

Attendant doing his work for order no. 2

Cook doing his work for order no. 2

Deliver food item no. 2

Helper doing his work for order no. 3

Cook doing his work for order no. 3

Attendant doing his work for order no. 3

Deliver food item no. 3

Delivery order completed . . . give it to the customer

In this program, you create a Phaser object to support the synchronizing of three Worker thread objects. Youcreate a Phaser object by calling the default constructor of the Phaser object. When the Worker thread objects arecreated, they register themselves to the Phaser object. Alternatively, you could have called

Phaser deliveryOrder = new Phaser(3); // for three parties (i.e., threads)

In this case, you would not need to call the register() method on the Phaser object in the Worker thread

constructor. In this case, you’ve assumed that a delivery order consists of processing three food items, so the for loop runs three times. For each iteration, you call deliveryOrder.arriveAndAwaitAdvance(). For this statement to proceed, all the three parties (the Cook, Helper, and Attendant) have to complete their part of the work to prepare the food item. You simulate “preparing food” by calling the sleep() method in the run method for these Worker threads. These worker threads call deliveryOrder.arriveAndAwaitAdvance() for preparing each food item. As each food item is prepared (i.e., each phase is completed), the work progresses to the next phase. Once three phases are complete, the delivery order processing is complete and the program returns.

|  |  |
| --- | --- |
| Class/Interface | Short Description |
| BlockingQueue | This interface extends the Queue interface. In BlockingQueue, if the queue is empty,it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waitsfor an element to be removed from the queue. |
| ArrayBlockingQueue | This class provides a fixed-sized array based implementation of theBlockingQueue interface. |
| LinkedBlockingQueue | This class provides a linked-list-based implementation of the BlockingQueueinterface. |
| DelayQueue | This class implements BlockingQueue and consists of elements that are of typeDelayed. An element can be retrieved from this queue only after its delay period. |
| PriorityBlockingQueue | Equivalent to java.util.PriorityQueue, but implements the BlockingQueueinterface. |
| SynchronousQueue | This class implements BlockingQueue. In this container, each insert() by a threadwaits (blocks) for a corresponding remove() by another thread and vice versa. |
| LinkedBlockingDeque | This class implements BlockingDeque where insert and remove operations couldblock; uses a linked-list for implementation. |
| ConcurrentHashMap | Analogous to Hashtable, but with safe concurrent access and updates. |
| ConcurrentSkipListMap | Analogous to TreeMap, but provides safe concurrent access and updates. |
| ConcurrentSkipListSet | Analogous to TreeSet, but provides safe concurrent access and updates. |
| CopyOnWriteArrayList | Similar to ArrayList, but provides safe concurrent access. When the ArrayList isupdated, it creates a fresh copy of the underlying array. |
| CopyOnWriteArraySet | A Set implementation, but provides safe concurrent access and is implementedusing CopyOnWriteArrayList. When the container is updated, it creates a freshcopy of the underlying array. |

**Collections**

**ArrayList** : An ArrayList is the resizable-array implementation of the List interface.

1. ArrayList allows null element to be added.
2. It allows duplicate element to be added.
3. It is fail-fast which indicate that as soon as there is structural modification to the lis, it will fail.
4. It is roughly equivalent to Vector except that it is unsynchronized.
5. Each ArrayList instance has a “capacity”. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
6. ArrayList can be synchronized using Collections.synchronizedList(new ArrayList(...));
7. Iterator returned by ArrayList are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a {@link ConcurrentModificationException}. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.
8. ensureCapacity work like below:

int newCapacity = (oldCapacity 3)/2 + 1;

/\*

Replaces the element at the specified position in this list with

the specified element and return the old element.

**@param** index index of the element to replace

**@param** element element to be stored at the specified position

**@return** the element previously at the specified position

**@throws** IndexOutOfBoundsException {@inheritDoc}

\*/

**public** E set(**int** index, E element) {

RangeCheck(index);

E oldValue = (E) elementData[index];

elementData[index] = element;

**return** oldValue;

}

/\*\*

Appends the specified element to the end of this list.

**@param** e element to be appended to this list

**@return**<tt>true</tt> (as specified by {@link Collection#add})

\* /

**publicboolean** add(E e) {

ensureCapacity(size + 1); // Increments modCount!!

elementData[size++] = e;

**returntrue**;

}

/\*\*

Increases the capacity of this <tt>ArrayList</tt> instance, if

necessary, to ensure that it can hold at least the number of elements

specified by the minimum capacity argument.

**@param** minCapacity the desired minimum capacity\*\* /

**publicvoid**ensureCapacity(**int** minCapacity) {

modCount++;

**int** oldCapacity = elementData.length;

**if** (minCapacity > oldCapacity) {

Object oldData[] = elementData;

**int** newCapacity = (oldCapacity 3)/2 + 1;

**if** (newCapacity < minCapacity)

newCapacity = minCapacity;

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.*copyOf*(elementData, newCapacity);

}

}

/\*\*

Removes the first occurrence of the specified element from this list,

if it is present. If the list does not contain the element, it is

unchanged. More formally, removes the element with the lowest index

<tt>i</tt> such that

<tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))</tt>

(if such an element exists). Returns <tt>true</tt> if this list

contained the specified element (or equivalently, if this list

changed as a result of the call).

\*/

**publicboolean** remove(Object o) {

**if** (o == **null**) {

**for** (**int** index = 0; index <size; index++)

**if** (elementData[index] == **null**) {

fastRemove(index);

**returntrue**;

}

} **else** {

**for** (**int** index = 0; index <size; index++)

**if** (o.equals(elementData[index])) {

fastRemove(index);

**returntrue**;

}

}

**returnfalse**;

}

/\*\*

Appends all of the elements in the specified collection to the end of

this list, in the order that they are returned by the

specified collection's Iterator. The behavior of this operation is

undefined if the specified collection is modified while the operation

is in progress. (This implies that the behavior of this call is

undefined if the specified collection is this list, and this

list is nonempty.) \*\*/

**publicboolean** addAll(Collection<? **extends** E> c) {

Object[] a = c.toArray();

**int** numNew = a.length;

ensureCapacity(size + numNew); // Increments modCount

System.*arraycopy*(a, 0, elementData, size, numNew);

size += numNew;

**return** numNew != 0;

}

**Vector**

1. Vector allows null element to be added.
2. It allows duplicate element to be added.
3. Each Vector instance has a “capacity”. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an Vector, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
4. Vector is synchronized;
5. Enumeration returned by Vector’s is fails-safe.
6. The Iterators returned by Vector's iterator and listIterator methods are <em>fail-fast</em>: if the Vector is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException.
7. Vector size doubled when it is full. ensureCapacity work like below:

**privatevoid** ensureCapacityHelper(**int** minCapacity) {

**int** oldCapacity = elementData.length;

**if** (minCapacity > oldCapacity) {

Object[] oldData = elementData;

**int** newCapacity = (capacityIncrement> 0) ?

(oldCapacity + capacityIncrement) : (oldCapacity \* 2);

**if** (newCapacity < minCapacity) {

newCapacity = minCapacity;

}

elementData = Arrays.*copyOf*(elementData, newCapacity);

}

}

**Vector Demo:**

**package** com.test.collections;

**import** java.util.Enumeration;

**import** java.util.Vector;

**publicclass** VectorDemo {

**privatestatic** Vector<Emp>*empVector* = **new** Vector<Emp>();

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

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

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

*displayVectors*();

}

**publicstaticvoid** addEmp(Emp emp){

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

*empVector*.add(emp);

}

}

**publicstaticvoid** displayVectors(){

Enumeration<Emp> empEnum=*empVector*.elements();

**while**(empEnum.hasMoreElements()){

Emp emp=empEnum.nextElement();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**ArrayDeques**

**ArrayDeque**: Resizable-array implementation of the {@link Deque} interface. Its internal representation is as array.

1. Arraydeques have no capacity restrictions; they grow as necessary to supportusage.
2. They are not thread-safe; in the absence of externalsynchronization, they do not support concurrent access by multiple threads
3. Null elements are prohibited.
4. This class is likely to be faster than {@link Stack} when used as a stack, and faster than {@link LinkedList}
5. The iterators returned by this class's iterator method are

fail-fast: If the deque is modified at any time after the iteratoris created, in any way except through the iterator's own remove method, the iterator will generally throw a {@link ConcurrentModificationException}. Thus, in the face of concurrentmodification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

1. **Offer method**: is used to Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method.
2. **Poll method:** Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty
3. **Element method:** Retrieves, but does not remove, the head of the queue represented by this deque. This method differs from {@link #peek peek} only in that it throws an exception if this deque is empty. This method is equivalent to {@link #getFirst}.
4. **Peek method:** Retrieves, but does not remove, the head of the queue represented by this deque, or returns null if this deque is empty. This method is equivalent to {@link #peekFirst}.@return the head of the queue represented by this deque, or null if this deque is empty.
5. **Push method:** Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque. This method is equivalent to {@link #addFirst}
6. **Pop method:** Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque. This method is equivalent to {@link removeFirst()}.
7. **Delete method:** Removes the element at the specified position in the elements array, adjusting head and tail as necessary. This can result in motion of elements backwards or forwards in the array

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.Iterator;

**publicclass** ArraydequeDemo {

**privatestatic** Deque<Emp>*empDeque* = **new** ArrayDeque<Emp>();

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

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

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

Emp emp=*empDeque*.getFirst();

System.*out*.println("empId--"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

//Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method

*empDeque*.offer(emp);

System.*out*.println(*empDeque*.size());

/\*Retrieves and removes the head of the queue represented by this deque

(in other words, the first element of this deque), or returns null if this deque is empty

\*/

*empDeque*.poll();

//displayDeque();

//Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque

*empDeque*.push(emp);

//Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque

*empDeque*.poll();

}

**publicstaticvoid** addEmp(Emp emp){

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

*empDeque*.add(emp);

}

}

**publicstaticvoid** displayDeque(){

Iterator<Emp> it=*empDeque*.iterator();

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

Emp emp=it.next();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**LinkedList**

Linked list implementation of the List interface.

1. Allows null to be added.
2. Provides uniformly named methods to get remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, {@linkplain Queue queue}, or {@linkplain Deque double-ended queue}.
3. It is not thread safe.
4. Iterator returned by this class is fail-fast.
5. Internal representation as LinkList.
6. It implements the Dqueue interface, so all the method of Deque is already implemented.

**package** com.test.collections;

**import** java.util.Deque;

**import** java.util.Iterator;

**import**java.util.LinkedList;

**publicclass** LinkedListDemo {

**privatestatic** Deque<Emp>*empDeque* = **new**LinkedList<Emp>();

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

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

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

Emp emp=*empDeque*.getFirst();

System.*out*.println("empId--"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

//Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method

*empDeque*.offer(emp);

System.*out*.println(*empDeque*.size());

/\*Retrieves and removes the head of the queue represented by this deque

(in other words, the first element of this deque), or returns null if this deque is empty

\*/

*empDeque*.poll();

//displayDeque();

//Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque

*empDeque*.push(emp);

//Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque

*empDeque*.poll();

}

**publicstaticvoid** addEmp(Emp emp){

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

*empDeque*.add(emp);

}

}

**publicstaticvoid** displayDeque(){

Iterator<Emp> it=*empDeque*.iterator();

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

Emp emp=it.next();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**PriorityQueue**

PriorityQueue: An unbounded priority {@linkplain Queue queue} based on a priority heap.

1. The elements of the priority queue are ordered according to their {@linkplain Comparable natural ordering}, or by a {@link Comparator} provided at queue construction time, depending on which constructor is used.
2. A priority queue does not permit {@code null} elements.
3. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in {@code ClassCastException}).
4. The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily.
5. The queue retrieval operations {@code poll}, {@code remove}, {@code peek}, and {@code element} access the element at the head of the queue.
6. It is not synchronized.
7. Iterator returned by this class is fail-fast.

**EnumMap**

EnumMap is a specialized {@link Map} implementation for use with enum type keys. All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient.

1. Enum maps are maintained in the <i>natural order</i> of their keys (the order in which the enum constants are declared). This is reflected in the iterators returned by the collections views ({@link #keySet()}, {@link #entrySet()}, and {@link #values()}).
2. Iterators returned by the collection views are <i>weakly consistent</i>:

they will never throw {@link ConcurrentModificationException} and they may or may not show the effects of any modifications to the map that occur while the iteration is in progress.

1. Null keys are not permitted. Attempts to insert a null key will throw {@link NullPointerException}. Attempts to test for the presence of a null key or to remove one will, however, function properly.
2. Null values are permitted.
3. EnumMap is not synchronized. If multiple threads access an enum map concurrently, and at least one of the threads modifies the map, it should be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the enum map. If no such object exists, the map should be "wrapped" using the {@link Collections#synchronizedMap} method. This is best done at creation time, to prevent accidental unsynchronized access:
4. All basic operations execute in constant time. They are likely (though not guaranteed) to be faster than their {@link HashMap} counterparts.
5. For each loop cannot be used with EnumMap.

**package** com.test.collections;

**import** java.util.EnumMap;

**import** java.util.Iterator;

/

EnumMap is a specialized {@link Map} implementation for use with enum type

keys. All of the keys in an enum map must come from a single enum type that

is specified, explicitly or implicitly, when the map is created. Enum maps

are represented internally as arrays. This representation is extremely

compact and efficient.

i) Enum maps are maintained in the <i>natural

order</i> of their keys (the order in which the enum constants are declared).

This is reflected in the iterators returned by the collections views (

{@link #keySet()}, {@link #entrySet()}, and {@link #values()}).

ii) Iterators returned by the collection views are <i>weakly consistent</i>: they will

never throw {@link ConcurrentModificationException} and they may or may not

show the effects of any modifications to the map that occur while the

iteration is in progress.

iii) Null keys are not permitted. Attempts to insert a null key will throw

{@link NullPointerException}. Attempts to test for the presence of a null key

or to remove one will, however, function properly. iv) Null values are

permitted. v) EnumMap is not synchronized. If multiple threads access an enum

map concurrently, and at least one of the threads modifies the map, it should

be synchronized externally. This is typically accomplished by synchronizing

on some object that naturally encapsulates the enum map. If no such object

exists, the map should be "wrapped" using the

{@link Collections#synchronizedMap} method. This is best done at creation

time, to prevent accidental unsynchronized access: vi) All basic operations

execute in constant time. They are likely (though not guaranteed) to be

faster than their {@link HashMap} counterparts.

**@author**rakeshku

/

**publicclass** EnumMapDemo {

**publicenum** Number {

*ONE*, *TWO*, *THREE*

}

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

EnumMap<Number, String> em = **new** EnumMap<Number, String>(Number.**class**);

em.put(Number.*ONE*, "1");

em.put(Number.*TWO*, "2");

em.put(Number.*THREE*, "3");

Iterator<Number> it = em.keySet().iterator();

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

System.*out*.println("---" + it.next());

}

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

System.*out*.println("---" + it.next());

}

}

}

**package** java.util;

**import** java.util.Map.Entry;

**import** sun.misc.SharedSecrets;

/

A specialized {@link Map} implementation for use with enum type keys. All

of the keys in an enum map must come from a single enum type that is

specified, explicitly or implicitly, when the map is created. Enum maps

are represented internally as arrays. This representation is extremely

compact and efficient.

<p>Enum maps are maintained in the <i>natural order</i> of their keys

(the order in which the enum constants are declared). This is reflected

in the iterators returned by the collections views ({@link #keySet()},

{@link #entrySet()}, and {@link #values()}).

<p>Iterators returned by the collection views are <i>weakly consistent</i>:

they will never throw {@link ConcurrentModificationException} and they may

or may not show the effects of any modifications to the map that occur while

the iteration is in progress.

<p>Null keys are not permitted. Attempts to insert a null key will

throw {@link NullPointerException}. Attempts to test for the

presence of a null key or to remove one will, however, function properly.

Null values are permitted.

<P>Like most collection implementations <tt>EnumMap</tt> is not

synchronized. If multiple threads access an enum map concurrently, and at

least one of the threads modifies the map, it should be synchronized

externally. This is typically accomplished by synchronizing on some

object that naturally encapsulates the enum map. If no such object exists,

the map should be "wrapped" using the {@link Collections#synchronizedMap}

method. This is best done at creation time, to prevent accidental

unsynchronized access:

<pre>

Map&lt;EnumKey, V&gt; m

= Collections.synchronizedMap(new EnumMap&lt;EnumKey, V&gt;(...));

</pre>

<p>Implementation note: All basic operations execute in constant time.

They are likely (though not guaranteed) to be faster than their

{@link HashMap} counterparts.

<p>This class is a member of the

<a href="{@docRoot}/../technotes/guides/collections/index.html">

Java Collections Framework</a>.

**@author** Josh Bloch

**@version** %I%, %G%

**@see** EnumSet

**@since** 1.5

/

**publicclass** EnumMap<K **extends** Enum<K>, V>**extends** AbstractMap<K, V>

**implements** java.io.Serializable, Cloneable

{

/

The <tt>Class</tt> object for the enum type of all the keys of this map.

**@serial**

/

**privatefinal** Class<K>keyType;

/

All of the values comprising K. (Cached for performance.)

/

**privatetransient** K[] keyUniverse;

/

Array representation of this map. The ith element is the value

to which universe[i] is currently mapped, or null if it isn't

mapped to anything, or NULL if it's mapped to null.

/

**privatetransient** Object[] vals;

/

The number of mappings in this map.

/

**privatetransientint**size = 0;

/

Distinguished non-null value for representing null values.

/

**privatestaticfinal** Object *NULL* = **new** Object();

**private** Object maskNull(Object value) {

**return** (value == **null** ? *NULL* : value);

}

**private** V unmaskNull(Object value) {

**return** (V) (value == *NULL* ? **null** : value);

}

**privatestatic** Enum[] *ZERO\_LENGTH\_ENUM\_ARRAY* = **new** Enum[0];

/

Creates an empty enum map with the specified key type.

**@param** keyType the class object of the key type for this enum map

**@throws** NullPointerException if <tt>keyType</tt> is null

/

**public** EnumMap(Class<K> keyType) {

**this**.keyType = keyType;

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

}

/

Creates an enum map with the same key type as the specified enum

map, initially containing the same mappings (if any).

**@param** m the enum map from which to initialize this enum map

**@throws** NullPointerException if <tt>m</tt> is null

/

**public** EnumMap(EnumMap<K, ? **extends** V> m) {

keyType = m.keyType;

keyUniverse = m.keyUniverse;

vals = (Object[]) m.vals.clone();

size = m.size;

}

/

Creates an enum map initialized from the specified map. If the

specified map is an <tt>EnumMap</tt> instance, this constructor behaves

identically to {@link #EnumMap(EnumMap)}. Otherwise, the specified map

must contain at least one mapping (in order to determine the new

enum map's key type).

**@param** m the map from which to initialize this enum map

**@throws** IllegalArgumentException if <tt>m</tt> is not an

<tt>EnumMap</tt> instance and contains no mappings

**@throws** NullPointerException if <tt>m</tt> is null

/

**public** EnumMap(Map<K, ? **extends** V> m) {

**if** (m **instanceof** EnumMap) {

EnumMap<K, ? **extends** V> em = (EnumMap<K, ? **extends** V>) m;

keyType = em.keyType;

keyUniverse = em.keyUniverse;

vals = (Object[]) em.vals.clone();

size = em.size;

} **else** {

**if** (m.isEmpty())

**thrownew** IllegalArgumentException("Specified map is empty");

keyType = m.keySet().iterator().next().getDeclaringClass();

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

putAll(m);

}

}

// Query Operations

/

Returns the number of key-value mappings in this map.

**@return** the number of key-value mappings in this map

/

**publicint** size() {

**return**size;

}

/

Returns <tt>true</tt> if this map maps one or more keys to the

specified value.

**@param** value the value whose presence in this map is to be tested

**@return**<tt>true</tt> if this map maps one or more keys to this value

/

**publicboolean** containsValue(Object value) {

value = maskNull(value);

**for** (Object val : vals)

**if** (value.equals(val))

**returntrue**;

**returnfalse**;

}

/

Returns <tt>true</tt> if this map contains a mapping for the specified

key.

**@param** key the key whose presence in this map is to be tested

**@return**<tt>true</tt> if this map contains a mapping for the specified

key

/

**publicboolean** containsKey(Object key) {

**return** isValidKey(key) &&vals[((Enum)key).ordinal()] != **null**;

}

**privateboolean** containsMapping(Object key, Object value) {

**return** isValidKey(key) &&

maskNull(value).equals(vals[((Enum)key).ordinal()]);

}

/

Returns the value to which the specified key is mapped,

or {@code null} if this map contains no mapping for the key.

<p>More formally, if this map contains a mapping from a key

{@code k} to a value {@code v} such that {@code (key == k)},

then this method returns {@code v}; otherwise it returns

{@code null}. (There can be at most one such mapping.)

<p>A return value of {@code null} does not <i>necessarily</i>

indicate that the map contains no mapping for the key; it's also

possible that the map explicitly maps the key to {@code null}.

The {@link #containsKey containsKey} operation may be used to

distinguish these two cases.

/

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

**return** (isValidKey(key) ?

unmaskNull(vals[((Enum)key).ordinal()]) : **null**);

}

// Modification Operations

/

Associates the specified value with the specified key in this map.

If the map previously contained a mapping for this key, the old

value is replaced.

**@param** key the key with which the specified value is to be associated

**@param** value the value to be associated with the specified key

**@return** the previous value associated with specified key, or

<tt>null</tt> if there was no mapping for key. (A <tt>null</tt>

return can also indicate that the map previously associated

<tt>null</tt> with the specified key.)

**@throws** NullPointerException if the specified key is null

/

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

typeCheck(key);

**int** index = ((Enum)key).ordinal();

Object oldValue = vals[index];

vals[index] = maskNull(value);

**if** (oldValue == **null**)

size++;

**return** unmaskNull(oldValue);

}

/

Removes the mapping for this key from this map if present.

**@param** key the key whose mapping is to be removed from the map

**@return** the previous value associated with specified key, or

<tt>null</tt> if there was no entry for key. (A <tt>null</tt>

return can also indicate that the map previously associated

<tt>null</tt> with the specified key.)

/

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

**if** (!isValidKey(key))

**returnnull**;

**int** index = ((Enum)key).ordinal();

Object oldValue = vals[index];

vals[index] = **null**;

**if** (oldValue != **null**)

size--;

**return** unmaskNull(oldValue);

}

**privateboolean** removeMapping(Object key, Object value) {

**if** (!isValidKey(key))

**returnfalse**;

**int** index = ((Enum)key).ordinal();

**if** (maskNull(value).equals(vals[index])) {

vals[index] = **null**;

size--;

**returntrue**;

}

**returnfalse**;

}

/

Returns true if key is of the proper type to be a key in this

enum map.

/

**privateboolean** isValidKey(Object key) {

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

**returnfalse**;

// Cheaper than instanceof Enum followed by getDeclaringClass

Class keyClass = key.getClass();

**return** keyClass == keyType || keyClass.getSuperclass() == keyType;

}

// Bulk Operations

/

Copies all of the mappings from the specified map to this map.

These mappings will replace any mappings that this map had for

any of the keys currently in the specified map.

**@param** m the mappings to be stored in this map

**@throws** NullPointerException the specified map is null, or if

one or more keys in the specified map are null

/

**publicvoid** putAll(Map<? **extends** K, ? **extends** V> m) {

**if** (m **instanceof** EnumMap) {

EnumMap<? **extends** K, ? **extends** V> em =

(EnumMap<? **extends** K, ? **extends** V>)m;

**if** (em.keyType != keyType) {

**if** (em.isEmpty())

**return**;

**thrownew** ClassCastException(em.keyType + " != " + keyType);

}

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

Object emValue = em.vals[i];

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

**if** (vals[i] == **null**)

size++;

vals[i] = emValue;

}

}

} **else** {

**super**.putAll(m);

}

}

/

Removes all mappings from this map.

/

**publicvoid** clear() {

Arrays.*fill*(vals, **null**);

size = 0;

}

// Views

/

This field is initialized to contain an instance of the entry set

view the first time this view is requested. The view is stateless,

so there's no reason to create more than one.

/

**privatetransient** Set<Map.Entry<K,V>>entrySet = **null**;

/

Returns a {@link Set} view of the keys contained in this map.

The returned set obeys the general contract outlined in

{@link Map#keySet()}. The set's iterator will return the keys

in their natural order (the order in which the enum constants

are declared).

**@return** a set view of the keys contained in this enum map

/

**public** Set<K> keySet() {

Set<K> ks = keySet;

**if** (ks != **null**)

**return** ks;

**else**

**return**keySet = **new** KeySet();

}

**privateclass** KeySet **extends** AbstractSet<K> {

**public** Iterator<K> iterator() {

**returnnew** KeyIterator();

}

**publicint** size() {

**return**size;

}

**publicboolean** contains(Object o) {

**return** containsKey(o);

}

**publicboolean** remove(Object o) {

**int** oldSize = size;

EnumMap.**this**.remove(o);

**return**size != oldSize;

}

**publicvoid** clear() {

EnumMap.**this**.clear();

}

}

/

Returns a {@link Collection} view of the values contained in this map.

The returned collection obeys the general contract outlined in

{@link Map#values()}. The collection's iterator will return the

values in the order their corresponding keys appear in map,

which is their natural order (the order in which the enum constants

are declared).

**@return** a collection view of the values contained in this map

/

**public** Collection<V> values() {

Collection<V> vs = values;

**if** (vs != **null**)

**return** vs;

**else**

**return**values = **new** Values();

}

**privateclass** Values **extends** AbstractCollection<V> {

**public** Iterator<V> iterator() {

**returnnew** ValueIterator();

}

**publicint** size() {

**return**size;

}

**publicboolean** contains(Object o) {

**return** containsValue(o);

}

**publicboolean** remove(Object o) {

o = maskNull(o);

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

**if** (o.equals(vals[i])) {

vals[i] = **null**;

size--;

**returntrue**;

}

}

**returnfalse**;

}

**publicvoid** clear() {

EnumMap.**this**.clear();

}

}

/

Returns a {@link Set} view of the mappings contained in this map.

The returned set obeys the general contract outlined in

{@link Map#keySet()}. The set's iterator will return the

mappings in the order their keys appear in map, which is their

natural order (the order in which the enum constants are declared).

**@return** a set view of the mappings contained in this enum map

/

**public** Set<Map.Entry<K,V>> entrySet() {

Set<Map.Entry<K,V>> es = entrySet;

**if** (es != **null**)

**return** es;

**else**

**return**entrySet = **new** EntrySet();

}

**privateclass** EntrySet **extends** AbstractSet<Map.Entry<K,V>> {

**public** Iterator<Map.Entry<K,V>> iterator() {

**returnnew** EntryIterator();

}

**publicboolean** contains(Object o) {

**if** (!(o **instanceof** Map.Entry))

**returnfalse**;

Map.Entry entry = (Map.Entry)o;

**return** containsMapping(entry.getKey(), entry.getValue());

}

**publicboolean** remove(Object o) {

**if** (!(o **instanceof** Map.Entry))

**returnfalse**;

Map.Entry entry = (Map.Entry)o;

**return** removeMapping(entry.getKey(), entry.getValue());

}

**publicint** size() {

**return**size;

}

**publicvoid** clear() {

EnumMap.**this**.clear();

}

**public** Object[] toArray() {

**return** fillEntryArray(**new** Object[size]);

}

@SuppressWarnings("unchecked")

**public**<T> T[] toArray(T[] a) {

**int** size = size();

**if** (a.length< size)

a = (T[])java.lang.reflect.Array

.*newInstance*(a.getClass().getComponentType(), size);

**if** (a.length> size)

a[size] = **null**;

**return** (T[]) fillEntryArray(a);

}

**private** Object[] fillEntryArray(Object[] a) {

**int** j = 0;

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

**if** (vals[i] != **null**)

a[j++] = **new** AbstractMap.SimpleEntry<K,V>(

keyUniverse[i], unmaskNull(vals[i]));

**return** a;

}

}

**privateabstractclass** EnumMapIterator<T>**implements** Iterator<T> {

// Lower bound on index of next element to return

**int**index = 0;

// Index of last returned element, or -1 if none

**int**lastReturnedIndex = -1;

**publicboolean** hasNext() {

**while** (index<vals.length&&vals[index] == **null**)

index++;

**return**index != vals.length;

}

**publicvoid** remove() {

checkLastReturnedIndex();

**if** (vals[lastReturnedIndex] != **null**) {

vals[lastReturnedIndex] = **null**;

size--;

}

lastReturnedIndex = -1;

}

**privatevoid** checkLastReturnedIndex() {

**if** (lastReturnedIndex< 0)

**thrownew** IllegalStateException();

}

}

**privateclass** KeyIterator **extends** EnumMapIterator<K> {

**public** K next() {

**if** (!hasNext())

**thrownew** NoSuchElementException();

lastReturnedIndex = index++;

**return**keyUniverse[lastReturnedIndex];

}

}

**privateclass** ValueIterator **extends** EnumMapIterator<V> {

**public** V next() {

**if** (!hasNext())

**thrownew** NoSuchElementException();

lastReturnedIndex = index++;

**return** unmaskNull(vals[lastReturnedIndex]);

}

}

/

Since we don't use Entry objects, we use the Iterator itself as entry.

/

**privateclass** EntryIterator **extends** EnumMapIterator<Map.Entry<K,V>>

**implements** Map.Entry<K,V>

{

**public** Map.Entry<K,V> next() {

**if** (!hasNext())

**thrownew** NoSuchElementException();

lastReturnedIndex = index++;

**returnthis**;

}

**public** K getKey() {

checkLastReturnedIndexForEntryUse();

**return**keyUniverse[lastReturnedIndex];

}

**public** V getValue() {

checkLastReturnedIndexForEntryUse();

**return** unmaskNull(vals[lastReturnedIndex]);

}

**public** V setValue(V value) {

checkLastReturnedIndexForEntryUse();

V oldValue = unmaskNull(vals[lastReturnedIndex]);

vals[lastReturnedIndex] = maskNull(value);

**return** oldValue;

}

**publicboolean** equals(Object o) {

**if** (lastReturnedIndex< 0)

**return** o == **this**;

**if** (!(o **instanceof** Map.Entry))

**returnfalse**;

Map.Entry e = (Map.Entry)o;

V ourValue = unmaskNull(vals[lastReturnedIndex]);

Object hisValue = e.getValue();

**return** e.getKey() == keyUniverse[lastReturnedIndex] &&

(ourValue == hisValue ||

(ourValue != **null**&& ourValue.equals(hisValue)));

}

**publicint** hashCode() {

**if** (lastReturnedIndex< 0)

**returnsuper**.hashCode();

Object value = vals[lastReturnedIndex];

**return**keyUniverse[lastReturnedIndex].hashCode()

^ (value == *NULL* ? 0 : value.hashCode());

}

**public** String toString() {

**if** (lastReturnedIndex< 0)

**returnsuper**.toString();

**return**keyUniverse[lastReturnedIndex] + "="

+ unmaskNull(vals[lastReturnedIndex]);

}

**privatevoid** checkLastReturnedIndexForEntryUse() {

**if** (lastReturnedIndex< 0)

**thrownew** IllegalStateException("Entry was removed");

}

}

// Comparison and hashing

/

Compares the specified object with this map for equality. Returns

<tt>true</tt> if the given object is also a map and the two maps

represent the same mappings, as specified in the {@link

Map#equals(Object)} contract.

**@param** o the object to be compared for equality with this map

**@return**<tt>true</tt> if the specified object is equal to this map

/

**publicboolean** equals(Object o) {

**if** (!(o **instanceof** EnumMap))

**returnsuper**.equals(o);

EnumMap em = (EnumMap)o;

**if** (em.keyType != keyType)

**return**size == 0 && em.size == 0;

// Key types match, compare each value

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

Object ourValue = vals[i];

Object hisValue = em.vals[i];

**if** (hisValue != ourValue &&

(hisValue == **null** || !hisValue.equals(ourValue)))

**returnfalse**;

}

**returntrue**;

}

/

Returns a shallow copy of this enum map. (The values themselves

are not cloned.

**@return** a shallow copy of this enum map

/

**public** EnumMap<K, V> clone() {

EnumMap<K, V> result = **null**;

**try** {

result = (EnumMap<K, V>) **super**.clone();

} **catch**(CloneNotSupportedException e) {

**thrownew** AssertionError();

}

result.vals = (Object[]) result.vals.clone();

**return** result;

}

/

Throws an exception if e is not of the correct type for this enum set.

/

**privatevoid** typeCheck(K key) {

Class keyClass = key.getClass();

**if** (keyClass != keyType&& keyClass.getSuperclass() != keyType)

**thrownew** ClassCastException(keyClass + " != " + keyType);

}

/

Returns all of the values comprising K.

The result is uncloned, cached, and shared by all callers.

/

**privatestatic**<K **extends** Enum<K>> K[] getKeyUniverse(Class<K> keyType) {

**return** SharedSecrets.*getJavaLangAccess*()

.getEnumConstantsShared(keyType);

}

**privatestaticfinallong***serialVersionUID* = 458661240069192865L;

/

Save the state of the <tt>EnumMap</tt> instance to a stream (i.e.,

serialize it).

**@serialData** The <i>size</i> of the enum map (the number of key-value

mappings) is emitted (int), followed by the key (Object)

and value (Object) for each key-value mapping represented

by the enum map.

/

**privatevoid** writeObject(java.io.ObjectOutputStream s)

**throws** java.io.IOException

{

// Write out the key type and any hidden stuff

s.defaultWriteObject();

// Write out size (number of Mappings)

s.writeInt(size);

// Write out keys and values (alternating)

**for** (Map.Entry<K,V> e : entrySet()) {

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

/

Reconstitute the <tt>EnumMap</tt> instance from a stream (i.e.,

deserialize it).

/

**privatevoid** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException

{

// Read in the key type and any hidden stuff

s.defaultReadObject();

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

// Read in size (number of Mappings)

**int** size = s.readInt();

// Read the keys and values, and put the mappings in the HashMap

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

K key = (K) s.readObject();

V value = (V) s.readObject();

put(key, value);

}

}

}

**EnumSet**

EnumSet is a specialized {@link Set} implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors. This representation is extremely compact and efficient. The space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based "bit flags." Even bulk operations (such as containsAll and retainAll) should run very quickly if their argument is also an enum set.

1. The iterator returned by the iterator method traverses the elements in their <i>natural order</i> (the order in which the enum constants are declared). The returned iterator is <i>weakly consistent</i>: it will never throw {@link ConcurrentModificationException} and it may or may not show the effects of any modifications to the set that occur while the iteration is in progress
2. Null elements are not permitted. Attempts to insert a null element will throw {@link NullPointerException}. Attempts to test for the presence of a null element or to remove one will, however, function properly.
3. EnumSet is not synchronized.
4. All basic operations execute in constant time. They are likely (though not guaranteed) to be faster than their {@link HashSet} counterparts.

**publicabstractclass** EnumSet<E **extends** Enum<E>>**extends** AbstractSet<E>

**implements** Cloneable, java.io.Serializable

{

/\*\*

\* The class of all the elements of this set.

\*/

**final** Class<E>elementType;

/\*\*

\* All of the values comprising T. (Cached for performance.)

\*/

**final** Enum[] universe;

**privatestatic** Enum[] *ZERO\_LENGTH\_ENUM\_ARRAY* = **new** Enum[0];

EnumSet(Class<E>elementType, Enum[] universe) {

**this**.elementType = elementType;

**this**.universe = universe;

}

/\*\*

\* Creates an empty enum set with the specified element type.

\*

\* **@param** elementType the class object of the element type for this enum

\* set

\* **@throws** NullPointerException if <tt>elementType</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {

Enum[] universe = *getUniverse*(elementType);

**if** (universe == **null**)

**thrownew** ClassCastException(elementType + " not an enum");

**if** (universe.length<= 64)

**returnnew** RegularEnumSet<E>(elementType, universe);

**else**

**returnnew** JumboEnumSet<E>(elementType, universe);

}

/\*\*

\* Creates an enum set containing all of the elements in the specified

\* element type.

\*

\* **@param** elementType the class object of the element type for this enum

\* set

\* **@throws** NullPointerException if <tt>elementType</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> allOf(Class<E> elementType) {

EnumSet<E> result = *noneOf*(elementType);

result.addAll();

**return** result;

}

/\*\*

\* Adds all of the elements from the appropriate enum type to this enum

\* set, which is empty prior to the call.

\*/

**abstractvoid** addAll();

/\*\*

\* Creates an enum set with the same element type as the specified enum

\* set, initially containing the same elements (if any).

\*

\* **@param** s the enum set from which to initialize this enum set

\* **@throws** NullPointerException if <tt>s</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> copyOf(EnumSet<E> s) {

**return** s.clone();

}

/\*\*

\* Creates an enum set initialized from the specified collection. If

\* the specified collection is an <tt>EnumSet</tt> instance, this static

\* factory method behaves identically to {@link #copyOf(EnumSet)}.

\* Otherwise, the specified collection must contain at least one element

\* (in order to determine the new enum set's element type).

\*

\* **@param** c the collection from which to initialize this enum set

\* **@throws** IllegalArgumentException if <tt>c</tt> is not an

\* <tt>EnumSet</tt> instance and contains no elements

\* **@throws** NullPointerException if <tt>c</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> copyOf(Collection<E> c) {

**if** (c **instanceof** EnumSet) {

**return** ((EnumSet<E>)c).clone();

} **else** {

**if** (c.isEmpty())

**thrownew** IllegalArgumentException("Collection is empty");

Iterator<E> i = c.iterator();

E first = i.next();

EnumSet<E> result = EnumSet.*of*(first);

**while** (i.hasNext())

result.add(i.next());

**return** result;

}

}

/\*\*

\* Creates an enum set with the same element type as the specified enum

\* set, initially containing all the elements of this type that are

\* <i>not</i> contained in the specified set.

\*

\* **@param** s the enum set from whose complement to initialize this enum set

\* **@throws** NullPointerException if <tt>s</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> complementOf(EnumSet<E> s) {

EnumSet<E> result = *copyOf*(s);

result.complement();

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified element.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e the element that this set is to contain initially

\* **@throws** NullPointerException if <tt>e</tt> is null

\* **@return** an enum set initially containing the specified element

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E e) {

EnumSet<E> result = *noneOf*(e.getDeclaringClass());

result.add(e);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E e1, E e2) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@param** e4 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

result.add(e4);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@param** e4 another element that this set is to contain initially

\* **@param** e5 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4,

E e5)

{

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

result.add(e4);

result.add(e5);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\* This factory, whose parameter list uses the varargs feature, may

\* be used to create an enum set initially containing an arbitrary

\* number of elements, but it is likely to run slower than the overloadings

\* that do not use varargs.

\*

\* **@param** first an element that the set is to contain initially

\* **@param** rest the remaining elements the set is to contain initially

\* **@throws** NullPointerException if any of the specified elements are null,

\* or if <tt>rest</tt> is null

\* **@return** an enum set initially containing the specified elements

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> of(E first, E... rest) {

EnumSet<E> result = *noneOf*(first.getDeclaringClass());

result.add(first);

**for** (E e : rest)

result.add(e);

**return** result;

}

/\*\*

\* Creates an enum set initially containing all of the elements in the

\* range defined by the two specified endpoints. The returned set will

\* contain the endpoints themselves, which may be identical but must not

\* be out of order.

\*

\* **@param** from the first element in the range

\* **@param** to the last element in the range

\* **@throws** NullPointerException if <tt>first</tt> or <tt>last</tt> are

\* null

\* **@throws** IllegalArgumentException if <tt>first.compareTo(last) &gt; 0</tt>

\* **@return** an enum set initially containing all of the elements in the

\* range defined by the two specified endpoints

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E> range(E from, E to) {

**if** (from.compareTo(to) > 0)

**thrownew** IllegalArgumentException(from + " > " + to);

EnumSet<E> result = *noneOf*(from.getDeclaringClass());

result.addRange(from, to);

**return** result;

}

/\*\*

\* Adds the specified range to this enum set, which is empty prior

\* to the call.

\*/

**abstractvoid** addRange(E from, E to);

/\*\*

\* Returns a copy of this set.

\*

\* **@return** a copy of this set

\*/

**public** EnumSet<E> clone() {

**try** {

**return** (EnumSet<E>) **super**.clone();

} **catch**(CloneNotSupportedException e) {

**thrownew** AssertionError(e);

}

}

/\*\*

\* Complements the contents of this enum set.

\*/

**abstractvoid** complement();

/\*\*

\* Throws an exception if e is not of the correct type for this enum set.

\*/

**finalvoid** typeCheck(E e) {

Class eClass = e.getClass();

**if** (eClass != elementType&& eClass.getSuperclass() != elementType)

**thrownew** ClassCastException(eClass + " != " + elementType);

}

/\*\*

\* Returns all of the values comprising E.

\* The result is uncloned, cached, and shared by all callers.

\*/

**privatestatic**<E **extends** Enum<E>> E[] getUniverse(Class<E> elementType) {

**return** SharedSecrets.*getJavaLangAccess*()

.getEnumConstantsShared(elementType);

}

/\*\*

\* This class is used to serialize all EnumSet instances, regardless of

\* implementation type. It captures their "logical contents" and they

\* are reconstructed using public static factories. This is necessary

\* to ensure that the existence of a particular implementation type is

\* an implementation detail.

\*

\* **@serial** include

\*/

**privatestaticclass** SerializationProxy <E **extends** Enum<E>>

**implements** java.io.Serializable

{

/\*\*

\* The element type of this enum set.

\*

\* **@serial**

\*/

**privatefinal** Class<E>elementType;

/\*\*

\* The elements contained in this enum set.

\*

\* **@serial**

\*/

**privatefinal** Enum[] elements;

SerializationProxy(EnumSet<E> set) {

elementType = set.elementType;

elements = (Enum[]) set.toArray(*ZERO\_LENGTH\_ENUM\_ARRAY*);

}

**private** Object readResolve() {

EnumSet<E> result = EnumSet.*noneOf*(elementType);

**for** (Enum e : elements)

result.add((E)e);

**return** result;

}

**privatestaticfinallong***serialVersionUID* = 362491234563181265L;

}

Object writeReplace() {

**returnnew** SerializationProxy<E>(**this**);

}

}

**BlockingQueue**

## BlockingQueue Usage

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:



**A BlockingQueue with one thread putting into it, and another thread taking from it.**

The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

### BlockingQueue Methods

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

The 4 different sets of behavior means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immediately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a **BlockingQueue**. If you try to insert null, the **BlockingQueue** will throw aNullPointerException.

It is also possible to access all the elements inside a **BlockingQueue**, and not just the elements at the start and end. For instance, say you have queued an object for processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

## BlockingQueue Implementations

Since **BlockingQueue** is an interface, you need to use one of its implementations to use it. Thejava.util.concurrent package has the following implementations of the **BlockingQueue** interface (in Java 6):

* [**ArrayBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html)
* [**DelayQueue**](http://tutorials.jenkov.com/java-util-concurrent/delayqueue.html)
* [**LinkedBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html)
* [**PriorityBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/priorityblockingqueue.html)
* [**SynchronousQueue**](http://tutorials.jenkov.com/java-util-concurrent/synchronousqueue.html)

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

## Java BlockingQueue Example

Here is a Java **BlockingQueue** example. The example uses the **ArrayBlockingQueue** implementation of the**BlockingQueue** interface.

First, the **BlockingQueueExample** class which starts a Producer and a Consumer in separate threads. TheProducer inserts strings into a shared **BlockingQueue**, and the Consumer takes them out.

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.BlockingQueue;

**publicclass** BlockingQueueExample {

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

BlockingQueue<String> queue = **new** ArrayBlockingQueue<String>(1024);

Producer producer = **new** Producer(queue);

Consumer consumer = **new** Consumer(queue);

**new** Thread(producer).start();

**new** Thread(consumer).start();

Thread.*sleep*(4000);

}

}

**package** com.test.blocking.queue.example;

**import**java.util.concurrent.BlockingQueue;

**publicclass** Consumer **implements** Runnable{

**protected**BlockingQueue<String>queue = **null**;

**public** Consumer(BlockingQueue<String> queue) {

**this**.queue = queue;

}

**publicvoid** run() {

**try** {

System.*out*.println(queue.take());

System.*out*.println(queue.take());

System.*out*.println(queue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**package** com.test.blocking.queue.example;

**import**java.util.concurrent.BlockingQueue;

**publicclass** Producer **implements** Runnable{

**protected**BlockingQueue<String>queue = **null**;

**public** Producer(BlockingQueue<String> queue) {

**this**.queue = queue;

}

**publicvoid** run() {

**try** {

queue.put("1");

Thread.*sleep*(1000);

queue.put("2");

Thread.*sleep*(1000);

queue.put("3");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**ArrayBlockingQueue** is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.

The **ArrayBlockingQueue** stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use an ArrayBlockingQueue:

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

Here is a **BlockingQueue** example that uses Java Generics. Notice how you can put and take String's instead of :

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

The **DelayQueue** blocks the elements internally until a certain delay has expired. The elements must implement the interface java.util.concurrent.Delayed. Here is how the interface looks:

public interface Delayed extends Comparable<Delayed< {

public long getDelay(TimeUnit timeUnit);

}

The value returned by the **getDelay**() method should be the delay remaining before this element can be released. If 0 or a negative value is returned, the delay will be considered expired, and the element released at the next take() etc. call on the DelayQueue.

The TimeUnit instance passed to the **getDelay**() method is an Enum that tells which time unit the delay should be returned in. The TimeUnit enum can take these values:

DAYS

HOURS

MINUTES

SECONDS

MILLISECONDS

MICROSECONDS

NANOSECONDS

The Delayed interface also extends the java.lang.Comparable interface, as you can see, which means thatDelayed objects can be compared to each other. This is probably used internally in the **DelayQueue** to order the elements in the queue, so they are released ordered by their expiration time.

Here is an example of how to use the **DelayQueue**:

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

The DelayedElement is an implementation of the Delayed interface that I have created. It is not part of thejava.util.concurrent package. You will have to create your own implementation of the Delayed interface to use the DelayQueue class.

**SynchronousQueue.** It is an implementation of [BlockingQueue](http://javapapers.com/java/java-blockingqueue/). Among all [Java concurrent collections](http://javapapers.com/category/java/concurrent-util/), SynchronousQueue is different. Capacity of a synchrounous queue is always zero. It is because in SynchronousQueuean insert will wait for a remove operation by another thread and vice versa.

* put() call to a **SynchronousQueue** will not return until there is a corresponding take() call.
* peek is not possible with a **SynchronousQueue**
* As there is no element iteration is also not possible.
* Insert is not possible if there is a thread trying to remove it.
* **SynchronousQueue** should be imagined like a baton in a relay race.
* If there are more than one thread waiting for a  removal so that they can do insert then with fairness set to true, threads are granted access in FIFO order.
* **SynchronousQueue** is the default **BlockingQueue** used for theExecutors.newCachedThreadPool() methods.

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.SynchronousQueue;

**publicclass** SynchronousQueueExample {

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

**final** BlockingQueue<String> synchronousQueue = **new** SynchronousQueue<String>();

SynchronousQueueProducer queueProducer = **new** SynchronousQueueProducer(

synchronousQueue);

**new** Thread(queueProducer).start();

SynchronousQueueConsumer queueConsumer1 = **new** SynchronousQueueConsumer(

synchronousQueue);

**new** Thread(queueConsumer1).start();

SynchronousQueueConsumer queueConsumer2 = **new** SynchronousQueueConsumer(

synchronousQueue);

**new** Thread(queueConsumer2).start();

}

}

**package** com.test.blocking.queue.example;

**import** java.util.Random;

**import** java.util.UUID;

**import** java.util.concurrent.BlockingQueue;

**publicclass**SynchronousQueueProducer**implements** Runnable {

**protected** BlockingQueue<String>blockingQueue;

**final** Random random = **new** Random();

**public** SynchronousQueueProducer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**publicvoid** run() {

**while** (**true**) {

**try** {

String data = UUID.*randomUUID*().toString();

System.*out*.println("Put: " + data);

blockingQueue.put(data);

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**publicclass** SynchronousQueueConsumer **implements** Runnable {

**protected** BlockingQueue<String>blockingQueue;

**public** SynchronousQueueConsumer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**publicvoid** run() {

**while** (**true**) {

**try** {

String data = blockingQueue.take();

System.*out*.println(Thread.*currentThread*().getName()

+ " take(): " + data);

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

The **LinkedBlockingQueue** class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The **LinkedBlockingQueue** keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified,**Integer.MAX\_VALUE**is used as the upper bound.

The **LinkedBlockingQueue** stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a **LinkedBlockingQueue**:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

The **PriorityBlockingQueue** class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The **PriorityBlockingQueue** is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.**PriorityQueue** class. You cannot insert null into this queue.

All elements inserted into the **PriorityBlockingQueue** must implement the **java.lang.Comparable** interface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation.

Notice that the **PriorityBlockingQueue** does not enforce any specific behavior for elements that have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a **PriorityBlockingQueue**, the Iterator does not guarantee to iterate the elements in priority order.

Here is an example of how to use the **PriorityBlockingQueue**:

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.LinkedBlockingQueue;

**publicclass** PriorityBlockingQueueExample {

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

**final** BlockingQueue<String> priorityBlockingQueue = **new** LinkedBlockingQueue<String>();

PriorityBlockingQueueProducer queueProducer = **new** PriorityBlockingQueueProducer(

priorityBlockingQueue);

**new** Thread(queueProducer).start();

PriorityBlockingQueueConsumer queueConsumer1 = **new** PriorityBlockingQueueConsumer(

priorityBlockingQueue);

**new** Thread(queueConsumer1).start();

PriorityBlockingQueueConsumer queueConsumer2 = **new** PriorityBlockingQueueConsumer(

priorityBlockingQueue);

**new** Thread(queueConsumer2).start();

}

}

**package** com.test.blocking.queue.example;

**import** java.util.Random;

**import** java.util.UUID;

**import** java.util.concurrent.BlockingQueue;

**publicclass** PriorityBlockingQueueProducer **implements** Runnable {

**protected** BlockingQueue<String>blockingQueue;

**final** Random random = **new** Random();

**public** PriorityBlockingQueueProducer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**publicvoid** run() {

**while** (**true**) {

**try** {

String data = UUID.*randomUUID*().toString();

System.*out*.println("Put: " + data);

blockingQueue.put(data);

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**publicclass** PriorityBlockingQueueConsumer **implements** Runnable {

**protected** BlockingQueue<String>blockingQueue;

**public** PriorityBlockingQueueConsumer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**publicvoid** run() {

**while** (**true**) {

**try** {

String data = blockingQueue.take();

System.*out*.println(Thread.*currentThread*().getName()

+ " take(): " + data);

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

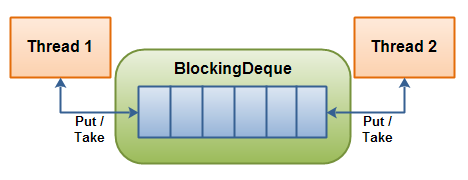
The **BlockingDeque** interface in the java.util.concurrent class represents a deque which is thread safe to put into, and take instances from. In this text I will show you how to use this **BlockingDeque**.

The **BlockingDeque** class is a **Deque** which blocks threads tring to insert or remove elements from the deque, in case it is either not possible to insert or remove elements from the deque.

A deque is short for "Double Ended Queue". Thus, a deque is a queue which you can insert and take elements from, from both ends.

BlockingDeque Usage

A **BlockingDeque** could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producing thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:



**A BlockingDeque - threads can put and take from both ends of the deque.**

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

**BlockingDeque methods:**

A **BlockingDeque** has 4 different sets of methods for inserting, removing and examining the elements in the deque. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| Insert | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| Remove | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| Examine | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| Insert | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| Remove | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| Examine | getLast(o) | peekLast(o) |  |  |

The 4 different sets of behaviour means this:

Throws Exception:   
If the attempted operation is not possible immediately, an exception is thrown.

Special Value:   
If the attempted operation is not possible immediately, a special value is returned (often true / false).

Blocks:   
If the attempted operation is not possible immedidately, the method call blocks until it is.

Times Out:   
If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

**BlockingDeque** Extends **BlockingQueue**

The **BlockingDeque** interface extends the **BlockingQueue** interface. That means that you can use a**BlockingDeque** as a **BlockingQueue**. If you do so, the various inserting methods will add the elements to the end of the deque, and the removing methods will remove the elements from the beginning of the deque.

The inserting and removing methods of the BlockingQueue interface, that is.

Here is a table of what the methods of the **BlockingQueue** does in a **BlockingDeque** implementation:

|  |  |
| --- | --- |
| BlockingQueue | BlockingDeque |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

**BlockingDeque** Implementations

Since BlockingDeque is an interface, you need to use one of its many implementations to use it. Thejava.util.concurrent package has the following implementations of the BlockingDeque interface:

[**LinkedBlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingdeque.html)

**BlockingDeque** Code Example

Here is a small code example of how to use the **BlockingDeque** methods:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

The **LinkedBlockingDeque** class implements the [**BlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) interface. Read the [BlockingDeque](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) text for more information about the interface.

The word **Deque** comes from the term "Double Ended Queue". A **Deque** is thus a queue where you can insert and remove elements from both ends of the queue.

The **LinkedBlockingDeque** is a **Deque** which will block if a thread attempts to take elements out of it while it is empty, regardless of what end the thread is attempting to take elements from.

Here is how to instantiate and use a **LinkedBlockingDeque**:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

A **java.util.concurrent.locks.Lock** is a thread synchronization mechanism just like synchronized blocks. A**Lock** is, however, more flexible and more sophisticated than a synchronized block.

By the way, in my [Java Concurrency tutorial](http://tutorials.jenkov.com/java-concurrency/index.html) I have described how to implement your own locks, in case you are interested (or need it). See my text on [Locks](http://tutorials.jenkov.com/java-concurrency/locks.html) for more details.

Java Lock Example

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. Here is a simple usage example:

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

First a Lock is created. Then it's **lock()** method is called. Now the Lock instance is locked. Any other thread calling **lock()** will be blocked until the thread that locked the lock calls unlock(). Finally **unlock()** is called, and the **Lock** is now unlocked so other threads can lock it.

## Java Lock Implementations

The java.util.concurrent.locks package has the following implementations of the Lock interface:

* ReentrantLock

## Main Differences between Locks and Synchronized Blocks

The main differences between a Lock and a synchronized block are:

* A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
* You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
* The synchronized block must be fully contained within a single method. A Lock can have it's calls tolock() and unlock() in separate methods.

## Lock Methods

The Lock interface has the following primary methods:

* **lock()**
* **lockInterruptibly()**
* **tryLock()**
* **tryLock(long timeout, TimeUnit timeUnit)**
* **unlock()**

The **lock()** method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The **lockInterruptibly()** method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The **tryLock()** method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The **tryLock(long timeout, TimeUnit timeUnit)** works like **the tryLock()** method, except it waits up the given timeout before giving up trying to lock the Lock.

The **unlock()** method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

***ConcurrentHashMap***

java.util.**ConcurrentHashMap** is implementation of the java.util.[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) interface.

java.util.**ConcurrentHashMap** enables us to store data in key-value pair form. Insertion order of key-value pairs is not maintained. **ConcurrentHashMap**is synchronized.

**CreatingConcurrentHashMap**

Constructs a new ConcurrentHashMap, Its **initial capacity** is **16**. And **load factor** is **0.75** (We’ll discuss it later in post)

|  |
| --- |
| Map<Integer,String> concurrentHashMap=**new** ConcurrentHashMap<Integer,String>(); |

Defining **ConcurrentHashMap<Integer,String>** means key can of Integer type and value can be String type only, using any other type will cause compilation error.

**What is concurrency level? What is default concurrency level of ConcurrentHashMap?**

Concurrency level tells how many threads can access ConcurrentHashMap concurrently, default **concurrency level** of ConcurrentHashMap is **16**.

|  |
| --- |
| new ConcurrentHashMap() |

Creates a new ConcurrentHashMap with concurrency level of 16.

**How ConcurrentHashMap works? Can 2 threads on same ConcurrentHashMap object access it concurrently?**

**ConcurrentHashMap**is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently.

**Can threads read the segment locked by some other thread?**

Yes. When thread locks one segment for updation it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method), but it will be able to read the data before locking.

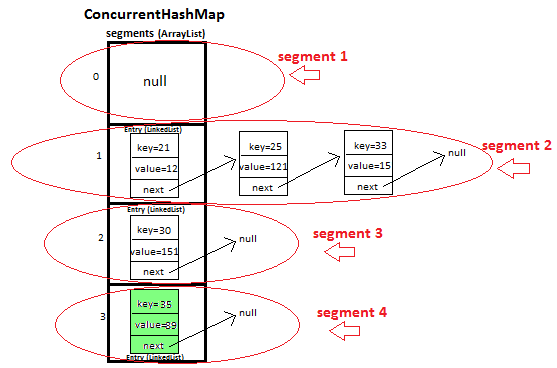
For operations such as putAll concurrent retrievals may reflect removal of only some entries.

For operations such as clear concurrent retrievals may reflect removal of only some entries.

***Segments*** *in ConcurrentHashMap with* ***diagram*** *>*

we have ConcurrentHashMap with **4 segments -**

(Diagram shows how **segments** are formed in ConcurrentHashMap)

****

*Now let’s form few questions to clear your doubts (based on above diagram) >*

**Question 1** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.get(25)**?

***Answer*** : When **map.put(25,12)** is called **segment 2** will be locked,

**key=25** also lies in **segment 2**, *When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking* (hence **map.get(25)** will return **121**)

**Question 2** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.get(33)**?

***Answer*** : When **map.put(25,12)** is called **segment 2** will be locked,

**key=33** also lies in **segment 2**, *When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking* (hence **map.get(33)** will return **15**)

**Question 3** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.put(33,24)**?

***Answer*** : When **map.put(25,12)** is called **segment 2** will be locked,

**key=33** also lies in **segment 2**, *When thread locks one segment for updation it does not allow any other thread to perform updations in same segment until lock is not released on segment*.

hence **map.put(33,24)** will have to wait for **map.put(25,12)** operation to release lock on segment.**Question 4** : What will happen **map.put(25,12)** is called and some other thread concurrently calls **map.put(30,29)**?

***Answer*** : When **map.put(25,12)** is called **segment 2** will be locked,

but **key=30** lies in **segment 3**.

*Both the kays lies in different segments,* ***hence both operations can be performed concurrently.***

**Question 5** : What will happen updations (put/remove) are in process in certain segments and new key-pair have to be put/remove in same segment ?

***Answer*** : When updations are in process *thread locks the segment and it does not allow any other thread to perform updations (put/remove) in same segment until lock is not released on segment*.

*Let’s* ***summarize*** *above section >*

*What operations lock ConcurrentHashMap segment & what operations are allowed when ConcurrentHashMap segment is locked >*

* *thread locks one segment for updation (put/remove) & it does not block it for retrieval (get) hence some other thread can read the same segment, but it will be able to read the data before locking*

*It’s important to know get operations does not lock any segment.*  
*ConcurrentHashMap* ***putIfAbsent*** *method*

*Definition of* ***putIfAbsent*** *method >*

|  |
| --- |
| **public** V putIfAbsent(K **key**, V **value**) |

*What do* ***putIfAbsent*** *method do>*

If map does not contain specified **key**, put specified **key-value** pair in map and return null.

If map already contains specified **key**, return value corresponding to specified **key**.

***putIfAbsent*** *method is equivalent to writing following code >*

|  |
| --- |
| **synchronized** (map){  **if** (!*map*.containsKey(key))  **return***map*.put(key, value);  **else**  **return***map*.get(key);    } |

***Program 1*** *to use ConcurrentHashMap’s putIfAbsent method >*

**import** java.util.concurrent.ConcurrentHashMap;

**import** java.util.concurrent.ConcurrentMap;

**publicclass**ConcurrentHashMapTest {

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

ConcurrentMap<Integer, String> concurrentHashMap =

**new** ConcurrentHashMap<Integer, String>();

concurrentHashMap.put(1, "easyJava");

System.*out*.println("concurrentHashMap : "+concurrentHashMap);

System.*out*.println("\n putIfAbsent method >> "+

concurrentHashMap.putIfAbsent(1, "test"));

System.*out*.println("concurrentHashMap : "+concurrentHashMap);

System.*out*.println("\n putIfAbsent method >> "+

concurrentHashMap.putIfAbsent(2, "audi"));

System.*out*.println("concurrentHashMap : "+concurrentHashMap);

}

}

|  |
| --- |
| /\*OUTPUT  concurrentHashMap : {1=easyJava}  putIfAbsent method >>easyJava  concurrentHashMap : {1=easyJava}  putIfAbsent method >> null  concurrentHashMap : {2=audi, 1=easyJava}  \*/ |

concurrentHashMap.putIfAbsent(1, "test") > returned easyJavabecause map was already having that key.

concurrentHashMap.putIfAbsent(2, "audi") > putted specified key-value pair in map and  returned null because map wasn’t having that key.***Program 2*** *to create method that provides* ***functionality similar to putIfAbsent method of ConcurrentHashMap*** *and to be used with HashMap >*

**import** java.util.HashMap;

**import** java.util.Map;

**publicclass**HashMapTest {

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

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

*map*.put(1, "easyJava");

System.*out*.println("hashMap : " + *map*);

System.*out*.println("\n functionalityOfPutIfAbsent method >> "

+ *functionalityOfPutIfAbsent*(1, "test"));

System.*out*.println("hashMap : " + *map*);

System.*out*.println("\n functionalityOfPutIfAbsent method >> "

+ *functionalityOfPutIfAbsent*(2, "audi"));

System.*out*.println("hashMap : " + *map*);

}

/\*\*

\* Method is created to be used with HashMap, And method provides

\* functionality similar to putIfAbsent method of ConcurrentHashMap.

\*/

**publicstaticsynchronized** String functionalityOfPutIfAbsent(Integer key,

String value) {

**if** (!*map*.containsKey(key))

**return***map*.put(key, value);

**else**

**return***map*.get(key);

}

}

|  |
| --- |
| /\*OUTPUT  hashMap : {1=easyJava}  functionalityOfPutIfAbsent method >>easyJava  hashMap : {1=easyJava}  functionalityOfPutIfAbsent method >> null  hashMap : {1=easyJava, 2=audi}  \*/ |

Please note **functionalityOfPutIfAbsent** method is **synchronized,** because this method provides same functionality as that of **ConcurrentHashMap’s putIfAbsent** method and all methods in **ConcurrentHashMap** are **synchronized**.

*functionalityOfPutIfAbsent*(1, "test") > returned easyJavabecause map was already having that key.

*functionalityOfPutIfAbsent*(2, "audi") > putted specified key-value pair in map and  returned null because map wasn’t having that key.*put element in ConcurrentHashMap*

*put(K key, V value)*

Method allows you put specified *key-value pair* in ConcurrentHashMap. If the map already contains a mapping for the *key*, the old *value* is replaced.

|  |
| --- |
| concurrentHashMap.put(11, "audi"); |

*get elements from ConcurrentHashMap*

*get(Object key)*

Method returns value corresponding to *key*.

Method returns null if map does not contain *key.*

|  |
| --- |
| concurrentHashMap.get(2); |

Method returns element on 2nd index.

*Remove element from ConcurrentHashMap*

*remove(Object key)*

Method removes *key*-value pair from ConcurrentHashMap.

|  |
| --- |
| concurrentHashMap.remove(11); |

*contains element in ConcurrentHashMap*

*contains(Object object)*

Method returns true if HAshmap contains specified on specified index.

|  |
| --- |
| concurrentHashMap.get(2); |

Method returns element on 2nd index.

*Size of ConcurrentHashMap*

*size()*

Method returns size of **ConcurrentHashMap.**

|  |
| --- |
| System.*out*.println(concurrentHashMap.size()); |

will print size of concurrentHashMap.*Iterate over ConcurrentHashMap*

Before iterating we will put 3 key-value pairs in concurrentHashMap.

  concurrentHashMap.put(11, "audi");

  concurrentHashMap.put(21, "bmw");

  concurrentHashMap.put(31, "ferrari");

Iterate over keys -

*concurrentHashMap.keySet().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap.

|  |
| --- |
| Iterator<Integer> keyIterator=concurrentHashMap.keySet().iterator();  **while**(keyIterator.hasNext()){   System.*out*.println(keyIterator.next());  }  /\*OUTPUT  21  11  31  \*/ |

***Iteration using enhanced for loop.***

**concurrentHashMap.keySet()** returns set of keys.

|  |
| --- |
| Set<Integer> keySet=concurrentHashMap.keySet();  **for**(Integer key :keySet){   System.*out*.println(key);  } |

***iterator returned by ConcurrentHashMap over key is*** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)***.*Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

|  |
| --- |
| Iterator<String> iterator=concurrentHashMap.iterator();  **while**(iterator.hasNext()){   System.*out*.println(iterator.next());   concurrentHashMap.put(4, "d");  } |

key-value has been added (map didn’t contained this key previously) during iteration and no exception is thrown.

Iterate over values -

*concurrentHashMap.values().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap.

|  |
| --- |
| Iterator<String> valueIterator=concurrentHashMap.values().iterator();  **while**(valueIterator.hasNext()){   System.*out*.println(valueIterator.next());  }  /\*OUTPUT  bmw  audi  ferrari  \*/ |

***Iteration using enhanced for loop.***

**concurrentHashMap.values()** returns collection of values.

|  |
| --- |
| Collection<String> collection=concurrentHashMap.values();  **for**(String value :collection){   System.*out*.println(value);  } |

***iterator returned by ConcurrentHashMap over values is*** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)***.*Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

|  |
| --- |
| Iterator<String> iterator=concurrentHashMap.iterator();  **while**(iterator.hasNext()){   System.*out*.println(iterator.next());   concurrentHashMap.put(5, "d");  } |

key-value has been added (map didn’t contained this key previously) during iteration and no exception is thrown.

Iterate over Entry-

*concurrentHashMap.entrySet().iterator()* method returns iterator to iterate over keys in ConcurrentHashMap.

|  |
| --- |
| Iterator<Entry<Integer, String>> entryIterator=concurrentHashMap.entrySet().iterator();  **while**(entryIterator.hasNext()){    System.*out*.println(entryIterator.next());  }  /\*OUTPUT  21=bmw  11=audi  31=ferrari  \*/ |

***Iteration using enhanced for loop.***

**concurrentHashMap.entrySet()** returns collection of values.

|  |
| --- |
| Set<Entry<Integer, String>> entrySet=concurrentHashMap.entrySet();  **for**(Entry<Integer, String> entry:entrySet){        System.*out*.println(entry);  } |

***iterator returned by ConcurrentHashMap over entry is fail-safe.*Means any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will not throw any Exception**.

*Some other important methods*

**isEmpty()** method returns true if this map contains any key-value pair.

**clear()** method removes all key-value pair from map.*Complexity of methods in ConcurrentHashMap*

|  |  |  |
| --- | --- | --- |
| Operation/ method | **Worst case** | **Best case** |
| *put(K key, V value)* | O(n) | O(1) |
| *get(Object key)* | O(n) | O(1) |

***10 features*** *of ConcurrentHashMap*

**ConcurrentHashMap** enables us to store data in key-value pair form.

**ConcurrentHashMap** is implementation of the java.util.**map** interface.

**Duplicate key**- Does not allows to store duplicate keys. If the map already contains a mapping for the key, the old value is replaced.

**Null elements -** It does **not allow to store null key or null value**. Any attempt to store null key or value throws runtimeException (NullPointerException).

1. **Insertion order -** Does not maintains insertion order.

Example-

|  |
| --- |
| Let’s say we add 3 elements in concurrentHashMap  concurrentHashMap.put(1,"ind");  concurrentHashMap.put(2,"aus");  concurrentHashMap.put(3,"sa");  On displaying insertion order will not be maintained i.e.  3,sa  2,aus  1,ind |

**synchronized -** It is synchronized.

**Performance -** ConcurrentHashMap is synchronized, hence its operations are slower as compared to some unSynchronized implementation of map interface.

**Provides locking insegments -** *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently.

1. ***iterator* are** [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) ***-***
2. *concurrentHashMap.keySet().iterator()*
3. *concurrentHashMap.values().iterator()*
4. *concurrentHashMap.entrySet().iterator()*

all three iterators are ***fail-safe.***

1. ***putIfAbsent method is present -*** If map does not contain specified **key**, put specified **key-value** pair in map and return null.

If map already contains specified **key**, return value corresponding to specified **key**.  
*When to use ConcurrentHashMap*

When we want to store data in key-value pair form.

When we don’t care about insertion order.

when we are working in multithreading environment.

Hashtable is **obsolete in java 5 i.e. JDK 1.5**, it is better to use ConcurrentHashMap than using Hashtable.*Comparison of performance between HashMap and ConcurrentHashMap*

We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**.

*We can synchronize hashMap by using Collections’s class synchronizedList method.*

|  |
| --- |
| *Map synchronizedMap = Collections.synchronizedMap(hashMap);* |

*Now, no 2 threads can access same instance of map concurrently.*

**Hence synchronized HashMap’s performance is slower as compared to ConcurrentHashMap.**

**But why we didn’t compared HashMap (unSynchronized) with ConcurrentHashMap?**

Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn’t cause any locking.

*Comparison of performance between Hashtable and ConcurrentHashMap*

Hashtable is **obsolete in java 5 i.e. JDK 1.5**, it is better to use ConcurrentHashMap than using Hashtable, because of concurrency level ConcurrentHashMap’s performance is better than Hashtable.

*What is Load Factor?*

Default load factor is 0.75

That means when set will be 75% filled,  it’s capacity will be doubled.

Example >

Initially when number of elements is 0,  default capacity =16, Load Factor =0.75, ConcurrentHashMap is 0% full.

|  |  |  |  |
| --- | --- | --- | --- |
| number of elements | capacity of ConcurrentHashMap | Load factor | ConcurrentHashMap filled in %age |
| 0 | 16 | 0.75 | 0% |
| 4 | 16 | 0.75 | 25% |
| 8 | 16 | 0.75 | 50% |
| 11 | 16 | 0.75 | 68.7% |

When next element will be added (i.e. 12th element), concurrentHashMap will be 75% filled and capacity will be doubled i.e. from 16 to 32.

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | 32 | 0.75 | 37.5% |

**Custom implementation of Datastructure**

**ArrayList custom implementation**

**How ArrayList works internally with diagrams and full program**

**Methods used in custom ArrayList >**

|  |  |
| --- | --- |
| public void **add**(E value) | Add objects in **ArrayListCustom** |
| public E **get**(int index) | Method returns element on specific index. |
| public Object **remove**(int index) | Method returns removedElement on specific index, else it throws IndexOutOfBoundException if index is negative or greater than size of size. |
| public void **display**() | -Method displays all objects in **ArrayListCustom**.  **-Insertion order is guaranteed**. |
| private void **ensureCapacity**() | Method increases capacity of list by making it double. |

**package** com.custom;

**import** java.util.Arrays;

**classArrayListCustom**<E> {

**privatestaticfinalint***INITIAL\_CAPACITY* = 10;

**private** Object elementData[]={};

**privateint**size = 0;

 /\*\*

 \* constructor.

 \*/

**public** ArrayListCustom() {

   elementData = **new** Object[*INITIAL\_CAPACITY*];

 }

 /\*\*

  \* method adds elements in ArrayListCustom.

  \*/

**publicvoidadd**(E e) {

**if** (size == elementData.length) {

     ensureCapacity(); //increase current capacity of list, make it double.

   }

   elementData[size++] = e;

 }

 /\*\*

  \* method returns element on specific index.

  \*/

 @SuppressWarnings("unchecked")

**public** E **get**(**int** index) {

**if** ( index <0 || index>= size) { //if index is negative or greater than size of size, we throw Exception.

**thrownew** IndexOutOfBoundsException("Index: " + index + ", Size " + index);

   }

**return** (E) elementData[index]; //return value on index.

 }

 /\*\*

  \* method returns removedElement on specific index.

  \* else it throws IndexOutOfBoundException if index is negative or greater than size of size.

  \*/

**public** Object **remove**(**int** index) {

**if** ( index <0 || index>= size) { //if index is negative or greater than size of size, we throw Exception.

**thrownew** IndexOutOfBoundsException("Index: " + index + ", Size " + index);

   }

   Object removedElement=elementData[index];

**for**(**int** i=index;i<size;i++){

      elementData[i]=elementData[i+1];

   }

   size--;   //reduce size of ArrayListCustom after removal of element.

**return** removedElement;

 }

 /\*\*

  \* method increases capacity of list by making it double.

  \*/

**privatevoidensureCapacity**() {

**int** newIncreasedCapacity = elementData.length \* 2;

   elementData = Arrays.*copyOf*(elementData, newIncreasedCapacity);

 }

 /\*\*

  \* method displays all the elements in list.

  \*/

**publicvoid** display() {

    System.*out*.print("Displaying list : ");

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

           System.*out*.print(elementData[i]+" ");

    }

 }

}

/\*\*

\* Main class to test ArrayListCustom functionality.

\*/

**publicclass** ArrayListCustomApp {

**publicstaticvoid** main(String...a) {

          ArrayListCustom<Integer> list = **new** ArrayListCustom<Integer>();

   list.add(1);

   list.add(2);

   list.add(3);

   list.add(4);

   list.add(1);

   list.add(2);

   list.display();

   System.*out*.println("\nelement at index "+1+" = "+list.get(1));

   System.*out*.println("element removed from index "+1+" = "+list.remove(1));

   System.*out*.println("\nlet's display list again after removal at index 1");

   list.display();

   //list.remove(11); //will throw IndexOutOfBoundsException, because there is no element to remove on index 11.

   //list.get(11);   //will throw IndexOutOfBoundsException, because there is no element to get on index 11.

    }

}

/\*Output

Displaying list : 1 2 3 4 1 2

element at index 1 = 2

element removed from index 1 = 2

let's display list again after removal at index 1

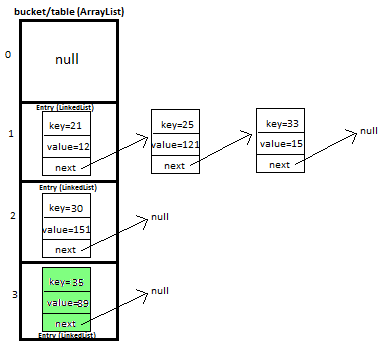
Displaying list : 1 3 4 1 2

\*/

***Complexity of methods in ArrayList >***

|  |  |  |
| --- | --- | --- |
| Operation/ method | Worst case | Best case |
| add | O(n), when array is full it needs restructuring,  operation runs in *amortized constant time.* | O(1), when array does not need any restructuring. |
| remove | O(n), when removal is done from between restructuring is needed. | O(1), when removal is done at last position, no restructuring is needed. |
| get | O(1), it is index based structure. So, complexity of  get operation is always done in O(1). | O(1) it is index based structure. So, complexity of  get operation is always done in O(1). |
| display | O(n), because iteration is done over each and every element. | O(n), because iteration is done over each and every element. |

***Custom HashMap***



This is very **important** and **trending** topic. In this post i will be explaining **HashMap** custom implementation in lots of detail with diagrams which will help you in **visualizing** the HashMap implementation.

I will be explaining how we will **put** and **get** key-value pair in HashMap by overriding-

>**equals** method - helps in checking equality of entry objects.

>**hashCode** method - helps in finding bucket’s index on which data will be stored.

We will maintain **bucket (**[**ArrayList**](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)**)** which will store **Entry (**[**LinkedList**](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html)**).**

***Entry<K,V>***

We store key-value pair by using **Entry<K,V>**

Entry contains

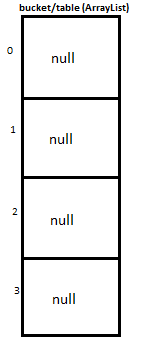
* K **key**,
* V **value** and
* Entry<K,V>**next**(i.e. next entry on that location of bucket).

|  |
| --- |
| **staticclass** Entry<K, V> {        K key;        V value;        Entry<K,V>next;  **public** Entry(K key, V value, Entry<K,V> next){  **this**.key = key;  **this**.value = value;  **this**.next = next;        }    } |

*Putting* ***5 key-value pairs in custom HashMap (step-by-step)>***

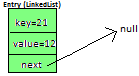
I will explain you the whole concept of HashMap by putting **5 key-value pairs in HashMap.**

**Initially,** we have bucket of **capacity=4.** (all indexes of bucket i.e. 0,1,2,3 are pointing to null)

****

**Let’s put first key-value pair in HashMap-**

**Key=21, value=12**

**newEntry Object** will be formed like this >

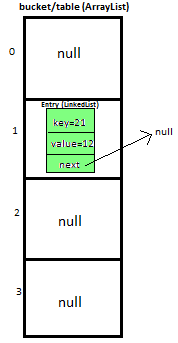
We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 21%4= 1**.

So, **1** will be the **index of bucket** on which **newEntry object** will be stored.

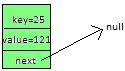
We will go to **1st**index as it is pointing to null we will **put our newEntry object there**.

At completion of this step, our HashMap will look like this-

****

**Let’s put second key-value pair in HashMap-**

**Key=25, value=121**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity=** 25**%4= 1.**

**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

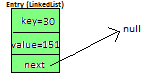
We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **if next is null** we will **put** our **newEntry object**on **next.**

At completion of this step our HashMap will look like this-

****

**Let’s put third key-value pair in HashMap-**

**Key=30, value=151**

**newEntry Object** will be formed like this >

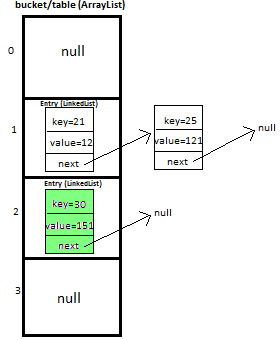
We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 30%4= 2**.

So, **2** will be the **index of bucket** on which **newEntry object** will be stored.

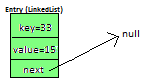
We will go to **2nd** index as it is pointing to null we will **put our newEntry object there**.

At completion of this step, our HashMap will look like this-



**Let’s put fourth key-value pair in HashMap-**

**Key=33, value=15**

Entry Object will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 33%4= 1,**

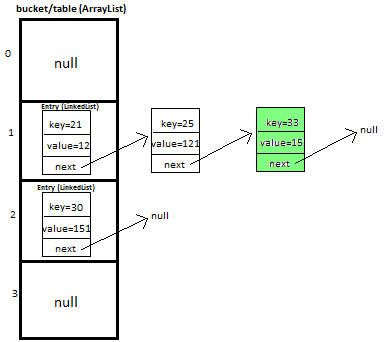
**So, 1** will be the **index of bucket** on which**newEntry object** will be stored.

We will go to **1st** index -

**>**it contains **entry with key=21**, we will **compare** two keys (i.e. **compare 21 with 33** by using **equals method**, as **two keys are different,**proceed to next  of **entry with key=21 (**proceed only if **next is not null).**

**>**now, next contains **entry with key=25**, we will **compare** two keys (i.e. **compare 25 with 33** by using **equals method**, as **two keys are different,**now **next of entry with key=25** is pointing to **null** so we won’t proceed **further,** we will **put our newEntry object on next.**

At completion of this step our HashMap will look like this-

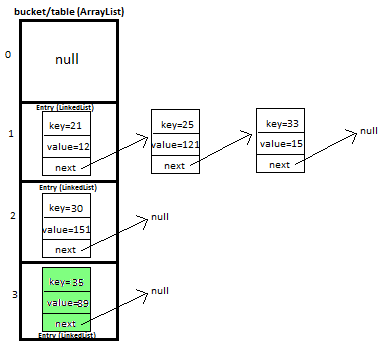
****

**Let’s put fifth key-value pair in HashMap-**

**Key=35, value=89**

**Repeat above mentioned steps.**

At completion of this step our HashMap will look like this-

****

***Methods used in custom HashMap >***

|  |  |
| --- | --- |
| public void **put**(K newKey, V data) | -Method allows you put key-value pair in HashMap  -If the map already contains a mapping for the key, the old value is replaced.  -provide complete functionality how to override equals method.  -provide complete functionality how to override hashCode method. |
| public V **get**(K key) | Method returns value corresponding to key. |
| public boolean **remove**(K deleteKey) | Method removes key-value pair from HashMapCustom. |
| public void **display**() | -Method displays all key-value pairs present in HashMapCustom.,  -insertion order is not guaranteed, for maintaining insertion order refer [LinkedHashMapCustom](http://javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html). |
| private int **hash**(K key) | -Method implements hashing functionality, which helps in finding the appropriate bucket location to store our data.  -This is very important method, as performance of HashMapCustom is very much dependent on  this method's implementation. |

**package** com.custom;

**classHashMapCustom**<K, V> {

**private** Entry<K,V>[] table;  //Array of Entry.

**privateint**capacity= 4;  //Initial capacity of HashMap

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

      K key;

      V value;

      Entry<K,V>next;

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

      }

  }

   @SuppressWarnings("unchecked")

**public** HashMapCustom(){

      table = **new** Entry[capacity];

   }

   /\*\*

    \* Method allows you put key-value pair in HashMapCustom.

    \* If the map already contains a mapping for the key, the old value is replaced.

    \* Note: method does not allows you to put null key though it allows null values.

    \* Implementation allows you to put custom objects as a key as well.

    \* Key Features: implementation provides you with following features:-

    \*    >provide complete functionality how to override equals method.

    \*  >provide complete functionality how to override hashCode method.

    \* **@param** newKey

    \* **@param** data

    \*/

**publicvoid put(K newKey, V data)**{

**if**(newKey==**null**)

**return**;   //does not allow to store null.

      //calculate hash of key.

**int** hash=hash(newKey);

      //create new entry.

      Entry<K,V> newEntry = **new** Entry<K,V>(newKey, data, **null**);

      //if table location does not contain any entry, store entry there.

**if**(table[hash] == **null**){

       table[hash] = newEntry;

       }**else**{

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

         Entry<K,V> current = table[hash];

**while**(current != **null**){//we have reached last entry of bucket.

**if**(current.key.equals(newKey)){

**if**(previous==**null**){  //node has to be insert on first of bucket.

                  newEntry.next=current.next;

                  table[hash]=newEntry;

**return**;

            }

**else**{

               newEntry.next=current.next;

               previous.next=newEntry;

**return**;

            }

         }

         previous=current;

           current = current.next;

       }

       previous.next = newEntry;

       }

   }

   /\*\*

    \* Method returns value corresponding to key.

    \* **@param** key

    \*/

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

**int** hash = hash(key);

**if**(table[hash] == **null**){

**returnnull**;

       }**else**{

       Entry<K,V> temp = table[hash];

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

**if**(temp.key.equals(key))

**return** temp.value;

           temp = temp.next; //return value corresponding to key.

       }

**returnnull**;   //returns null if key is not found.

       }

   }

   /\*\*

    \* Method removes key-value pair from HashMapCustom.

    \* **@param** key

    \*/

**publicboolean remove(K deleteKey)**{

**int** hash=hash(deleteKey);

**if**(table[hash] == **null**){

**returnfalse**;

    }**else**{

    Entry<K,V> previous = **null**;

    Entry<K,V> current = table[hash];

**while**(current != **null**){//we have reached last entry node of bucket.

**if**(current.key.equals(deleteKey)){

**if**(previous==**null**){  //delete first entry node.

                  table[hash]=table[hash].next;

**returntrue**;

            }

**else**{

                  previous.next=current.next;

**returntrue**;

            }

         }

         previous=current;

           current = current.next;

        }

**returnfalse**;

    }

   }

   /\*\*

    \* Method displays all key-value pairs present in HashMapCustom.,

    \* insertion order is not guaranteed, for maintaining insertion order

    \* refer [LinkedHashMapCustom](http://javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html).

    \* **@param** key

    \*/

**publicvoid display()**{

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

**if**(table[i]!=**null**){

                Entry<K, V> entry=table[i];

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

                      System.*out*.print("{"+entry.key+"="+entry.value+"}" +" ");

                      entry=entry.next;

                }

         }

      }

   }

   /\*\*

    \* Method implements hashing functionality, which helps in finding the appropriate

    \* bucket location to store our data.

    \* This is very important method, as performance of HashMapCustom is very much

    \* dependent on  this method's implementation.

    \* **@param** key

    \*/

**privateint hash(K key)**{

**return** Math.*abs*(key.hashCode()) % capacity;

   }

}

/\*\*

\* Main class- to test HashMap functionality.

\*/

**publicclass HashMapCustomApp** {

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

         HashMapCustom<Integer, Integer> hashMapCustom = **new** HashMapCustom<Integer, Integer>();

         hashMapCustom.put(21, 12);

         hashMapCustom.put(25, 121);

         hashMapCustom.put(30, 151);

         hashMapCustom.put(33, 15);

         hashMapCustom.put(35, 89);

         System.*out*.println("value corresponding to key 21="

                      + hashMapCustom.get(21));

         System.*out*.println("value corresponding to key 51="

                      + hashMapCustom.get(51));

         System.*out*.print("Displaying : ");

         hashMapCustom.display();

         System.*out*.println("\n\nvalue corresponding to key 21 removed: "

                      + hashMapCustom.remove(21));

         System.*out*.println("value corresponding to key 51 removed: "

                      + hashMapCustom.remove(51));

         System.*out*.print("Displaying : ");

         hashMapCustom.display();

  }

}

/\*Output

value corresponding to key 21=12

value corresponding to key 51=null

Displaying : {21=12} {25=121} {33=15} {30=151} {35=89}

value corresponding to key 21 removed: true

value corresponding to key 51 removed: false

Displaying : {25=121} {33=15} {30=151} {35=89}

\*/

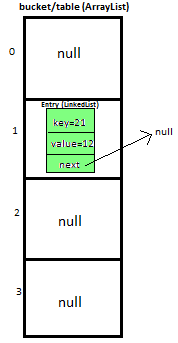
***Complexity calculation of put and get methods in HashMap >***

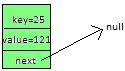
***put*** *method -* ***worst Case*** *complexity >*

**O(n).**

**But how complexity is O(n)?**

Initially, let's say map is like this -

****

And we have to insert **newEntry Object** with **Key=25, value=121** 

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 25%4= 1.**

**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **if next is null** we will **put** our **newEntry object**on **next.**

At completion of this step our HashMap will look like this-

****

**Now let’s do complexity calculation -**

Earlier there was 1 element in HashMap and for putting **newEntry Object** we iterated on it. Hence complexity was O(n).

**Note**: We may calculate complexity by adding more elements in HashMap as well, but to keep explanation simple i kept less elements in HashMap.

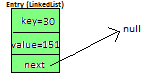
***put*** *method -* ***best Case*** *complexity >*

**O(1).**

**But how complexity is O(n)?**

Let's say map is like this -

****

And we have to insert **newEntry Object** with **Key=30, value=151** 

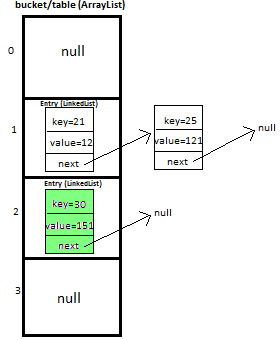
We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 30%4= 2**.

So, **2** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **2nd** index as it is pointing to null we will **put our newEntry object there**.

At completion of this step our HashMap will look like this-



**Now let’s do complexity calculation -**

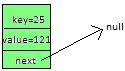
Earlier there 2 elements in HashMap but we were able to put **newEntry Object** in first go. Hence complexity was **O(1). *get*** *method -* ***worst Case*** *complexity >*

**O(n).**

**But how complexity is O(n)?**

Initially, let's say map is like this -

****

And we have to get **Entry Object** with **Key=25, value=121** 

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 25%4= 1.**

**So, 1** will be the **index of bucket** on which **Entry object** is stored.

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **next is not null** so we will repeat same process and ultimately will be able to get **Entry object**.

**Now let’s do complexity calculation -**

There were 2 elements in HashMap and for getting **Entry Object** we iterated on both of them. Hence complexity was O(n).

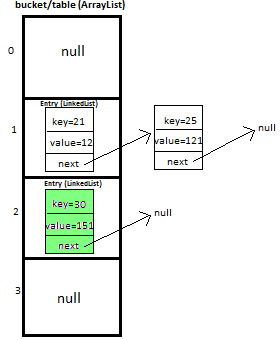
**Note**: We may calculate complexity by using HashMap of larger size, but to keep explanation simple i kept less elements in HashMap.

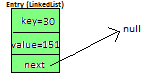
***get*** *method -* ***best Case*** *complexity >*

**O(1).**

**But how complexity is O(n)?**

Initially, let's say map is like this -



And we have to get **Entry Object** with **Key=30, value=151** 

We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 30%4= 2**.

So, **2** will be the **index of bucket** on which **Entry object** is stored.

We will go to **2nd** index and get **Entry object**.

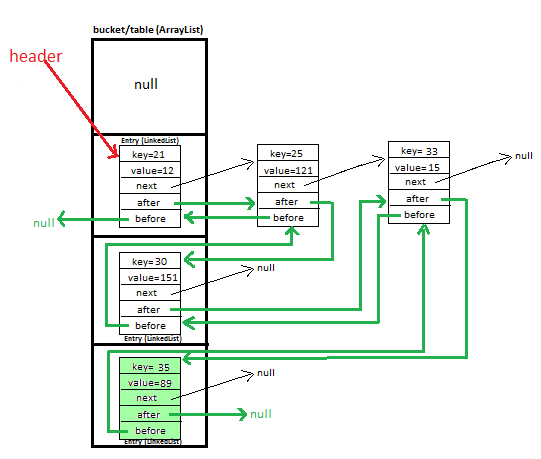
**Now let’s do complexity calculation -**

There were 3 elements in HashMap but we were able to get **Entry Object** in first go.

Hence complexity was O(1). ***Summary*** *of complexity of methods in HashMap >*

|  |  |  |
| --- | --- | --- |
| Operation/ method | **Worst case** | **Best case** |
| *put(K key, V value)* | O(n) | O(1) |
| *get(Object key)* | O(n) | O(1) |

***Custom LinkedHashMap >***



This is very **important andtrending** topic. In this post i will be explaining **LinkedHashMap** custom implementation with diagrams which will help you in **visualizing** the LinkedHashMap implementation.

I will be explaining how we will **put** and **get** key-value pair in HashMap by overriding-

>**equals** method - helps in checking equality of entry objects.

>**hashCode** method - helps in finding bucket’s index on which data will be stored.

We will maintain **bucket (**[**ArrayList**](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)**)** which will store **Entry (**[**LinkedList**](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html)**).**

Most salient feature of **LinkedHashMap** is that it **maintains insertion order** of key-value pairs. We will maintain [doubly Linked List](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html) for doing so.

While our [HashMap](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html)didn’t maintained insertion order.

***Entry<K,V>***

We store key-value pair by using **Entry<K,V>**

By using,  **Entry<K,V>before, after -**  we keep track of newly added entry in LinkedHashMap, which helps us in **maintaining insertion order**.

Entry contains

* K **key,**
* V **value,**
* Entry<K,V>**next**(i.e. next entry on that location of bucket),
* Entry<K,V>**before**and
* Entry<K,V>**after**

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

       K key;

       V value;

       Entry<K,V>next;

**Entry<K,V>before, after**;

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

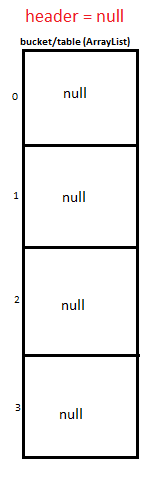
       }

   }

*Putting* ***5 key-value pairs in custom LinkedHashMap (step-by-step)>***

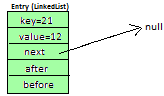
I will explain you the whole concept of **LinkedHashMap** by putting **5 key-value pairs in HashMap.**

**Initially,** we have bucket of **capacity=4.** (all indexes of bucket i.e. 0,1,2,3 are pointing to null)

****

**Let’s put first key-value pair in LinkedHashMap-**

**Key=21, value=12**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 21%4= 1**.

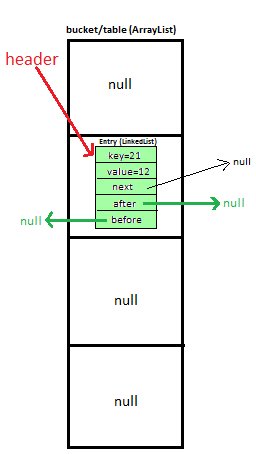
So, **1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st**index as it is pointing to null we will **put our newEntry object there**.

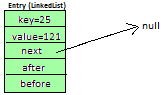
**Additionally, for maintaining insertion order-**

Update **header**, it will start pointing to **newEntry object**

At completion of this step, our HashMap will look like this-

**Let’s put second key-value pair in LinkedHashMap-**

**Key=25, value=121**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 25%4= 1.**

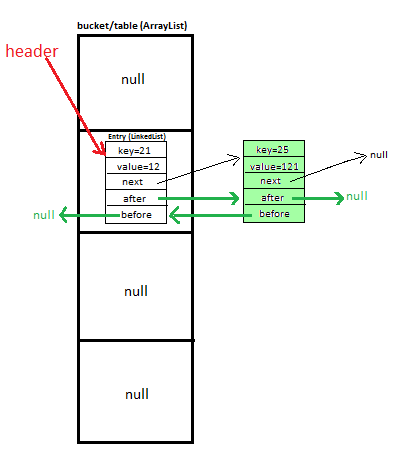
**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **if next is null** we will **put** our **newEntry object**on **next.**

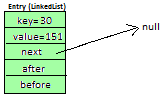
**Additionally, for maintaining insertion order-**

Update **header.after**, it will start pointing to **newEntry object** (i.e make Entry with key=21’s after point to **newEntry object**], and also make  **newEntry object’s** before point to header(Entry with key=21’)

At completion of this step our HashMap will look like this-

**Let’s put third key-value pair in HashMap-**

**Key=30, value=151**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 30%4= 2**.

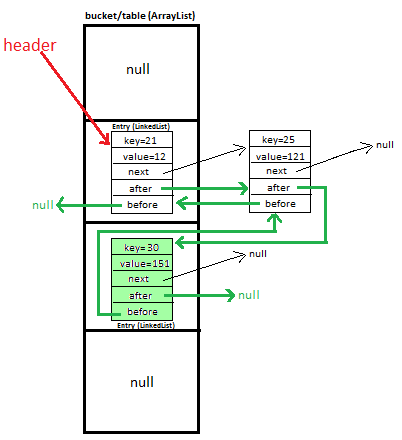
So, **2** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **2nd** index as it is pointing to null we will **put our newEntry object there**.

**Additionally, for maintaining insertion order-**

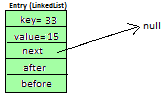
Update doubly linked list ’s **after and before.**

At completion of this step, our HashMap will look like this-



**Let’s put fourth key-value pair in LinkedHashMap-**

**Key=33, value=15**

Entry Object will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 33%4= 1,**

**So, 1** will be the **index of bucket** on which**newEntry object** will be stored.

We will go to **1st** index -

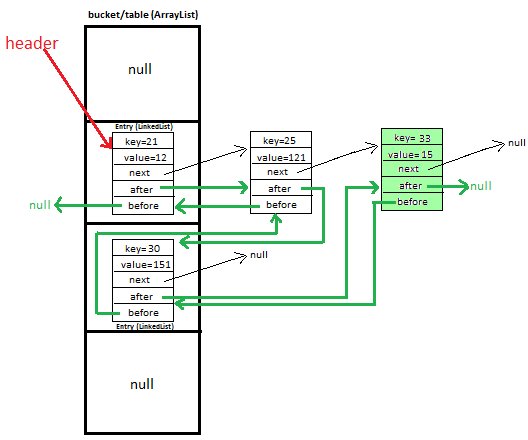
**>**it contains **entry with key=21**, we will **compare** two keys (i.e. **compare 21 with 33** by using **equals method**, as **two keys are different,**proceed to next  of **entry with key=21 (**proceed only if **next is not null).**

**>**now, next contains **entry with key=25**, we will **compare** two keys (i.e. **compare 25 with 33** by using **equals method**, as **two keys are different,**now **next of entry with key=25** is pointing to **null** so we won’t proceed **further,** we will **put our newEntry object on next.**

**Additionally, for maintaining insertion order-**

Update doubly linked list’s **after and before**(for maintaining insertion order)

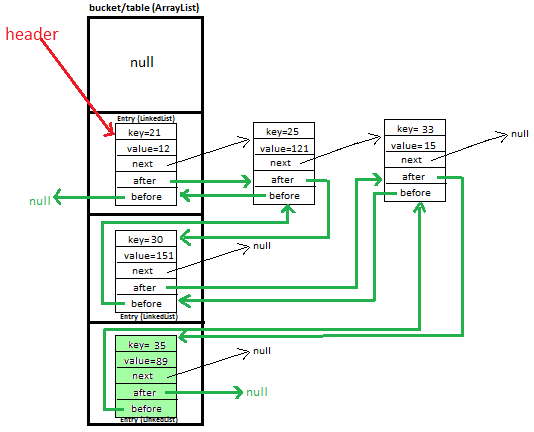
At completion of this step our HashMap will look like this-

****

**Let’s put fifth key-value pair in LinkedHashMap-**

**Key=35, value=89**

**Repeat above mentioned steps.**

At completion of this step our HashMap will look like this-****

*Methods used in custom LinkedHashMap >*

|  |  |
| --- | --- |
| public void **put**(K newKey, V data) | -Method allows you put key-value pair in HashMap  -If the map already contains a mapping for the key, the old value is replaced.  -provide complete functionality how to override equals method.  -provide complete functionality how to override hashCode method. |
| public V **get**(K key) | Method returns value corresponding to key. |
| public boolean **remove**(K deleteKey) | Method removes key-value pair from **LinkedHashMapCustom**. |
| public void **display**() | -Method displays all key-value pairs present in **LinkedHashMapCustom**.,  **-insertion order is guaranteed**. |
| private int **hash**(K key) | -Method implements hashing functionality, which helps in finding the appropriate bucket location to store our data.  -This is very important method, as performance of **LinkedHashMapCustom** is very much dependent on  this method's implementation. |
| private void **maintainOrderAfterInsert**(Entry<K, V> newEntry) | Methods helps in maintaining insertion order after insertion of key-value pair. |
| private void **maintainOrderAfterDeletion**(Entry<K, V> deleteEntry) | Methods helps in maintaining insertion order after deletion of  key-value pair. |

**package** com.custom;

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

**private** Entry<K,V>[] table;   //Array of Entry.

**privateint**capacity= 4;  //Initial capacity of HashMap

**private** Entry<K,V>header; //head of the doubly linked list.

**private** Entry<K,V>last; //last of the doubly linked list.

   /\*

    \* before and after are used for maintaining insertion order.

    \*/

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

       K key;

       V value;

       Entry<K,V>next;

**Entry<K,V>before,after;**

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

       }

   }

   @SuppressWarnings("unchecked")

**public** LinkedHashMapCustom(){

      table = **new** Entry[capacity];

   }

   /\*\*

    \* Method allows you put key-value pair in LinkedHashMapCustom.

    \* If the map already contains a mapping for the key, the old value is replaced.

    \* Note: method does not allows you to put null key thought it allows null values.

    \* Implementation allows you to put custom objects as a key as well.

    \* Key Features: implementation provides you with following features:-

    \*     >provide complete functionality how to override equals method.

    \*  >provide complete functionality how to override hashCode method.

    \* **@param** newKey

    \* **@param** data

    \*/

**publicvoid put(K newKey, V data)**{

**if**(newKey==**null**)

**return**;    //does not allow to store null.

**int** hash=hash(newKey);

      Entry<K,V> newEntry = **new** Entry<K,V>(newKey, data, **null**);

      maintainOrderAfterInsert(newEntry);

**if**(table[hash] == **null**){

        table[hash] = newEntry;

       }**else**{

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

          Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry of bucket.

**if**(current.key.equals(newKey)){

**if**(previous==**null**){  //node has to be insert on first of bucket.

                   newEntry.next=current.next;

                   table[hash]=newEntry;

**return**;

             }

**else**{

                newEntry.next=current.next;

                previous.next=newEntry;

**return**;

             }

         }

         previous=current;

            current = current.next;

        }

        previous.next = newEntry;

       }

   }

   /\*\*

    \* below method helps us in ensuring insertion order of LinkedHashMapCustom

    \* after new key-value pair is added.

    \*/

**privatevoid maintainOrderAfterInsert(Entry<K, V> newEntry)** {

**if**(header==**null**){

          header=newEntry;

          last=newEntry;

**return**;

      }

**if**(header.key.equals(newEntry.key)){

          deleteFirst();

          insertFirst(newEntry);

**return**;

      }

**if**(last.key.equals(newEntry.key)){

          deleteLast();

          insertLast(newEntry);

**return**;

      }

      Entry<K, V> beforeDeleteEntry=    deleteSpecificEntry(newEntry);

**if**(beforeDeleteEntry==**null**){

          insertLast(newEntry);

      }

**else**{

          insertAfter(beforeDeleteEntry,newEntry);

      }

   }

   /\*\*

    \* below method helps us in ensuring insertion order of LinkedHashMapCustom,

    \* after deletion of key-value pair.

    \*/

**privatevoid maintainOrderAfterDeletion(Entry<K, V> deleteEntry) {**

**if**(header.key.equals(deleteEntry.key)){

          deleteFirst();

**return**;

      }

**if**(last.key.equals(deleteEntry.key)){

          deleteLast();

**return**;

      }

      deleteSpecificEntry(deleteEntry);

   }

   /\*\*

    \* returns entry after which new entry must be added.

    \*/

**privatevoid insertAfter(Entry<K, V> beforeDeleteEntry, Entry<K, V> newEntry) {**

      Entry<K, V> current=header;

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

                 current=current.after; //move to next node.

          }

          newEntry.after=beforeDeleteEntry.after;

          beforeDeleteEntry.after.before=newEntry;

          newEntry.before=beforeDeleteEntry;

          beforeDeleteEntry.after=newEntry;

   }

   /\*\*

    \* deletes entry from first.

    \*/

**privatevoid deleteFirst(){**

**if**(header==last){ //only one entry found.

                 header=last=**null**;

**return**;

          }

          header=header.after;

          header.before=**null**;

   }

   /\*\*

   \* inserts entry at first.

   \*/

**privatevoid insertFirst(Entry<K, V> newEntry){**

**if**(header==**null**){ //no entry found

                 header=newEntry;

                 last=newEntry;

**return**;

          }

          newEntry.after=header;

          header.before=newEntry;

          header=newEntry;

   }

   /\*\*

   \* inserts entry at last.

   \*/

**privatevoid insertLast(Entry<K, V> newEntry){**

**if**(header==**null**){

                 header=newEntry;

                 last=newEntry;

**return**;

          }

          last.after=newEntry;

          newEntry.before=last;

          last=newEntry;

   }

   /\*\*

   \* deletes entry from last.

   \*/

**privatevoid deleteLast(){**

**if**(header==last){

                 header=last=**null**;

**return**;

          }

          last=last.before;

          last.after=**null**;

   }

   /\*\*

   \* deletes specific entry and returns before entry.

   \*/

**private Entry<K, V> deleteSpecificEntry(Entry<K, V> newEntry){**

          Entry<K, V> current=header;

**while**(!current.key.equals(newEntry.key)){

**if**(current.after==**null**){   //entry not found

**returnnull**;

                 }

                 current=current.after; //move to next node.

          }

          Entry<K, V> beforeDeleteEntry=current.before;

          current.before.after=current.after;

          current.after.before=current.before; //entry deleted

**return** beforeDeleteEntry;

   }

   /\*\*

    \* Method returns value corresponding to key.

    \* **@param** key

    \*/

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

**int** hash = hash(key);

**if**(table[hash] == **null**){

**returnnull**;

       }**else**{

        Entry<K,V> temp = table[hash];

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

**if**(temp.key.equals(key))

**return** temp.value;

            temp = temp.next; //return value corresponding to key.

        }

**returnnull**;   //returns null if key is not found.

       }

   }

   /\*\*

    \* Method removes key-value pair from HashMapCustom.

    \* **@param** key

    \*/

**publicboolean remove(K deleteKey)**{

**int** hash=hash(deleteKey);

**if**(table[hash] == **null**){

**returnfalse**;

     }**else**{

     Entry<K,V> previous = **null**;

     Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry node of bucket.

**if**(current.key.equals(deleteKey)){

             maintainOrderAfterDeletion(current);

**if**(previous==**null**){  //delete first entry node.

                   table[hash]=table[hash].next;

**returntrue**;

             }

**else**{

                   previous.next=current.next;

**returntrue**;

             }

         }

         previous=current;

            current = current.next;

         }

**returnfalse**;

     }

   }

   /\*\*

    \* Method displays all key-value pairs present in HashMapCustom.,

    \* insertion order is not guaranteed, for maintaining insertion order

    \* refer linkedHashMapCustom.

    \* **@param** key

    \*/

**publicvoid display(){**

      Entry<K, V> currentEntry=header;

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

          System.*out*.print("{"+currentEntry.key+"="+currentEntry.value+"}" +" ");

          currentEntry=currentEntry.after;

      }

   }

   /\*\*

    \* Method implements hashing functionality, which helps in finding the appropriate

    \* bucket location to store our data.

    \* This is very important method, as performance of HashMapCustom is very much

    \* dependent on this method's implementation.

    \* **@param** key

    \*/

**privateint hash(K key){**

**return** Math.*abs*(key.hashCode()) % capacity;

   }

}

/\*\* \*/

/\*\*

\* Main class- to test HashMap functionality.

\*/

**publicclass LinkedHashMapCustomApp** {

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

          LinkedHashMapCustom<Integer, Integer> linkedHashMapCustom = **new** LinkedHashMapCustom<Integer, Integer>();

          linkedHashMapCustom.put(21, 12);

          linkedHashMapCustom.put(25, 121);

          linkedHashMapCustom.put(30, 151);

          linkedHashMapCustom.put(33, 15);

          linkedHashMapCustom.put(35, 89);

          System.*out*.println("Display values corresponding to keys>");

          System.*out*.println("value corresponding to key 21="

                       + linkedHashMapCustom.get(21));

          System.*out*.println("value corresponding to key 51="

                       + linkedHashMapCustom.get(51));

          System.*out*.print("Displaying : ");

          linkedHashMapCustom.display();

          System.*out*.println("\n\nvalue corresponding to key 21 removed: "

                       + linkedHashMapCustom.remove(21));

          System.*out*.println("value corresponding to key 22 removed: "

                       + linkedHashMapCustom.remove(22));

          System.*out*.print("Displaying : ");

          linkedHashMapCustom.display();

   }

}

/\*Output

Display values corresponding to keys>

value corresponding to key 21=12

value corresponding to key 51=null

Displaying : {21=12} {25=121} {30=151} {33=15} {35=89}

value corresponding to key 21 removed: true

value corresponding to key 22 removed: false

Displaying : {25=121} {30=151} {33=15} {35=89}

\*/

In this post i will be explaining how to **put, get, remove Employee object in custom** [**HashSet**](http://javamadesoeasy.com/2015/02/set-custom-implementation.html).

Employee object overrides:

>**equals** method - helps in checking equality of employee objects used as key in entry objects.

>**hashCode** method - helps in finding bucket’s index on which data will be stored.

|  |
| --- |
| @Override  **publicbooleanequals**(Object o){  **if**(o==**null**)  **returnfalse**;  **if**(**this**.getClass()!=o.getClass())  **returnfalse**;            Employee e=(Employee)o;  **return** e.id.equals(**this**.id) && e.name.equals(**this**.name);     }     @Override  **publicinthashCode**(){  **return**id.hashCode() + name.hashCode();     } |

**package** com.custom;

**class** HashSetCustom<E>{

**private** HashMapCustom<E, Object>hashMapCustom;

**public** HashSetCustom(){

          hashMapCustom=**new** HashMapCustom<>();

   }

   /\*\*

   \* add objects in SetCustom.

   \*/

**publicvoid** add(E value){

          hashMapCustom.put(value, **null**);

   }

   /\*\*

    \* Method returns true if set contains the object.

    \* **@param** key

    \*/

**publicboolean** contains(E obj){

**return**hashMapCustom.contains(obj) !=**null** ? **true** :**false**;

   }

   /\*\*

    \* Method displays all objects in setCustom.

    \* insertion order is not guaranteed, for maintaining insertion order refer LinkedHashSet.

    \*/

**publicvoid** display(){

      hashMapCustom.displaySet();

   }

   /\*\*

    \* Method removes object from setCustom.

    \* insertion order is not guaranteed, for maintaining insertion order refer LinkedHashSet.

    \* **@param** obj

    \*/

**publicboolean** remove(E obj){

**return**hashMapCustom.remove(obj);

   }

}

**classEmployee** {

**private** String id;

**private** String name;

   /\*\*

   \* Employee constructor

   \*/

**public** Employee(String id, String name) { // constructor

**this**.id = id;

**this**.name = name;

   }

   @Override

**public** String toString() {

**return**"Employee[id=" + id + ", name=" + name + "] ";

   }

   @Override

**publicboolean** equals(Object o){

**if**(o==**null**)

**returnfalse**;

**if**(**this**.getClass()!=o.getClass())

**returnfalse**;

          Employee e=(Employee)o;

**return** e.id.equals(**this**.id) && e.name.equals(**this**.name);

   }

   @Override

**publicint** hashCode(){

**return**id.hashCode() + name.hashCode();

   }

}

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

**private** Entry<K,V>[] table;   //Array of Entry.

**privateint**capacity= 4;  //Initial capacity of HashMap

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

       K key;

       V value;

       Entry<K,V>next;

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

       }

   }

   @SuppressWarnings("unchecked")

**public** HashMapCustom(){

      table = **new** Entry[capacity];

   }

   /\*\*

    \* Method allows you put key-value pair in HashMapCustom.

    \* If the map already contains a mapping for the key, the old value is replaced.

    \* Note: method does not allows you to put null key thought it allows null values.

    \* Implementation allows you to put custom objects as a key as well.

    \* Key Features: implementation provides you with following features:-

    \*     >provide complete functionality how to override equals method.

    \*  >provide complete functionality how to override hashCode method.

    \* **@param** newKey

    \* **@param** data

    \*/

**publicvoid** put(K newKey, V data){

**if**(newKey==**null**)

**return**;    //does not allow to store null.

**int** hash=hash(newKey);

      Entry<K,V> newEntry = **new** Entry<K,V>(newKey, data, **null**);

**if**(table[hash] == **null**){

        table[hash] = newEntry;

       }**else**{

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

          Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry of bucket.

**if**(current.key.equals(newKey)){

**if**(previous==**null**){  //node has to be insert on first of bucket.

                   newEntry.next=current.next;

                   table[hash]=newEntry;

**return**;

             }

**else**{

             newEntry.next=current.next;

             previous.next=newEntry;

**return**;

             }

         }

         previous=current;

            current = current.next;

        }

        previous.next = newEntry;

       }

   }

   /\*\*

    \* Method returns value corresponding to key.

    \* **@param** key

    \*/

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

**int** hash = hash(key);

**if**(table[hash] == **null**){

**returnnull**;

       }**else**{

        Entry<K,V> temp = table[hash];

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

**if**(temp.key.equals(key))

**return** temp.value;

            temp = temp.next; //return value corresponding to key.

        }

**returnnull**;   //returns null if key is not found.

       }

   }

   /\*\*

    \* Method removes key-value pair from HashMapCustom.

    \* **@param** key

    \*/

**publicboolean** remove(K deleteKey){

**int** hash=hash(deleteKey);

**if**(table[hash] == **null**){

**returnfalse**;

     }**else**{

     Entry<K,V> previous = **null**;

     Entry<K,V> current = table[hash];

**while**(current != **null**){ //we have reached last entry node of bucket.

**if**(current.key.equals(deleteKey)){

**if**(previous==**null**){  //delete first entry node.

                   table[hash]=table[hash].next;

**returntrue**;

             }

**else**{

                   previous.next=current.next;

**returntrue**;

             }

         }

         previous=current;

            current = current.next;

         }

**returnfalse**;

     }

   }

   /\*\*

    \* Method displays all key-value pairs present in HashMapCustom.,

    \* insertion order is not guaranteed, for maintaining insertion order refer LinkedHashMapCustom.

    \* **@param** key

    \*/

**publicvoid** display(){

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

**if**(table[i]!=**null**){

                 Entry<K, V> entry=table[i];

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

                       System.*out*.print("{"+entry.key+"="+entry.value+"}" +" ");

                       entry=entry.next;

                 }

          }

      }

   }

   /\*\*

    \* Method returns true if set contains the object.

    \* **@param** key

    \*/

**public** K contains(K key){

**int** hash = hash(key);

**if**(table[hash] == **null**){

**returnnull**;

       }**else**{

        Entry<K,V> temp = table[hash];

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

**if**(temp.key.equals(key))

**return** key;

            temp = temp.next; //return value corresponding to key.

        }

**returnnull**;   //returns null if key is not found.

       }

   }

   /\*\*

    \* Method displays all objects in setCustom.

    \* insertion order is not guaranteed, for maintaining insertion order refer LinkedHashSet.

    \*/

**publicvoid** displaySet(){

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

**if**(table[i]!=**null**){

                 Entry<K, V> entry=table[i];

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

                       System.*out*.print(entry.key+" ");

                       entry=entry.next;

                 }

          }

      }

   }

   /\*\*

    \* Method implements hashing functionality, which helps in finding the appropriate bucket location to store our data.

    \* This is very important method, as performance of HashMapCustom is very much dependent on  this method's implementation.

    \* **@param** key

    \*/

**privateint** hash(K key){

**return** Math.*abs*(key.hashCode()) % capacity;

   }

}

/\*\*

\* Main class- to test HashMap functionality.

\*/

**publicclass** HashSetCustomEmployee {

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

          HashSetCustom<Employee> hashSetCustom = **new** HashSetCustom<Employee>();

      hashSetCustom.add(**new** Employee("10", "sam"));

      hashSetCustom.add(**new** Employee("21", "amy"));

      hashSetCustom.add(**new** Employee("31", "rob"));

      hashSetCustom.add(**new** Employee("41", "sam"));

      hashSetCustom.add(**new** Employee("42", "wil"));

          System.*out*.println("HashSetCustom contains employee with id=21 & name='amy' : "+hashSetCustom.contains(**new** Employee("21", "amy")));

          System.*out*.println("HashSetCustom contains employee with id=51 & name='pat' : "+hashSetCustom.contains(**new** Employee("51", "pat")));

       System.*out*.print("Displaying : ");

       hashSetCustom.display();

       System.*out*.println("\n\nemployee with id=21 & name='amy' removed: "+hashSetCustom.remove(**new** Employee("21", "amy")));

       System.*out*.println("employee with id=51 & name='pat' removed: "+hashSetCustom.remove(**new** Employee("51", "pat")));

          System.*out*.print("Displaying : ");

       hashSetCustom.display();

   }

}

/\*Output

HashSetCustom contains employee with id=21 & name='amy' : true

HashSetCustom contains employee with id=51 & name='pat' : false

Displaying : Employee[id=21, name=amy]  Employee[id=41, name=sam]  Employee[id=42, name=wil]  Employee[id=10, name=sam]  Employee[id=31, name=rob]

employee with id=21 & name='amy' removed: true

employee with id=51 & name='pat' removed: false

Displaying : Employee[id=41, name=sam]  Employee[id=42, name=wil]  Employee[id=10, name=sam]  Employee[id=31, name=rob]

\*/

[**Difference between Method overloading and Method overriding in java - in detail with programs**](http://www.javamadesoeasy.com/2015/06/10-difference-between-method.html)

|  |  |  |
| --- | --- | --- |
|  | **Method overloading** | **Method overriding** |
| **1** | When a class have same method name with different argument, than it is called method overloading. | Method overriding - Method of superclass is **overridden** in subclass **to provide more specific implementation**. |
| **2** | Method is overloaded by - keeping same name of method and only changing number of arguments  Let’s compare with method overriding.   1. **Method name - same method name.** 2. **Access modifier -** Does not matter.**Return type -** Does not matter.**Number of parameters in java -**Have **different number of** [**parameters**](http://www.javamadesoeasy.com/2015/06/difference-between-arguments-and.html) 3. **Exception thrown -** Does not matter. | In Method overriding **-** Method of superclass is **overridden** in subclass when overriding method of subclass -   1. **Method name -** Have **same name as of superclass method,** 2. [**Access modifier**](http://www.javamadesoeasy.com/2015/06/access-modifier-access-specifier-in.html) **-** Must not have more restrictive modifier. Example - public method cannot be overridden by private method.   https://lh4.googleusercontent.com/57jtmPxTx9UfXgbCbDGbAUgpEG4xsBuQIBfO790kOsGKYeFsmqlp7SsIemRoefbLVzcBrZqEDLk3JnAEKr7q2Rd3EOGV_3BOGPsNMH7ZSvw5W9AAsn12WJy0rbWL2SnWo9fBz-A   1. **Return type -** Java **allow overriding by changing the return type**, but only **Covariant return type** are allowed. 2. **Number of parameters in java -**Have **same number of parameters** 3. [**Exception thrown**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html) **-** Must **not throw new or broader** [**unchecked exception**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)**,** though method may **throw new or broader runtime exception.** |
| **3** | Method overloading is **generally done in same class** but **can also be done in SubClass** (See [Program 3](http://www.javamadesoeasy.com/2015/06/method-overloading-in-java-in-detail.html)) | Method overriding is always done in subClass. |
| **4** | Both [**Static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) **and instance method can be overloaded** in java. | **Only instance methods can be overridden** in java.  **Static methods can’t be overridden** in java. ([Please refer this article for detailed analysis and explanation with program](http://www.javamadesoeasy.com/2015/05/why-static-method-cannot-be-overridden.html)) |
| **5** | **Main method can also be overloaded** in java (In [Program 4](http://www.javamadesoeasy.com/2015/06/method-overloading-in-java-in-detail.html)) | **Main method can’t be overridden** in java, because main is static method and static methods can’t be overridden in java (as mentioned in above point) |
| **6** | **private methods can be overloaded** in java. | **private methods can’t be overridden** in java, because private methods are not inherited in subClass. |
| **7** | [**final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) **methods can be overloaded** in java. | **final methods can’t be overridden** in java, because final methods are not inherited in subClass. |
| **8** | Call to overloaded method is **bonded** at **compile time**. | Call to overridden method is **bonded** at **runtime.** |
| **9** | Method overloading concept is also known as **compile time polymorphism** in java. | Method overriding concept is also known as **runtime time polymorphism** in java. |
| **10** | Example  - Please see below | Example - Please see below |

**Example of method overloading -**

|  |
| --- |
| /\*     \* Method to calculate sum of 2 arguments     \*/  **publicvoidsum(int x, int y)** {            System.*out*.println("sum of 2 arguments = "+ (x+y));     }     /\*     \* Method to calculate sum of 3 arguments     \*/  **publicvoidsum(int x, int y, int z)** {            System.*out*.println("sum of 3 arguments = "+ (x+y+z));     } |

**sum()** method **logically perform almost similar tasks and the only difference is in number of arguments.** Method overloading enables same method name **sum()** to be **reused** in program.**Example of method overriding-**

Different animals eat different food, like Lion eat flesh and Goat eat grass. So we can have generic **SuperClass** which tells that **Animal** might eat flesh, grass or may be some other thing.

Now, we can have **SubClasses** like **Lion** which more specifically that **Lion eat flesh**.

So at runtime, rather than calling food() method of SuperClass, food() method of subclass will be called **and this way we could derive advantage of creating more specific SubClasses and overriding method.** We will elaborate this in[**Program 1**](http://www.javamadesoeasy.com/2015/06/method-overloading-in-java-in-detail.html).

|  |
| --- |
| /\*  \* superclass - Animal  \*/  **class** Animals {  **void** food() {            System.*out*.println("Animal may eat flesh, grass or ....");     }  }  /\*  \* subclass of Animal - Lion  \*/  **class** Lion **extends** Animals {     @Override  **void** food() {            System.*out*.println("Lion eat - flesh");     }  } |

[**Differences between Interface and abstract class in java - in detail with programs**](http://www.javamadesoeasy.com/2015/06/10-differences-between-interface-and.html)

|  |  |  |
| --- | --- | --- |
|  | Interface | Abstract class |
| *1* | Interface helps in achieving **pure abstraction** in java. | Abstract class aren’t **purely abstraction** in java |
| *2* | All Interface are **abstract by default.**  So, it’s **not mandatory** to write abstract keyword with interface.  Example-   |  | | --- | | **interface** MyInterface {//compiler will add  **abstract**  } |   Because of default additions done by compiler, **above code** will be **same as** writing **below code**-   |  | | --- | | **abstractinterface** MyInterface {  } | | It’s **mandatory** to write abstract keyword to make class abstract.  Example-  **abstractclass** MyAbstractClass{  **abstractvoid m();**  } |
| *3* | All the variables in Interface are **public,** [**static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) **and** [**final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) by default. Interface variables are also known as **constants**.   |  | | --- | | **interface** MyInterface {  **int***i*=2;//compiler will add **public, static and final**  } |   Because of default additions done by compiler, writing **above code** will be **same as** writing **below code**-   |  | | --- | | **interface** MyInterface {  **publicstaticfinalint***i*=2;  } | | Abstract class can have private, instance variables as well. |
| *4* | All the methods in Interface are **public and abstract** by default.   |  | | --- | | **interface** MyInterface {  **void** m(); //compiler will add **public and abstract**  } |   Because of default additions done by compiler, writing **above code** will be **same as** writing **below code**-   |  | | --- | | **interface** MyInterface {  **publicabstractvoid** m();  } | | Abstract class can have private, final, abstract, static and **instance** methods.  **Note** : Method cannot be private and abstract.  Method cannot be final and abstract. |
|  |  |  |
| *5* | Interface does **not have constructors.** | Abstract class **have constructors.**  [Constructors](http://www.javamadesoeasy.com/2015/06/constructor-in-java-constructor.html) are called by constructors of sub-class when object of subclass is created. |
|  |  |  |
| *6* | Interface allows us to achieve **multiple inheritance** in java.  Example - See [*Program 2*](http://www.javamadesoeasy.com/2015/06/interface-in-java-multiple-inheritance.html).  Also see [*Program 2.1*](http://www.javamadesoeasy.com/2015/06/interface-in-java-multiple-inheritance.html)to find out *why* ***interface allow multiple inheritance.*** | Abstract class  **doesn’t allows multiple inheritance** in java.see [*Program 2.2*](http://www.javamadesoeasy.com/2015/06/interface-in-java-multiple-inheritance.html)to find out *why* ***classes doesn’t allow multiple inheritance.*** |
| *7* | Interface doesn’t extend classes. | classes **implement** Interface.  classes **can implement more than one** interface. |
|  |  |  |
| *8* | If any new method is added in Interface then all concrete classes which **implements** that interface **must provide implementation of newly added method**, because all methods in interface are abstract by default.Example -  Please see Program 3 | If any new **instance method** is added in Abstract class then all concrete classes which **extends** that abstract class **need not to provide implementation of newly added instance method**..  If any new **abstract method** is added in Abstract class then all concrete classes which **extends** that abstract class **must provide implementation of newly added abstract method**.  Example -  Please see Program 4 and Program 5 |
| *9* | Interface methods **cannot have synchronized method**, concrete **class which implements interface can make methods synchronized.** | Abstract class can have **synchronized instance methods**,  but  **abstract methods cannot be synchronized**. |
| *10* | *When to* ***use interface practically*** *-*  Let’s say we have to choose between **class or interface** for **Animals**, than **habitat of  animals might be land or water**.  And food of all animals might be different.*So, we will* ***create interface*** *with -*  ***abstract method = habitat()*** *[because animals might be living on land or water]*  ***abstract method = food()*** *[because food of all animals might be different]***Example -**  see [program 4.1](http://www.javamadesoeasy.com/2015/06/interface-in-java-multiple-inheritance.html) | *When to* ***use abstract class practically*** *-*  Let’s say we have to choose between **class or interface** for **TerrestrialAnimals**, than one thing will be for sure that **habitat of all Terrestrial animals must be land**. That means we can have **same implementation of habitat method for all Terrestrial animals**. And **food of all Terrestrial animals might be different**.  *So, we will* ***create abstract class****with -*  ***instance method = habitat()*** *[because all Terrestrial animals live on land]*  ***abstract method = food()*** *[because food of all Terrestrial animals might be different]*  **instance method habitat()** will be inherited in all subclasses.  **Example -**  see [program 3.1](http://www.javamadesoeasy.com/2015/06/abstract-class-in-java-when-to-use.html) |

***What is constructor chaining in java?whenever the object of class is created, implicitly default no-arg*** [***constructor of class***](http://www.javamadesoeasy.com/2015/06/constructor-in-java-constructor.html) ***and its super class constructor is called.*Q. But how constructor of superclass is called?**

**A.** Implicitly first statement of constructor is super(), [that means by default first statement of constructor super() is called, super() calls implicit/explicit no-arg constructor of superclass].Let’s we have superclass and subclass like this -

|  |
| --- |
| **class** SuperClass{  }  **class** SubClass **extends** SuperClass{  } |

compiler will add default implicit no-arg constructor -

|  |
| --- |
| **class** SuperClass{  **SuperClass(){ //no-arg constructor**  **super();**  **}**  }  **class** SubClass **extends** SuperClass{  **SubClass(){**  **super();**  **}**  } |

***What is cloning in java?***Cloning is done for copying the object, cloning can be done using shallow or deep copy, we will discuss it later in post.***Few key points about clone method>***

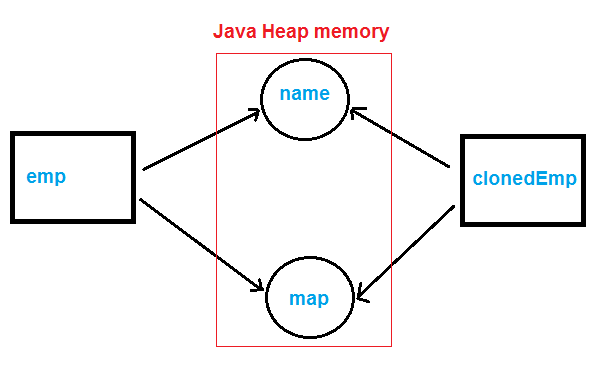
* ***1)* Definition** of clone method -

|  |
| --- |
| **protectednative** Object clone() **throws** CloneNotSupportedException; |

* Clone is a **protected** method - clone method can’t be called outside class without inheritance.
* Clone is **native** method, if not overridden its implementation is provided by JVM.
* It returns Object - Means explicitly cast is needed to convert it to original object.
* ***2)*By default clone method do shallow copy.**
* ***3)*** Class must implement marker interface java.lang.Cloneable. If class doesn’t implement Cloneable than calling clone method on its object will throw **CloneNotSupportedException**.
* ***4)*shallow copy-** If we implement Cloneable interface, we must override clone method and call super.clone() from it, invoking super.clone() will do shallow copy.
* ***5)* Deep copy** - We need to provide custom implementation of clone method for deep copying.  When the copied object contains some other object its references are copied recursively in deep copy.

***Shallow copy*** *>*

Let’s say we want to shallow copy **emp** object using clone method.



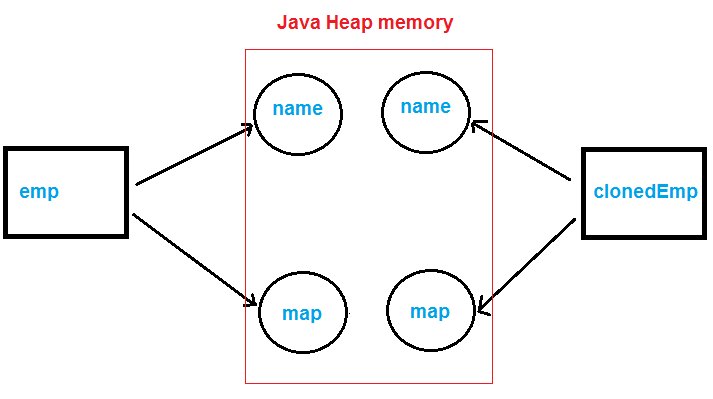
In shallow copy, different object is created after cloning (i.e. **clonedEmp** is created from **emp**) but member variables keep on referring to same object (i.e. **name** and **map**).

*Program for shallow copy>*

|  |
| --- |
| **import** java.util.\*;  **publicclass** CloneShallow **implements Cloneable** {  **private** String name;  **private** Map<Integer, Integer>map;  **public** CloneShallow(String name,Map<Integer,Integer> map){  **this**.name=name;  **this**.map=map;     }     /\*     \* override clone method for doing **shallow copy**.     \*/  **@Override**  **public Object clone() {**  **System.*out*.println("Doing shallow copy");**  **try {**  **returnsuper.clone();**  **} catch (CloneNotSupportedException e) {**  **returnnull;**  **}**  **}**  **publicstaticvoid** main(String[] args) **throws** CloneNotSupportedException  {            Map<Integer,Integer> map=**new** HashMap<Integer,Integer>();            map.put(11, 11);            CloneShallow obj=**new** CloneShallow("test",map);            CloneShallow clonedObj=(CloneShallow)**obj.clone()**;  **System.*out*.println(obj==clonedObj);             //false**  **System.*out*.println(obj.name==clonedObj.name);   //true**  **System.*out*.println(obj.map==clonedObj.map);     //true**       }  }  /\*OUTPUT  Doing shallow copy  false  true  true  \*/ |

***Deep copy >***

Let’s say we want to deep copy **emp** object using clone method.



In deep copy, different object is created after cloning (i.e. **clonedEmp** is created from **emp**) , also member variables starts referring to different objects (i.e. **name** and **map**).

*Program for deep copy>*

|  |
| --- |
| **package** clone;  **import** java.util.\*;  **publicclass** CloneDeep **implements Cloneable** {  **public** CloneDeep(String name,Map<Integer,Integer> map){  **this**.name=name;  **this**.map=map;     }  **private** String name;  **private** Map<Integer,Integer>map;     /\*     \* override clone method for doing **deep copy**.     \*/  **@Override**  **public CloneDeep clone(){**  **System.*out*.println("Doing deep copy");**  **Map<Integer,Integer> map=new HashMap<Integer,Integer>();**  **Iterator<Integer> it=this.map.keySet().iterator();**  **while(it.hasNext()){**  **Integer key=it.next();**  **map.put(key,this.map.get(key) );**  **}**  **CloneDeep cloneDetailedDeep=new CloneDeep(new String(name), map);**  **return cloneDetailedDeep;**  **}**  **publicstaticvoid** main(String[] args) **throws** CloneNotSupportedException  {              Map<Integer,Integer> map=**new** HashMap<Integer,Integer>();            map.put(1, 11);            CloneDeep obj1=**new** CloneDeep("sam",map);            CloneDeep obj2=(CloneDeep)**obj1.clone()**;  **System.*out*.println(obj1==obj2);           //false**  **System.*out*.println(obj1.name==obj2.name); //false**  **System.*out*.println(obj1.map==obj2.map);   //false**         }  }  /\*OUTPUT  Doing deep copy  false  false  false  \*/ |

***8*** *Different* ***techniques*** *for deep copying>*

* 1) [**XStream**](http://xstream.codehaus.org/tutorial.html)can be used for cloning.

|  |
| --- |
| privatestaticfinalXStream XSTREAM = newXStream();  //code  Object clonedObject = XSTREAM.fromXML(XSTREAM.toXML(obj)); |

* 2) [**Jackson JSON**](http://jackson.codehaus.org/) can also be used for [serializing](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object.html) object to JSON and read it back.
* 3) [**Kyro**](https://github.com/EsotericSoftware/kryo)for deep cloning.
* 4) [**Cloner**](https://github.com/kostaskougios/cloning)can be used for cloning which performs relatively fast cloning with reflection ( faster than serialization methods).

|  |
| --- |
| **Cloner**cloner=newCloner();  MyClass clone=cloner.deepClone(obj); |

* 5) [**Apache commons**](https://commons.apache.org/) allows you to do deep cloning

Let's say emp is reference variable referring to object of Employee class.

|  |
| --- |
| *Employee empClone = org.apache.commons.lang.SerializationUtils.clone(emp);* |

empClone will refer to cloned object.

* 6) [**Dozer**](https://github.com/DozerMapper/dozer) - Dozer for deep cloning.

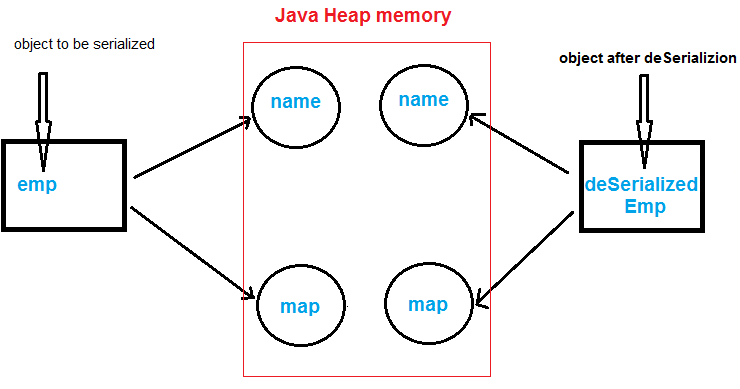
Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another, it is an open source mapping framework that is robust, generic, flexible, reusable, and configurable.

* 7) Deep cloning using [**Serialization and deserialization**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object.html)- Simply serialize object by creating **object ofObjectOutputStream** and then using its **writeObject method**. And then deserializing the object. (As done in above program)
* 8) Deep cloning using **Reflection** - (As done in above program)

***Deep copy using***[***Serialization and Deserialization***](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object.html)*>*

*Program for* ***deep copy usingSerialization and Deserialization*** *>*

Let’s say we want to deep copy **emp** object using serialization.



In serialization and deserialization process, different object is created after deserialization ( i.e. **deSerializedEmp** is created from **emp**) , also member variables starts referring to different objects (i.e. **name** and **map**).

**package**clone;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.ObjectInput;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.io.Serializable;

**import** java.util.HashMap;

**import** java.util.Map;

**class** Employee **implements** Serializable {

**privatestaticfinallong***serialVersionUID* = 1L;

**private** String name;

**private** Map<Integer,Integer>map;

**public** Employee(String name,Map<Integer,Integer> map) {

**this**.name = name;

**this**.map=map;

}

@Override

**public** String toString() {

**return**"Employee [name=" + name + "]";

}

**public** String getName() {

**return**name;

}

**public** Map<Integer, Integer> getMap() {

**return**map;

}

}

/\*\*

\* Author

\* Main class

\*/

**publicclass**CloneUsingSerialization {

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

Map<Integer,Integer> map=**new** HashMap<Integer,Integer>();

map.put(1, 11);

Employee emp = **new** Employee("test",map);

**try** {

OutputStream fout = **new** FileOutputStream("ser.txt");

ObjectOutput oout = **new** ObjectOutputStream(fout);

System.*out*.println("Serialization process has started, "

+ "serializing employee object...");

oout.writeObject(emp);

fout.close();

oout.close();

System.*out*.println("employee Serialization completed.");

//DeSerialization process >

InputStream fin=**new** FileInputStream("ser.txt");

ObjectInput oin=**new** ObjectInputStream(fin);

System.*out*.println("\nDeSerialization process has started, "

+ "deSerializing employee object...");

Employee deSerializedEmp=(Employee)oin.readObject();

fin.close();

oin.close();

System.*out*.println("employee DeSerialization completed.");

System.*out*.println(emp==deSerializedEmp); //false

System.*out*.println(emp.getName()==deSerializedEmp.getName()); //false

System.*out*.println(emp.getMap()==deSerializedEmp.getMap()); //false

} **catch** (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

|  |
| --- |
| /\*OUTPUT  Serialization process has started, serializing employee objects...  Object Serialization completed.  DeSerialization process has started, displaying employee objects...  Object DeSerialization completed.  false  false  false  \*/ |

***Is there any difference between creating string with and without new operator?***When String is created **without new** operator, it will be created in [**string pool**](http://www.javamadesoeasy.com/2015/05/string-pool-string-literal-pool-string.html).When String is created using **new** operator, it will force JVM to create new string in heap (not in string pool).*Let’s discuss step-by-step what will happen when below 5 statements will be executed >*

|  |
| --- |
| **String s1 = "abc";**  **String s2 = new String("abc");**  **String s3 = "abc";**  **String s4 = new String("abc");**  **String s5 = new String("abc").intern();** |

**String s1 = "abc";**

No string with “abc” is there in pool, so JVM will create string in **string pool** and s1 will be a reference variable which will refer to it.**String s2 = new String("abc");**

string is created using **new** operator, it will force JVM to create new string in heap (not in string pool).**String s3 = "abc";**

string with “abc” is there in pool, so s3 will be a reference variable which will refer to “abc” in **string pool.String s4 = new String("abc");**

string is created using **new** operator, it will force JVM to create new string in heap (not in string pool).**String s5 = new String("abc").intern();**

string is created using new operator but intern method has been invoked on it, so s5 will be a reference variable which will refer to “abc” in **string pool.**

Difference between String, StringBuffer and StringBuilder in java - In depth coverage

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Property | **String** | **StringBuffer** | **StringBuilder** |
| 1 | **Storage area** | When String is created **without using new operator**, JVM will create string in [**string pool**](http://www.javamadesoeasy.com/2015/05/string-pool-string-literal-pool-string.html) **area of heap.**  When String is created using **new** operator, it will force JVM to create new string **in heap** (not in string pool).  **Example -->**  **String s1 = "abc";**  **> in string pool area of heap.**  **String s2 = new String("abc");**  **> in heap** | StringBuffer is created in **heap**. | StringBuilder is created in **heap**. |
| **2** | **mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html) | [**String is immutable class in java**](http://www.javamadesoeasy.com/2015/05/string-is-immutable-in-java.html), any changes made to Sting class produces new String.Please see  **example 1a** and **example 1b** below. | StringBuffer is **mutable** class in java, any changes made to StringBuffer class won’t produces new String.  Please see  **example 2** below. | StringBuilder is **mutable** class in java, any changes made to StringBuilder class won’t produces new String.Please see  **example 3** below. |
| **3** | [**Thread-safe**](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)**/** [**Synchronized**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) | **String is immutable that makes it a thread -safe class.** | StringBuffer methods are **synchronized**.  Means no 2 threads on same StringBuffer object **cannot** access methods concurrently. | StringBuilder methods are **not synchronized**.  Means 2 threads on same StringBuilder object **can** access methods concurrently.  **Note** : Methods of StringBuffer and StringBuilder are same, the only difference is of synchronization. |
| **4** | **Performance** | Value of String in String pool is **cached**, hence making it **fast**.  String created with new operator is also **fast** process. | Because of **synchronized methods** its **slow**. | Because of **non synchronized methods** its **fast**. |
| **5** | **Internal working** | Let’s say we have following statements -  *String str = "abc" ;*  *str = str + "def";*  [Internally](http://www.javamadesoeasy.com/2015/05/string-pool-string-literal-pool-string.html)*+* operator uses StringBuffer for concatenating strings.  So,  **String str = new StringBuilder(str).append("def").toString();** | Internally StringBuffer adds new characters to previous StringBuffer. | Internally StringBuilder adds new characters to previous StringBuilder. |
| **6** | **Introduced in** | Java 1  jdk 1.0 | Java 1  jdk 1.0 | Java 5  jdk 1.5 |

**String mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)**Example 1a -->String str= "ab";**

> No string with “**ab**” is there in string pool, so JVM will create string “**ab**” in **string pool** and **str** will be a reference variable which will refer to it.**str.concat("cd")**

>**cd**will be concatenated with **ab** and new string  “**abcd**”  will be formed. No string with “**abcd**” is there in pool, so JVM will create string “**abcd**” in **string pool, but** there won’t be any reference variable to “**abcd**”  (we are just using it only in syso statement), meanwhile str will still be pointing to “**ab**”.**System.*out*.println(str);**

**str** is referring to  “**ab**” and string “**abcd**” will be eligible for [garbage collection](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html).**String mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)**Example 1b -->**

What will happen when below 2 statements will execute >

|  |
| --- |
| **String str= "ab";**  **str = "abcd";** |

**String str= "ab";**

No string with “**ab**” is there in string pool, so JVM will create string “**ab**” in **string pool** and **str** will be a reference variable which will refer to it.**str = "abcd";**

Now, No string with “**abcd**” is there in string pool, so JVM will create new string “**abcd**” in **string pool** and **str** will be a reference variable which will refer to it.String **"abcd"** will stay in string pool but reference to it will be lost, and it will be eligible for garbage collection.**StringBuffer mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)**Example 2-->StringBuffer sBuffer = new StringBuffer("ab") ;**

>JVM will create stringBuffer object “**ab**” in **heap** and **sBuffer** will be a reference variable which will refer to it.**sBuffer= sBuffer.append("cd");**>  “**cd**” will be added to StringBuffer object referred by **sBuffer.** So,   “**abcd**” will be formed.*(Note: addition was made to previous object, no new object was formed,*

***Behaviour was different as compared to immutable String’s concat function***

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

**sBuffer** is referring to  “**abcd**” . **StringBuilder mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)**Example 3-->StringBuilder sBuilder = new StringBuilder("ab") ;**>JVM will create stringBuilder object “**ab**” in **heap** and **sBuilder** will be a reference variable which will refer to it.**sBuilder=sBuilder. append("cd");**

>  “**cd**” will be added to StringBuilder object referred by **sBuilder.** So,   “**abcd**” will be formed.*(Note: addition was made to previous object, no new object was formed,*

***Behaviour was different as compared to immutable String’s concat function***

*)***System.*out*.println(sBuilder);sBuilder** is referring to  “**abcd**” .

***What are immutable classes in java? How we can create immutable classes in java? And what are advantages of using immutable classes?***

**Any change made to object of immutable class produces new object.**

Example- [**String is Immutable class in java**](http://www.javamadesoeasy.com/2015/05/string-is-immutable-in-java.html), any changes made to Sting class.We must follow following steps for creating immutable classes -

1) [**Final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) **class** - Make class final so that it cannot be inherited2) **private member variable** -> Making member variables private ensures that fields cannot be accessed outside class.3) **final member variable** -> Make member variables final so that once assigned their values cannot be changed4) **Constructor** -> Initialize all fields in constructor.

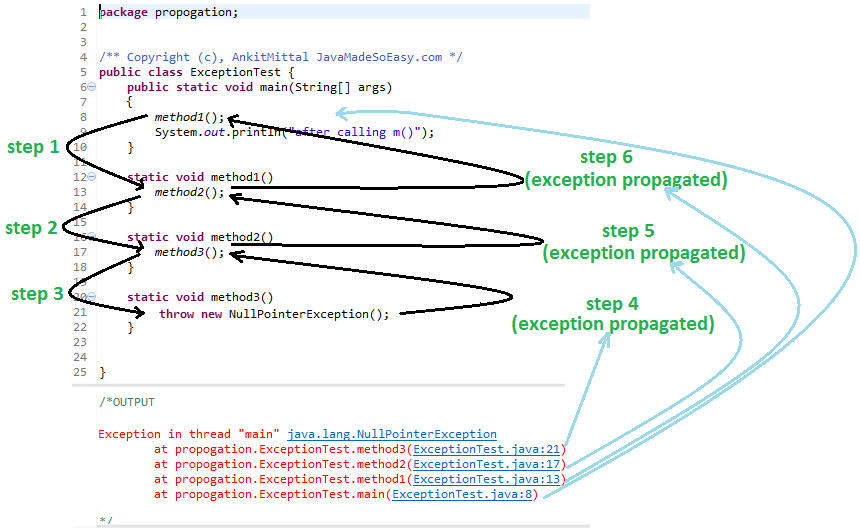
     assign all mutable member variable using new keyword.5) **Don't provide setter methods** in class/ provide only getter methods.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | [**Exception**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) | [**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html) |
| 1 | serious problem? | Exception does **not indicate any serious problem**. | Error **indicates some serious problems** that our **application should not try to catch.** |
| 2 | divided into  [**checked** and **unchecked**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) | Exception are divided into **checked** and **unchecked exceptions**. | Error are **not divided** further into such classifications. |
| 3 | Which classes are which type of exception? either  **checked orunchecked** exception? | The class **Exception and all its subclasses** that are **not also subclasses of RuntimeException** are checked exceptions.  The class **RuntimeException and all its subclasses** are unchecked exceptions.  Likewise,  The class **Error and all its subclasses** are unchecked exceptions. | Error and its subclasses are regarded as **unchecked** exceptions |
| 4 | Most frequently faced exception and errors | **checked exceptions>**  SQLException,  IOException,  ClassNotFoundException  **unchecked exceptions>**  [NullPointerException](http://www.javamadesoeasy.com/2015/05/nullpointerexception-in-java.html), ArithmeticException, | **VirtualMachineError, IOError, AssertionError,** [**ThreadDeath**](http://www.javamadesoeasy.com/2015/04/threaddeath-error-calling-stop-method.html),  **OutOfMemoryError, StackOverflowError.** |
| 5 | Why to catch or not to catch? | Application **must catch** the Exception because they does not cause any major threat to application. | Application **must not catch** the Error because they does cause any major threat to application.  Example >  Let’s say errors like OutOfMemoryError and StackOverflowError occur and are caught then JVM might not be able to free up memory for rest of application to execute, so it will be better if application don’t catch these errors and is allowed to terminate. |

*Exception Propagation >*

Whenever methods are called [stack](http://javamadesoeasy.com/2015/01/stacks.html) is formed and an exception is first thrown from the top of the stack and if it is not caught, it starts coming down the stack to previous methods until it is not caught.

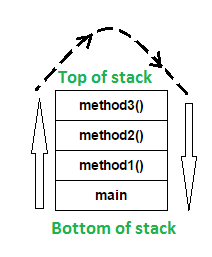
If exception remains uncaught even after reaching bottom of the stack it is propagated to JVM and program is terminated. *Propagating* [***unchecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception (NullPointerException) >*

**unchecked** exceptions are **automatically propagated** in java.

***Now, i’ll be explaining you how unchecked exception*** *was* ***propagated.***

***Let’s see step by step what happened in above program >***

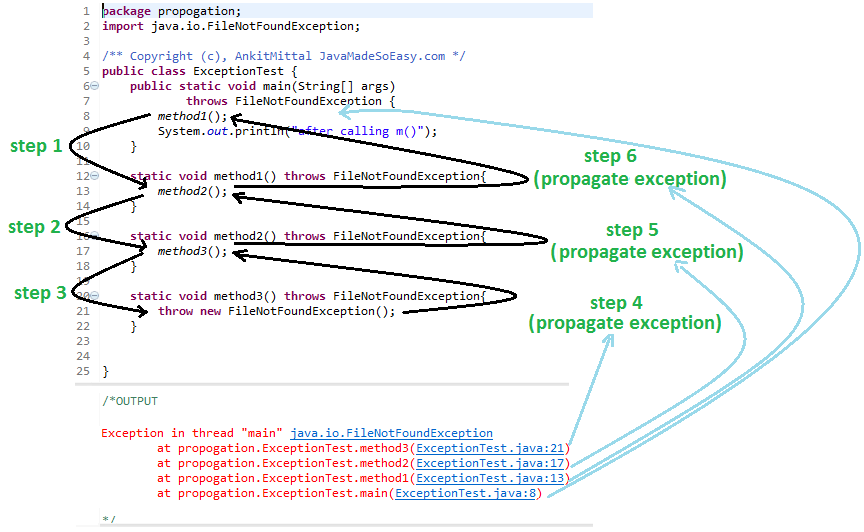
* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***automatically propagated exception*** *to method2() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 5*** *- method2* ***automatically propagated exception*** *to method1() [because, unchecked exceptions are propagated* ***automatically****]*
* ***step 6*** *- method2* ***automatically propagated exception*** *to main() [because, unchecked exceptions are propagated* ***automatically****]*

*main()* ***automatically propagated exception*** *to* ***JVM*** *[because, unchecked exceptions are propagated* ***automatically****]Lets see how* [***stack***](http://javamadesoeasy.com/2015/01/stacks.html)*of methods is formed >*

In the above program, stack is formed and an exception is first thrown from the top of the stack [ **method3()** ] and it remains uncaught there, and starts coming down the stack to previous methods to **method2()**,then to **method1()**,than to **main()** and it remains uncaught throughout.

exception remains uncaught even after reaching bottom of the stack [ **main()** ] so it is propagated to JVM and ultimately program is terminated by throwing exception [ as shown in output ]. *Propagating* [***checked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception (FileNotFoundException) using throws keyword >*

For **propagating checked** exceptions method must throw exception by using [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html)keyword.



***Now, i’ll be explaining you how checked exception*** *was* ***propagated****.*

***Let’s see step by step what happened in above program >***

* ***JVM*** *called main method*
* ***step 1*** *- main called method1()*
* ***step 2*** *- method1 called method2()*
* ***step 3*** *- method2 called method3()*
* ***step 4*** *- method3* ***propagated exception*** *to method2() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 5*** *- method2* ***propagated exception*** *to method1() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* ***step 6*** *- method2* ***propagated exception*** *to main() using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*
* *main()* ***propagated exception*** *to* ***JVM*** *using* ***throws*** *keyword****.****[because, checked exceptions are not propagated* ***automatically****]*

Difference between [**multiple catch block**](http://www.javamadesoeasy.com/2015/05/multiple-catch-block-in-java.html) and [**multi catch syntax**](http://www.javamadesoeasy.com/2015/05/catch-block-and-automatic-resource.html) *in java >*

|  |  |  |
| --- | --- | --- |
|  | **multiple catch block** | **multi catch syntax** |
| 1 | multiple catch blocks were introduced in prior versions of Java 7 and does not provide any automatic resource management. | **multi catch syntax was introduced in** java 7 for improvements in multiple exception handling which helps in **automatic resource management.** |
| 2 | Here is the syntax for writing **multiple catch block**>   |  | | --- | | **try**{  //code . . . . .  }**catch**(**IOException** ex1){  //code . . . . .  } **catch**(**SQLException** ex2){  //code . . . . .  } | | Here is the **multi catch syntax**>   |  | | --- | | **try**{  //code . . . . .  }**catch**(IOException **|** SQLException ex){  //code . . . . .  } |   We could separate different exceptions using **pipe** ( **|** ) |
| 3 | For catching IOException and SQLException we need to write **two catch block** like this >  https://lh6.googleusercontent.com/-K2qsORyl4iyZSQl1ZeuZHABUIsT0isJ1SK8kmSHVHXDSa-SUDq8SlUNC57NIMNsXrJ6bosIOE-W8Vh9AlcAkGNczBI7fxvz6Mhsq0bv6OjHTjF0M-DBkK5ZiHUNRhAMuOBZA-M | with the help of multi catch syntax we can catch IOException and SQLException in one catch block using **multi catch syntax** like this >  https://lh4.googleusercontent.com/ZSbaj7zBAA_FBhbJEw6Ig99fXl2EvMXf0Vdt9lu5ZwQ6VLACJBtRQNtzzcpW1o1_C3nh09MBdQ5UJpNV1suE8uEehkgqALdftglLUUfa2fPTygxihPYjVYltVBecmnCsk1K3yw4 |
|  | **When multiple catch blocks** are used , first catch block could be subclass of Exception class handled in following catch blocks like this >  IOException is subclass of Exception.https://lh3.googleusercontent.com/2AbC14EGFY7bvxgqRf6OI86ozSKcOJSRsqzzONNVqHQlSz3lxBojLnLhIw9TPe2tLlKHy52dW7e2xGPHjnc3Nb6CwXQE7WmGlqrpBInsM6oqtEfhFWwMhT5sIFHoKrbo_46ioR4 | If **Multi catch syntax** is used to catch subclass and its superclass than compilation error will be thrown.  IOException and Exception in **multi catch syntax** will cause compilation error “The exception **IOException** is already caught by the alternative **Exception**”.  https://lh6.googleusercontent.com/_vCmGxqWFe9nGl3xf0flEI3ORQtECF4GjJIDOL0pbc1ZUUclFmXsGgYisHmuSixHSptpJu74_qReZsEXD_2HxTTW8-cG7UlOvn5So3BQYibKXR8S1dgjNgMOHw8itGa_9pEz1Nc  **Solution >**  We must use only **Exception** to catch its subclass like this >  https://lh5.googleusercontent.com/Feb4kItN018HbRv9UIMeSizbLMOP9aJhPGvfFEiN-QEDDeUfFRii_Dk40sNfNliN7lRdfkXv6Z7n7uZPsQLk87wbij7nFqr6v4xQGs7FoFfKX950_WswEBPQL_kQsEpe4togesU |
| 5 | Does not provide such features. | *Features of* ***multi catch syntax*** *>*   * Has **improved way of catching multiple** [**exceptions**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html)**.** * This syntax does **not looks clumsy**. * **Reduces developer efforts** of writing multiple catch blocks. * Allows us to **catch more than one exception in one catch block**. * Helps in **automatic resource management.** |

*Difference between Final, Finally and Finalize*

|  |  |  |  |
| --- | --- | --- | --- |
|  | [***final***](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) | [***finally***](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) | [***finalize***](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html) |
| **1** | [final](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) can be applied to **variable**, **method** and **class** in java. | [finally](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) is a block. | [finalize](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html) is a method. |
| **2** | ***2.1) Final variable***  **final memberVariable**  **final local variable**  **final static variable**  **Final memberVariable** of class must be initialized at time of declaration, once initialized final memberVariable cannot be assigned a new value.  Final variables are called **constants** in java.   |  | | --- | | **class** FinalTest {  **finalint**x=1; //memberVariable/instanceVariable  } |   If constructor is defined then final memberVariable can be initialized in constructor but  once initialized cannot be assigned a new value.   |  | | --- | | **class** FinalTest {  **finalint**x; //memberVariable/instanceVariable     FinalTest() {            x = 1; //final memberVariable can be initialized in constructor.     }  } |   **Final local variable** can be left uninitialized at time of declaration and can be initialized later, but once initialized cannot be assigned a new value.   |  | | --- | | **class** FinalTest {  **void** method(){  **finalint** x; //uninitialized at time of declaration        x=1;    }  } |   **Final static variable** of class must be initialized at time of declaration or can be initialized in static block, once initialized final static variable cannot be assigned a new value.  If static block is defined then final static variable can be initialized in static block, once initialized final static variable cannot be assigned a new value.   |  | | --- | | **class** FinalTest {  **finalstaticint***x*; //static variable  **static**{ //static block  *x*=1;   }  } |   ***2.2) Final method***  **Final method** cannot be overridden, any attempt to do so will cause compilation error.  https://lh6.googleusercontent.com/8SUKN-wP12sbO_DIADiE3NVIfqdvyLJHzQjbdu4qZyN8pUmzGIuIBeFQJ1na9y5C7iNKUW9Ukqj-giaS2-_QgVppGOVIVgyFWKSQUwKo4PGhCAIi_iY9zeqAWhMCrrh8_YCGY24  Runtime polymorphism is not applicable on final methods because they cannot be inherited.  ***2.3) Final class***  **Final class** cannot be extended, any attempt to do so will cause compilation error.  https://lh4.googleusercontent.com/raufHpm6mAE2blzdHqcb0NrIjDsOzPswd8ugBRIZ3x_jgFvffoUXJ1xP8lrAgLUCKafozlj4cqqzZ8GqwZM9HkUEZkdfPY0s3vRjci4xjt2xp_biFFKVZdEvwSscpNF0tdhtziI | *try or* [*try-catch*](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) *block can be followed by finally block >*  try**-finally** block, or   |  | | --- | | **try{**  **//Code to be enclosed in try-finally block**  **}finally{**  **}** |   try-catch**-finally** block.   |  | | --- | | **try{**  **//Code to be enclosed in try-catch-finally block**  **}catch(Exception e){**  **}finally{**  **}** |   ***finally block*** *can can only exist if try or try-catch block is there, finally block can’t be used alone in java.*  ***finally block is not executed in following scenarios >***  finally is not executed when **System.exit** is called.  if in case **JVM crashes** because of some java.util.[**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html). | finalize method is called before garbage collection by JVM,  finalize method is called for any cleanup action that may be required before garbage collection.   |  | | --- | | @Override  **protectedvoid finalize() throws Throwable** {  **try** {        System.*out*.println("in   finalize() method, "                                +   "doing cleanup activity");  } **catch** (Throwable throwable) {  **throw** throwable;  }  } |   finalize() method is defined **in java.lang.Object** |
| **3** | - | finally block can only exist if try or try-catch block is there, finally block can’t be used alone in java. | We can *force early garbage collection in java* by using following methods >  **System.*gc*(); Runtime.*getRuntime*().gc();**  **System.*runFinalization*(); Runtime.*getRuntime*().runFinalization();** |
| **4** | - | finally is always executed irrespective of exception thrown. | If any uncaught [exception](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) is thrown inside finalize method -  **exception is ignored,**  **thread is terminated and**  **object is discarded.**  **Note :** Any exception thrown by the finalize method causes the finalization of this object to be halted, but is otherwise ignored. |
| **5** | - | Currently executing thread calls finally method. | JVM does not guarantee which [***daemon***](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)[thread](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) will invoke the finalize method for an object. |
| **6** | final is a keyword in java. | finally Is a keyword in java. | finalize is not a keyword in java. |

Difference between throw and throws:

|  |  |  |
| --- | --- | --- |
|  | [**throw**](http://www.javamadesoeasy.com/2015/05/throw-exception-in-java.html) | [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) |
| 1 | **throw**[keyword](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html) is used to throw an [exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) explicitly. | **throws**keyword is used to declare an exception. |
| 2 | **throw**is used **inside method**.  Example >  **staticvoid** m(){  **thrownew** FileNotFoundException();  } | **throws**is used **inmethod declaration**.  Example >  **staticvoid** m() throws FileNotFoundException{  } |
| 3 | **throw**is always **followed byinstanceof** Exception class.  Example >  **thrownew** FileNotFoundException() | **throws**is always **followed by name of Exception class**.  Example >  **throws**FileNotFoundException |
| 4 | **throw**can be used to throw **only one exception at time**.  Example >  **thrownew** FileNotFoundException() | **throws**can be used to throw **multiple exception at time**.  Example >  **throws**FileNotFoundException, NullPointerException  and many more... |
| 5 | **throw**cannot propagate exception to calling method.  https://lh3.googleusercontent.com/OUJb1vS33pbqa0dN5qRbR22JKpbAz4HVBqDemG8Cs2_LjrsXX9qN--vRgRKWZr6Zem5Rso2JSzvfAeGXlkylWnvtIjQrrDmRLfFQ6kTBi75cfykySrc-XUCjYSDPcqI4fxq_vlg | **throws**can [propagate exception](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html) to calling method.  Please see these programs to understand how exception is propagated to calling method.  [**Program 1**](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html) - Handling Exception by throwing it from m() method (using throws keyword) and handling it in try-catch block from where call to method m() was made.  [**Program 2**](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html)- Throwing Exception from m() method and then again throwing it from calling method [ i.e. main method] |

***Difference between Comparable and Comparator***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***Comparable*** | ***Comparator*** |
| 1 | Comparing instances of class | Comparable is used to compare instances of same class | Comparator can be used to compare instances of same or different classes. |
| **2** | **sorting order** | Comparable can be implemented by class which need to define a **natural ordering for its objects.**  **Example** - String, Integer, Long , [Date](http://www.javamadesoeasy.com/2015/07/creating-date-in-java-using-calendar.html) and all other wrapper classes implements Comparable. | Comparator is implemented when one wants a **different sorting order** and define custom way of comparing two instances. |
| 3 | Changes to class | For using Comparable, original Class must implement it.**Example-**  **class** Employee **implements Comparable<Employee>**For using Comparable, Employee Class must implement it, no other class can implement it.  As used in **Program 1** | Class itself can implement Comparator  or  any other class can implement Comparator. Hence avoiding modification to original class.  **Example-**  **class ComparatorName implements Comparator<Employee>**  **class ComparatorId implements Comparator<Employee>**  In above example modifications were made to **ComparatorName** and **ComparatorId.** Hence avoiding modification to Employee class.  As used in **Program 4** |
| 4 | Sorting on basis on one or many criteria | Provides sorting only on **one** criteria, **because** Comparable can be implemented by original class only. | We can use Comparator to sort class on **many** criterias **because** class itself or any other class can implement Comparator. |
| 5 | Method | compareTo method  **@Override**  **publicint compareTo(Employee obj) {**  **//sort Employee on basis of name(ascending order)**  **returnthis.name.compareTo(obj.name);**  **}**  Method compares **this** with **obj** object and returns a integer.   * positive – **this** is **greater** than **obj** * zero – **this** is **equal** to **obj**   negative – **this** is **less** than **obj** As used in **Program 1** | compare method  **@Override**  **publicint compare(Employee obj1, Employee obj2) {**  **//sort Employee on basis of name(ascending order)**  **return obj1.name.compareTo(obj2.name);**  **}**  Method compares **obj1** with **obj2** object and returns a integer.   * positive – **obj1** is **greater** than **obj2** * zero – **obj1** is **equal** to **obj2**   negative – **obj1** is **less** than **obj2** As used in **Program 3** |
| 6 | Package | **java.lang**  **java.lang** package is automatically imported by every program in java.  Hence, we need to write explicit statement for importing java.lang.Comparable. | **java.util**  We need to write explicit import statement -  **import** java.util.Comparator |
| 7 | Using **Collections.sort** | Lets say we wanna sort list of Employee,  **Collections.sort(**list**)** uses Comparable interface for sorting class.  As used in Program 1 | Lets say we wanna sort list of Employee,  **Collections.*sort*(list,new ComparatorName());**  uses Comparator interface for sorting class.  As used in Program 5 |

***Algorithm*** *used by* ***Comparator*** *for* ***sorting*** *>*

*Implement* ***Comparator*** *interface and override its* ***compare*** *method.*

*Call Collections.sort and pass* ***list*** *[i.e.****list****]as parameter.*

*Collections.sort internally calls* [*Arrays.sort*](http://www.javamadesoeasy.com/2015/04/arrayssort-to-sort-arrays-by.html)*,*

*Arrays.Sort() internally calls*[*Merge Sort*](http://javamadesoeasy.blogspot.in/2015/01/merge-sort.html)*.*

*Merge sort calls overridden* ***compare*** *method of* ***Comparator*** *interface for* ***comparison of values****.***Must know fact :**

If number of elements is less than 7 then [Insertion Sort](http://www.javamadesoeasy.com/2015/01/insertion-sort.html) is used rather than [*Merge Sort*](http://javamadesoeasy.blogspot.in/2015/01/merge-sort.html). (because in case elements are less than 7 it offers better time complexity)

**What are differences between** [**ArrayList and LinkedList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-linkedlist-similarity-and.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***ArrayList*** | **LinkedList** |
| 1 | Structure | It is index based structure.  [https://lh5.googleusercontent.com/7Z3uDNC-TEYFLsA57360frVEOStDwZNS84FzDqmCfqEefMy0N947BZKfCzbmAM02z0GhyYwOU1gT8Xd2G_k0km12kubNaTUNyk09kBZEJrt-YgVHj90gcrDeXOc_lfAF4ft3Gc4](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | A **linked list** is a data structure consisting of a group of **nodes** which together represent a sequence.  node is composed of a data and a reference (in other words, a **link**) to the next node in the sequence.  [https://lh4.googleusercontent.com/jTz0miinxjk2p_45HwqYVfCtAVHEtREm8cP-lURQR91DJHWHJL_62CKbLNLzkMu2WgDBJOS5vGnjQ3FIDLL1jCF06yJfm2w36mEKRB9PdKYhBCQeE1MQY_q3Y2GQ5sEkpbmx7bg](http://www.javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html) |
| 2 | **Resizable** | **It is Resizable-array.** | New node is created for storing new element. |
| 3 | **Initial capacity** | ArrayList is created with initial capacity of 10. | For storing every element node is created, so linkedList’s initial capacity is 0. |
| 4 | Ensuring **Capacity**/ resizing. | ArrayList is created with initial capacity of 10.  ArrayList’s size is **increased by 50%** i.e. after resizing it’s size become 15. | For storing every element node is created, so linkedList’s initial capacity is 0, it’s size grow with addition of each and every element. |
| 5 | RandomAccess interface | ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) | It does not implement RandomAccess interface. |
| 6 | AbstractList and AbstractSequentialList | ArrayList extends AbstractList (abstract class) which provides implementation to  List interface to minimize the effort required to implement this interface backed by RandomAccess interface. | LinkedList extends AbstractSequentialList (abstract class), AbstractSequentialList extends AbstractList.  In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially. |
| 7 | How **get(index)** method works?  (Though difference has been discussed briefly in above 2 points but in this in point we will figure difference in detail.) | Get method directly gets element on specified index. Hence, offering O(1) complexity. | Get method iterates on nodes sequentially to get element on specified index. Hence, offering O(n) complexity. |
| **8** | **When to use** | **When get operations is more frequent than add and remove operations.** | **When add and remove operations are more frequent than get operations.** |
| **9** | **Complexity** offered by methods are different | |  |  |  | | --- | --- | --- | | Operation/ method | Worst case | Best case | | *add* | O(n), when array is full it needs restructuring,  operation runs in *amortized constant time.* | O(1), when array does not need any restructuring. | | *remove* | O(n), when removal is done from between restructuring is needed. | O(1), when removal is done at last position, no restructuring is needed. | | *Get* | O(1), it is index based structure. So, complexity of  get operation is always done in O(1). | O(1) it is index based structure. So, complexity of  get operation is always done in O(1). | | *Set* | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | | *iterator* | O(n), because iteration is done over each and every element. | O(n), because iteration is done over each and every element. | | *listIterator* | O(n), its same as iterator. | O(n), its same as iterator. | | *enumeration* | O(n), its same as iterator. | O(n), its same as iterator. | | |  |  |  | | --- | --- | --- | | Operation/ method | Worst case | Best case | | *add(E element)* | **O(1),** Adds specified *element* to the end of LinkedList. | **O(1),** Adds specified *element* to the end of LinkedList. | | *add(int index, E element)* | **O(n)**, because iteration is done on all elements one by one to find out specified index.  Current *element* is placed at specified *index* and one is added to indices of subsequent elements on right. | **O(n)** | | *addFirst(E element)* | **O(1)** | **O(1)** | | *addLast(E element)* | **O(1)** | **O(1)** | |  |  |  | | *remove()* | **O(1),** Method retrieves and removes the first element (head) of this list. | **O(1)** | | *remove(int index)* | **O(n)**, because iteration is done on all elements one by one to find out specified index.  one is subtracted from indices of subsequent elements on right. | **O(n)** | | *remove(Object object)* | **O(n)**, because iteration is done on all elements one by one to find out specified object.  one is subtracted from indices of subsequent elements on right. | **O(n)** | | *removeFirst()* | O(1) | O(1) | | *removeLast()* | O(1) | O(1) | | iterator | O(n), because iteration is done over each and every element. | O(n), because iteration is done over each and every element. | | listIterator | O(n), its same as iterator. | O(n), its same as iterator. | | enumeration | O(n), its same as iterator. | O(n), its same as iterator. | |

**What are differences between** [**ArrayList and Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***ArrayList*** | ***Vector*** |
| 1 | synchronization | It is **not synchronized**  (because 2 threads on same ArrayList object can access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/consequence-of-using-arraylist-in.html)to show consequence of using ArrayList in multithreading environment.  In the program we will implement our own arrayList. | It is **synchronized** (because 2 threads on same Vector object cannot  access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/advantage-of-using-vector-in.html)to show advantage of using Vector in multithreading environment.  In the program we will implement our own vector. |
| 2 | Performance | ArrayList is not synchronized, hence its operations are **faster** as compared to Vector. | Vector is synchronized, hence its operations are **slower** as compared to ArrayList.  If we are working not working in multithreading environment jdk recommends us to use ArrayList. |
| 3 | Enumeration | **Enumeration** is [**fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html), means any modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException. | **Enumeration** is **fail-safe**, means any modification made to Vector during iteration using Enumeration don’t throw any exception. |
| 4 | Introduced in which JDK version | It was introduced in second version of java i.e. **JDK 2.0** | It was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the List interface, hence making it a member of member of the[Java Collections Framework](http://www.javamadesoeasy.com/2015/04/collection-in-java.html). |
| 5 | Ensuring Capacity/ resizing. | ArrayList is created with initial capacity of 10.  When its full size is **increased by 50%** i.e. after resizing it’s size become 15. | Vector is created with initial capacity of 10.  Vector’s size is **increased by 100%** i.e. after resizing it’s size become 20. |
| 6 | Custom implementation | [https://lh5.googleusercontent.com/9uR4b09ULu-m-WudToWm95GDL2gmJ08wfPMosGlf2rh4MrSVjfbx3owmF-7m121PM4dDr1JBMXqKkQ5f5Md08LAOyin35qu1cncz6ASj0o7fLl_xa587eGDcIa3IaNoV6hF907MArrayList custom implementation](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | [https://lh6.googleusercontent.com/p05v8DqO4QOAM1rFgEgnMiW0GumdHttGJ2hjcsAnPevk_DOODdvEpWYPacQkQ_fEBZ0pjf_1ycJEvEHWfVYV97cfyzN3oo3RIRnZqRrvKGyheiq7F-syou5nx3t10bxG34BJuZ8](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html)  [Vector custom implementation](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html) |

**What are differences between** [**List and Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) **interface?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***List*** | ***Set*** |
| 1 | Insertion order | List is ordered collection it **maintain insertion order**. | *Most of the Set implementation* does not **maintain insertion order**.  HashSet does not maintains insertion order.Thought LinkedHashSet maintains insertion order.TreeSet is sorted by natural order. |
| 2 | Duplicate elements | List **allows to store duplicate elements**. | *Set does* ***not allow to store duplicate elements****.* |
| 3 | Null keys | It allows to store **many null keys**. | Most of the implementations allow to add only **one null.**  TreeSet does not allow to add null. |
| 4 | Getting element on specific **index** | List implementations provide get method to get element on specific index.  ArrayList, Vector, copyOnWriteArrayList and LinkedList provides -  *get(int index)*  Method returns element on specified *index*.  **Get method directly gets element on specified index. Hence, offering O(1) complexity.** | Set implementations does not provide any such get method to get element on specified index. |
| 5 | Implementing classes | Classes like ArrayList, Vector, LinkedList and copyOnWriteArrayList implements List. | Classes like HashSet, LinkedHashSet, TreeSet and CopyOnWriteArraySet implements Set. |
| 6 | listIterator | **listIterator** method returns listIterator to iterate over elements in List.  **listIterator provides** additional methods as compared to iterator like  **hasPrevious(), previous(), nextIndex(), previousIndex(), add(E element), set(E element)** | It does not provide any listIterator. |
| 7 | Structure and resizable | **List** are Resizable-array implementation of the java.util.**List** interface. | Set uses [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)for their implementation.  Hence, structure is map based and resizing depends on Map implementation.  *Example >*[***HashSet***](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html) *internally uses* [*HashMap*](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html)*.* |
| 8 | Index based structure | As **ArrayList** uses array for implementation it is index based structure. | Set is not index based structure. |

**What are differences between** [**Iterator and ListIterator**](http://www.javamadesoeasy.com/2015/04/iterator-vs-listiterator-similarity-and.html)**?**

**Answer**.

|  |  |  |
| --- | --- | --- |
|  | ***ListIterator*** | ***Iterator*** |
| 1 | **hasPrevious()**  method returns true if this listIterator has more elements when traversing the list in the reverse direction. | No such method. |
| 2 | **previous()**  returns previous element in iteration (traversing in backward direction).  if the iteration has no previous elements than NoSuchElementException is thrown. | No such method. |
| 3 | **nextIndex()**  method returns the index of the element that would be returned by a subsequent call to next() method. If listIterator is at the end of the list than method returns size of list. | No such method. |
| 4 | **previousIndex()**  method returns the index of the element that would be returned by a subsequent call to previous() method. If listIterator is at the start of the list than method returns -1. | No such method. |
| 5 | **add(E element)**  Method inserts the specified **element** into the list.  The element is inserted immediately before the element that would be returned by next (So, subsequent call to next would be unaffected), if any, and after the element that would be returned by previous (So,subsequent call to previous would return the new**element**), if any.  If the list does not contain any element than new **element** will be the sole element in the list. | No such method. |
| 6 | **set(E element)**  Method replaces the last element returned by next() or previous() method with the specified **element**. This call can be made only if neither remove nor add have been called after the last call to next or previous.  If call to set() method is followed up by any call made to remove() or add() method after next() or previous() than UnsupportedOperationException is thrown. | No such method. |
| 7 | All the implementations of [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) interface like [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes returns listIterator. | All Implementation classes of [**Collection**](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) interface’s subinterfaces like [Set and List](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) return iterator. |

**What are differences between** [**Collection and Collections**](http://www.javamadesoeasy.com/2015/04/collection-vs-collections-differences.html)**?**

**Answer**.

java.util.[***Collection***](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) ***​*** *is the* root **interface** in the ​*hierarchy of Java Collection framework​*.

The JDK does not provide any classes which directly implements Collection interface, but it  provides classes such as [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html), [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html), [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html), [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [CopyOnWriteArrayList](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html), [CopyOnWriteArraySet](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html), [ConcurrentSkipListSet](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)  which implements more specific subinterfaces like ​[Set and List​](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html). java.util.**Collections** is a utility **class** which **consists** of **static methods** that **operate on** or return **Collection**.Collections provides method like >

* **reverse** method for reversing [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html).
* **shuffle** method for shuffling elements of **List**.
* **unmodifiableCollection**, **unmodifiableSet**, **unmodifiableList**, **unmodifiableMap** methods for making **List**, [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) and [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) unmodifiable.
* **min** method to return smallest element in **Collection**.
* **max** method to return smallest element in **Collection**.
* **sort** method for sorting **List**.
* **synchronizedCollection**, **synchronizedSet**, **synchronizedList**, **synchronizedMap** methods for synchronizing **List**, **Set** and **Map** respectively**.**

**What are differences between** [**Iterator and Enumeration**](http://www.javamadesoeasy.com/2015/04/iterator-vs-enumeration-differences-and.html)**?**

**Answer**.

*Differences between Iterator and Enumeration*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | Enumeration | Iterator |
| 1 | Remove elements during iteration | Enumeration **doesn’t allows** to remove elements from collection during iteration. | Iterator **allows** to remove elements from collection during iteration by using **remove()** method. |
| 2 | Improved naming conventions in Iterator | **nextElement()**  Method Returns the next element of this enumeration if this enumeration object has at least one more element to provide.  **hasMoreElements()**  returns true if enumeration contains more elements. | **nextElement()** has been changed to **hasNext()**  And  **hasMoreElements()** has been changed to **next()** |
| 3 | Introduced in  which JDK  version | It was introduced in first version  of java i.e. ​**JDK 1.0** | It was introduced in second version  of java i.e. ​**JDK 2.0**  Iterator was introduced to replace Enumeration in the Java Collections Framework. |
| 4 | Recommendation | **Java docs** recommends iterator over enumeration**.** | **Java docs** recommends iterator over enumeration**.** |
| 5 | Enumeration and Iterator over [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html) | **Enumeration** returned by Vector is **fail-safe**, means any modification made to Vector during iteration using Enumeration don’t throw any exception. | **Iterator** returned by **ArrayList**are **fail-fast,** means any structural modification made to ArrayList during iteration will throw ConcurrentModificationException |

**How do we override equals and hashcode method, write a code to use Employee as key in HashMap? (Important)**

**Answer**. We will override equals() and hashCode() like this -By overriding equals() and hashCode() method we could use custom object as key in HashMap.1)  Check whether obj is null or not.**if(obj==null) //If obj is null, return without comparing obj& Employee class.**2)  check whether  obj is instance of Employee class or not.**if(this.getClass()!=obj.getClass()) //identifies whether obj is instance of Employee class or not.**3) Then, type cast obj into employee instance.

**Employee emp=(Employee)obj;  //type cast obj into employee instance.**

|  |
| --- |
| **@Override**  **publicbooleanequals(Object obj){**  **if(obj==null)**  **returnfalse;**  **if(this.getClass()!=obj.getClass())**  **returnfalse;**  **Employee emp=(Employee)obj;**  **return (emp.id==this.id || emp.id.equals(this.id))**  **&& (emp.name==this.name || emp.name.equals(this.name));**  **}**  **@Override**  **publicinthashCode(){**  **int hash=(this.id==null ? 0: this.id.hashCode() ) +**  **(this.name==null ? 0: this.name.hashCode() );**  **return hash;**  **}** |

**What classes should i prefer to use a key in HashMap? (Important)**

**Answer**. This question will check your in depth knowledge of Java’s Collection Api’s. we should prefer **String, Integer, Long, Double, Float, Short and any other wrapper class.** Reason behind using them as a key is that they override equals() and hashCode() method, we need not to write any explicit code for overriding equals() and hashCode() method.Let’s use Integer class as key in HashMap.

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  **publicclass** StringInMap {  **publicstaticvoid** main(String...a){           //HashMap's key=Integer class  (Integer’s api has already overridden hashCode() and equals() method for us )            Map<Integer, String> hm=**new** HashMap<Integer, String>();            hm.put(1, "data");            hm.put(1, "data OVERRIDDEN");            System.*out*.println(hm.get(1));     }  }  /\*OUTPUT  data OVERRIDDEN  \*/ |

If, we note above program, what we will see is we didn’t override equals() and hashCode() method, but still we were able to store data in HashMap, override data and retrieve data using get method.

**What are differences between** [**HashMap and Hashtable**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.HashMap*** | ***java.util.Hashtable*** |
| 1 | synchronization | It is **not synchronized**  (because 2 threads on same HashMap object can access it at same time). | It is **synchronized** (because 2 threads on same Hashtable object cannot access it at same time). |
| 2 | Performance | HashMap is not synchronized, hence its operations are **faster** as compared to Hashtable. | Hashtable is synchronized, hence its operations are **slower** as compared to HashMap.  If we are working not working in multithreading environment jdk recommends us to use HashMap. |
| 3 | Null keys and values | It allows to store **one null key** and **many null values** i.e. many keys can have null value. | It does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException). |
| 4 | Introduced in | It was introduced in second version of java i.e. **JDK 2.0** | It was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the[Java Collections Framework](http://download.oracle.com/javase/7/docs/technotes/guides/collections/index.html). |
| 5 | Recommendation | In non-multithreading environment it is recommended to use HashMap than using Hashtable. | I**n java 5 i.e. JDK 1.5**, it is **recommended** to use [ConcurrentHashMap](http://www.javamadesoeasy.com/2015/04/concurrenthashmap-in-java.html) than using Hashtable. |

**when to use** [**HashSet vs LinkedHashSet vs TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)**?**

**Answer**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Property | *HashSet* | *LinkedHashSet* | *TreeSet* |
| 1 | Insertion order | HashSet does not maintains insertion order.  Example >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **No specific order** | LinkedHashSet maintains insertion order.  Example >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **b**  **c**  **a** | TreeSet is sorted by natural order.  Example >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **a**  **b**  **c** |
| 2 | Null elements | HashSet allows to store **one null.** | LinkedHashSet allows to store **one null**. | TreeSet does **not** allows to store **any null**.  Any attempt to add null throws runtimeException (NullPointerException). |
| 3 | Data structure internally used for storing data | For storing elements it internally uses HashMap. | For storing elements it internally uses  LinkedHashMap. | For storing elements it internally uses TreeMap. |
| 4 | Introduced in which JDK version | introduced in second version of java (1.2) i.e. **JDK 2.0** | introduced in second version of java (1.4) i.e. **JDK 4.0** | introduced in second version of java (1.2) i.e. **JDK 2.0** |

**What are differences between** [**HashMap and ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)**?**

**Answer**.

|  |  |  |
| --- | --- | --- |
| Property | *HashMap* | *ConcurrentHashMap* |
| synchronization | It is **not synchronized.** | It is **synchronized**. |
| 2 threads on same Map object can access it at concurrently? | Yes, because it is not synchronized**.** | Yes.  But how despite of being synchronized, 2 threads on same *ConcurrentHashMap* object can access it at same time?  *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently. |
| Performance | We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**.  *We can synchronize hashMap by using Collections’s class* ***synchronizedMap*** *method.*   |  | | --- | | *Map synchronizedMap = Collections.****synchronizedMap****(hashMap);* |   *Now, no 2 threads can access same instance of map concurrently.*  **Hence synchronized HashMap’s performance is slower as compared to ConcurrentHashMap.**  *SCROLL BELOW FOR* ***PERFORMANCE COMPARISON*** *WITH DIAGRAMS.*  But why we didn’t compared HashMap (unSynchronized) with ConcurrentHashMap?  Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn’t cause any locking. | **ConcurrentHashMap’s performance is faster as compared to HashMap (**because it is divided into segments, as discussed in above point**).**  [*Click here*](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)*FOR* ***PERFORMANCE COMPARISON*** *WITH DIAGRAMS.* |
| Null keys and values | It allows to store **one null key** and **many null values** i.e. any key can have null value. | It does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException). |
| iterators | The iterators returned by the iterator() method of the collections returned by all three HashMap's “collection view methods" are [***fail-fast***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) *>*  *hashMap.keySet().iterator()*  *hashMap.values().iterator()*  *hashMap.entrySet().iterator()*  all three iterators are ***fail-fast*** | iterators are [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)*.*  *concurrentHashMap.keySet().iterator()*  *concurrentHashMap.values().iterator()*  *concurrentHashMap.entrySet().iterator()*  all three iterators are ***fail-safe.*** |
| **putIfAbsent** | HashMap does not contain putIfAbsent method.  ***putIfAbsent*** *method is equivalent to writing following code >*   |  | | --- | | **synchronized** (map){  **if** (!*map*.containsKey(key))  **return***map*.put(key, value);  **else**  **return***map*.get(key);  } |   [**Program to create method that provides functionality similar to putIfAbsent method of ConcurrentHashMap and to be used with HashMap**](http://www.javamadesoeasy.com/2015/04/program-to-create-method-that-provides.html) | If map does not contain specified **key**, put specified **key-value** pair in map and return null.  If map already contains specified **key**, return value corresponding to specified **key**.[**Program to use ConcurrentHashMap’s putIfAbsent method**](http://www.javamadesoeasy.com/2015/04/program-to-use-concurrenthashmaps.html) |
| Introduced in | It was introduced in **java 2 i.e. JDK 1.2**, | It was introduced in **java 5** i.e. **JDK 1.5**, since its introduction Hashtable has become obsolete, because of concurrency level its performance is better than Hashtable. |
| Implements which interface | [Map](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | Map  ConcurrentMap |
| Package | java.util | java.util.concurrent |

**When to use** [**HashMap vs Hashtable vs LinkedHashMap vs TreeMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html)**?**

**Answer**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Property** | ***HashMap*** | ***Hashtable*** | ***LinkedHashMap*** | ***TreeMap*** |
| 1 | Insertion order | HashMap does not maintains insertion order. | Hashtable does not maintains insertion order. | LinkedHashMap  maintains insertion order. | TreeMap is sorted by natural order of keys. |
| 2 | Performance | HashMap is not synchronized, hence its operations are **faster** as compared to Hashtable. | Hashtable is synchronized, hence its operations are **slower** as compared HashMap.  If we are working not working in multithreading environment jdk recommends us to use HashMap. | LinkedHashMap must be used only when we want to maintain insertion order. **Time and space overhead** is there because for maintaining order it internally uses **Doubly Linked list**. | TreeMap must be used only when we want sorting based on natural order. Otherwise sorting operations cost performance. (Comparator is called for sorting purpose) |
| 3 | Null keys and values | It allows to store **one null key** and **many null values** i.e. many keys can have null value. | It does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException). | It allows to store **one null key** and **many null values** i.e. any key can have null value. | It does **not allow to store null key but allow many null values**.  Any attempt to store null key throws runtimeException (NullPointerException). |
|  | Implements which interface | [Map](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | Map | Map | Map  SortedMap  NavigableMap |
|  | Implementation | use [**buckets**](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html) | use **buckets** | Uses [**doubly linked lists**](http://www.javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html) | Uses **Red black tree** |
|  | Complexity of put, get and remove methods | O(1) | O(1) | O(1)  **overhead** of updating **Doubly Linked list** for maintaining order it internally uses. | O(log(n)) |
| 4 | Extends Dictionary (Abstract class, which is obsolete) | No | Extends Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key) | No | No |
| 6 | Introduced in | It was introduced in second version of java i.e. **JDK 2.0** | It was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the[Java Collections Framework](http://download.oracle.com/javase/7/docs/technotes/guides/collections/index.html). | It was introduced in fourth version of java i.e. **JDK 4.0** | It was introduced in second version of java i.e. **JDK 2.0** |

**What are differences between** [**HashMap vs IdentityHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-identityhashmap-similarity.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.HashMap*** | ***java.util.IdentityHashMap*** |
| 1 | **Keys comparison***object-equality  vs reference-equality* | **HashMap** when comparing keys (and values) performs object-equality not reference-equality. In an HashMap, two keys k1 and k2 are equal if and only if (k1==null ? k2==null : k1.equals(k2)) | **IdentityHashMap** when comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if (k1==k2) |
| 2 | Initial size | Constructs a new HashMap, Its initial capacity is 16.   |  | | --- | | **new** HashMap(); | | Constructs a new IdentityHashMap, with maximum size of 21.   |  | | --- | | **new** IdentityHashMap(); | |
| 3 | Introduced in | It was introduced in second version of java i.e. **JDK 2.0** | It was introduced in fourth version of java i.e. **JDK 4.0** |
| 4 | *Program* | Program 1 shows > *comparing keys (and values) performs object-equality in place of reference-equality . In an HashMap, two keys k1 and k2 are equal if and only if* **(k1==null ? k2==null : k1.equals(k2)).** | Program 2 shows >  *comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if* **(k1==k2).** |
| 5 | overridden equals() and hashCode() method call? | [*overridden equals() and hashCode() method*](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html)are called when put, get methods are called in ***HashMap***.  As shown in Program 3. | *overridden equals() and hashCode() method* are not called when put, get methods are called in ***IdentityHashMap***.  *Because IdentityHashMap implements equals() and hashCode() method by itself and checks for reference-equality of keys.*  As shown in Program 4. |

**What is WeakHashMap in java?**

**Answer**. [WeakHashMap](http://www.javamadesoeasy.com/2015/04/weakhashmap-in-java.html) is hash table based implementation of the Map interface, with *weak keys*.

An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use. Mapping for a given key will not prevent the key from being discarded by the garbage collector, (i.e. made finalizable, finalized, and then reclaimed). When a key has been discarded its entry is removed from the map.java.util.**WeakHashMap** is implementation of the java.util.[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) interface.***The behavior of the WeakHashMap class depends upon garbage collector >***

The behavior of the WeakHashMap class depends upon garbage collector. Because the garbage collector may discard keys at any time, in WeakHashMap it may look like some unknown thread is silently removing entries. Even if you synchronize WeakHashMap instance and invoke none of its methods,

* it is possible for the **size** method to return smaller values over time,
* for **isEmpty** method to return false and then true,
* for **containsKey** method to return true and later false for a given key,
* for **get** method to return a value for a given key but later return null,
* for **put** method to return null, and
* for **remove** method to return false for a key that previously existed in the WeakHashMap.

Each key object in a WeakHashMap is stored indirectly as the referent of a weak reference. Therefore a key will be removed automatically only after the weak references to it, both inside and outside of the map, have been cleared by the garbage collector.

**What is EnumSet in java?**

**Answer**. A [EnumSet](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) is specialized **Set** implementation for use with enum types.

EnumSet all elements comes from a single enum type that is specified when the set is created. *Order of elements*

The EnumSet maintains ***natural order***(the order in which the enum constants are declared) of elements. *Iterator*

The iterator returned by the iterator method traverses the elements in their ***natural order***(the order in which the enum constants are declared).

iterator never throw ConcurrentModificationException and it may or may not show the effects of any modifications to the set that occur while the iteration is in progress.*Null elements*

Null elements are not allowed. Attempts to insert a null element will throw NullPointerException.

EnumSet is an [abstract class](http://java67.blogspot.sg/2012/09/what-is-difference-between-interface-abstract-class-java.html) and it provides two concrete implementations, java.util.RegularEnumSet and java.util.JumboEnumSet. Main difference between RegularEnumSet and JumboEnumSet is that former uses a long variable to store elements while later uses a long[] to store its element. Since RegularEnumSet uses long variable, which is a 64 bit data type, it can only hold that much of element. That's why when an empty EnumSet is created using EnumSet.noneOf() method, it choose RegularEnumSet if key universe (number of enum instances in Key Enum) is less than or equal to 64 and JumboEnumSet if key universe is more than 64. Here is the code which does that :

/\*\*

\* Creates an empty enum set with the specified element type.

\*

\* **@param** elementType the class object of the element type for this enum

\* set

\* **@throws** NullPointerException if <tt>elementType</tt> is null

\*/

**publicstatic**<E **extends** Enum<E>> EnumSet<E>noneOf(Class<E> elementType) {

Enum[] universe = *getUniverse*(elementType);

**if** (universe == **null**)

**thrownew** ClassCastException(elementType + " not an enum");

**if** (universe.length<= 64)

**returnnew** RegularEnumSet<>(elementType, universe);

**else**

**returnnew** JumboEnumSet<>(elementType, universe);

}

Though it's pretty low level implementation detail, it's good to know about it to impress Interviewer, if you happened to ask same question in your interview.  
  
Now let's recap some of the important properties of EnumSet in Java  
  
1) EnumSet is not thread-safe, which means if it needs to be externally synchronized, when multiple thread access it and one of them modifies the Collection.  
  
2) EnumSet can not be used to store any other object except Enum, at the same time you can not store instances of two different Enum.   
  
3) EnumSet doesn't allow Null elements.  
  
4) EnumSet Iterators are fail-safe in nature.

**package** com.test;

**import** java.util.EnumSet;

**import** java.util.Set;

**publicclass** EnumSetDemo {

/\*\*

\*

\* **@author**aayushraj

\*

\*/

**privateenum** Color {

*RED*(255, 0, 0), *GREEN*(0, 255, 0), *BLUE*(0, 0, 255);

**privateint**r;

**privateint**g;

**privateint**b;

**private** Color(**int** r, **int** g, **int** b) {

**this**.r = r;

**this**.g = g;

**this**.b = b;

}

**publicint**getR() {

**returnthis**.r;

}

**publicint**getG() {

**return**g;

}

**publicint**getB() {

**return**b;

}

}

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

EnumSet<Color> yellow=EnumSet.*of*(Color.*RED*,Color.*GREEN*);

*drawLine*(yellow);

EnumSet<Color> white=EnumSet.*of*(Color.*RED*,Color.*GREEN*,Color.*BLUE*);

*drawLine*(white);

EnumSet pink=EnumSet.*of*(Color.*RED*,Color.*BLUE*);

*drawLine*(pink);

}

**publicstaticvoid** drawLine(Set<Color> colors){

**for**(Color c: colors){

System.*out*.println("Drawning line in color --> "+c);

}

}

}

**Question 22. What is EnumMap in java?**

**Answer**. A [**EnumMap**](http://www.javamadesoeasy.com/2015/04/enummap-in-java-with-program.html) is specialized [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) implementation for use with enum type keys.

EnumMap all keys comes from a single enum type that is specified when the set is created. *Order of keys*

The EnumMap maintains ***natural order***(the order in which the enum constants are declared) of keys.*Iterator*

The iterator returned by the iterator method traverses the elements in their ***natural order* of keys**(the order in which the enum constants are declared).

iterator never throw ConcurrentModificationException and it may or may not show the effects of any modifications to the map that occur while the iteration is in progress.*Null allowed?*

Null keys are not allowed. Attempts to insert a null key will throw NullPointerException.

Null values are allowed.

**import** java.util.EnumMap;

**import** java.util.Iterator;

/\*\*

\* Java program to demonstrate How to use EnumMap in Java If Key Object is Enum

\* than it’s best to EnumMap to get better performance. Most of IDE like

\* Netbeans and Eclipse suggest you to use EnumMap instead of HashMap or any

\* other Map implementation when key object is Enum.

\*

\* **@author**Javarevisited

\*/

**publicclass**EnumMapExample {

**publicenum** STATE {

*NEW*, *RUNNING*, *WAITING*, *FINISHED*;

}

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

// Java EnumMap Example 1: creating EnumMap in java with key as enum

// type STATE

EnumMap<STATE, String> stateMap = **new** EnumMap<STATE, String>(

STATE.**class**);

// Java EnumMap Example 2:

// putting values inside EnumMap in Java

// we are inserting Enum keys on different order than their natural

// order

stateMap.put(STATE.*RUNNING*, "Program is running");

stateMap.put(STATE.*WAITING*, "Program is waiting");

stateMap.put(STATE.*NEW*, "Program has just created");

stateMap.put(STATE.*FINISHED*, "Program has finished");

// Java EnumMap Example 3:

// printing size of EnumMap in java

System.*out*.println("Size of EnumMap in java: " + stateMap.size());

// Java EnumMap Example 5:

// printing Java EnumMap , should print EnumMap in natural order

// of enum keys (order on which they are declared)

System.*out*.println("EnumMap: " + stateMap);

// Java EnumMap Example 5:

// retrieving value from EnumMap in java

System.*out*.println("EnumMap key : " + STATE.*NEW* + " value: "

+ stateMap.get(STATE.*NEW*));

// Java EnumMap Example 6:

// Iterating over Java EnumMap

Iterator<STATE> enumKeySet = stateMap.keySet().iterator();

**while** (enumKeySet.hasNext()) {

STATE currentState = enumKeySet.next();

System.*out*.println("key : " + currentState + " value : "

+ stateMap.get(currentState));

}

// Java EnumMap Example 7: checking if EnumMap contains a particular key

System.*out*.println("Does stateMap has :" + STATE.*NEW* + " : "

+ stateMap.containsKey(STATE.*NEW*));

// Java EnumMap Example 8: checking if EnumMap contains a particular

// value

System.*out*.println("Does stateMap has :" + STATE.*NEW* + " : "

+ stateMap.containsValue(**null**));

}

}

1. All keys used in EnumMap must be  from same [Enum type](http://javarevisited.blogspot.sg/2012/04/what-is-bounded-and-unbounded-wildcards.html) which is specified while creating EnumMap in Java. For example if you can not use different enum instances from two different enum.

2. EnumMap is [ordered collection](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) and they are maintained in the natural order of their keys( natural order of keys means  the order on which enum constant are declared inside enum type ). you can verify this while Iterating over an EnumMap in Java.

3. Iterators of EnumMap are [fail-fast Iterator](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) , much like of [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) and doesn't throw ConcurrentModificationException and may not show effect of any modification on EnumMap during Iteration process.

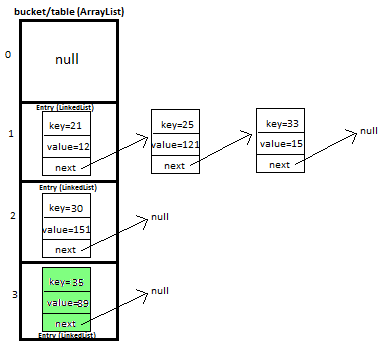
4. You can not insert null keys inside EnumMap in Java.  EnumMap doesn't allow null key and throw [NullPointerException](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html), at same time null values are permitted.

5. EnumMap is not synchronized and it has to be [synchronized](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) manually before using it in a concurrent or multi-threaded environment. like synchronized Map in Java  you can also make EnumMap synchronized by using Collections.synchronizedMap() method and as per javadoc this should be done while creating EnumMap in java to avoid accidental non synchronized access.

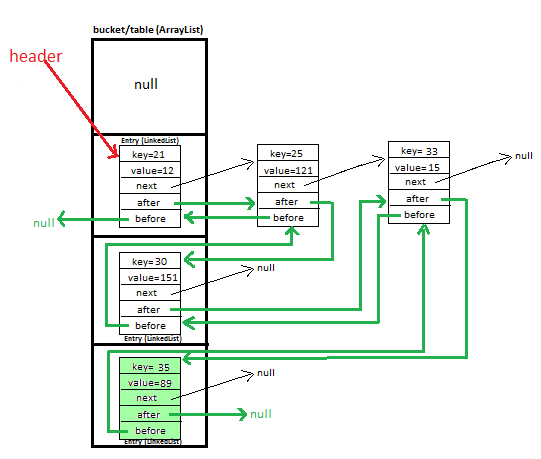
6. EnumMap is likely give better performance than HashMap in Java. So prefer EnumMap if you are going to use enum keys.

**How to implement own HashMap in java?**

**Answer**.

[](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html)

**Answer**.

[](http://javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html)

**What do you mean by fail-fast and fast-safe? What is ConcurrentModificationException?**

**Answer**.

Iterator returned by few Collection framework Classesare **fail-fast,** means any structural modification made to these classes during iteration will throw ConcurrentModificationException.

Some important classes whose returned iterator is **fail-fast >**

* [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)
* [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)
* [**vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)

[**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)Iterator returned by few Collection framework Classes are **fail-safe,** means any structural modification made to these classes during iteration won’t throw any Exception.

Some important classes whose returned iterator is **fail-safe >**

* [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)
* [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)
* [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)

**What is difference between** [**Comparable and Comparator**](http://www.javamadesoeasy.com/2015/04/comparable-vs-comparator-differences.html)**? How can you sort List?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***Comparable*** | ***Comparator*** |
| 1 | Comparing instances of class | Comparable is used to compare instances of same class | Comparator can be used to compare instances of same or different classes. |
| **2** | **sorting order** | Comparable can be implemented by class which need to define a **natural ordering for its objects.**  **Example** - String, Integer, Long , [Date](http://www.javamadesoeasy.com/2015/07/creating-date-in-java-using-calendar.html) and all other wrapper classes implements Comparable. | Comparator is implemented when one wants a **different sorting order** and define custom way of comparing two instances. |
| 3 | Changes to class | For using Comparable, original Class must implement it.**Example-**  **class** Employee **implements Comparable<Employee>**For using Comparable, Employee Class must implement it, no other class can implement it.  As used in **Program 1** | Class itself can implement Comparator  or  any other class can implement Comparator. Hence avoiding modification to original class.  **Example-**  **class ComparatorName implements Comparator<Employee>**  **class ComparatorId implements Comparator<Employee>**  In above example modifications were made to **ComparatorName** and **ComparatorId.** Hence avoiding modification to Employee class.  As used in **Program 4** |
| 4 | Sorting on basis on one or many criteria | Provides sorting only on **one** criteria, **because** Comparable can be implemented by original class only. | We can use Comparator to sort class on **many** criterias **because** class itself or any other class can implement Comparator. |
| 5 | Method | compareTo method  **@Override**  **publicint compareTo(Employee obj) {**  **//sort Employee on basis of name(ascending order)**  **returnthis.name.compareTo(obj.name);**  **}**  Method compares **this** with **obj** object and returns a integer.   * positive – **this** is **greater** than **obj** * zero – **this** is **equal** to **obj**   negative – **this** is **less** than **obj** As used in **Program 1** | compare method  **@Override**  **publicint compare(Employee obj1, Employee obj2) {**  **//sort Employee on basis of name(ascending order)**  **return obj1.name.compareTo(obj2.name);**  **}**  Method compares **obj1** with **obj2** object and returns a integer.   * positive – **obj1** is **greater** than **obj2** * zero – **obj1** is **equal** to **obj2**   negative – **obj1** is **less** than **obj2** As used in **Program 3** |
| 6 | Package | **java.lang**  **java.lang** package is automatically imported by every program in java.  Hence, we need to write explicit statement for importing java.lang.Comparable. | **java.util**  We need to write explicit import statement -  **import** java.util.Comparator |
| 7 | Using **Collections.sort** | Let's say we wanna sort list of Employee,  **Collections.sort(**list**)** uses Comparable interface for sorting class.  As used in Program 1 | Let's say we wanna sort list of Employee,  **Collections.*sort*(list,new ComparatorName());**  uses Comparator interface for sorting class.  As used in Program 5 |

**What are differences between** [**ArrayList vs CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***ArrayList*** | **CopyOnWriteArrayList** |
| 1 | synchronization | ArrayList is not **synchronized**  (because 2 threads on same ArrayList object can access it at same time).  I have created **program** to show see consequence of using ArrayList in multithreading environment.  In the program i will implement our own arrayList. | **CopyOnWriteArrayList**is **synchronized**  (because 2 threads on same CopyOnWriteArrayList object cannot access it at same time). |
| 2 | Iterator and listIterator | Iterator and listIterator returned by ArrayList are **Fail-fast,** means any structural modification made to ArrayList during iteration using Iterator or listIterator will throw ConcurrentModificationException. | Iterator and listIterator returned by CopyOnWriteArrayList are **Fail-safe**. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by ArrayList is **fail-fast**, means any structural modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException. | **Enumeration** returned by CopyOnWriteArrayList is **fail-safe.** |
| 4 | Iterate using **enhanced for loop** | Iteration done on ArrayList using **enhanced for loop** is **Fail-fast,** means any structural modification made to ArrayList during iteration using **enhanced for loop** will throw ConcurrentModificationException. | Iteration done on CopyOnWriteArrayList using **enhanced for loop** is **Fail-safe.** |
| 5 | Performance | ArrayList is not synchronized, hence its operations are **faster** as compared to CopyOnWriteArrayList. | CopyOnWriteArrayList is synchronized, hence its operations are **slower** as compared to ArrayList. |
| 6 | Introduced in which JDK version | introduced in second version of java (1.2) i.e. **JDK 2.0** | introduced in second version of java (1.5) i.e. **JDK 5.0** |
| 7 | Package | java.util | java.util.**concurrent** |

**For more detail read :**[ArrayList vs CopyOnWriteArrayList - Similarity and Differences with program](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)**Question 39. What are differences between** [**HashSet vs CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***HashSet*** | **CopyOnWriteArraySet** |
| 1 | synchronization | HashSet is not **synchronized**  (because 2 threads on same HashSet object can access it at same time). | **CopyOnWriteArraySet**is **synchronized**  (because 2 threads on same CopyOnWriteArraySet object cannot access it at same time). |
| 2 | Iterator | Iterator returned by HashSet is **Fail-fast,** means any structural modification made to HashSet during iteration using Iterator will throw ConcurrentModificationException. | Iterator returned by **CopyOnWriteArraySet** is **Fail-safe**. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by HashSet is **fail-fast**, means any structural modification made to HashSet during iteration using Enumeration will throw ConcurrentModificationException. | **Enumeration** returned by CopyOnWriteArraySet is **fail-safe.** |
| 4 | Iterate using **enhanced for loop** | Iteration done on HashSet using **enhanced for loop** is **Fail-fast,** means any structural modification made to HashSet during iteration using **enhanced for loop** will throw ConcurrentModificationException. | Iteration done on CopyOnWriteArraySet using **enhanced for loop** is **Fail-safe.** |
| 5 | Performance | HashSet is not synchronized, hence its operations are **faster** as compared to CopyOnWriteArraySet. | CopyOnWriteArraySet is synchronized, hence its operations are **slower** as compared to HashSet. |
| 6 | Introduced in which JDK version | introduced in second version of java (1.2) i.e. **JDK 2.0** | introduced in second version of java (1.5) i.e. **JDK 5.0** |
| 7 | Package | java.util | java.util.**concurrent** |

**What are differences between** [**TreeSet vs ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***TreeSet*** | **ConcurrentSkipListSet** |
| 1 | synchronization | TreeSet is not **synchronized**  (because 2 threads on same TreeSet object can access it at same time).  I have created **program** to show see consequence of using TreeSet in multithreading environment.  In the program i will implement our own arrayList. | **ConcurrentSkipListSet**is **synchronized**  (because 2 threads on same ConcurrentSkipListSet object cannot access it at same time). |
| 2 | Iterator | Iterator returned by TreeSet is **Fail-fast,** means any structural modification made to TreeSet during iteration using Iterator will throw ConcurrentModificationException. | Iterator returned by **ConcurrentSkipListSet** is **Fail-safe**. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by TreeSet is **fail-fast**, means any structural modification made to TreeSet during iteration using Enumeration will throw ConcurrentModificationException. | **Enumeration** returned by ConcurrentSkipListSet is **fail-safe.** |
| 4 | Iterate using **enhanced for loop** | Iteration done on TreeSet using **enhanced for loop** is **Fail-fast,** means any structural modification made to TreeSet during iteration using **enhanced for loop** will throw ConcurrentModificationException. | Iteration done on ConcurrentSkipListSet using **enhanced for loop** is **Fail-safe.** |
| 5 | Performance | TreeSet is not synchronized, hence its operations are **faster** as compared to ConcurrentSkipListSet. | ConcurrentSkipListSet is synchronized, hence its operations are **slower** as compared to TreeSet. |
| 6 | Introduced in which JDK version | introduced in second version of java (1.2) i.e. **JDK 2.0** | introduced in second version of java (1.6) i.e. **JDK 6.0** |
| 7 | Package | java.util | java.util.**concurrent** |

***For more detail read :*** [**TreeSet vs ConcurrentSkipListSet - Similarity and Differences with program**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)**Question 41. What are differences between** [**TreeMap vs ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html)**?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***TreeMap*** | ***ConcurrentSkipListMap*** |
| 1 | synchronization | It is **not synchronized**  (because 2 threads on same TreeMap object can access it at same time). | It is **synchronized** (because 2 threads on same ConcurrentSkipListMap object cannot access it at same time). |
| 2 | Iterator | The iterators returned by the iterator() method of the collections returned by all three Map's “collection view methods" are *fail-fast >*   * *map.keySet().iterator()* * *map.values().iterator()* * *map.entrySet().iterator()*   all three iterators are ***fail-fast*,** means any structural modification made to TreeMap during iteration using any of 3 Iterator will throw ConcurrentModificationException. | The iterators returned by the iterator() method of the collections returned by all three Map's “collection view methods" are *fail-safe >*   * *map.keySet().iterator()* * *map.values().iterator()* * *map.entrySet().iterator()*   all three iterators are ***fail-safe.*** |
| 3 | Performance | TreeMap is not synchronized, hence its operations are **faster** as compared to ConcurrentSkipListMap. | ConcurrentSkipListMap is synchronized, hence its operations are **slower** as compared to TreeMap. |
| 4 | Introduced in | It was introduced in second version of java i.e. **JDK 2.0** | It was introduced in sixth version of java i.e. **JDK 6.0** |
| 5 | Package | java.util | java.util.**concurrent** |

**Can we use null element in TreeSet? Give reason?**

**Answer**. No, [TreeSet](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) does **not** allows to store **any null keys**.

Any attempt to add null throws runtimeException (NullPointerException).

TreeSet internally compares elements for sorting elements by natural order ([comparator may be used for sorting](http://www.javamadesoeasy.com/2015/04/program-to-sort-set-in-ascending-order_24.html), if defined at creation time)and null is not comparable, Any attempt to compare null with other object will throw NullPointerException.

**Question 44. Can we use null key in TreeMap? Give reason?**

**Answer**. No, [TreeMap](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html)**not allow to store null key.**

Any attempt to store null key throws runtimeException (NullPointerException).

TreeMap internally compares keys for sorting keys by natural order ([comparator may be used for sorting](http://www.javamadesoeasy.com/2015/04/program-to-sort-set-in-ascending-order_24.html), if defined at creation time) and null is not comparable, Any attempt to compare null with other object will throw NullPointerException.

**Question 45.  How ConcurrentHashMap works? Can 2 threads on same ConcurrentHashMap object access it concurrently?**

**Answer**. [*ConcurrentHashMap*](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently.

**Can threads read the segment locked by some other thread?**

Yes. When thread locks one segment for updation it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method), but it will be able to read the data before locking.

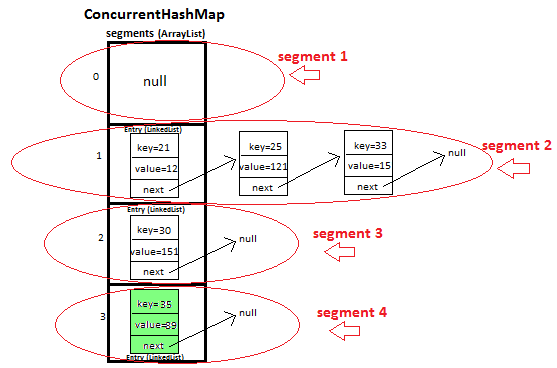
For operations such as putAll concurrent retrievals may reflect removal of only some entries.

For operations such as clear concurrent retrievals may reflect removal of only some entries.

***Segments*** *in ConcurrentHashMap with* ***diagram*** *>*

we have ConcurrentHashMap with **4 segments -**

(Diagram shows how **segments** are formed in ConcurrentHashMap)

[****](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)

**Let’s say you have to build dictionary and multiple users can add data in that dictionary? And you can use 2 Collection classes? Which Collection classes you will prefer and WHY?**

**Answer**. It’s very **important question** which test your **logical** reasoning and your ability to create robust applications in [multithreading](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) environment.We must use [**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) and [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)  >

|  |
| --- |
| **ConcurrentSkipListMap<String, TreeSet<String>> myDictionary =**  **new ConcurrentSkipListMap<String, TreeSet<String>>();** |

Store words in [**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) as key>

* keys are sorted in **natural order** (words will be sorted in natural order),
* **doesn’t allow null** keys (words can’t be null)
* **doesn’t allow duplicate** keys (words can’t be duplicate) and

[synchronized](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html), so 2 threads won’t create synchronization problems (will take care of different uses adding words concurrently)for storing meaning of word in dictionary we must use [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) as value in ConcurrentSkipListMap **because one word can have many meanings**>

* elements are sorted in **natural order** (meaning of word are sorted in natural order),
* **doesn’t allow null** elements (meaning of word can’t be null),

**doesn’t allow duplicate** elements (meaning of word can’t be duplicate)**Program for creating and using Java dictionary using Collection classes>**

|  |
| --- |
| **package** com.test.dictionary;  **import** java.util.TreeSet;  **import** java.util.concurrent.ConcurrentSkipListMap;  **publicclass** MyDictionary {  **publicstaticvoid** main(String[] args) {  **ConcurrentSkipListMap<String, TreeSet<String>> myDictionary =**  **new ConcurrentSkipListMap<String, TreeSet<String>>();**            TreeSet<String> innocentMeaning = **new** TreeSet<String>();            innocentMeaning.add("not responsible for an event yet suffering its consequences");            innocentMeaning.add("not guilty of a crime");            myDictionary.put("innocent", innocentMeaning);            TreeSet<String> appealingMeaning = **new** TreeSet<String>();            appealingMeaning.add("attractive");            appealingMeaning.add("expressing a desire for help");            myDictionary.put("appealing", appealingMeaning);            System.*out*.println(myDictionary);     }  }  /\* OUTPUT  {**appealing**=[attractive, expressing a desire for help], **innocent**=[not guilty of a crime, not responsible for an event yet suffering its consequences]}  \*/ |

**Floating point literal:**

double d=55456.99969;

float f=23.5678; //compilation error, possible loss of precision.

float g=356.385f; //OK

float g=356.385F; //OK

double d=456.78D; //D is option but OK

double g=345.39; //OK but the literal is double by default.

**Character literal:**

char a='a';

char b='@';

char letterN='\u004E';

Remember the character are just 16 bit unsigned integer under the hood. That means you can assign a number literal, assuming it will be fit into the unsigned 16 bit range(65535 or less)

char a =0x892;

char b=982;

char c=(char)70000;//cast required and 70000 is out of range

char d=(char)-98;// Rediculous but legal

char e=-29;//Possible loss of precision.

char f=70000;//Possible loss of precision.

byte a=3; //OK fit into byte

byte b=5;//OK fit into byte

byte c=a+b; //compilation error, Possible loss of precision, found : int, but required: byte //byte c=b+c;

byte c=(byte)(a+b);

But

byte b=3;

b+=7; //legal as compiler will convert this as b=(byte)(b+7);

**Casting:**

int a=100;

long b=a;//Implicit cast, as int always fit into long

float f=100.001f;

int a=(int)f;//Explicit cast, float could lose info.

double d=100L; //legal

int x=3.56;//illegal

int x=(int)3.67;//legal

**Overloading:**

* **Overloaded method must change the argument list.**
* **Overloaded method can change the return type.**
* **Overloaded method can change the access modifier.**
* **Overloaded method can declare border or newer exception.**
* **A method can be overloaded in same class or in subclass. Two method with same name in different class can be considered as overloaded if subclass inherits one version of the method and then declare another overloaded version in its calss.**

**class A{**

**public void doStuff(int x,String s){**

**}**

**}**

**class B extends A{**

**public void doStuff(int y, long s){**

**}**

**}**

* **You cannot overload a method with method differing return type alone.**
* **You cannot overload methods with the methods differing in exception specifications alone.**
* **For overload resolution to succeed, you need to define methods such that the compiler finds one exact match. If the compiler finds no matches for your call or if the matching is ambiguous, the overload resolution fails and the compiler issues an error.**

**The key benefit of the overriding is the ability to define the behavior that is specific to subclass type.**

The key benefit of the overriding is the ability to define the behavior that is specific to subclass type.

class Animal{

public eat(){

System.out.println("Generic animal");

}

}

class Hoarse extends Animal{

public eat(){

System.out.print("Hoarse eating hay, oats and gram");

}

}

public class TestAnimal{

public static void main(String [] args){

Animal a=new Animal();

Animal b=new Hoarse();

a.eat();//animal version of eat

b.eat(); //hoarse version of eat

}

}

class Animal{

public eat(){

System.out.println("Generic animal");

}

}

class Hoarse extends Animal{

public eat(){

System.out.print("Hoarse eating hay, oats and gram");

}

public buck(){

System.out.print("buck");

}

}

Remember the compiler will allow only method on class Animal to be invoke using reference of Animal.

Animal c=new Hoarse();

c.buck();//compiler error

Rules for overriding methods are as follow:

* The argument list must exactly match that of overriden method. if they do not match they can end up with the overloaded method that you did not intend.
* The return type must be same or as sub type of the return type declared in original overriden method declared in super class.
* Access level cannot be more restrictive than the overriden method.
* Access level can be less restrictive than the overriden method.
* In a same package other than private or non-final method can be overrided but in different package only public or protected method can be overrided.
* Overriding method can throw uncheck exception, regardless, if the overriden method declare the exception.
* Overriding method must not throw the checked exception that are new or boarder than overriden metho. For example a method that declare FileNotFoundException cannot be overriden by a method that declare a SQLException.
* Overriding method can throw narrower exception.
* You cannot override a method marked final.
* You cannot override a method marked static.
* If a method cannot be inherited, it cannot be override.

**Serialization**

**Question 1.What is Serialization in java?**

**Answer**. Let’s start by understanding what is Serialization, it’s most basic question which **you will have to answer almost in each and every java interview**. Serialization is process of converting **object into byte stream**.

Serialized object (byte stream) can be:

>Transferred over network.

>Persisted/saved into file.

>Persisted/saved into database.

Once, object have have been transferred over network or persisted in file or in database, we could deserialize the object and retain its state as it is in which it was serialized.

**Question 2. How do we Serialize object, write a program to serialize and deSerialize object and persist it in file (Important)?**

**Answer**. **You must be able to write Serialization code** to impress interviewer. In order to serialize object our class needs to implement **java.io.Serializable** interface. Serializable interface is **Marker interface** i.e. it **does not have any methods** of its own, **but** it **tells Jvm that object has to converted into byte stream**.

**SERIALIZATION:**

Create object of ObjectOutput and give it’s reference variable name oout and call writeObject() method and pass our employee object as parameter [**oout.writeObject(object1) ]**

|  |
| --- |
| OutputStream fout = **new** FileOutputStream("ser.txt");  ObjectOutput oout = **new** ObjectOutputStream(fout);  System.*out*.println("Serialization process has started, serializing employee objects...");  **oout.writeObject(object1);** |

**DESERIALIZATION:**

Create object of ObjectInput and give it’s reference variable name oin and call readObject() method [**oin.readObject() ]**

|  |
| --- |
| InputStream fin=**new** FileInputStream("ser.txt");  ObjectInput oin=**new** ObjectInputStream(fin);  System.*out*.println("DeSerialization process has started, displaying employee objects...");  Employee emp;  emp=(Employee)**oin.readObject();** |

**Question 3 . Difference between Externalizable and Serialization interface (Important)?**

**Answer**. Here comes the time to **impress interviewer** by differentiating Serializable and Externalizable use.

|  |  |  |
| --- | --- | --- |
|  | **SERIALIZABLE** | **EXTERNALIZABLE** |
| Methods | It is a **marker** interface it doesn’t have any method. | It’s not a marker interface.  It has method’s called **writeExternal()** and **readExternal()** |
| Default Serialization process | **YES**, Serializable provides its own **default serialization process**, we just need to implement Serializable interface. | **NO**, we need to override **writeExternal()** and **readExternal()** for serialization process to happen. |
| Customize serialization process | We **can** customize **default serialization process** by **defining following** methods in our class >**readObject()** and **writeObject()**  **Note**: We are not overriding these methods, we are defining them in our class. | Serialization process is completely customized  We need to **override** Externalizable interface’s **writeExternal()** and **readExternal()** methods. |
| Control over Serialization | It provides **less control** over Serialization as it’s not mandatory to define **readObject()** and **writeObject()** methods. | Externalizable provides you **great control** over serialization process as it is important to override  **writeExternal()** and **readExternal()** methods. |
| Constructor call during **deSerialization** | Constructor is **not** called during deSerialization. | Constructor **is called** during deSerialization. |

**Question 4. How can you customize Serialization and DeSerialization process when you have implemented Serializable interface (Important)?**

**Answer**.  Here comes the quite **challenging question**, where you could prove how strong your Serialization concepts are.We can customize **Serialization** process by defining **writeObject()**  method &**DeSerialization** process by defining **readObject()** method.Let’s customize **Serialization** process by defining **writeObject()**  method :

|  |
| --- |
| **privatevoid writeObject(ObjectOutputStream os) {**           System.*out*.println("In, writeObject() method.");  **try** {                  os.writeInt(**this**.id);                  os.writeObject(**this**.name);           } **catch** (Exception e) {                  e.printStackTrace();           }    } |

We have serialized id and name manually by writing them in file.

Let’s customize **DeSerialization** process by defining **readObject()**  method :

|  |
| --- |
| **privatevoid readObject(ObjectInputStream ois) {**           System.*out*.println("In, readObject() method.");  **try** {                  id=ois.readInt();                  name=(String)ois.readObject();           } **catch** (Exception e) {                  e.printStackTrace();           }    } |

We have DeSerialized id and name manually by reading them from file.

**Question 5. Wap to explain how can we Serialize and DeSerialize object by implementing Externalizable interface (Important)?**

**Answer**. For [serializing object by implementing Externalizable interface](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html), we need to override writeExternal() and readExternal() for serialization process to happen. For **Serialization** process override **writeExternal()**  method & for **DeSerialization** process by override **readExternal()** method.

Let’s customize **Serialization** process by overriding [**writeExternal()**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html)method :

|  |
| --- |
| **publicvoidwriteExternal**(ObjectOutput oo) **throws** IOException {           System.*out*.println("in writeExternal()");           oo.writeInt(id);           oo.writeObject(name);    } |

We have serialized id and name manually by writing them in file.

Let’s customize **DeSerialization** process by overriding [**readExternal()**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html)  method :

|  |
| --- |
| **publicvoidreadExternal**(ObjectInput in) **throws** IOException, ClassNotFoundException {           System.*out*.println("in readExternal()");  **this**.id=in.readInt();  **this**.name=(String)in.readObject();    } |

We have DeSerialized id and name manually by reading them from file.

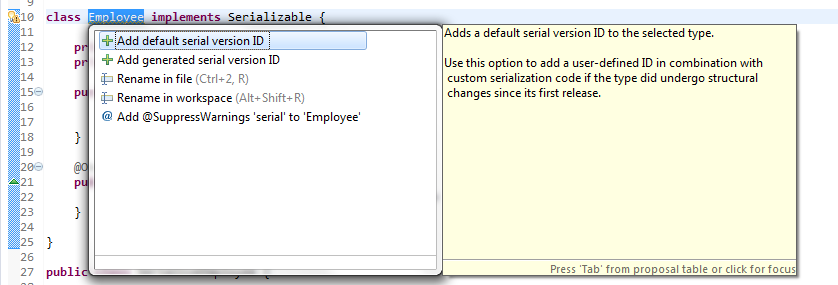
**Question 6. How can you avoid certain member variables of class from getting Serialized?**

**Answer**. Mark member variables as [**static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html)or **transient**, and those member variables will no more be a part of Serialization.

Question 7. What is serialVersionUID?

**Answer**. The serialization at 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.

We can use eclipse to generate serialVersionUID for our class (as done in below snapshot)



How to avoid **warning** ‘The serializable class Employee does not declare a static final serialVersionUID field of type long’ ?

Again answer is we can use eclipse to generate serialVersionUID for our class (as mentioned in above screenshot, click on warning button on left in line 10).

**Question 8. What will be impact of not defining serialVersionUID in class (Important)?**

**Answer**.  This is one my favourite question, i am going to discuss it in a very detailed manner. serialVersionUID is used for **version control of object**.

If we  don’t define serialVersionUID in the class, and any **modification** is made in class, then we **won’t be able to deSerialize our class** because **serialVersionUID generated by java compiler for modified class will be different from old serialized object**. And deserialization process will end up throwing **java.io.InvalidClassException**  (because of serialVersionUID mismatch)

Let’s frame another question by twisting few words in it.

*If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing* ***InvalidClassException****?*

**>**Simply we need to define **serialVersionUID** in class.

When we Deserialize class ( class which has been modified after Serialization and also class **doesn’t declare SerialVersionUID**) **InvalidClassException** is thrown.

When we Deserialize class ( class which has been modified after Serialization and also class **declare SerialVersionUID**) its gets DeSerialized **successfully**.

Let’s discuss this interesting topic in detail - Impact of not defining serialVersionUID in class and  avoiding **InvalidClassException**:

**If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing InvalidClassException?**

>Simply we need to define serialVersionUID in class.

When we Deserialize class ( class which has been modified after Serialization and also class doesn’t declare SerialVersionUID) InvalidClassException is thrown.

When we Deserialize class ( class which has been modified after Serialization and also class declare SerialVersionUID) its gets DeSerialized successfully.

Let’s discuss this interesting topic in detail with programs-

First we will serialize a class (class which implements Serialization, but we haven’t declared SerialVersionUID)

**Program 1 -** to Serialize Object (without **serialVersionUID**)**>**

**package**serDeser4AddSUID;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.io.Serializable;

/\*\* Copyright (c), Test.com \*/

/\*Author : Copyright- contents must not be reproduced in any form\*/

**class**Employee**implements** Serializable {

//we haven’t declared SerialVersionUId

**private** Integer id;

**private** String name;

**public** Employee(Integer id, String name) {

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + ", name=" + name + "]";

}

}

**publicclass**SerializeEmployee {

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

Employee object1 = **new** Employee(1, "amy");

Employee object2 = **new** Employee(2, "test");

**try** {

OutputStream fout = **new** FileOutputStream("ser.txt");

ObjectOutput oout = **new** ObjectOutputStream(fout);

System.*out*.println("Serialization process has started, serializing employee objects...");

oout.writeObject(object1);

oout.writeObject(object2);

fout.close();

oout.close();

System.*out*.println("Object Serialization completed.");

} **catch** (IOException ioe) {

ioe.printStackTrace();

}

}

}

|  |
| --- |
| /\*OUTPUT  Serialization process has started, serializing employee objects...  Object Serialization completed.  \*/ |

Then modify class by adding one field in class, but ensure that you **don’t run the Serialization process again**.

Modify the Serialized class (but don’t serialize the class again)**>**

**class** Employee **implements**Serializable {

**private** Integer id;

**private** String name;

**private** String addedField;

**public** Employee(Integer id, String name) {

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + ", name=" + name + "]";

}

}

|  |
| --- |
|  |

Now, we have added **addedField** in class which was already Serialized, let’s see in absence of SerialVersionUID whether we will be able to DeSerialize our class or not.

**Program 2 -**  to DeSerialize object - program will throw **InvalidClassException>**

**package**serDeser4AddSUID;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.ObjectInput;

**import** java.io.ObjectInputStream;

**publicclass**DeSerializeEmployee {

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

**try**{

InputStream fin=**new** FileInputStream("ser.txt");

ObjectInput oin=**new** ObjectInputStream(fin);

System.*out*.println("DeSerialization process has started, displaying employee objects...");

Employee emp;

**while**( (emp=(Employee)oin.readObject())!=**null** ){

System.*out*.println(emp);

}

fin.close();

oin.close();

}**catch**(IOException | ClassNotFoundException e){

e.printStackTrace();

}

System.*out*.println("Object deSerialization completed.");

}

}

|  |
| --- |
| /\*OUTPUT  DeSerialization process has started, displaying employee objects...  java.io.InvalidClassException: serDeser4AddSUID.Employee; local class incompatible: stream classdesc serialVersionUID = 4822384361417160410, local class serialVersionUID = 5590647880449995492  Object deSerialization completed.     at java.io.ObjectStreamClass.initNonProxy(Unknown Source)     at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)     at java.io.ObjectInputStream.readClassDesc(Unknown Source)     at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)     at java.io.ObjectInputStream.readObject0(Unknown Source)     at java.io.ObjectInputStream.readObject(Unknown Source)     at serDeser4AddSUID.DeSerializeEmployee.main(DeSerializeEmployee.java:18)  \*/ |

DeSerialization process has ended up throwing InvalidClassException.

Now, let’s see what will happen when we declare serialVersionUID in Serializable class.

**Program 3 -**to Serialize Object (with**serialVersionUID**)**>**

**package**serDeser4AddSUID;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.io.Serializable;

/\*Author : Copyright- contents must not be reproduced in any form\*/

**class** Employee **implements** Serializable {

**privatestaticfinallong***serialVersionUID* = 1L;

**private** Integer id;

**private** String name;

**public** Employee(Integer id, String name) {

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + ", name=" + name + "]";

}

}

**publicclass**SerializeEmployee {

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

Employee object1 = **new** Employee(1, "amy");

Employee object2 = **new** Employee(2, "test");

**try** {

OutputStream fout = **new** FileOutputStream("ser.txt");

ObjectOutput oout = **new** ObjectOutputStream(fout);

System.*out*.println("Serialization process has started, serializing employee objects...");

oout.writeObject(object1);

oout.writeObject(object2);

fout.close();

oout.close();

System.*out*.println("Object Serialization completed.");

} **catch** (IOException ioe) {

ioe.printStackTrace();

}

}

}

|  |
| --- |
| /\*OUTPUT  Serialization process has started, serializing employee objects...  Object Serialization completed.  \*/ |

Then modify class by adding one field in class, but ensure that you don’t run the Serialization process again.

Modify the Serialized class (but don’t serialize the class again)>

**class** Employee **implements**Serializable {

**privatestaticfinallong***serialVersionUID* = 1L;

**private** Integer id;

**private** String name;

**private** String addedField;

**public** Employee(Integer id, String name) {

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + ", name=" + name + "]";

}

}

Now, we have added addedField in class which was already Serialized, let’s see in presence of SerialVersionUID whether we will be able to DeSerialize our class or not.

**Program 4 -  to DeSerialize object - Object will be DeSerialized successfully (without ClassCastException) >**

**package**serDeser4AddSUID;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.ObjectInput;

**import** java.io.ObjectInputStream;

**publicclass**DeSerializeEmployee {

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

**try** {

InputStream fin = **new** FileInputStream("ser.txt");

ObjectInput oin = **new** ObjectInputStream(fin);

System.*out*

.println("DeSerialization process has started, displaying employee objects...");

Employee emp;

**while** ((emp = (Employee) oin.readObject()) != **null**) {

System.*out*.println(emp);

}

fin.close();

oin.close();

} **catch** (IOException | ClassNotFoundException e) {

// e.printStackTrace();

}

System.*out*.println("Object deSerialization completed.");

}

}

|  |
| --- |
| /\*OUTPUT  DeSerialization process has started, displaying employee objects...  Employee [id=1, name=amy]  Employee [id=2, name=test]  Object deSerialization completed.  \*/ |

DeSerialization process has ended up **successfully**.

**SUMMARY>**

Deserialize class ( class has been modified after Serialization and also class doesn’t declare SerialVersionUID)

>Serialize a class (class which implements Serialization, but don’t declare SerialVersionUID)

>Then modify class by adding one field in class, but ensure that you don’t run the Serialization process again.

>DeSerialization process will end up throwing InvalidClassException.

Deserialize class ( class has been modified after Serialization and also class declare SerialVersionUID)

>Now, let’s see what will happen when we declare serialVersionUID in Serializable class.

>Then modify class by adding one field in class, but ensure that you don’t run the Serialization process again.

>DeSerialization process will end successfully.

**Question 9. What are compatible and incompatible changes in Serialization process?**

**Answer**.

**Compatible Changes :**Compatible changes are those changes which **does not affect** deSerialization process even if class was updated after being serialized (provided serialVersionUID has been declared)

* **Adding new fields** - We can add new member variables in class.
* **Adding writeObject()/readObject()  methods** - We may add these methods to customize serialization process.
* **Removing writeObject()/readObject() methods** - We may remove these methods and then default customization process will be used.
* **Changing access modifier of a field** - The change to access modifiers i.e. public, default, protected, and private have no effect on the ability of serialization to assign values to the fields.
* **Changing a field from static to non static OR changing transient filed to non transient field**. - it’s like addition of fields.

**InCompatible Changes :**InCompatible changes are those changes which affect deSerialization process if class was updated after being serialized (provided serialVersionUID has been declared)

* **Deletion of fields.**
* **Changing a nonstatic field to static or  non transient field to transient field. -** it’s equal to deletion of fields.
* **Modifying the writeObject() / readObject() method** - we must not modify these method, though adding or removing them completely is compatible change.

**Question 10. What if Serialization is not available, is any any other alternative way to transfer object over network?**

**Answer**.

>We can can convert **JSON** to transfer the object. JSON is helpful in stringifying and de stringifying object.

>**Hibernate** (ORM tool) helps in persisting object as it in database and later we can read persisted object.

>We can convert object into **XML** (as done in web services) and transfer object over network.

**Question 11. Why static member variables are not part of java serialization process (Important)?**

**Answer**. Serialization is applicable on objects or primitive data types only, but [**static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html)members are **class level variables**, therefore, **different object’s of same class have same value for static member**.

So, serializing static member will consume unnecessary space and time.

Also, if modification is made in static member by any of the object, it won’t be in sync with other serialized object’s value.

**Question 12. What is significance of transient variables?**

**Answer**. Serialization is not applicable on transient variables (it helps in saving time and space during Serialization process), we **must mark all rarely used variables as transient**. We can initialize transient variables during deSerialization by customizing deSerialization process.

**Question 13. What will happen if one the member of class does not implement Serializable interface (Important)?**

**Answer**. This is classy question which will check your in depth knowledge of Serialization concepts. If any of the member does not implement Serializable than  NotSerializableException is thrown.

**Question 14. What will happen if we have used List, Set and Map as member of class?**

**Answer**. This question which will check your in depth knowledge of Serialization and Java Api’s. ArrayList, HashSet and HashMap implements Serializable interface, so if we will use them as member of class they will get Serialized and DeSerialized as well.

**Question 15. Is constructor of class called during DeSerialization process?**

**Answer**. This question which will check your in depth knowledge of Serialization and constructor chaining concepts. It depends on whether our object has implemented Serializable or Externalizable.

If **Serializable** has been implemented - constructor is **not called** during DeSerialization process.

But, if **Externalizable** has been implemented - constructor **is called** during DeSerialization process.

**Question 16 . Are primitive types part of serialization process?**

**Answer**. **Yes**, primitive types are part of serialization process. Interviewer tends to check your basic java concepts over here.

**Question 17. Is constructor of super class called during DeSerialization process of subclass (Important)?**

**Answer**. Again your basic java concepts will be tested over here. It is depends on whether our superclass has implemented Serializable or not.

If superclass **has implemented Serializable** - constructor **is not called** during DeSerialization process.

If superclass has **not implemented Serializable** - constructor **is called** during DeSerialization process.

**Question 18. What values will int and Integer will be initialized to during DeSerialization process if they were not part of Serialization?**

**Answer**.  int will be initialized to 0 and Integer will be initialized to null during DeSerialization (if they were not part of Serialization process).

**Question 19. How you can avoid Deserialization process creating another instance of Singleton class (Important)?**

**Answer**.This is another classy and very important question which will check your in depth knowledge of Serialization and Singleton concepts. I’ll prefer you must understand this concept in detail. We can simply use **readResove()** method to return same instance of class, rather than creating a new one.

Defining readResolve() method ensures that we don't break singleton pattern during DeSerialization process.

|  |
| --- |
| **private** Object readResolve() **throws** ObjectStreamException {  **return***INSTANCE*;   } |

Also define readObject() method, rather than creating new instance, assign current object to INSTANCE like done below :

|  |
| --- |
| **privatevoidreadObject**(ObjectInputStream ois) **throws** IOException,ClassNotFoundException{         ois.defaultReadObject();  **synchronized** (SingletonClass.**class**) {  **if** (*INSTANCE* == **null**) {  ***INSTANCE* = this;**         }         }   } |

**Question 20. Can you Serialize Singleton class such that object returned by Deserialization process  is in same state as it was during Serialization time (regardless of any change made to it after Serialization)  (Important)?**

**Answer**. It’s another very important question which will be important in testing your Serialization and Singleton related concepts, you must try to understand the concept and question in detail.

**YES**, we can Serialize Singleton class such that object returned by Deserialization process is in same state as it was during Serialization time (regardless of any change made to it after Serialization)

Defining readResolve() method ensures that we don't break singleton pattern during DeSerialization process.

|  |
| --- |
| **private** Object readResolve() **throws** ObjectStreamException {  **return***INSTANCE*;   } |

Also define readObject() method, rather than creating new instance, assign current object to INSTANCE like done below :

|  |
| --- |
| **privatevoidreadObject**(ObjectInputStream ois) **throws** IOException,ClassNotFoundException{         ois.defaultReadObject();  **synchronized** (SingletonClass.**class**) {  **if** (*INSTANCE* == **null**) {  ***INSTANCE* = this;**         }         }   } |

**Question 21. Purpose of serializing Singleton class OR  purpose of saving singleton state?**

**Answer**.Let’s take example of our laptop, daily eod we need to shut it down, but rather than shutting it down hibernate (save state of  laptop) is better option because it enables us to resume at same point where we leaved it, like wise serializing singleton OR saving state of Singleton can be very handy.

**Question 22. How can subclass avoid Serialization if its superClass has implemented Serialization interface (Important)?**

**Answer**. If superClass has implemented Serializable that means subclass is also Serializable (**as subclass always inherits all features from its parent class**), for avoiding Serialization in sub-class we can **define writeObject()** method and **throw NotSerializableException()** from there as done below.

|  |
| --- |
| **privatevoid writeObject(ObjectOutputStream os) throws NotSerializableException {**  **thrownew NotSerializableException("This class cannot be Serialized");**  **}** |

**Question 23. Find output of following code :**

**package**serDeser6ListSetMap;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.ObjectInput;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.io.Serializable;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Set;

/\*Author : - contents must not be reproduced in any form\*/

**class** MyClass **implements** Serializable {

**privatestaticfinallong***serialVersionUID* = 1L;

**private** List<Integer>list;

**private** Set<Integer>set;

**private** Map<Integer, Integer>map;

**public** MyClass(List<Integer> list, Set<Integer> set,

Map<Integer, Integer> map) {

**super**();

**this**.list = list;

**this**.set = set;

**this**.map = map;

}

@Override

**public** String toString() {

**return**"MyClass [list=" + list + ", set=" + set + ", map=" + map + "]";

}

}

**publicclass**SerializeEmployee {

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

List<Integer> list = **new** ArrayList<Integer>();

list.add(2);

list.add(3);

Set<Integer> set = **new** HashSet<Integer>();

set.add(4);

set.add(5);

Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();

map.put(6, 34);

map.put(7, 35);

MyClass object1 = **new** MyClass(list, set, map);

**try** {

OutputStream fout = **new** FileOutputStream("ser.txt");

ObjectOutput oout = **new** ObjectOutputStream(fout);

System.*out*

.println("Serialization process has started, serializing objects...");

oout.writeObject(object1);

fout.close();

oout.close();

System.*out*.println("Object Serialization completed.");

// DeSerialization process >

InputStream fin = **new** FileInputStream("ser.txt");

ObjectInput oin = **new** ObjectInputStream(fin);

System.*out*

.println("\nDeSerialization process has started, displaying objects...");

MyClass object = (MyClass) oin.readObject();

System.*out*.println(object);

fin.close();

oin.close();

System.*out*.println("Object DeSerialization completed.");

} **catch** (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

|  |
| --- |
|  |

**Answer**. Here intention of interviewer will be to find out whether you know that list, set and map can be serialized or not.

/\*OUTPUT

Serialization process has started, serializing objects...

Object Serialization completed.

DeSerialization process has started, dispalying objects...

MyClass [list=[2, 3], set=[4, 5], map={6=34, 7=35}]

Object DeSerialization completed.

\*/

**Question 24.  Find output of following code  (Important):**

**package**SerDeser10memberNotSer;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.OutputStream;

**import** java.io.Serializable;

**class** MyClass {}

/\*Author : - contents must not be reproduced in any form\*/

**class** Employee **implements** Serializable {

**privatestaticfinallong***serialVersionUID* = 1L;

**private** Integer id;

**private** MyClass myClass ;

**public** Employee(Integer id) {

**this**.id = id;

myClass=**new** MyClass();

}

@Override

**public** String toString() {

**return**"Employee [id=" + id + "]";

}

}

**publicclass**SerializeDeser {

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

Employee object1 = **new** Employee(8);

**try** {

OutputStream fout = **new** FileOutputStream("ser.txt");

ObjectOutput oout = **new** ObjectOutputStream(fout);

System.*out*.println("Serialization process has started, serializing objects...");

oout.writeObject(object1);

System.*out*.println("Object Serialization completed.");

fout.close();

oout.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**Answer**. Here intention of interviewer will be to find out whether you know that if any of the member does not implement Serializable than  NotSerializableException is thrown.

/\*OUTPUT

Serialization process has started, serializing objects...

java.io.NotSerializableException: SerDeser10memberNotSer.MyClass

  at java.io.ObjectOutputStream.writeObject0(Unknown Source)

  at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)

  at java.io.ObjectOutputStream.writeSerialData(Unknown Source)

  at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)

  at java.io.ObjectOutputStream.writeObject0(Unknown Source)

  at java.io.ObjectOutputStream.writeObject(Unknown Source)

  at SerDeser10memberNotSer.SerializeConstructorCheck.main(SerializeConstructorCheck.java:42)

\*/

**Question 1. What is executor framework?**

**Answer.**Executor and ExecutorService are used for  following purposes >

* creating thread,
* starting threads,
* managing whole life cycle of Threads.

Executor creates pool of threads and manages life cycle of all threads in it.

In Executor framework, **Executor** interface and  **ExecutorService**  class are most prominently used.

*Executor* interface defines very important execute() method which executes command.

*ExecutorService* interface extends **Executor** interface.

An Executor interface provides following type of methods >

* methods for managing termination and
* methods that can produce a Future for tracking progress of tasks.

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

For more information read [**Executor and ExecutorService framework in java**](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html).

**ExecutorService methods >**

**boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException**

Blocks until one of the following things happen >

* all tasks have completed execution after a shutdown request, or
* specified **timeout**elapses, or
* current thread is interrupted.

**<T> Future<T> submit(Callable<T> task)**

Submits a ***task*** for execution.

Method **returns** a Future which represents pending results of the task.

Once task is completed Future's get method will return the task's result.

**<T> Future<T> submit(Runnable task, T result)**

Submits a Runnable ***task*** for execution.

Method **returns** a Future which represents that task. Once task is completed Future's get method will return result.

**Future<?> submit(Runnable task)**

Submits a Runnable ***task*** for execution.

Method **returns** a Future which represents that task. Once task is completed Future's get method will return null.

**void shutdown()**

Initiates shutdown of executor, previously submitted tasks are executed, but no new tasks will be accepted.

**List<Runnable> shutdownNow()**

* executor shutDowns immediately,
* all actively executing tasks are stopped,
* awaiting tasks will never execute, and
* method returns list all tasks that were awaiting execution.

**boolean isTerminated()**

Method returns true if all tasks have completed following shut down.

**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException**

Executes the given tasks and returns a list of Futures holding their status.

Program to demonstrate usage of Executor and ExecutorService >

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** MyRunnable **implements** Runnable {

**int**taskNumber;

MyRunnable(**int** taskNumber) {

**this**.taskNumber = taskNumber;

}

@Override

**publicvoid** run() {

System.*out*.println(Thread.*currentThread*().getName()

+ " executing task no " + taskNumber);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\* Copyright (c), Test.com \*/

**publicclass**ExecutorServiceTest {

//nThreads number of threads will be created and started in executor.

//here we will create 2 threads.

**privatestaticint***nThreads* = 2;

//nTasks number of tasks will be executed.

//here we will execute 10 tasks.

**privatestaticint***nTasks* = 10;

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

ExecutorService executor = Executors.*newFixedThreadPool*(*nThreads*);

System.*out*.println("executor created with 2 threads.");

System.*out*.println("2 threads in executor will be used for executing 10 tasks. "

+ "So, at a time only 2 tasks will be executed");

**for** (**int** i = 1; i <= *nTasks*; i++) {

Runnable task = **new** MyRunnable(i);

executor.execute(task);

}

/\*

\* Initiates shutdown of executor, previously submitted tasks are

\* executed, but no new tasks will be accepted.

\*/

executor.shutdown();

System.*out*.println("executor has been shutDown.");

}

}

|  |
| --- |
| /\*OUTPUT  executor created with 2 threads.  2 threads in executor will be used for executing 10 tasks. So, at a time only 2 tasks will be executed  pool-1-thread-1 executing task no 1  pool-1-thread-2 executing task no 2  executor has been shutDown.  pool-1-thread-1 executing task no 3  pool-1-thread-2 executing task no 4  pool-1-thread-1 executing task no 5  pool-1-thread-2 executing task no 6  pool-1-thread-1 executing task no 7  pool-1-thread-2 executing task no 8  pool-1-thread-2 executing task no 9  pool-1-thread-1 executing task no 10  \*/ |

**Let’s discuss output in detail, to get better understanding of Executor and ExecutorService usage in program >**

Note : I have mentioned output in green text.

executor created with 2 threads.

ExecutorService executor = Executors.newFixedThreadPool(nThreads), creates

2 threads in executor.

2 threads in executor will be used for executing 10 tasks. So, at a time only 2 tasks will be executed

2 created threads in executor will be used for executing 10 tasks. So, at a time only 2 tasks will be executed.

pool-1-thread-1 executing task no 1

pool-1-thread-2 executing task no 2

executor has been shutDown.

executor.shutdown(), was called but all previously submitted tasks will be executed.

pool-1-thread-1 executing task no 3

pool-1-thread-2 executing task no 4

pool-1-thread-1 executing task no 5

pool-1-thread-2 executing task no 6

pool-1-thread-1 executing task no 7

pool-1-thread-2 executing task no 8

pool-1-thread-2 executing task no 9

pool-1-thread-1 executing task no 10

If we analyze output at runtime we will notice that at a time only 2 tasks were executed.

We must shutdown executor after executing tasks.

**Future<V>**

Future interface provides methods >

* for **returning result** of computation, wait until computation is not completed and
* for **cancelling** the computation in between.

Future Methods >

**V get() throws InterruptedException, ExecutionException;**

Method returns the result of computation, method waits for computation to complete.

**V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;**

Method waits for at most timeout time for computation to complete, and then returns the result, if available.

cancel method

method cancels the task.

**Callable<V>**

Callable interface provides method for computing a result and returning that computed result or throws an exception if unable to do so

Any class implementing Callable interface must override call() method.

what type of results Callable’s call() method can return?

The Callable<V> is a generic interface, so its call method can return generic result spcified by V.

**V call() throws Exception;**

method for computing a result.

Method returns computed result or throws an exception if unable to do so.

**How Callable and Future are related?**

If you submit a Callable object to an Executor returned object is of Future type.

|  |
| --- |
| Future<Double> futureDouble=executor.submit(new SquareDoubleCallable(2.2)); |

This Future object can check the status of a Callable call’s method and wait until Callable’s call() method is not completed.

SquareDoubleCallable is a class which implements Callable.

Program to demonstrate usage of Callable and Future >

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**class** SumIntegerCallable **implements** Callable<Integer> {

Integer n;

SumIntegerCallable(Integer n) {

**this**.n = n;

}

@Override

**public** Integer call() **throws** Exception {

Integer sum = 0;

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

sum += i;

}

**return** sum;

}

}

**class** SquareDoubleCallable **implements** Callable<Double> {

Double n;

SquareDoubleCallable(Double n) {

**this**.n = n;

}

@Override

**public** Double call() **throws** Exception {

**return**n \* n;

}

}

**publicclass**CallableFutureTest {

**privatestaticfinalint***NTHREDS* = 10;

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

ExecutionException {

ExecutorService executor = Executors.*newFixedThreadPool*(*NTHREDS*);

Future<Integer> futureInteger = executor.submit(**new** SumIntegerCallable(

4));

Future<Double> futureDouble = executor.submit(**new** SquareDoubleCallable(

2.2));

System.*out*.println("SumIntegerCallable has returned > "

+ futureInteger.get());

System.*out*.println("SquareDoubleCallable has returned > "

+ futureDouble.get());

executor.shutdown();

}

}

|  |
| --- |
| /\*OUTPUT  SumIntegerCallable has returned > 10  SquareDoubleCallable has returned > 4.840000000000001  \*/ |

In the above program - we submit a Callable object to an Executor and returned object was of Future type.

**Similarity and differences between java.util.concurrent.Callable and  java.lang.Runnable?**

**Similarity between java.util.concurrent.Callable and  java.lang.Runnable?**

Instances of class which implements callable are executed by another thread.

**Difference between java.util.concurrent.Callable and  java.lang.Runnable?**

Class implementing Callable interface must override call() method. call() method returns computed result or throws an exception if unable to do so.

Class implementing Runnable interface must override run() method.

A Runnable does not return a result and can neither throw a checked exception.

##### **Using <T> Future<T> submit(Runnable task, T result) and Future<?> submit(Runnable task) in program >**

**Let me brief you about both methods again-**

##### <T> Future<T> submit(Runnable task, T result)

Submits a Runnable task for execution. Method **returns** a Future which represents that task. Once task is completed Future's get method will return the given result.

##### Future<?> submit(Runnable task)

Submits a Runnable task for execution. Method **returns** a Future which represents that task. Once task is completed Future's get method will return null.

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**class** MyRunnable **implements** Runnable {

@Override

**publicvoid** run() {

System.*out*.println("MyRunnable's run()");

}

}

**publicclass**SubmitRunnableTest {

**privatestaticfinalint***NTHREDS* = 10;

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

ExecutionException {

ExecutorService executor = Executors.*newFixedThreadPool*(*NTHREDS*);

Future<Integer> futureInteger = executor.submit(**new** MyRunnable(), 1);

System.*out*.println("futureInteger.get() > " + futureInteger.get());

Future<?> future = executor.submit(**new** MyRunnable());

System.*out*.println("future.get() > " + future.get());

}

}

|  |
| --- |
| /\*OUTPUT  MyRunnable's run()  futureInteger.get() > 1  MyRunnable's run()  future.get() > null  \*/ |

**Let’s analyze output -**

when executor.submit(**new** MyRunnable(), 1) was called, it internally called call() method and on successful completion of MyRunnable’s run() method, futureInteger.get() returned **1**, i.e. second parameter passed in submit method.

This type of submit method could be handy when in certain scenarios we want to return status of task.

when executor.submit(**new** MyRunnable()) was called, it internally called call() method and on successful completion of MyRunnable’s run() method, futureInteger.get() returned **null.**

**Question 2. What are differences between execute() and submit() method of executor framework?**

**Answer.**

|  |  |
| --- | --- |
| **execute()** method | **submit()** method |
| **execute()** method is defined in *Executor* interface. | **submit()** method is defined in *ExecutorService* interface. |
| It can be used for executing **runnable task**. | It can be used for executing **runnable  task** or **callable task**, submitted callable returns future and Future's get method will return the task's result. |
| **Signature of execute method is  >*****void*** *execute****(Runnable*** *task****)*** | submit method has 3 forms >  *<T> Future<T>****submit****(Callable<T>****task****)*  Submits a callable ***task*** for execution.  Method **returns** a Future which represents pending results of the task.  Once task is completed Future's get method will return the task's result.  *<T> Future<T>****submit****(Runnable* ***task****, T* ***result****)*  Submits a Runnable ***task*** for execution.  Method **returns** a Future which represents that task. Once task is completed Future's get method will return ***result***.  *Future<?>****submit****(Runnable* ***task****)*  Submits a Runnable ***task*** for execution.  Method **returns** a Future which represents that task. Once task is completed Future's get method will return null. |

**Question 3.What is Semaphore in java 7?**

**Answer.**  A [**semaphore**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html) controls access to a shared resource by using permits.

* **If permits are greater than zero**, then semaphore **allow access to shared resource**.
* **If permits are zero or less than zero**, then semaphore **does not allow access to shared resource**.

These permits are sort of counters, which allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

*Semaphore has 2 constructors >*

* **Semaphore**(int ***permits***)

***permits*** is the **initial number of permits available**.

This value can be negative, in which case releases must occur before any acquires will be granted, ***permits*** is number of threads that can access shared resource at a time.

If ***permits*** is 1, then only one threads that can access shared resource at a time.

* **Semaphore**(int **permits**, boolean **fair**)

**permits** is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted.

By setting **fair** to **true**, we ensure that **waiting threads are granted a permit in the order in which they requested access**.

*Semaphore’s acquire( ) method has 2 forms :*

* void **acquire**( ) throws InterruptedException

Acquires a permit if one is available and **reduces the number of available permits by 1**.

If no permit is available then the current thread becomes dormant until

>some other thread calls release() method on this semaphore or,

>some other thread interrupts the current thread.

* void **acquire**(int **permits**) throws InterruptedException

Acquires **permits** number of permits if available and **reduces the number of available permits by permits.**

If **permits** *number of* permits are not available then the current thread becomes dormant until  one of the following things happens -

>some other thread calls release() method on this semaphore and available permits become equal to **permits** or,

>some other thread interrupts the current thread.

*Semaphore’s release( ) method has 2 forms :*

* void **release**( )

Releases a permit and **increases the number of available permits by 1**.

For releasing lock by calling release() method it’s not mandatory that thread must have acquired permit by calling acquire() method.

* void **release**(int **permits**)

Releases **permits** number of permits and **increases the number of available permits by permits.**

For releasing lock by calling release(int *permits*) method it’s not mandatory that thread must have acquired permit by calling acquire()/acquire(int permit) method.

**Question 4. How can you implement Producer Consumer pattern using Semaphore?**

**Answer.**  [**Semaphore**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html) **on producer is created with permit =1**. So, that **producer can get the permit to produce**.

**Semaphore on consumer is created with permit =0**. So, that **consumer could wait for permit to consume**. [because initially producer hasn’t produced any product]

**Producer gets permit by** calling **semaphoreProducer.acquire()** and **starts producing**, **after producing** it calls **semaphoreConsumer.release()**. So, that **consumer could get the  permit to consume**.

|  |
| --- |
| **semaphoreProducer.acquire();**  **System.*out*.println("Produced : "+i);**  **semaphoreConsumer.release();** |

**Consumer gets permit by** calling **semaphoreConsumer.acquire()** and **starts consuming**, **after consuming** it calls **semaphoreProducer.release()**. So, that **producer could get the  permit to produce**.

|  |
| --- |
| **semaphoreConsumer.acquire();**  **System.*out*.println("Consumed : "+i);**  **semaphoreProducer.release();** |

**Question 5. How can you implement your own Semaphore?**

**Answer.** [**Implementation of custom/own Semaphore in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown-semaphore.html)**.**

**Question 6. What is significance of atomic classes in java 7?**

**Answer.** Java provides some classes in [**java.util.concurrent.atomic**](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html) which offers an alternative to the other [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html).

*Classes found in* ***java.util.concurrent.atomic*** *are >*

* **AtomicInteger**,
* **AtomicLong,** and

**AtomicBoolean.***Methods provided by these classes >*

* **get( ),**
* **set( ),**
* **getAndSet( )**,
* **compareAndSet( ), and**
* **decrementAndGet( ).**

In [multithreading](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) environment we can use these classes without any explicit synchronization, as all these classes are [thread safe](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html).For more information on atomic read [**Atomic operations in java**](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html).

**Question 7. What are Future and Callable? How are they related?**

**Answer.**

[*Future<V>*](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) interface provides methods >

* for **returning result** of computation, wait until computation is not completed and
* for **cancelling** the computation in between.

*Future Methods >*

*V* ***get****()* method returns the result of computation, method waits for computation to complete.

*cancel method* cancels the task.

[*Callable<V>*](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) interface provides method for computing a result and returning that computed result or throws an exception if unable to do so

Any class implementing Callable interface must override ***call****()* method for computing a result.

Method returns computed result or throws an exception if unable to do so.

what type of results Callable’s call() method can return?

The Callable<V> is a generic interface, so its call method can return generic result specified by *V*.

How Callable and Future are related?

If you submit a Callable object to an Executor returned object is of Future type.

|  |
| --- |
| Future<Double> futureDouble=executor.submit(**new** SquareDoubleCallable(2.2)); |

where, SquareDoubleCallable is a class which implements Callable.

This Future object can check the status of a Callable call’s method and wait until Callable’s call() method is not completed.

For more information read [**Executor and ExecutorService framework in java**](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html).

**Question 8. Similarity and differences between java.util.concurrent.Callable and  java.lang.Runnable?**

**Answer.**

Similarity between java.util.concurrent.Callable and  java.lang.Runnable?

Instances of class which implements callable are executed by another thread.

Difference between java.util.concurrent.Callable and  java.lang.Runnable?

Class implementing Callable interface must override call() method. call() method returns computed result or throws an exception if unable to do so.

Class implementing Runnable interface must override run() method.

A Runnable does not return a result and can neither throw a checked exception.

**Question 9. What is CountDownLatch?**

**Answer.**  There might be situation where we might like our thread to wait until one or more threads completes certain operation.

A [CountDownLatch](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) is initialized with a given ***count*** .

***count*** specifies the number of events that must occur before latch is released.

Every time a event happens ***count*** is reduced by 1. Once count reaches 0 latch is released.

*CountDownLatch’s  constructor >*

* **CountDownLatch**(int ***count***)

CountDownLatch is initialized with given ***count***.

***count*** specifies the number of events that must occur befor latch is released.

*CountDownLatch’s await() method has 2 forms :*

* void **await**( ) throws InterruptedException

Causes the current thread to wait until  one of the following things happens-

* latch ***count*** has down to reached 0, or
* unless the thread is interrupted.
* boolean **await**(long **timeout**, TimeUnit **unit**)

Causes the current thread to wait until  one of the following things happens-

* latch ***count*** has down to reached 0,
* unless the thread is interrupted, or
* specified **timeout** elapses.

*CountDownLatch’s countDown() method :*

* void **countDown**( )

Reduces latch ***count*** by 1.

If ***count*** reaches 0, all waiting threads are released.

Read more about [**CountDownLatch in java**](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html).

**Question 10. Where can you use CountDownLatch in real world?**

**Answer.**  When you go in amusement park, you must have seen on certain rides there is mandate that at least 3 people (**3 is count**) should be there to take a ride. So, ride keeper (**ride keeper is main thread**) waits for 3 persons (**ride keeper has called await()**).

Every time a person comes count is reduced by 1 (**let’s say every person is calling countDown() method**). Ultimately when 3 persons reach count becomes 0 & wait for ride keeper comes to end.  
**Question 11. How can you implement your own CountDownLatch in java?**

**Answer.** Please see [**Implementation of custom/own CountDownLatch in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown_31.html).

**Question 12.What is CyclicBarrier?**

**Answer.**  There might be situation where we might have to trigger event only when one or more threads completes certain operation.

**2 or more threads wait for each other to reach a common barrier point**. When all **threads** have **reached** common **barrier point** (i.e. when all threads have called await() method) >

* **All waiting threads are released**, and
* **Event can be triggered** as well.

[*CyclicBarrier’s*](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) *constructor >*

* **CyclicBarrier(**int **parties)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released.

* **CyclicBarrier(**int **parties,** Runnable **barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction** (event)is triggered.

*CyclicBarrier’s await() method has 2 forms :*

* int **await**() throws InterruptedException, BrokenBarrierException

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

* The last thread to call arrive(i,.e. call await() method), or
* Some other thread interrupts the current thread, or
* Some other thread interrupts one of the other waiting threads, or
* Some other thread times out while waiting for barrier, or
* Some other thread invokes reset() method on this cyclicBarrier.
* int **await**(long **timeout**, TimeUnit **unit**) throws InterruptedException, BrokenBarrierException, TimeoutException

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

* The last thread to call arrive(i,.e. call await() method), or
* The specified **timeout** elapses, or
* Some other thread interrupts the current thread, or
* Some other thread interrupts one of the other waiting threads, or
* Some other thread times out while waiting for barrier, or
* Some other thread invokes reset() method on this cyclicBarrier.

Read more about [**CyclicBarrier in java**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html).

**Question 13. Why is CyclicBarrier cyclic?**

**Answer.**  The barrier is called *cyclic* because [CyclicBarrier](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) can be reused after -

* All the waiting threads are released and
* event has been triggered.

**Question 14. Where could we use** [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) **in real world?**

**Answer.** Let’s say 10 friends (**friends are threads**) have planned for picnic on place A (Here **place A is common barrier** point). And they all decided to play certain game (**game is event**) only on everyones arrival at place A. So, all 10 friends must wait for each other to reach place A before launching event.

Now, when all **threads** have **reached** common **barrier point** (i.e. all friends have reached place A) >

* **All waiting threads are released**(All friends can play game), and

**Event can be triggered** (they will start playing game).**Question 15. How can you implement your own** [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) **in java?**

**Answer.** Please see [**Implementation of custom/own CyclicBarrier in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown_39.html).

**Question 16. Similarity and Difference between CyclicBarrier and CountDownLatch in Java?**

**Answer.**

1. [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) and [**CountDownLatch**](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) **are similar because** they wait for specified number of thread to reach certain point and make count/parties equal to 0. But,

for completing wait in CountDownLatch specified number of threads must call **countDown()** method.

for completing wait in CyclicBarrier specified number of threads must call **await()** method.

Let’ see there constructor’s >

|  |  |
| --- | --- |
| **CountDownLatch**(int ***count***)  CountDownLatch is initialized with given ***count***.  ***count*** specifies the number of events that must occur before latch is released. | **CyclicBarrier(**int **parties)**  New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released. |

1. **CyclicBarrier** can be **awaited repeatedly**, but **CountDownLatch** can’t be awaited repeatedly. i.e. once count has become 0 cyclicBarrier can be used again but CountDownLatch cannot be used again.

**CyclicBarrier** can be used to trigger event, but **CountDownLatch** can’t be used to launch event. i.e. once count has become 0 cyclicBarrier can trigger event but CountDownLatch can’t.**How can cyclicBarrier launch event?**

CyclicBarrier provides constructor for triggering event.

**CyclicBarrier(**int **parties,** Runnable **barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction (event)is triggered**.

**Question 17. What is Phaser in java? Is Phaser similar to CyclicBarrier?**

**Answer.**[Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) is somewhat **similar** in functionality of [CyclicBarrier](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) and [CountDownLatch](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) but it provides more flexibility than both of them.

Phaser provides us flexibility of registering and deRegistering parties at any time.

**For registering parties**, we may use any of the following -

* constructors, or
* int register(), or
* bulkRegister().

**For deRegistering parties**, we may use any of the following -

* arriveAndDeregister()

we have methods like **getPhase()** which returns the current phase number. And

**isTerminated()** method returns **true** if phaser has been **terminated**.

**Question 18. Differences and similarity between Phaser and CyclicBarrier?**

**Answer.**  Like a [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html), a [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) can be **awaited repeatedly**.

But, in CyclicBarrier we used to register parties in constructor but Phaser provides us flexibility of registering and deRegistering parties at any time.

**Question 19.Difference between arrive() and arriveAndAwaitAdvance() method of Phaser?**

**Answer.arrive**() method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html)**does not cause current thread to wait for other registered threads to complete current phase**. That means current thread can immediately start next phase without waiting for any other registered thread to complete current phase.

But, **arriveAndAwaitAdvance**() **method causes current thread to wait for other registered threads to complete current phase.** That means current thread can proceed to next phase only when all other threads have completed current phase (i.e. by calling **arriveAndAwaitAdvance() method**).

**Question 20. When is phaser terminated?**

**Answer.**  When calling arriveAndDeregister() method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) has caused the number of registered parties to become 0. Termination can also be triggered when an **onAdvance()** method returns **true**.

**Question 21. How can you control number of phase you want to execute in Phaser?**

**Answer.**  We can override the **onAdvance( )** method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) to control number of phases which we want to execute.

Signature of onAdavance method is *boolean onAdvance(int* ***phase****, int* ***registeredParties****).*

Where, **phase** is the current phase number when we enter onAdvance() method i.e. before advancing to next phase.

**registeredParties** is the current number of registered parties

**Every Time before advancing to next phase overridden onAdvance() method is called** and returns either true or false.

If method returns **true** than **phaser isterminated** ,or

If method returns **false** then **phaser continues** and can **advance to next phase**.

**Question 22. Where could we use** [**Phaser**](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) **in real world?**

**Answer.** Software process management is done in phases.

* First phase could be **requirement gathering**,
* second could be **software development** and
* third could be **testing**.

Second phase will not start until first is not completed, like wise third phase will not start until second is not completed.

**Question 23. What is maximum number of parties that could be registered with phaser at a time ?**

**Answer.** Maximum number of parties that could be registered with phaser at a time is **65535**, if we try to register more parties **IllegalStateException** will be thrown.

**Question 24. What is exchanger in Java?**

**Answer.**  [Exchanger](http://www.javamadesoeasy.com/2015/03/exchanger-in-java.html) enables two threads to exchange their data between each other. Exchanger can be handy in solving Producer Consumer pattern where Producer and consumer threads can exchange their data.

* **exchange**(V **x**)

exchange() method enables two threads to exchange their data between each other.

**If current thread is first one to call exchange()** method then it will until one of following things happen >

* Some other thread calls exchange() method, or
* Some other thread interrupts the current thread, or

**If some other thread has already called exchanger()** method then it resumes its execution and following things happen -

* waiting thread is resumed and receives data from current thread.
* current thread receives data from that waiting thread and it returns immediately.
* V **exchange**(V **x**, long **timeout**, TimeUnit **unit**)

exchanger() method enables two threads to exchange their data between each other.**If current thread is first one to call exchange()** method then it will until one of following things happen >

* Some other thread calls exchange() method, or
* Some other thread interrupts the current thread, or
* The specified **timeout** elapses.

**If some other thread has already called exchanger()** method then it resumes its execution and following things happen -

* waiting thread is resumed and receives data from current thread.
* current thread receives data from that waiting thread and it returns immediately.

**Question 25. How can you implement Producer Consumer pattern using Exchanger?**

**Answer.**  Exchanger is created,  which will enable Producer and consumer threads to exchange their data.

Producer thread produces and called exchanger() method, now it will wait for consumer thread to call exchange() method.

Consumer thread calls exchanger() method and following things will happens >

* current thread(consumerThread) will receive data from that waiting thread(producerThread) and it returns immediately.
* waiting thread (producerThread) will resume and receive data from current thread (consumerThread).

**Question 26. How can you solve consumer producer pattern by using BlockingQueue?**

**Answer.** Now it’s time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using [wait() and notify() method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html). Generally you might wonder why interviewer's are so much interested in asking about [solving consumer producer problem using BlockingQueue](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html), answer is they want to know how strong knowledge you have about java concurrent Api’s, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.

[**BlockingQueue is a interface** and we will use its **implementation class LinkedBlockingQueue**.](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html)

Key methods for solving consumer producer pattern are >

|  |
| --- |
| **put(i);**   //used by producer to put/produce in sharedQueue.  **take();**//used by consumer to take/consume from sharedQueue. |

**Question 27. How can you implement your own LinkedBlockingQueue to solve consumer producer pattern?**

**Answer.**  Please read [Producer Consumer pattern using Custom implementation of BlockingQueue interface](http://www.javamadesoeasy.com/2015/03/producer-consumer-pattern-using-custom.html)

**Question 28. What is Lock in java?**

**Answer.** The java.util.concurrent.locks.**Locks** is a  interface and its implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.

**A lock helps in controlling access to a shared resource by multiple threads. Only one thread at a time can acquire the lock and access the shared resource.**

If a second thread attempts to acquire the lock on shared resource when it is acquired by another thread, the second thread will wait until the lock is released. In this way we can achieve [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and [race conditions](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html) can be avoided.

Read lock of a ReadWriteLock may allow concurrent access to a shared resource.

**Question 29. Explain key methods of Lock interface?**

**Answer.**

[*Lock interface*](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html) *key methods >*

*void lock()*

Acquires the lock if it is not held by another thread. And sets **lock hold count** to 1.

If current thread already holds lock then **lock hold count** is increased by 1.

If the lock is held by another thread then the current thread waits for another thread to release lock.

*void unLock()*

If the current thread is the holding the lock then the **lock hold count** is decremented by 1. If the **lock hold count** has reached 0, then the lock is released.

If **lock hold count** is still greater than 0 then lock is not released.

If the current thread is not holding the lock then IllegalMonitorStateException is thrown.*boolean tryLock()*

Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then method return false.

*boolean tryLock(long timeout, TimeUnit unit)*

*throws InterruptedException*

Acquires the lock if it is not held by another thread and returns true.  And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then current thread will wait until one of the following things happen -

* **Another thread releases lock and the lock is acquired by the current thread, or**
* **Some other thread interrupts the current thread, or**
* **The specified timeout elapses .**

**If** the **lock is acquired** then method **returns true.** And sets **lock hold count** to 1.

**If specified timeout elapses then** method return false.

*Condition newCondition()*

Method returns a Condition instance to be used with this Lock instance.

Condition instance are similar to using [**Wait(), notify() and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods.

* IllegalMonitorStateException is thrown **if this lock is not held when** any of the **Condition waiting** or **signalling methods** are called.
* **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called.
* If a **thread is interrupted while waiting** then InterruptedException will be thrown and following things will happen -
  + the **wait will be over**, and
  + **thread's interrupted status will be cleared**.
* Waiting threads are signalled in FIFO (first in first out order) order.
* When lock is  ***fair***, first lock is obtained by longest-waiting thread.

If lock is not  ***fair***, any waiting thread could get lock, at discretion of implementation.

**Question 30. Explain usage of newCondition() method of Lock interface in detail? And can it be used to implement producer consumer pattern?**

Answer.  Please read [ReentrantLock class provides implementation of Lock’s newCondition() method - description and solving producer consumer program using this method.](http://www.javamadesoeasy.com/2015/03/reentrantlock-class-provides.html)**Question 31. Explain key methods of ReentrantLock class?**

**Answer.**

[ReentrantLock class](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html) provides implementation of all Lock interface methods

*void lock()*

*void unLock()*

*boolean tryLock()*

*boolean tryLock(long timeout, TimeUnit unit)*

*void lockInterruptibly() throws InterruptedException*

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If the lock is held by another thread then the current thread waits until one of the following thing happens -

* The lock is acquired by the current thread, or
* Some other thread **interrupts the current thread**.

As soon as current thread acquires the lock it sets **lock hold count** to 1.

*int getWaitQueueLength(Condition condition)*

Method returns number of threads that may be waiting to acquire this lock.

Method is used just for monitoring purposes and not for any kind of synchronization purposes.

*boolean isHeldByCurrentThread()*

Method returns true if lock is held by current thread. Its similar to **Thread.holdsLock()** method.

**Question 32. Write Program to demonstrate usage of ReentrantLock?**

**Question 33. How can you implement your own ReentrantLock in java?**

**Question 34. What is Fork/Join Framework ?**

**Answer.**  *Fork/Join Framework has been added in* JDK 7 and is defined in the **java.util.concurrent** package.

*Fork/Join framework enables* ***parallel programming***. Parallel programming means taking **advantage two or more processors (multicore) in computers**.  Parallel programming improves program performance.

*The Fork/Join Framework also* ***improves program performance*** *in following ways >*

* Fork/Join framework makes use of multiple processors available in computer. Hence enabling parallel processing, and
* It managing whole [life cycle of Threads](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**Question 35. What is Divide-and-conquer in Fork/Join framework ?**

**Answer.** The **divide-and-conquer** strategy recursively divides a task into smaller subtasks until  subtask isn’t small enough to be solved independently.

**Question 36. What approach does ForkJoinPool uses for managing tasks?**

**Answer.**[**ForkJoinPool**](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)uses ***work-stealing approach*** *for managing threads*. Each thread in ForkJoinPool maintains a queue of tasks. If one thread’s queue is empty, it can take task from another thread. This overall improves the program/applications performance.

**Question 37. What are ForkJoinPool and ForkJoinTask?**

##### Answer.

##### [***ForkJoinPool***](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)

**ForkJoinPool** implements ExecutorService framework. The execution of **ForkJoinTasks** takes place within a **ForkJoinPool**, which also manages the execution of the tasks.*ForkJoinPool constructors >*

* ***ForkJoinPool( )***
* Creates a pool.
* **level of parallelism = number of processors available in the system**
* ***ForkJoinPool****(int* ***parallelism****)*
* The ***parallelism*** is the **level of parallelism**. Its value must be greater than 0 and must not be more than number of processors in system.

**level of parallelism** determines the number of threads that can execute simultaneously. As a number of threads are determined it also determines number of tasks that could be executed **parallely**. *ForkJoinPool important methods >*

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. **The first task started is the main task. Main task begins subtasks that are also managed by the pool**. Different methods for starting tasks have been discussed below >

* *<T> T* ***invoke****(ForkJoinTask<T>****task****)*

This method starts the ***task*** and returns the result of the ***task***.  Calling code waits until method returns.

* *void* ***execute****(ForkJoinTask<?>****task****)*

The execute() method can be used to start a ***task*** without waiting for its completion.

This method starts the ***task***. Calling code continues its execution asynchronously and does not waits for method completion like in invoke method.

* ***submit****() method comes in 4 different forms.*
* submit() method can also be used for submitting task.*int* ***getParallelism****()*
* The method returns **level of parallelism** i.e. number of processors available in the system.*void* ***shutdown****()*
* Initiates shutdown, previously submitted tasks are executed, but no new tasks will be accepted.*List<Runnable>****shutdownNow****()*
* attempts to stop all actively executing tasks,
* submitted tasks may or may not execute.
* awaiting tasks will never execute, and

##### method cancels both existing and unexecuted tasks, so it returns empty list.[**ForkJoinTask<V>**](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)

**ForkJoinTask is abstract class for tasks that run within a ForkJoinPool.**

**ForkJoinTask<V>** is an abstract class that defines a task that can be managed by a **ForkJoinPool**.

The *V* specifies the result type of the task.

**Threads managed by ForkJoinPool executes ForkJoinTasks**. Small number of threads are used to serve large number of tasks.

*ForkJoinTask* ***important methods*** *>*

**ForkJoinTask** core methods are **fork( )** and **join()**

* *ForkJoinTask<V>****fork()***  
  The fork( ) method **submits the task** for asynchronous execution, means that the thread that calls fork( ) method to submit task continues to run. Task are executed in the compute() method, which is running within a ForkJoinPool.
* *V* ***join( )***

The **join( )** method waits for task completion on which it is called. The method returns result of the task.

* *In short, about* ***fork( )*** *and* ***join( )***are used for starting one or more new tasks and then wait for them to complete.*V* ***invoke****( )*

**The invoke() method combines the functionally of fork() and join()** methods. invoke() submits the task and waits for completion of submitted task.

* The method returns result of task.*static void* ***invokeAll****(ForkJoinTask<?>****t1****, ForkJoinTask<?>****t2****)*

invokeAll() method  submits ***t1*** and ***t2*** and waits for completion of ***t1*** and ***t1***.

* *static void* ***invokeAll****(ForkJoinTask<?> …* ***tasks****)*

invokeAll() method submits list of tasks i.e. ***tasks*** and waits for completion of all tasks in list.The ***invokeAll*( )** method can only be called from within the overridden compute() method of another **ForkJoinTask**, which is running within a **ForkJoinPool**.*Some* ***other important methods*** *for checking status of submitted task -*

*boolean* ***isDone****()* method returns true if a task completes.*boolean* ***isCompletedNormally****()* method returns true if a task completed normally without cancellation or without throwing any exception.*boolean* ***isCompletedAbnormally****()* returns true if a task completed abnormally either by cancellation or by throwing any exception.*boolean* ***isCancelled****()* returns true if the task was cancelled.**Question 38. Similarity and Difference between RecursiveAction and RecursiveTask?**

**Answer.**

***Difference between RecursiveAction and RecursiveTask***

|  |  |
| --- | --- |
| *RecursiveAction* | *RecursiveTask<V>* |
| This **submits a task** and **does not return a result**. | This **submits a task** and **returns a result**. |
| Definition of compute method  *protected abstract void* ***compute()*** | *protected abstract V* ***compute()*** The *V* specifies the result type of the task. |

##### ***Similarity between RecursiveAction and RecursiveTask***

> Both **extends ForkJoinPool**.

> All **computations by tasks are performed inside compute() method**.**Question 39. How can we use Fork/Join framework in real world?**

**Answer.** We can use Fork/Join framework for calculating sum of array of 100000 or even may be more numbers. *Fork/Join framework uses* **divide-and-conquer** strategy for *enabling* ***parallel programming***. Divide-and-conquer strategy recursively divides a array into smaller subarrays until  subarray isn’t small enough to be solved independently.

Also, **ForkJoinPool** uses ***work-stealing approach*** *for managing threads*. Each thread in **ForkJoinPool** maintains a queue of tasks. If one thread’s queue is empty, it can take task from another thread. This overall improve the programs performance. Please see [program](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html) to calculate sum of array of 100000 numbers.**Question 40. Difference between synchronized and ReentrantLock?**

**Answer.**

|  |  |
| --- | --- |
| ***synchronized*** | ***ReentrantLock*** |
| **Does not provide anyfair locks**. | **provides fair locks**, when lock is fair - first lock is obtained by longest-waiting thread.  Constructor to provide fairness -  ***ReentrantLock****(boolean* ***fair****)*  Creates an instance of ReentrantLock.  When ***fair*** is set true, first lock is obtained by longest-waiting thread.  If  ***fair*** is set false, any waiting thread could get lock, at discretion of implementation. |
| **Does not provide tryLock() method or its functionality**. Thread always waits for lock. | **Provide tryLock() method. If lock is held by another thread then method return false.**  *boolean tryLock()*  Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.  If current thread already holds lock then method returns true. And increments **lock hold count** by 1.  If lock is held by another thread then method return false. |
| There is **no method for lock interruptibility**, though current thread waits until one of the following thing happens -   * The **lock is acquiredby** the **current thread**, or * Some other thread **interrupts the current thread**. | *void lockInterruptibly()*  If current thread already holds lock then method returns true. And increments **lock hold count** by 1.  If the lock is held by another thread then the current thread waits until one of the following thing happens -   * The **lock is acquiredby** the **current thread**, or * Some other thread **interrupts the current thread**.   As soon as current thread acquires the lock it sets **lock hold count** to 1. |
| **Does not provide any method to return number of threads that may be waiting to acquire this lock**. | provide *int getQueueLength()* method to return number of threads that may be waiting to acquire this lock. |
| **holdsLock()** method is used to **find out whether lock is held by current thread or not**. If current thread holds lock method returns true. | *isHeldByCurrentThread()*method is **used to find out whether lock is held by current thread or not**. If current thread holds lock method returns true. |
| Thread can hold lock on object monitor only once. | if current thread **already holds lock** then **lock hold count** is increased by 1 when lock() method is called.  method to maintain **lock hold count** -  *void lock()*  Acquires the lock if it is not held by another thread. And sets **lock hold count** to 1.  If current thread already holds lock then **lock hold count** is increased by 1. |
| Does not provide any new condition() method. | provides *newCondition()* method.  Method returns a Condition instance to be used with this Lock instance.  Condition instance are similar to using [**Wait(), notify() and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods on object.   * IllegalMonitorStateException is thrown **if this lock is not held when** any of the **Condition waiting** or **signalling methods** are called. * **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called. * If a **thread is interrupted while waiting** then InterruptedException will be thrown and following things will happen -   + the **wait will be over**, and   + **thread's interrupted status will be cleared**. * Waiting threads are signalled in FIFO (first in first out order) order. * When lock is  ***fair***, first lock is obtained by longest-waiting thread.   If lock is not  ***fair***, any waiting thread could get lock, at discretion of implementation. |

**Question 41. Difference between traditional** [**multithreading**](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) **and parallel programming?**

**Answer.** MultiThreading primarily was designed to work with single CPU and utilize idle time of CPU. If two or more processors are there multithreading won’t be able to utilize multi processors but parallel programing using Fork/Join framework can utilize multiple processors available in computer.**Question 42. Explain atomic operations in java?**

### Answer. Java provides some classes in java.util.concurrent.atomic which offers an alternative to the other [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html).

Please see [Atomic operations in java](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html).**Question 43. What is AtomicInteger? Explain key methods of AtomicInteger?**

**Answer.***AtomicInteger  provides you with* ***int value*** *that is updated atomically. i.e. we can use these classes without any explicit synchronization in*[*multithreading*](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) *environment, because any operation done on these classes is*[*thread safe*](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)*.*

[*AtomicInteger*](http://www.javamadesoeasy.com/2015/03/atomicinteger-in-java.html) *important Methods >*

* *int* ***get****()*

method returns the current value

* *void* ***set****(int newValue)*
* Sets to **newValue.***int* ***getAndSet****(int newValue)*
* Sets to **newValue** and returns the old value.*boolean* ***compareAndSet****(int expect, int update)*

Compare with *expect*, if equal, set to *update* and return true.*Addition methods >*

* *int* ***addAndGet****(int value)*
* adds *value* to the current value. And **return updated value.***int* ***incrementAndGet****()*
* increments current value by 1. And **return updated value.***int* ***getAndAdd****(int value)*
* Method **return current value**. And adds *value* to the current value.*int* ***getAndIncrement****()*

Method **return current value**. And increments current value by 1.*Subtraction methods >*

* *int* ***decrementAndGet****()*
* decrements current value by 1. And **return updated value.***int* ***getAndDecrement****()*

Method **return current value**. And decrements current value by 1.**Question 44. How can you implement your own AtomicInteger?**

**Answer.** Please see [**Implementation of custom/own AtomicInteger in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown.html).**Question 45. What is AtomicLong? Explain key methods of AtomicLong?**

**Answer.** AtomicLong  provides you with **long value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in[multithreading](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) environment, because any operation done on these classes is[thread safe](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html).

[*AtomicLong*](http://www.javamadesoeasy.com/2015/03/atomiclong-in-java.html) *important Methods >*

* *long* ***get****()*

method returns the current value

* *void* ***set****(long newValue)*

Sets to **newValue.**

* *long* ***getAndSet****(long newValue)*

Sets to **newValue** and returns the old value.

* *boolean* ***compareAndSet****(long expect, long update)*

*Addition methods >*

* *long* ***addAndGet****(long value)*
* adds *value* to the current value. And **return updated value.***long* ***incrementAndGet****()*
* increments current value by 1. And **return updated value.***long* ***getAndAdd****(long value)*
* Method **return current value**. And adds *value* to the current value.*long* ***getAndIncrement****()*

Method **return current value**. And increments current value by 1.*Subtraction methods >*

* *long* ***decrementAndGet****()*

decrements current value by 1. And **return updated value.**

* *long* ***getAndDecrement****()*

Method **return current value**. And decrements current value by 1.

Custom implementation of **ReadWriteLock**

is a  **interface,** it provides methods  >

* which allows multiple threads to **read** *shared resource* at same time, but
* only one thread can **write** to *shared resource*at same time.

If a thread attempts to acquire the writeLock on shared resource when it is acquired by another thread, then it waits until the lock is released. In this way we achieve [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and [race conditions](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html) is avoided.

In next post we will learn [**Implementation of custom/own ReadWriteLock and ReentrantReadWriteLock in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown-lock-and.html).

***ReadWriteLock*** *interface important methods >*

*void readLock()*

**More than one threads can acquire readLock** at a time, provided **no other thread is acquiring writeLock** at same time.

**CODE >**

ReadLock has been created as Inner class which provides readlock.

**publicclass**ReadLock {

**publicsynchronizedvoid** lock() {

/\*

\* More than one threads can acquire readLock at a time, provided no

\* other thread is acquiring writeLock at same time.

\*/

**if** (writeLockCount == 0) {

readLockCount++;

}

/\*

\* if some other thread is acquiring write lock at that time, than

\* current thread waits.

\*/

**else** {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**publicsynchronizedvoid** unlock() {

readLockCount--; // decrement readLockCount.

/\*

\* If readLockCount has become 0, all threads waiting to write will be

\* notified and can acquire lock.

\*/

**if** (readLockCount == 0)

notifyAll();

}

}

|  |
| --- |
|  |

*void writeLock()*

**Only one threads can acquire writeLock** at a time. Means writeLock can only be obtained if no other thread is acquiring read or write lock at that time.

**CODE >**

WriteLock has been created as Inner class which provides writelock.

**publicclass**WriteLock {

**publicsynchronizedvoid** lock() {

/\*

\* writeLock can only be obtained if no other thread is acquiring read

\* or write lock at that time.

\*/

**if** (writeLockCount == 0 &&readLockCount == 0) {

writeLockCount++;

}

/\*

\* if some other thread is acquiring read or write lock at that time,

\* than current thread waits.

\*/

**else** {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**publicsynchronizedvoid** unlock() {

writeLockCount--; // decrement writeLockCount.

notifyAll(); // notify all waiting threads.

}

}

*ReentrantReadWriteLock class >*

**java.util.concurrent.locks.ReentrantReadWriteLock** is a class which implements **ReadWriteLock** interface.

***Custom ReadWriteLock and ReentrantReadWriteLock’s code >***

/\*\* \*/

**interface** ReadWriteLock {

**public** ReentrantReadWriteLock.WriteLock writeLock();

**public** ReentrantReadWriteLock.ReadLock readLock();

}

**class** ReentrantReadWriteLock **implements** ReadWriteLock {

// Variables to maintain read and write lock count.

**privateint**readLockCount;

**privateint**writeLockCount;

/\* Inner class providing readlock \*/

**privatefinal** ReentrantReadWriteLock.ReadLock readerLock;

/\* Inner class providing writelock \*/

**privatefinal** ReentrantReadWriteLock.WriteLock writerLock;

**public** ReentrantReadWriteLock.WriteLock writeLock() {

**return**writerLock;

}

**public** ReentrantReadWriteLock.ReadLock readLock() {

**return**readerLock;

}

/\*\*

\* Constructor

\*/

**public** ReentrantReadWriteLock() {

readerLock = **new** ReadLock();

writerLock = **new** WriteLock();

}

/\*\*

\* More than one threads can acquire readLock at a time, provided no other

\* thread is acquiring writeLock at same time.

\*/

**publicclass** ReadLock {

**publicsynchronizedvoid** lock() {

/\*

\* More than one threads can acquire readLock at a time, provided no

\* other thread is acquiring writeLock at same time.

\*/

**if** (writeLockCount == 0) {

readLockCount++;

}

/\*

\* if some other thread is acquiring write lock at that time, than

\* current thread waits.

\*/

**else** {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**publicsynchronizedvoid** unlock() {

readLockCount--; // decrement readLockCount.

/\*

\* If readLockCount has become 0, all threads waiting to write will

\* be notified and can acquire lock.

\*/

**if** (readLockCount == 0)

notifyAll();

}

}

/\*\*

\* Only one threads can acquire writeLock at a time. Means writeLock can

\* only be obtained if no other thread is acquiring read or write lock at

\* that time.

\*/

**publicclass** WriteLock {

**publicsynchronizedvoid** lock() {

/\*

\* writeLock can only be obtained if no other thread is acquiring

\* read or write lock at that time.

\*/

**if** (writeLockCount == 0 &&readLockCount == 0) {

writeLockCount++;

}

/\*

\* if some other thread is acquiring read or write lock at that

\* time, than current thread waits.

\*/

**else** {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**publicsynchronizedvoid** unlock() {

writeLockCount--; // decrement writeLockCount.

notifyAll(); // notify all waiting threads.

}

}

}

**Thread Interview questions**

**Question 1. What is Thread in java?**

**Answer.**

* Threads **consumes CPU in best possible manner**, hence enables multi processing. Multi threading **reduces idle time of CPU** which improves performance of application.
* Thread are **light weight process**.
* A thread class belongs to **java.lang package**.
* We can create multiple threads in java, **even if we don’t create any Thread, one Thread at least  do exist** i.e. **main thread**.
* **Multiple threads run parallely in java.**
* Threads have their **own stack**.
* Advantage of Thread : Suppose one thread needs 10 minutes to get certain task, 10 threads used at a time could complete that task in 1 minute, because threads can run parallely.

**Question 2. What is difference between Process and Thread in java?**

**Answer.**  One process can have multiple Threads,

Thread are **subdivision** of Process. One or more Threads runs in the context of process. Threads can execute any part of process. And same part of process can be executed by multiple Threads.

Processes have their own **copy of the data segment of the parent process** while Threads have **direct access to the data segment of its process**.

Processes have their **own address** while Threads share the **address space of the process that created it**.

Process creation needs whole lot of stuff to be done, we **might need to copy whole parent process**, but Thread can be **easily created**.

Processes can **easily communicate with child processes** but **interprocess communication is difficult**. While, Threads **can easily communicate with other threads of the same process using** [**wait() and notify() methods**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html).

In process **all threads share system resource** like heap Memory etc. while Thread has its **own stack.**

Any change made to process **does not affect child processes**, but any change made to thread **can affect the behavior of the other threads of the process**.

**Question 3. How to implement Threads in java?**

**Answer.**  This is very basic threading question. Threads can be created in two ways i.e. by implementing **java.lang.Runnable** interface or extending **java.lang.Thread** class and then extending run method.

Thread has its own variables and methods, it lives and dies on the heap. But a thread of execution is an individual process that has its own call stack. Thread are lightweight process in java.

1. Thread creation by  implementing**java.lang.Runnable**interface.

We will create object of class which implements Runnable interface :

**MyRunnable runnable=new MyRunnable();**

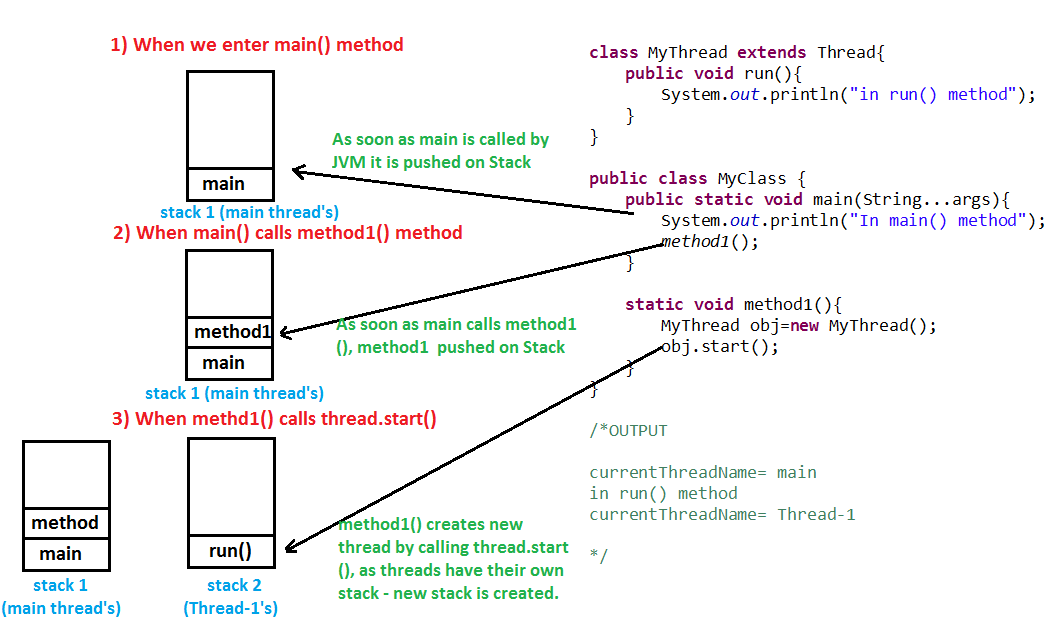
**Thread thread=new Thread(runnable)**

     2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e.  **runnable** object :

|  |
| --- |
| **Thread thread=new Thread(runnable);** |

**Question 4 . Does Thread implements their own Stack, if yes how? (Important)**

**Answer.**  **Yes**, Threads have their own stack. This is very interesting question, where interviewer tends to check your basic knowledge about how threads internally maintains their own stacks. I’ll be explaining you the concept by diagram.

[](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)

**Question 5. We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?**

**Answer.** Well the answer is you must [**extend Thread**](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) only when you are looking to **modify run() and other methods as well**. If you are simply looking to **modify only the run() method** [**implementing Runnable**](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) **is the best option (**Runnable interface has only one abstract method i.e. run() **)**.

***Differences between implementing Runnable interface and extending Thread class -***

1. ***Multiple inheritance in not allowed in java :*** When we [implement Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) interface **we can extend another class as well**, but if we extend Thread class **we cannot extend any other class** because java does not allow multiple inheritance. So, same work is done by implementing Runnable and [extending Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) but in case of implementing Runnable we are still left with option of extending some other class. **So, it’s better to implement Runnable.**
2. [***Thread safety***](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) ***:*** When we implement Runnable interface, **same object is shared amongst multiple threads**, but when we extend Thread class **each and every thread gets associated with new object**.
3. ***Inheritance (Implementing Runnable is lightweight operation) :*** When we extend Thread **unnecessary all Thread class features are inherited**, but when we implement Runnable interface no **extra feature are inherited**, as Runnable only consists only of one abstract method i.e. run() method.  **So, implementing Runnable is lightweight operation.**
4. ***Coding to interface :*** Even **java recommends coding to interface**. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.
5. ***Don’t extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run()  :*** We must [**extend Thread**](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) only when you are looking to **modify run() and other methods as well**. If you are simply looking to **modify only the run() method** [**implementing Runnable**](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) **is the best option (**Runnable interface has only one abstract method i.e. run() **). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.**
6. ***Flexibility in code when we implement Runnable :*** When we extend Thread first a fall all thread features are inherited and **our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class**. But, when we implement Runnable **we create a new thread and pass runnable object as parameter,we could pass runnable object to executorService & much more**. So, we have more options when we implement Runnable and **our code becomes more flexible.**
7. ***ExecutorService :*** If we implement Runnable, **we can start multiple thread created on runnable object  with ExecutorService** (because we can start Runnable object with new threads), **but not in the case when we extend** Thread (because thread can be started only once).

**Question 6. How can you say Thread behaviour is unpredictable? (Important)**

**Answer.** The solution to question is quite simple, [Thread behaviour is unpredictable](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html) because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform.

To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start  both threads. There is no surety that which threads will complete first,  both threads will enter anonymously in for loop.

**Question 7 . When threads are not lightweight process in java?**

**Answer.** Threads are [**lightweight process**](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html) **only if threads of same process are executing concurrently**. But **if threads of different processes are executing concurrently then threads are** [**heavy weight process**](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html).

**Question 8. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)**

**Answer.**  Interviewers tend to know interviewees knowledge about Thread methods. So this is time to prove your point by answering correctly. We can use [**join() method**](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html)to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words **waits for this thread to die**. **Calling join() method internally calls join(0);**

10 salient features of **join()** method >

* **Definition** : join()We can use **join() method** to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words **waits for thread to die on which thread has been called**.
* **Exception :** join**()** method [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) **InterruptedException**, in our case we have thrown exception.
* **instance method :** join()is a **instance method**, hence we need to have thread  instance for calling this method.
* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** join() **method is called on thread it goes from running to waiting state. And wait for thread to die.**
* **Not a native method :** implementation of join() method is provided in java.lang.Thread class.

Let’s see definition of join() method as given in java.lang.Thread -

|  |
| --- |
| **publicfinalvoid**join() **throws** InterruptedException; |

* **synchronized block :** thread **need not to to acquire object lock** before calling join()method i.e. join() method **can be called from outside synchronized block**.
* **Waiting time :** join() **method have got few options.**

1. **join() :** Waits for this thread to die.

|  |
| --- |
| **publicfinalvoid**join() **throws** InterruptedException; |

This method internally calls **join(0).** And timeout of 0 means to wait forever;

1. **join(long millis) -** Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis) **throws** InterruptedException; |

1. **join(long millis, int nanos) -** Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis,**int** nanos) **throws** InterruptedException; |

* **Belongs to which class :**join**() method belongs to java.lang.Thread** class.

To achieve we are going to create 2 threads on Runnable Object, create for loop in run() method and start  both threads. After starting each Thread call join() method on them to ensure they end in order in which they has started.

**Full Program to show usage of join() method>**

**class** MyRunnable **implements** Runnable{

**publicvoid** run(){

System.*out*.println("in run() method");

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

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

}

}

}

/\*\* Copyright (c), Test.com \*/

**publicclass** MyClass {

**publicstaticvoid** main(String...args) **throws** InterruptedException{

System.*out*.println("In main() method");

MyRunnable runnable=**new** MyRunnable();

Thread thread1=**new** Thread(runnable);

Thread thread2=**new** Thread(runnable);

thread1.start();

thread1.join();

thread2.start();

thread2.join();

System.*out*.println("end main() method");

}

}

/\*OUTPUT

In main() method

in run() method

i=0 ,ThreadName=Thread-0

i=1 ,ThreadName=Thread-0

i=2 ,ThreadName=Thread-0

i=3 ,ThreadName=Thread-0

i=4 ,ThreadName=Thread-0

in run() method

i=0 ,ThreadName=Thread-1

i=1 ,ThreadName=Thread-1

i=2 ,ThreadName=Thread-1

i=3 ,ThreadName=Thread-1

i=4 ,ThreadName=Thread-1

end main() method

\*/

If we note output, all threads ended in order in which they were called and main thread has ended last.

First, main thread was called, it started Thread1 and then we called join() method on Thread1, once Thread1 ended main thread started Thread2 and we called join() method on Thread2, once Thread2 ended main thread also ended.

**In short - calling thread1.join()  made main thread to wait until Thread-1 dies.**

**Let’s discuss waiting time in detail :** join() **method have got few options.**

1. **join() :** Waits for this thread to die.

|  |
| --- |
| **publicfinalvoid**join() **throws** InterruptedException; |

This method internally calls **join(0).** And timeout of 0 means to wait forever;

1. **join(long millis) -** Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis) **throws** InterruptedException; |

1. **join(long millis, int nanos) -** Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis,**int** nanos) **throws** InterruptedException; |

Let’s create a program to use **join(long millis)** >First, join(1000) will be called on Thread-1, **but once 1000 millisec are up, main thread can resume and start thread2 (main thread won’t wait for Thread-1 to die).**

**class** MyRunnable **implements** Runnable{

**publicvoid** run(){

System.*out*.println("in run() method");

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

}

/\*\* Copyright (c), Test.com \*/

**publicclass** MyClass {

**publicstaticvoid** main(String...args) **throws** InterruptedException{

System.*out*.println("In main() method");

MyRunnable runnable=**new** MyRunnable();

Thread thread1=**new** Thread(runnable);

Thread thread2=**new** Thread(runnable);

thread1.start();

thread1.join(1000); //once 1000 millisec are up, main thread can resume and start thread2.

thread2.start();

thread2.join();

System.*out*.println("end main() method");

}

}

/\*OUTPUT

In main() method

in run() method

i=0 ,ThreadName=Thread-0

i=1 ,ThreadName=Thread-0

in run() method

i=2 ,ThreadName=Thread-0

i=0 ,ThreadName=Thread-1

i=1 ,ThreadName=Thread-1

i=3 ,ThreadName=Thread-0

i=2 ,ThreadName=Thread-1

i=4 ,ThreadName=Thread-0

i=3 ,ThreadName=Thread-1

i=4 ,ThreadName=Thread-1

end main() method

\*/

**Question 9.What is difference between starting thread with run() and start() method? (Important)**

**Answer.** This is quite interesting question, it might confuse you a bit and at time may make you think is there really any difference between starting thread with run() and start() method.

When you **call start()** method, **main thread internally calls run() method** to start newly created Thread, so **run() method is ultimately called by newly created thread**.

When you **call run()** method **main thread** rather than starting run() method with newly thread it start **run() method by itself**.

**Question 10. What is significance of using** [**Volatile**](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) **keyword? (Important)**

**Answer.** Java allows threads to **access shared variables**. As a rule, to ensure that **shared variables are consistently updated**, a thread should ensure that it has **exclusive use of such variables by obtaining a lock** that enforces mutual exclusion for those shared variables.

**If a field is declared**[**volatile**](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)**, in that case the Java memory model ensures that all threads see a consistent value for the variable.**

Few small questions>

Q. Can we have [volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) methods in java?

1. **No**, volatile is only a keyword, can be used only with variables.

Q. Can we have synchronized variable in java?

1. **No**, synchronized can be used only with methods, i.e. in method declaration.

### DETAILED DESCRIPTION : Volatile keyword in java- difference between synchronized and volatile with programs, 10 key points about volatile keyword, why volatile variables are not cached in memory

**Question 11. Differences between synchronized and volatile keyword in Java? (Important)**

**Answer.**Its very important question from interview perspective.

1. [Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) can be used as a **keyword** against the variable, we **cannot** use volatile against method declaration.

**volatilevoid** method1(){} //it’s illegal, compilation error.

**volatile** int i;**//legal**

While [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) can be used in method declaration or we can create synchronization blocks (In both cases thread acquires lock on object’s monitor). Variables cannot be synchronized.

Synchronized method:

**synchronizedvoid** method2(){} //legal

Synchronized block:

**void** method2(){

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

         //code inside synchronized block.

         }

}

Synchronized variable (illegal):

**synchronized** int i;//it’s illegal, compilatiomn error.

1. Volatile does not acquire any lock on variable or object, but [Synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) acquires lock on method or block in which it is used.
2. [Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) variables are not cached, but variables used inside [synchronized](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) method or block are cached.
3. When volatile is used will never create deadlock in program, as volatile never obtains any kind of lock . But in case if synchronization is not done properly, we might end up creating dedlock in program.
4. Synchronization may cost us performance issues, as one thread might be waiting for another thread to release lock on object. But volatile is never expensive in terms of performance.

### DETAILED DESCRIPTION : Differences between synchronized and volatile keyword in detail with programs.

**Question 12. Can you again start Thread?**

**Answer.No**, we cannot start Thread again, doing so will throw runtimeException java.lang.IllegalThreadStateException. The reason is once run() method is executed by Thread, it goes into [**dead state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

Let’s take an example-

Thinking of starting thread again and calling start() method on it (which internally is going to call run() method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to **dead state**.

**Question 13. What is race condition in multithreading and how can we solve it? (Important)**

**Answer.** This is very important question, this forms the core of multi threading, you should be able to explain about race condition in detail. When more than one thread try to access same resource without synchronization causes race condition.

**So we can solve race condition by using either synchronized block or synchronized method.** When no two threads can access same resource at a time phenomenon is also called as **mutual exclusion**.

**Few sub questions>**

What if two threads try to **read** same resource without synchronization?

When two threads try to read on same resource without synchronization, **it’s never going to create any problem**.

What if two threads try to **write** to same resource without synchronization?

When two threads try to **write** to same resource without synchronization, **it’s going to create synchronization problems**.

**Question 14. How threads communicate between each other?**

**Answer.** This is very must know question for all the interviewees, you will most probably face this question in almost every time you go for interview.

**Threads can communicate** with each other by using **wait(), notify() and notifyAll()** methods.

**Question 15. Why wait(), notify()  and notifyAll() are in Object class and not in Thread class? (Important)**

**Answer.**

1. **Every Object has a monitor**, acquiring that monitors allow thread to hold lock on object. But **Thread class does not have any monitors**.
2. wait(), notify() and notifyAll()are called on objects only >**When wait() method** is called on object by thread **it waits for another thread** on that object to **release object monitor by calling** [**notify() or notifyAll()**](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on that object.

**When notify() method is called** on object by thread **it notifies all the threads**

which are **waiting for that object monitor** that object monitor is available now.

So, this shows that wait(), notify() and notifyAll() are called on objects only.

[Now, Straight forward question that comes to mind is how thread acquires object lock by](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

[acquiring object monitor? Let’s try to understand this basic concept in detail?](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

1. Wait(), notify() and notifyAll() method being in Object class allows all the **threads created on that object** to **communicate** with other.  [As multiple threads may exist on same object].
2. As **multiple threads exists on same object**. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).
3. Now let’s discuss one **hypothetical** scenario, **what will happen if Thread class contains wait(), notify() and notifyAll() methods**?

Having wait(), notify() and notifyAll() methods **means Thread class also must have their monitor**.

Every thread having their monitor will create few problems -

>**Thread communication problem.**

>**Synchronization on object won’t be possible**- Because object has monitor, one object can have multiple threads and thread hold lock on object by holding object monitor. But if each thread will have monitor, we won’t have any way of achieving synchronization.

>**Inconsistency in state of object** (because synchronization won't be possible).

**Question 16. Is it important to acquire object lock before calling wait(), notify() and notifyAll()?**

**Answer.Yes**, it’s mandatory to acquire object lock before calling these methods on object. As discussed above [**wait(), notify()  and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) **methods are always called from** [**Synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **only**, and **as soon as thread enters synchronized block it acquires object lock** (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang. IllegalMonitorStateException is thrown at runtime.

Wait() method needs to enclosed in try-catch block, because it throws compile time exception i.e. InterruptedException.

**Question 17. How can you solve consumer producer problem by using wait() and notify() method? (Important)**

**Answer.**  Here come the time to answer **very very important question from interview perspective**. Interviewers tends to check how sound you are in threads inter communication. Because for solving this problem we got to **use synchronization blocks, wait() and notify() method very cautiously**. **If you misplace synchronization block or any of the method**, that **may cause your program to go horribly wrong**. So, before going into this question first i’ll recommend you to understand how to use synchronized blocks, wait() and notify() methods.

**Key points** we need to ensure before programming :

**>**Producer will produce total of 10 products and cannot produce more than 2 products at a time until products are being consumed by consumer.

**Example**> when sharedQueue’s size is 2, wait for consumer to consume (consumer will consume by calling remove(0) method on sharedQueue and reduce sharedQueue’s size). As soon as size is less than 2, producer will start producing.

**>**Consumer can consume only when there are some products to consume.

**Example**> when sharedQueue’s size is 0, wait for producer to produce (producer will produce by calling add() method on sharedQueue and increase sharedQueue’s size).   As soon as size is greater than 0, consumer will start consuming.

Explanation of **Logic** >

We will create sharedQueue that will be shared amongst Producer and Consumer. We will now start consumer and producer thread.

Note: it does not matter order in which threads are started (because rest of code has taken care of synchronization and key points mentioned above)

First we will start consumerThread >

|  |
| --- |
| consumerThread.start(); |

consumerThread will enter run method and call consume() method. There it will check for sharedQueue’s size.

-if size is equal to 0 that means producer hasn’t produced any product, wait for producer to produce by using below piece of code-

**synchronized** (sharedQueue) {

**while** (sharedQueue.size() == 0) {

sharedQueue.wait();

}

}

if size is greater than 0, consumer will start consuming by using below piece of code.

**synchronized** (sharedQueue) {

Thread.sleep((**long**)(Math.random() \* 2000));

System.out.println("consumed : "+ sharedQueue.remove(0));

sharedQueue.notify();

}

Than we will start producerThread >

|  |
| --- |
| producerThread.start(); |

producerThread will enter run method and call produce() method. There it will check for sharedQueue’s size.

if size is equal to 2 (i.e. maximum number of products which sharedQueue can hold at a time), wait for consumer to consume by using below piece of code-

**synchronized** (sharedQueue) {

**while** (sharedQueue.size() == maxSize) { //maxsize is 2

sharedQueue.wait();

}

}

-if size is less than 2, producer will start producing by using below piece of code.

**synchronized** (sharedQueue) {

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

sharedQueue.add(i);

Thread.sleep((**long**)(Math.random() \* 1000));

sharedQueue.notify();

}

**Question 18.** [**How to solve Consumer Producer problem without using wait() and notify() methods, where consumer can consume only when production is over.**](http://www.javamadesoeasy.com/2015/03/how-to-solve-consumer-producer-problem.html)**?**

**Answer.** In this problem, producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).

We will approach by keeping one boolean variable **productionInProcess** and initially setting it to **true**, and later when production will be over we will set it to **false**.

**Question 19. How can you solve consumer producer pattern by using BlockingQueue? (Important)**

**Answer.** Now it’s time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using wait() and notify() method. Generally you might wonder why interviewer's are so much interested in asking about [solving consumer producer problem using BlockingQueue](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html), answer is they want to know how strong knowledge you have about java concurrent Api’s, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.

[**BlockingQueue is a interface** and we will use its **implementation class LinkedBlockingQueue**.](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html)

Key methods for solving consumer producer pattern are >

|  |
| --- |
| **put(i);**   //used by producer to put/produce in sharedQueue.  **take();**//used by consumer to take/consume from sharedQueue. |

**Question 20. What is** [**deadlock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) **in multithreading? Write a program to form** [**DeadLock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) **in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important)**

**Answer.**  This is very important question from interview perspective. But, what makes this question important is it checks interviewees capability of [**creating and detecting deadlock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html). If you can write a code to form deadlock, than I am sure you must be well capable in solving that deadlock as well. If not, later on this post we will learn how to solve deadlock as well.First question comes to mind is, [**what is deadlock in multi threading program**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)**?**

**Deadlock is a situation where two threads are waiting for each other to release lock holded by them on resources.**

But how [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) could be formed :

**Thread-1 acquires lock on String.class** and then calls [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 acquires lock on Object.class** then calls sleep() method and **now it waits for Thread-1 to release lock on String.class**.

**Conclusion:**

Now, **Thread-1 is waiting for Thread-2 to release lock on Object.class** and **Thread-2 is waiting for Thread-1 to release lock on String.class** and deadlock is formed.

Code called by Thread-1

**publicvoid**run() {

**synchronized** (String.**class**) {

Thread.sleep(100);

**synchronized** (Object.**class**) {

}

}

}

Codecalledby Thread-2

**publicvoid**run() {

**synchronized** (Object.**class**) {

Thread.sleep(100);

**synchronized** (String.**class**) {

}

}

}

Here comes the **important** part, how above formed **deadlock** could be **solved** :

**Thread-1 acquires lock on String.class** and then calls [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 tries to acquire lock on String.class** but lock is holded by Thread-1. Meanwhile, Thread-1 completes successfully. As Thread-1 has completed successfully it releases lock on String.class, Thread-2 can now acquire lock on String.class and complete successfully without any deadlock formation.

**Conclusion:** No deadlock is formed.

CodecalledbyThread-1

**publicvoid**run() {

**synchronized** (String.**class**) {

Thread.sleep(100);

**synchronized** (Object.**class**) {

}

}

}

Code calledbyThread-2

**publicvoid**run() {

**synchronized** (String.**class**) {

Thread.sleep(100);

**synchronized** (Object.**class**) {

}

}

}

Few important measures to avoid [Deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)>

1. **Lock specific member variables of class rather than locking whole class**: We must try to lock specific member variables of class rather than locking whole class.
2. **Use join() method:** If possible try touse join() method, although it may refrain us from taking full advantage of multithreading environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.
3. **If possible try avoid using nested synchronization blocks.**

**Question 21. Have you ever generated thread dumps or analyzed Thread Dumps? (Important)**

**Answer.** Answering this questions will show your in depth knowledge of Threads. Every experienced must know how to generate Thread Dumps.

[**VisualVM**](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)  is most popular way to generate Thread Dump and is most widely used by developers. It’s important to understand usage of VisualVM for in depth knowledge of VisualVM. I’ll recommend every developer must understand this topic to become master in multi threading.

It helps us in analyzing threads performance, [thread states](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html), CPU consumed by threads, garbage collection and much more.  For detailed information see [**Generating and analyzing Thread Dumps using VisualVM - step by step detail to setup VisualVM with screenshots**](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)

[**jstack**](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html) is very easy way to generate Thread dump and is widely used by developers. I’ll recommend every developer must understand this topic to become master in multi threading. For creating Thread dumps we **need not to download any jar or any extra software**. For detailed information see [**Generating and analyzing Thread Dumps using JSATCK - step by step detail to setup JSTACK with screenshots**](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html).

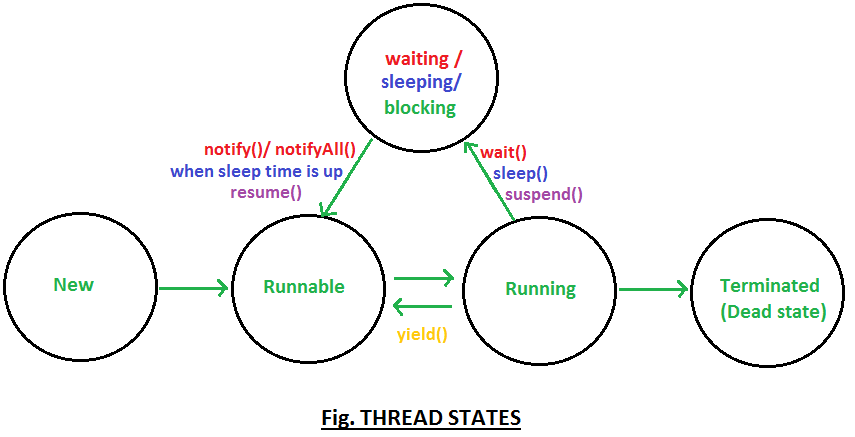
**Question 22. What is life cycle of Thread, explain thread states? (Important)**

**Answer.**  [**Thread states/ Thread life cycle**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **is very basic question, before going deep into concepts we must understand Thread life cycle.**

**Thread have following states >**

* **New**
* **Runnable**
* **Running**
* **Waiting**/**blocked/sleeping**
* **Terminated (Dead)**

**Thread states/ Thread life cycle in diagram >**

****

**Thread states in detail >**

**New : When instance of thread is created using new operator it is in new state**, but the start() method has not been invoked on the thread yet, thread is not eligible to run yet.

**Runnable :**  **When start() method is called on thread it enters runnable state**. **Running :** Thread scheduler selects thread to go fromrunnable to running state. In running state Thread starts executing by entering run() method.

**Waiting/blocked/sleeping :** In this state a thread is not eligible to run.

>Thread is still alive, but currently it’s not eligible to run. In other words.

**> How can Thread go from running to waiting state?**

  By calling **wait()**[method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

**> How can Thread go from running to sleeping state?**

  By calling **sleep()** [method](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.

**Terminated (Dead) :** A thread is considered dead **when its run() method completes**.

**Question 23. Are you aware of preemptive scheduling and time slicing?**

**Answer.** In **preemptive scheduling**, the **highest priority thread executes until** it enters into the [**waiting or dead state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**In time slicing**, a **thread executes for a certain predefined time** and **then enters runnable pool**. Than thread can enter running state when selected by thread scheduler.

**Question 24. What are** [**daemon threads**](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)**?**

**Answer.**[Daemon threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) are low priority threads which **runs intermittently in background** for doing **garbage collection**.

   12 Few salient features of [**daemon()** threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)>

* **Thread scheduler schedules these threads** only **when CPU is idle**.

[Daemon threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) are **service oriented threads**, they **serves all other threads**.

These threads are **created before user threads are created** and **die after all other user threads dies**.

**Priority of daemon threads is always 1** (i.e. MIN\_PRIORITY).

* **User created threads are non daemon threads**.
* **JVM can exit** when only daemon threads exist in system.
* we can use **isDaemon()** method to check whether thread is daemon thread or not.
* we can use **setDaemon(boolean on)** method to make any user method a daemon thread.
* If **setDaemon(boolean on)** is called on thread after calling start() method than IllegalThreadStateException is thrown.
* You may like to see how daemon threads work, for that you can use VisualVM or jStack. I have provided Thread dumps over there which shows daemon threads which were intermittently running in background.

Some of the daemon threads which intermittently run in background are >

|  |
| --- |
| "RMI TCP Connection(3)-10.175.2.71" daemon"RMI TCP Connection(idle)" daemon "RMI Scheduler(0)" daemon "C2 CompilerThread1" daemon  "GC task thread#0 (ParallelGC)" |

**Question 25. Why** [**suspend() and resume() methods are deprecated**](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html)**?**

**Answer.**[Suspend()](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) method is [**deadlock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) **prone**. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is [resumed](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html). [If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, it results in **deadlock formation**](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html).These [**deadlocks**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)are generally called **Frozen processes**.

**Suspend() method puts thread from** [**running to waiting state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). And thread can go **from waiting to runnable** state **only when resume() method is called** on thread. It is deprecated method.

**Resume()** method is **only used with suspend()** method that’s why it’s also deprecated method.

**Question 26. Why destroy() methods is deprecated?**

**Answer.** This question is again going to check your in depth knowledge of thread methods i.e. [destroy() method](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) is [**deadlock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) **prone**. If the target thread holds a lock on object when it is destroyed, no thread can lock this object (Deadlock formed are similar to deadlock formed when suspend() and resume() methods are used improperly). It results in **deadlock formation**. These [**deadlocks**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)are generally called **Frozen processes**.

Additionally you must know calling destroy() method on Threads throw runtimeException i.e. NoSuchMethodError. [**Destroy() method**](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) **puts thread from running to** [**dead state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)**.**

**Question 27. As stop() method is deprecated,  How can we terminate or stop infinitely running thread in java? (Important)**

**Answer.** This is very interesting question where interviewees thread basics basic will be tested. Interviewers tend to know user’s knowledge about main thread’s and thread invoked by main thread.

We will try to address the problem by creating new thread which will run infinitely until certain condition is satisfied and will be called by main Thread.

1. Infinitely running thread can be stopped **using boolean variable.**
2. [Infinitely running thread can be stopped **using interrupt() method**](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html)**.**

**Let’s understand Why stop() method is deprecated :**

Stopping a thread with Thread.stop() causes it to release all of the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, which might lead to unpredictable behavior.

**Question 28. what is significance of yield() method, what state does it put thread in?**

[**yield()**](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) is a **native method** it’s implementation in java 6 has been changed as compared to its implementation java 5. As method is native it’s implementation is provided by JVM.

**In java 5,** yield() method **internally used to call** [**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **method** giving all the other threads of same or higher priority to execute before yielded thread by leaving allocated CPU for time gap of 15 millisec.

**But java 6**, calling **yield() method gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor**. The **thread scheduler is free to ignore this hint**. So, sometimes even after using yield() method, you may not notice any difference in output.

salient features of [**yield()**](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) method >

* **Definition** : [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) **method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor.The thread scheduler is free to ignore this hint**.
* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** yield() **method is called on thread it goes from running to runnable state**, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.
* **Waiting time :** yield() **method stops thread for unpredictable time.**
* **Static method :** yield()is a **static method**, hence calling Thread.yield() causes currently executing thread to yield.
* **Native method :** implementation of yield() method is provided by **JVM**.

Let’s see definition of yield() method as given in java.lang.Thread -

|  |
| --- |
| **publicstaticnativevoid** yield(); |

* [**synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **:** thread **need not to to acquire object lock** before calling yield()method i.e. yield() method **can be called from outside synchronized block**.

**Question 29.What is significance of sleep() method in detail, what** [**state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **does it put thread in ?**

[**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) is a **native method**, its implementation is provided by JVM.

10 salient features of [**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method >

* **Definition** : [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **methods causes current thread to sleep for specified number of milliseconds** (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.
* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** sleep() **is called on thread it goes from running to waiting state** and can return to runnable state when sleep time is up.
* **Exception :** sleep() **method must catch or throw compile time exception** i.e. InterruptedException.
* **Waiting time :** sleep() **method have got few options.**
  1. **sleep(long millis) -** Causes the currently executing thread to sleep for the specified number of milliseconds

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis) **throws** InterruptedException; |

* 1. **sleep(long millis, int** nanos**) -** Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis,**int** nanos) **throws** InterruptedException; |

* **static method :** sleep()is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.
* **Native method :** implementation of sleep() method is provided by **JVM**.

Let’s see definition of yield() method as given in java.lang.Thread -

|  |
| --- |
| **publicstaticnativevoid** sleep(**long** millis) **throws** InterruptedException; |

* **Belongs to which class :**[sleep**()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **method belongs to java.lang.Thread** class.

**synchronized block :** thread **need not to to acquire object lock** before calling sleep()method i.e. sleep() method **can be called from outside synchronized block**.

**Question 30. Difference between** [**wait()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) **and** [**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **? (Important)**

**Answer.**

* **Should be called from** [**synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **:**wait() **method is always called from synchronized block** i.e. [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method needs to lock object monitor before object on which it is called.  But sleep() **method can be called from outside synchronized block** i.e. sleep() method doesn’t need any object monitor.

**IllegalMonitorStateException : if** wait() **method is called without acquiring object lock** than IllegalMonitorStateException is thrown at runtime, but sleep() method**never throws such exception**.

* **Belongs to which class : wait() method belongs to java.lang.Object** class but **sleep() method belongs to java.lang.Thread** class.
* **Called on object or thread :** wait() **method is called on objects** but sleep() **method is called on Threads** not objects**.**
* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** wait() **method is called on object, thread that holded object’s monitor goes from running to waiting state** and can **return to runnable state only when notify() or notifyAll()method is called on that object**. And later thread scheduler schedules that thread to go from from runnable to running state.

when sleep() **is called on thread it goes from running to waiting state** and can **return to runnable state when sleep time is up.**

**When called from** [**synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **:**when wait() method is called **thread leaves the object lock**.  But sleep()method **when called from synchronized block or method thread doesn’t leaves object lock.**

**Question 31. Differences and similarities between yield() and sleep() ?**

**Answer.**

Differences [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) and [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) :

* **Definition** : yield() **method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor.The thread scheduler is free to ignore this hint**. sleep() **methods causes current thread to sleep for specified number of milliseconds** (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.

[**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** sleep() **is called on thread it goes from running to waiting state** and can return to runnable state when sleep time is up. **when** yield() **method is called on thread it goes from running to runnable state**, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.

* **Exception :** yield() **method need not to catch or throw any exception.** But sleep() **method must catch or throw compile time exception** i.e. InterruptedException.
* **Waiting time :** yield() **method stops thread for unpredictable time, that depends on thread scheduler.** But sleep() **method have got few options.**
  1. **sleep(long millis) -** Causes the currently executing thread to sleep for the specified number of milliseconds
  2. **sleep(long millis, int nanos) -** Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

similarity between [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) and [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html):

**>** yield() and sleep() method **belongs to java.lang.Thread** class.

**>**yield() and sleep() method can be **called from outside synchronized block**.

**>** yield() and sleep() method are **called on Threads not objects**.

**Question 32. Mention some g**[**uidelines to write thread safe code, most important point we must take care of in multithreading programs**](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)**?**

1. **Answer.**  In multithreading environment it’s important very important to [write thread safe code](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html), thread unsafe code can cause a major threat to your application. I have posted many articles regarding thread safety. So overall this will be revision of what we have learned so far i.e. writing thread safe healthy code and avoiding any kind of [deadlocks](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html).If method is exposed in multithreading environment and it’s not synchronized (thread unsafe) than it might lead us to [race condition](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html), we must try to use [synchronized block and **synchronized methods**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). [Multiple threads may exist on same object](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html) but only one thread of that object can enter **synchronized method** at a time, though  [threads on different object](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) can enter same method at same time.
2. Even static variables are not thread safe, they are used in static methods and if static methods are not synchronized then thread on same or different object can enter method concurrently. Multiple threads may exist on [same](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) or [different objects](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) of class but only one thread can enter [**static synchronized method**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) at a time, we must consider making [static methods as synchronized](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).

If possible, try to use [**volatile** variables](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html). If a field is declared volatile all threads see a consistent value for the variable. Volatile variables at times can be used as alternate to synchronized methods as well.

**Final variables** are thread safe because once assigned some reference of object they cannot point to reference of other object.

s is pointing to String object.

**publicclass** MyClass {

**final** String s = **new** String("a");

**void** method() {

s = "b"; // compilation error, s cannot point to new reference.

}

}

If final is holding some primitive value it cannot point to other value.

**publicclass** MyClass {

**finalint**i=0;

**void** method(){

i=0; //compilation error, i cannot point to new value.

}

}

1. Usage of **local variables** : If possible try to use local variables, local variables are thread safe, because every [thread has its own **stack**](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html), i.e. every thread has its own local variables and its pushes all the local variables on stack.

**publicclass** MyClass {

**void** method(){

**int**i=0; //Local variable, is thread safe.

}

}

1. We must avoid using  [**deadlock prone**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) deprecated thread methods such as [destroy()](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html), [stop()](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html), [suspend() and resume()](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html).
2. Using thread safe **collections** : Rather than using ArrayList we must Vector and in place of using HashMap we must use ConcurrentHashMap or HashTable.
3. We must use [VisualVM](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)  or [jstack](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html)  to detect problems such as deadlocks and time taken by threads to complete in multi threading programs.
4. Using [*ThreadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html):ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.
5. Rather than StringBuffer try using **immutable classes** such as String. Any change to String produces new String.

**Question 33. How thread can enter waiting, sleeping and blocked state and how can they go to runnable state ?**

**Answer.**  This is very prominently asked question in interview which will test your knowledge about [thread states](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). And it’s very important for developers to have in depth knowledge of this [thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) transition. I will try to explain this thread state transition by framing few sub questions. I hope reading sub questions will be quite interesting.

**> How can Thread go from running to waiting state ?**

  By calling **wait()**[method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

**> How can Thread return from waiting to runnable state ?**

  Once **notify() or notifyAll()**[method](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) is called object monitor/lock becomes available and thread can again return to runnable state.

**> How can Thread go from running to sleeping state ?**

  By calling **sleep()** [method](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) thread go from running to [sleeping](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) state. In sleeping state it will wait for sleep time to get over.

**> How can Thread return from sleeping to runnable state ?**

  Once specified **sleep time is up** thread can again return to runnable state.

**Suspend()** [method](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) can be used to put thread in waiting state and **resume()** method is the only way which could put thread in runnable state.

Thread also may go from running to waiting state if it is waiting for some I/O operation to take place. Once input is available thread may return to running state.

**>**When threads are in running state, **yield()**[method](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) can make thread to go in Runnable state.

**Question 34. Difference between notify() and notifyAll() methods, can you write a code to prove your point?**

**Answer.** Goodness. Theoretically you must have heard or you must be aware of differences between [notify() and notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html).But have you created program to achieve it? If not let’s do it.

First, I will like give you a brief description of what notify() and notifyAll() methods do.

**notify()**- Wakes up a single thread that is [waiting](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is random and occurs at the discretion of the implementation. A thread [waits](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) on an object's monitor by calling one of the wait methods.

[**The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.**](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| **publicfinalnativevoid**notify(); |

**notifyAll()**- Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

[**The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.**](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| **publicfinalnativevoid**notifyAll(); |

[Now it’s time to write down a program to prove the point.](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html)

**Question 35. Does thread leaves object lock when** [**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **method is called?**

**Answer.** When [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method is called Thread does not leaves object lock and goes from running to waiting state. Thread [waits](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for sleep time to over and once sleep time is up it goes from [waiting to runnable state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**Question 36. Does thread leaves object lock when wait() method is called?**

**Answer.** When [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method is called Thread leaves the object lock and goes from [running to waiting state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). Thread waits for other threads on same object to call notify() or notifyAll() and once any of [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) is called it goes from waiting to runnable state and again acquires object lock.

**Question 37. What will happen if we don’t override run method?**

**Answer.**  This question will test your basic knowledge how start and run methods work internally in Thread Api.

**When we call start() method** on thread, **it internally calls run() method** with newly created thread. **So, if we don’t override run() method newly created thread won’t be called and nothing will happen**.

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

//don't override run() method

}

**publicclass**DontOverrideRun {

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

System.*out*.println("main has started.");

MyThread thread1=**new** MyThread();

thread1.start();

System.*out*.println("main has ended.");

}

}

|  |
| --- |
| /\*OUTPUT  main has started.  main has ended.  \*/ |

As we saw in output, we didn’t override run() method that’s why on calling start() method nothing happened.

**Question 38. What will happen if we override start method?**

**Answer.** This question will again test your basic core java knowledge how overriding works at runtime, what what will be called at runtime and how start and run methods work internally in Thread Api.

**When we call start() method** on thread, **it internally calls run()** method with newly created thread. **So, if we override start() method, run() method will not be called** until we write code for calling run() method.

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

@Override

**publicvoid** run() {

System.*out*.println("in run() method");

}

@Override

**publicvoid** start(){

System.*out*.println("In start() method");

}

}

**publicclass**OverrideStartMethod {

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

System.*out*.println("main has started.");

MyThread thread1=**new** MyThread();

thread1.start();

System.*out*.println("main has ended.");

}

}

|  |
| --- |
| /\*OUTPUT  main has started.  In start() method  main has ended.  \*/ |

If we note output. we have overridden start method and didn’t called run() method from it, so, run() method wasn’t call.

**Question 39. Can we acquire lock on class? What are ways in which you can acquire lock on class?**

**Answer.**  **Yes**, we can acquire lock on [class’s class object in 2 ways to acquire lock on class](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).

1. Thread can acquire lock on class’s class object by-Entering **synchronized block or**

     Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass

**synchronized** (MyClass.class) {

        //thread has acquired lock on MyClass’s class object.

      }

by entering **static synchronized methods.**

**public staticsynchronizedvoid** method1() {

        //thread has acquired lock on MyRunnable’s class object.

      }

As soon as thread entered Synchronization method, thread acquired lock on class’s class object.

Thread will leave lock when it exits static synchronized method.

**Question 40. Difference between object lock and class lock?**

**Answer.**  It is very important question from multithreading point of view. We must understand [difference between object lock and class lock](http://www.javamadesoeasy.com/2015/03/difference-between-object-lock-and.html) to answer interview, ocjp answers correctly.

|  |  |
| --- | --- |
| [**Object lock**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) | [**Class lock**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) |
| Thread can acquire [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) by-   1. Entering **synchronized block or** 2. by entering **synchronized methods.** | Thread can acquire lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) by-   1. Entering **synchronized block or** 2. by entering **static synchronized methods.** |
| [Multiple threads may exist on same object but only one thread of that object can enter **synchronized method** at a time.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)  [Threads on different object can enter same method at same time.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) | Multiple threads may exist on [same](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) or [different objects](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) of class but only one thread can enter **static synchronized method** at a time. |
| **Multiple objects of class may exist and every object has it’s own lock.** | **Multiple objects of class may exist but there is always one class’s class object lock available**. |
| First let’s acquire [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) by entering **synchronized block**.  Example- Let’s say there is one class MyClassand we have created it’s object and reference to that object is myClass. Now we can create synchronization block, and parameter passed with synchronization tells which object has to be synchronized. In below code, we have synchronized object reference by myClass.  MyClass myClass=**new**Myclass();  **synchronized** (myClass) {      }  As soon thread entered Synchronization block, thread acquired object lock on object referenced by myClass (by acquiring object’s monitor.)  Thread will leave lock when it exits synchronized block. | First let’s acquire lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) by entering **synchronized block.**  Example- Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass  **synchronized** (MyClass.class) {     }  As soon as thread entered Synchronization block, thread acquired MyClass’s class object. Thread will leave lock when it exits synchronized block. |
| **publicsynchronizedvoid** method1() {  }  As soon as thread entered **Synchronization method**, thread acquired [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html).  Thread will leave lock when it exits synchronized method. | **public staticsynchronizedvoid** method1() {}  As soon as thread entered **static Synchronization method**, thread acquired lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).  Thread will leave lock when it exits synchronized method. |

**Let’s me give you some tricky situation based question,**

**Question 41.** Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **synchronized method2()** at same time?

**Answer.No**, here when Thread-1 is in **synchronized method1()** it must be **holding** [**lock on object’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and will release lock on object’s monitor only when it exits **synchronized method1()**. So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) for Thread-1 to release lock on object’s monitor so that it could enter **synchronized method2()**.

**Likewise**, Thread-2 even cannot enter **synchronized method1()** which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on object’s monitor so that it could enter **synchronized method1()**. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)

**Question 42.** Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **static synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

**Answer.No**, here when Thread-1 is in **static synchronized method1()** it must be **holding lock on** [**class class’s object**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and will release lock on class’s classobject only when it exits **static synchronized method1()**. So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) for Thread-1 to release lock on class’s classobject so that it could enter **static synchronized method2()**.

**Likewise**, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on  class’s classobject so that it could enter **static synchronized method1()**. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html)

**Question 43.** Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

**Answer.Yes**, here when Thread-1 is in **synchronized method1()** it must be **holding** [**lock on object’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and Thread-2 can enter **static synchronized method2()** by acquiring lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_65.html)

**Question 44.** Suppose you have thread and it is in **synchronized method** and now can thread **enter other synchronized method** from that method?

**Answer.Yes**, here when thread is in **synchronized method** it must be **holding** [**lock on object’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and **using that lock** thread can **enter other synchronized method**. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_5.html)

**Question 45.** Suppose you have thread and it is in **static synchronized method** and now can thread **enter other static synchronized method** from that method?

**Answer.**  **Yes**, here when thread is in **static synchronized method** it must be **holding lock on** [**class’s class object**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and **using that lock** thread can **enter other static synchronized method**. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_16.html)

**Question 46.** Suppose you have thread and it is in **static synchronized method** and now can thread **enter other non static synchronized method** from that method?

**Answer.Yes**, here when thread is in **static synchronized method** it must be **holding lock on** [**class’s class object**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and when it **enters synchronized method** it will **hold** [**lock on object’s monitor**](http://v/) **as well**.

So, now thread holds 2 locks (it’s also called nested synchronization)-

**>**first one on **class’s class object.**

**>**second one on **object’s monitor** (This lock will be released when thread exits non static method)**.**[Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_41.html)

**Question 47.** Suppose you have thread and it is in **synchronized method** and now can thread **enter other static synchronized method** from that method?

**Answer.Yes**, here when thread is in synchronized method it must be holding [**lock on object’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and when it enters static synchronized method it will hold lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) as well.

So, now thread holds 2 locks (it’s also called nested synchronization)-

**>**first one on [**object’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)**.**

**>**second one on **class’s class object.**(This lock will be released when thread exits static method)**.**[Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_17.html)

**Question 48.** Suppose you have **2 threads (Thread-1 on object1 and Thread-2 on object2)**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **synchronized method2()** at same time?

**Answer.Yes**, here when Thread-1 is in **synchronized method1()** it must be **holding** [**lock on object1’s monitor**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). Thread-2 will acquire lock on **object2’s monitor** and enter **synchronized method2()**.

**Likewise**, Thread-2 even enter **synchronized method1()** as well which is being executed by Thread-1 (because threads are created on different objects). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html)

**Question 49.** Suppose you have **2 threads (Thread-1 on object1 and Thread-2 on object2)**. **Thread-1** is in **static synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

**Answer.No**, it might confuse you a bit that threads are created on different objects. But, not to forgot that **multiple objects may exist but there is always one** [**class’s class object**](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) **lock available**.

Here, when Thread-1 is in **static synchronized method1()** it must be **holding lock on class class’s object** and will release lock on class’s classobject only when it exits **static synchronized method1()**. So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on class’s classobject so that it could enter **static synchronized method2()**.

**Likewise**, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on  [class’s classobject](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) so that it could enter **static synchronized method1()**. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html)

**Question 50. Difference between wait() and wait(long timeout), What are** [**thread states**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **when these method are called?**

**Answer.**

|  |  |
| --- | --- |
| [**wait()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) | **wait(long timeout)** |
| When [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method is called on object, it causes causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. | **wait(long timeout) -** Causes the current thread to wait until either another thread invokes the notify() or notifyAll() methods for this object, or a specified timeout time has elapsed. |
| **When** [**wait()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) **is called** on object - Thread enters from [**running to waiting state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).  **It** [**waits**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) **for some other thread to call notify so that it could enter runnable state**. | **When wait(1000) is called** on object - Thread enters from **running to waiting state**. Than **even if notify() or notifyAll() is not called after  timeout time has elapsed thread will go from** [**waiting to runnable state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)**.** |

**Question 51.  How can you implement your own Thread Pool in java?**

**Answer.***What is* [*ThreadPool*](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html)*?*

ThreadPool is a pool of threads which **reuses a fixed number of threads**  to execute tasks. At any point, **at most nThreads threads will be active processing tasks**. **If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available**.

ThreadPool implementation internally uses [LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/custom-implementation-of.html) for adding and removing tasks.

In this post i will be using LinkedBlockingQueue provide by java Api, you can refer this post for [implementing ThreadPool using custom LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/implementing-threadpool-using-custom.html).

*Need/Advantage of ThreadPool?*

**Instead of creating new thread every time for executing tasks**, we can create **ThreadPool** which **reuses a fixed number of threads for executing tasks**.

As threads are reused, performance of our application improves drastically.

*How ThreadPool works?*

We will instantiate ThreadPool, in ThreadPool’s **constructor** nThreads number of threads are created and started.

|  |
| --- |
| ThreadPool threadPool=**new** ThreadPool(2); |

Here 2 threads will be created and started in ThreadPool.

Then, threads will enter **run()** method of **ThreadPoolsThread** class and will call take() method on taskQueue.

* If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable).

**publicvoid** run() {

. . .

**while** (**true**) {

        . . .

**Runnable runnable = taskQueue.take();**

**runnable.run();**

        . . .

    }

. . .

}

* Else waits for tasks to become available.

**When tasks are added?**

When execute() method of **ThreadPool** is called, it internally calls put() method on taskQueue to add tasks.

|  |
| --- |
| taskQueue.put(task); |

Once tasks are available all waiting threads are notified that task is available.

More detail on how to [Implement Thread pool in java](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html).

**Question 52.  What is significance of using** [**ThreadLocal**](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)**?**

**Answer.**  This question will test your command in multi threading, can you really create some perfect multithreading application or not. [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) is a class which provides thread-local variables.

*What is* [*ThreadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) *?*

ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.

*For how long Thread holds ThreadLocal value?*

Thread holds ThreadLocal value till it hasn’t entered [dead state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

*Can one thread see other thread’s ThreadLocal value?*

**No**, thread can see only it’s ThreadLocal value.

*Are ThreadLocal variables thread safe. Why?*

**Yes**, ThreadLocal variables are thread safe. As every thread has its own ThreadLocal value and one thread can’t see other threads ThreadLocal value.

*Application of* [*ThreadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)*?*

1. ThreadLocal are **used by many web frameworks** for maintaining some context (may be session or request) related value.
   * In any **single threaded application**, same thread is assigned for every request made to same action, so ThreadLocal values will be available in next request as well.
   * In **multi threaded application**, different thread is assigned for every request made to same action, so ThreadLocal values will be different for every request.

When threads have started at different time they might like to store time at which they have started. **So, thread’s start time can be stored in ThreadLocal.**

*Creating ThreadLocal >*

|  |
| --- |
| **private** ThreadLocal<String>threadLocal =   **new** ThreadLocal<String>(); |

We will create instance of ThreadLocal. ThreadLocal is a generic class, i will be using String to demonstrate threadLocal.

**All threads will see same instance of ThreadLocal**, **but a thread will be able to see value which was set by it only**.

*How thread set value of ThreadLocal >*

|  |
| --- |
| threadLocal.set( **new** Date().toString()); |

Thread set value of ThreadLocal by calling set(“”) method on threadLocal.

*How thread get value of ThreadLocal >*

|  |
| --- |
| threadLocal.get() |

Thread get value of ThreadLocal by calling get() method on threadLocal.

See here for detailed explanation of [threadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html).

**Question 53. What is busy spin?**

**Answer.**

*What is* [*busy spin*](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html)*?*

When one thread loops continuously waiting for another thread to signal.

*Performance point of view* - Busy spin is **very bad** from performance point of view, because one thread keeps on looping continuously ( and consumes CPU) waiting for another thread to signal.

*Solution to busy spin -*

We must use [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) or [wait() and notify()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method. Using wait() is better option.

*Why using wait() and notify() is much better option to solve busy spin?*

Because in case when we use sleep() method, thread will wake up again and again after specified sleep time until boolean variable is true. But, in case of wait() thread will wake up only when when notified by calling [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html), hence end up consuming CPU in best possible manner.

*Program - Consumer Producer problem with busy spin >*

Consumer thread continuously execute (**busy spin**) in while loop till**productionInProcess** is true. Once producer thread has ended it will make boolean variable **productionInProcess** false and **busy spin** will be over.

|  |
| --- |
| **while(productionInProcess){**  **System.*out*.println("BUSY SPIN - Consumer waiting for production to get over");**  **}** |

[See here for Busy spin in detail](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html).

**Question 54. Can a constructor be synchronized?**

**Answer.**  No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

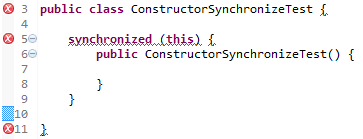
**Enclosing** constructor in synchronized block will generate compilation error.

Using synchronized in **constructor definition** will also show compilation error.

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted

**Though we can use synchronized block inside constructor.**

Any attempt to do so is against java coding syntax.



***Using synchronized in constructor definition will also show compilation error >***

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted

**Question 55. Can you find whether thread holds lock on object or not?**

**Answer.**  holdsLock(object) method can be used to find out whether current thread holds the lock on monitor of specified object.

holdsLock(object) method returns true if the current thread holds the lock on monitor of specified object.

**Question 56. What do you mean by thread starvation?**

**Answer.**   When thread does not enough CPU for its execution **Thread starvation happens.**

**Thread starvation** may happen in following scenarios >

* Low priority threads gets less CPU (time for execution) as compared to high priority threads. **Lower priority thread** may **starve** away waiting to get enough CPU to perform calculations.
* In [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) two threads waits for each other to release lock holded by them on resources. There both **Threads starves away to get CPU.**
* Thread might be waiting indefinitely for lock on object’s monitor (by calling [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method), because no other thread is calling [notify()/notifAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on object. In that case, **Thread starves** away to get CPU.
* Thread might be waiting indefinitely for lock on object’s monitor (by calling wait() method), but notify() may be repeatedly awakening some other threads. In that case also **Thread starves** away to get CPU.

**Question 57. What is addShutdownHook method in java?**

**Answer.**  [addShutdownHook](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html) method in java >

* addShutdownHook method **registers a new virtual-machine shutdown hook**.
* A shutdown hook is a **initialized but unstarted thread**.
* When **JVM starts its shutdown** it will **start all registered shutdown hooks** in some unspecified order and let them run concurrently.
* When JVM (Java virtual machine)  shuts down >
* When the last non-[daemon](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) thread finishes, or
* when the System.exit is called.

*Once JVM’s shutdown has begun***new shutdown hook cannot be registered** neither  **previously-registered hook can be de-registered**. Any attempt made to do any of these operations causes an IllegalStateException.

*For more detail with program read :* [***Threads addShutdownHook method in java***](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html)

**Question 58. How you can handle uncaught runtime exception generated in run method?**

**Answer.**  We can use [setDefaultUncaughtExceptionHandler](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html) method which can handle uncaught unchecked(runtime) exception generated in run() method.

What is setDefaultUncaughtExceptionHandler method?

setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.

*setDefaultUncaughtExceptionHandler method features >*

* **setDefaultUncaughtExceptionHandler** method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.
* **setDefaultUncaughtExceptionHandler** is a static method method, so we can directly call  Thread.***setDefaultUncaughtExceptionHandler*** to set the default handler to handle uncaught unchecked(runtime) exception.
* It avoids abrupt termination of thread caused by uncaught runtime exceptions.

Defining setDefaultUncaughtExceptionHandler method >

|  |
| --- |
| Thread.***setDefaultUncaughtExceptionHandler***(**new** Thread.UncaughtExceptionHandler(){  **publicvoid** uncaughtException(Thread thread, Throwable throwable) {    System.*out*.println(thread.getName() + " has thrown " + throwable);    }    }); |

*For more detail read :* [*Program to demonstrate* ***setDefaultUncaughtExceptionHandler*** *method.*](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html)

**Question 59. What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?**

**Answer.**  When program starts **JVM creates  a ThreadGroup** named ***main***. Unless specified, all  newly created threads become members of the ***main*** thread group.

**ThreadGroup is initialized with default priority of 10.**

*ThreadGroup* ***important methods*** *>*

* **getName()** 
  + name of ThreadGroup.
* **activeGroupCount()**
  + count of active groups in ThreadGroup.
* **activeCount()**
  + count of active threads in ThreadGroup.
* **list()**
  + list() method has prints ThreadGroups information
* **getMaxPriority()**
  + Method returns the maximum priority of ThreadGroup.
* **setMaxPriority(int pri)**
  + Sets the maximum priority of ThreadGroup.

*Read more about* [***ThreadGroup in java***](http://www.javamadesoeasy.com/2015/03/threadgroup-in-java.html)*.*

**Question 60. What are thread priorities?**

**Answer.**

[*Thread Priority*](http://www.javamadesoeasy.com/2015/03/thread-priorities-setpriority-and.html) *range is from 1 to 10.*

Where **1 is minimum priority** and **10 is maximum priority.**

Thread class provides variables of **final static int** type for setting thread priority.

|  |
| --- |
| /\* The minimum priority that a thread can have. \*/  **publicfinalstaticint*MIN\_PRIORITY***= 1;    /\* The default priority that is assigned to a thread. \*/  **publicfinalstaticint*NORM\_PRIORITY***= 5;     /\* The maximum priority that a thread can have. \*/  **publicfinalstaticint*MAX\_PRIORITY***= 10; |

Thread with **MAX\_PRIORITY is likely to get more CPU** as compared to low priority threads. But **occasionally low priority thread might get more CPU**. Because thread scheduler schedules thread on discretion of implementation and [thread behaviour is totally unpredictable](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html).

Thread with **MIN\_PRIORITY is likely to get less CPU** as compared to high priority threads. But **occasionally high priority thread might less CPU**. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.

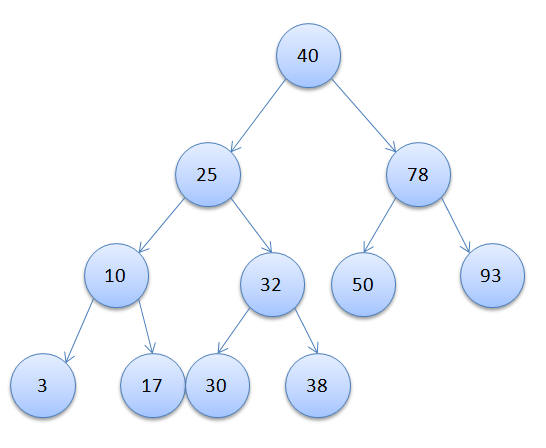
***setPriority()* method is used for Changing the priority of thread.**

***getPriority()* method returns the thread’s priority.**

**Binary Search Tree (BST)**

**What is a Binary Search Tree (BST)?**

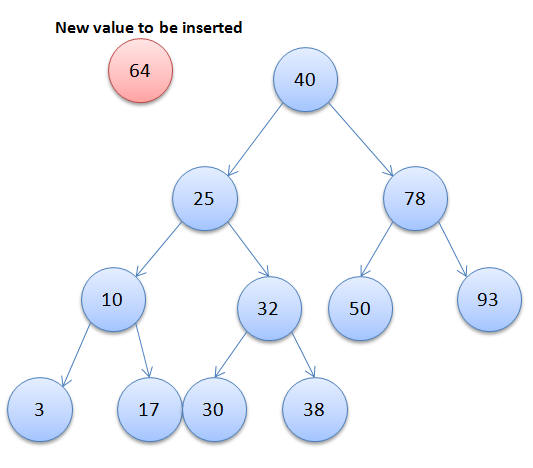
**Binary Search Tree (BST) is a binary tree data structure with a special feature where in the value store at each node is greater than or equal to the value stored at its left sub child and lesser than the value stored at its right sub child**. Lets look at an example of a BST:

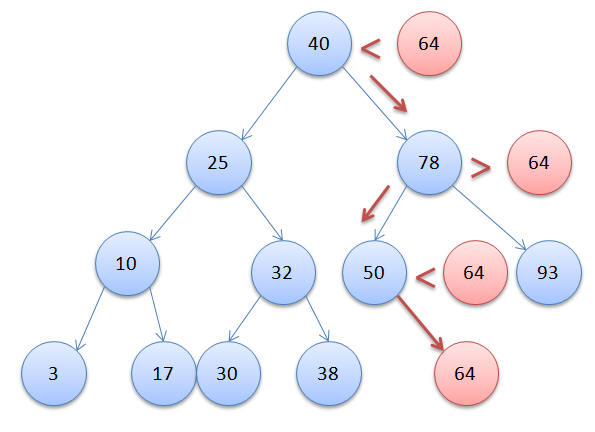


**In the above example you can see that at each node the value in the left child is lesser than or equal to the value in the node and the value in the right child is greater than the value in the node.**

## Building a Binary Search Tree (BST)

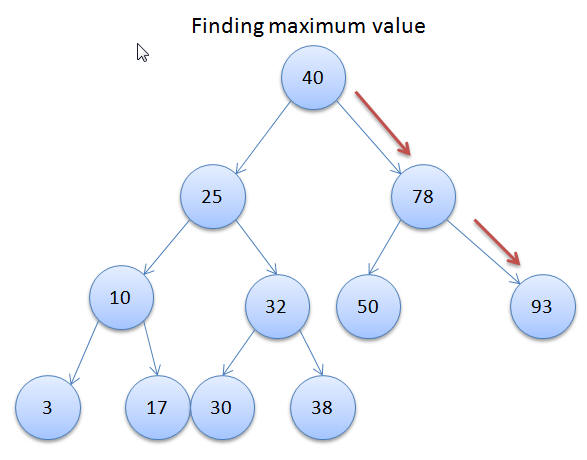
Now that we have seen how a BST looks, let me show you how one can build a BST and insert nodes into the tree by implementing the algorithm in Java. **The basic idea is that at each node we compare with the value being inserted. If the value is lesser then we traverse through the left sub tree and if the value is greater we traverse through the right subtree.** Suppose we have to insert the value **64** in the above BST, lets look at the nodes traversed before its inserted at the right place:

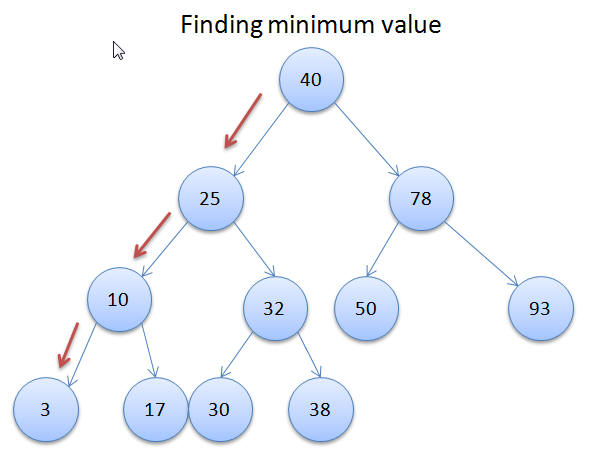




## Finding Maximum and Minimum Value in BST

If you have noticed in the above example that the leftmost node has the lowest value and the rightmost node has the highest value. This is due to the sorted nature of the tree.





Using this principle the below methods return us the lowest and highest value in the Binary Search Tree:

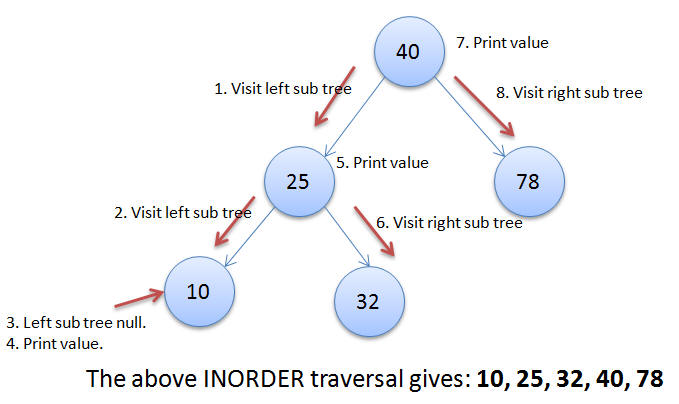
## Traversing the Binary Search Tree (BST)

Traversing the tree or BST in this case is visiting each of the nodes present in the tree and performing some operation with the value present in the node which in this case will be printing the value present in the node. **When we traverse the tree we have to visit the value present in the node, then node’s right sub tree and the left sub tree**. Visiting the right and left sub tree will be a recursive operation. The order in which we perform the three operations i.e visiting the value, right sub tree and left sub tree gives rise to three traversal techniques:

1. **Inorder Traversal**
2. **Preorder Traversal**
3. **Postorder Traversal**

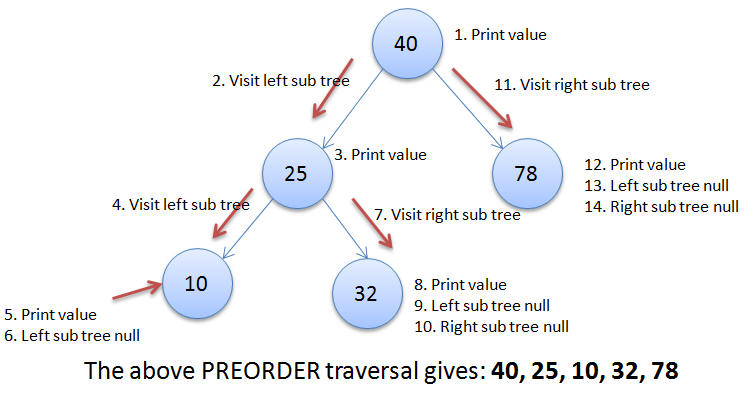
## Inorder Traversal

In this traversal the left sub tree of the given node is visited first, then the value at the given node is printed and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



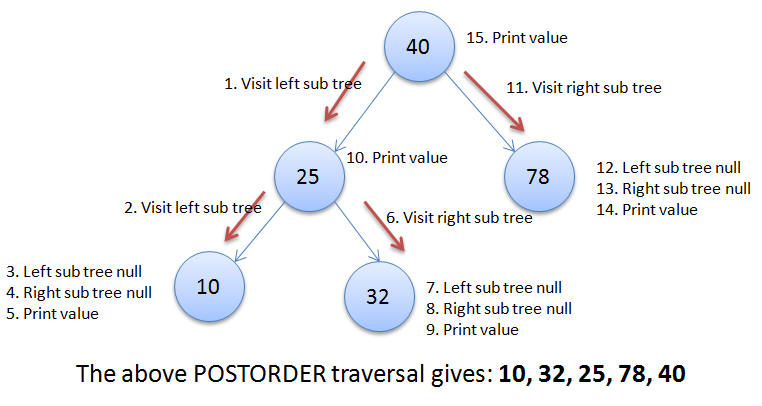
## Preorder traversal

In this traversal the value at the given node is printed first and then the left sub tree of the given node is visited and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



**Postorder Traversal**

In this traversal the left sub tree of the given node is traversed first, then the right sub tree of the given node is traversed and then the value at the given node is printed. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



**package** com.test.binary.tree;

**publicclass** BinaryTree {

**private** Node root;

**publicvoid** addNode(**int** key, String name) {

Node newNode = **new** Node(key, name);

**if** (root == **null**) {

root = newNode;

} **else** {

Node focus = root;

Node parent;

**while** (**true**) {

parent = focus;

**if** (key < focus.key) {

focus = focus.left;

**if** (focus == **null**) {

parent.left = newNode;

**return**;

}

} **else** {

focus = focus.right;

**if** (focus == **null**) {

parent.right = newNode;

**return**;

}

}

}

}

}

/\*\*

\* Aim for in order traversal Start at lowest left child

\*\* 50

\* 25 75

\* 15 30 85

\* also known as left-->root-->right

\*

\* **@param** focus

\*/

**publicvoid** inOrderTraverseTree(Node focus) {

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

inOrderTraverseTree(focus.left);

System.*out*.println(focus);

inOrderTraverseTree(focus.right);

}

}

/\*\*

\* root-->left-->right

\*

\* **@param** focus

\*/

**publicvoid** preOrderTraverseTree(Node focus) {

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

System.*out*.println(focus);

preOrderTraverseTree(focus.left);

preOrderTraverseTree(focus.right);

}

}

/\*\*

\* left-->right-->root

\*

\* **@param** focus

\*/

**publicvoid** postOrderTraverseTree(Node focus) {

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

postOrderTraverseTree(focus.left);

postOrderTraverseTree(focus.right);

System.*out*.println(focus);

}

}

**public** Node findNode(**int** key){

Node focus=root;

**while**(focus.key!=key){

**if**(key < focus.key){

focus=focus.left;

}**else**{

focus=focus.right;

}

**if**(focus==**null**){

**returnnull**;

}

}

**return** focus;

}

/\*\*

\* 50

\* 25 75

\* 15 30 85

\*

\* **@param** args

\*/

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

BinaryTree bTree = **new** BinaryTree();

bTree.addNode(50, "Boss");

bTree.addNode(25, "Vice President");

bTree.addNode(15, "Office Manager");

bTree.addNode(30, "Seceratory");

bTree.addNode(75, "Sales Manager");

bTree.addNode(85, "Sales Man1");

System.*out*.println("InOrder Traversal\n");

bTree.inOrderTraverseTree(bTree.root);

System.*out*.println("\nPreOrder Traversal\n");

bTree.preOrderTraverseTree(bTree.root);

System.*out*.println("\nPostOrder Traversal\n");

bTree.postOrderTraverseTree(bTree.root);

**for**(**int** i=15;i<85;i=i+5){

System.*out*.println("search -->"+bTree.findNode(i));

}

}

**staticclass** Node {

**int**key;

String name;

Node left;

Node right;

Node(**int** key, String name) {

**this**.key = key;

**this**.name = name;

}

@Override

**public** String toString() {

**return**name + " has key " + key;

}

}

}

# **Red–black tree**

A **red–black tree** is a kind of [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

 Following are the properties for a red–black tree:

1. A node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) from a given node to any of its descendant NIL nodes contains the same number of black nodes. Some definitions: the number of black nodes from the root to a node is the node's **black depth**; the uniform number of black nodes in all paths from root to the leaves is called the **black-height** of the red–black tree.

# **Red-black tree (Java)**

A [red-black tree](http://en.wikipedia.org/wiki/Red-black_tree) is a type of self-balancing binary search tree typically used to implement associative arrays. It has O(log *n*) worst-case time for each operation and is quite efficient in practice. Unfortunately, it's also quite complex to implement, requiring a number of subtle cases for both insertion and deletion.

This article walks through a Java implementation of red-black trees, organized in a way to make correctness and completeness easier to understand. For all practical purposes you should use Java's[TreeMap](http://java.sun.com/javase/6/docs/api/java/util/TreeMap.html) class instead.

## Node structure and node relationships

A red-black tree is a type of binary search tree, so each node in the tree has a parent (except the root node) and at most two children. The tree as a whole will be identified by its root node. For ease of implementation, we will have each node retain a pointer to both its children as well as its parent node (null for the root). Keeping parent nodes costs space and is not strictly necessary, but makes it easy to follow the tree in any direction without maintaining an auxiliary stack. Our red-black tree will implement an associative array, and we will allow key and values of any class type:

**<<****[RBTree.java](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:RBTree.java)>>=**

enumColor**{**RED, BLACK**}**

**class**Node<K**extends**Comparable<? **super**K>,V>

**{**

**public**Kkey;

**public**Vvalue;

**public**Node<K,V>left;

**public**Node<K,V>right;

**public**Node<K,V>parent;

**public**Colorcolor;

[node constructor](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:node constructor)

[node relationships](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:node relationships)

**}**

**publicclass**RBTree<K**extends**Comparable<? **super**K>,V>

**{**

[constants](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:constants)

**public**Node<K,V>root;

[create operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:create operation)

[verify properties functions](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:verify properties functions)

[lookup operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:lookup operation)

[rotation operations](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:rotation operations)

[replace node operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:replace node operation)

[insertion operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:insertion operation)

[deletion operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:deletion operation)

[print operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:print operation)

[main method](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk def:main method)

**}**

Each node also stores its color, either red or black, using an enumeration. The role of the color bit will be explained in the properties. There is some internal fragmentation due to using a reference type to store a single bit, but we avoid optimizing this here for simplicity.

The parent of a node n is available by simply using n.parent. We're also interested in three other more complex relationships between nodes:

* The grandparent of a node, its parent's parent. We use assertions to make sure that we don't attempt to access the grandparent of a node that doesn't have one, such as the root node or its children:

**<<**[**node relationships**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:node relationships)**>>=**

**public**Node<K,V>grandparent**(){**

assertparent!=**null**;// Not the root node

assertparent.parent!=**null**;// Not child of root

**return**parent.parent;

**}**

1. The sibling of a node, defined as the other child of its parent. Note that the sibling *may* be null, if the parent has only one child.

**<<**[**node relationships**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:node relationships)**>>=**

**public**Node<K,V>sibling**(){**

assertparent!=**null**;// Root node has no sibling

**if(this**==parent.left**)**

**return**parent.right;

**else**

**return**parent.left;

**}**

1. The uncle of a node, defined as the sibling of its parent. The uncle may also be null, if the grandparent has only one child.

**<<****[node relationships](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:node relationships)>>=**

**public**Node<K,V>uncle**(){**

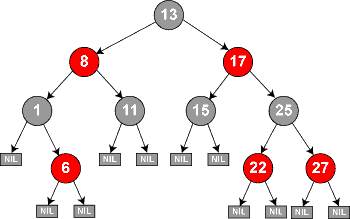
assertparent!=**null**;// Root node has no uncle

assertparent.parent!=**null**;// Children of root have no uncle

**return**parent.sibling**()**;

**}**

## Properties

[](http://en.literateprograms.org/File:Red-black_tree_example.png)

We will at all times enforce the following five properties, which provide a theoretical guarantee that the tree remains balanced. We will have a helper function verifyProperties() that asserts all five properties in a debug build, to help verify the correctness of our implementation and formally demonstrate their meaning. Note that many of these tests walk the tree, making them very expensive - for this reason we require the constant VERIFY\_RBTREE to be true to turn them on.

As shown, the tree terminates in NIL leaves, which we represent using null (we set the child pointers of their parents to null). In an empty tree, the root pointer is null. This saves substantial space compared to explicit representation of leaves.

We will implement each of the testing methods as static methods of the RBTree class that take a Node argument. We could have implemented them as methods of Node; but sometimes we want to test properties on leaf nodes, which we represent with null; and we can't invoke a method on a null reference, so it is easier this way.

**<<**[**constants**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:constants)**>>=**

**publicstaticfinalboolean**VERIFY\_RBTREE=**true**;

**<<**[**verify properties functions**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:verify properties functions)**>>=**

**publicvoid**verifyProperties**(){**

**if(**VERIFY\_RBTREE**){**

verifyProperty1**(**root**)**;

verifyProperty2**(**root**)**;

// Property 3 is implicit

verifyProperty4**(**root**)**;

verifyProperty5**(**root**)**;

**}**

**}**

1. Each node is either red or black:

Technically speaking, because the way colors are represented by the Color enumeration, which only has the values RED and BLACK, the only way this can fail is if the color reference is null.

**<<**[**verify properties functions**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:verify properties functions)**>>=**

**privatestaticvoid**verifyProperty1**(**Node<?,?>n**){**

assertnodeColor**(**n**)**==Color.RED||nodeColor**(**n**)**==Color.BLACK;

**if(**n==**null)return**;

verifyProperty1**(**n.left**)**;

verifyProperty1**(**n.right**)**;

**}**

1. The root node is black.

**<<**[**verify properties functions**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:verify properties functions)**>>=**

**privatestaticvoid**verifyProperty2**(**Node<?,?>root**){**

assertnodeColor**(**root**)**==Color.BLACK;

**}**

1. All leaves (shown as NIL in the above diagram) are black and contain no data. Since we represent these empty leaves using null, this property is implicitly assured by always treating null as black. To this end we create a nodeColor() helper function:

**<<**[**verify properties functions**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:verify properties functions)**>>=**

**privatestatic**ColornodeColor**(**Node<?,?>n**){**

**return**n==**null** ? Color.BLACK:n.color;

**}**

1. Every red node has two children, and both are black (or equivalently, the parent of every red node is black).

**<<**[**verify properties functions**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:verify properties functions)**>>=**

**privatestaticvoid**verifyProperty4**(**Node<?,?>n**){**

**if(**nodeColor**(**n**)**==Color.RED**){**

assertnodeColor**(**n.left**)**==Color.BLACK;

assertnodeColor**(**n.right**)**==Color.BLACK;

assertnodeColor**(**n.parent**)**==Color.BLACK;

**}**

**if(**n==**null)return**;

verifyProperty4**(**n.left**)**;

verifyProperty4**(**n.right**)**;

**}**

1. All paths from any given node to its leaf nodes contain the same number of black nodes. This one is the trickiest to verify; we do it by traversing the tree, incrementing a black node count as we go. The first time we reach a leaf we save the count. We return the count so that when we subsequently reach other leaves, we compare the count to the saved count.

**<<****[verify properties functions](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:verify properties functions)>>=**

**privatestaticvoid**verifyProperty5**(**Node<?,?>root**){**

verifyProperty5Helper**(**root, 0, -1**)**;

**}**

**privatestaticint**verifyProperty5Helper**(**Node<?,?>n, **int**blackCount, **int**pathBlackCount**){**

**if(**nodeColor**(**n**)**==Color.BLACK**){**

blackCount++;

**}**

**if(**n==**null){**

**if(**pathBlackCount==-1**){**

pathBlackCount=blackCount;

**}else{**

assertblackCount==pathBlackCount;

**}**

**return**pathBlackCount;

**}**

pathBlackCount=verifyProperty5Helper**(**n.left, blackCount, pathBlackCount**)**;

pathBlackCount=verifyProperty5Helper**(**n.right, blackCount, pathBlackCount**)**;

**return**pathBlackCount;

**}**

Properties 4 and 5 together guarantee that no path in the tree is more than about twice as long as any other path, which guarantees that it has O(log *n*) height.

## Operations

### Construction

The constructor of RBTree initializes an empty tree, which is represented by a tree with a null root.

**<<****[create operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:create operation)>>=**

**public**RBTree**(){**

root=**null**;

verifyProperties**()**;

**}**

We have a constructor for Node:

**<<****[node constructor](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:node constructor)>>=**

**public**Node**(**Kkey, Vvalue, ColornodeColor, Node<K,V>left, Node<K,V>right**){**

**this**.key=key;

**this**.value=value;

**this**.color=nodeColor;

**this**.left=left;

**this**.right=right;

**if(**left!=**null)**left.parent=**this**;

**if(**right!=**null)**right.parent=**this**;

**this**.parent=**null**;

**}**

### Search

Read-only operations on a red-black tree, such as searching for a key and getting the corresponding value, require no modification from those used for binary search trees, because every red-black tree is a specialization of a simple binary search tree.

We begin by creating a helper function that gets a pointer to the node with a given key. If the key is not found, it returns null. This will be useful later for deletion:

**<<**[**lookup operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:lookup operation)**>>=**

**private**Node<K,V>lookupNode**(**Kkey**){**

Node<K,V>n=root;

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

**int**compResult=key.compareTo**(**n.key**)**;

**if(**compResult==0**){**

**return**n;

**}elseif(**compResult<0**){**

n=n.left;

**}else{**

assertcompResult>0;

n=n.right;

**}**

**}**

**return**n;

**}**

We are using the natural ordering of the key type K, which we have required earlier to implement the Comparable interface.

Now looking up a value is straightforward, by finding the node and extracting the data if lookup succeeded. We return null if the key was not found (implying that null cannot be used as a value unless all lookups are expected to succeed).

**<<****[lookup operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:lookup operation)>>=**

**public**Vlookup**(**Kkey**){**

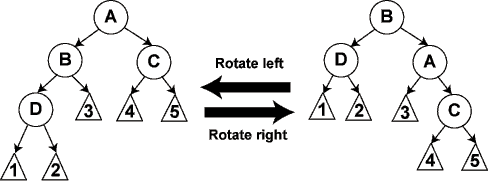
Node<K,V>n=lookupNode**(**key**)**;

**return**n==**null** ? **null**:n.value;

**}**

### Rotations

Both insertion and deletion rely on a fundamental operation for reducing tree height called a *rotation*. A rotation locally changes the structure of the tree without changing the in-order order of the sequence of values that it stores.

[](http://en.literateprograms.org/File:Tree_rotation.png)

We create two helper functions, one to perform a left rotation and one to perform a right rotation; each takes the highest node in the subtree as an argument:

**<<****[rotation operations](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:rotation operations)>>=**

**privatevoid**rotateLeft**(**Node<K,V>n**){**

Node<K,V>r=n.right;

replaceNode**(**n, r**)**;

n.right=r.left;

**if(**r.left!=**null){**

r.left.parent=n;

**}**

r.left=n;

n.parent=r;

**}**

**privatevoid**rotateRight**(**Node<K,V>n**){**

Node<K,V>l=n.left;

replaceNode**(**n, l**)**;

n.left=l.right;

**if(**l.right!=**null){**

l.right.parent=n;

**}**

l.right=n;

n.parent=l;

**}**

Here, replaceNode() is a helper function that cuts a node away from its parent, substituting a new node (or null) in its place. It simplifies consistent updating of parent and child pointers. It needs the tree passed in because it may change which node is the root.

**<<****[replace node operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:replace node operation)>>=**

**privatevoid**replaceNode**(**Node<K,V>oldn, Node<K,V>newn**){**

**if(**oldn.parent==**null){**

root=newn;

**}else{**

**if(**oldn==oldn.parent.left**)**

oldn.parent.left=newn;

**else**

oldn.parent.right=newn;

**}**

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

newn.parent=oldn.parent;

**}**

**}**

We'll find replaceNode() useful again later on when discussing insertion and deletion.

### Insertion

When inserting a new value, we first insert it into the tree as we would into an ordinary binary search tree. If the key already exists, we just replace the value (since we're implementing an associative array). Otherwise, we find the place in the tree where the new pair belongs, then attach a newly created red node containing the value:

**<<**[**insertion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:insertion operation)**>>=**

**publicvoid**insert**(**Kkey, Vvalue**){**

Node<K,V>insertedNode=**new**Node<K,V>**(**key, value, Color.RED, **null**, **null)**;

**if(**root==**null){**

root=insertedNode;

**}else{**

Node<K,V>n=root;

**while(true){**

**int**compResult=key.compareTo**(**n.key**)**;

**if(**compResult==0**){**

n.value=value;

**return**;

**}elseif(**compResult<0**){**

**if(**n.left==**null){**

n.left=insertedNode;

**break**;

**}else{**

n=n.left;

**}**

**}else{**

assertcompResult>0;

**if(**n.right==**null){**

n.right=insertedNode;

**break**;

**}else{**

n=n.right;

**}**

**}**

**}**

insertedNode.parent=n;

**}**

insertCase1**(**insertedNode**)**;

verifyProperties**()**;

**}**

The problem is that the resulting tree may not satisfy our five red-black tree properties. The call to insertCase1() above begins the process of correcting the tree so that it satisfies the properties once more.

**Case 1:** In this case, the new node is now the root node of the tree. Since the root node must be black, and changing its color adds the same number of black nodes to every path, we simply recolor it black. Because only the root node has no parent, we can assume henceforth that the node has a parent.

**<<**[**insertion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:insertion operation)**>>=**

**privatevoid**insertCase1**(**Node<K,V>n**){**

**if(**n.parent==**null)**

n.color=Color.BLACK;

**else**

insertCase2**(**n**)**;

**}**

**Case 2:** In this case, the new node has a black parent. All the properties are still satisfied and we return.

**<<**[**insertion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:insertion operation)**>>=**

**privatevoid**insertCase2**(**Node<K,V>n**){**

**if(**nodeColor**(**n.parent**)**==Color.BLACK**)**

**return**;// Tree is still valid

**else**

insertCase3**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 3](http://en.literateprograms.org/File:Red-black_tree_insert_case_3.png)  **Case 3:** In this case, the uncle node is red. We recolor the parent and uncle black and the grandparent red. However, the red grandparent node may now violate the red-black tree properties; we recursively invoke this procedure on it from case 1 to deal with this. |

**<<**[**insertion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:insertion operation)**>>=**

**void**insertCase3**(**Node<K,V>n**){**

**if(**nodeColor**(**n.uncle**())**==Color.RED**){**

n.parent.color=Color.BLACK;

n.uncle**()**.color=Color.BLACK;

n.grandparent**()**.color=Color.RED;

insertCase1**(**n.grandparent**())**;

**}else{**

insertCase4**(**n**)**;

**}**

**}**

|  |
| --- |
| [Diagram of case 4](http://en.literateprograms.org/File:Red-black_tree_insert_case_4.png)  **Case 4:** In this case, we deal with two cases that are mirror images of one another:   * The new node is the right child of its parent and the parent is the left child of the grandparent. In this case we rotate left about the parent. * The new node is the left child of its parent and the parent is the right child of the grandparent. In this case we rotate right about the parent.   Neither of these fixes the properties, but they put the tree in the correct form to apply case 5. |

**<<**[**insertion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:insertion operation)**>>=**

**void**insertCase4**(**Node<K,V>n**){**

**if(**n==n.parent.right&&n.parent==n.grandparent**()**.left**){**

rotateLeft**(**n.parent**)**;

n=n.left;

**}elseif(**n==n.parent.left&&n.parent==n.grandparent**()**.right**){**

rotateRight**(**n.parent**)**;

n=n.right;

**}**

insertCase5**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 5](http://en.literateprograms.org/File:Red-black_tree_insert_case_5.png)  **Case 5:** In this final case, we deal with two cases that are mirror images of one another:   * The new node is the left child of its parent and the parent is the left child of the grandparent. In this case we rotate right about the grandparent. * The new node is the right child of its parent and the parent is the right child of the grandparent. In this case we rotate left about the grandparent.   Now the properties are satisfied and all cases have been covered. |

**<<****[insertion operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:insertion operation)>>=**

**void**insertCase5**(**Node<K,V>n**){**

n.parent.color=Color.BLACK;

n.grandparent**()**.color=Color.RED;

**if(**n==n.parent.left&&n.parent==n.grandparent**()**.left**){**

rotateRight**(**n.grandparent**())**;

**}else{**

assertn==n.parent.right&&n.parent==n.grandparent**()**.right;

rotateLeft**(**n.grandparent**())**;

**}**

**}**

Note that inserting is actually in-place, since all the calls above use tail recursion. Moreover, it performs at most two rotations, since the only recursive call occurs before making any rotations.

### Removal

We begin by finding the node to be deleted with lookupNode() and deleting it precisely as we would in a binary search tree. There are two cases for removal, depending on whether the node to be deleted has at most one, or two non-leaf children. A node with at most one non-leaf child can simply be replaced with its non-leaf child. When deleting a node with two non-leaf children, we copy the value from the in-order predecessor (the maximum or rightmost element in the left subtree) into the node to be deleted, and then we then delete the predecessor node, which has only one non-leaf child. This same procedure also works in a red-black tree without affecting any properties.

Error creating thumbnail: /bin/bash: rsvg: command not found

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**publicvoid**delete**(**Kkey**){**

Node<K,V>n=lookupNode**(**key**)**;

**if(**n==**null)**

**return**;// Key not found, do nothing

**if(**n.left!=**null**&&n.right!=**null){**

// Copy key/value from predecessor and then delete it instead

Node<K,V>pred=maximumNode**(**n.left**)**;

n.key=pred.key;

n.value=pred.value;

n=pred;

**}**

assertn.left==**null**||n.right==**null**;

Node<K,V>child=**(**n.right==**null)** ? n.left:n.right;

**if(**nodeColor**(**n**)**==Color.BLACK**){**

n.color=nodeColor**(**child**)**;

deleteCase1**(**n**)**;

**}**

replaceNode**(**n, child**)**;

verifyProperties**()**;

**}**

The maximumNode() helper function just walks right until it reaches the last non-leaf:

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatestatic**<K**extends**Comparable<? **super**K>,V>Node<K,V>maximumNode**(**Node<K,V>n**){**

assertn!=**null**;

**while(**n.right!=**null){**

n=n.right;

**}**

**return**n;

**}**

However, before deleting the node, we must ensure that doing so does not violate the red-black tree properties. If the node we delete is black, and we cannot change its child from red to black to compensate, then we would have one less black node on every path through the child node. We must adjust the tree around the node being deleted to compensate.

**Case 1:** In this case, N has become the root node. The deletion removed one black node from every path, so no properties are violated.

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatevoid**deleteCase1**(**Node<K,V>n**){**

**if(**n.parent==**null)**

**return**;

**else**

deleteCase2**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 2](http://en.literateprograms.org/File:Red-black_tree_delete_case_2.png)  **Case 2:** N has a red sibling. In this case we exchange the colors of the parent and sibling, then rotate about the parent so that the sibling becomes the parent of its former parent. This does not restore the tree properties, but reduces the problem to one of the remaining cases. |

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatevoid**deleteCase2**(**Node<K,V>n**){**

**if(**nodeColor**(**n.sibling**())**==Color.RED**){**

n.parent.color=Color.RED;

n.sibling**()**.color=Color.BLACK;

**if(**n==n.parent.left**)**

rotateLeft**(**n.parent**)**;

**else**

rotateRight**(**n.parent**)**;

**}**

deleteCase3**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 3](http://en.literateprograms.org/File:Red-black_tree_delete_case_3.png)  **Case 3:** In this case N's parent, sibling, and sibling's children are black. In this case we paint the sibling red. Now all paths passing through N's parent have one less black node than before the deletion, so we must recursively run this procedure from case 1 on N's parent. |

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatevoid**deleteCase3**(**Node<K,V>n**){**

**if(**nodeColor**(**n.parent**)**==Color.BLACK&&

nodeColor**(**n.sibling**())**==Color.BLACK&&

nodeColor**(**n.sibling**()**.left**)**==Color.BLACK&&

nodeColor**(**n.sibling**()**.right**)**==Color.BLACK**)**

**{**

n.sibling**()**.color=Color.RED;

deleteCase1**(**n.parent**)**;

**}**

**else**

deleteCase4**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 4](http://en.literateprograms.org/File:Red-black_tree_delete_case_4.png)  **Case 4:** N's sibling and sibling's children are black, but its parent is red. We exchange the colors of the sibling and parent; this restores the tree properties. |

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatevoid**deleteCase4**(**Node<K,V>n**){**

**if(**nodeColor**(**n.parent**)**==Color.RED&&

nodeColor**(**n.sibling**())**==Color.BLACK&&

nodeColor**(**n.sibling**()**.left**)**==Color.BLACK&&

nodeColor**(**n.sibling**()**.right**)**==Color.BLACK**)**

**{**

n.sibling**()**.color=Color.RED;

n.parent.color=Color.BLACK;

**}**

**else**

deleteCase5**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 5](http://en.literateprograms.org/File:Red-black_tree_delete_case_5.png)  **Case 5:** There are two cases handled here which are mirror images of one another:   * N's sibling S is black, S's left child is red, S's right child is black, and N is the left child of its parent. We exchange the colors of S and its left sibling and rotate right at S. * N's sibling S is black, S's right child is red, S's left child is black, and N is the right child of its parent. We exchange the colors of S and its right sibling and rotate left at S.   Both of these function to reduce us to the situation described in case 6. |

**<<**[**deletion operation**](http://en.literateprograms.org/Red-black_tree_(Java)#chunk use:deletion operation)**>>=**

**privatevoid**deleteCase5**(**Node<K,V>n**){**

**if(**n==n.parent.left&&

nodeColor**(**n.sibling**())**==Color.BLACK&&

nodeColor**(**n.sibling**()**.left**)**==Color.RED&&

nodeColor**(**n.sibling**()**.right**)**==Color.BLACK**)**

**{**

n.sibling**()**.color=Color.RED;

n.sibling**()**.left.color=Color.BLACK;

rotateRight**(**n.sibling**())**;

**}**

**elseif(**n==n.parent.right&&

nodeColor**(**n.sibling**())**==Color.BLACK&&

nodeColor**(**n.sibling**()**.right**)**==Color.RED&&

nodeColor**(**n.sibling**()**.left**)**==Color.BLACK**)**

**{**

n.sibling**()**.color=Color.RED;

n.sibling**()**.right.color=Color.BLACK;

rotateLeft**(**n.sibling**())**;

**}**

deleteCase6**(**n**)**;

**}**

|  |
| --- |
| [Diagram of case 6](http://en.literateprograms.org/File:Red-black_tree_delete_case_6.png)  **Case 6:** There are two cases handled here which are mirror images of one another:   * N's sibling S is black, S's right child is red, and N is the left child of its parent. We exchange the colors of N's parent and sibling, make S's right child black, then rotate left at N's parent. * N's sibling S is black, S's left child is red, and N is the right child of its parent. We exchange the colors of N's parent and sibling, make S's left child black, then rotate right at N's parent.   This accomplishes three things at once:   * We add a black node to all paths through N, either by adding a black S to those paths or by recoloring N's parent black. * We remove a black node from all paths through S's red child, either by removing P from those paths or by recoloring S. * We recolor S's red child black, adding a black node back to all paths through S's red child.   S's left child has become a child of N's parent during the rotation and so is unaffected. |

**<<****[deletion operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:deletion operation)>>=**

**privatevoid**deleteCase6**(**Node<K,V>n**){**

n.sibling**()**.color=nodeColor**(**n.parent**)**;

n.parent.color=Color.BLACK;

**if(**n==n.parent.left**){**

assertnodeColor**(**n.sibling**()**.right**)**==Color.RED;

n.sibling**()**.right.color=Color.BLACK;

rotateLeft**(**n.parent**)**;

**}**

**else**

**{**

assertnodeColor**(**n.sibling**()**.left**)**==Color.RED;

n.sibling**()**.left.color=Color.BLACK;

rotateRight**(**n.parent**)**;

**}**

**}**

Again, the function calls all use tail recursion, so the algorithm is in-place. Additionally, no recursive calls will be made after a rotation, so no more than three rotations are made.

### Printing

We also implement a method for printing the tree to standard output. This allows us to check that the tree looks as we expect, as well as providing a way to visualize the results of an operation. We print the right subtree before the left subtree so that the tree is displayed sideways. Only the keys are displayed.

**<<****[constants](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:constants)>>=**

**privatestaticfinalint**INDENT\_STEP=4;

**<<****[print operation](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:print operation)>>=**

**publicvoid**print**(){**

printHelper**(**root, 0**)**;

**}**

**privatestaticvoid**printHelper**(**Node<?,?>n, **int**indent**){**

**if(**n==**null){**

System.out.print**(**"<empty tree>"**)**;

**return**;

**}**

**if(**n.right!=**null){**

printHelper**(**n.right, indent+INDENT\_STEP**)**;

**}**

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

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

**if(**n.color==Color.BLACK**)**

System.out.println**(**n.key**)**;

**else**

System.out.println**(**"<"+n.key+">"**)**;

**if(**n.left!=**null){**

printHelper**(**n.left, indent+INDENT\_STEP**)**;

**}**

**}**

## Test driver

To ensure that all the cases of the complex insert and delete operations are exercised, we will perform a large number of operations on some simple integer data. All properties are verified after each operation, providing strong evidence of correctness.

**<<****[main method](http://en.literateprograms.org/Red-black_tree_(Java)" \l "chunk use:main method)>>=**

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

RBTree<Integer,Integer>t=**new**RBTree<Integer,Integer>**()**;

t.print**()**;

java.util.Randomgen=**new**java.util.Random**()**;

**for(int**i=0;i<5000;i++**){**

**int**x=gen.nextInt**(**10000**)**;

**int**y=gen.nextInt**(**10000**)**;

t.print**()**;

System.out.println**(**"Inserting "+x+" -> "+y**)**;

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

t.insert**(**x, y**)**;

assertt.lookup**(**x**)**.equals**(**y**)**;

**}**

**for(int**i=0;i<60000;i++**){**

**int**x=gen.nextInt**(**10000**)**;

t.print**()**;

System.out.println**(**"Deleting key "+x**)**;

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

t.delete**(**x**)**;

**}**

**}**

We print the tree after each operation.

**package** com.test.red.black.tree;

**enum** Color {

*RED*, *BLACK*

}

**class** Node<K **extends** Comparable<? **super** K>, V> {

**public** K key;

**public** V value;

**public** Node<K, V>left;

**public** Node<K, V>right;

**public** Node<K, V>parent;

**public** Color color;

**public** Node(K key, V value, Color nodeColor, Node<K, V> left,

Node<K, V> right) {

**this**.key = key;

**this**.value = value;

**this**.color = nodeColor;

**this**.left = left;

**this**.right = right;

**if** (left != **null**)

left.parent = **this**;

**if** (right != **null**)

right.parent = **this**;

**this**.parent = **null**;

}

**public** Node<K, V> grandparent() {

**assert**parent != **null**; // Not the root node

**assert**parent.parent != **null**; // Not child of root

**return**parent.parent;

}

**public** Node<K, V> sibling() {

**assert**parent != **null**; // Root node has no sibling

**if** (**this** == parent.left)

**return**parent.right;

**else**

**return**parent.left;

}

**public** Node<K, V> uncle() {

**assert**parent != **null**; // Root node has no uncle

**assert**parent.parent != **null**; // Children of root have no uncle

**return**parent.sibling();

}

}

**publicclass** RedBlackTree<K **extends** Comparable<? **super** K>, V> {

**publicstaticfinalboolean***VERIFY\_RBTREE* = **true**;

**privatestaticfinalint***INDENT\_STEP* = 4;

**public** Node<K, V>root;

**public** RedBlackTree() {

root = **null**;

verifyProperties();

}

**publicvoid** verifyProperties() {

**if** (*VERIFY\_RBTREE*) {

*verifyProperty1*(root);

*verifyProperty2*(root);

// Property 3 is implicit

*verifyProperty4*(root);

*verifyProperty5*(root);

}

}

/\*\*

\* 1. Each node is either red or black: Technically speaking, because the

\* way colors are represented by the Color enumeration, which only has the

\* values RED and BLACK, the only way this can fail is if the color

\* reference is null.

\*

\* **@param** n

\*/

**privatestaticvoid** verifyProperty1(Node<?, ?> n) {

**assert***nodeColor*(n) == Color.*RED* || *nodeColor*(n) == Color.*BLACK*;

**if** (n == **null**)

**return**;

*verifyProperty1*(n.left);

*verifyProperty1*(n.right);

}

/\*\*

\* 2. The root node is black.

\*

\* **@param** root

\*/

**privatestaticvoid** verifyProperty2(Node<?, ?> root) {

**assert***nodeColor*(root) == Color.*BLACK*;

}

/\*\*

\* 3. All leaves (shown as NIL in the above diagram) are black and contain

\* no data. Since we represent these empty leaves using null, this property

\* is implicitly assured by always treating null as black. To this end we

\* create a nodeColor() helper function:

\*

\* **@param** n

\* **@return**

\*/

**privatestatic** Color nodeColor(Node<?, ?> n) {

**return** n == **null** ? Color.*BLACK* : n.color;

}

/\*\*

\* 4. Every red node has two children, and both are black (or equivalently,

\* the parent of every red node is black).

\*

\* **@param** n

\*/

**privatestaticvoid** verifyProperty4(Node<?, ?> n) {

**if** (*nodeColor*(n) == Color.*RED*) {

**assert***nodeColor*(n.left) == Color.*BLACK*;

**assert***nodeColor*(n.right) == Color.*BLACK*;

**assert***nodeColor*(n.parent) == Color.*BLACK*;

}

**if** (n == **null**)

**return**;

*verifyProperty4*(n.left);

*verifyProperty4*(n.right);

}

/\*\*

\* 5. All paths from any given node to its leaf nodes contain the same

\* number of black nodes.

\* This one is the trickiest to verify; we do it by

\* traversing the tree, incrementing a black node count as we go. The first

\* time we reach a leaf we save the count. We return the count so that when

\* we subsequently reach other leaves, we compare the count to the saved

\* count.

\*

\* **@param** root

\*/

**privatestaticvoid** verifyProperty5(Node<?, ?> root) {

*verifyProperty5Helper*(root, 0, -1);

}

**privatestaticint** verifyProperty5Helper(Node<?, ?> n, **int** blackCount,

**int** pathBlackCount) {

**if** (*nodeColor*(n) == Color.*BLACK*) {

blackCount++;

}

**if** (n == **null**) {

**if** (pathBlackCount == -1) {

pathBlackCount = blackCount;

} **else** {

**assert** blackCount == pathBlackCount;

}

**return** pathBlackCount;

}

pathBlackCount = *verifyProperty5Helper*(n.left, blackCount,

pathBlackCount);

pathBlackCount = *verifyProperty5Helper*(n.right, blackCount,

pathBlackCount);

**return** pathBlackCount;

}

**private** Node<K, V> lookupNode(K key) {

Node<K, V> n = root;

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

**int** compResult = key.compareTo(n.key);

**if** (compResult == 0) {

**return** n;

} **elseif** (compResult < 0) {

n = n.left;

} **else** {

**assert** compResult > 0;

n = n.right;

}

}

**return** n;

}

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

Node<K, V> n = lookupNode(key);

**return** n == **null** ? **null** : n.value;

}

**privatevoid** rotateLeft(Node<K, V> n) {

Node<K, V> r = n.right;

replaceNode(n, r);

n.right = r.left;

**if** (r.left != **null**) {

r.left.parent = n;

}

r.left = n;

n.parent = r;

}

**privatevoid** rotateRight(Node<K, V> n) {

Node<K, V> l = n.left;

replaceNode(n, l);

n.left = l.right;

**if** (l.right != **null**) {

l.right.parent = n;

}

l.right = n;

n.parent = l;

}

**privatevoid** replaceNode(Node<K, V> oldn, Node<K, V> newn) {

**if** (oldn.parent == **null**) {

root = newn;

} **else** {

**if** (oldn == oldn.parent.left)

oldn.parent.left = newn;

**else**

oldn.parent.right = newn;

}

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

newn.parent = oldn.parent;

}

}

**publicvoid** insert(K key, V value) {

Node<K, V> insertedNode = **new** Node<K, V>(key, value, Color.*RED*, **null**,

**null**);

**if** (root == **null**) {

root = insertedNode;

} **else** {

Node<K, V> n = root;

**while** (**true**) {

**int** compResult = key.compareTo(n.key);

**if** (compResult == 0) {

n.value = value;

**return**;

} **elseif** (compResult < 0) {

**if** (n.left == **null**) {

n.left = insertedNode;

**break**;

} **else** {

n = n.left;

}

} **else** {

**assert** compResult > 0;

**if** (n.right == **null**) {

n.right = insertedNode;

**break**;

} **else** {

n = n.right;

}

}

}

insertedNode.parent = n;

}

insertCase1(insertedNode);

verifyProperties();

}

**privatevoid** insertCase1(Node<K, V> n) {

**if** (n.parent == **null**)

n.color = Color.*BLACK*;

**else**

insertCase2(n);

}

**privatevoid** insertCase2(Node<K, V> n) {

**if** (*nodeColor*(n.parent) == Color.*BLACK*)

**return**; // Tree is still valid

**else**

insertCase3(n);

}

**void** insertCase3(Node<K, V> n) {

**if** (*nodeColor*(n.uncle()) == Color.*RED*) {

n.parent.color = Color.*BLACK*;

n.uncle().color = Color.*BLACK*;

n.grandparent().color = Color.*RED*;

insertCase1(n.grandparent());

} **else** {

insertCase4(n);

}

}

**void** insertCase4(Node<K, V> n) {

**if** (n == n.parent.right&& n.parent == n.grandparent().left) {

rotateLeft(n.parent);

n = n.left;

} **elseif** (n == n.parent.left&& n.parent == n.grandparent().right) {

rotateRight(n.parent);

n = n.right;

}

insertCase5(n);

}

**void** insertCase5(Node<K, V> n) {

n.parent.color = Color.*BLACK*;

n.grandparent().color = Color.*RED*;

**if** (n == n.parent.left&& n.parent == n.grandparent().left) {

rotateRight(n.grandparent());

} **else** {

**assert** n == n.parent.right&& n.parent == n.grandparent().right;

rotateLeft(n.grandparent());

}

}

**publicvoid** delete(K key) {

Node<K, V> n = lookupNode(key);

**if** (n == **null**)

**return**; // Key not found, do nothing

**if** (n.left != **null**&& n.right != **null**) {

// Copy key/value from predecessor and then delete it instead

Node<K, V> pred = *maximumNode*(n.left);

n.key = pred.key;

n.value = pred.value;

n = pred;

}

**assert** n.left == **null** || n.right == **null**;

Node<K, V> child = (n.right == **null**) ? n.left : n.right;

**if** (*nodeColor*(n) == Color.*BLACK*) {

n.color = *nodeColor*(child);

deleteCase1(n);

}

replaceNode(n, child);

verifyProperties();

}

**privatestatic**<K **extends** Comparable<? **super** K>, V> Node<K, V> maximumNode(

Node<K, V> n) {

**assert** n != **null**;

**while** (n.right != **null**) {

n = n.right;

}

**return** n;

}

**privatevoid** deleteCase1(Node<K, V> n) {

**if** (n.parent == **null**)

**return**;

**else**

deleteCase2(n);

}

**privatevoid** deleteCase2(Node<K, V> n) {

**if** (*nodeColor*(n.sibling()) == Color.*RED*) {

n.parent.color = Color.*RED*;

n.sibling().color = Color.*BLACK*;

**if** (n == n.parent.left)

rotateLeft(n.parent);

**else**

rotateRight(n.parent);

}

deleteCase3(n);

}

**privatevoid** deleteCase3(Node<K, V> n) {

**if** (*nodeColor*(n.parent) == Color.*BLACK*

&&*nodeColor*(n.sibling()) == Color.*BLACK*

&&*nodeColor*(n.sibling().left) == Color.*BLACK*

&&*nodeColor*(n.sibling().right) == Color.*BLACK*) {

n.sibling().color = Color.*RED*;

deleteCase1(n.parent);

} **else**

deleteCase4(n);

}

**privatevoid** deleteCase4(Node<K, V> n) {

**if** (*nodeColor*(n.parent) == Color.*RED*

&&*nodeColor*(n.sibling()) == Color.*BLACK*

&&*nodeColor*(n.sibling().left) == Color.*BLACK*

&&*nodeColor*(n.sibling().right) == Color.*BLACK*) {

n.sibling().color = Color.*RED*;

n.parent.color = Color.*BLACK*;

} **else**

deleteCase5(n);

}

**privatevoid** deleteCase5(Node<K, V> n) {

**if** (n == n.parent.left&&*nodeColor*(n.sibling()) == Color.*BLACK*

&&*nodeColor*(n.sibling().left) == Color.*RED*

&&*nodeColor*(n.sibling().right) == Color.*BLACK*) {

n.sibling().color = Color.*RED*;

n.sibling().left.color = Color.*BLACK*;

rotateRight(n.sibling());

} **elseif** (n == n.parent.right&&*nodeColor*(n.sibling()) == Color.*BLACK*

&&*nodeColor*(n.sibling().right) == Color.*RED*

&&*nodeColor*(n.sibling().left) == Color.*BLACK*) {

n.sibling().color = Color.*RED*;

n.sibling().right.color = Color.*BLACK*;

rotateLeft(n.sibling());

}

deleteCase6(n);

}

**privatevoid** deleteCase6(Node<K, V> n) {

n.sibling().color = *nodeColor*(n.parent);

n.parent.color = Color.*BLACK*;

**if** (n == n.parent.left) {

**assert***nodeColor*(n.sibling().right) == Color.*RED*;

n.sibling().right.color = Color.*BLACK*;

rotateLeft(n.parent);

} **else** {

**assert***nodeColor*(n.sibling().left) == Color.*RED*;

n.sibling().left.color = Color.*BLACK*;

rotateRight(n.parent);

}

}

**publicvoid** print() {

*printHelper*(root, 0);

}

**privatestaticvoid** printHelper(Node<?, ?> n, **int** indent) {

**if** (n == **null**) {

System.*out*.print("<empty tree>");

**return**;

}

**if** (n.right != **null**) {

*printHelper*(n.right, indent + *INDENT\_STEP*);

}

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

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

**if** (n.color == Color.*BLACK*)

System.*out*.println(n.key);

**else**

System.*out*.println("<" + n.key + ">");

**if** (n.left != **null**) {

*printHelper*(n.left, indent + *INDENT\_STEP*);

}

}

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

RedBlackTree<Integer, Integer> t = **new** RedBlackTree<Integer, Integer>();

t.print();

java.util.Random gen = **new** java.util.Random();

**for** (**int** i = 0; i < 50; i++) {

**int** x = gen.nextInt(100);

**int** y = gen.nextInt(100);

t.print();

System.*out*.println("Inserting " + x + " -> " + y);

System.*out*.println();

t.insert(x, y);

**assert** t.lookup(x).equals(y);

}

**for** (**int** i = 0; i < 60000; i++) {

**int** x = gen.nextInt(10000);

t.print();

System.*out*.println("Deleting key " + x);

System.*out*.println();

t.delete(x);

}

}

}

# **Binary Heap**

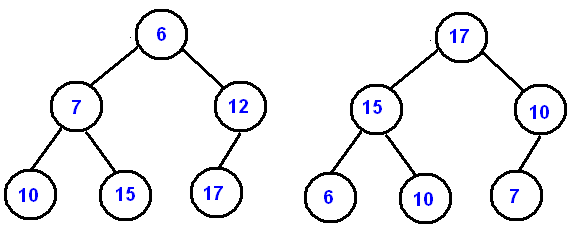
# Binary Heaps

## Introduction

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

* the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
* the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Throughout this chapter the word "heap" will always refer to a min-heap.



In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

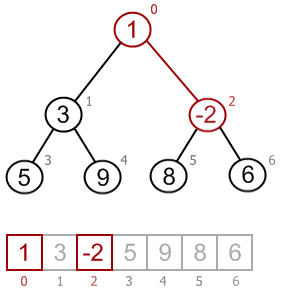
Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height.

A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

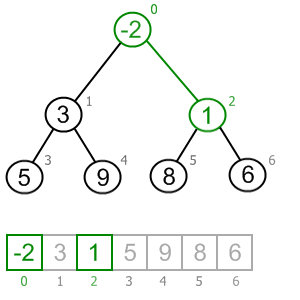
**Example**

|  |  |
| --- | --- |
| * 1. Insert -2 into a following heap:   http://www.algolist.net/img/binary-heap-insertion-source.png | 3) In the general case, after insertion, heap property near the new node is broken:  http://www.algolist.net/img/binary-heap-insertion-2.png |
| * 1. Insert a new element to the end of the array:http://www.algolist.net/img/binary-heap-insertion-1.png | 4) To restore heap property, algorithm *sifts up* the new element, by swapping it with its parent:  http://www.algolist.net/img/binary-heap-insertion-3.png |

Now heap property is broken at the root node:



Keep sifting:



Heap property is fulfilled, sifting is over.

|  |  |
| --- | --- |
| **Source heap** | **After -2 insertion** |
| http://www.algolist.net/img/binary-heap-insertion-source.png | http://www.algolist.net/img/binary-heap-insertion-final.png |

# Removing the minimum from a heap

Removal operation uses the same idea as was used for insertion. Root's value, which is minimal by the heap property, is replaced by the last array's value. Then new value is sifted down, until it takes right position.

## Removal algorithm

1. Copy the last value in the array to the root;
2. Decrease heap's size by 1;
3. Sift down root's value. Sifting is done as following:
   * if current node has no children, sifting is over;
   * if current node has one child: check, if heap property is broken, then swap current node's value and child value; sift down the child;
   * if current node has two children: find the smallest of them. If heap property is broken, then swap current node's value and selected child value; sift down the child.

### Example

Remove the minimum from a following heap:

|  |  |
| --- | --- |
| http://www.algolist.net/img/binary-heap-removal-source.png Copy the last value in the array to the root and decrease heap's size by 1 | http://www.algolist.net/img/binary-heap-removal-1.png  Now heap property is broken at root: |
| 1. http://www.algolist.net/img/binary-heap-removal-2.png   Root has two children. Swap root's value with the smallest: | 1. http://www.algolist.net/img/binary-heap-removal-3.png   Heap property is broken in node 1: |
| 1. http://www.algolist.net/img/binary-heap-removal-4.png   Recover heap property: | http://www.algolist.net/img/binary-heap-removal-5.png |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Node 3 has no children. Sifting is complete.

|  |  |
| --- | --- |
| **Source heap** | **After minimum removal** |
| http://www.algolist.net/img/binary-heap-removal-source.png | http://www.algolist.net/img/binary-heap-removal-final.png |

## Complexity analysis

Complexity of the removal operation is O(h) = O(log n), where **h** is heap's height, **n**is number of elements in a heap.

**Sorting**

**Insertion Sort**

|  |  |
| --- | --- |
| http://mathbits.com/MathBits/Java/arrays/cards.gif | The**insertion sort**, unlike the other sorts, **passes through the array only once.**  The insertion sort is commonly compared to organizing a handful of playing cards.  You pick up the random cards one at a time.  As you pick up each card, you insert it into its correct position in your hand of organized cards.    The insertion sort splits an array into two sub-arrays. The first sub-array (like the cards in your hand) is always sorted and increases in size as the sort continues. The second sub-array (like the cards to be picked up) is unsorted, contains all the elements yet to be inserted into the first sub-array, and decreases in size as the sort continues. |
| Let's look at our same example using the insertion sort for descending order. | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 |
| |  |  |  | | --- | --- | --- | | |  | | --- | |  | | = 1st sub-array | | 84 | 69 | 76 | 86 | 94 | 91 |
| |  |  |  | | --- | --- | --- | | |  | | --- | |  | | = 2nd sub-array | | 84 | 69 | 76 | 86 | 94 | 91 |
|  | 84 | 76 | 69 | 86 | 94 | 91 |
|  | 86 | 84 | 76 | 69 | 94 | 91 |
|  | 94 | 86 | 84 | 76 | 69 | 91 |
| 2nd sub-array empty | 94 | 91 | 86 | 84 | 76 | 69 |

The insertion sort maintains the two sub-arrays within the same array.  At the beginning of the sort, the first element of the first sub-array is considered the "sorted array".  With each pass through the loop, the next element in the unsorted second sub-array is placed into its proper position in the first sorted sub-array.

The insertion sort can be very fast and efficient when used with smaller arrays.  Unfortunately, it loses this efficiency when dealing with large amounts of data

**// Insertion Sort Method for Descending Order**

**package** sorting;

**publicclass** InsertionSort {

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

**int** arr[]={2,9,3,7,6,4,1};

*insertionsort*(arr);

*display*(arr);

}

**publicstaticvoid** display(**int** arr[]){

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

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

}

}

**publicstaticvoid** insertionsort(**int** arr[]){

**int** j;

**for**(**int** i=1;i<arr.length;i++){

**int** key=arr[i];

j=i-1;

/\*while((j>=0) && key>arr[j]){

arr[j+1]=arr[j];

j=j-1;

}\*/

**for**(j=i-1; j>=0 && arr[j]<key;j--){

arr[j+1]=arr[j];

}

arr[j+1]=key;

}

}

}

### Bubble Sort

The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

**package** sorting;

**publicclass** Bubblesort {

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

**int** arr[] = { **7, 5**, 2, 4, 3, 9 };

*bubbleSort*(arr);

*display*(arr);

}

**publicstaticvoid** display(**int** arr[]) {

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

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

}

}

**publicstaticvoid** bubbleSort(**int** ar[]) {

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

**for** (**int** j = 1; j < (ar.length - i); j++) {

**if** (ar[j - 1] > ar[j]) {

// swap the elements!

**int** temp = ar[j - 1];

ar[j - 1] = ar[j];

ar[j] = temp;

}

}

}

}

}

**Example.** Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

**7, 5**, 2, 4, 3, 9  
5, **7, 2**, 4, 3, 9  
5, 2, **7, 4**, 3, 9  
5, 2, 4, **7, 3**, 9  
5, 2, 4, 3, **7, 9**  
5, 2, 4, 3, 7, 9

The worst-case runtime complexity is O(n2). See explanation below 

### Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

**package** sorting;

/\*\*

\* The algorithm works by selecting the smallest unsorted item and then swapping

\* it with the item in the next position to be filled. The selection sort works

\* as follows: you look through the entire array for the smallest element, once

\* you find it you swap it (the smallest element) with the first element of the

\* array. Then you look for the smallest element in the remaining array (an

\* array without the first element) and swap it with the second element. Then

\* you look for the smallest element in the remaining array (an array without

\* first and second elements) and swap it with the third element, and so on

\*

\* **@author**rakeshku

\*

\*/

**publicclass** SelectionSort {

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

**int** arr[] = { 29, 64, 73, 34, 20 };

*selectionSort*(arr);

*display*(arr);

}

**publicstaticvoid** display(**int** arr[]) {

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

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

}

}

**publicstaticvoid** selectionSort(**int**[] ar) {

**for** (**int** i = 0; i < ar.length - 1; i++) {

**int** min = i;

**for** (**int** j = i + 1; j < ar.length; j++)

**if** (ar[j] < ar[min])

min = j;

**int** temp = ar[i];

ar[i] = ar[min];

ar[min] = temp;

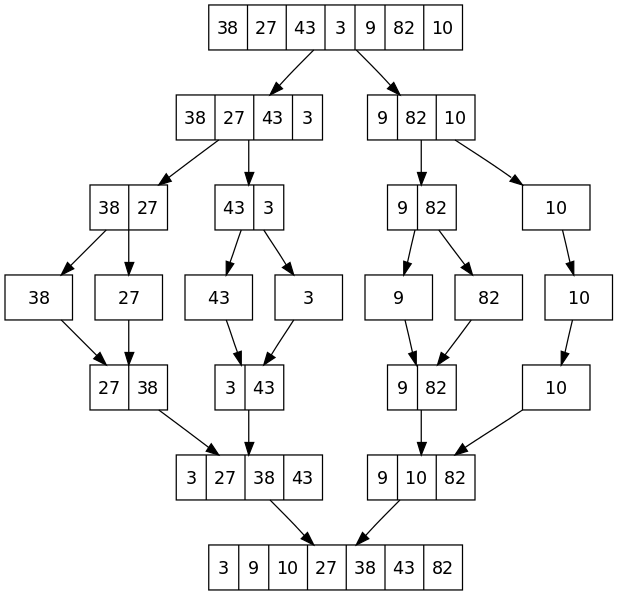
}

}

}

**Example.**

**29**, 64, 73, 34, **20**,   
20, **64**, 73, 34, **29**,   
20, 29, **73**, **34**, 64   
20, 29, 34, **73**, **64**   
20, 29, 34, 64, 73

**Merge sort algorithm**Merge sort algorithm

Conceptually, merge sort works as follows in recursive fashion:

1. Divide the unsorted list into two sublists of about half the size
2. Sort each of the two sublists
3. Merge the two sorted sublists back into one sorted list

**import** java.util.Arrays;

**publicclass** JavaMergeSort {

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

Integer arr[]={6,2,5,7,1,8};

**int** arr1[]={2,5,7,1,8};

*mergeSort*(arr);

System.*out*.println(Arrays.*toString*(arr));

*mergeSort*(arr1);

System.*out*.println(Arrays.*toString*(arr1));

}

**publicstaticvoid** mergeSort(Comparable<Integer>[] list){

**if** (list.length<= 1) {

**return** ;

}

**int** mid=list.length/2;

Comparable<Integer>[] firstHalf=**new** Comparable[mid];

Comparable<Integer>[] secondHalf=**new** Comparable[list.length-firstHalf.length];

System.*arraycopy*(list, 0, firstHalf,0, firstHalf.length);

System.*arraycopy*(list, firstHalf.length, secondHalf,0, secondHalf.length);

*mergeSort*(firstHalf);

*mergeSort*(secondHalf);

*merge*(firstHalf,secondHalf,list);

}

**privatestaticvoid** merge(Comparable[] firstHalf, Comparable[] secondHalf,

Comparable[] list) {

**int** iFirst=0;

**int** iSecond=0;

**int** iMerged=0;

**while**(iFirst<firstHalf.length&& iSecond<secondHalf.length){

**if**(firstHalf[iFirst].compareTo(secondHalf[iSecond])<0){

list[iMerged++]=firstHalf[iFirst++];

}**else** {

list[iMerged++]=secondHalf[iSecond++];

}

}

System.*arraycopy*(firstHalf, iFirst, list, iMerged, firstHalf.length-iFirst);

System.*arraycopy*(secondHalf, iSecond, list, iMerged, secondHalf.length-iSecond);

}

**publicstaticvoid** mergeSort(**int**[] list){

**if** (list.length<= 1) {

**return** ;

}

**int** mid=list.length/2;

**int**[] firstHalf=**newint**[mid];

**int**[] secondHalf=**newint**[list.length-firstHalf.length];

//Split the array in half in two parts

/\*System.arraycopy(list, 0, firstHalf,0, firstHalf.length);

System.arraycopy(list, firstHalf.length, secondHalf,0, secondHalf.length);\*/

*arrayCopy*(list,0,firstHalf,0,firstHalf.length);

*arrayCopy*(list, firstHalf.length, secondHalf, 0, secondHalf.length);

//Sort each half recursively

*mergeSort*(firstHalf);

*mergeSort*(secondHalf);

//Merge both halves together, overwriting to original array

*merge*(firstHalf,secondHalf,list);

}

**publicstaticvoid** arrayCopy(**int** [] source,**int** sIndex,**int** [] dest,**int** dIndex, **int** size){

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

dest[dIndex++]=source[sIndex++];

}

}

**privatestaticvoid** merge(**int**[] firstHalf, **int**[] secondHalf,

**int**[] list) {

//Index Position in first array - starting with first element

**int** iFirst=0;

//Index Position in second array - starting with first element

**int** iSecond=0;

//Index Position in merged array - starting with first position

**int** iMerged=0;

//Compare elements at iFirst and iSecond,

//and move smaller element at iMerged

**while**(iFirst<firstHalf.length&& iSecond<secondHalf.length){

**if**(firstHalf[iFirst]<secondHalf[iSecond]){

list[iMerged++]=firstHalf[iFirst++];

}**else** {

list[iMerged++]=secondHalf[iSecond++];

}

}

//copy remaining elements from both halves - each half will have already sorted elements

*arrayCopy*(firstHalf, iFirst, list, iMerged, firstHalf.length-iFirst);

*arrayCopy*(secondHalf, iSecond, list, iMerged, secondHalf.length-iSecond);

}

}

**package** com.ds.List;

/\*\*

\*

\* **@author**rakeshku

\*

\*/

**publicclass** MergeSortDemo {

**staticint***ar*[] = { 5, 2, 1, 0, 8, 7, 8 };

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

**int**tmp[]=**newint**[*ar*.length];

*mergeSort*(tmp, 0, *ar*.length-1);

*display*(*ar*);

*mergeSort*(*ar*);

*display*(tmp);

tmp = *mergeSorts*(*ar*);

*display*(tmp);

}

**publicstaticvoid** display(**int**tmp[]) {

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

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

}

}

**publicstaticvoid** mergeSort(**int**tmp[], **int**lowerBound, **int**upperBound) {

**if** (lowerBound == upperBound) {

**return**;

}

**int**middlePoint = (lowerBound + upperBound) / 2;

*mergeSort*(tmp, lowerBound, middlePoint);

*mergeSort*(tmp, middlePoint + 1, upperBound);

*merge*(tmp, lowerBound, middlePoint + 1, upperBound);

}

**publicstaticint**[] mergeSort(**int** [] list) {

**if** (list.length<= 1) {

**return**list;

}

// Split the array in half

**int**[] first = **newint**[list.length / 2];

**int**[] second = **newint**[list.length - first.length];

System.*arraycopy*(list, 0, first, 0, first.length);

System.*arraycopy*(list, first.length, second, 0, second.length);

// Sort each half

*mergeSort*(first);

*mergeSort*(second);

// Merge the halves together, overwriting the original array

*merge*(first, second, list);

**return**list;

}

**publicstaticint**[] mergeSorts(**int**arr[]) {

**if** (arr.length<= 1) {

**return**arr;

}

**int**lArray[] = **newint**[arr.length / 2];

**int**rArray[] = **newint**[arr.length - lArray.length];

System.*arraycopy*(arr, 0, lArray, 0, lArray.length);

System.*arraycopy*(arr, lArray.length, rArray, 0, rArray.length);

**return***merge*(*mergeSorts*(lArray), *mergeSorts*(rArray));

}

**publicstaticint**[] merge(**int**leftArr[], **int**rightArr[]) {

**int**li = 0;

**int**lLength = leftArr.length;

**int**ri = 0;

**int**rLength = rightArr.length;

**int**result[] = **newint**[lLength + rLength];

**int**i = 0;

**while** (li<lLength&&ri<rLength) {

**if** (leftArr[li] <rightArr[ri]) {

result[i++] = leftArr[li++];

} **else** {

result[i++] = rightArr[ri++];

}

}

/\*while (li< lLength) {

result[i++] = leftArr[li++];

}

while (ri< rLength) {

result[i++] = rightArr[ri++];

}

\*/

//either use above two loop or like below

System.*arraycopy*(leftArr, li, result, i, leftArr.length - li);

System.*arraycopy*(rightArr, ri, result, i, rightArr.length - ri);

**return**result;

}

**privatestaticvoid** merge(**int**[] first, **int**[] second, **int**[] result) {

// Merge both halves into the result array

// Next element to consider in the first array

**int**iFirst = 0;

// Next element to consider in the second array

**int**iSecond = 0;

// Next open position in the result

**int**j = 0;

// As long as neither iFirst nor iSecond is past the end, move the

// smaller element into the result.

**while** (iFirst<first.length&&iSecond<second.length) {

**if** (first[iFirst] <second[iSecond]) {

result[j] = first[iFirst];

iFirst++;

} **else** {

result[j] = second[iSecond];

iSecond++;

}

j++;

}

// copy what's left

System.*arraycopy*(first, iFirst, result, j, first.length - iFirst);

System.*arraycopy*(second, iSecond, result, j, second.length - iSecond);

}

**publicstaticvoid** merge(**int**[] tmp, **int**low, **int**high, **int**upperBound) {

**int**middlePoint = high - 1;

**int**lowerBound = low;

**int**n = upperBound - lowerBound + 1;

**int**i = 0;

**while** (low<= middlePoint&&high<= upperBound) {

**if** (*ar*[low] <*ar*[high]) {

tmp[i++] = *ar*[low++];

} **else** {

tmp[i++] = *ar*[high++];

}

}

**while** (low<= middlePoint) {

tmp[i++] = *ar*[low++];

}

**while** (high<= upperBound) {

tmp[i++] = *ar*[high++];

}

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

*ar*[lowerBound + i] = tmp[i];

}

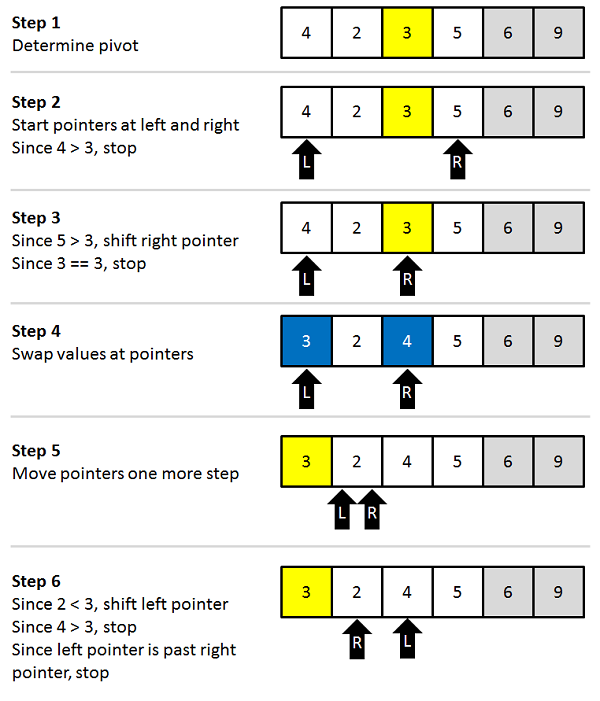
}

}

**Quick Sort**

## It's a divide and conquer algorithm, where we divide the given array with respect to a particular element, known as 'pivot' such that the lower partition of the array are less than the pivot and upper partition elements of the array are higher than the pivot. Quicksort is also one of the best example of recursion. It's naturally recursive, because it sort the large list by dividing into smaller sub-list and then applying same algorithm on those. Base case of recursion is when list contains either one or zero element, in that case they are sorted. Quicksort is well ahead with primitive sorting algorithms e.g. insertion sort, selection sort and bubble sort. Average time complexity of quicksort is O(nlogn), while in worst case it's performance is similar to bubble sort i.e.O(n^2). Apparently worst case of quicksort is the best case of insertion sort, where they have to sort an already sorted list. In this article, we will learn *how to implement quicksort algorithm in Java using recursion*. We will also learn how quicksort works, and how it sorts large list of unsorted number. In last section, we will see some important things about quicksort. How QuickSort Algorithm Perform Sorting

An old saying is, a picture is worth more than a thousand words. This is completely true in case of understanding *how sorting algorithm works*. In past, I have understood insertion sort, selection sort and quicksort much better by following a diagram rather then reading about it. That's why I am sharing this diagram which explains how quicksort algorithm works, how it sort a list of integers. It's similar to flowchart but doesn't use the notation flowchart uses, instead it practically shows how sorting happens. Once you go through this diagram, read the explanation, it will make more sense.



**package** sorting;

**import** java.util.Arrays;

/\*\*

\* Java program to Sort integer array using QuickSort algorithm using recursion.

\* Recursive QuickSort algorithm, partitioned list into two parts by a pivot,

\* and then recursively sorts each list

\*

\* **@author**rakeshku

\*

\*/

**publicclass** QuickSort {

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

**int**[] input = { 27, 31, 1, 21, 36, 72 };

System.*out*.println("Before sorting : " + Arrays.*toString*(input));

*quickSort*(input);

// sort the integer array using quick sort algorithm

System.*out*.println("After sorting : " + Arrays.*toString*(input));

// input with duplicates

**int**[] withDuplicates = { 16, 14, 16, 12, 11, 15 };

System.*out*.println("Before sorting : "

+ Arrays.*toString*(withDuplicates));

*quickSort*(withDuplicates);

// sort the integer array using quick sort algorithm

System.*out*

.println("After sorting : " + Arrays.*toString*(withDuplicates));

}

/\*\*

\* \* public method exposed to client, sorts given array using QuickSort \*

\* Algorithm in Java \* **@param** array

\*/

**publicstaticvoid** quickSort(**int**[] array) {

*recursiveQuickSort*(array, 0, array.length - 1);

}

/\*\*

\* \* Recursive quicksort logic \* \* **@param** array input array \* **@param**

\* startIdx start index of the array \* **@param** endIdx end index of the array

\*/

**publicstaticvoid** recursiveQuickSort(**int**[] array, **int** startIdx, **int** endIdx) {

**int** idx = *partition*(array, startIdx, endIdx);

// Recursively call quicksort with left part of the partitioned array

**if** (startIdx < idx - 1) {

*recursiveQuickSort*(array, startIdx, idx - 1);

}

// Recursively call quick sort with right part of the partitioned array

**if** (endIdx > idx) {

*recursiveQuickSort*(array, idx, endIdx);

}

}

/\*\*

\* \* Divides array from pivot, left side contains elements less than \* Pivot

\* while right side contains elements greater than pivot. \* \* **@param** array

\* array to partitioned \* **@param** left lower bound of the array \* **@param**

\* right upper bound of the array \* **@return** the partition index

\*/

**publicstaticint** partition(**int**[] array, **int** left, **int** right) {

**int** pivot = array[left];

// taking first element as pivot

**while** (left <= right) { // searching number which is greater than pivot,

// bottom up

**while** (array[left] < pivot) {

left++;

}

// searching number which is less than pivot, top down

**while** (array[right] > pivot) {

right--;

}

// swap the values

**if** (left <= right) {

**int** tmp = array[left];

array[left] = array[right];

array[right] = tmp;

// increment left index and decrement right index

left++;

right--;

}

}

**return** left;

}

}

**package** com.ds.List;

**publicclass** QuickSort **extends** BaseSort{

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

**int**arr[]={5,6,7,1,0,9};

*quicksort*(arr,0,arr.length-1);

*display*(arr);

}

**publicstaticvoid** quicksort(**int**list[], **int**left, **int**right) {

**if** (left>=right) {

**return**;

}

**int**p = *partition*(list, left, right);

*quicksort*(list,left,p);

*quicksort*(list, p+1, right);

}

**privatestaticint** partition(**int**[] list, **int**left, **int**right) {

**int**pivot = list[left];

**while** (left<right) {

**while** (list[left] <pivot) {

left++;

}

**while** ( right>0 &&list[right] >pivot) {

right--;

}

**if** (left<= right) {

**int**temp = list[left];

list[left] = list[right];

list[right] = temp;

}

}

**return**left;

}

}

**Heap Sort**

Heap is a useful data structure for designing efficient sorting algorithms and priority queues. A *heap* is a binary tree with the following properties:

**•** It is a complete binary tree.

• Each node is greater than or equal to any of its children

**Complete Binary Tree**

A binary tree is *complete* if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most. For example, in Figure 20.23, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).



Representing a Heap

For a node at position *i*, its left child is at position *2i+1* and its right child is at position *2i+2*, and its parent is (*i-1)/2*. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 (*2\*4+1*), its right child (element 33) is at 10 (*2\*4+2*), and its parent (element 42) is at 1 (*(4-1)/2*).



**Adding Elements to the Heap:**

Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty



**Rebuild the heap after adding a new node**



**Removing the Root and Rebuild the Tree**

Removing root 62 from the heap



Move 9 to root



Swap 9 with 59



Swap 9 with 44



Swap 9 with 30

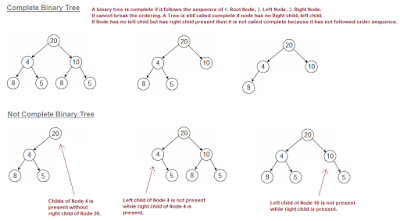
**Heap Sort**



**Before looking into Heap Sort, let's understand what is Heap and how it helps in sorting.**

What is Complete Binary Tree?

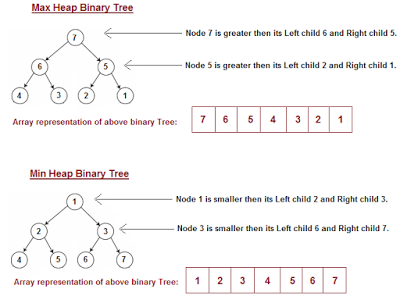
A Complete binary tree is a binary tree in which every node other than the leaves has two children. In complete binary tree at every level, except possibly the last, is completely filled, and all nodes are as far left as possible.  
  
Let's understand with simple words now,  
If a Binary Tree is filled level by level, left to right (Left child followed by Right child.) then it is called complete binary tree.   
If Right child is present without Left child then it is not complete.

[](http://3.bp.blogspot.com/-2JT-9xxZU8c/VkmFFcLNNAI/AAAAAAAAAqY/DLMfhTbRXJg/s1600/completeBinaryTree.png)

**What is Heap property in Binary Tree?**

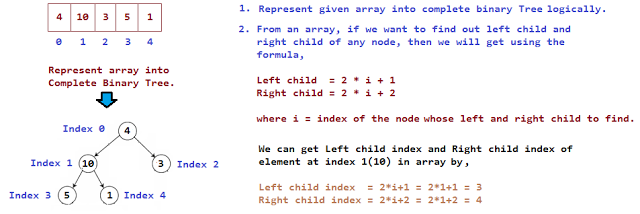
A binary Tree is said to follow a heap property if tree is complete binary tree and every element of the tree is Larger (or Smaller) than any of its descendants if they exists.  

**Depending on the ordering, a heap is called a max-heap or a min-heap.**  
**In a Max-heap,** the keys of parent nodes are always greater than or  equal to those of the children.   
In max-heap, Largest element of the Tree is always at top(Root Node).  
  
**In a Min-heap,**the keys of parent nodes are less than or equal to those of the children.  
In min-heap, Smallest element of the Tree is always at top(Root Node). 

[](http://2.bp.blogspot.com/-b4uSozAkmkQ/VkmLOCYek9I/AAAAAAAAAqs/0WlxmMdQP_U/s1600/Tree-Satisfying-Heap-property.png)

### Important aspects of Heap sort. (Prerequisites)

Before going into Heapsort algorithm, Let's understand few points,  
If we have an array say [4, 10, 3, 5, 1], then we can represent array as complete binary tree(start adding nodes from left to right) like shown below.

[](http://2.bp.blogspot.com/-Fi15Em4OZWc/VkmMMX_X4BI/AAAAAAAAAqw/K9134vVFWDA/s1600/heapsort.png)

Each element has left and right child present in array except for leaf nodes, but how to find left and right child of non-leaf nodes in array. 

**We will get left and right child of non leaf elements using formula,**  
**Left child index   = 2 \* (index of root, whose left and right child to find) + 1**  
**Right child index = 2 \* (index of root, whose left and right child to find) + 1**

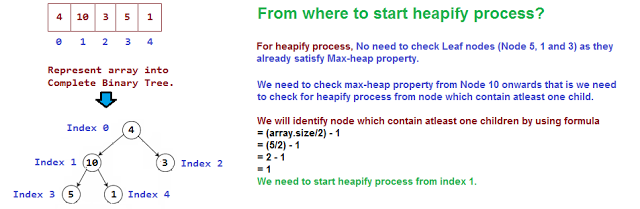
Left child and Right child of element at index 0 (element 4) is,  
Left child index  = 2 \* i + 1   = 2 \* 0 + 1   = 1    
Right child index = 2 \* i + 2   = 2 \* 0 + 2   = 2  
  
Left child and Right child of element at index 1 (element 10) is,  
Left child index  = 2 \* i + 1   = 2 \* 1 + 1   = 3    
Right child index = 2 \* i + 2   = 2 \* 1 + 2   = 4  
  
Left child and Right child of element at index 2 (element 3) is,  
Left child index  = 2 \* i + 1   = 2 \* 2 + 1   = 5   
(index 5 is greater than length of array, so element 3 has no left child)    
   
Right child index = 2 \* i + 2   = 2 \* 2 + 2   = 6   
(index 6 is greater than length of array, so element 3 has no right child)  
  

### Algorithm

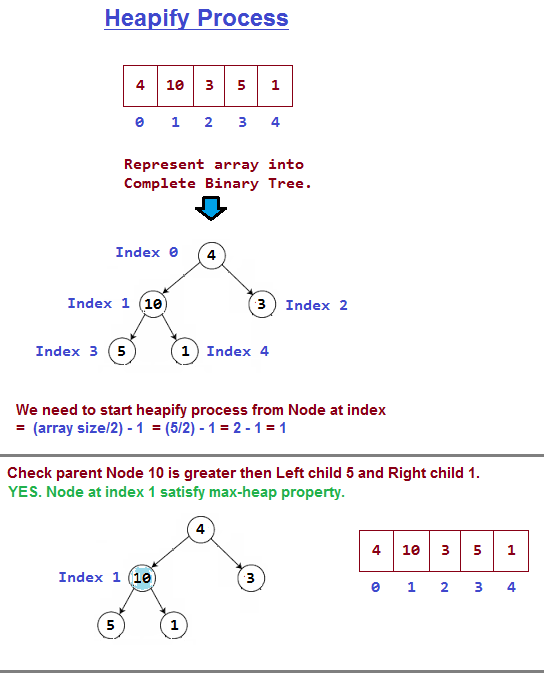
STEP 1:  Logically, think the given array as Complete Binary Tree,   
  
STEP 2:  For sorting the array in ascending order, check whether the tree is satisfying Max-heap   
               property at each node,   
               (For descending order, Check whether the tree is satisfying Min-heap property)   
               Here we will be sorting in Ascending order,  
  
STEP 3: If the tree is satisfying Max-heap property, then largest item is stored at the root of the heap.   
               (At this point we have found the largest element in array, Now if we place this element at  
               the end(nth position) of the array then 1 item in array is at proper place.)  
               We will remove the largest element from the heap and put at its proper place(nth position) in  
               array.  
     
              After removing the largest element, which element will take its place?     
              We will put last element of the heap at the vacant place. After placing the last element at the  
              root, The new tree formed may or may not satisfy max-heap property.  
              So, If it is not satisfying max-heap property then first task is to make changes to the tree, So   
              that it satisfies max-heap property.  
              
              (Heapify process: The process of making changes to tree so that it satisfies max-heap   
               property is called heapify)  
  
              When tree satisfies max-heap property, again largest item is stored at the root of the heap.      
              We will remove the largest element from the heap and put at its proper place(n-1 position) in  
              array.  
**Repeat step 3 until size of array is 1 (At this point all elements are sorted.)**

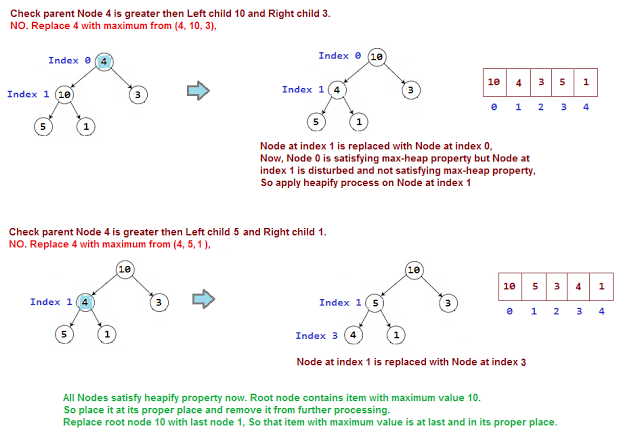
**Heapify Process with Example**

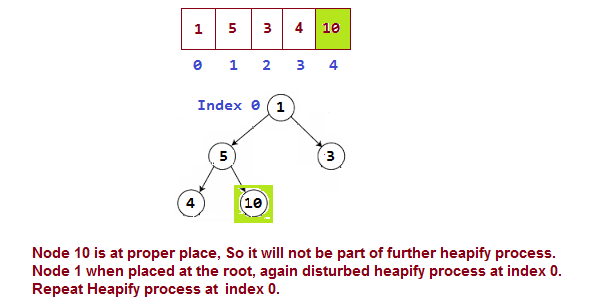
**Heapify process checks whether item at parent nodes has larger value than its left and right child.**  
If parent node is not largest compared to its left and right child, then it finds the largest item among parent, its left and right child and replaces largest with parent node.  
  
**It repeat the process for each node and at one point tree will start satisfying max-heap property.  
At this point, stop heapify process and largest element will be at root node.**We found the largest element, Remove it and put it at its proper place in array,   
Put the last element of the tree at the place we removed the node(that is at root of the tree)   
Placing last node at the root may disturbed the max-heap property of root node.   
**So again repeat the Heapify process for root node. Continue heapify process until all nodes in tree satisfy max-heap property.  
  
  
Initially, From which node we will start heapify process? Do we need to check each and every node that they satisfy heap property?  
We do not have to look into leaf nodes as they don't have children and already satisfying max-heap property.  
So, we will start looking from the node which has at least one child present.**  
**How we will get that item in array, which has at least one child present?  
By using the formula (array.length/2) - 1, we will be able to get the index of the item to start Heapify process.**

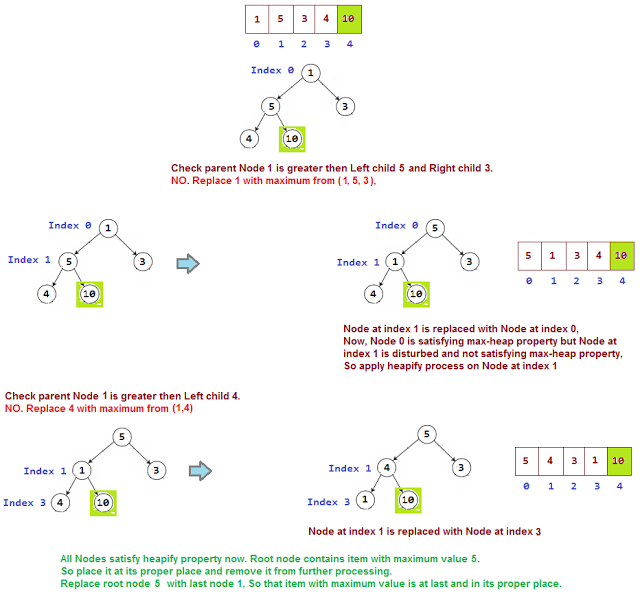
[](http://4.bp.blogspot.com/-KtBerk_cPjs/Vkmx-kmArYI/AAAAAAAAArA/11kzjNIbb8U/s1600/heapify-process-start-node.png)

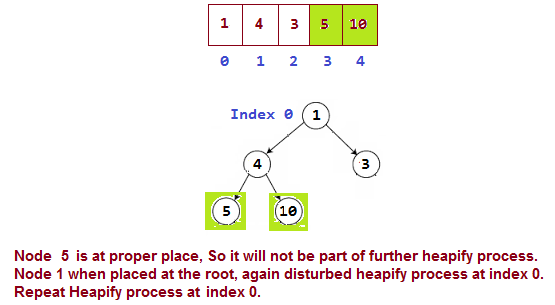
**Lets understand Heapify process with help of an example.**

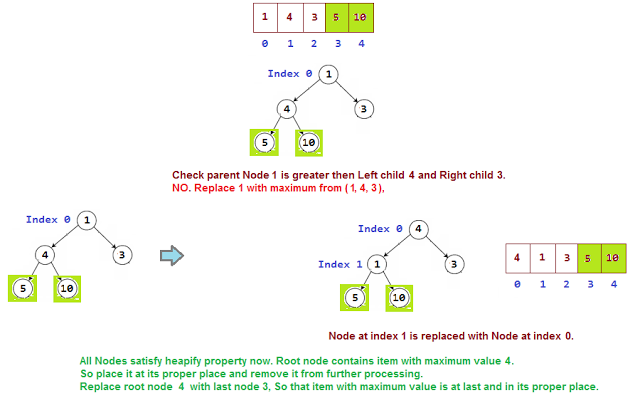
[](http://3.bp.blogspot.com/-e_CNlxHj5Bs/Vkoe738GAjI/AAAAAAAAArg/KmIbfAE3hKw/s1600/Heapify-process-Step1-2.png)

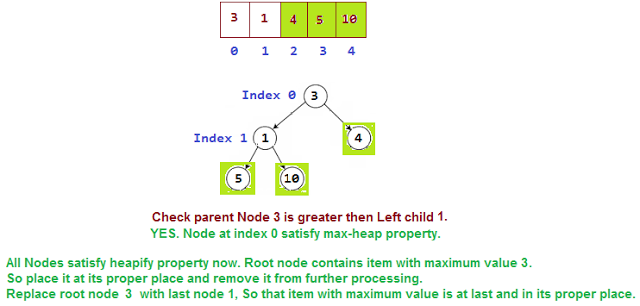
[](http://3.bp.blogspot.com/-qZtycYGmrbY/VkofGHecdII/AAAAAAAAAro/lGEdqlVd9uc/s1600/Heapify-process-Step3.png)

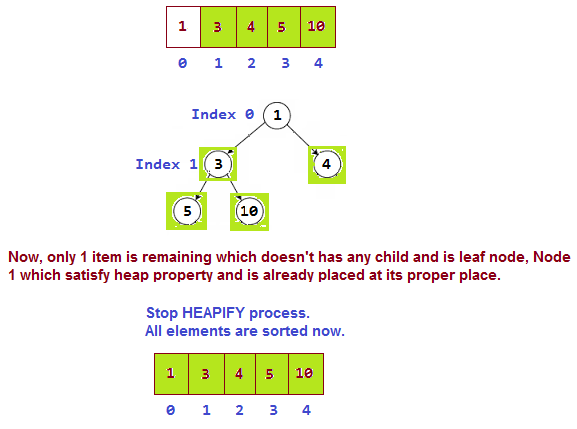
[[](http://3.bp.blogspot.com/-yHSl9-mCSrY/VkofN3NUnUI/AAAAAAAAArw/HXZvdD3qEAo/s1600/Heapify-process-Step4.png)](http://3.bp.blogspot.com/-yHSl9-mCSrY/VkofN3NUnUI/AAAAAAAAArw/HXZvdD3qEAo/s1600/Heapify-process-Step4.png)

[](http://1.bp.blogspot.com/-br1LdT46SBI/VkofV7gQnxI/AAAAAAAAAr4/jqWbKvGw1z4/s1600/Heapify-process-Step5.png)

[](http://3.bp.blogspot.com/-Tfhv3D4hNcc/VkofljCSrfI/AAAAAAAAAsA/rogcIBeV-J0/s1600/Heapify-process-Step6.png)

[](http://1.bp.blogspot.com/-A9sjW-cSW_E/VkofqTdiGrI/AAAAAAAAAsI/ROkePgWvf9s/s1600/Heapify-process-Step7.png)

[](http://2.bp.blogspot.com/-ivFdfHu3RFU/Vkofv9RSPzI/AAAAAAAAAsQ/ZKC567H4cDI/s1600/Heapify-process-Step8.png)

[](http://2.bp.blogspot.com/-qM3EncOJIvE/Vkof0yip2sI/AAAAAAAAAsY/fB-WSLOX63s/s1600/Heapify-process-Step9.png)

**package** sorting;

**publicclass** HeapSort {

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

**int**[] array = **newint**[] { 4, 10, 3, 5, 1 };

**new** HeapSort().sort(array);

**for** (**int** i : array) {

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

}

}

**publicvoid** sort(**int** data[]) {

**int** size = data.length;

**for** (**int** i = size / 2 - 1; i >= 0; i--) {

heapify(i, data, size);

}

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

**int** temp = data[0];

data[0] = data[i];

data[i] = temp;

// reduce the heap window by 1

size = size - 1;

// call max heapify on the reduced heap

heapify(0, data, size);

}

}

**privateint** leftChild(**int** i) {

**return** 2 \* i + 1;

}

**privateint** rightChild(**int** i) {

**return** 2 \* i + 2;

}

**privatevoid** heapify(**int** i, **int**[] data, **int** size) {

**int** largestElementIndex = i;

**int** leftChildIndex = leftChild(i);

**if** (leftChildIndex < size

&& data[leftChildIndex] > data[largestElementIndex]) {

largestElementIndex = leftChildIndex;

}

**int** rightChildIndex = rightChild(i);

**if** (rightChildIndex < size

&& data[rightChildIndex] > data[largestElementIndex]) {

largestElementIndex = rightChildIndex;

}

**if** (largestElementIndex != i) {

**int** swap = data[i];

data[i] = data[largestElementIndex];

data[largestElementIndex] = swap;

// Recursively heapify the affected sub-tree

heapify(largestElementIndex, data, size);

}

}

}

**Summarize Heap Sort algorithm.**

* 1. We build a heap(Max or Min) from the given array elements.
  2. The root is the max (or min number). So extract it and put it in an array at its proper position.
  3. Put last element at the root of the tree and Heapify the remaining elements.
  4. Again extract the root and repeat heapification until there is one element in array.

**Advantage of using Heap Sort algorithm for Sorting**

* 1. Heap sort has the best possible worst case running time complexity of O(n Log n).  
     2. It doesn't need any extra storage and that makes it good for situations where array size is large.

**Recursion**

**Write a program to find *permutation/ combination/ anagram of String* using recursion**.

*Example-*

*Permutations of inputString(ABC) are: [ACB, ABC, BCA, CBA, CAB, BAC]*

*Example-*

*Permutations of inputString(ABCD) are: [DABC, CADB, BCAD, DBAC, BACD, ABCD, ABDC, DCBA, ADCB, ADBC, CBDA, CBAD, DACB, ACBD, CDBA, CDAB, DCAB, ACDB, DBCA, BDAC, CABD, BADC, BCDA, BDCA]*

*In this post you will find two programs to find permutation/ combination/ anagram of String.*

**Program 1 - to find permutation/combination/anagram of String >**

**import** java.util.HashSet;

**import** java.util.Set;

/\*\* \*/

**publicclass**PemutationOfStringRecursion {

**publicstaticvoid** main(String... args) {

String inputString = "XYZ";

System.*out*.println("Permutations of inputString(" + inputString

+ ") are: " + *findPermutation*(inputString));

}

/\*\*

\* method returns permutations of string.

\*/

**publicstatic** Set<String> findPermutation(String inputString) {

Set<String> set = **new** HashSet<String>();

Set<String> set2;

String stringWithoutFirstChar;

**char** firstChar;

**if** (inputString.length() == 0) {

set.add("");

**return** set;

}

firstChar = inputString.charAt(0);

stringWithoutFirstChar = inputString.substring(1);

set2 = *findPermutation*(stringWithoutFirstChar);

**for** (String s : set2) {

**for** (**int** k = 0; k <= s.length(); k++) {

set.add(*insertCharacter*(s, firstChar, k));

}

}

**return** set;

}

**publicstatic** String insertCharacter(String s, **char** ch, **int** i) {

String begin = s.substring(0, i);

String end = s.substring(i);

**return** begin + ch + end;

}

}

/\*OUTPUT

Permutations of inputString(XYZ) are: [XYZ, XZY, YZX, ZYX, ZXY, YXZ]

\*/

**Program 2 - to find permutation/combination/anagram of String >**

**publicclass**AnagramOrPermutation {

**staticchar**[] *ar* = **newchar**[100];

**staticint***length*;

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

String inputString = "abc";

*length* = inputString.length();

*ar* = inputString.toCharArray();

System.*out*.print("Anagram of inputString (" + inputString + ") are: ");

*anagram*(*length*);

}

/\*

\* Anagram method.

\*/

**publicstaticvoid** anagram(**int** newLength) {

**if** (newLength == 1) // return if newLength is 1.

**return**;

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

*anagram*(newLength - 1); // recursively find the remaining anagrams.

**if** (newLength == 2)

*displayWord*();

*rotate*(newLength);

}

}

/\*

\* Rotate the string.

\*/

**publicstaticvoid** rotate(**int** newLength) {

**char** temp = *ar*[*length* - newLength];

**int** i = 0;

**for** (i = (*length* - newLength) + 1; i <*length*; i++)

*ar*[i - 1] = *ar*[i]; // shift elements left.

*ar*[i - 1] = temp;

}

/\*

\* Method displays word.

\*/

**publicstaticvoid** displayWord() {

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

System.*out*.print(*ar*[i]);

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

}

}

/\* output

Anagram of inputString (abc) are: abc acb bca bac cab cba

\*/

### Fibonacci series using recursion in java

Write a program to generate Fibonacci series using [recursion](http://javamadesoeasy.com/search/label/Recursion).

**0 1 1 2 3 5 8 13 21 34 55 89.**

First number of series is 0 & second number of series is 1.

So, logic behind the series generation is that the subsequent number generated is sum of previous two number of series.

/\*\*\*/

**publicclass**FibonacciRecursion {

**publicstaticvoid** main(String...args) {

**int** n = 10; //generate series upto n.

System.*out*.print("FibonacciSeries : 0 1 ");

**for**(**int** i=2; i<=n; i++){

System.*out*.print(*fibonacciRecursion*(i) +" ");

}

}

/\*

\* return next number of FibonacciSeries using recursion.

\*/

**publicstaticint** fibonacciRecursion(**int** n){

**if**(n == 1 || n == 2){

**return** 1;

}

**return***fibonacciRecursion*(n-1) + *fibonacciRecursion*(n-2); //using tail recursion

}

}

/\*OUTPUT

FibonacciSeries : 0 1 1 2 3 5 8 13 21 34 55

\*/

[**Reverse String using recursion in java**](http://www.javamadesoeasy.com/2015/02/reverse-string-using-recursion.html)

Hi! In this post we will reverse string using [recursion](http://javamadesoeasy.com/search/label/Recursion).

Original String: **abcde**

Reversed String: **edcba**

/\*\*\*/

**publicclass**ReverseStringRecursion {

**publicstaticvoid** main(String...args){

String originalString="abcde"; //String to be reversed

System.*out*.println("Original String: "+originalString);

System.*out*.print("Reversed String: ");

*reverseRecursively*(originalString);

}

/\*

\* reverse string recursively.

\*/

**publicstaticvoid** reverseRecursively(String str) {

**if** (str.length() == 1){

System.*out*.print(str);

}

**else** {

*reverseRecursively*(str.substring(1, str.length()));

System.*out*.print(str.substring(0, 1));

}

}

}

|  |
| --- |
| /\*OUTPUT  Original String: abcde  Reversed String: edcba  \*/ |

[**Palindrome using recursion in java**](http://www.javamadesoeasy.com/2015/02/palindrome-using-recursion.html)

**Q.**What is palindrome?

**A.** If reverse of string is same as original one, than our string is palindrome.

OR

A **palindrome** is a string which reads the same backward or forward.

Example>

input String : **aabaa**

output: **it’s palindrome.**

/\*\*\*/

**publicclass**PalindromeRecursion {

**publicstaticvoid** main(String...args){

String inputString="aabaa";

System.*out*.println(*isPalindromeUsingRecursion*(inputString) ? inputString+ " is a palindrome." : inputString+ "is not a palindrome.");

}

/\*\*

\* This methods finds out whether inputString is palindrome or not recursively.

\* Returns true if inputString is palindrome.

\*/

**publicstaticboolean** isPalindromeUsingRecursion(String inputString){

**if**(inputString.length()==0 || inputString.length()==1){

**returntrue**;

}

**if**(inputString.charAt(0)==inputString.charAt(inputString.length()-1)){

**return***isPalindromeUsingRecursion*(inputString.substring(1,inputString.length()-1));

}

**returnfalse**;

}

}

/\*OUTPUT

aabaa is a palindrome.

\*/

**Q.**What is palindrome?

**A.** If reverse of number is same as original one, than its palindrome or **palindrome** is a number which reads the same backward or forward.

Example>

input : **121**

output: **it’s palindrome.**

***Logic*** *used here will offer you best time and space complexity.*

***time complexity -***

*taken for each number will be O(n/2).*

*so, overall complexity of program is* ***1000 x O(n/2)****, where n will change from 1 to 1000.*

***space complexity -***

*because we are using only one variable i.e. j for palindrome calculation of each number. (i is not counted because it is just used for maintaining 1 to 202 count)*

**that prints all the palindrome till 202 in java >**

/\*\* Copyright (c), Test.com \*/

**publicclass**PalindromeTill {

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

System.*out*.println("List of Palindrome between 1 to 1000");

**for**(**int** i=10; i<=202; i++){

**char**[] ar = String.*valueOf*(i).toCharArray();

**boolean** isPalindrome = **true**;

**for**(**int** j=0;j<ar.length/2;j++){

**if**(ar[j]!=ar[ar.length-1-j]){

isPalindrome =**false**;

**break**;

}

}

**if**(isPalindrome)

System.*out*.println(i);

}

}

}

|  |
| --- |
| /\*  List of Palindrome between 1 to 500  11  22  33  44  55  66  77  88  99  101  111  121  131  141  151  161  171  181  191  202  \*/ |

*Question 1. Where are objects created in memory? On stack or heap?*

*Answer.  Let’s start garbage collection interview question with very basic question. All Java objects are always created on heap in java.*

*Question 2. What is Garbage Collection process in java?*

*Answer.  Basic garbage collection interview question. Definitely all developers must know it :)*

*GC (Garbage collection) is the process by which JVM cleans objects (unused objects) from heap to reclaim heap space in java.*

*Question 3. What is Automatic Garbage Collection in JVM heap memory in java?*

*Answer.  Very important thing you must know in garbage collection interview.*

*Automatic garbage collection is the process of*

*Identifying objects which are in use in java heap memory and*

*Which objects are not in use in java heap memory and*

*deleting the unused objects in java heap memory.*

*Question 4. How to Identify objects which are in use in JVM heap memory in java?*

*Answer. It is very basic garbage collection interview question.*

*Objects in use (or referenced objects) are those objects which are still needed by java program, some part of java program is still pointing to that object.*

*Question 5. Which objects are not in use in JVM heap memory in java?*

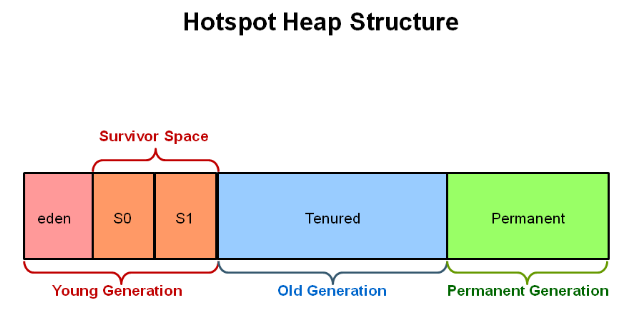
*Answer. Another very basic garbage collection interview question.*

*Objects not in use (or unreferenced objects) are those objects which are not needed by java program, no part of java program is pointing to that object.*

*So, these unused objects can be cleaned in GC (garbage collection) process and memory used by an unreferenced object can be reclaimed.*

*Question 6. Explain JVM Heap memory (Hotspot heap structure) with diagram in java?*

*Answer.  A very very important garbage collection interview question for freshers, experienced software professionals. Architects must know it on finger tips.*

**

*JVM Heap memory (Hotspot heap structure)  in java consists of following elements>*

*Young Generation*

*1a) Eden,*

*1b) S0 (Survivor space 0)*

*1c) S1 (Survivor space 1)*

*Old Generation (Tenured)*

*Permanent Generation.*

*So, JVM Heap memory (Hotspot heap structure) is divided into three parts Young Generation, Old Generation (tenured) and Permanent Generation.*

*Young Generation is further divided into Eden, S0 (Survivor space 0) and S1 (Survivor space 1).*

*Read in detail about* [*JVM Heap memory (Hotspot heap structure) with diagram in java*](http://www.javamadesoeasy.com/2016/10/jvm-heap-memory-hotspot-heap-structure.html)

*Question 7. What is Throughput in gc(garbage collection) in java ?*

*Answer.  Basic terms you should know about garbage collection interview.*

*In short, Throughput is the time not spent in garbage collection (GC) ( in percent).*

*Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include >*

*The number of transactions completed in a given time.*

*The number of jobs that a batch program can complete in an hour.*

*The number of database queries that can be completed in an hour.*

*Question 8. What are pauses in gc(garbage collection) in java ?*

*Answer.  Another basic term which is used very often in garbage collection. Pauses is applications pauses i.e. when application doesn’t gives any response because of garbage collection (GC).*

*Question 9. What is Young Generation in JVM Heap memory in java?*

*Answer.  This is important garbage collection interview question.*

*New objects are allocated in Young generation.*

*Young Generation consists of >*

*1a) Eden,*

*1b) S0 (Survivor space 0)*

*1c) S1 (Survivor space 1)*

*Young Generation is further divided into three parts Eden, S0 (Survivor space 0) and S1 (Survivor space 1).*

*Read in more detail about >* [*What is Young Generation in JVM in java*](http://www.javamadesoeasy.com/2016/10/what-are-young-old-tenured-and.html)

*Question 10. What is Old Generation in JVM Heap memory in java?*

*Answer.  This is another important garbage collection interview question.*

*The Old Generation(tenured generation) is used for storing the long surviving aged objects (Some of the objects which aren't cleaned up survive in young generation and gets aged.  Eventually such objects are moved from young to old generation).*

*Major garbage collection occurs in Old Generation.*

*At what time (or what age) objects are moved from young to old generation in JVM heap?*

*There is some threshold set for young generation object and when that age is met, the object gets moved to the old generation.*

*What is major garbage collection in java? (We will read about it in detail in upcoming interview questions)*

*When the old generation fills up, this causes a major garbage collection. Objects are cleaned up from old generation.*

*Read :* [*JVM (java virtual machine) in detail in java*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *and* [*How Garbage Collection (GC) works internally*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html)

*Question 11. What is Permanent Generation in JVM Heap memory in java?*

*Answer.  This is third generation in java heap and hence is very important garbage collection interview question. It is complex question and hence experienced software professionals must know it.*

*Permanent generation (Permgen) Space contains metadata required by JVM to describe the classes and methods used in the application.*

*The permanent generation is included in a full garbage collection in java.*

*The permanent generation space is populated at runtime by JVM based on classes in use in the application.*

*The permanent generation space also contains Java SE library classes and methods in java.*

*JVM garbage collects those classes when classes are no longer required and space may be needed for other classes in java.*

*Read in more detail about >* [*What is Permanent Generation in JVM in java*](http://www.javamadesoeasy.com/2016/10/what-are-young-old-tenured-and.html)

*Question 12. What is Minor garbage collection in JVM Heap memory in java?*

*Answer.  It's important to know about minor garbage collection before interview.*

*Minor garbage collection occurs in Young Generation*

*New objects are allocated in Young generation.*

*When the young generation fills up, this causes a minor garbage collection.*

*All the unreferenced (dead) objects are cleaned up from young generation.*

*When objects are moved from young to old generation in JVM heap?*

*Some of the objects which aren't cleaned up survive in young generation and gets aged.  Eventually such objects are moved from young to old generation.*

*What is Stop the World Event?*

*Minor garbage collections are called Stop the World events.*

*All the non-daemon threads running in application are stopped during minor garbage collections (i.e. the application stops for while).*

[*Daemon threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *performs minor garbage collection. (Daemon threads are low* [*priority threads*](http://www.javamadesoeasy.com/2015/03/thread-priorities-setpriority-and.html) *which runs intermittently in background for doing garbage collection).*

*Read in more detail about >*[*What is Minor garbage collection in JVM*](http://www.javamadesoeasy.com/2016/10/what-are-minor-major-and-full-garbage_29.html)

*Question 13. What is Major garbage collection in JVM Heap memory in java?*

*Answer.  It’s easy garbage collection interview question. Prepare it well.*

*Major garbage collection occurs in Old Generation*

*The Old Generation is used for storing the long surviving aged objects.*

*At what time (or what age) objects are moved from young to old generation in JVM heap?*

*There is some threshold set for young generation object and when that age is met, the object gets moved to the old generation during garbage collection in java.*

*When the old generation fills up, this causes a major garbage collection. Objects are cleaned up from old generation.*

*Major collection is much slower than minor garbage collection in jvm heap because it involves all live objects.*

*Major garbage collection are Stop the World Event in java?*

*Major garbage collections are also called Stop the World events.*

*All the non-daemon threads running in application are stopped during major garbage collections.*

[*Daemon threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *performs major garbage collection.*

*Major garbage collections in responsive applications in java?*

*Major garbage collections should be minimized for responsive applications because applications must not be stopped for long.*

*Optimizing Major garbage collections in responsive applications in java?*

*Selection of appropriate garbage collector for the old generation space affects the length of the “Stop the World” event for a major garbage collection.*

*Read in more detail about >* [*What is Major garbage collection in JVM*](http://www.javamadesoeasy.com/2016/10/what-are-minor-major-and-full-garbage_29.html)

*Question 14. What is Full garbage collection in JVM Heap memory in java?*

*Answer.  You must be able to differentiate between garbage collection and full garbage collection. Don’t mess up between two terms.*

*Full garbage collection occurs in permanent generation in java*

*Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.*

*The permanent generation space is populated at runtime by JVM based on classes in use in the application.*

*JVM garbage collects those classes when classes are no longer required and space may be needed for other classes in java.*

*Read in more detail about >* [*What is Full garbage collection in JVM*](http://www.javamadesoeasy.com/2016/10/what-are-minor-major-and-full-garbage_29.html)

*Question 15. Mention some of the most important VM (JVM) PARAMETERS you have used for Young Generation in JVM Heap memory?*

*Answer.  It is important and challenging garbage collection interview question for experienced developers to specifically identify and answer jvm parameters for young generation.*

*-Xmn : -Xmn sets the size of young generation.*

*Example of using -Xmn VM (JVM) option in java >*

*java -Xmn512m MyJavaProgram*

*For more explanation and example - Read :*[*-Xmn JVM parameters*](http://www.javamadesoeasy.com/2016/10/xmn-jvm-parameters-in-java-setting.html)

[*-XX:NewRatio*](http://www.javamadesoeasy.com/2016/10/what-is-xxnewratio-jvm-parameters-in.html) *: NewRatio controls the size of young generation.*

*Example of using -XX:NewRatio  VM option in java >*

*-XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.*

[*-XX:NewSize*](http://www.javamadesoeasy.com/2016/10/what-are-xxnewsize-and-xxmaxnewsize-jvm.html) *- NewSize is minimum size of young generation which is allocated at initialization of JVM.*

*-XX:MaxNewSize - MaxNewSize is the maximum size of young generation that JVM can use.*

[*-XX:SurvivorRatio*](http://www.javamadesoeasy.com/2016/10/xxsurvivorratio-jvm-parameters-in-java.html) *:   (for survivor space)*

*SurvivorRatio can be used to tune the size of the survivor spaces, but this is often not as important for performance.*

*Example of using -XX:SurvivorRatio > -XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6.*

*Question 16. Mention some of the most important VM (JVM) PARAMETERS you have used for Old Generation (tenured) in JVM Heap memory?*

*Answer.   Another is difficult and challenging garbage collection interview question for experienced developers to specifically identify and answer jvm parameters for old generation.*

*-XX:NewRatio : NewRatio controls the size of young and old generation.*

*Example of using -XX:NewRatio, -XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.*

*For more explanation and example* [*-XX:NewRatio JVM parameters*](http://www.javamadesoeasy.com/2016/10/what-is-xxnewratio-jvm-parameters-in.html)

*Question 17. Mention some of the most important VM (JVM) PARAMETERS you have used for Permanent Generation?*

*Answer.  Another difficult garbage collection interview question for experienced developers to specifically identify and answer jvm parameters for permanent generation.*

*-XX:PermSize: It’s is initial value of Permanent Space which is allocated at startup of JVM.*

*Example of using -XX:PermSize VM (JVM) option in java >*

*java -XX:PermSize=512m MyJavaProgram*

*It will set initial value of Permanent Space as 512 megabytes to JVM*

*-XX:MaxPermSize: It’s maximum value of Permanent Space that JVM can allot up to.*

*Example of using -XX:MaxPermSize VM (JVM) option in java >*

*java -XX:MaxPermSize=512m MyJavaProgram*

*It will set maximum value of Permanent Space as 512 megabytes to JVM*

*For more details -* [*What are -XX:PermSize and -XX:MaxPermSize with Differences*](http://www.javamadesoeasy.com/2016/10/what-are-xxpermsize-and-xxmaxpermsize.html)

*Question 18. What are different Garbage collectors available in JVM?*

*Answer.  Very very important garbage collection interview question. Experienced developers must know about all of them.*

*Different type of garbage collectors in java>*

[*Serial collector / Serial GC (Garbage collector) in java*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html)

[*Throughput GC (Garbage collector) or Parallel collector in java*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html)

[*Concurrent Mark Sweep (CMS) collector / concurrent low pause garbage collector*](http://www.javamadesoeasy.com/2016/10/concurrent-mark-sweep-cms-collector.html)

[*G1 garbage collector / Garbage first collector*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html)

[*PS Scavenge*](http://www.javamadesoeasy.com/2016/11/ps-scavenge-and-ps-marksweep.html)

[*PS MarkSweep*](http://www.javamadesoeasy.com/2016/11/ps-scavenge-and-ps-marksweep.html)

[*ParNew collector*](http://www.javamadesoeasy.com/2016/11/parnew-collector.html)

*Question 19 . What is Serial collector / Serial GC (Garbage collector) in java?*

*Answer.  Freshers must know about this garbage collection interview question.*

*Features of* [*Serial GC (Garbage collector)*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html) *in java  >*

*Serial collector is also called Serial GC (Garbage collector) in java.*

*Serial collector is simply also called Serial collector in java.*

*Serial GC (Garbage collector) is rarely used in java.*

*Serial GC is designed for the single threaded environments in java.*

*In Serial GC (Garbage collector) , both minor and major garbage collections are done serially by one thread in java.*

*Serial GC uses a mark-compact collection method. The serial garbage collector is the default for client style machines in Java SE 5 and 6.*

*When to Use the Serial GC (garbage Collector) in java >*

*The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines.*

*Serial garbage collector is also popular in environments where a high number of JVMs are run on the same machine.*

*Vm (JVM) option for enabling serial GC (garbage Collector) in java >*

*-XX:+UseSerialGC*

*Example of Passing Serial GC in Command Line for starting jar>*

*java -Xms256m -Xms512m  -XX:+UseSerialGC -jar d:\MyJar.jar*

*Read in lots of detail about* [*Serial collector / Serial GC (Garbage collector)*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html)

*Question 20. What is Throughput GC (Garbage collector) or Parallel collector in java?*

*Answer.  Experienced developers must know about this garbage collection interview question.*

*1. Features of Throughput GC (Garbage collector) in java  >*

[*Throughput collector*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html) *is also called*

*Throughput GC (garbage collector)*

*ParallelGC (garbage collector)*

*Throughput collector*

*ParallelGC collector*

*Throughput garbage collector is the default garbage collector for JVM in java.*

*Throughput garbage collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application in java.*

*2. When to Use the Throughput GC (Garbage collector) in java >*

*The Throughput garbage collector should be used when application can afford low pauses in java.*

*And application is running on host with multiple CPU’s in java.*

*3. Vm (JVM) option for enabling throughput GC (Garbage collector) in java >*

*-XX:+UseParallelGC*

*Example of using throughput collector in Command Line for starting jar>*

*java -Xms256m -Xms512m  -XX:+UseParallelGC -jar d:\MyJar.jar*

*With this Vm (JVM) option you get a*

*Multi-threaded young generation garbage collector in java,*

*single-threaded old generation garbage collector in java and*

*single-threaded compaction of old generation in java.*

*Vm (JVM) option for enabling throughput collector with n number of threads in java >*

*-XX:ParallelGCThreads=<numberOfThreads>*

*Another Vm (JVM) option for enabling throughput collector in java >*

*-XX:+UseParallelOldGC*

*4. Goals for Throughput GC (Garbage collector) in java >*

*Maximum pause time goal (Highest priority)*

*Throughput goal*

*Minimum footprint goal (Lowest priority)*

*5.Read in more detail about following* [*features of Throughput GC (Garbage collector) in java*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html)

*Read :* [*JVM (java virtual machine) in detail in java*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *and* [*How Garbage Collection (GC) works internally*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html)

[*Top 30 JVM(Java Virtual Machine) interview Questions and answers*](http://www.javamadesoeasy.com/2017/03/top-30-jvmjava-virtual-machine.html)

*Question 21. What is Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java?*

*Answer.  Very important garbage collection interview question for experienced developers and software architects.*

*1. Features of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java  >*

[*Concurrent Mark Sweep Collector*](http://www.javamadesoeasy.com/2016/10/concurrent-mark-sweep-cms-collector.html) *is also called*

*concurrent low pause collector*

*concurrent low pause GC (garbage collector)*

*CMS GC (garbage Collector)*

*CMS Collector*

*concurrent low pause collector*

*concurrent low pause GC (garbage collector)*

*Concurrent Mark Sweep (CMS) collector collects the old/tenured generation in java.*

*Concurrent Mark Sweep (CMS) Collector minimize the pauses by doing most of the garbage collection work concurrently with the application threads in java.*

*Concurrent Mark Sweep (CMS) Collector on live objects >*

*Concurrent Mark Sweep (CMS) Collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap in java.*

*2. When to Use the Concurrent Low Pause Collector in java*

*Concurrent Low Pause Collector should be used if your applications that require low garbage collection pause times in java.*

*Concurrent Low Pause Collector should be used when your application can afford to share processor resources with the garbage collector while the application is running in java.*

*Concurrent Low Pause Collector is beneficial to applications which have a relatively large set of long-lived data (a large tenured generation) and run on machines with two or more processors in java.*

*Examples when to use  Concurrent Mark Sweep (CMS) collector / concurrent low pause collector should be used for >*

*Example 1 - Desktop UI application that respond to events,*

*Example 2 - Web server responding to a request and*

*Example 3 - Database responding to queries.*

*3. Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java >*

*-XX:+UseConcMarkSweepGC*

*Example of using Concurrent Mark Sweep (CMS) collector / concurrent low pause collector in Command Line for starting jar>*

*java -Xms256m -Xms512m  -XX:+UseConcMarkSweepGC -jar d:\MyJar.jar*

*4. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java >*

*As mentioned above Concurrent Mark Sweep (CMS) collector collects the old/tenured generation (i.e. performs Major garbage collection process).*

*5. Heap Structure for CMS garbage Collector*

*CMS garbage collectors didies heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.*

*Young Generation is further divided into Eden, S0 (Survivor space 0) and S1 (Survivor space 1).*

*5.4.6. Detailed Steps in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >*

*Young Generation GC (garbage Collection) in java*

*Live objects are copied from the Eden space and survivor space to the other survivor space.*

*Any older objects that have reached their aging threshold are promoted to old generation.*

*After Young generation GC (garbage Collection) in java*

*After a young GC, the Eden space and one of the survivor spaces is cleared.*

*promoted objects (older objects that have reached their aging threshold in young GC) are are available in old generation.*

*Old Generation GC (garbage Collection) with CMS in java*

*Initial mark phase - (First pause happens/ stop the world event ) - mark live/reachable objects (Example - objects on thread stack, static objects etc.) and elsewhere in the heap (Example - the young generation).*

*Concurrent marking phase - (No pause phase ) -  finds live objects while the application continues to execute.*

*Remark - (Second pause happens/ stop the world events) - It finds objects that were missed during the concurrent marking phase due to the concurrent execution of the application threads.*

*Old Generation GC (garbage Collection) - Sweep phase (Concurrent Sweep phase) in java*

*Sweep phase -  do the concurrent sweep, memory is freed up.*

*Objects that were not marked in the previous phase are deallocated in place.*

*There is no compaction*

*Unmarked objects are equal to Dead Objects.*

*Old Generation GC (garbage Collection) - After Sweeping*

*Reset phase - do the concurrent reset.*

*Read in more detail about following* [*features of Concurrent Mark Sweep (CMS) collector / concurrent low pause garbage collector in java*](http://www.javamadesoeasy.com/2016/10/concurrent-mark-sweep-cms-collector.html)

*Question 22. What is G1 Garbage Collector (or Garbage First) in java?*

*Answer.  Very important garbage collection interview question for senior software developers. But I must say freshers must have at least some knowledge of this garbage collector.*

*1. The G1 garbage collector features -*

[*G1 garbage collector*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html) *is also called*

*G1 garbage collector*

*G1 collector*

*G1 GC (garbage collector)*

*Garbage first collector*

*G1 garbage collector was introduced in Java 7*

*G1 garbage collector  - default garbage collector in* [*Java 9*](http://www.javamadesoeasy.com/2016/08/java-9-new-features.html)

*G1 garbage collector was designed to replace CMS collector(Concurrent Mark-Sweep garbage Collector).*

*G1 garbage collector is parallel,*

*G1 garbage collector is concurrent, and*

*G1 garbage collector is incrementally compacting low-pause garbage collector in java.*

*G1 garbage collector has much better layout from the other garbage collectors like serial, throughput and CMS garbage collectors in java.*

*G1(Garbage First) collector compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions.*

*G1(Garbage First) collector allows customizations by allowing users to specify pause times.*

*G1 Garbage Collector (or Garbage First) limits GC pause times and maximizes throughput.*

*2. Vm (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >*

*-XX:+UseG1GC*

*Example of using G1 Garbage Collector in Command Line for starting jar>*

*java -Xms256m -Xms512m  -XX:+UseG1GC -jar d:\MyJar.jar*

*3. G1(Garbage First) collector functioning >*

*CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.*

*All memory objects end up in one of these three sections.*

*The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.*

*The heap is split/partitioned into many fixed sized regions (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage.*

*4. When to use G1 garbage collector >*

*G1 must be used when applications that require large heaps with limited GC latency.*

*Example - Application that require*

*heaps around 5-6GB or larger and*

*pause time required below 0.5 seconds*

*5. When to switch from CMS (or old garbage collectors) to G1 garbage collector >*

*Applications using CMS garbage collector may switch to G1 when >*

*Full GC durations are too long or too frequent.*

*The rate of object allocation or promotion varies significantly.*

*Long garbage collection (longer than 0.5 to 1 second)*

*6. The G1(Garbage First) collector working Step by Step >*

*The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.*

*6.1. G1(Garbage First) garbage collector Heap Structure >*

*The heap is split/partitioned into many fixed sized regions (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage.*

*Each region’s size is chosen by JVM at startup.*

*Generally heap is divided into 2000 regions by JVM varying in size from 1 to 32Mb.*

*6.2. G1(Garbage First) garbage collector Heap Allocation >*

*As mentioned above there are following region in heap >*

*Eden, survivor and old generation region. Also,*

*Humongous and unused regions are there in heap.*

*6.3. Young Generation in G1 garbage collector*

*Generally heap is divided into 2000 regions by JVM.*

*Minimum size of region can be 1Mb and*

*Maximum size of region can be 32Mb.*

*Regions are not required to be contiguous like CMS garbage collector.*

*Young GC in G1 garbage collector*

*Live objects are copied or moved to survivor regions.*

*If objects aging threshold is met it get promoted to old generation regions.*

*It is STW (stop the world) event. Eden size and survivor size is calculated for the next young GC.*

*The young GC is done parallely using multiple threads.*

*End of a Young GC with G1 garbage collector*

*At this stage Live objects have been evacuated (copied or moved) to >*

*survivor regions or*

*old generation regions.*

*6.4. Old Generation Collection with G1 garbage collector*

*G1 collector is low pause collector for old generation objects.*

*Initial Mark -*

*It is STW (stop the world) event.*

*With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation.*

*Root Region Scanning -*

*Scan survivor regions for references into the old generation.*

*This happens while the application continues to run. The phase must be completed before a young GC can occur.*

*Concurrent Marking -*

*Find live objects over the entire heap.*

*This happens while the application is running.*

*This phase can be interrupted by young generation garbage collections.*

*Remark (Stop the World Event) -*

*Completes the marking of live object in the heap.*

*Uses an algorithm called snapshot-at-the-beginning (SATB) which is much faster than algorithm used in the CMS collector.*

*Cleanup (Stop the World Event and Concurrent) -*

*Performs accounting on live objects and completely free regions. (Stop the world)*

*Young generation and old generation are reclaimed at the same time*

*Old generation regions are selected based on their liveness.*

*Scrubs the Remembered Sets. (Stop the world)*

*Reset the empty regions and return them to the free list. (Concurrent)*

*7. Read in more detail about following features of* [*G1 garbage collector / Garbage first collector in java.*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html)

*Question 23. What are Difference Serial and Throughput gc (garbage Collector) ?*

*Answer.  It is important garbage collection interview question.*

*Serial collector uses one thread to execute garbage collection.*

*Throughput collector uses multiple threads to execute garbage collection.*

*Serial GC is the garbage collector of choice for applications that do not have low pause time requirements and run on client-style machines.*

*Throughput GC is the garbage collector of choice for applications that have low pause time requirements.*

*Question 24. Which methods is called for garbage collection in java?*

*Answer. It is very important garbage collection interview question.*

*Gc (garbage collector) calls finalize method for garbage collection.*

*finalize method is called only once by garbage collector for an object in java.*

*finalize method is in java.lang.Object class.*

*Program to call System.gc() method in Java >*

|  |
| --- |
| *public class RunGarbageCollectorExample {*  *public static void main(String[] args) {*  *System.out.println("About to call garbage collection - using System.gc() method");*  *System.gc();*  *System.out.println("garbage collection - using System.gc() method called");*  *}*  *}*  */\*OUTPUT*  *About to call garbage collection - using System.gc() method*  *garbage collection - using System.gc() method called*  *\*/* |

*Question 25. Can we force early garbage collection in java?*

*Answer. Another garbage collection interview question for experienced developers*

*Yes it is possible to force early garbage collection in java. But how?*

*We can force early gc (garbage collection) in java by using following methods >*

*System.gc();*

*Runtime.getRuntime().gc();*

*System.runFinalization();*

*Runtime.getRuntime().runFinalization();*

*Question 26. Is it good practice to call System.gc() in Java?*

*Answer. Answer in short is No, but let’s learn why we must not call System.gc() in java >*

*First of all calling System.gc() does not guarantee that it will immediately start performing garbage collection.*

*Even calling System.gc() may not do anything. Call to perform garbage collection may be ignored completely.*

*JVM is different for different platforms because* [*java is platform independent language*](http://www.javamadesoeasy.com/2015/06/java-tutorial-advantage-where-is-java.html)*, so you never know about which garbage collector your jvm will run i.e. what algorithm does it follow to perform garbage collection.*

*Read in detail :* [Is it good practice to call System.gc() in Java?](http://www.javamadesoeasy.com/2016/10/is-it-good-practice-to-call-systemgc-in.html)

*Question 27. Is garbage collection guaranteed when we call finalize() methods?*

*Answer. Answer in short is No.*

*By calling finalize method JVM runs the finalize() methods of any objects pending finalization i.e. objects which have been discarded but there finalize method is yet to be run. After finalize method is executed JVM reclaims space from all the discarded objects in java.*

*Calling finalize methods does not guarantee that it will immediately start performing garbage collection.*

*Finalize method execution is not assured - We must not override finalize method to write some critical code of application because methods execution is not assured. Writing some critical code in finalize method and relying on it may make application to go horribly wrong in java.*

*Question 28. Which thread performs garbage collection in java?*

*Answer. It is simple garbage collection interview question.*

[Daemon threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *are low priority threads which runs intermittently in background for doing garbage collection (gc) in java.*

*Question 29. Tell us something about ParNew collector in java?*

*Answer. This is garbage collection interview question for 5 years+ experienced developers only.  ParNew collector is the young generation collector. It is the parallel copy collector, it uses multiple threads in parallel. Vm parameter for enabling ParNew collector is -XX:+UseParNewGC.*

*Question 30. Do you know about PS Scavenge and PS MarkSweep in java?*

*Answer.  This is garbage collection interview question for 3 to 5 years+ experienced developers only.*

*PS Scavenge >*

*PS Scavenge is the Young generation collectors*

*It is the parallel scavenge collector.*

*PS Scavenge uses multiple threads in parallel for garbage collection.*

*Vm parameter for enabling PS Scavenge >*

*-XX:+UseParallelGC*

*PS MarkSweep >*

*PS MarkSweep is the old generation collector.*

*PS MarkSweep is the parallel scavenge mark sweep collector.*

*It uses the multiple threads in parallel for garbage collection.*

*Vm parameter for enabling PS MarkSweep >*

*-XX:+UseParallelOldGC*

*Read :* [*PS Scavenge and PS MarkSweep*](http://www.javamadesoeasy.com/2016/11/ps-scavenge-and-ps-marksweep.html)

*Read :* [*JVM (java virtual machine) in detail in java*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *and* [*How Garbage Collection (GC) works internally*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html)

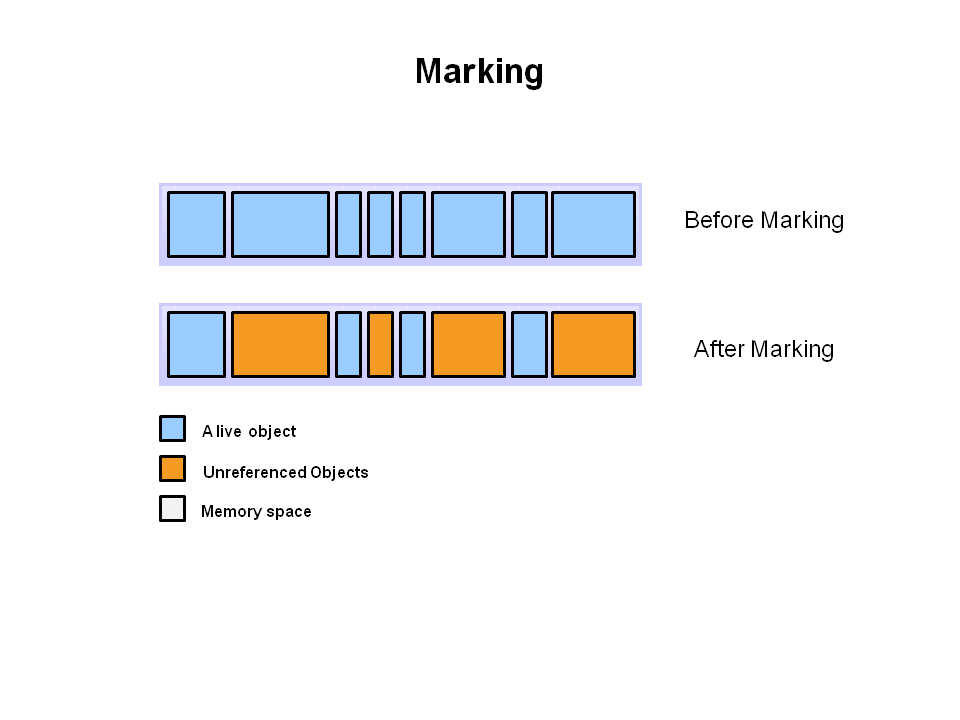
[*Top 30 JVM(Java Virtual Machine) interview Questions and answers*](http://www.javamadesoeasy.com/2017/03/top-30-jvmjava-virtual-machine.html)

*Question 31. How garbage collection is done using Marking and deletion in java?*

*Answer.  It is very very important garbage collection interview question for all java developers.*

*1) Marking*

*Marking is a process in which gc (garbage collector) identifies which parts of memory (occupied by objects) are in use and which are not.*

**

*Before Marking >*

*All the objects are shown in blue, at this stage*

*some of objects might be in use (referenced objects)  and*

*some of objects might not be in use (unreferenced objects) .*

*After Marking >*

*Objects in use (or referenced objects or Alive objects) are shown in blue.*

*Objects not in use (or unreferenced objects) objects are shown in Orange.*

*2) Deletion*

*Step 2a : Normal Deletion*

*Normal deletion removes all the unreferenced objects and*

*leaves referenced objects and pointers to free space.*

*Step 2b : Deletion with Compacting*

*Deletion with Compacting is done to improve the performance than normal deleting.*

*Deletion with Compacting removes all the unreferenced objects and*

*compacts the remaining referenced objects by moving all the referenced objects together.*

*Read in more detail with diagrams about -* [*Marking and deleting objects for garbage collection in java - Mark and sweep algorithm*](http://www.javamadesoeasy.com/2016/10/marking-and-deleting-objects-for.html)

*Question 32. Mention how to use different garbage collectors by passing vm parameters in java?*

*Answer.  It is very important to know how to use different garbage collectors by passing vm parameters.*

*1) Vm (JVM) option for enabling serial GC (garbage Collector) in java >*

[*-XX:+UseSerialGC*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html)

*Example of Passing Serial GC in Command Line for starting jar>*

*java -Xms256m -Xms512m  -XX:+UseSerialGC -jar d:\MyJar.jar*

*2) Vm (JVM) option for enabling throughput GC (Garbage collector) in java >*

[*-XX:+UseParallelGC*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html)

*Or  (throughput collector with n number of threads)*

*-XX:ParallelGCThreads=<numberOfThreads>*

*Or*

*-XX:+UseParallelOldGC*

*3) Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java >*

[*-XX:+UseConcMarkSweepGC*](http://www.javamadesoeasy.com/2016/10/concurrent-mark-sweep-cms-collector.html)

*Or  (CMS garbage collector with n number of threads)*

*-XX:ParallelCMSThreads=<n>*

*4) Vm (JVM) option for enabling G1 Garbage Collector*

[*-XX:+UseG1GC*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html)

*Question 33. Can you make objects eligible for garbage collection in java?*

*Answer.  Another tricky and challenging garbage collection interview question for experienced developers in java.*

*Yes, we can make object explicitly eligible for garbage collection.*

*Object which is set explicitly set to null becomes eligible for gc (garbage collection) in java .*

*Example 1 >*

*String s=”abc”; //s is currently not eligible for gc (garbage collection) in java.*

*s = null;   //Now, s is currently eligible for gc (garbage collection) in java.*

*Example 2 >*

*List list =new ArrayList(); //list is currently not eligible for gc (garbage collection).*

*list = null;   //Now, list is currently eligible for gc (garbage collection).*

*Question 34. What is Difference in garbage collection in C/C++ and Java (Hint : In terms of memory allocation and deallocation of objects)?*

*Answer.  It will test of your awareness and skills to compare java with other languages.*

*In java garbage collection (memory allocation and deallocation of objects) is an automatic process.*

*But, In C and C++ memory allocation and deallocation of objects) is a manual process.*

*Question 35. Does variables declared inside block becomes eligible for gc (garbage collection) when we exit that block in java?*

*Answer.  Yes, All the variables declared inside block becomes eligible for gc (garbage collection) when we exit that block (As scope of those variable is only that block) in java.*

*Example of garbage collection while using block in java -*

|  |
| --- |
| *class MyClass {*  *public static void main(String[] args) {*  *boolean var = false;*  *if (var) { // begin block 1*  *int x = 1; // x is declared inside block*  *//..........*  *//code inside block...*  *//..........*  *} // end block 1 //And now x is eligible for gc (garbage collection)*  *else { // begin block 2*  *int y = 1;*  *//..........*  *//code inside block...*  *//..........*  *} // end block 2 //And now y is eligible for gc (garbage collection)*  *}*  *}* |

*Question 36. Have you used verbose for understanding garbage collection?*

*Answer.  It is garbage collection interview question for experienced developers and software architects.*

*[GC 325407K->83000K(776768K), 0.2300771 secs]*

*GC - GC indicates minor Garbage Collection (i.e. in young generation).*

*325407K - The combined size of live objects before gc(garbage collection).*

*83000K - The combined size of live objects after gc(garbage collection).*

*0.2300771 secs - time it took for gc(garbage collection) to occur.*

*.[Full GC 325407K->83000K(776768K), 0.2300771 secs]*

*Full GC - Full GC Indicates major garbage collection (i.e. in tenured generation).*

*Read more about* [*-verbose:gc*](http://www.javamadesoeasy.com/2016/11/how-to-use-verbosegc-vm-argument.html)

*Question 37. What is monitoring and analyzing the garbage collection in java?*

*Answer.  It is very important garbage collection interview question for specially 3 years+ experienced developers.*

[*Monitoring and analyzing garbage collection*](http://www.javamadesoeasy.com/2016/12/how-to-monitor-and-analyze-garbage.html) *in java can be used to get  following information >*

*Understanding the* [*garbage collection process*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html)*.*

*Understanding how* [*JVM*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *is currently working.*

*What type of garbage collection* [*algorithms*](http://www.javamadesoeasy.com/2016/10/marking-and-deleting-objects-for.html) *are used*

[*Improving garbage collection performance*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html)*,*

*Analyzing Java heap dumps,*

*Monitoring live Java applications,*

*Analyze profiling data,*

*Detecting Memory leak for classes and arrays,*

*Detecting* [*Thread deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)*,*

*Detecting abnormal thread termination,*

*Detecting* [*outOfMemoryError*](http://www.javamadesoeasy.com/2015/05/outofmemoryerror-in-java.html) *problems,*

*Finding System and process CPU utilization thresholds,*

*Find Heap usage thresholds.*

*Find time taken in Garbage collection and* [*Finalizer*](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html) *queue length.*

*Question 38. How to monitor and analyze the garbage collection in java ?*

*Answer.  This is in continuation to above garbage collection interview question. We can use different tools to generate the garbage collection information and then analyze it.*

[*VisualVM*](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html) *-*

*VisualVM is most popular way to generate Thread Dump and is most widely used by developers. It helps us in analyzing threads performance,* [*thread states*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*, CPU consumed by threads etc.*

*For more details please read :*

[*JSTACK*](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html) *- Java stack traces*

*jstack is very easy way to generate Thread dump and is widely used by developers.*

[*-verbose:gc VM argument*](http://www.javamadesoeasy.com/2016/11/how-to-use-verbosegc-vm-argument.html)

[*Jstat*](http://www.javamadesoeasy.com/2016/11/how-to-use-jstat-for-monitoring-garbage.html) *- Java Virtual Machine Statistics Monitoring Tool*

*“The jstat command displays performance statistics for an instrumented Java HotSpot VM. The target JVM is identified by its virtual machine identifier, or vmid option.”*

[*JHAT*](http://www.javamadesoeasy.com/2016/11/how-to-use-jhat-to-analyze-heat-dump.html) *- Java Heap Analysis Tool.*

[*jconsole*](http://docs.oracle.com/javase/7/docs/technotes/tools/share/jconsole.html) *- jconsole option can be used to obtain a heap dump.*

[*hprof*](http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html)

[*eclipse plugin*](http://www.eclipse.org/mat/)

[*HPjmeter*](https://h20392.www2.hpe.com/portal/swdepot/displayProductInfo.do?productNumber=HPJMETER)

[*JFR*](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks001.html) *(Java Flight Recorder) can be used for detecting memory leak.*

*Read in detail :* [*How to monitor and analyze the garbage collection in 10 ways in java*](http://www.javamadesoeasy.com/2016/12/how-to-monitor-and-analyze-garbage.html)

*Question 39. What is memory leak in java? Consequences of memory leak?*

*Answer.  It is must know question especially for experienced developers, freshers must at least have little information on it.*

*Memory leak happens when number of objects(these objects are not needed) created becomes large and time spent in* [*garbage collection*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html) *increases.*

*Ultimately application becomes very slow, non responsive and ends up throwing  OutOfmemoryError.*

*“Memory leaks ends up throwing* [*OutOfmemoryError*](http://www.javamadesoeasy.com/2015/05/outofmemoryerror-in-java.html) *but OutOfmemoryError doesn’t means memory leak in java”.*

*Consequences of memory leak >*

*Application becomes very slow.*

*Time spent in garbage collection increases.*

*Question 40. Can you please explain some scenarios where you have faced memory leak, OR scenarios where memory leak could happen in java?*

*Answer.   It is must know question for experienced developers, 3 to 5 years developers must know at least three scenarios and 5 years+ developes must know at least 5 scenarios. And for software architects this count must be eight.*

*Here I have mentioned summary of memory leak points, for detail on each of these points, please refer* [*Detecting and fixing memory leak in java*](http://www.javamadesoeasy.com/2016/12/detecting-and-fixing-memory-leak-in-java.html)

*Scenarios where memory leak can happen in java >*

*1) Static variables/ fields are not garbage collected and can cause memory leak in java >*

*Static variables are only garbage collected when the class loader which has loaded the class in which static field is there is garbage collected.*

*So, be cautious as these static variables can create a memory leak in java.*

*For more details click here -* [*Static variables are not garbage collected?*](http://www.javamadesoeasy.com/2016/11/static-variables-are-not-garbage.html)

*2) Thread Local Variables can cause memory leak in java >*

*A* [*thread local*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) *variable is member field in the Thread class.*

*Such thread local variable can be used to hold the thread state.*

*But, thread local variable aren’t garbage collector till thread is alive.*

*3) Memory leak while using Autoboxing and unboxing in java >*

*For addition of numbers we must prefer to use primitive data type, not the Object wrapper class.*

*Addition of numbers using Integers turns out into very costly operation in terms of performance, boxing/unboxing and unnecessary object formations.*

*Just imagine a situation where 1000000’s... of number are being added using Integer, it will literally explode our heap memory and boxing/unboxing operations will have adverse effect on performance.*

*Read :* [*Autoboxing and unboxing in java - How it works internally in detail with 10 examples- Widening, AutoBoxing and Var-args*](http://www.javamadesoeasy.com/2015/09/autoboxing-and-unboxing-in-java-how-it.html)

*4) Avoid memory leak using WeakHashMap in java*

*An entry in a* [*WeakHashMap*](http://www.javamadesoeasy.com/2015/04/weakhashmap-in-java.html) *will be automatically removed by garbage collector when its key is no longer in ordinary use. So, using WeakHashMap in place of* [*HashMap*](http://weakhashmap/) *can help us in avoiding memory leaks.*

*5) Using custom key in map without* [*Overriding equals() and hashCode() method*](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html) *can cause memory leak >*

*If custom key is used and* [*equals() and hashCode() method*](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html) *are not overridden then, key will not be retrieved by using get() method.*

*Because get() method internally calls equals() and hashCode() method for retrieving keys.*

*So, these keys will neither be used nor be garbage collected, so it’s a clear case of memory leak.*

*So, to avoid memory leak while using custom key you must always*

[*Override equals() and hashCode() method*](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html)

*Learn how you can use custom key (Employee object) in* [*custom HashMap Custom implementation - put, get, remove*](http://javamadesoeasy.blogspot.in/2015/02/hashmap-custom-implementation-put-get.html)

*So, to avoid memory leak you must try use* [*String, Integer, Long, Double, Float, Short and any other wrapper class as key in HashMap*](http://www.javamadesoeasy.com/2015/04/string-integer-long-double-float-short.html)*.*

*For more details read :* [*How Integer is used as key in Hashmap*](http://www.javamadesoeasy.com/2015/02/how-integer-is-used-as-key-in-hashmap.html)

*3.6) Close JDBC* [*Statement*](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlstatement-in-java.html)*,* [*PreparedStatement*](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlpreparedstatement.html)*,* [*CallableStatement*](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlcallablestatement.html) *,* [*ResultSet*](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-resultset-in-java-types.html) *and Connections in java to avoid memory leaks >*

*You must ensure that you close all the JDBC* [Statement](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlstatement-in-java.html)*,* [PreparedStatement](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlpreparedstatement.html)*,* [CallableStatement](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-javasqlcallablestatement.html) *,* [*ResultSet*](http://www.javamadesoeasy.com/2015/11/jdbc-what-is-resultset-in-java-types.html) *and Connections in java to avoid memory leaks. You must always close all the above mentioned objects in* [*finally block*](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) *in java because finally block is always executed irrespective of exception is thrown or not by java code.*

*Also read :* [*Java JDBC best practices tutorial*](http://www.javamadesoeasy.com/2015/12/java-jdbc-best-practices.html)

*7) Memory leaks can also be caused by native methods in java > Memory allocated through native methods can cause some serious memory leak.*

*8) Memory leak in web applications in java >*

*Unused Objects stored in application scope are memory leak because they are not collected until web application is stopped.*

*Read :* [*JVM (java virtual machine) in detail in java*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *and* [*How Garbage Collection (GC) works internally*](http://www.javamadesoeasy.com/2015/09/how-garbage-collection-works-internally.html)

*Question 41. Comment on relation between OutOfMemoryError and garbage collection in java?*

*Answer.  There is huge relationship between* [*OutOfMemoryError*](http://www.javamadesoeasy.com/2015/05/outofmemoryerror-in-java.html) *and garbage collection in java.*

*OutOfMemoryError may be thrown when an excessive amount of time is being by jvm in performing garbage collection and very little memory is being freed.*

*A long lived application might be unintentionally holding references to objects and this prevents the objects from being garbage collected. Holding of objects for a long time is also a kind of memory leak in java.*

*Question 42. Discuss* [*OutOfMemoryError: GC Overhead limit exceeded*](http://www.javamadesoeasy.com/2017/02/javalangoutofmemoryerror-gc-overhead.html) *in java?*

*Answer.  OutOfMemory*[*Error*](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html)*: GC Overhead limit exceeded - indicates that the garbage collector is running all the time and Java program is making very slow progress.*

*After a GC (garbage collection), if the garbage collector is spending more than 98% of its time in doing garbage collection and if less than 2% of the java heap memory space is reclaimed, then OutOfMemoryError - GC Overhead limit exceeded - is thrown in java.*

*This OutOfMemoryError is generally thrown because all the live objects are not getting garbage collected properly and java heap space is not available for new objects.*

*Question 43. What you know about* [*OutOfMemoryError: permgen*](http://www.javamadesoeasy.com/2016/12/how-to-set-or-change-permgen-size-in.html) *in java?*

*Answer.  Another tricky and important garbage collection interview question which will check your in depth knowledge about jvm and garbage collection. Generally when we are facing java.lang.*[*OutOfMemoryError - Java permgen space*](http://www.javamadesoeasy.com/2015/05/outofmemoryerror-in-java.html)*, then we need to change permgen size of tomcat or eclipse or JVM wherever you are facing this error.*

*Read In detail >* [*OutOfMemoryError: Permgen space - How to set or change permgen size in tomcat server, eclipse?*](http://www.javamadesoeasy.com/2016/12/how-to-set-or-change-permgen-size-in.html)

*Question 44. How to Solve OutOfMemoryError : unable to create new native Thread by passing appropriate jvm parameter?*

*Answer.  A*

*You can resolve “java.lang.*[*OutOfMemoryError : unable to create new native Thread*](http://www.javamadesoeasy.com/2016/11/solve-javalangoutofmemoryerror-unable.html)*” by setting the appropriate size using -Xss vm option.*

*Solution 1 to “java.lang.OutOfMemoryError : unable to create new native Thread”  >*

*Try to increase the the -Xss value so that new threads gets enough stack space.*

*Solution 2 to “java.lang.OutOfMemoryError : unable to create new native Thread”  >*

*Alternatively you could also increase the heap size available using -*[*Xms and -Xmx options*](http://www.javamadesoeasy.com/2016/10/what-are-xms-and-xmx-jvm-parameters-in.html) *and then try to increase and set appropriate -Xss value.*

*Example of using -Xss*

*Pass memory value you want to allocate to thread stack with -Xss.*

*java -Xss512m MyJavaProgram*

*It will set the default stack size of JVM  to 512 megabytes.*

*Question 45. How to Solve OutOfMemoryError : Java heap space by passing appropriate jvm parameter?*

*Answer.  A*

[*OutOfMemoryError : Java heap space*](http://www.javamadesoeasy.com/2017/02/exception-in-thread-javalangoutofmemory.html) *- is thrown whenever there is insufficient space to allocate an object in the Java heap.*

*Does Exception in thread threadName - java.lang.OutOfMemoryError - Java heap space indicates memory leak?*

*No, this OutOfMemoryError does not necessarily means that it is memory leak.*

*Increase the heap size using* [*-Xms and -Xmx*](http://www.javamadesoeasy.com/2016/10/what-are-xms-and-xmx-jvm-parameters-in.html)[*jvm*](http://www.javamadesoeasy.com/2015/06/jvm-java-virtual-machine.html) *parameters as a solution to this issue.*

*Must read:* [*How to set, change, increase or decrease heap size in tomcat server and eclipse to avoid OutOfMemoryError ?*](http://www.javamadesoeasy.com/2016/12/how-to-set-change-increase-or-decrease.html)

*>*[*How to set or change permgen size in tomcat server, eclipse?*](http://www.javamadesoeasy.com/2016/12/how-to-set-or-change-permgen-size-in.html)

*Question 46. How to set appropriate heap size in eclipse in java?*

*Answer.  We can make changes in* [*eclipse.ini file.*](http://www.javamadesoeasy.com/2016/10/what-is-eclipseini-file-how-to-pass.html)

*Where we can configure*

[*-Xms*](http://www.javamadesoeasy.com/2016/10/what-are-xms-and-xmx-jvm-parameters-in.html) *(minimum heap size which is allocated at initialization of JVM),*

*-Xmx (maximum heap size that JVM can use. )*

[*-XX:MaxPermSize*](http://www.javamadesoeasy.com/2016/10/what-are-xxpermsize-and-xxmaxpermsize.html)*: It’s maximum value of Permanent Space that JVM can allot up to.*

*Question 47. What is default garbage collectors for Java 7?*

*Answer.  Another very important garbage collection interview question for all developers.*

[*default garbage collector for Java 7*](http://www.javamadesoeasy.com/2016/12/what-is-default-garbage-collector-for.html) *>*

*For server class machine -* [*parallel collector*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html)*.*

*For client class machine -* [*serial collector*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html)*.*

*Question 48. What is default garbage collectors for Java 8 ?*

*Answer.*

*What is default garbage collector for Java 8 >*

*For server class machine -* [*parallel collector*](http://www.javamadesoeasy.com/2016/10/throughput-gc-garbage-collector-or.html)*.*

*For client class machine -* [*serial collector*](http://www.javamadesoeasy.com/2016/10/serial-collector-serial-gc-garbage.html)*.*

*Question 49. What is default garbage collectors for Java 9?*

*Answer.  Another very important garbage collection interview question for experienced developers.*

*default garbage collector for Java 9 >*

[*G1 garbage collector*](http://www.javamadesoeasy.com/2016/10/g1-garbage-collector-garbage-first.html)

*Read in detail :* [*What are default garbage collectors for Java 7, 8 and 9*](http://www.javamadesoeasy.com/2016/12/what-is-default-garbage-collector-for.html)

*interview Question 50. Throw some light on garbage collection and WeakHashMap in java?*

*Answer.  Another very tricky garbage collection interview question for experienced developers in java.*

*java.util.*[*WeakHashMap*](http://www.javamadesoeasy.com/2015/04/weakhashmap-in-java.html) *is hash table based implementation of the Map interface, with weak keys.*

*An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use. Mapping for a given key will not prevent the key from being discarded by the garbage collector, (i.e. made finalizable, finalized, and then reclaimed). When a key has been discarded its entry is removed from the map in java.*

*java.util.WeakHashMap is implementation of the java.util.*[*Map*](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) *interface in java.*

*The behavior of the java.util.WeakHashMap class depends upon garbage collector*

*The behavior of the WeakHashMap class depends upon garbage collector in java. Because the garbage collector may discard keys at any time, in WeakHashMap it may look like some unknown thread is silently removing entries.*

*Each key object in a WeakHashMap is stored indirectly as the referent of a weak reference. Therefore a key will be removed automatically only after the weak references to it, both inside and outside of the map, have been cleared by the garbage collector.*

*Multithreading*

*Question 1. What is Thread in java?*

*Answer.*

*Threads consumes CPU in best possible manner, hence enables multi processing. Multi threading reduces idle time of CPU which improves performance of application.*

*Thread are light weight process.*

*A thread class belongs to java.lang package.*

*We can create multiple threads in java, even if we don’t create any Thread, one Thread at least  do exist i.e. main thread.*

*Multiple threads run parallely in java.*

*Threads have their own stack.*

*Advantage of Thread : Suppose one thread needs 10 minutes to get certain task, 10 threads used at a time could complete that task in 1 minute, because threads can run parallely.*

*Also Read :* [Top and most important Interview Questions and answers in Java](http://www.javamadesoeasy.com/p/interview-questions.html)

*Question 2. What is difference between Process and Thread in java?*

*Answer.  One process can have multiple Threads,*

*Thread are subdivision of Process. One or more Threads runs in the context of process. Threads can execute any part of process. And same part of process can be executed by multiple Threads.*

*Processes have their own copy of the data segment of the parent process while Threads have direct access to the data segment of its process.*

*Processes have their own address while Threads share the address space of the process that created it.*

*Process creation needs whole lot of stuff to be done, we might need to copy whole parent process, but Thread can be easily created.*

*Processes can easily communicate with child processes but interprocess communication is difficult. While, Threads can easily communicate with other threads of the same process using* [*wait() and notify() methods*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html)*.*

*In process all threads share system resource like heap Memory etc. while Thread has its own stack.*

*Any change made to process does not affect child processes, but any change made to thread can affect the behavior of the other threads of the process.*

[*Example to see where threads on are created on different processes and same process.*](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html)

*Question 3. How to implement Threads in java?*

*Answer.  This is very basic threading question. Threads can be created in two ways i.e. by* [*implementing java.lang.Runnable interface or extending java.lang.Thread class*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *and then extending run method.*

*Thread has its own variables and methods, it lives and dies on the heap.* [*But a thread of execution is an individual process that has its own call stack*](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)*. Thread are lightweight process in java.*

*Thread creation by  implementing java.lang.Runnable interface.*

*We will create object of class which implements Runnable interface :*

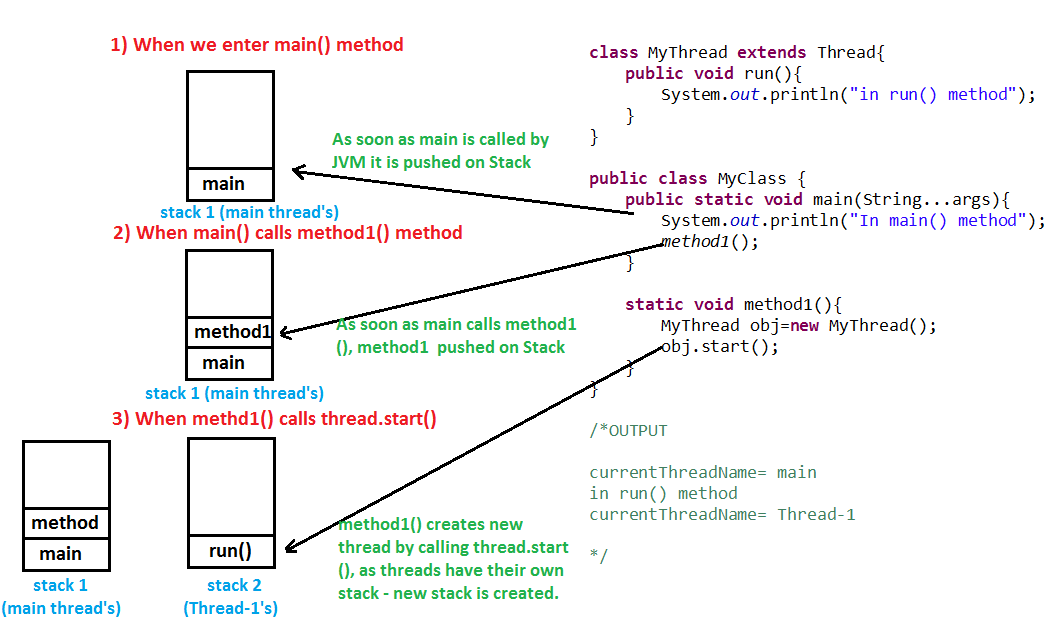
|  |
| --- |
| *MyRunnable runnable=new MyRunnable();*  *Thread thread=new Thread(runnable);* |

*2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e.  runnable object :*

|  |
| --- |
| *Thread thread=new Thread(runnable);* |

*Question 4 . Does Thread implements their own Stack, if yes how? (Important)*

*Answer.  Yes,* [*Threads have their own stack*](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)*. This is very interesting question, where interviewer tends to check your basic knowledge about how* [*threads internally maintains their own stacks*](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)*. I’ll be explaining you the concept by diagram.*

*[](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)*

*Question 5. We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?*

*Answer. Well the answer is you must* [*extend Thread*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method* [*implementing Runnable*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *is the best option (Runnable interface has only one abstract method i.e. run() ).*

[Differences between implementing Runnable interface and extending Thread class](http://www.javamadesoeasy.com/2015/03/differences-between-implementing.html) *-*

*Multiple inheritance in not allowed in java : When we* [*implement Runnable*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *interface we can extend another class as well, but if we extend Thread class we cannot extend any other class because java does not allow multiple inheritance. So, same work is done by implementing Runnable and* [*extending Thread*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *but in case of implementing Runnable we are still left with option of extending some other class. So, it’s better to implement Runnable.*

[Thread safety](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) *: When we implement Runnable interface, same object is shared amongst multiple threads, but when we extend Thread class each and every thread gets associated with new object.*

*Inheritance (Implementing Runnable is lightweight operation) : When we extend Thread unnecessary all Thread class features are inherited, but when we implement Runnable interface no extra feature are inherited, as Runnable only consists only of one abstract method i.e. run() method.  So, implementing Runnable is lightweight operation.*

*Coding to interface : Even java recommends coding to interface. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.*

*Don’t extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run()  : We must* [*extend Thread*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method* [*implementing Runnable*](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) *is the best option (Runnable interface has only one abstract method i.e. run() ). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.*

*Flexibility in code when we implement Runnable : When we extend Thread first a fall all thread features are inherited and our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class. But, when we implement Runnable we create a new thread and pass runnable object as parameter, we could pass runnable object to executorService & much more. So, we have more options when we implement Runnable and our code becomes more flexible.*

*ExecutorService : If we implement Runnable, we can start multiple thread created on runnable object  with ExecutorService (because we can start Runnable object with new threads), but not in the case when we extend Thread (because thread can be started only once).*

*Question 6. How can you say Thread behaviour is unpredictable? (Important)*

*Answer. The solution to question is quite simple,* [*Thread behaviour is unpredictable*](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html) *because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform.*

*To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start  both threads. There is no surety that which threads will complete first,  both threads will enter anonymously in for loop.*

*Question 7 . When threads are not lightweight process in java?*

*Answer. Threads are* [*lightweight process*](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html) *only if threads of same process are executing concurrently. But if threads of different processes are executing concurrently then threads are* [*heavy weight process*](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html)*.*

*Question 8. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)*

*Answer.  Interviewers tend to know interviewees knowledge about Thread methods. So this is time to prove your point by answering correctly. We can use* [*join() method*](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html) *to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words waits for this thread to die. Calling join() method internally calls join(0);*

[*DETAILED DESCRIPTION : Join() method - ensure all threads that started from main must end in order in which they started and also main should end in last. Types of join() method with programs- 10 salient features of join.*](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html)

*Question 9.What is difference between starting thread with run() and start() method? (Important)*

*Answer. This is quite interesting question, it might confuse you a bit and at time may make you think is there really any* [*difference between starting thread with run() and start() method*](http://www.javamadesoeasy.com/2015/03/difference-between-starting-thread-with.html)*.*

*When you call start() method, main thread internally calls run() method to start newly created Thread, so run() method is ultimately called by newly created thread.*

*When you call run() method main thread rather than starting run() method with newly thread it start run() method by itself.*

*Question 10. What is significance of using* [*Volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) *keyword? (Important)*

*Answer. Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables.*

*If a field is declared* [*volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)*, in that case the Java memory model ensures that all threads see a consistent value for the variable.*

*Few small questions>*

*Q. Can we have* [*volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) *methods in java?*

*No, volatile is only a keyword, can be used only with variables.*

*Q. Can we have synchronized variable in java?*

*No, synchronized can be used only with methods, i.e. in method declaration.*

*DETAILED DESCRIPTION :* [*Volatile keyword in java- difference between synchronized and volatile with programs, 10 key points about volatile keyword, why volatile variables are not cached in memory*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)

*Question 11. Differences between synchronized and volatile keyword in Java? (Important)*

*Answer. Its very important question from interview perspective.*

[*Volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) *can be used as a keyword against the variable, we cannot use volatile against method declaration.*

*volatile void method1(){} //it’s illegal, compilation error.*

*volatile int i; //legal*

*While* [*synchronization*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *can be used in method declaration or we can create synchronization blocks (In both cases thread acquires lock on object’s monitor). Variables cannot be synchronized.*

*Synchronized method:*

*synchronized void method2(){} //legal*

*Synchronized block:*

*void method2(){*

*synchronized (this) {*

*//code inside synchronized block.*

*}*

*}*

*Synchronized variable (illegal):*

*synchronized int i; //it’s illegal, compilatiomn error.*

[*Volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) *does not acquire any lock on variable or object, but* [*Synchronization*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *acquires lock on method or block in which it is used.*

[*Volatile*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) *variables are not cached, but variables used inside* [*synchronized*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *method or block are cached.*

*When volatile is used will never create deadlock in program, as volatile never obtains any kind of lock . But in case if synchronization is not done properly, we might end up creating dedlock in program.*

*Synchronization may cost us performance issues, as one thread might be waiting for another thread to release lock on object. But volatile is never expensive in terms of performance.*

*DETAILED DESCRIPTION :* [*Differences between synchronized and volatile keyword in detail with programs.*](http://www.javamadesoeasy.com/2015/03/differences-between-synchronized-and.html)

*Question 12. Can you again start Thread?*

*Answer. No,* [*we cannot start Thread again*](http://www.javamadesoeasy.com/2015/03/can-we-start-thread-again.html)*, doing so will throw runtimeException java.lang.IllegalThreadStateException. The reason is once run() method is executed by Thread, it goes into* [*dead state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*

*Let’s take an example-*

*Thinking of starting thread again and calling start() method on it (which internally is going to call run() method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to dead state.*

*Question 13. What is race condition in multithreading and how can we solve it? (Important)*

*Answer. This is very important question, this forms the core of multi threading, you should be able to explain about* [*race condition in detail*](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html)*. When more than one thread try to access same resource without synchronization causes race condition.*

*So we can* [*solve race condition*](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html) *by using either* [*synchronized block or synchronized method*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*. When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.*

*Few sub questions>*

*What if two threads try to read same resource without* [*synchronization*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*?*

*When two threads try to read on same resource without synchronization, it’s never going to create any problem.*

*What if two threads try to write to same resource without* [*synchronization*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*?*

*When two threads try to write to same resource without synchronization, it’s going to create synchronization problems.*

*Question 14. How threads communicate between each other?*

*Answer. This is very must know question for all the interviewees, you will most probably face this question in almost every time you go for interview.*

*Threads can communicate with each other by using* [*wait(), notify() and notifyAll()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *methods.*

*Question 15. Why wait(), notify()  and notifyAll() are in Object class and not in Thread class? (Important)*

*Answer.*

*Every Object has a monitor, acquiring that monitors allow thread to hold lock on object. But Thread class does not have any monitors.*

*wait(), notify() and notifyAll() are called on objects only > When wait() method is called on object by thread it waits for another thread on that object to release object monitor by calling* [*notify() or notifyAll()*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) *method on that object.*

*When notify() method is called on object by thread it notifies all the threads*

*which are waiting for that object monitor that object monitor is available now.*

*So, this shows that wait(), notify() and notifyAll() are called on objects only.*

[*Now, Straight forward question that comes to mind is how thread acquires object lock by*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

[*acquiring object monitor? Let’s try to understand this basic concept in detail?*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

*Wait(), notify() and notifyAll() method being in Object class allows all the threads created on that object to communicate with other.  [As multiple threads may exist on same object].*

*As multiple threads exists on same object. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).*

*Now let’s discuss one hypothetical scenario, what will happen if Thread class contains wait(), notify() and notifyAll() methods?*

*Having wait(), notify() and notifyAll() methods means Thread class also must have their monitor.*

*Every thread having their monitor will create few problems -*

*>Thread communication problem.*

*>Synchronization on object won’t be possible- Because object has monitor, one object can have multiple threads and thread hold lock on object by holding object monitor. But if each thread will have monitor, we won’t have any way of achieving synchronization.*

*>Inconsistency in state of object (because synchronization won't be possible).*

*Question 16. Is it important to acquire object lock before calling wait(), notify() and notifyAll()?*

*Answer. Yes, it’s mandatory to acquire object lock before calling these methods on object. As discussed above* [*wait(), notify()  and notifyAll()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *methods are always called from* [*Synchronized block*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *only, and as soon as thread enters synchronized block it acquires object lock (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang. IllegalMonitorStateException is thrown at runtime.*

*Wait() method needs to enclosed in try-catch block, because it throws compile time exception i.e. InterruptedException.*

*Question 17. How can you solve consumer producer problem by using wait() and notify() method? (Important)*

*Answer.  Here come the time to answer very very important question from interview perspective. Interviewers tends to check how sound you are in threads inter communication. Because for solving this problem we got to use synchronization blocks, wait() and notify() method very cautiously. If you misplace synchronization block or any of the method, that may cause your program to go horribly wrong. So, before going into this question first i’ll recommend you to understand how to use synchronized blocks, wait() and notify() methods.*

*Key points we need to ensure before programming :*

*>Producer will produce total of 10 products and cannot produce more than 2 products at a time until products are being consumed by consumer.*

*Example> when sharedQueue’s size is 2, wait for consumer to consume (consumer will consume by calling remove(0) method on sharedQueue and reduce sharedQueue’s size). As soon as size is less than 2, producer will start producing.*

*>Consumer can consume only when there are some products to consume.*

*Example> when sharedQueue’s size is 0, wait for producer to produce (producer will produce by calling add() method on sharedQueue and increase sharedQueue’s size).   As soon as size is greater than 0, consumer will start consuming.*

*Explanation of Logic >*

*We will create sharedQueue that will be shared amongst Producer and Consumer. We will now start consumer and producer thread.*

*Note: it does not matter order in which threads are started (because rest of code has taken care of synchronization and key points mentioned above)*

*First we will start consumerThread >*

|  |
| --- |
| *consumerThread.start();* |

*consumerThread will enter run method and call consume() method. There it will check for sharedQueue’s size.*

*-if size is equal to 0 that means producer hasn’t produced any product, wait for producer to produce by using below piece of code-*

|  |
| --- |
| *synchronized (sharedQueue) {*  *while (sharedQueue.size() == 0) {*  *sharedQueue.wait();*  *}*  *}* |

*-if size is greater than 0, consumer will start consuming by using below piece of code.*

|  |
| --- |
| *synchronized (sharedQueue) {*  *Thread.sleep((long)(Math.random() \* 2000));*  *System.out.println("consumed : "+ sharedQueue.remove(0));*  *sharedQueue.notify();*  *}* |

*Than we will start producerThread >*

|  |
| --- |
| *producerThread.start();* |

*producerThread will enter run method and call produce() method. There it will check for sharedQueue’s size.*

*-if size is equal to 2 (i.e. maximum number of products which sharedQueue can hold at a time), wait for consumer to consume by using below piece of code-*

|  |
| --- |
| *synchronized (sharedQueue) {*  *while (sharedQueue.size() == maxSize) { //maxsize is 2*  *sharedQueue.wait();*  *}*  *}* |

*-if size is less than 2, producer will start producing by using below piece of code.*

|  |
| --- |
| *synchronized (sharedQueue) {*  *System.out.println("Produced : " + i);*  *sharedQueue.add(i);*  *Thread.sleep((long)(Math.random() \* 1000));*  *sharedQueue.notify();*  *}* |

*DETAILED DESCRIPTION* [*with program : Solve Consumer Producer problem by using wait() and notify() methods in multithreading.*](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-pattern-by.html)

*Another illustration with program :* [*How to solve Consumer Producer problem by using wait() and notify() methods, where consumer can consume only when production is over.*](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by_2.html)

*Question 18.* [*How to solve Consumer Producer problem without using wait() and notify() methods, where consumer can consume only when production is over.*](http://www.javamadesoeasy.com/2015/03/how-to-solve-consumer-producer-problem.html)*?*

*Answer. In this problem, producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).*

*We will approach by keeping one boolean variable productionInProcess and initially setting it to true, and later when production will be over we will set it to false.*

*DETAILED DESCRIPTION :* [*How to solve Consumer Producer problem without using wait() and notify() methods, where consumer can consume only when production is over.*](http://www.javamadesoeasy.com/2015/03/how-to-solve-consumer-producer-problem.html)

*Question 19. How can you solve consumer producer pattern by using BlockingQueue? (Important)*

*Answer. Now it’s time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using wait() and notify() method. Generally you might wonder why interviewer's are so much interested in asking about* [*solving consumer producer problem using BlockingQueue*](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html)*, answer is they want to know how strong knowledge you have about java concurrent Api’s, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.*

[*BlockingQueue is a interface and we will use its implementation class LinkedBlockingQueue.*](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html)

*Key methods for solving consumer producer pattern are >*

|  |
| --- |
| *put(i);      //used by producer to put/produce in sharedQueue.*  *take(); //used by consumer to take/consume from sharedQueue.* |

*Question 20. What is* [*deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *in multithreading? Write a program to form* [*DeadLock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important)*

*Answer.  This is very important question from interview perspective. But, what makes this question important is it checks interviewees capability of* [*creating and detecting deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)*. If you can write a code to form deadlock, than I am sure you must be well capable in solving that deadlock as well. If not, later on this post we will learn how to solve deadlock as well.*

*First question comes to mind is,* [*what is deadlock in multi threading program*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)*?*

*Deadlock is a situation where two threads are waiting for each other to release lock holded by them on resources.*

*But how* [*deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *could be formed :*

*Thread-1 acquires lock on String.class and then calls* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and Thread-2 acquires lock on Object.class then calls sleep() method and now it waits for Thread-1 to release lock on String.class.*

*Conclusion:*

*Now, Thread-1 is waiting for Thread-2 to release lock on Object.class and Thread-2 is waiting for Thread-1 to release lock on String.class and deadlock is formed.*

|  |
| --- |
| *Code called by Thread-1*    *public void run() {*  *synchronized (String.class) {*  *Thread.sleep(100);*  *synchronized (Object.class) {*  *}*  *}*  *}*  *Code called by Thread-2*    *public void run() {*  *synchronized (Object.class) {*  *Thread.sleep(100);*  *synchronized (String.class) {*  *}*  *}*  *}* |

*Here comes the important part, how above formed deadlock could be solved :*

*Thread-1 acquires lock on String.class and then calls* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and Thread-2 tries to acquire lock on String.class but lock is holded by Thread-1. Meanwhile, Thread-1 completes successfully. As Thread-1 has completed successfully it releases lock on String.class, Thread-2 can now acquire lock on String.class and complete successfully without any deadlock formation.*

*Conclusion: No deadlock is formed.*

|  |
| --- |
| *Code called by Thread-1*    *public void run() {*  *synchronized (String.class) {*  *Thread.sleep(100);*  *synchronized (Object.class) {*  *}*  *}*  *}*  *Code called by Thread-2*    *public void run() {*  *synchronized (String.class) {*  *Thread.sleep(100);*  *synchronized (Object.class) {*  *}*  *}*  *}* |

*Few important measures to avoid* [*Deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *>*

*Lock specific member variables of class rather than locking whole class: We must try to lock specific member variables of class rather than locking whole class.*

*Use join() method: If possible try to use join() method, although it may refrain us from taking full advantage of multithreading environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.*

*If possible try avoid using nested synchronization blocks.*

*Question 21. Have you ever generated thread dumps or analyzed Thread Dumps? (Important)*

*Answer. Answering this questions will show your in depth knowledge of Threads. Every experienced must know how to generate Thread Dumps.*

[*VisualVM*](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html) *is most popular way to generate Thread Dump and is most widely used by developers. It’s important to understand usage of VisualVM for in depth knowledge of VisualVM. I’ll recommend every developer must understand this topic to become master in multi threading.*

*It helps us in analyzing threads performance,* [*thread states*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*, CPU consumed by threads, garbage collection and much more.  For detailed information see* [*Generating and analyzing Thread Dumps using VisualVM - step by step detail to setup VisualVM with screenshots*](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)

[*jstack*](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html) *is very easy way to generate Thread dump and is widely used by developers. I’ll recommend every developer must understand this topic to become master in multi threading. For creating Thread dumps we need not to download any jar or any extra software. For detailed information see* [*Generating and analyzing Thread Dumps using JSATCK - step by step detail to setup JSTACK with screenshots*](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html)*.*

*Question 22. What is life cycle of Thread, explain thread states? (Important)*

*Answer.*[*Thread states/ Thread life cycle*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *is very basic question, before going deep into concepts we must understand Thread life cycle.*

*Thread have following states >*

*New*

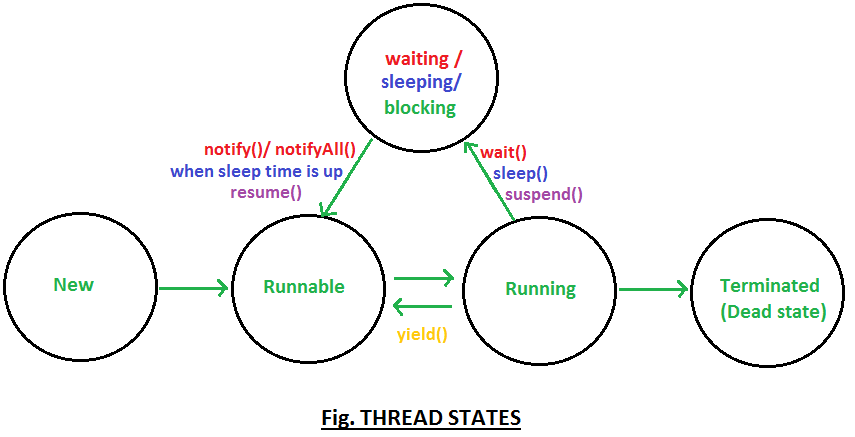
*Runnable*

*Running*

*Waiting/blocked/sleeping*

*Terminated (Dead)*

*Thread states/ Thread life cycle in diagram >*

**

*Thread states in detail >*

*New : When instance of thread is created using new operator it is in new state, but the start() method has not been invoked on the thread yet, thread is not eligible to run yet.*

*Runnable :  When start() method is called on thread it enters runnable state.*

*Running : Thread scheduler selects thread to go from runnable to running state. In running state Thread starts executing by entering run() method.*

*Waiting/blocked/sleeping : In this state a thread is not eligible to run.*

*>Thread is still alive, but currently it’s not eligible to run. In other words.*

*> How can Thread go from running to waiting state?*

*By calling wait()* [*method*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.*

*> How can Thread go from running to sleeping state?*

*By calling sleep()* [*method*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.*

*Terminated (Dead) : A thread is considered dead when its run() method completes.*

*You may like to have in depth knowledge of* [*Thread states/ Thread life cycle in java & explanation of thread methods which method puts thread from which state to which state.*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)

*Question 23. Are you aware of preemptive scheduling and time slicing?*

*Answer. In preemptive scheduling, the highest priority thread executes until it enters into the* [*waiting or dead state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*

*In time slicing, a thread executes for a certain predefined time and then enters runnable pool. Than thread can enter running state when selected by thread scheduler.*

*Question 24. What are* [*daemon threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)*?*

*Answer.* [*Daemon threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *are low priority threads which runs intermittently in background for doing garbage collection.*

*12 Few salient features of* [*daemon() threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)*>*

*Thread scheduler schedules these threads only when CPU is idle.*

[*Daemon threads*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *are service oriented threads, they serves all other threads.*

*These threads are created before user threads are created and die after all other user threads dies.*

*Priority of daemon threads is always 1 (i.e. MIN\_PRIORITY).*

*User created threads are non daemon threads.*

*JVM can exit when only daemon threads exist in system.*

*we can use isDaemon() method to check whether thread is daemon thread or not.*

*we can use setDaemon(boolean on) method to make any user method a daemon thread.*

*If setDaemon(boolean on) is called on thread after calling start() method than IllegalThreadStateException is thrown.*

*You may like to see how daemon threads work, for that you can use VisualVM or jStack. I have provided Thread dumps over there which shows daemon threads which were intermittently running in background.*

*Some of the daemon threads which intermittently run in background are >*

|  |
| --- |
| *"RMI TCP Connection(3)-10.175.2.71" daemon "RMI TCP Connection(idle)" daemon "RMI Scheduler(0)" daemon "C2 CompilerThread1" daemon*  *"GC task thread#0 (ParallelGC)"* |

*Question 25. Why* [*suspend() and resume() methods are deprecated*](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html)*?*

*Answer.* [*Suspend()*](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) *method is* [*deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *prone. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is* [*resumed*](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html)*.* [*If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, it results in deadlock formation*](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html)*.*

*These* [*deadlocks*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *are generally called Frozen processes.*

*Suspend() method puts thread from* [*running to waiting state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*. And thread can go from waiting to runnable state only when resume() method is called on thread. It is deprecated method.*

*Resume() method is only used with suspend() method that’s why it’s also deprecated method.*

*Question 26. Why destroy() methods is deprecated?*

*Answer. This question is again going to check your in depth knowledge of thread methods i.e.* [*destroy() method*](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) *is* [*deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *prone. If the target thread holds a lock on object when it is destroyed, no thread can lock this object (Deadlock formed are similar to deadlock formed when suspend() and resume() methods are used improperly). It results in deadlock formation. These* [*deadlocks*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *are generally called Frozen processes.*

*Additionally you must know calling destroy() method on Threads throw runtimeException i.e. NoSuchMethodError.* [*Destroy() method*](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) *puts thread from running to* [*dead state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*

*Question 27. As stop() method is deprecated,  How can we terminate or stop infinitely running thread in java? (Important)*

*Answer. This is very interesting question where interviewees thread basics basic will be tested. Interviewers tend to know user’s knowledge about main thread’s and thread invoked by main thread.*

*We will try to address the problem by creating new thread which will run infinitely until certain condition is satisfied and will be called by main Thread.*

*Infinitely running thread can be stopped using boolean variable.*

[*Infinitely running thread can be stopped using interrupt() method*](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html)*.*

*Let’s understand Why stop() method is deprecated :*

*Stopping a thread with Thread.stop() causes it to release all of the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, which might lead to unpredictable behavior.*

*Question 28. what is significance of yield() method, what state does it put thread in?*

[*yield()*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *is a native method it’s implementation in java 6 has been changed as compared to its implementation java 5. As method is native it’s implementation is provided by JVM.*

*In java 5, yield() method internally used to call* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method giving all the other threads of same or higher priority to execute before yielded thread by leaving allocated CPU for time gap of 15 millisec.*

*But java 6, calling yield() method gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint. So, sometimes even after using yield() method, you may not notice any difference in output.*

*salient features of* [*yield()*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *method >*

*Definition :* [*yield()*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint.*

[*Thread state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *: when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.*

*Waiting time : yield() method stops thread for unpredictable time.*

*Static method : yield() is a static method, hence calling Thread.yield() causes currently executing thread to yield.*

*Native method : implementation of yield() method is provided by JVM.*

*Let’s see definition of yield() method as given in java.lang.Thread -*

|  |
| --- |
| *public static native void yield();* |

[*synchronized block*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *: thread need not to to acquire object lock before calling yield() method i.e. yield() method can be called from outside synchronized block.*

*Question 29.What is significance of sleep() method in detail, what* [*state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *does it put thread in ?*

[*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *is a native method, its implementation is provided by JVM.*

*10 salient features of* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method >*

*Definition :* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *methods causes current thread to sleep for specified number of milliseconds (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.*

[*Thread state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *: when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.*

*Exception : sleep() method must catch or throw compile time exception i.e. InterruptedException.*

*Waiting time : sleep() method have got few options.*

*sleep(long millis) - Causes the currently executing thread to sleep for the specified number of milliseconds*

|  |
| --- |
| *public static native void sleep(long millis) throws InterruptedException;* |

*sleep(long millis, int nanos) - Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.*

|  |
| --- |
| *public static native void sleep(long millis,int nanos) throws InterruptedException;* |

*static method : sleep() is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.*

*Native method : implementation of sleep() method is provided by JVM.*

*Let’s see definition of yield() method as given in java.lang.Thread -*

|  |
| --- |
| *public static native void sleep(long millis) throws InterruptedException;* |

*Belongs to which class :* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method belongs to java.lang.Thread class.*

*synchronized block : thread need not to to acquire object lock before calling sleep() method i.e. sleep() method can be called from outside synchronized block.*

*Question 30. Difference between* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *and* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *? (Important)*

*Answer.*

*Should be called from* [*synchronized block*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *: wait() method is always called from synchronized block i.e.* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *method needs to lock object monitor before object on which it is called.  But sleep() method can be called from outside synchronized block i.e. sleep() method doesn’t need any object monitor.*

*IllegalMonitorStateException : if wait() method is called without acquiring object lock than IllegalMonitorStateException is thrown at runtime, but sleep() method never throws such exception.*

*Belongs to which class : wait() method belongs to java.lang.Object class but sleep() method belongs to java.lang.Thread class.*

*Called on object or thread : wait() method is called on objects but sleep() method is called on Threads not objects.*

[*Thread state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *: when wait() method is called on object, thread that holded object’s monitor goes from running to waiting state and can return to runnable state only when notify() or notifyAll() method is called on that object. And later thread scheduler schedules that thread to go from from runnable to running state.*

*when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.*

*When called from* [*synchronized block*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *: when wait() method is called thread leaves the object lock.  But sleep() method when called from synchronized block or method thread doesn’t leaves object lock.*

*Question 31. Differences and similarities between yield() and sleep() ?*

*Answer.*

*Differences* [*yield()*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *and* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *:*

*Definition : yield() method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint. sleep() methods causes current thread to sleep for specified number of milliseconds (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.*

[*Thread state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *: when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up. when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.*

*Exception : yield() method need not to catch or throw any exception. But sleep() method must catch or throw compile time exception i.e. InterruptedException.*

*Waiting time : yield() method stops thread for unpredictable time, that depends on thread scheduler. But sleep() method have got few options.*

*sleep(long millis) - Causes the currently executing thread to sleep for the specified number of milliseconds*

*sleep(long millis, int nanos) - Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.*

*similarity between* [*yield()*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *and* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)*:*

*> yield() and sleep() method belongs to java.lang.Thread class.*

*> yield() and sleep() method can be called from outside synchronized block.*

*> yield() and sleep() method are called on Threads not objects.*

*Question 32. Mention some g*[*uidelines to write thread safe code, most important point we must take care of in multithreading programs*](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)*?*

*Answer.  In multithreading environment it’s important very important to* [*write thread safe code*](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)*, thread unsafe code can cause a major threat to your application. I have posted many articles regarding thread safety. So overall this will be revision of what we have learned so far i.e. writing thread safe healthy code and avoiding any kind of* [*deadlocks*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)*.*

*If method is exposed in multithreading environment and it’s not synchronized (thread unsafe) than it might lead us to* [*race condition*](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html)*, we must try to use* [*synchronized block and synchronized methods*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*.* [*Multiple threads may exist on same object*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html) *but only one thread of that object can enter synchronized method at a time, though*[*threads on different object*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) *can enter same method at same time.*

*Even static variables are not thread safe, they are used in static methods and if static methods are not synchronized then thread on same or different object can enter method concurrently. Multiple threads may exist on* [*same*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) *or* [*different objects*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) *of class but only one thread can enter* [*static synchronized method*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *at a time, we must consider making* [*static methods as synchronized*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html)*.*

*If possible, try to use* [*volatile variables*](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)*. If a field is declared volatile all threads see a consistent value for the variable. Volatile variables at times can be used as alternate to synchronized methods as well.*

*Final variables are thread safe because once assigned some reference of object they cannot point to reference of other object.*

*s is pointing to String object.*

|  |
| --- |
| *public class MyClass {*  *final String s=new String("a");*  *void method(){*  *s="b"; //compilation error, s cannot point to new reference.*  *}*  *}* |

*If final is holding some primitive value it cannot point to other value.*

|  |
| --- |
| *public class MyClass {*  *final int i=0;*  *void method(){*  *i=0;  //compilation error, i cannot point to new value.*  *}*  *}* |

*Usage of local variables : If possible try to use local variables, local variables are thread safe, because every* [*thread has its own stack*](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)*, i.e. every thread has its own local variables and its pushes all the local variables on stack.*

|  |
| --- |
| *public class MyClass {*  *void method(){*  *int i=0; //Local variable, is thread safe.*  *}*  *}* |

*We must avoid using*[*deadlock prone*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *deprecated thread methods such as* [*destroy()*](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html)*,* [*stop()*](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html)*,* [*suspend() and resume()*](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html)*.*

*Using thread safe collections : Rather than using ArrayList we must Vector and in place of using HashMap we must use ConcurrentHashMap or HashTable.*

*We must use* [*VisualVM*](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html) *or* [*jstack*](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html) *to detect problems such as deadlocks and time taken by threads to complete in multi threading programs.*

*Using* [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) *: ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.*

*Rather than StringBuffer try using immutable classes such as String. Any change to String produces new String.*

*Question 33. How thread can enter waiting, sleeping and blocked state and how can they go to runnable state ?*

*Answer.  This is very prominently asked question in interview which will test your knowledge about* [*thread states*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*. And it’s very important for developers to have in depth knowledge of this* [*thread state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *transition. I will try to explain this thread state transition by framing few sub questions. I hope reading sub questions will be quite interesting.*

*> How can Thread go from running to waiting state ?*

*By calling wait()* [*method*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.*

*> How can Thread return from waiting to runnable state ?*

*Once notify() or notifyAll()* [*method*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) *is called object monitor/lock becomes available and thread can again return to runnable state.*

*> How can Thread go from running to sleeping state ?*

*By calling sleep()* [*method*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *thread go from running to* [*sleeping*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *state. In sleeping state it will wait for sleep time to get over.*

*> How can Thread return from sleeping to runnable state ?*

*Once specified sleep time is up thread can again return to runnable state.*

*Suspend()* [*method*](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) *can be used to put thread in waiting state and resume() method is the only way which could put thread in runnable state.*

*Thread also may go from running to waiting state if it is waiting for some I/O operation to take place. Once input is available thread may return to running state.*

*>When threads are in running state, yield()* [*method*](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) *can make thread to go in Runnable state.*

*Question 34. Difference between notify() and notifyAll() methods, can you write a code to prove your point?*

*Answer. Goodness. Theoretically you must have heard or you must be aware of differences between* [*notify() and notifyAll()*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html)*.But have you created program to achieve it? If not let’s do it.*

*First, I will like give you a brief description of what notify() and notifyAll() methods do.*

*notify() - Wakes up a single thread that is* [*waiting*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is random and occurs at the discretion of the implementation. A thread* [*waits*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *on an object's monitor by calling one of the wait methods.*

[*The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.*](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| *public final native void notify();* |

*notifyAll() - Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.*

[*The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.*](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| *public final native void notifyAll();* |

[*Now it’s time to write down a program to prove the point.*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html)

*Question 35. Does thread leaves object lock when* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method is called?*

*Answer. When* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *method is called Thread does not leaves object lock and goes from running to waiting state. Thread* [*waits*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for sleep time to over and once sleep time is up it goes from* [*waiting to runnable state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*

*Question 36. Does thread leaves object lock when wait() method is called?*

*Answer. When* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *method is called Thread leaves the object lock and goes from* [*running to waiting state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*. Thread waits for other threads on same object to call notify() or notifyAll() and once any of* [*notify() or notifyAll()*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) *is called it goes from waiting to runnable state and again acquires object lock.*

*Question 37. What will happen if we don’t override run method?*

*Answer.  This question will test your basic knowledge how start and run methods work internally in Thread Api.*

*When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don’t override run() method newly created thread won’t be called and nothing will happen.*

|  |
| --- |
| *class MyThread extends Thread {*  *//don't override run() method*  *}*  *public class DontOverrideRun {*  *public static void main(String[] args) {*  *System.out.println("main has started.");*  *MyThread thread1=new MyThread();*  *thread1.start();*  *System.out.println("main has ended.");*  *}*  *}*  */\*OUTPUT*  *main has started.*  *main has ended.*  *\*/* |

*As we saw in output, we didn’t override run() method that’s why on calling start() method nothing happened.*

*Question 38. What will happen if we override start method?*

*Answer. This question will again test your basic core java knowledge how overriding works at runtime, what what will be called at runtime and how start and run methods work internally in Thread Api.*

*When we call start() method on thread, it internally calls run() method with newly created thread. So, if we override start() method, run() method will not be called until we write code for calling run() method.*

|  |
| --- |
| *class MyThread extends Thread {*  *@Override*  *public void run() {*  *System.out.println("in run() method");*  *}*    *@Override*  *public void start(){*  *System.out.println("In start() method");*  *}*    *}*  *public class OverrideStartMethod {*  *public static void main(String[] args) {*  *System.out.println("main has started.");*    *MyThread thread1=new MyThread();*  *thread1.start();*    *System.out.println("main has ended.");*  *}*  *}*  */\*OUTPUT*  *main has started.*  *In start() method*  *main has ended.*  *\*/* |

*If we note output. we have overridden start method and didn’t called run() method from it, so, run() method wasn’t call.*

*Question 39. Can we acquire lock on class? What are ways in which you can acquire lock on class?*

*Answer.  Yes, we can acquire lock on* [*class’s class object in 2 ways to acquire lock on class*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html)*.*

*Thread can acquire lock on class’s class object by-*

*Entering synchronized block or*

*Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass*

*synchronized (MyClass.class) {*

*//thread has acquired lock on MyClass’s class object.*

*}*

*by entering static synchronized methods.*

*public static synchronized void method1() {*

*//thread has acquired lock on MyRunnable’s class object.*

*}*

*As soon as thread entered Synchronization method, thread acquired lock on class’s class object.*

*Thread will leave lock when it exits static synchronized method.*

*Question 40. Difference between object lock and class lock?*

*Answer.  It is very important question from multithreading point of view. We must understand* [*difference between object lock and class lock*](http://www.javamadesoeasy.com/2015/03/difference-between-object-lock-and.html) *to answer interview, ocjp answers correctly.*

|  |  |
| --- | --- |
| [*Object lock*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) | [*Class lock*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) |
| *Thread can acquire* [*object lock*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *by-*  *Entering synchronized block or*  *by entering synchronized methods.* | *Thread can acquire lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *by-*  *Entering synchronized block or*  *by entering static synchronized methods.* |
| [*Multiple threads may exist on same object but only one thread of that object can enter synchronized method at a time.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)  [*Threads on different object can enter same method at same time.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) | *Multiple threads may exist on* [*same*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) *or* [*different objects*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) *of class but only one thread can enter static synchronized method at a time.* |
| *Multiple objects of class may exist and every object has it’s own lock.* | *Multiple objects of class may exist but there is always one class’s class object lock available.* |
| *First let’s acquire* [*object lock*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *by entering synchronized block.*  *Example- Let’s say there is one class MyClass and we have created it’s object and reference to that object is myClass. Now we can create synchronization block, and parameter passed with synchronization tells which object has to be synchronized. In below code, we have synchronized object reference by myClass.*  *MyClass myClass=new Myclass();*  *synchronized (myClass) {*  *}*  *As soon thread entered Synchronization block, thread acquired object lock on object referenced by myClass (by acquiring object’s monitor.)*  *Thread will leave lock when it exits synchronized block.* | *First let’s acquire lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *by entering synchronized block.*  *Example- Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass*  *synchronized (MyClass.class) {*  *}*  *As soon as thread entered Synchronization block, thread acquired MyClass’s class object. Thread will leave lock when it exits synchronized block.* |
| *public synchronized void method1() {*  *}*  *As soon as thread entered Synchronization method, thread acquired* [*object lock*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*.*  *Thread will leave lock when it exits synchronized method.* | *public static synchronized void method1() {}*  *As soon as thread entered static Synchronization method, thread acquired lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html)*.*  *Thread will leave lock when it exits synchronized method.* |

*Let’s me give you some tricky situation based question,*

*Question 41. Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?*

*Answer. No, here when Thread-1 is in synchronized method1() it must be holding* [*lock on object’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *and will release lock on object’s monitor only when it exits synchronized method1(). So, Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *for Thread-1 to release lock on object’s monitor so that it could enter synchronized method2().*

*Likewise, Thread-2 even cannot enter synchronized method1() which is being executed by Thread-1. Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for Thread-1 to release lock on object’s monitor so that it could enter synchronized method1().* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)

*Question 42. Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?*

*Answer. No, here when Thread-1 is in static synchronized method1() it must be holding lock on* [*class class’s object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *and will release lock on class’s class object only when it exits static synchronized method1(). So, Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *for Thread-1 to release lock on class’s class object so that it could enter static synchronized method2().*

*Likewise, Thread-2 even cannot enter static synchronized method1() which is being executed by Thread-1. Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for Thread-1 to release lock on  class’s class object so that it could enter static synchronized method1().* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html)

*Question 43. Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter static synchronized method2() at same time?*

*Answer. Yes, here when Thread-1 is in synchronized method1() it must be holding* [*lock on object’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *and Thread-2 can enter static synchronized method2() by acquiring lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html)*.* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_65.html)

*Question 44. Suppose you have thread and it is in synchronized method and now can thread enter other synchronized method from that method?*

*Answer. Yes, here when thread is in synchronized method it must be holding* [*lock on object’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *and using that lock thread can enter other synchronized method.* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_5.html)

*Question 45. Suppose you have thread and it is in static synchronized method and now can thread enter other static synchronized method from that method?*

*Answer.  Yes, here when thread is in static synchronized method it must be holding lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *and using that lock thread can enter other static synchronized method.* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_16.html)

*Question 46. Suppose you have thread and it is in static synchronized method and now can thread enter other non static synchronized method from that method?*

*Answer. Yes, here when thread is in static synchronized method it must be holding lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *and when it enters synchronized method it will hold* [*lock on object’s monitor*](http://v/) *as well.*

*So, now thread holds 2 locks (it’s also called nested synchronization)-*

*>first one on class’s class object.*

*>second one on object’s monitor (This lock will be released when thread exits non static method).* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_41.html)

*Question 47. Suppose you have thread and it is in synchronized method and now can thread enter other static synchronized method from that method?*

*Answer. Yes, here when thread is in synchronized method it must be holding* [*lock on object’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) *and when it enters static synchronized method it will hold lock on* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *as well.*

*So, now thread holds 2 locks (it’s also called nested synchronization)-*

*>first one on* [*object’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*.*

*>second one on class’s class object.(This lock will be released when thread exits static method).* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_17.html)

*Question 48. Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?*

*Answer. Yes, here when Thread-1 is in synchronized method1() it must be holding* [*lock on object1’s monitor*](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)*. Thread-2 will acquire lock on object2’s monitor and enter synchronized method2().*

*Likewise, Thread-2 even enter synchronized method1() as well which is being executed by Thread-1 (because threads are created on different objects).* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html)

*Question 49. Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?*

*Answer. No, it might confuse you a bit that threads are created on different objects. But, not to forgot that multiple objects may exist but there is always one* [*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *lock available.*

*Here, when Thread-1 is in static synchronized method1() it must be holding lock on class class’s object and will release lock on class’s class object only when it exits static synchronized method1(). So, Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for Thread-1 to release lock on class’s class object so that it could enter static synchronized method2().*

*Likewise, Thread-2 even cannot enter static synchronized method1() which is being executed by Thread-1. Thread-2 will have to* [*wait*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for Thread-1 to release lock on*[*class’s class object*](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) *so that it could enter static synchronized method1().* [*Now, let’s see a program to prove our point.*](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html)

*Question 50. Difference between wait() and wait(long timeout), What are* [*thread states*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) *when these method are called?*

*Answer.*

|  |  |
| --- | --- |
| [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) | *wait(long timeout)* |
| *When* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *method is called on object, it causes causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object.* | *wait(long timeout) - Causes the current thread to wait until either another thread invokes the notify() or notifyAll() methods for this object, or a specified timeout time has elapsed.* |
| *When* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *is called on object - Thread enters from* [*running to waiting state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*  *It* [*waits*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *for some other thread to call notify so that it could enter runnable state.* | *When wait(1000) is called on object - Thread enters from running to waiting state. Than even if notify() or notifyAll() is not called after  timeout time has elapsed thread will go from* [*waiting to runnable state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.* |

*Question 51.  How can you implement your own Thread Pool in java?*

*Answer.*

*What is* [ThreadPool](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html)*?*

*ThreadPool is a pool of threads which reuses a fixed number of threads  to execute tasks.*

*At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.*

*ThreadPool implementation internally uses* [*LinkedBlockingQueue*](http://www.javamadesoeasy.com/2015/03/custom-implementation-of.html) *for adding and removing tasks.*

*In this post i will be using LinkedBlockingQueue provide by java Api, you can refer this post for* [*implementing ThreadPool using custom LinkedBlockingQueue*](http://www.javamadesoeasy.com/2015/03/implementing-threadpool-using-custom.html)*.*

*Need/Advantage of ThreadPool?*

*Instead of creating new thread every time for executing tasks, we can create ThreadPool which reuses a fixed number of threads for executing tasks.*

*As threads are reused, performance of our application improves drastically.*

*How ThreadPool works?*

*We will instantiate ThreadPool, in ThreadPool’s constructor nThreads number of threads are created and started.*

|  |
| --- |
| *ThreadPool threadPool=new ThreadPool(2);* |

*Here 2 threads will be created and started in ThreadPool.*

*Then, threads will enter run() method of ThreadPoolsThread class and will call take() method on taskQueue.*

*If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable).*

|  |
| --- |
| *public void run() {*  *. . .*  *while (true) {*  *. . .*  *Runnable runnable = taskQueue.take();*  *runnable.run();*  *. . .*  *}*  *. . .*  *}* |

*Else waits for tasks to become available.*

*When tasks are added?*

*When execute() method of ThreadPool is called, it internally calls put() method on taskQueue to add tasks.*

|  |
| --- |
| *taskQueue.put(task);* |

*Once tasks are available all waiting threads are notified that task is available.*

*More detail on how to* [*Implement Thread pool in java*](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html)*.*

*Question 52.  What is significance of using* [*ThreadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)*?*

*Answer.  This question will test your command in multi threading, can you really create some perfect multithreading application or not.* [*ThreadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) *is a class which provides thread-local variables.*

*What is* [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) *?*

*ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.*

*For how long Thread holds ThreadLocal value?*

*Thread holds ThreadLocal value till it hasn’t entered* [*dead state*](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)*.*

*Can one thread see other thread’s ThreadLocal value?*

*No, thread can see only it’s ThreadLocal value.*

*Are ThreadLocal variables thread safe. Why?*

*Yes, ThreadLocal variables are thread safe. As every thread has its own ThreadLocal value and one thread can’t see other threads ThreadLocal value.*

*Application of* [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)*?*

*ThreadLocal are used by many web frameworks for maintaining some context (may be session or request) related value.*

*In any single threaded application, same thread is assigned for every request made to same action, so ThreadLocal values will be available in next request as well.*

*In multi threaded application, different thread is assigned for every request made to same action, so ThreadLocal values will be different for every request.*

*When threads have started at different time they might like to store time at which they have started. So, thread’s start time can be stored in ThreadLocal.*

*Creating ThreadLocal >*

|  |
| --- |
| *private ThreadLocal<String> threadLocal =   new ThreadLocal<String>();* |

*We will create instance of ThreadLocal. ThreadLocal is a generic class, i will be using String to demonstrate threadLocal.*

*All threads will see same instance of ThreadLocal, but a thread will be able to see value which was set by it only.*

*How thread set value of ThreadLocal >*

|  |
| --- |
| *threadLocal.set( new Date().toString());* |

*Thread set value of ThreadLocal by calling set(“”) method on threadLocal.*

*How thread get value of ThreadLocal >*

|  |
| --- |
| *threadLocal.get()* |

*Thread get value of ThreadLocal by calling get() method on threadLocal.*

*See here for detailed explanation of* [*threadLocal*](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)*.*

*Question 53. What is busy spin?*

*Answer.*

*What is* [busy spin](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html)*?*

*When one thread loops continuously waiting for another thread to signal.*

*Performance point of view - Busy spin is very bad from performance point of view, because one thread keeps on looping continuously ( and consumes CPU) waiting for another thread to signal.*

*Solution to busy spin -*

*We must use* [*sleep()*](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) *or* [*wait() and notify()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *method. Using wait() is better option.*

*Why using wait() and notify() is much better option to solve busy spin?*

*Because in case when we use sleep() method, thread will wake up again and again after specified sleep time until boolean variable is true. But, in case of wait() thread will wake up only when when notified by calling* [*notify() or notifyAll()*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html)*, hence end up consuming CPU in best possible manner.*

*Program - Consumer Producer problem with busy spin >*

*Consumer thread continuously execute (busy spin) in while loop till productionInProcess is true. Once producer thread has ended it will make boolean variable productionInProcess false and busy spin will be over.*

|  |
| --- |
| *while(productionInProcess){*  *System.out.println("BUSY SPIN - Consumer waiting for production to get over");*  *}* |

[*See here for Busy spin in detail*](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html)*.*

*Question 54. Can a constructor be synchronized?*

*Answer.  No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.*

*Enclosing constructor in synchronized block will generate compilation error.*

*Using synchronized in constructor definition will also show compilation error.*

*COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted*

*Though we can use synchronized block inside constructor.*

*Read More about :* [Constructor in java cannot be synchronized](http://www.javamadesoeasy.com/2015/03/constructor-in-java-cannot-be.html)

*Question 55. Can you find whether thread holds lock on object or not?*

*Answer.  holdsLock(object) method can be used to find out whether current thread holds the lock on monitor of specified object.*

*holdsLock(object) method returns true if the current thread holds the lock on monitor of specified object.*

*Question 56. What do you mean by thread starvation?*

*Answer.   When thread does not enough CPU for its execution Thread starvation happens.*

*Thread starvation may happen in following scenarios >*

*Low priority threads gets less CPU (time for execution) as compared to high priority threads. Lower priority thread may starve away waiting to get enough CPU to perform calculations.*

*In* [*deadlock*](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) *two threads waits for each other to release lock holded by them on resources. There both Threads starves away to get CPU.*

*Thread might be waiting indefinitely for lock on object’s monitor (by calling* [*wait()*](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) *method), because no other thread is calling* [*notify()/notifAll()*](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) *method on object. In that case, Thread starves away to get CPU.*

*Thread might be waiting indefinitely for lock on object’s monitor (by calling wait() method), but notify() may be repeatedly awakening some other threads. In that case also Thread starves away to get CPU.*

*Question 57. What is addShutdownHook method in java?*

*Answer.*[*addShutdownHook*](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html) *method in java >*

*addShutdownHook method registers a new virtual-machine shutdown hook.*

*A shutdown hook is a initialized but unstarted thread.*

*When JVM starts its shutdown it will start all registered shutdown hooks in some unspecified order and let them run concurrently.*

*When JVM (Java virtual machine)  shuts down >*

*When the last non-*[*daemon*](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) *thread finishes, or*

*when the System.exit is called.*

*Once JVM’s shutdown has begun new shutdown hook cannot be registered neither  previously-registered hook can be de-registered. Any attempt made to do any of these operations causes an IllegalStateException.*

*For more detail with program read :* [Threads addShutdownHook method in java](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html)

*Question 58. How you can handle uncaught runtime exception generated in run method?*

*Answer.  We can use* [*setDefaultUncaughtExceptionHandler*](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html) *method which can handle uncaught unchecked(runtime) exception generated in run() method.*

*What is setDefaultUncaughtExceptionHandler method?*

*setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.*

*setDefaultUncaughtExceptionHandler method features >*

*setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.*

*setDefaultUncaughtExceptionHandler is a static method method, so we can directly call  Thread.setDefaultUncaughtExceptionHandler to set the default handler to handle uncaught unchecked(runtime) exception.*

*It avoids abrupt termination of thread caused by uncaught runtime exceptions.*

*Defining setDefaultUncaughtExceptionHandler method >*

|  |
| --- |
| *Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler(){*  *public void uncaughtException(Thread thread, Throwable throwable) {*  *System.out.println(thread.getName() + " has thrown " + throwable);*  *}*  *});* |

*For more detail read :* [Program to demonstrate setDefaultUncaughtExceptionHandler method.](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html)

*Question 59. What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?*

*Answer.  When program starts JVM creates  a ThreadGroup named main. Unless specified, all  newly created threads become members of the main thread group.*

*ThreadGroup is initialized with default priority of 10.*

*ThreadGroup important methods >*

*getName()*

*name of ThreadGroup.*

*activeGroupCount()*

*count of active groups in ThreadGroup.*

*activeCount()*

*count of active threads in ThreadGroup.*

*list()*

*list() method has prints ThreadGroups information*

*getMaxPriority()*

*Method returns the maximum priority of ThreadGroup.*

*setMaxPriority(int pri)*

*Sets the maximum priority of ThreadGroup.*

*Read more about* [ThreadGroup in java](http://www.javamadesoeasy.com/2015/03/threadgroup-in-java.html)*.*

*Question 60. What are thread priorities?*

*Answer.*

[Thread Priority](http://www.javamadesoeasy.com/2015/03/thread-priorities-setpriority-and.html) *range is from 1 to 10.*

*Where 1 is minimum priority and 10 is maximum priority.*

*Thread class provides variables of final static int type for setting thread priority.*

|  |
| --- |
| */\* The minimum priority that a thread can have. \*/*  *public final static int MIN\_PRIORITY = 1;*    */\* The default priority that is assigned to a thread. \*/*  *public final static int NORM\_PRIORITY = 5;*  */\* The maximum priority that a thread can have. \*/*  *public final static int MAX\_PRIORITY = 10;* |

*Thread with MAX\_PRIORITY is likely to get more CPU as compared to low priority threads. But occasionally low priority thread might get more CPU. Because thread scheduler schedules thread on discretion of implementation and* [*thread behaviour is totally unpredictable*](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html)*.*

*Thread with MIN\_PRIORITY is likely to get less CPU as compared to high priority threads. But occasionally high priority thread might less CPU. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.*

*setPriority() method is used for Changing the priority of thread.*

*getPriority() method returns the thread’s priority.*