## HashMap vs Hashtable in Java

1.The HashMap class is roughly equivalent to Hashtable, except that it is non synchronized and permits nulls. (HashMap allows null as key and value whereas Hashtable doesn't allow nulls).

2. One of the major differences between HashMap and Hashtable is that HashMap is non synchronized whereas Hashtable is synchronized.

3. Another significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the enumerator for the Hashtable is not and throw ConcurrentModificationException if any other Thread modifies the map structurally by adding or removing any element except Iterator's own remove() method. But this is not a guaranteed behavior and will be done by JVM on best effort.

4. One more notable difference between Hashtable and HashMap is that because of thread-safety and synchronization Hashtable is much slower than HashMap if used in Single threaded environment. So if you don't need synchronization and HashMap is only used by one thread, it out perform Hashtable in Java.

5. HashMap does not guarantee that the order of the map will remain constant over time.

Some Important Terms

1)Synchronized means only one Thread can modify a hash table at one point of time. Basically, it means that any thread before performing an update on a Hashtable will have to acquire a lock on the object while others will wait for lock to be released.

2)Fail-safe is relevant from the context of iterators. If an Iterator or ListIterator has been created on a collection object and some other thread tries to modify the collection object "structurally", a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally". However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

3)Structurally modification means deleting or inserting element which could effectively change the structure of map.

## What is the difference between Synchronized Collection classes and Concurrent Collection Classes ?

The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes provide a basic conditionally thread-safe implementation of Map and List.

However, several factors make them unsuitable for use in highly concurrent applications -- their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.

The ConcurrentHashMap and CopyOnWriteArrayList implementations provide much higher concurrency while preserving thread safety, with some minor compromises in their promises to callers.

Since ConcurrentHashMap indroduced concept of segmentation , how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.

## What is the problem while using "==" in autoboxing?

Some of the JVM cache objects of some wrapper class e.g. Integer from -128 to 127 and return same object which if compare via “ ==” can return true but after this range this validity doesn’t work and to make it worse this behavior is JVM dependented so better avoid this kind of check and use equals() method.

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

* Byte data type is an 8-bit signed two's complement integer.
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
* Example: byte a = 100 , byte b = -50

short:

* Short data type is a 16-bit signed two's complement integer.
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
* Default value is 0.
* Example: short s = 10000, short r = -20000

int:

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648.(-2^31)
* Maximum value is 2,147,483,647(inclusive).(2^31 -1)
* Int is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0.
* Example: int a = 100000, int b = -200000

long:

* Long data type is a 64-bit signed two's complement integer.
* Minimum value is -9,223,372,036,854,775,808.(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
* This type is used when a wider range than int is needed.
* Default value is 0L.
* Example: long a = 100000L, int b = -200000L

float:

* Float data type is a single-precision 32-bit IEEE 754 floating point.
* Float is mainly used to save memory in large arrays of floating point numbers.
* Default value is 0.0f.
* Float data type is never used for precise values such as currency.
* Example: float f1 = 234.5f

double:

* double data type is a double-precision 64-bit IEEE 754 floating point.
* This data type is generally used as the default data type for decimal values, generally the default choice.
* Double data type should never be used for precise values such as currency.
* Default value is 0.0d.
* Example: double d1 = 123.4

boolean:

* boolean data type represents one bit of information.
* There are only two possible values: true and false.
* This data type is used for simple flags that track true/false conditions.
* Default value is false.
* Example: boolean one = true

char:

* char data type is a single 16-bit Unicode character.
* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).
* Char data type is used to store any character.
* Example: char letterA ='A'

## How do you find length of a Singly Linked list

**Iterative Solutions**

public int length(){

int count=0;

Node current = this.head;

while(current != null){

count++;

current=current.next()

}

return count;

}

**Recursive Solution:**

public int length(Node current){

if(current == null) //base case

return 0;

return 1+length(current.next());

}

## Why String is immutable or final in Java

1. Imagine StringPool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables , so if any one of them change the value others will be automatically gets affected
2. String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string. In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization.
3. Since String is immutable it can safely shared between many threads ,which is very important for multithreaded programming and to avoid any synchronization issues in Java, Immutability also makes String instance thread-safe in Java.
4. Another reason of Why String is immutable in Java is to allow String to cache its hashcode , being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in hashmap in Java.
5. The absolutely most important reason that String is immutable is that it is used by the class loading mechanism, and thus have profound and fundamental security aspects. Had String been mutable, a request to load "java.io.Writer" could have been changed to load "mil.vogoon.DiskErasingWriter"

## How HashMap works in Java

HashMap works on principle of hashing, we have put() and get() method for storing and retrieving object form HashMap .When we pass an both key and value to put() method to store on HashMap , it uses key object hashcode() method to calculate hashcode and they by applying hashing on that hashcode it identifies bucket location for storing value object. While retrieving it uses key object equals method to find out correct key value pair and return value object associated with that key. HashMap uses linked list in case of collision and object will be stored in next node of linked list. Also HashMap stores both key and value tuple in every node of linked list in form of Map.Entry object.

**Hashing ->** storing 100 balls in 10 buckets. Whenever there is an collision in bucket for two differnet hashcode, equals() method will come in to picture to form the link list at bucket location.

## What is difference between Executor.submit() and Executer.execute() method

There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with **execute** this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with **submit** any thrown exception, checked exception or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will re-throw this exception, wrapped in an ExecutionException.

## Atomicity, Visibility and Ordering

*Atomicity* deals with which actions and sets of actions have indivisible effects. This is the aspect of concurrency most familiar to programmers: it is usually thought of in terms of mutual exclusion.

*Visibility* determines when the effects of one thread can be seen by another.

*Ordering* determines when actions in one thread can be seen to occur out of order with respect to another.

If an action is (or a set of actions are) *atomic*, its result must be seen to happen ``all at once'', or indivisibly. Atomicity is the traditional bugbear of concurrent programming.

 if an action in one thread is *visible* to another thread, then the result of that action can be observed by the second thread. In order to guarantee that the results of one action are observable to a second action, then you have to use some form of synchronization to make sure that the second thread sees what the first thread did.

*Ordering* constraints describe what order things are seen to occur. You only get intuitive ordering constraints by synchronizing correctly.

The assignments to a and b in threadOne() can be seen to be performed out of order. Compilers have a lot of freedom to reorder code in the absence of synchronization; they could either reorder the writes in threadOne or the reads in threadTwo freely.

boolean threadTwo() {  
 boolean r1 = b; // sees true  
 boolean r2 = a; // sees false  
 boolean r3 = a; // sees true  
 return (r1 && !r2) && r3; // returns true  
 }

**Synchronize your code carefully!** In this case, you can throw a lock around threadOne or threadTwo, or you can declare them both to be volatile, and get the ordering you want.

## Reason Why Wait , Notify and NotifyAll are in Object Class

1. Wait and notify is not just normal methods or synchronization utility, more than that they are communication mechanism between two threads in Java.   
   And Object class is correct place to make them available for every object if this mechanism is not available via any java keyword like synchronized.
2. Locks are made available on per Object basis, which is another reason wait and notify is declared in Object class rather then Thread class.
3. In Java in order to enter critical section of code, Threads needs lock and they wait for lock, they don't know which threads holds lock instead they just know the lock is hold by some thread and they should wait for lock instead of knowing which thread is inside the synchronized block and asking them to release lock.

Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:  
- a Thread  
- a monitor (any object)  
  
In the Java design, the thread cannot be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an "intrusion", psosing difficulties on designing/programming concurrent programs. Remember that in Java all operations that are intrusive in another thread's execution are deprecated   
  
Difference between ConcurrentHashMap and Collections.synchronizedMap and Hashtable in Java

The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes, Collections.synchronizedMap and Collections.synchronizedList, provide a basic conditionally thread-safe implementation of Map and List. However, several factors make them unsuitable for use in highly concurrent applications  for example their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.

Hashtable and ConcurrentHashMap , both can be used in multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for longer duration.  
ConcurrentHashMap introduced concept of segmentation , how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.  
  
ConcurrentHashMap do not allow null keys or null values while HashMap allows null keys

Collections.synchronizedMap is able to use a different monitor than itself.

Using synchronized methods is the same as using sychnchonized(this)-blocks, which means, the wrapper would be the monitor and could be locked from the outside of the wrapper.

If you doesn't want an outside application to lock your monitor, you need to hide it.

On the other side, if you want to call multiple methods in a thread safe fashion, it is the easiest way to lock the whole collection (but it's not very scaleable, indeed).

Collections.synchronizedMap(Map) synchronizes all operations (get, put, size, etc).

The main difference between these two is that ConcurrentHashMap will lock only portion of the data which are being updated while other portion of data can be accessed by other threads. However, Collections.synchronizedMap() will lock all the data while updating, other threads can only access the data when the lock is released. If there are many update operations and relative small amount of read operations, you should choose ConcurrentHashMap.

Also one other difference is that ConcurrentHashMap will not preserve the order of elements in the Map passed in. It is similar to HashMap when storing data. There is no guarantee that the element order is preserved. While Collections.synchronizedMap(0 will preserve the elements order of the Map passed in. For example, if you pass a TreeMap to ConcurrentHashMap, the elements order in the ConcurrentHashMap may not be the same as the order in the TreeMap, but Collections.synchronizedMap() will preserve the order.

## What is the difference between Serializable and Externalizable interface in Java?

This is most frequently asked question in Java serialization interview. Here is my version Externalizable provides us writeExternal() and readExternal() method which gives us flexibility to control java serialization mechanism instead of relying on Java's default serialization. Correct implementation of Externalizable interface can improve performance of application drastically.

While working with Externalizable we should remember that the default constructer should be public else the code will throw exception. Please follow the below code:

public class MyExternalizable implements Externalizable

{

private String userName;

private String passWord;

private Integer roll;

public MyExternalizable()

{

}

public MyExternalizable(String userName, String passWord, Integer roll)

{

this.userName = userName;

this.passWord = passWord;

this.roll = roll;

}

@Override

public void writeExternal(ObjectOutput oo) throws IOException

{

oo.writeObject(userName);

oo.writeObject(roll);

}

@Override

public void readExternal(ObjectInput oi) throws IOException, ClassNotFoundException

{

userName = (String)oi.readObject();

roll = (Integer)oi.readObject();

}

public String toString()

{

StringBuilder b = new StringBuilder();

b.append("userName: ");

b.append(userName);

b.append(" passWord: ");

b.append(passWord);

b.append(" roll: ");

b.append(roll);

return b.toString();

}

public static void main(String[] args)

{

try

{

MyExternalizable m = new MyExternalizable("nikki", "student001", 20);

System.out.println(m.toString());

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("/Users/Desktop/files/temp1.txt"));

oos.writeObject(m);

oos.close();

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

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("/Users/Desktop/files/temp1.txt"));

MyExternalizable mm = (MyExternalizable)ois.readObject();

System.out.println(mm.toString());

}

catch (ClassNotFoundException ex)

{

Logger.getLogger(MyExternalizable.class.getName()).log(Level.SEVERE, null, ex);

}

catch(IOException ex)

{

Logger.getLogger(MyExternalizable.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

Externalizable interface extends Serializable interface.

Serialization is a recursive algorithm. What I mean to say here is, apart from the fields that are required, starting from a single object, until all the objects that can be reached from that object by following instance variables, are also serialized. This includes the super class of the object until it reaches the "Object" class and the same way the super class of the instance variables until it reaches the "Object" class of those variables.

Both serializing and deserializing require the serialization mechanism to discover information about the instance it is serializing. Using the default serialization mechanism, will use reflection to discover all the field values. Also the information about class description is added to the stream which includes the descption of all the serializable superclasses, the description of the class and the instance data associated with the specific instance of the class.

You know that serialization needs serialVersionUID, a unique Id to identify the information persisted. If you dont explicitly set a serialiVersionUID, serialization will compute the serialiVersionUID by going through all the fields and methods.

When an object that implements Serializable interface, is serialized or de-serialized, no constructor of the object is called and hence any initialization which is done in the constructor cannot be done.

Externalization is nothing but serialization but by implementing Externalizable interface to persist and restore the object. To externalize your object, you need to implement Externalizable interface that extends Serializable interface. Here only the identity of the class is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances which means you will have complete control of what to serialize and what not to serialize.**But to externalize an object, you need a default public constructor.**  
  
 Externalizable interface is not a marker interface and it provides two methods - writeExternal and readExternal. These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of writeObject and readObject methods.

**How serialization happens**? JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal method. If the object does not support Externalizable but implement Serializable, then the object is saved using ObjectOutputStream. Now when an Externalizable object is reconstructed, an instance is created first using the public no-arg constructor, then the readExternal method is called. Again if the object does not support Externalizable, then Serializable objects are restored by reading them from an ObjectInputStream.

Externalization efficiency comes at a price. The default serialization mechanism adapts to application changes due to the fact that metadata is automatically extracted from the class definitions

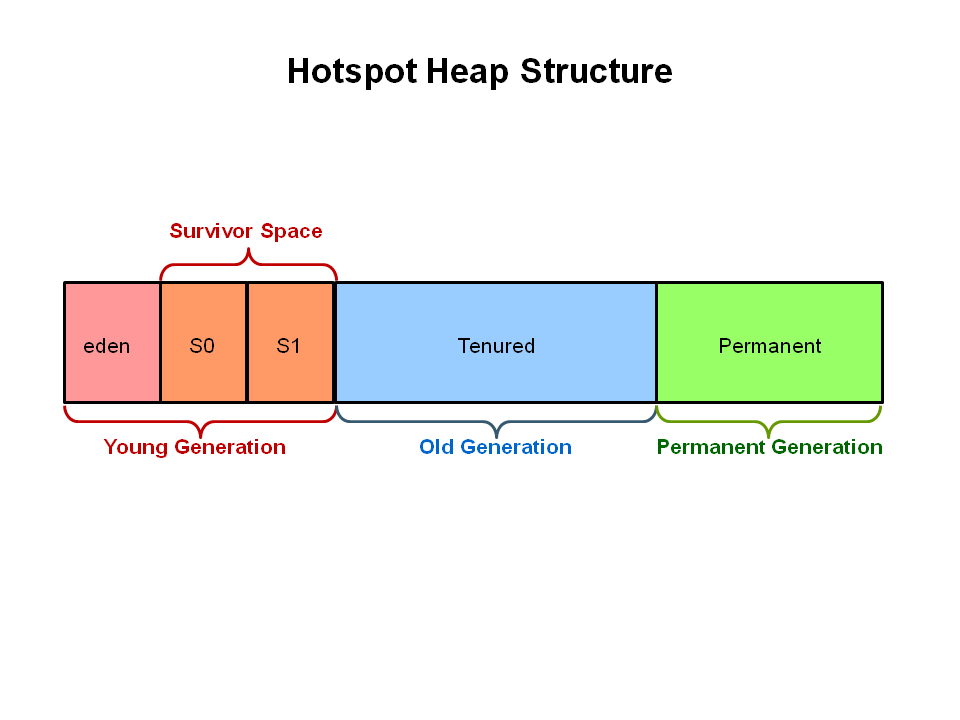
Externalization on the other hand isn't very flexible and requires you to rewrite your marshalling and demarshalling code whenever you change your class definitions.

As you know a default public no-arg constructor will be called when serializing the objects that implements Externalizable interface. Hence, Externalizable interface can't be implemented by Inner Classes in Java

Methods in externalizable interface are public. So any malicious program can invoke which results into losing the prior serialized state.

This makes it easy to evolve your application in response to actual performance data and shifting requirements. You can do the following thing:   
\* Make all your classes implement Serializable.  
\* Then make some of them, the ones you send often and for which serialization is inefficient, implement Externalizable instead.

## How Garbage Collection works in Java



 The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation

The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

**Stop the World Event** - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

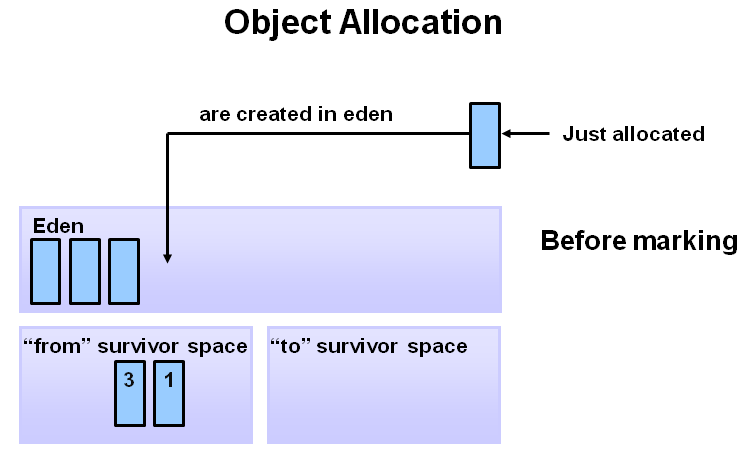
The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

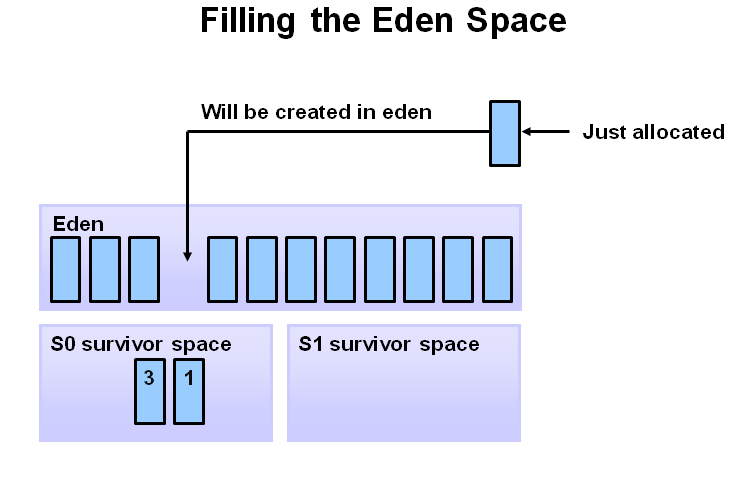
| **Switch** | **Description** |
| --- | --- |
| -Xms | Sets the initial heap size for when the JVM starts. |
| -Xmx | Sets the maximum heap size. |
| -Xmn | Sets the size of the Young Generation. |
| -XX:PermSize | Sets the starting size of the Permanent Generation. |
| -XX:MaxPermSize | Sets the maximum size of the Permanent Generation |

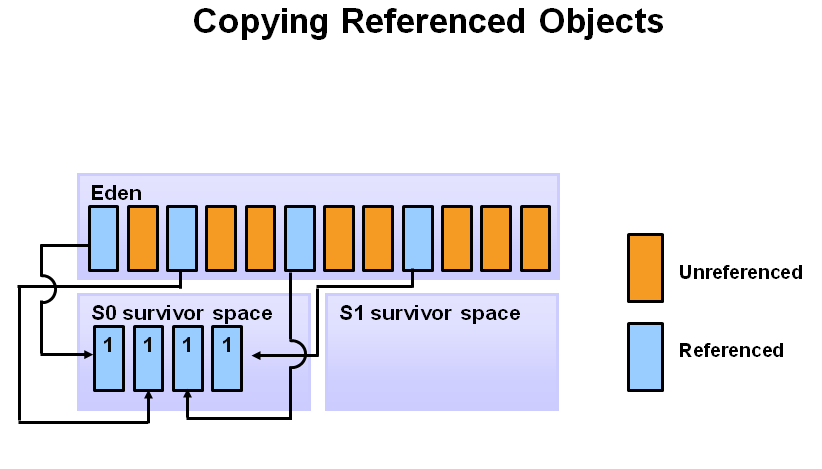
Now that you understand why the heap is separted into different generations, it is time to look at how exactly these spaces interact. The pictures that follow walks through the object allocation and aging process in the JVM.

First, any new objects are allocated to the eden space. Both survivor spaces start out empty.

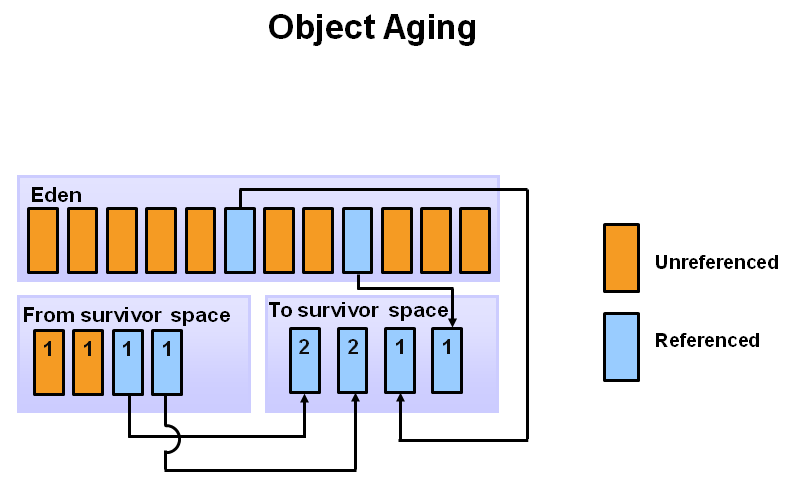


When the eden space fills up, a minor garbage collection is triggered.

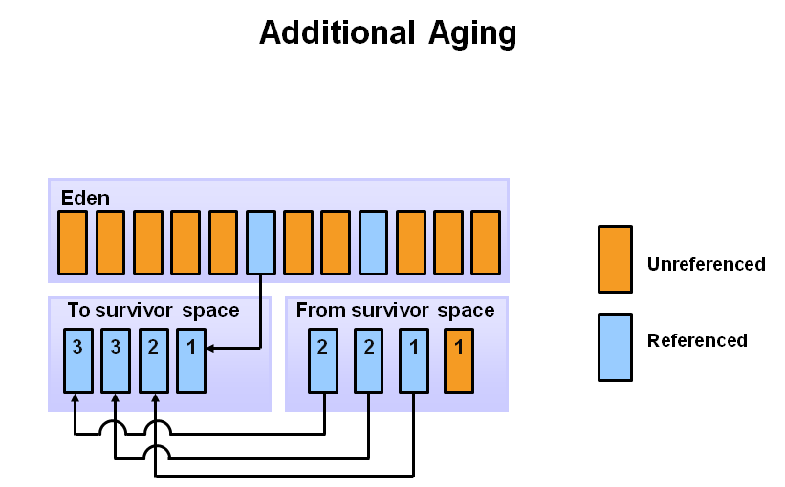


Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.

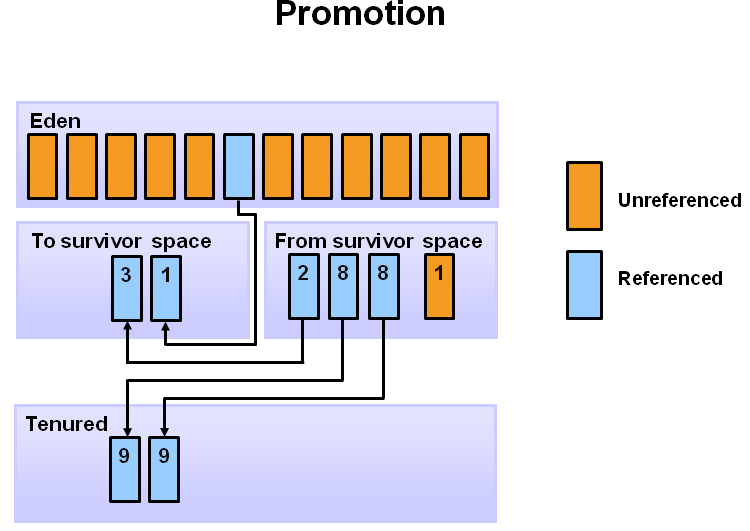
At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.



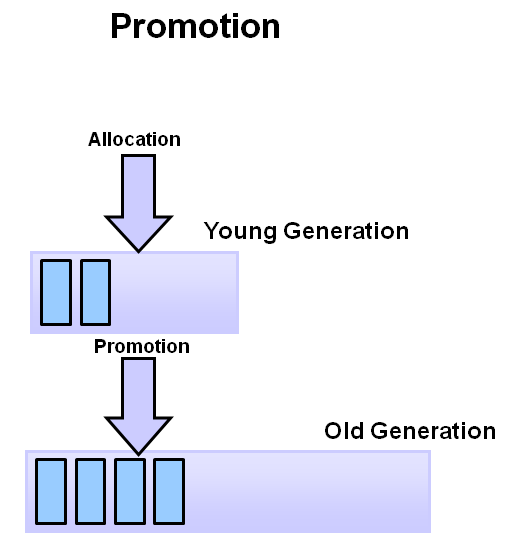
At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



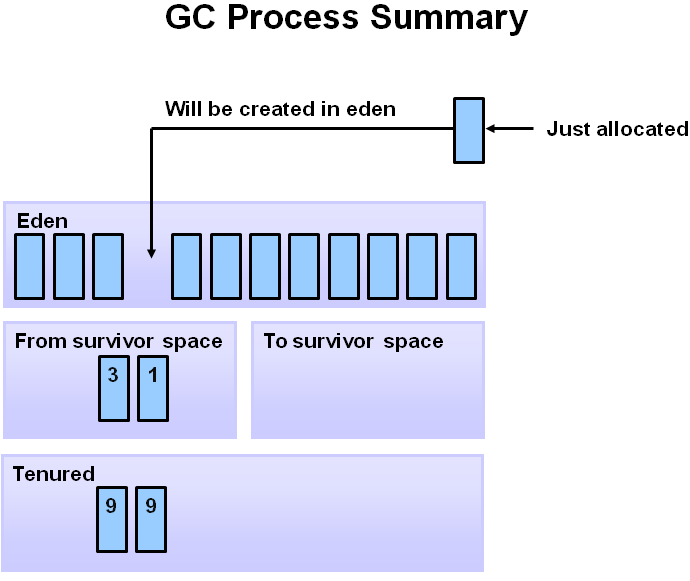
This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



As minor GCs continue to occur, objects will continue to be promoted to the old generation space.

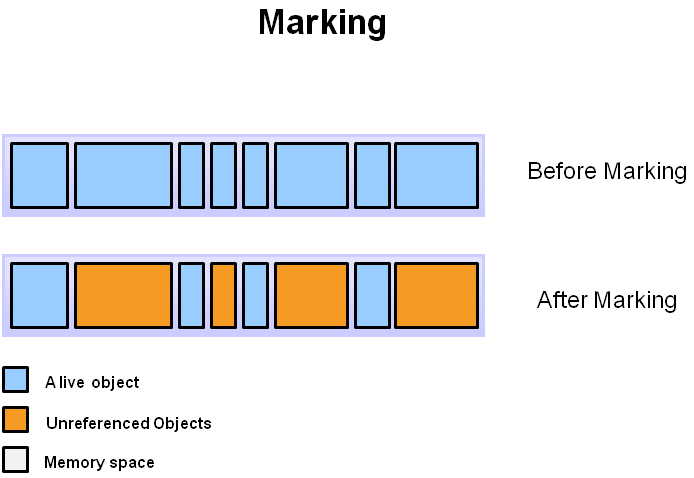


So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.



#### Step 1: Marking

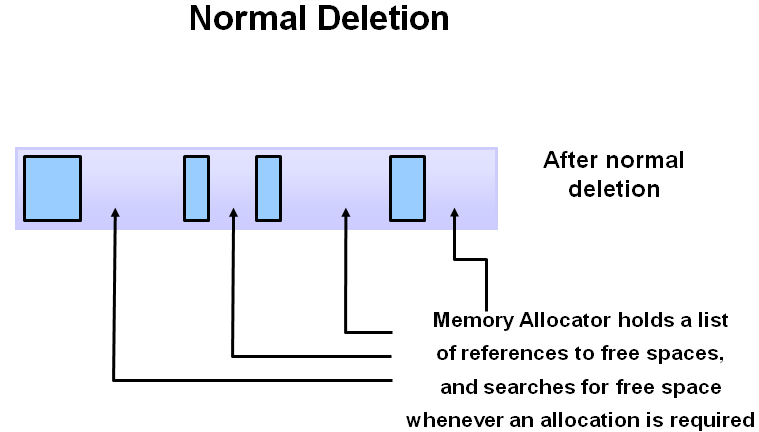
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.



Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.

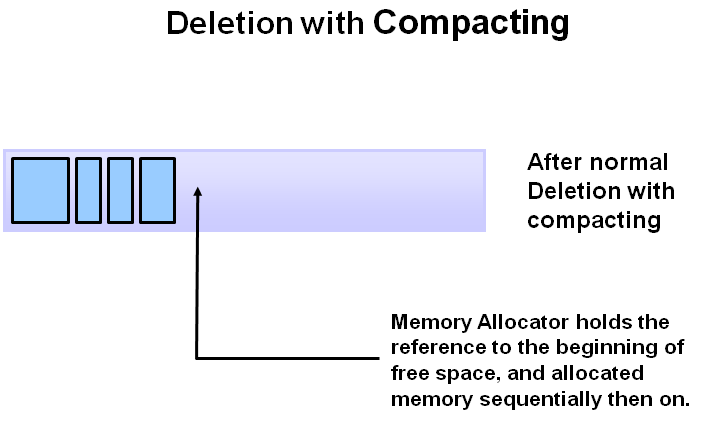
#### Step 2: Normal Deletion

Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.



The memory allocator holds references to blocks of free space where new object can be allocated.

#### Step 2a: Deletion with Compacting

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

An Object becomes eligible for Garbage collection or GC if its not reachable from any live threads or any static references in other words you can say that an object becomes eligible for garbage collection if its all references are null. Cyclic dependencies are not counted as reference so if Object A has reference of object B and object B has reference of Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

## What is immutable object? Can you write immutable object?

Immutable classes are those class, whose [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) can not be modified once created, it means any modification on immutable object will result in another immutable object.   
Despite of few disadvantages, Immutable object still offers several benefits in multi-threaded programming and it’s a great choice to achieve thread safety in Java code. here are few rules, which helps to make a class immutable in Java :

1. State of immutable object can not be modified after construction, any modification should result in new immutable object.

2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. Object should be final in order to restrict sub-class for altering immutability of parent class.

By the way, you can still create immutable object by violating few rules, like String has its [hashcode](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) in non final field, but its always guaranteed to be same.   
  
public final class Contacts {

    private final String name;

    private final String mobile;

    public Contacts(String name, String mobile) {

        this.name = name;

        this.mobile = mobile;

    }

    public String getName(){

        return name;

    }

    public String getMobile(){

        return mobile;

    }

}

Some time you may need to write immutable class which includes mutable classes like java.util.Date, despite storing Date into final field it can be modified internally, if internal date is returned to the client. In order to preserve immutability in such cases, its advised to return copy of original object

 public Date getRemindingDate() {

        return (Date) remindingDate.clone();

    }

## Benefits of Immutable Classes in Java

As I said earlier Immutable classes offers several benefits, here are few to mention:

1) Immutable objects are by default [thread safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), can be shared without synchronization in concurrent environment.

2) Immutable object simplifies development, because it’s easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing [synchronization](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) in code.  
  
4) Another important benefit of Immutable objects is **reusability**, you can cache Immutable object and reuse them, much like String literals and Integers.

as stated above you can achieve same functionality by making member as non-final but private and not modifying them except in constructor. Don't provide setter method for them and if it is a mutable object, then don't ever leak any reference for that member.

## What is the difference between creating String as new() and literal?

When we create string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in [PermGen area of heap](http://javarevisited.blogspot.sg/2012/01/tomcat-javalangoutofmemoryerror-permgen.html).

String str = new String("Test");  
   
does not  put the object str in String pool , we need to call String.intern() method which is used to put  them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool. **By the way there is a catch here, Since we are passing arguments as "Test", which is a String literal, it will also create another object as "Test"on string pool.**

## How does substring () inside String works

The [**Java language specification**](http://java.sun.com/docs/books/jls/)defines precisely the API of class String, its Unicode details, its immutability, and the way String literals should be handled. Naturally, it does not define how Strings should be  represented internally, and what are the required time/space complexities of the different operations on them.

Sun’s implementation (and also IBM’s) represent a string with a combination of 3 data members:

* value – A reference to a char array
* offset – A start index, inside the array
* count – The actual length of the string

When allocating a new string, the char array contains exactly the string content. Offset is set to 0, and count is set to the char array length. Then, when calling the method substring(..) upon it, the new string being returned contains a reference to the same char array, but its offset and count members are modified, reflecting the requested subsequence of chars. The sharing of the same char array by multiple String instances is possible, since strings are immutable. There are two benefits of this implementation approach in comparison to a substring implementation based on copying:  
1) Memory usage is usually reduced, specially in cases where many substrings of the same string are taken, or if the substrings are long  
2) substring(..) runs in constant time, instead of linear time

 Obviously there is a tradeoff here – if the string utilization pattern is not as described in (1), we may suffer from excessive memory consumption. The code above demonstrates this edge case: we take very small substrings of very large strings, where the latter are temporary. The temporary strings (referenced by randomStr) ARE collected by the garbage collector during the loop. However, their internal char arrays can not be collected, since they are being shared with the sub-strings. Therefore, the very long char arrays are kept unintentionally in memory, and that constitutes a memory leak.

The solution is simple – forcing the substrings to be copied into a new compact char array. This is done by replacing line 22 above with:

|  |  |
| --- | --- |
| 1 | String subStr = new String(randomStr.substring(1000,1004)); |

This is improved in JDK 7. In JDK 7, the substring() method actually create a new array in the heap.

*//JDK 7*

**public** String(**char** value[], **int** offset, **int** count) {

*//check boundary*

**this**.value = Arrays.copyOfRange(value, offset, offset + count);

}

**public** String substring(**int** beginIndex, **int** endIndex) {

*//check boundary*

**int** subLen = endIndex - beginIndex;

**return** **new** String(value, beginIndex, subLen);

}

## What are differences between wait and sleep method in java?

1) wait is called from synchronized context only while sleep can be called without synchronized block

2) wait is called on Object while sleep is called on Thread.

3) waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awaken by calling notify method.

4) wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.

5) wait release lock on object while waiting while sleep doesn’t release lock while waiting.

## Difference between yield and sleep in java

Major difference between yield and sleep in Java is that yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn’t guarantee  that current thread will pause or stop but it guarantee that CPU will be relinquish by current Thread as a result of call to Thread.yield() method in java.

## Volatile

**Volatile keyword in Java** is used as an indicator to Java compiler and  [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable.   
  
Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made by one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread. Java volatile keyword cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees visibility and ordering , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from synchronization barrier.

**When to use Volatile variable in Java**

1. You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. long and double both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) data type and by default writing of long and double is not atomic and platform dependence.
2. Volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. with volatile variable its guaranteed that all reader thread will see updated value of volatile variable once write operation  completed, without volatile keyword different reader thread may see different values.
3. volatile variable can be used to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment
4. Another place where volatile variable can be used is to fixing double checked locking in Singleton pattern
5. **Java volatile keyword doesn't means atomic,**

## What is race condition?

Race condition in Java is a type of concurrency bug or issue which is introduced in your program because  parallel execution of your program by multiple threads at same time, Since Java is a multi-threaded programming language hence risk of Race condition is higher in Java which demands clear understanding of what causes a race condition and how to avoid that.

Race conditions occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other.

Classical example of Race condition is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. if two [threads](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread.

## wait notify and notifyAll

We use wait () and notify () or notifyAll () method mostly for inter-thread communication. One thread is waiting after checking a condition e.g. In Producer Consumer example Producer Thread is waiting if buffer is full and Consumer thread notify Producer thread after he creates a space in buffer by consuming an element. calling notify() or notifyAll() issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block , all the threads which are waiting fight for object lock on which they are waiting and lucky thread returns from wait() method after reacquiring the lock and proceed further.  
  
This race condition is resolved by using [synchronized keyword and locking provided by java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html). In order to call the wait (), notify () or notifyAll () methods in Java, we must have obtained the lock for the object on which we're calling the method. Since the wait () method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the wait () method, we must use this lock to ensure that checking the condition (buffer is full or not) and setting the condition (taking element from buffer) is atomic which can be achieved by using synchronized method or block in Java.

**IllegalMonitorStateException in Java** which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.

## Processes

## A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

The **join** method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

*Memory consistency errors* occur when different threads have inconsistent views of what should be the same data.  To see this, consider the following example. Suppose a simple int field is defined and initialized:

int counter = 0;

The counter field is shared between two threads, A and B. Suppose thread A increments counter:

counter++;

Then, shortly afterwards, thread B prints out counter:

System.out.println(counter);

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

A concurrent application's ability to execute in a timely manner is known as its ***liveness***.

*Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. T

When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke bowBack. Neither block will ever end, because each thread is waiting for the other to exit bow.

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

## Starvation

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other.

**JSR 166: Concurrency Utilities – specification for java memory model**

## Java Concurrency

The [Java Concurrency Utilities framework](http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/) is a library of [types](http://www.javaworld.com/jw-01-2001/jw-0119-type.html) that are designed to be used as building blocks for creating concurrent classes or applications. These types are thread-safe, have been thoroughly tested, and offer high performance.

Types in the Java Concurrency Utilities are organized into small frameworks; namely, Executor framework, synchronizer, concurrent collections, locks, atomic variables, and Fork/Join. They are further organized into a main package and a pair of subpackages:

**java.util.concurrent** contains high-level utility types that are commonly used in concurrent programming. Examples include semaphores, barriers, thread pools, and concurrent hashmaps.

* The **java.util.concurrent.atomic** subpackage contains low-level utility classes that support lock-free thread-safe programming on single variables.
* The **java.util.concurrent.locks** subpackage contains low-level utility types for locking and waiting for conditions, which are different from using Java's low-level synchronization and monitors.

## The Executor framework

java.util.concurrent includes the [Executor framework](http://www.javaworld.com/javaworld/jw-09-2006/jw-0904-threads.html), a small framework of types that decouple task submission from task-execution policies. Using the Executor framework, it is possible to easily tune a program's task-execution policy without having to significantly rewrite your code.

The Executor framework is based on the Executor interface, which describes an executor as any object capable of executing java.lang.Runnable tasks. This interface declares the following solitary method for executing a Runnable task:

*void execute(Runnable command)*

You submit a Runnable task by passing it to execute(Runnable). If the executor cannot execute the task for any reason (for instance, if the executor has been shut down), this method will throw a RejectedExecutionException.

The key concept is that task submission is decoupled from the task-execution policy, which is described by an Executor implementation. The runnable task is thus able to execute via a new thread, a pooled thread, the calling thread, and so on.

 Executor is very limited. For example, you can't shut down an executor or determine whether an asynchronous task has finished. You also can't cancel a running task. For these and other reasons, the Executor framework provides an [ExecutorService](http://www.javaworld.com/javaworld/jw-10-2011/111004-jtip-recursion-in%20-java-7.html) interface, which extends Executor.

Five of ExecutorService's methods are especially noteworthy:

* **boolean awaitTermination(long timeout, TimeUnit unit)** blocks the calling thread until all tasks have completed execution after a shutdown request, the timeout occurs, or the current thread is interrupted, whichever happens first. The maximum time to wait is specified by timeout, and this value is expressed in the unit units specified by the TimeUnit enum; for example, TimeUnit.SECONDS. This method throws java.lang.InterruptedException when the current thread is interrupted. It returns true when the executor is terminated and false when the timeout elapses before termination.
* **boolean isShutdown()** returns true when the executor has been shut down.
* **void shutdown()** initiates an orderly shutdown in which previously submitted tasks are executed but no new tasks are accepted.
* **<T> Future<T> submit(Callable<T> task)** submits a value-returning task for execution and returns a Future representing the pending results of the task.
* **Future<?> submit(Runnable task)** submits a Runnable task for execution and returns a Future representing that task.

The Future<V> interface represents the result of an asynchronous computation. The result is known as a future because it typically will not be available until some moment in the future. You can invoke methods to cancel a task, return a task's result (waiting indefinitely or for a timeout to elapse when the task hasn't finished), and determine if a task has been cancelled or has finished.

The Callable<V> interface is similar to the Runnable interface in that it provides a single method describing a task to execute. Unlike Runnable's void run() method, Callable<V>'s V call() throws Exception method can return a value and throw an exception.

### Executor factory methods

* **ExecutorService newCachedThreadPool()** creates a thread pool that creates new threads as needed, but which reuses previously constructed threads when they're available. Threads that haven't been used for 60 seconds are terminated and removed from the cache. This thread pool typically improves the performance of programs that execute many short-lived asynchronous tasks.
* **ExecutorService newSingleThreadExecutor()** creates an executor that uses a single worker thread operating off an unbounded queue -- tasks are added to the queue and execute sequentially (no more than one task is active at any one time). If this thread terminates through failure during execution before shutdown of the executor, a new thread will be created to take its place when subsequent tasks need to be executed.
* **ExecutorService newFixedThreadPool(int nThreads)** creates a thread pool that re-uses a fixed number of threads operating off a shared unbounded queue. At most **n**Threads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue until a thread is available. If any thread terminates through failure during execution before shutdown, a new thread will be created to take its place when subsequent tasks need to be executed. The pool's threads exist until the executor is shut down.

The Executor framework offers additional types (such as theScheduledExecutorService interface), but the types you are likely to work with most often are ExecutorService, Future,Callable, and Executors.

**import** java.io.**IOException**;

**import** java.net.**ServerSocket**;

**import** java.net.**Socket**;

**import** java.util.concurrent.**Executor**;

**import** java.util.concurrent.**Executors**;

**class** **Server**

{

**static** **Executor** pool = **Executors**.newFixedThreadPool(5);

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

{

**ServerSocket** socket = **new** **ServerSocket**(9000);

**while** (**true**) {

**final** **Socket** s = socket.accept();

**Runnable** r = **new** **Runnable**()

{

@Override

**public** **void** run()

{ doWork(s);

}

};

pool.execute(r);

}

}

**static** **void** doWork(**Socket** s) {

}

}

newFixedThreadPool(int) to obtain a thread pool-based executor that reuses five threads. It also replaces new Thread(r).start(); with pool.execute(r);for executing runnable tasks via any of these threads.

**ExecutorService** executor = **Executors**.newSingleThreadExecutor();

**Callable**<**List**<**String**>> callable;

callable = **new** **Callable**<**List**<**String**>>()

{

@Override **public** **List**<**String**> call()

**throws** **IOException**, **MalformedURLException**

{

**List**<**String**> lines = **new** **ArrayList**<>();

URL url = **new** URL(args[0]);

**HttpURLConnection** con;

con = (**HttpURLConnection**) url.openConnection();

**InputStreamReader** isr;

isr = **new** **InputStreamReader**(con.getInputStream());

**BufferedReader** br;

br = **new** **BufferedReader**(isr);

**String** line;

**while** ((line = br.readLine()) != **null**)

lines.add(line);

**return** lines;

}

};

**Future**<**List**<**String**>> future = executor.submit(callable);

**try**

{

**List**<**String**> lines = future.**get**(5, **TimeUnit**.SECONDS);

**for** (**String** line: lines)

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

}

**catch** (**ExecutionException** ee)

{

**System**.err.println("Callable through exception: "+ee.getMessage());

}

**catch** (**InterruptedException** | **TimeoutException** eite)

{

**System**.err.println("URL not responding");

}

executor.shutdown();

 main() method first verifies that a single (URL-based) command-line argument has been specified. It then creates a single-thread executor and a callable that tries to open a connection to this URL, read its contents line by line, and save these lines in a list, which it returns.

The callable is subsequently submitted to the executor and a future representing the list of strings is returned. main()invokes the future's V get(long timeout, TimeUnit unit)method to obtain this list.

get() throws TimeoutException when the callable doesn't finish within five seconds. It throws ExecutionException when the callable throws an exception (for instance, the callable will throw java.net.MalformedURLException when the URL is invalid). Regardless of whether an exception is thrown or not, the executor must be shut down before the application exits. If the executor isn't shut down, the application won't exit because the non-daemon thread-pool threads are still executing.

*Synchronizers* are high-level constructs that coordinate and control thread execution. The Java Concurrency Utilities framework provides classes that implement semaphore, cyclic barrier, countdown latch, exchanger, and phaser synchronizers.

## Semaphores

A semaphore is a thread-synchronization construct for controlling thread access to a common resource. It's often implemented as a protected variable whose value is incremented by an acquire operation and decremented by a release operation.

The acquire operation either returns control to the invoking thread immediately or causes that thread to block when the semaphore's current value reaches a certain limit. The release operation decreases the current value, which causes a blocked thread to resume.

Semaphores whose current values can be incremented past 1 are known as [counting semaphores](http://www.javaworld.com/jw-12-1998/jw-12-toolbox.html), whereas semaphores whose current values can be only 0 or 1 are known as binary semaphores or [mutexes](http://www.javaworld.com/jw-11-1998/jw-11-toolbox.html). In either case, the current value cannot be negative.

The java.lang.concurrent.Semaphore class conceptualizes a semaphore as an object maintaining a set of permits. This class provides Semaphore(int permits) and Semaphore(int permits, boolean fair) constructors for specifying the number of permits. Each call to the Semaphore's void acquire() method takes one of the available permits or blocks the calling thread when one isn't available. Each call to Semaphore's void release() method returns an available permit, potentially releasing a blocking acquirer thread.

**final** **class** **Pool**

{

**public** **static** **final** **int** MAX\_AVAILABLE = 10;

**private** **Semaphore** available = **new** **Semaphore**(MAX\_AVAILABLE, **true**);

**private** **String**[] items;

**private** **boolean**[] used = **new** **boolean**[MAX\_AVAILABLE];

**Pool**()

{

items = **new** **String**[MAX\_AVAILABLE];

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

items[i] = "ITEM"+i;

}

**String** getItem() **throws** **InterruptedException**

{

available.acquire();

**return** getNextAvailableItem();

}

**void** putItem(**String** item)

{

**if** (markAsUnused(item))

available.release();

}

**private** **synchronized** **String** getNextAvailableItem()

{

**for** (**int** i = 0; i < MAX\_AVAILABLE; ++i)

{

**if** (!used[i])

{

used[i] = **true**;

**return** items[i];

}

}

**return** **null**; *// not reached*

}

**private** **synchronized** **boolean** markAsUnused(**String** item)

{

**for** (**int** i = 0; i < MAX\_AVAILABLE; ++i)

{

**if** (item == items[i])

{

**if** (used[i])

{

used[i] = **false**;

**return** **true**;

}

**else**

**return** **false**;

}

}

**return** **false**;

}}

**public** **class** SemaphoreDemo

{

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

{

**final** Pool pool = **new** Pool();

Runnable r = **new** Runnable()

{

@Override

**public** **void** run()

{

String name = Thread.*currentThread*().getName();

**try**

{

**while** (**true**)

{

String item;

System.*out*.printf("%s acquiring %s%n", name,

item = pool.getItem());

Thread.*sleep*(200+(**int**)(Math.*random*()\*100));

System.*out*.printf("%s putting back %s%n",

name,

item);

pool.putItem(item);

}

}

**catch** (InterruptedException ie)

{

System.*out*.printf("%s interrupted%n", name);

}

}

};

ExecutorService[] executors = **new**

ExecutorService[Pool.MAX\_AVAILABLE+1];

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

{

executors[i] = Executors.newSingleThreadExecutor();

executors[i].execute(r);

}

}

}

Pool provides String getItem() and void putItem(String item) methods for obtaining and returning resources. Before obtaining an item in getItem(), a thread must acquire a permit from the semaphore, guaranteeing that an item is available for use. When the thread has finished with the item, it calls putItem(String), which returns the item to the pool and then returns a permit to the semaphore, which lets another thread acquire that item.

No synchronization lock is held when acquire() is called because that would prevent an item from being returned to the pool. However, String getNextAvailableItem() and boolean markAsUnused(String item) are synchronized to maintain pool consistency.

## Cyclic barriers

A [cyclic barrier](http://www.javaworld.com/javaworld/jw-03-2009/jw-03-java-concurrency-with-thread-gates.html) is a thread-synchronization construct that lets a set of threads wait for each other to reach a common barrier point. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A cyclic barrier is implemented by the java.lang.concurrent.CyclicBarrier class. This class provides the following constructors:

**CyclicBarrier(int nthreads, Runnable barrierAction)** causes a maximum of nthreads-1 threads to wait at the barrier. When one more thread arrives, it executes the non null barrierAction and then all threads proceed. This action is useful for updating shared state before any of the threads continue.

**CyclicBarrier(int nthreads)** is similar to the previous constructor except that no runnable is executed when the barrier is tripped.

Either constructor throws java.lang.IllegalArgumentException when the value passed to nthreads is less than 1.

CyclicBarrier declares an int await() method that typically causes the calling thread to wait unless the thread is the final thread. If so, and if a nonnull Runnable was passed to barrierAction, the final thread executes the runnable before the other threads continue.

await() throws InterruptedException when the thread that invoked this method is interrupted while waiting. This method throws BrokenBarrierException when another thread was interrupted while the invoking thread was waiting, the barrier was broken when await() was called, or the barrier action (when present) failed because an exception was thrown from the runnable's run() method.

**public** **class** **CyclicBarrierDemo**

{

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

{

**Runnable** action = **new** **Runnable**()

{

@Override

**public** **void** run()

{

**String** name =

**Thread**.currentThread().getName();

**System**.**out**.printf("Thread %s "+

"executing barrier

action.%n", name);

}

};

**final** **CyclicBarrier** barrier = **new** **CyclicBarrier**(3, action);

**Runnable** task = **new** **Runnable**()

{

@Override

**public** **void** run()

{

**String** name = **Thread**.currentThread().getName();

**System**.**out**.printf("%s about to join game...%n",

name);

**try**

{

barrier.await();

}

**catch** (**BrokenBarrierException** bbe)

{

**System**.**out**.println("barrier is broken");

**return**;

}

**catch** (**InterruptedException** ie)

{

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

**return**;

}

**System**.**out**.printf("%s has joined game%n", name);

}

};

**ExecutorService**[] executors = **new** **ExecutorService**[]

{

**Executors**.newSingleThreadExecutor(),

**Executors**.newSingleThreadExecutor(),

**Executors**.newSingleThreadExecutor()

};

**for** (**ExecutorService** executor: executors) {

executor.execute(task);

executor.shutdown();

}

}

}

The above main() method first creates a barrier action that's run by the last thread to reach the barrier. Next, a cyclic barrier is created. When three players arrive it trips and executes the barrier action.

main() now creates a runnable that outputs various status messages and invokes await(), followed by a three-executor array. Each executor runs this runnable and shuts down after the runnable finishes. To reuse a CyclicBarrier instance, invoke its void reset() method.

## Countdown latches

A countdown latch is a thread-synchronization construct that causes one or more threads to wait until a set of operations being performed by other threads finishes. It consists of a count and "cause a thread to wait until the count reaches zero" and "decrement the count" operations.

A thread invokes the void await() method to wait until the count has reached zero (or the thread has been interrupted). Subsequent calls to await() for a zero count return immediately. A thread calls void countDown() to decrement the count

Countdown latches are useful for decomposing a problem into smaller pieces and giving a piece to a separate thread, as follows:

1 A main thread creates a countdown latch with a count of 1 that's used as a "starting gate" to start a group of worker threads simultaneously.

2 Each worker thread waits on the latch and the main thread decrements this latch to let all worker threads proceed.

3 The main thread waits on another countdown latch initialized to the number of worker threads.

4 When a worker thread completes, it decrements this count. After the count reaches zero (meaning that all worker threads have finished), the main thread proceeds and gathers the results.

**import** java.util.concurrent.**CountDownLatch**;

**public** **class** **CountDownLatchDemo**

{

**final** **static** **int** N = 3;

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

{

**CountDownLatch** startSignal = **new** **CountDownLatch**(1);

**CountDownLatch** doneSignal = **new** **CountDownLatch**(N);

**for** (**int** i = 0; i < N; ++i) *// create and start threads*

**new** **Thread**(**new** **Worker**(startSignal, doneSignal)).start();

**System**.**out**.println("about to let threads proceed");

startSignal.countDown(); *// let all threads proceed*

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

**System**.**out**.println("waiting for threads to finish");

doneSignal.await(); *// wait for all threads to finish*

**System**.**out**.println("main thread terminating");

}

}

**class** **Worker** **implements** **Runnable**

{

**private** **final** **static** **int** N = 5;

**private** **final** **CountDownLatch** startSignal;

**private** **final** **CountDownLatch** doneSignal;

**Worker**(**CountDownLatch** startSignal, **CountDownLatch** doneSignal)

{

**this**.startSignal = startSignal;

**this**.doneSignal = doneSignal;

}

@Override

**public** **void** run()

{

**try**

{

**String** name = **Thread**.currentThread().getName();

startSignal.await();

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

{

**System**.**out**.printf("thread %s is working%n", name);

**try**

{

**Thread**.sleep((**int**)(**Math**.random()\*300));

}

**catch** (**InterruptedException** ie)

{

}

}

**System**.**out**.printf("thread %s finishing%n", name);

doneSignal.countDown();

}

**catch** (**InterruptedException** ie)

{

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

}

}

}

## Exchangers

An *exchanger* (also known as a *rendezvous*) is a thread-synchronization construct that lets a pair of threads exchange data items. An exchanger is similar to a cyclic barrier whose count is set to 2 but also supports exchange of data when both threads reach the barrier.

For example, V exchange(V x) throws InterruptedException waits for another thread to arrive at the exchange point (unless the current thread is interrupted) and then transfers the given object to it, receiving its object in return.

**public** **class** **ExchangerDemo**

{

**static** **Exchanger**<**DataBuffer**> exchanger = **new**

**Exchanger**<**DataBuffer**>();

**static** **DataBuffer** initialEmptyBuffer = **new** **DataBuffer**();

**static** **DataBuffer** initialFullBuffer = **new** **DataBuffer**("ITEM");

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

{

**class** **FillingLoop** **implements** **Runnable**

{

**int** count = 0;

@Override

**public** **void** run()

{

**DataBuffer** currentBuffer = initialEmptyBuffer;

**try**

{

**while** (**true**)

{

addToBuffer(currentBuffer);

**if** (currentBuffer.isFull())

{

**System**.**out**.println("filling loop thread wants to

exchange");

currentBuffer = exchanger.exchange(currentBuffer);

**System**.**out**.println("filling loop thread observes an

exchange");

}

}

}

**catch** (**InterruptedException** ie)

{

**System**.**out**.println("filling loop thread interrupted");

}

}

**void** addToBuffer(**DataBuffer** buffer)

{

**String** item = "NEWITEM"+count++;

**System**.**out**.printf("Adding %s%n", item);

buffer.add(item);

}

}

**class** **EmptyingLoop** **implements** **Runnable**

{

@Override

**public** **void** run()

{

**DataBuffer** currentBuffer = initialFullBuffer; **try**

{

**while** (**true**)

{

takeFromBuffer(currentBuffer);

**if** (currentBuffer.isEmpty())

{

**System**.**out**.println("emptying loop thread wants to

exchange");

currentBuffer = exchanger.exchange(currentBuffer);

**System**.**out**.println("emptying loop thread observes an

exchange");

}

}

}

**catch** (**InterruptedException** ie)

{

**System**.**out**.println("emptying loop thread interrupted");

}

}

**void** takeFromBuffer(**DataBuffer** buffer)

{

**System**.**out**.printf("taking %s%n", buffer.remove());

}

}

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

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

}

}

## Phasers

A phaser is a thread-synchronization construct that's similar to a cyclic barrier in that it lets a group of threads wait on a barrier and then proceed after the last thread arrives. It also offers the equivalent of a barrier action. However, a phaser is more flexible.

Unlike a cyclic barrier, which coordinates a fixed number of threads, a phaser can coordinate a variable number of threads, which can register at any time.

The **Phaser(int threads)** constructor creates a phaser that initially coordinates nthreads threads (which have yet to arrive at the phaser barrier) and whose phase number is initially set to 0.

The int **register()** method adds a new unarrived thread to this phaser and returns the phase number to which the arrival applies. This number is known as thearrival phase number.

The int **arriveAndAwaitAdvance()** method records arrival and waits for the phaser to advance (which happens after the other threads have arrived). It returns the phase number to which the arrival applies.

The int **arriveAndDeregister()** method arrives at this phaser and deregisters from it without waiting for others to arrive, reducing the number of threads required to advance in future phases.

**static** **void** runTasks(**List**<**Runnable**> tasks)

{

**final** **Phaser** phaser = **new** **Phaser**(1); *// "1" to register self*

*// create and start threads*

**for** (**final** **Runnable** task: tasks)

{

phaser.**register**();

**new** **Thread**()

{

@Override

**public** **void** run()

{

**try**

{

**Thread**.sleep(50+(**int**)(**Math**.random()\*300));

}

**catch** (**InterruptedException** ie)

{

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

}

phaser.arriveAndAwaitAdvance(); *// await all creation*

task.run();

}

}.start();

}

*// allow threads to start and deregister self*

phaser.arriveAndDeregister();

}

## ForkJoinPool

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html) class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html) processes.

Fork/Join (F/J) framework is a structure that supports parallel programming where problems are recursively split into smaller parts, solved in parallel and recombined

Result solve(Problem problem) {

**if** (problem is small)  
         directly solve problem  
**else** {  
        split problem into independent parts  
        fork **new** subtasks to solve each part  
        join all subtasks  
        compose result from subresults  
    }  
}

Basically the **Fork-Join breaks the task at hand into mini-tasks** until the mini-task is simple enough that it can be solved without further breakups. It’s **like a**[**divide-and-conquer algorithm**](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithms). One important concept to note in this framework is that **ideally no worker thread is idle**. They implement a **work-stealing algorithm** in that idle workers steal the work from those workers who are busy.

The core classes supporting the Fork-Join mechanism are [ForkJoinPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html) and [ForkJoinTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html).

Although you specify any initial pool size, the **pool adjusts its size dynamically in an attempt to maintain enough active threads** at any given point in time. Another important difference compared to other ExecutorService's is that this pool need not be explicitly shutdown upon program exit because all its threads are in daemon mode.

There are three different ways of submitting a task to the ForkJoinPool.

**1) execute() method** //Desired asynchronous execution; call its fork method to split the work between multiple threads.  
**2) invoke() method**: //Await to obtain the result; call the invoke method on the pool.  
**3) submit() method**: //Returns a Future object that you can use for checking status and obtaining the result on its completion.

This is an abstract class for creating tasks that run within a ForkJoinPool. The Recursiveaction and RecursiveTask are the only two direct, known subclasses of ForkJoinTask. The only difference between these two classes is that the RecursiveAction does not return a value while RecursiveTask does have a return value and returns an object of specified type.

In both cases, you would need to implement the compute method in your subclass that performs the main computation desired by the task. The ForkJoinTask class provides several methods for checking the execution status of a task. The **isDone()** method returns true if a task completes in any way. The **isCompletedNormally()** method returns true if a task completes without cancellation or encountering an exception, and **isCancelled()** returns true if the task was cancelled. Lastly, **isCompletedabnormally()** returns true if the task was either cancelled or encountered an exception.

You are going to implement a **program that will search for files with a determined extension inside a folder and its subfolders**. The ForkJoinTask class you’re going to implement will process the content of a folder. For each subfolder inside that folder, it will send a new task to the ForkJoinPool class in an asynchronous way.

public class **FolderProcessor** extends RecursiveTask<List<String>>

{

private static final long serialVersionUID = 1L;

***//This attribute will store the full path of the folder this task is going to process.***

private final String path;

**//This attribute will store the name of the extension of the files this task is going to look for.**

private final String extension;

**//Implement the constructor of the class to initialize its attributes**

public FolderProcessor(String path, String extension)

{

this.path = path;

this.extension = extension;

}

**//Implement the compute() method. As you parameterized the RecursiveTask class with the List&lt;String&gt; type,**

**//this method has to return an object of that type.**

@Override

protected <List<String>>compute()

{

**//List to store the names of the files stored in the folder**.

List<String> list = new ArrayList<String>();

**//FolderProcessor tasks to store the subtasks that are going to process the subfolders stored in the folder**

List<FolderProcessor> tasks = new ArrayList<FolderProcessor>();

**//Get the content of the folder.**

File file = new File(path);

File content[] = file.listFiles();

**//For each element in the folder, if there is a subfolder, create a new FolderProcessor object**

**//and execute it asynchronously using the fork() method**.

if (content != null)

{

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

{

if (content[i].isDirectory())

{

FolderProcessor task = new FolderProcessor(content[i].getAbsolutePath(), extension);

task.fork();

tasks.add(task);

}

**//Otherwise, compare the extension of the file with the extension you are looking for using the checkFile() method**

**//and, if they are equal, store the full path of the file in the list of strings declared earlier.**

else

{

if (checkFile(content[i].getName()))

{

list.add(content[i].getAbsolutePath());

}

}

}

}

**//If the list of the FolderProcessor subtasks has more than 50 elements,**

**//write a message to the console to indicate this circumstance**.

if (tasks.size() > 50)

{

System.out.printf(&quot;%s: %d tasks ran.\n&quot;, file.getAbsolutePath(), tasks.size());

}

**//add to the list of files the results returned by the subtasks launched by this task**.

addResultsFromTasks(list, tasks);

**//Return the list of strings**

return list;

}

**//For each task stored in the list of tasks, call the join() method that will wait for its finalization and then will return the result of the task.**

**//Add that result to the list of strings using the addAll() method.**

private void addResultsFromTasks(List&lt;String&gt; list, List&lt;FolderProcessor&gt; tasks)

{

for (FolderProcessor item : tasks)

{

list.addAll(item.join());

}

}

**//This method compares if the name of a file passed as a parameter ends with the extension you are looking for.**

private boolean checkFile(String name)

{

return name.endsWith(extension);

}

}

public class Main

{

public static void main(String[] args)

{

//Create ForkJoinPool using the default constructor.

ForkJoinPool pool = new ForkJoinPool();

//Create three FolderProcessor tasks. Initialize each one with a different folder path.

FolderProcessor system = new FolderProcessor(&quot;C:\\Windows&quot;, &quot;log&quot;);

FolderProcessor apps = new FolderProcessor(&quot;C:\\Program Files&quot;, &quot;log&quot;);

FolderProcessor documents = new FolderProcessor(&quot;C:\\Documents And Settings&quot;, &quot;log&quot;);

//Execute the three tasks in the pool using the execute() method.

pool.execute(system);

pool.execute(apps);

pool.execute(documents);

//Write to the console information about the status of the pool every second

//until the three tasks have finished their execution.

do

{

System.out.printf(&quot;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n&quot;);

System.out.printf(&quot;Main: Parallelism: %d\n&quot;, pool.getParallelism());

System.out.printf(&quot;Main: Active Threads: %d\n&quot;, pool.getActiveThreadCount());

System.out.printf(&quot;Main: Task Count: %d\n&quot;, pool.getQueuedTaskCount());

System.out.printf(&quot;Main: Steal Count: %d\n&quot;, pool.getStealCount());

System.out.printf(&quot;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n&quot;);

try

{

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e)

{

e.printStackTrace();

}

} while ((!system.isDone()) || (!apps.isDone()) || (!documents.isDone()));

//Shut down ForkJoinPool using the shutdown() method.

pool.shutdown();

//Write the number of results generated by each task to the console.

List&lt;String&gt; results;

results = system.join();

System.out.printf(&quot;System: %d files found.\n&quot;, results.size());

results = apps.join();

System.out.printf(&quot;Apps: %d files found.\n&quot;, results.size());

results = documents.join();

System.out.printf(&quot;Documents: %d files found.\n&quot;, results.size());

}

}

**If the task finds a folder, it creates another Task object to process that folder and sends it to the pool using the fork() method**. This method sends the task to the pool that will execute it if it has a free worker-thread or it can create a new one. The **method returns immediately, so the task can continue processing** the content of the folder. For every file, a task compares its extension with the one it’s looking for and, if they are equal, adds the name of the file to the list of results.

**Once the task has processed all the content of the assigned folder, it waits for the finalization of all the tasks it sent to the pool using the join() method**. This method called in a task waits for the finalization of its execution and returns the value returned by the **compute()** method. The task groups the results of all the tasks it sent with its own results and returns that list as a return value of the compute() method.

### Difference between Fork/Join Framework And ExecutorService

The **main difference between the Fork/Join and the Executor frameworks is the work-stealing algorithm**. Unlike the Executor framework, when a task is waiting for the finalization of the sub-tasks it has created using the join operation, the thread that is executing that task (called worker thread ) looks for other tasks that have not been executed yet and begins its execution. By this way, the threads take full advantage of their running time, thereby improving the performance of the application.

## Concurrent collections

There are a couple of problems with the thread-safe collections:

1. Code that iterates over a collection that might be modified by another thread during the iteration requires a lock to avoid a thrown java.util.ConcurrentModificationException. This requirement is necessary because Collections framework classes return fail-fast iterators, which are iterators that throw ConcurrentModificationException when a collection is modified during iteration. Fail-fast iterators are often an inconvenience to concurrent applications.
2. Performance often suffers when these synchronized collections are accessed frequently from multiple threads; this is a performance problem that impacts an application's scalability.

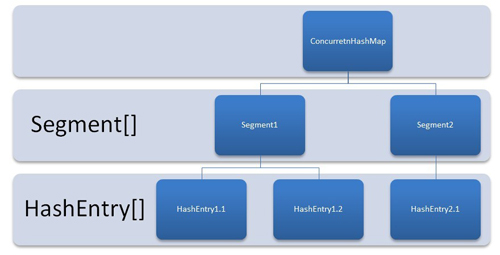
java.util.concurrent collections-oriented classes return *weakly consistent iterators*, which have the following properties:

* When an element is removed after iteration starts, but hasn't yet been returned via the iterator's next() method, it won't be returned.
* When an element is added after iteration starts, it may or may not be returned.
* Regardless of changes to the collection, no element is returned more than once in an iteration.

## ****ConcurrentHashMap****

In concurrentHashMap, the **difference lies in internal structure to store these key-value pairs**. ConcurrentHashMap has an addition concept of segments. It will be easier to understand it you think of one segment equal to one HashMap [conceptually]. A concurrentHashMap is divided into number of segments [default 16] on initialization. ConcurrentHashMap allows similar number (16) of threads to access these segments concurrently so that each thread work on a specific segment during high concurrency.

This way, if when your key-value pair is stored in segment 10; code does not need to block other 15 segments additionally. This structure provides a very high level of concurrency.



In other words, **ConcurrentHashMap uses a multitude of locks, each lock controls one segment of the map**. When setting data in a particular segment, the lock for that segment is obtained. So essentially **update operations are synchronized**.  
**When getting data, a volatile read is used** without any synchronization. If the volatile read results in a miss, then the lock for that segment is obtained and entry is again searched in synchronized block.

Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact.  Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not [synchronized](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html),**concurrent retrieval may not reflect most recent change on Map**

[**putIfAbsent**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html#putIfAbsent(K,%20V))([**K**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html) key, [**V**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html) value)

If the specified key is not already associated with a value, associate it with the given value.

ConcurrentHashMap doesn't allow null as key or value.  
  
Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

## CopyOnWriteArrayList and CopyOnWriteArraySet

CopyOnWriteArrayListimplements List interface like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html), [Vector](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html) and [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html)but its a thread-safe collection and it achieves its [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) in a slightly different way than Vector or other thread-safe collection class.

Normally CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient if you have a [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) where Iteration outnumber mutation e.g. you mostly need to [iterate the ArrayList](http://java67.blogspot.sg/2012/08/how-to-traverse-iterate-or-loop-ArrayList-in-java-example-tutorial.html) and don't modify it too often. Iterator ofCopyOnWriteArrayList is [fail-safe](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and doesn't throw ConcurrentModificationException even if underlyingCopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList. Consequently all the updates made on CopyOnWriteArrayList is not available to Iterator.

 difference between ArrayList and CopyOnWriteArrayList in Java , which is another implementation of List interface :

1) First and foremost difference between CopyOnWriteArrayList and ArrayList in Java is that CopyOnWriteArrayList is a [thread-safe collection](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) while ArrayList is not thread-safe and cannot be used in multi-threaded environment.

2) Second difference between ArrayList and CopyOnWriteArrayList is that [Iterator of ArrayList is fail-fast](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and throw ConcurrentModificationException once detect any modification in List once iteration begins but Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException.

3) Third difference between CopyOnWriteArrayList vs ArrayList is that [Iterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) of former doesn't support remove operation while Iterator of later supports remove() operation.

As will all data structures it is important to understand when to use them. As the name CopyOnWrite says, a copy of the whole list is made each time you write to the list like adding an element or remove an element. As you can figure out yourself, this can be pretty expensive when your list is large.  
This means that a CopyOnWriteArrayList (and CopyOnWriteArraySet) is mostly useful when you have few modifications but many reads because reads are very cheap and don’t require synchronization.

When you iterate over a CopyOnWriteArrayList and CopyOnWriteArraySet the iterator uses a snapshot of the underlying list (or set) and does not reflect any changes to the list or set after the snapshot was created. The iterator will never throw a ConcurrentModificationException.

## BlockingQueue

BlockingQueue is a unique collection type which not only store elements but also supports flow control by introducing blocking if either BlockingQueue is full or empty. take() method of BlockingQueue will block if Queue is empty and put() method of BlockingQueue will block if Queue is full.   
This property makes BlockingQueue an ideal choice for implementing [Producer consumer design pattern](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html) where one thread insert element into BlockingQueue and other thread consumes it.  
  
1) BlockingQueue in Java doesn't allow null elements, various implementation of BlockingQueue like ArrayBlockingQueue, LinkedBlockingQueue throws [NullPointerException](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html) when you try to add null on queue.

**BlockingQueue**<**String**> bQueue = **new** **ArrayBlockingQueue**<**String**>(10);  
*//bQueue.put(null); //NullPointerException - BlockingQueue in Java doesn't allow null*  
        
bQueue = **new** **LinkedBlockingQueue**<**String**>();  
bQueue.put(**null**);  
  
**Exception** in thread "main" java.lang.**NullPointerException**  
        at java.util.concurrent.**LinkedBlockingQueue**.put(**LinkedBlockingQueue**.java:288)  
  
2) BlockingQueue can be bounded or unbounded. A bounded BlockingQueue is one which is initialized with initial capacity and call to put() will be blocked if BlockingQueue is full and size is equal to capacity. This bounding nature makes it ideal to use a shared queue between multiple threads like in most common [Producer consumer solutions in Java](http://javarevisited.blogspot.de/2012/02/producer-consumer-design-pattern-with.html). An unbounded Queue is one which is initialized without capacity, actually by default it initialized with Integer.MAX\_VALUE.   
  
**BlockingQueue**<**String**> bQueue = **new** **ArrayBlockingQueue**<**String**>(2);  
bQueue.put("Java");  
**System**.out.println("Item 1 inserted into BlockingQueue");  
bQueue.put("JDK");  
**System**.out.println("Item 2 is inserted on BlockingQueue");  
bQueue.put("J2SE");  
**System**.out.println("Done");  
  
Output:  
Item 1 inserted into **BlockingQueue**  
Item 2 is inserted on **BlockingQueue**  
  
This code will only insert Java and JDK into BlockingQueue and then it will block while inserting 3rd element J2SE because size of BlockingQueue is 2 here.  
  
3)BlockingQueue implementations like ArrayBlockingQueue, LinkedBlockingQueue and PriorityBlockingQueue are [thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html). All queuing method uses concurrency control and internal locks to perform operation atomically. Since BlockingQueue also extend Collection, bulk Collection operations like addAll(), containsAll() are not performed atomically until any BlockingQueue implementation specifically supports it. So call to addAll() may fail after inserting couple of elements.  
  
4) Common methods of BlockingQueue is are put() and take() which are [blocking methods in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) and used to insert and retrive elements from BlockingQueue in Java. put() will block if BlockingQueue is full and take() will block if BlockingQueue is empty, call to take() removes element from head of Queue as shown in following example:

**BlockingQueue**<**String**> bQueue = **new** **ArrayBlockingQueue**<**String**>(2);  
bQueue.put("Java"); *//insert object into BlockingQueue*  
**System**.out.println("BlockingQueue after put: " + bQueue);  
bQueue.take(); *//retrieve object from BlockingQueue in Java*  
**System**.out.println("BlockingQueue after take: " + bQueue);  
  
Output:  
**BlockingQueue** after put: [Java]  
**BlockingQueue** after take: []

5) BlockingQueue interface extends Collection, Queue and Iterable interface which provides it all Collection and Queue related methods like poll(), and peak(), unlike take(), peek() method returns head of the queue without removing it, poll() also retrieves and removes elements from head but can wait till specified time if Queue is empty.  
  
6)Other important methods from BlockingQueue in Java is remainingCapacity() and offer(), former returns number remaining space in BlockingQueue, which can be filled without blocking while later insert object into queue if possible and return true if success and false if fail unlike add() method which [throws](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) IllegalStateException if it fails to insert object into BlockingQueue. Use offer() over add() wherever possible.

ArrayBlockingQueue is also fixed size bounded buffer on the other hand LinkedBlockingQueue is an optionally bounded queue built on top of Linked nodes. In terms of throughput LinkedBlockingQueue provides higher throughput than ArrayBlockingQueue in Java.  
  
**Producer consumer pattern** is every where in real life and depict coordination and collaboration. Like one person is preparing food (Producer) while other one is serving food (Consumer), both will use shared table for putting food plates and taking food plates. Producer which is the person preparing food will wait if table is full and Consumer (Person who is serving food) will wait if table is empty. table is a shared object here.  
  
1) Producer Consumer Pattern simple development. you can Code Producer and Consumer independently and Concurrently, they just need to know shared object.

2) Producer doesn't need to know about who is consumer or how many consumers are there. Same is true with Consumer.

3) Producer and Consumer can work with different speed. There is no risk of Consumer consuming half-baked item.

In fact by monitoring consumer speed one can introduce more consumer for better utilization.

4) Separating producer and Consumer functionality result in more clean, readable and manageable code.

public class **ProducerConsumerPattern** {

    public static void main(String args[]){

**//Creating shared object**

     BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

     Thread prodThread = new Thread(new Producer(sharedQueue));

     Thread consThread = new Thread(new Consumer(sharedQueue));

**//Starting producer and Consumer thread**

     prodThread.start();

     consThread.start();

    }

 }

**//Producer Class in java**

class **Producer** implements **Runnable** {

    private final **BlockingQueue** sharedQueue;

    public Producer(BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        for(int i=0; i<10; i++){

            try {

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

                sharedQueue.put(i);

            } catch (InterruptedException ex) {

                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);

            }

        }

    }

}

**//Consumer Class in Java**

class Consumer implements Runnable{

    private final BlockingQueue sharedQueue;

    public Consumer (BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        while(true){

            try {

                System.out.println("Consumed: "+ sharedQueue.take());

            } catch (InterruptedException ex) {

                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);

            }

        }

    }

}

Deque interface is added in Java 6 and it extends Queue interface to support insertion and removal from both end of Queue referred as head and tail. Java6 also provides concurrent implementation of Deque like ArrayDeque and LinkedBlockingDeque. Deque Can be used efficiently to increase parallelism in program by allowing set of [worker thread](http://javarevisited.blogspot.sg/2013/01/threadlocal-memory-leak-in-java-web.html) to help each other by taking some of work load from other thread by utilizing Deque double end consumption property. So if all [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) has there own set of task Queue and they are consuming from head; helper thread can also share some work load via consumption from tail.  
  
 ConcurrentSkipListMap and ConcurrentSkipListSet

Just like [ConcurrentHashMap](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) provides a concurrent alternative of [synchronized HashMap](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). ConcurrentSkipListMap and ConcurrentSkipListSet provide concurrent alternative for synchronized version of SortedMap and SortedSet. For example instead of using TreeMap or TreeSet wrapped inside synchronized Collection, You can consider using ConcurrentSkipListMap or ConcurrentSkipListSet from java.util.concurrent package. They also implement NavigableMap and NavigableSet to add additional navigation method we have seen in our post [How to use NavigableMap in Java](http://javarevisited.blogspot.sg/2013/01/what-is-navigablemap-in-java-6-example-submap-head-tail.html).  
  
These two classes vary in a few ways.

[ConcurrentHashMap](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ConcurrentHashMap.html) does not guarantee\* the runtime of its operations as part of its contract. It also allows tuning for certain load factors (roughly, the number of threads concurrently modifying it).

[ConcurrentSkipListMap](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), on the other hand, guarantees average O(log(n)) performance on a wide variety of operations. It also does not support tuning for concurrency's sake. ConcurrentSkipListMap also has a number of operations that ConcurrentHashMap doesn't: ceilingEntry/Key, floorEntry/Key, etc. It also maintains a sort order, which would otherwise have to be calculated (at notable expense) if you were using a ConcurrentHashMap.

Basically, different implementations are provided for different use cases. If you need quick single key/value pair addition and quick single key lookup, use the HashMap. If you need faster in-order traversal, and can afford the extra cost for insertion, use the SkipListMap

This class implements a concurrent variant of [SkipLists](http://en.wikipedia.org/wiki/Skip_list) providing expected average log(n) time cost for the containsKey, get, put and remove operations and their variants. Insertion, removal, update, and access operations safely execute concurrently by multiple threads. Iterators are weakly consistent, returning elements reflecting the state of the map at some point at or since the creation of the iterator. They do *not* throw [ConcurrentModificationException](https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html), and may proceed concurrently with other operations. Ascending key ordered views and their iterators are faster than descending ones.

All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do *not* support the Entry.setValue method. (Note however that it is possible to change mappings in the associated map using put, putIfAbsent, or replace, depending on exactly which effect you need.)

The headMap(T toKey) method returns a view of the map containing the keys which are strictly less than the given key.

If you make changes to the original map, these changes are reflected in the head map

Beware that, unlike in most collections, the size method is *not* a constant-time operation. Because of the asynchronous nature of these maps, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal. Additionally, the bulk operations putAll, equals, toArray, containsValue, and clear are *not* guaranteed to be performed atomically.

## ScheduledExecutorService

The java.util.concurrent.ScheduledExecutorService is an [**ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution. Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to theScheduledExecutorService.

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

### schedule (Runnable task, long delay, TimeUnit timeunit)

### scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

### scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

You shut down a ScheduledExecutorService using the shutdown() or shutdownNow() methods which are inherited from the ExecutorService interface.

## ReentrantLock

ReentrantLock is one of the most useful addition in Java concurrency package and several of concurrent collection classes from java.util.concurrent package is written using ReentrantLock,  
  
Two key feature of ReentrantLock, which provides more control on lock acquisition is trying to get a lock with ability to interrupt, and a timeout on waiting for lock, these are key for writing responsive and scalable systems in Java.  
ReentrantLock is a concrete implementation of Lock [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) provided in Java concurrency package from Java 1.5 onwards.  As per Javadoc, ReentrantLock is mutual exclusive lock, similar to implicit locking provided by [synchronized keyword in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html), with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)until a call to unlock() method. Fairness parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().  
  
 **main difference between synchronized and ReentrantLock**

1) Another significant difference between ReentrantLock and synchronized keyword is **fairness**. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is **tryLock()** method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html)of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, **ability to interrupt**Thread while waiting for Lock. In case of [synchronized](http://javarevisited.blogspot.com/2012/03/mixing-static-and-non-static.html) keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is [waiting for lock](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html). SimilarlytryLock() with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

Benefits

1) Ability to lock interruptibly.

2) Ability to timeout while waiting for lock.

3) Power to create fair lock.

4) API to get list of waiting thread for lock.

5) Flexibility to try for lock without blocking.

public class ReentrantLockHowto {

    private final ReentrantLock lock = new ReentrantLock();

    private int count = 0;

     //Locking using Lock and ReentrantLock

     public int getCount() {

        lock.lock();

        try {

            System.out.println(Thread.currentThread().getName() + " gets Count: " + count);

            return count++;

        } finally {

            lock.unlock();

        }

     }

     //Implicit locking using synchronized keyword

     public synchronized int getCountTwo() {

            return count++;

     }

A reentrant lock will allow the lock holder to enter blocks of code even after it has already obtained the lock by entering other blocks of code. A non-reentrant lock would have the lock holder block on itself as it would have to release the lock it obtained from another block of code to reobtain that same lock to enter the nested lock requiring block of code  
  
Extended capabilities of reentrant lock include :-  
===============================================  
  
1) The ability to have more than one condition variable per monitor. Monitors that use the synchronized keyword can only have one. This means reentrant locks support more than one wait()/notify() queue.  
2) The ability to make the lock "fair". "[fair] locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order." Synchronized blocks are unfair.  
3) The ability to check if the lock is being held.

Disadvantages of reentrant locks are :-  
====================================  
  
Need to add import statement.  
Need to wrap lock acquisitions in a try/finally block. This makes it more ugly than the synchronized keyword.  
The synchronized keyword can be put in method definitions which avoids the need for a block which reduces nesting.  
  
Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true. Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition. The key property that waiting for a condition provides is that it atomically releases the associated lock and suspends the current thread, just like Object.wait.

A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) instance use its [newCondition()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#newCondition()) method.

class BoundedBuffer {

**final Lock lock = new ReentrantLock();**

final Condition notFull = **lock.newCondition();**

final Condition notEmpty = **lock.newCondition();**

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

**lock.lock();**

**try {**

while (count == items.length)

**notFull.await();**

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

**notEmpty.signal();**

**} finally {**

**lock.unlock();**

**}**

}

public Object take() throws InterruptedException {

**lock.lock();**

**try {**

while (count == 0)

**notEmpty.await();**

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

**notFull.signal();**

return x;

**} finally {**

**lock.unlock();**

**}**

}

}

## Difference between Singleton Pattern vs Static Class

Since both Singleton pattern and static class provides good accessibility, and they share some similarities e.g. both can be used without creating object and both provide only one instance, at very high level it looks that they both are intended for same task.  
  
In order to answer these question, it’s important to remember fundamental difference between Singleton pattern and static class, former gives you an [Object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html), while later just provide static methods. Since an object is always much more capable than a method, it can guide you when to use Singleton pattern vs static methods.  
  
1) If your Singleton is not maintaining any state, and just providing global access to methods, than consider using static class, as static methods are much faster than Singleton, because of [static binding](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) during compile time. But remember its not advised to maintain state inside static class, especially in concurrent environment, where it could lead subtle [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) when modified parallel by multiple threads without adequate synchronization.  
  
Difference between Singleton vs Static in Java

1) Static class provides better performance than Singleton pattern, because static methods are bonded on compile time.

2) One more difference between Singleton and static is ability to override. Since [static methods in Java cannot be overridden](http://java67.blogspot.com/2012/08/can-we-override-static-method-in-java.html), they leads to inflexibility. On the other hand, you can override methods defined in Singleton class by extending it.

3) Static classes are hard to mock and consequently hard to test than Singletons, which are pretty easy to mock and thus easy to test. It’s easier to write [JUnit test](http://javarevisited.blogspot.com/2013/03/how-to-write-unit-test-in-java-eclipse-netbeans-example-run.html) for Singleton than static classes, because you can pass mock object whenever Singleton is expected, e.g. into constructor or as method arguments.

4) If your requirements needs to maintain state than Singleton pattern is better choice than static class, because maintaining  state in later case is nightmare and leads to subtle bugs.

5) Singleton classes can be [lazy loaded](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) if it’s a heavy object, but static class doesn't have such advantages and always eagerly loaded.

6) Many [Dependency Injection framework](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) manages Singleton quite well e.g. Spring, which makes using them very easy.

1) Singleton class can be extended. Polymorphism can save a lot of repetition.  
2) A Singleton class can implement an interface, which can come in handy when you want to separate implementation from API.  
3)Singleton can be extended. Static not.  
4)Singleton creation may not be threadsafe if it isn't implemented properly. Static not.  
5)Singleton can be passed around as an object. Static not.  
6)Singleton can be garbage collected. Static not.  
7)Singleton object stores in Heap but, static object stores in stack  
8)We can clone the object of Singleton but, we cannot clone the static class object  
9)Singleton class follow the OOP(object oriented principles) but not static class  
10)Another advantage of a singleton is that it can easily be serialized, which may be necessary if you need to save its state to disc, or send it somewhere remotely.

## Exception Handling

Choosing between checked and unchecked exception is always been confusing for Java programmers. Checked exceptions ensures that you provide exception handling code for error conditions, which is a way from language to enforcing you for writing robust code, but same time it also add lots of clutter into code and makes it unreadable. Also, it seems reasonable to catch exception and do something if you have alternatives or recovery strategies.  
  
Its become extremely important to log or print cause of root exception. Java Exception class provides getCause() method to retrieve cause which can be used to provide more information about root cause of Exception. message of Exception is the most important place, where you can point out cause of problem because this is the first place every programmer looks upon.  
One thing which is worth remembering is that Exceptions are costly, and can slow down your code. Suppose you have method which is reading from ResultSet and often throws SQLException than move to next element, will be much slower than normal code which doesn't throw that Exception. So minimizing catching unnecessary Exception and moving on, without fixing there root cause.   
  
1) Consider Making Business Exception class as Immutable, to avoid any accidental change on Error. Make them final class, with all private final fields, initialized by constructor arguments.  
  
2) Never Ever show Exception stacktrace in web pages, this is particularly important for Java developer, who writes JSP pages. Instead of showing Exception stacktrace to client, Use Global exception handler to show 404 page.  
  
3) While writing your own Exception classes, both checked and unchecked, make sure you have provided methods to get exception, message, error code, which is required by clients for effective exception handling. Failing to do so may result in Client parsing error message and extracting relevant information, which results in ugly, brittle code.

Trunk vs branch vs tag in subversion or SVN

Technically all three i.e. trunk, branch and tag are folders in SVN. If you are using tortoise SVN, a popular windows client for subversion, you can explore trunk, branch or tag.

As I said earlier, trunk is place where main development happens, and branches are places where different developer work on different functionalities. This division is purely based on how programmer uses trunk and branches. Similarly, tags are used to backup releases e.g. alpha release or beta release or any version of release. Main *difference between branch and tag in subversion* is that, tag is a read only copy of [source code](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) at any point and no further change on tag is accepted, while branch is mainly for development.   
  
- A trunk in SVN is main development area, where major development happens.

- A branch in SVN is sub development area where parallel development on different functionalities happens. After completion of functionality, a branch is usually merged back into trunk.

- A tag in SVN is read only copy of source code from branch or tag at any point of time. tag is mostly used to create a copy of released source code for restore and backup.

## Comparing Enum with equals and ==

**1) Using == for comparing Enum can prevent NullPointerException**

This one is easy to guess, but same time provide worth of money. If you compare any Enum with null, using == operator, it will result in false, but if you use equals() method to do this check, you may get NullPointerException, unless you are using calling equals in right way as shown in [how to avoid NullPointerException in Java](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html). Look at below code, here we are comparing an unknown Shape object with Shape enum which containsCIRCLE, RECTANGLE etc.

private enum Shape{

RECTANGLE, SQUARE, CIRCLE, TRIANGLE;

}

private enum Status{

ON, OFF;

}

Shape unknown = null;

Shape circle = Shape.CIRCLE;

boolean result = unknown == circle; *//return false*

result = unknown.equals(circle); *//throws NullPointerException*

I agree this can be avoided by simply **comparing known to unknown** i.e. circle.equals(unknown), but this is one of the most common [coding error Java programmers make](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html). By using == to compare enum, you can completely avoid it.

**2) == method provides type safety during compile time**

Another advantage of using == to compare enum is, compile time safety. Equality or == operator checks if both enum object are from same enum type or not at compile time itself, while equals() method will also return false but at runtime. Since it's always better to detect errors at compile time, == scores over equals in case of comparing enum. If you are using [Eclipse](http://javarevisited.blogspot.com/2012/10/eclipse-shortcut-to-remove-all-unused-imports-java.html) or Netbeans, you can detect these error as soon as you type. By the way Netbeans also shows warning when you call equals() method on two incomparable types, but that is completely IDE specific.

**3) == should be faster than equals method**

This is more from common sense, using operator should be a touch faster than calling method, and than using operator. Though I believe modern JIT compiler might inline equals() method, when you compare two enums in Java. Which means this would not be big difference in terms of performance.But, without any smartness from compiler or [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html), I think == should always be touch faster.

public final int compareTo(E o) {

 Enum other = (Enum)o;

Enum self = this;

if (self.getClass() != other.getClass() && *// optimization*

self.getDeclaringClass() != other.getDeclaringClass()) throw new ClassCastException();

return self.ordinal - other.ordinal;

}

## ParseInt vs valueOf in Java

public static Integer valueOf(String s) throws NumberFormatException {

return Integer.valueOf(parseInt(s, 10));

}

public static Integer valueOf(int i) {

if(i >= -128 && i <= IntegerCache.high)

return IntegerCache.cache[i + 128];

else

return new Integer(i);

}

actual job of converting String to integer is done by parseInt() method, valueOf() just provide caching of frequently used Integer objects,   
  
This method first calls parseInt() method, in order to [convert String to primitive int](http://java67.blogspot.com/2013/03/how-to-convert-java-string-to-int-or.html), and then creates Integer object from that value. You can see it internally maintains an Integer cache. If primitive int is within range of cache, it returns Integer object from pool, otherwise it create a new object.  
  
public static int parseInt(String s) throws NumberFormatException {  
return parseInt(s,10);}

## Difference between abstract class and interface in Java

1) Interface in Java only contains declaration. You can not declare any concrete methods inside interface. On the other hand abstract class may contain both abstract and concrete methods, which makes abstract class an ideal place to provide common or default functionality.

2) Java interface can extend multiple interface also Java class can implement multiple interfaces, Which means interface can provide more polymorphism support than abstract class . By extending abstract class, a class can only participate in one Type hierarchy but by using interface it can be part of multiple type hierarchies. E.g. a class can be Runnable and Displayable at same time. One example I can remember of this is writing GUI application in J2ME, where  class extends Canvas and implements CommandListener to provide both graphic and event-handling functionality..

3) In order to implement interface in Java, until your class is abstract, you need to provide implementation of all methods, which is very painful. On the other hand abstract class may help you in this case by providing default implementation. Because of this reason, I prefer to have minimum methods in interface, starting from just one, I don't like idea of [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html), once annotation is introduced in Java 5. If you look JDK or any framework like Spring, which I does to understand OOPS and design patter better, you will find that most of interface contains only one or two methods e.g. Runnable, Callable, ActionListener etc.

## Difference between green thread and native thread

Native threads use the operating system's native ability to manage multi-threaded processes - in particular, they use the pthread library. When you run with native threads, the kernel schedules and manages the various threads that make up the process.

Green threads emulate multithreaded environments without relying on any native OS capabilities. They run code in user space that manages and schedules threads; Sun wrote green threads to enable Java to work in environments that do not have native thread support.

* Native threads can switch between threads pre-emptively, switching control from a running thread to a non-running thread at any time. Green threads only switch when control is explicitly given up by a thread (Thread.yield(), Object.wait(), etc.) or a thread performs a blocking operation (read(), etc.).
* On multi-CPU machines, native threads can run more than one thread simultaneously by assigning different threads to different CPUs. Green threads run on only one CPU.

Native threads create the appearancethat many Java processes are running: each thread takes up its own entry in the process table. One clue that these are all threads of the same process is that the memory size is identical for all the threads - they are all using the same memory

*Unfortunately, this behavior limits the scalability of Java on Linux.*

## What is the difference between Process and Thread?

A process is a self-contained execution environment and it can be seen as a program or application whereas Thread is a single task of execution within the process. Java runtime environment runs as a single process which contains different classes and programs as processes. Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

## What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements.

public class MyCallable implements Callable<String> {

    private long waitTime;

    public MyCallable(int timeInMillis){

        this.waitTime=timeInMillis;

    }

    @Override

    public String call() throws Exception {

        Thread.sleep(waitTime);

        //return the thread name executing this callable task

        return Thread.currentThread().getName();

    }

}

public class FutureTaskExample {

    public static void main(String[] args) {

        MyCallable callable1 = new MyCallable(1000);

        MyCallable callable2 = new MyCallable(2000);

        FutureTask<String> futureTask1 = new FutureTask<String>(callable1);

        FutureTask<String> futureTask2 = new FutureTask<String>(callable2);

        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.execute(futureTask1);

        executor.execute(futureTask2);

        while (true) {

            try {

                if(futureTask1.isDone() && futureTask2.isDone()){

                    System.out.println("Done");

                    //shut down executor service

                    executor.shutdown();

                    return;

                }

                if(!futureTask1.isDone()){

                //wait indefinitely for future task to complete

                System.out.println("FutureTask1 output="+futureTask1.get());

                }

                System.out.println("Waiting for FutureTask2 to complete");

                String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);

                if(s !=null){

                    System.out.println("FutureTask2 output="+s);

                }

            } catch (InterruptedException | ExecutionException e) {

                e.printStackTrace();

            }catch(TimeoutException e){

                //do nothing

            }

        }

    }

}

## What is Thread Scheduler and Time Slicing?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it’s execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

## Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

## What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.  
Every thread has it’s own ThreadLocal variable and they can use it’s get() and set() methods to get the default value or change it’s value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread.

public class Context {  
  
 private String transactionId = null;  
  
 /\* getters and setters here \*/  
  
}

public class MyThreadLocal {  
  
 public static final ThreadLocal userThreadLocal = new ThreadLocal();  
  
 public static void set(Context user) {  
 userThreadLocal.set(user);  
 }  
  
 public static void unset() {  
 userThreadLocal.remove();  
 }  
  
 public static Context get() {  
 return userThreadLocal.get();  
 }  
}

public class ThreadLocalDemo extends Thread {  
  
 public static void main(String args[]) {  
  
 Thread threadOne = new ThreadLocalDemo();  
 threadOne.start();  
  
 Thread threadTwo = new ThreadLocalDemo();  
 threadTwo.start();  
 }

@Override  
 public void run() {  
 // sample code to simulate transaction id  
 Context context = new Context();  
 context.setTransactionId(getName());  
  
 // set the context object in thread local to access it somewhere else  
 MyThreadLocal.set(context);  
  
 /\* note that we are not explicitly passing the transaction id \*/  
 new BusinessService().businessMethod();  
 MyThreadLocal.unset();  
  
 }  
}

public class BusinessService {  
  
 public void businessMethod() {  
 // get the context from thread local  
 Context context = MyThreadLocal.get();  
 System.out.println(context.getTransactionId());  
 }  
}

## What is Thread Group? Why it’s advised not to use it?

ThreadGroup is a class which was intended to provide information about a thread group. ThreadGroup API is weak and it doesn’t have any functionality that is not provided by Thread. Two of the major feature it had are to get the list of active threads in a thread group and to set the uncaught exception handler for the thread. But Java 1.5 has added setUncaughtExceptionHandler(UncaughtExceptionHandler eh) method using which we can add uncaught exception handler to the thread. So ThreadGroup is obsolete and hence not advised to use anymore.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){      @Override      public void uncaughtException(Thread t, Throwable e) {          System.out.println("exception occured:"+e.getMessage());      }  }); |

## Difference between Executor and Executors in Java

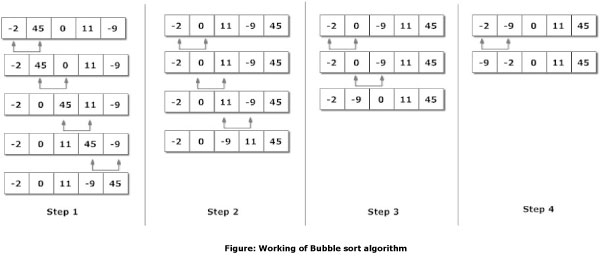
A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.**java.util.concurrent.Executors** provide implementation of**java.util.concurrent.Executor** interface to create the thread pool in java.

Bubble Sort Algorithm

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are *n* elements to be sorted then, the process mentioned above should be repeated *n-1* times to get required result. But, for better performance, in second step, last and second last elements are not compared becuase, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

A figure is worth a thousand words so, acknowledge this figure for better understanding of bubble sort.



Here, there are 5 elements to the sorted. So, there are 4 steps.

/\*

\* BUBBLE SORT

\* Finding the largest element with each iteration

\* Like the way, water bubble is coming out

\*/

**public** **static** **void** bubbleSort(**int**[] array) {

**for** (**int** i = 1; i < array.length ; i++)

**for** (**int** j = 0; j < array.length - i; j++)

**if** (array[j] > array[j + 1]) /\* For descending order use < \*/

{

**int** swap = array[j];

array[j] = array[j + 1];

array[j + 1] = swap;

}

}

Complexity of bubble sort is O(n2) which makes it a less frequent option for arranging in sorted order when quantity of numbers is high.

You can also use sort method of Arrays class to sort integers in ascending order but remember that sort method uses a variation of Quick sort algorithm.

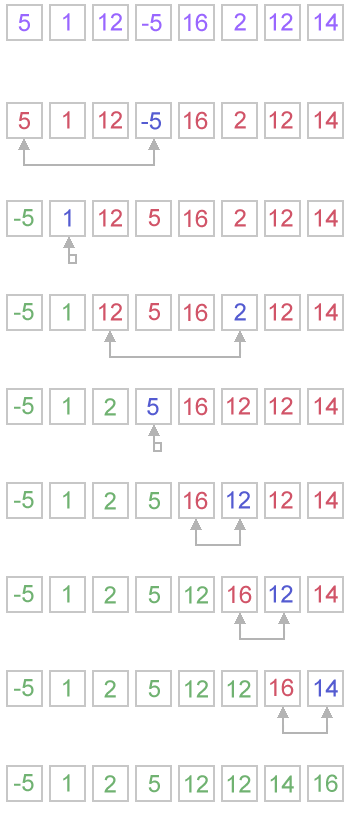
Selection Sort

The idea of algorithm is quite simple. Array is imaginary divided into two parts - **sorted one** and **unsorted one**. At the beginning, **sorted part** is **empty**, while **unsorted one** contains **whole array**. *At every step,* algorithm finds **minimal element** in the **unsorted part** and adds it to the end of the**sorted one**. When **unsorted part** becomes **empty**, algorithm *stops*.

When algorithm sorts an array, it swaps first element of unsorted part with minimal element and then it is included to the sorted part. This implementation of selection sort in **not stable**. In case of linked list is sorted, and, instead of swaps, minimal element is linked to the unsorted part, selection sort is **stable**.

Let us see an example of sorting an array to make the idea of selection sort clearer.

*Example.*Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



/\*

\* SELECTION SORT

\* Finding the smallest element with each iteration

\* Swapping it with the first element (of unsorted array)

\*/

**public** **static** **void** selectionSort(**int**[] data) {

**for** (**int** i = 0; i < data.length - 1; i++) {

**int** smallest = i;

**for** (**int** j = i + 1; j < data.length; j++)

**if** (data[j] < data[smallest])

smallest = j;

**int** smallerNumber = data[smallest];

data[smallest] = data[i];

data[i] = smallerNumber;

}

}

Every step of outer loop requires finding minimum in unsorted part. Summing up, n + (n - 1) + (n - 2) + ... + 1, results in O(n2) number of comparisons.

Number of swaps may vary from zero (in case of sorted array) to n - 1 (in case array was sorted in reversed order), which results in O(n) number of swaps. Overall algorithm complexity is O(n2).

##### Properties

* Not stable
* O(1) extra space
* Θ(n2) comparisons
* Θ(n) swaps
* Not adaptive

Worst Case Time Complexity : O(n2))

Best Case Time Complexity : O(n2)

Average Time Complexity : O(n2))

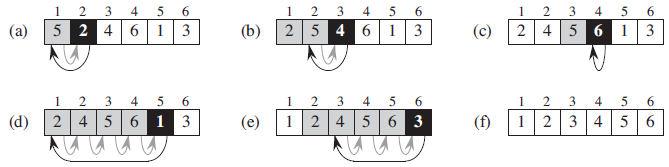
Space Complexity : O(1)

Insertion Sort

Insertion sort is a simple sorting algorithm, it builds the final sorted array one item at a time. It is much less efficient on large lists than other sort algorithms.

Advantages of Insertion Sort:   
  
1) It is very simple.  
2) It is very efficient for small data sets.  
3) It is stable; i.e., it does not change the relative order of elements with equal keys.  
4) In-place; i.e., only requires a constant amount O(1) of additional memory space.

Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.



The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)).

The simplest worst case input is an array sorted in reverse order. This gives insertion sort a quadratic running time (i.e., O(n2))

**public** **static** **void** insertionSort(**int**[] data)

{

// loop over data.length - 1 elements

**for** (**int** next = 1; next < data.length; next++)

{

**int** insert = data[next]; // value to insert

**int** position = next; // location to place element

// search for place to put current element

**while** (position > 0 && data[position - 1] > insert)

{

// shift element right one slot

data[position] = data[position - 1];

position--;

}

data[position] = insert; // place inserted element

// printPass(data, next, moveItem); // output pass of algorithm

}

#### }

#### Properties

* Stable
* O(1) extra space
* O(n2) comparisons and swaps
* Adaptive: O(n) time when nearly sorted
* Very low overhead

Worst Case Time Complexity : O(n2))

Best Case Time Complexity : O(n)

Average Time Complexity : O(n2))

Space Complexity : O(1)

Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. **Choose a pivot value.**We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.**Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.**Apply quicksort algorithm recursively to the left and the right parts.

**public** **void** quickSort(**int** arr[], **int** left, **int** right) {

**int** index = partition(arr, left, right);

**if** (left < index - 1)

quickSort(arr, left, index - 1);

**if** (index < right)

quickSort(arr, index, right);

}

**public** **int** partition(**int** arr[], **int** left, **int** right) {

**int** i = left, j = right;

**int** tmp;

**int** pivot = arr[(left + right) / 2];

**while** (i <= j) {

**while** (arr[i] < pivot)

i++;

**while** (arr[j] > pivot)

j--;

**if** (i <= j) {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

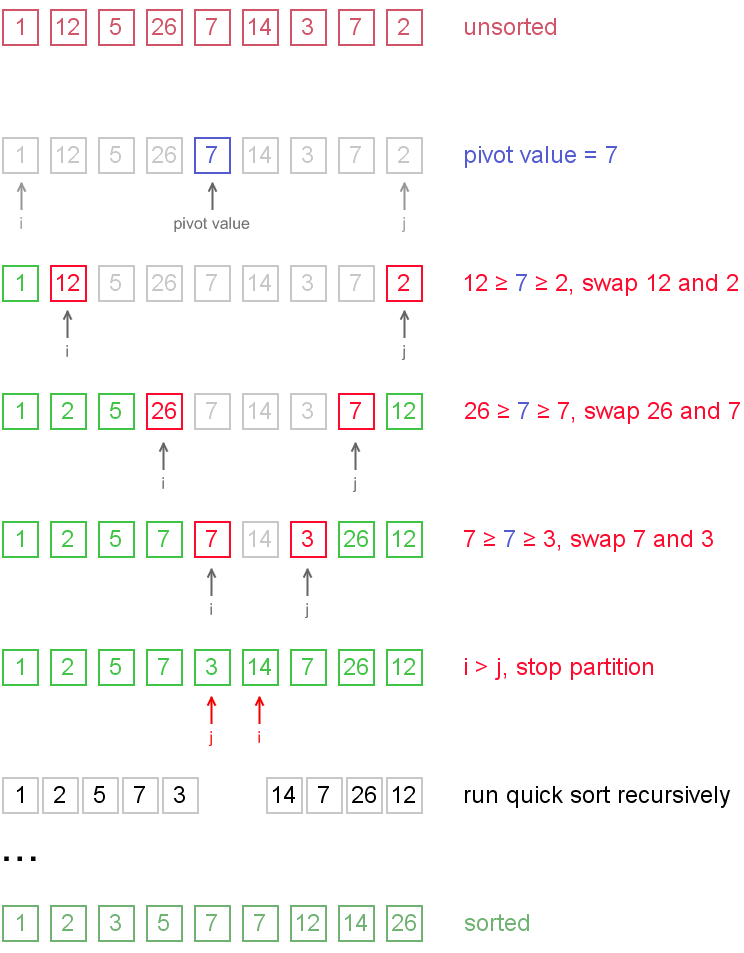
j--;

}

}

**return** i;

}



**public** **static** **void** quickSort(**int**[] data, **int** low, **int** high)

{

**int** i = low;

**int** j= high;

**int** pivot = data[ low + ( high-low)/2];

**while**(i<=j)

{

**while**(data[i]< pivot)

{

i++;

}

**while**( data[j] > pivot )

{

j--;

}

**if**(i<=j)

{

*exchangeNumbers*(data, i, j);

i++; j--;

}

}

**if**(low < j)

*quickSort*(data, low, j);

**if**(i < high)

*quickSort*(data, i, high);

}

**private** **static** **void** exchangeNumbers(**int**[] data, **int** i, **int** j) {

**int** temp = data[i];

data[i] = data[j];

data[j] = temp;

}

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :**O(n log n)

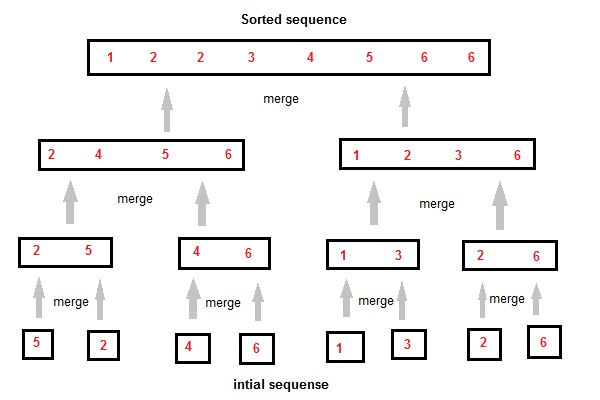
#### Properties

* Not stable
* O(log(n)) extra space (see discussion)
* O(n2) time, but typically O(n log(n)) time
* Not adaptive

Merge Sort

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.



**private** **static** **void** mergeSort(**int**[] data, **int** low, **int** high)

{

// test base case; size of array equals 1

**if** ((high - low) >= 1) // if not base case

{

**int** middle1 = (low + high) / 2; // calculate middle of array

**int** middle2 = middle1 + 1; // calculate next element over

// output split step

System.*out*.printf("split: %s%n",

*subarrayString*(data, low, high));

System.*out*.printf(" %s%n",

*subarrayString*(data, low, middle1));

System.*out*.printf(" %s%n%n",

*subarrayString*(data, middle2, high));

// split array in half; sort each half (recursive calls)

mergeSort(data, low, middle1); // first half of array

mergeSort(data, middle2, high); // second half of array

// merge two sorted arrays after split calls return

*merge* (data, low, middle1, middle2, high);

} // end if

}

**private** **static** String subarrayString(**int**[] data, **int** low, **int** high)

{

StringBuilder temporary = **new** StringBuilder();

// output spaces for alignment

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

temporary.append(" ");

// output elements left in array

**for** (**int** i = low; i <= high; i++)

temporary.append(" " + data[i]);

**return** temporary.toString();

}

**private** **static** **void** merge(**int**[] data, **int** left, **int** middle1,

**int** middle2, **int** right)

{

**int** leftIndex = left; // index into left subarray

**int** rightIndex = middle2; // index into right subarray

**int** combinedIndex = left; // index into temporary working array

**int**[] combined = **new** **int**[data.length]; // working array

// output two subarrays before merging

System.*out*.printf("merge: %s%n",

*subarrayString*(data, left, middle1));

System.*out*.printf(" %s%n",

*subarrayString*(data, middle2, right));

// merge arrays until reaching end of either

**while** (leftIndex <= middle1 && rightIndex <= right)

{

// place smaller of two current elements into result

// and move to next space in arrays

**if** (data[leftIndex] <= data[rightIndex])

combined[combinedIndex++] = data[leftIndex++];

**else**

combined[combinedIndex++] = data[rightIndex++];

}

// if left array is empty

**if** (leftIndex == middle2)

// copy in rest of right array

**while** (rightIndex <= right)

combined[combinedIndex++] = data[rightIndex++];

**else** // right array is empty

// copy in rest of left array

**while** (leftIndex <= middle1)

combined[combinedIndex++] = data[leftIndex++];

// copy values back into original array

**for** (**int** i = left; i <= right; i++)

data[i] = combined[i];

// output merged array

System.*out*.printf(" %s%n%n",

*subarrayString*(data, left, right));

}

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :**O(n)

#### Properties

* Stable
* Θ(n) extra space for arrays (as shown)
* Θ(log(n)) extra space for linked lists
* Θ(n·log(n)) time
* Not adaptive
* Does not require random access to data

Heap Sort

[Heapsort](http://en.wikipedia.org/wiki/Heapsort) is an in-place sorting algorithm with worst case and average complexity of O(n logn).

The basic idea is to turn the array into a binary heap structure, which has the property that it allows efficient retrieval and removal of the maximal element. We repeatedly "remove" the maximal element from the heap, thus building the sorted list from back to front. Heapsort requires random access, so can only be used on an array-like data structure.

Pseudocode:

**public** **static** **void** heapSort**(int[]** a**){**  
 **int** count = a.**length**;  
   
 *//first place a in max-heap order*  
 heapify**(**a, count**)**;  
   
 **int** end = count - **1**;  
 **while(**end > **0){**  
 *//swap the root(maximum value) of the heap with the*  
 *//last element of the heap*  
 **int** tmp = a**[**end**]**;  
 a**[**end**]** = a**[0]**;  
 a**[0]** = tmp;  
 *//put the heap back in max-heap order*  
 siftDown**(**a, **0**, end - **1)**;  
 *//decrement the size of the heap so that the previous*  
 *//max value will stay in its proper place*  
 end--;  
 **}**  
**}**  
   
**public** **static** **void** heapify**(int[]** a, **int** count**){**  
 *//start is assigned the index in a of the last parent node*  
 **int** start = **(**count - **2)** / **2**; *//binary heap*  
   
 **while(**start >= **0){**  
 *//sift down the node at index start to the proper place*  
 *//such that all nodes below the start index are in heap*  
 *//order*  
 siftDown**(**a, start, count - **1)**;  
 start--;  
 **}**  
 *//after sifting down the root all nodes/elements are in heap order*  
**}**  
   
**public** **static** **void** siftDown**(int[]** a, **int** start, **int** end**){**  
 *//end represents the limit of how far down the heap to sift*  
 **int** root = start;  
   
 **while((**root \* **2** + **1)** <= end**){** *//While the root has at least one child*  
 **int** child = root \* **2** + **1**; *//root\*2+1 points to the left child*  
 *//if the child has a sibling and the child's value is less than its sibling's...*  
 **if(**child + **1** <= end && a**[**child**]** < a**[**child + **1])**  
 child = child + **1**; *//... then point to the right child instead*  
 **if(**a**[**root**]** < a**[**child**]){** *//out of max-heap order*  
 **int** tmp = a**[**root**]**;  
 a**[**root**]** = a**[**child**]**;  
 a**[**child**]** = tmp;  
 root = child; *//repeat to continue sifting down the child now*  
 **}else**  
 **return**;  
 **}**  
**}**

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :**O(n)

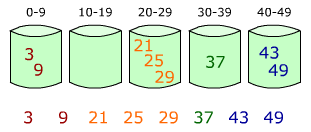
Bucket Sort

Bucket sort it’s the perfect sorting algorithm for the sequence above. We must know in advance that the integers are fairly well distributed over an interval (i, j). Then we can divide this interval in N equal sub-intervals (or buckets). We’ll put each number in its corresponding bucket. Finally for every bucket that contains more than one number we’ll use some linear sorting algorithm.

Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a **distribution sort**, and is a cousin of **radix sort** in the most to least significant digit flavor. Bucket sort is a generalization of **pigeonhole sort**. Since bucket sort is **not a comparison sort**, the (*n log n*) lower bound is inapplicable. The computational complexity estimates involve the number of buckets.

Bucket sort works as follows:

* Set up an array of initially empty buckets.
* Go over the original array, putting each object in its bucket.
* Sort each non-empty bucket.
* Visit the buckets in order and put all elements back into the original array.



public void bucketsort(int[] input) {

  //get hash codes

  final int[] code = hash(input);

  //create and initialize buckets to ArrayList: O(n)

  List<Integer>[] buckets = new List[code[1]];

  for (int i = 0; i < code[1]; i++) {

    buckets[i] = new ArrayList<Integer>();

  }

  //distribute data into buckets: O(n)

  for (int i : input) {

    buckets[hash(i, code)].add(i);

  }

  /\*\*

   \* Sort each bucket: O(n).

   \* I mentioned above that the worst case for bucket sort is counting

   \* sort. That's because in the worst case, bucket sort may end up

   \* with one bucket per key. In such case, sorting each bucket would

   \* take 1^2 = O(1). Even after allowing for some probabilistic

   \* variance, to sort each bucket would still take 2-1/n, which is

   \* still a constant. Hence, sorting all the buckets takes O(n).

   \*\*\*/

  for (List bucket : buckets) {

    Collections.sort(bucket);

  }

  int ndx = 0;

  //merge the buckets: O(n)

  for (int b = 0; b < buckets.length; b++) {

    for (int v : buckets[b]) {

      input[ndx++] = v;

    }

  }

}

private int[] hash(int[] input) {

  int m = input[0];

  for (int i = 1; i < input.length; i++) {

    if (m < input[i]) {

      m = input[i];

    }

  }

  return new int[]{m, (int) Math.sqrt(input.length)};

}

private int hash(int i, int[] code) {

  return (int) ((double) i / code[0] \* (code[1] - 1));

}

|  |  |
| --- | --- |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n+k) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n+k) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\cdot k) |

**radix sort**, like counting sort and bucket sort, is an integer based algorithm (i.e. the values of the input array are assumed to be   integers). Hence radix sort is among the fastest sorting algorithms   around, in theory. The particular distinction for radix sort is that it creates a bucket for each cipher (i.e. digit); as such, similar to  bucket sort, each bucket in radix sort must be a growable list that may admit different keys.

* If you are using radix sort and your numbers are decimal, then you need 10 buckets, one for each digit from 0 to 9.
* If you are using counting sort, then you need a bucket for each unique value in the input (actually you need a bucket for each value between 0 and max).
* If you are using bucketsort, you don't know how many buckets you will be using. Whatever hash function you are using will determine the number of buckets.

For **bucketsort** to work at

    its blazing efficiency, there are multiple prerequisites. First the

    hash function that is used to partition the elements need to be very

    good and must produce ordered hash: if i < k then hash(i) < hash(k).

    Second, the elements to be sorted must be uniformly distributed.

    The aforementioned aside, bucket sort is actually very good considering

    that counting sort is reasonably speaking its upper bound. And counting

    sort is very fast. The particular distinction for bucket sort is that

    it uses a hash function to partition the keys of the input array, so

    that multiple keys may hash to the same bucket. Hence each bucket must

    effectively be a growable list; similar to radix sort.

    Numerous Internet sites, including university pages, have erroneously

    written counting sort code and call them bucket sort. Bucket sort uses

    a hash function to distribute keys; counting sort creates a bucket for

    each key. Indeed there are perhaps greater similarities between radix

   sort and bucket sort, than there are between counting sort and bucket

sort.

**Counting sort**, like radix sort and bucket sort,

    is an integer based algorithm (i.e. the values of the input

    array are assumed to be integers). Hence counting sort is

    among the fastest sorting algorithms around, in theory. The

    particular distinction for counting sort is that it creates

    a bucket for each value and keep a counter in each bucket.

    Then each time a value is encountered in the input collection,

    the appropriate counter is incremented. Because counting sort

    creates a bucket for each value, an imposing restriction is

    that the maximum value in the input array be known beforehand.

Radix Sort

**public** **void** sortLSD(**int**[] array, **int** maxDigitSymbols) {

**int** mod = 10;

**int** dev = 1;

**for** (**int** i = 0; i < maxDigitSymbols; i++, dev \*= 10, mod \*= 10) {

*//System.out.println(Arrays.toString(array));*

**for**(**int** j = 0; j < array.length; j++) {

**int** bucket = (array[j] % mod) / dev;

counter[bucket].add(array[j]);

}

**int** pos = 0;

**for**(**int** j = 0; j < counter.length; j++) {

Integer value = **null**;

**while** ((value = counter[j].poll()) != **null**) {

*//System.out.println(value);*

array[pos++] = value;

}

}

}

}

Worst case performance O(kN)

Worst case space complexity O(K+N)

/\*\*

\* Counting sort

\* @param array array to be sorted

\* @return array sorted in ascending order

\*/

public static int[] countingSort(int[] array) {

// array to be sorted in, this array is necessary

// when we sort object datatypes, if we don't,

// we can sort directly into the input array

int[] aux = new int[array.length];

// find the smallest and the largest value

int min = array[0];

int max = array[0];

for (int i = 1; i < array.length; i++) {

if (array[i] < min) min = array[i];

else if (array[i] > max) max = array[i];

}

// init array of frequencies

int[] counts = new int[max - min + 1];

// init the frequencies

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

counts[array[i] - min]++;

}

// recalculate the array - create the array of occurences

counts[0]--;

for (int i = 1; i < counts.length; i++) {

counts[i] = counts[i] + counts[i-1];

}

// Sort the array right to the left

// 1) look up in the array of occurences the last occurence of the given value

// 2) place it into the sorted array

// 3) decrement the index of the last occurence of the given value

// 4) continue with the previous value of the input array (goto: 1), terminate if all values were already sorted

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

aux[counts[array[i] - min]--] = array[i];

}

return aux;

}

**counting sort is its complexity – O(n+k)**

Maven

*Apache Maven* is a build tool to support the developer at the whole process of a software project. Typical tasks of a build tool are the compilation of source code, running the tests and packaging the result into *JAR* files. In additional to these typical build capabilities, Maven can also perform related activities, e.g., create web sites, upload build results or generate reports.

Maven allows the developer to automate the process of the creation of the initial folder structure for the Java application, performing the compilation and testing and the packaging and deployment of the final product. It is implemented in Java which makes it platform-independent.

**Key features of Maven**

Apache Maven can be used in environments where common build tools like GNU Make or Apache Ant were used. The key features of Maven are:

* ***Convention over configuration:*** Maven tries to avoid as much configuration as possible, by choosing real world default values and supplying project templates (archtypes).
* ***Dependency management:*** it is possible to define dependencies to other projects. During the build, the Maven build system resolves the dependencies and, if needed, it also builds the dependent projects.
* ***Central repository:*** project dependencies can be loaded from the local file system, from the Internet or public repositories. The company behind the Maven project also provides a central repository called *Maven Central*.
* ***Extensible via plug-ins:*** The Maven build system is extensible via plug-ins, which allows keeping the Maven core small. The Maven core does for example not know how to compile Java source code, this is handled by the compiler plug-in.

## Project Object Model (POM)

The configuration of a Maven project is done via a *Project Object Model* (POM) , which is represented by a *pom.xml* file.

This file describes the project, configures plugins, and declares dependencies. The POM names the project, provides a set of unique identifiers (called coordinates) for a project, and defines the relationships between this project and others through dependencies, parents, and prerequisites.

A POM file can include a *modules* section, which tells Maven which directories have POM files which need to be built.

In the *build* section you can define plugins for which you need to build the artifacts in your project

#### Project unique identifier

| **Name** | **Description** |
| --- | --- |
| groupId | Defines a unique base name of the organization or group that created the project. This is normally a reverse domain name. For the generation the groupId also defines the package of the main class. |
| artifactId | Defines the unique name of the project. If you generate a new project via Maven this is also used as root folder for the project. |
| packaging | Defines the packaging method. This could be e.g. jar, war or ear. This setting define the basic sets of plug-ins which are bound to life cycle phase. This is not part of the unique identifier of the project.  If the packaging type is pom, Maven does not create anything for this project, it is just meta-data. |
| version | This defines the version of the project. |

A project’s groupId:artifactId:version (also known as GAV) make that project unique.

The full Maven coordinates are often written in the following format: groupId:artifactId:packaging:version

By default, this is the only configuration file required for the build process.

Maven always executes against an *effective POM*, a combination of settings from this project’s pom.xml, all parent POMs, a super-POM defined within Maven, user-defined settings, and active profiles.

The result of a build is called *artifact*. An artifact, for example, can be an executable or an archive of documents

## Maven Plug-ins and goals

A Maven plugin is a collection of one or more *goals*. A goal is a “unit of work” in Maven. It is possible to execute goals independently or a part of a larger chain of goals.

Goals can define parameters, which may have default values. Plugin goals can be attached to a life cycle phase. The goals are executed based on the information found in the POM of the project, e.g., the compiler:compile goal checks the POM for relevant parameters.

## Maven life cycle

Every build follows a specified *life cycle*. Maven comes with a default life cycle that includes the most common build *phases* like compiling, testing and packaging.

* **validate**: validate the project is correct and all necessary information is available
* **compile**: compile the source code of the project
* **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* **package**: take the compiled code and package it in its distributable format, such as a JAR.
* **integration-test**: process and deploy the package if necessary into an environment where integration tests can be run
* **verify**: run any checks to verify the package is valid and meets quality criteria
* **install**: install the package into the local repository, for use as a dependency in other projects locally
* **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

There are two other Maven lifecycles of note beyond the *default* list above. They are

* **clean**: cleans up artifacts created by prior builds
* **site**: generates site documentation for this project

If you instruct Maven to execute a phase, Maven executes all existing phases n the pre-defined sequence until has executed the defined phase. All relevant goals are executed during this process. A goal is relevant for a phase if the Maven plug-in or the POM bind this goal to the corresponding Maven life cycle phase.

**Packaging goal**

| **Life cycle phase** | **Goal binding** |
| --- | --- |
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | jar:jar |
| install | install:install |
| deploy | deploy:deploy |

### Adding goals to life cycle phases

You can add goals to life cycle phases by configure more Maven plug-ins and add them to a life cycle in your POM file. You need to specify which goal should be executed, and if the plug-in does not specify the default life cycle it should run, you must also specify the life cycle phase it should run.

<plugin>

<groupId>com.vogella.example</groupId>

<artifactId>vogella-some-maven-plugin</artifactId>

<version>1.0</version>

<executions>

<execution>

<phase>verify</phase>

<goals>

<goal>checklinks</goal>

</goals>

</execution>

</executions>

</plugin>

## Maven repositories

If you run a Maven build, Maven validates if you still have the most recent version of all required artifact dependencies and Maven plug-ins and if required it retrieves them from a Maven repository. A repository is a collection of project artifacts stored in a directory structure similar to the Maven coordinates of the project.

Maven downloads these artifacts and plug-in into a local repository. The local repository is located in the home directory of the user in the *.m2/repository* folder. If an artifact or a plug-in is available in the local repository Maven uses it from their.

Maven uses a default remote repository location (http://repo1.maven.org/maven2) from which it downloads the core Maven plugins and dependencies. To can configure Maven to use more repositories and replace the default one.

Every project can define dependencies using the unique identifier (GAV) of the component it required to compile.

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

During a build, the Maven system tries to resolve the dependencies of the modules which are build. To resolve dependencies, Maven uses the following sources in the given order:

* Project which are included in the same Maven run
* local repository
* Maven central repository

The projects which are included in the same Maven run are also called the Maven *reactor*.

### Multi module projects (Aggregator)

Maven supports handling multiple projects. A multi module project (aggregator) is defined by a parent POM referencing one or more projects. This aggregator can contain also the build configuration or include another parent POM to get this configuration.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.vogella.tychoexample</groupId>

<artifactId>com.vogella.tycho.aggregator</artifactId>

<version>1.0.0-SNAPSHOT</version>

<packaging>pom</packaging>

<parent>

<groupId>com.vogella.tychoexample</groupId>

<artifactId>com.vogella.tycho.parent</artifactId>

<version>1.0.0-SNAPSHOT</version>

<relativePath>../com.vogella.tycho.parent</relativePath>

</parent>

<modules>

<module>../com.vogella.build.targetdefinition</module>

<module>../com.vogella.tycho.plugin1</module>

<module>../com.vogella.tycho.plugin2</module>

<module>../com.vogella.tycho.rcp</module>

<module>../com.vogella.tycho.product</module>

<module>../com.vogella.tycho.feature</module>

<module>../com.vogella.tycho.update</module>

<module>../com.vogella.tycho.unittests</module>

</modules>

</project>

### Profiles

Maven supports the usage of profiles to define different configurations. If you start Maven you can instruct it to use a certain profile, i.e., on the command line via the *-P profilename* parameter.

<profiles>

<profile>

<id>dev</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<db.location>URL\_to\_dev\_system</db.location>

<logo.image>companylogo.png</logo.image>

</properties>

</profile>

<profile>

<id>production</id>

<properties>

<db.location>URL\_to\_prod\_system</db.location>

<logo.image>companylogo2.png</logo.image>

</properties>

</profile>

</profiles>

### Properties

You can specify properties in your build files and override them in on the command line. For example you can specify in your pom file that the test should be skipped during the build. This property and others are defined in the following snippet.

<properties>

<skipTests>true</skipTests>

<maven.build.timestamp.format>yyyyMMdd-HHmm</maven.build.timestamp.format>

<buildTimestamp>${maven.build.timestamp}</buildTimestamp>

<buildId>${buildType}${buildTimestamp}</buildId>

</properties>

On the command line you can override such parameters and demonstrated in the following listing.

mvn clean install -DskipTests=false

## Maven and version control systems

Maven generates its output into the *target* folder of each project. This build output should not get included into your version control system.

Add this directory to your ignore resources, e.g., in case you are using Git as version control system, add the "target/" entry to your*.gitignore* file in the root of each project.

### Scaffolding a project with Maven

Maven supports project scaffolding, based on project templates (called archtype) via the archetype plug-in. A number of archetypes are available in Maven for anything from a simple application to a complex web application,

The goal of this scaffolding is to allow a fast start into the Maven world and supports a "standardized" folder structure of software projects.

You can create a project by executing the generation goal on the archetype plugin via the following command:mvn archetype:generate.

This starts the generation process in the interactive mode and asks you for several settings.

## Maven on the command line

Maven provides a command line tool. To use this tooling, switch to the command line into a directory which contains a pom.xml.

To run a build with Maven, you just have to point your command line to the folder which includes the *pom.xml* file and run the command mvn followed by a life cycle phase or goal to execute. .

Maven reads the POM file and tries to resolve the dependencies of the project. Maven validates if required components are available in a local repository, which is placed in the *.m2/repository* folder of the users home directory. If not Maven tries to download the latest version of the depended artifacts from the central repository into the local repository. Maven executes all life cycle phases until it finishes the "install" phase. This trigger the registered goals, for the jar packaging type this includes compiling the sources, executing the tests and packaging the compiled files in a JAR file. As last step the resulting artifact is saved in the local repository, so it can be used by other projects.

Maven creates the build result in the *target* folder.

# compiles, build and install the build result

maven install

To ensure that the build target is removed before a new build, add the *clean* target.

# remove previous build results

# compiles, build and install the build result

maven clean install

By default, Maven checks online if the dependencies have been changed. If you want to use your local repository, you can use the*-o* to tell Maven to work offline.

# work offline , i.e. use the local maven repository

maven -o clean install

## Exercise: execute a Java program with Maven

If you want to execute a program you can use the exec-maven-plugin as demonstrated in the following *pom.xml* file. To trigger this use the exec:java target in maven.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.vogella.build.maven.intro</groupId>

<artifactId>com.vogella.build.maven.intro</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>mavenintroduction</name>

<build>

<sourceDirectory>src</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>exec-maven-plugin</artifactId>

<version>1.2.1</version>

<configuration>

<mainClass>com.vogella.build.maven.intro.Main</mainClass>

</configuration>

</plugin>

</plugins>

</build>

</project>

### Creating a Project

mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany.app</groupId>

<artifactId>my-app</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>Maven Quick Start Archetype</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.8.2</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

You executed the Maven goal ***archetype****:****generate***, and passed in various parameters to that goal. The prefix *archetype* is the [**plugin**](http://maven.apache.org/plugins/index.html) that contains the goal. If you are familiar with [**Ant**](http://ant.apache.org/), you may conceive of this as similar to a task. This goal created a simple project based upon an archetype. Suffice it to say for now that a ***plugin*** is a **collection** of ***goals*** with a general common purpose. For example the jboss-maven-plugin, whose purpose is "deal with various jboss items".

mvn package

Unlike the first command executed (*archetype:generate*) you may notice the second is simply a single word - *package*. Rather than a goal, this is a *phase*. A phase is a step in the [build lifecycle](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html), which is an ordered sequence of phases. When a phase is given, Maven will execute every phase in the sequence up to and including the one defined. For example, if we execute the *compile* phase, the phases that actually get executed are:

1. validate
2. generate-sources
3. process-sources
4. generate-resources
5. process-resources
6. compile

You may test the newly compiled and packaged JAR with the following command:

java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App

Which will print the quintessential:

Hello World!

Phases are actually mapped to underlying goals. The specific goals executed per phase is dependant upon the packaging type of the project. For example, *package* executes *jar:jar* if the project type is a JAR, and *war:war* if the project type is - you guessed it - a WAR.

An interesting thing to note is that phases and goals may be executed in sequence.

mvn clean dependency:copy-dependencies package

This command will clean the project, copy dependencies, and package the project (executing all phases up to *package*, of course).

Internal Working of Concurrent collections

# ArrayList

ArrayList works on the principle of creating a array and adding elements to it.  
ArrayList class has a member variable elementData which is a Object array;  
**Object[] elementData;**  
When we do List l = new ArrayList(); the array **elementData** is initalized with a size of **10**  
  
**add(E element)**  
When a new element is added the capacity of the array elementData is checked and if it is completely filled that is all element 10 are filled a new array is created with a new capacity by using Arrays.copyOf. If the elementData array is not exhausted the new element is added in the array.  
So adding a element in a array may take more time as a completely new array needs to be created with greater capacity and the data in the old array is transferred into the new array.

**add(index i, E element)**  
On adding a element at a particular index in ArrayList, ArrayList checks if a element is already present at that index. If no than the element passed in add() is added at that index, otherwise a new array is created with the index kept vacant and the remaining element shifted to right.

**If we create a arraylist and call set method, we get index out of bound.**

Next resize size = (3/2 \* initial size) + 1

/\*\*

\* Resizable-array implementation of the List interface. Implements

\* all optional list operations, and permits all elements, including

\* null. In addition to implementing the List interface,

\* this class provides methods to manipulate the size of the array that is

\* used internally to store the list. (This class is roughly equivalent to

\* Vector, except that it is unsynchronized.)

\*

\* <p>The size, isEmpty, get, set,

\* iterator, and listIterator operations run in constant

\* time. The add operation runs in <i>amortized constant time</i>,

\* that is, adding n elements requires O(n) time. All of the other operations

\* run in linear time (roughly speaking). The constant factor is low compared

\* to that for the LinkedList implementation.

\*

\* <p>Each ArrayList instance has a <i>capacity</i>. 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.

\*

\* <p>An application can increase the capacity of an ArrayList instance

\* before adding a large number of elements using the ensureCapacity

\* operation. This may reduce the amount of incremental reallocation.

\*

\* <p><strong>Note that this implementation is not synchronized.</strong>

\* If multiple threads access an ArrayList instance concurrently,

\* and at least one of the threads modifies the list structurally, it

\* <i>must</i> be synchronized externally. (A structural modification is

\* any operation that adds or deletes one or more elements, or explicitly

\* resizes the backing array; merely setting the value of an element is not

\* a structural modification.) This is typically accomplished by

\* synchronizing on some object that naturally encapsulates the list.

\*

\* If no such object exists, the list should be "wrapped" using the

\* {@link Collections#synchronizedList Collections.synchronizedList}

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

\* unsynchronized access to the list:<pre>

\* List list = Collections.synchronizedList(new ArrayList(...));</pre>

\*

\* <p><a name="fail-fast"/>

\* The iterators returned by this class's {@link #iterator() iterator} and

\* {@link #listIterator(int) listIterator} methods are <em>fail-fast</em>:

\* if the list is structurally modified at any time after the iterator is

\* created, in any way except through the iterator's own

\* {@link ListIterator#remove() remove} or

\* {@link ListIterator#add(Object) 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.

\*

\* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed

\* as it is, generally speaking, impossible to make any hard guarantees in the

\* presence of unsynchronized concurrent modification. Fail-fast iterators

\* throw {@code ConcurrentModificationException} on a best-effort basis.

\* Therefore, it would be wrong to write a program that depended on this

\* exception for its correctness: <i>the fail-fast behavior of iterators

\* should be used only to detect bugs.</i>

\*

\* <p>This class is a member of the

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

\* Java Collections Framework</a>.

\*/

/\*\*

\* Constructs an empty list with the specified initial capacity.

\*

\* **@param** initialCapacity the initial capacity of the list

\* **@throws** IllegalArgumentException if the specified initial capacity

\* is negative

\*/

**public** ArrayList(**int** initialCapacity) {

**super**();

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

**this**.elementData = **new** Object[initialCapacity];

}

/\*\*

\* Constructs an empty list with an initial capacity of ten.

\*/

**public** ArrayList() {

**this**(10);

}

/\*\*

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

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

174 \* specified by the minimum capacity argument.

175 \*

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

177 \*/

**private** **void** ensureCapacityInternal(**int** minCapacity) {

modCount++;

// overflow-conscious code

**if** (minCapacity - elementData.length > 0)

grow(minCapacity);

}

/\*\*

375 \* Returns the element at the specified position in this list.

376 \*

377 \* **@param** index index of the element to return

378 \* **@return** the element at the specified position in this list

379 \* **@throws** IndexOutOfBoundsException {@inheritDoc}

380 \*/

**public** E get(**int** index) {

rangeCheck(index);

**return** elementData(index);

}

/\*\*

388 \* Replaces the element at the specified position in this list with

389 \* the specified element.

390 \*

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

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

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

394 \* **@throws** IndexOutOfBoundsException {@inheritDoc}

395 \*/

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

rangeCheck(index);

E oldValue = elementData(index);

elementData[index] = element;

**return** oldValue;

}

/\*\*

405 \* Appends the specified element to the end of this list.

406 \*

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

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

409 \*/

**public** **boolean** add(E e) {

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

elementData[size++] = e;

**return** **true**;

}

/\*\*

417 \* Inserts the specified element at the specified position in this

418 \* list. Shifts the element currently at that position (if any) and

419 \* any subsequent elements to the right (adds one to their indices).

420 \*

421 \* **@param** index index at which the specified element is to be inserted

422 \* **@param** element element to be inserted

423 \* **@throws** IndexOutOfBoundsException {@inheritDoc}

424 \*/

**public** **void** add(**int** index, E element) {

rangeCheckForAdd(index);

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

System.*arraycopy*(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

/\*\*

597 \* Checks if the given index is in range. If not, throws an appropriate

598 \* runtime exception. This method does \*not\* check if the index is

599 \* negative: It is always used immediately prior to an array access,

600 \* which throws an ArrayIndexOutOfBoundsException if index is negative.

601 \*/

**private** **void** rangeCheck(**int** index) {

**if** (index >= size)

**throw** **new** IndexOutOfBoundsException(outOfBoundsMsg(index));

}

/\*\*

608 \* A version of rangeCheck used by add and addAll.

609 \*/

**private** **void** rangeCheckForAdd(**int** index) {

**if** (index > size || index < 0)

**throw** **new** IndexOutOfBoundsException(outOfBoundsMsg(index));

}

# HashSet

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**Note that this implementation is not synchronized.** If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

Set s = Collections.synchronizedSet(new HashSet(...));

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a [ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework

**public** **class** HashSet<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**static** **final** **long** *serialVersionUID* = -5024744406713321676L;

**private** **transient** HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object *PRESENT* = **new** Object();

/\*\*

\* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

\* default initial capacity (16) and load factor (0.75).

\*/

**public** HashSet() {

map = **new** HashMap<E,Object>();

}

/\*\*

\* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

\* the specified initial capacity and default load factor (0.75).

\*

\* **@param** initialCapacity the initial capacity of the hash table

\* **@throws** IllegalArgumentException if the initial capacity is less

\* than zero

\*/

**public** HashSet(**int** initialCapacity) {

map = **new** HashMap<E,Object>(initialCapacity);

}

/\*\*

\* Constructs a new, empty linked hash set. (This package private

\* constructor is only used by LinkedHashSet.) The backing

\* HashMap instance is a LinkedHashMap with the specified initial

\* capacity and the specified load factor.

\*

\* **@param** initialCapacity the initial capacity of the hash map

\* **@param** loadFactor the load factor of the hash map

\* **@param** dummy ignored (distinguishes this

\* constructor from other int, float constructor.)

\* **@throws** IllegalArgumentException if the initial capacity is less

\* than zero, or if the load factor is nonpositive

\*/

HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy) {

map = **new** LinkedHashMap<E,Object>(initialCapacity, loadFactor);

}

/\*\*

\* Returns <tt>true</tt> if this set contains the specified element.

\* More formally, returns <tt>true</tt> if and only if this set

\* contains an element <tt>e</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;e==null&nbsp;:&nbsp;o.equals(e))</tt>.

\*

\* **@param** o element whose presence in this set is to be tested

\* **@return** <tt>true</tt> if this set contains the specified element

\*/

**public** **boolean** contains(Object o) {

**return** map.containsKey(o);

}

/\*\*

\* Adds the specified element to this set if it is not already present.

\* More formally, adds the specified element <tt>e</tt> to this set if

\* this set contains no element <tt>e2</tt> such that

\* <tt>(e==null&nbsp;?&nbsp;e2==null&nbsp;:&nbsp;e.equals(e2))</tt>.

\* If this set already contains the element, the call leaves the set

\* unchanged and returns <tt>false</tt>.

\*

\* **@param** e element to be added to this set

\* **@return** <tt>true</tt> if this set did not already contain the specified

\* element

\*/

**public** **boolean** add(E e) {

**return** map.put(e, PRESENT)==**null**;

}

}

# TreeMap

A Red-Black tree based [NavigableMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/NavigableMap.java#NavigableMap) implementation. The map is sorted according to the natural ordering of its keys, or by a [Comparator](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Comparator.java#Comparator) provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be *consistent with equals* if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of *consistent with equals*.) This is so because the Map interface is defined in terms of the equals operation, but a map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

**Note that this implementation is not synchronized.** If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with an existing key is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using theCollections.synchronizedSortedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do *not* support the Entry.setValue method. (Note however that it is possible to change mappings in the associated map using put.)

This class is a member of the Java Collections Framework.

**public** **class** TreeMap<K,V>

**extends** AbstractMap<K,V>

**implements** NavigableMap<K,V>, Cloneable, java.io.Serializable

{

/\*\*

\* The comparator used to maintain order in this tree map, or

\* null if it uses the natural ordering of its keys.

\*

\* **@serial**

\*/

**private** **final** Comparator<? **super** K> comparator;

**private** **transient** Entry<K,V> root = **null**;

/\*\*

\* The number of entries in the tree

\*/

**private** **transient** **int** size = 0;

/\*\*

\* The number of structural modifications to the tree.

\*/

**private** **transient** **int** modCount = 0;

/\*\*

\* Constructs a new, empty tree map, using the natural ordering of its

\* keys. All keys inserted into the map must implement the {@link

\* Comparable} interface. Furthermore, all such keys must be

\* <i>mutually comparable</i>: <tt>k1.compareTo(k2)</tt> must not throw

\* a <tt>ClassCastException</tt> for any keys <tt>k1</tt> and

\* <tt>k2</tt> in the map. If the user attempts to put a key into the

\* map that violates this constraint (for example, the user attempts to

\* put a string key into a map whose keys are integers), the

\* <tt>put(Object key, Object value)</tt> call will throw a

\* <tt>ClassCastException</tt>.

\*/

**public** TreeMap() {

comparator = **null**;

}

/\*\*

\* Constructs a new, empty tree map, ordered according to the given

\* comparator. All keys inserted into the map must be <i>mutually

\* comparable</i> by the given comparator: <tt>comparator.compare(k1,

\* k2)</tt> must not throw a <tt>ClassCastException</tt> for any keys

\* <tt>k1</tt> and <tt>k2</tt> in the map. If the user attempts to put

\* a key into the map that violates this constraint, the <tt>put(Object

\* key, Object value)</tt> call will throw a

\* <tt>ClassCastException</tt>.

\*

\* **@param** comparator the comparator that will be used to order this map.

\* If <tt>null</tt>, the {@linkplain Comparable natural

\* ordering} of the keys will be used.

\*/

**public** TreeMap(Comparator<? **super** K> comparator) {

**this**.comparator = comparator;

}

/\*\*

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

\* key.

\*

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

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

\* specified key

\* **@throws** ClassCastException if the specified key cannot be compared

\* with the keys currently in the map

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

\* and this map uses natural ordering, or its comparator

\* does not permit null keys

\*/

**public** **boolean** containsKey(Object key) {

**return** getEntry(key) != **null**;

}

/\*\*

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

\* specified value. More formally, returns <tt>true</tt> if and only if

\* this map contains at least one mapping to a value <tt>v</tt> such

\* that <tt>(value==null ? v==null : value.equals(v))</tt>. This

\* operation will probably require time linear in the map size for

\* most implementations.

\*

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

\* **@return** <tt>true</tt> if a mapping to <tt>value</tt> exists;

\* <tt>false</tt> otherwise

\* **@since** 1.2

\*/

**public** **boolean** containsValue(Object value) {

**for** (Entry<K,V> e = getFirstEntry(); e != **null**; e = successor(e))

**if** (valEquals(value, e.value))

**return** **true**;

**return** **false**;

}

/\*\*

\* 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} compares

\* equal to {@code k} according to the map's ordering, 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.

\*

\* **@throws** ClassCastException if the specified key cannot be compared

\* with the keys currently in the map

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

\* and this map uses natural ordering, or its comparator

\* does not permit null keys

\*/

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

Entry<K,V> p = getEntry(key);

**return** (p==**null** ? **null** : p.value);

}

**public** Comparator<? **super** K> comparator() {

**return** comparator;

}

/\*\*

\* Returns this map's entry for the given key, or <tt>null</tt> if the map

\* does not contain an entry for the key.

\*

\* **@return** this map's entry for the given key, or <tt>null</tt> if the map

\* does not contain an entry for the key

\* **@throws** ClassCastException if the specified key cannot be compared

\* with the keys currently in the map

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

\* and this map uses natural ordering, or its comparator

\* does not permit null keys

\*/

**final** Entry<K,V> getEntry(Object key) {

// Offload comparator-based version for sake of performance

**if** (comparator != **null**)

**return** getEntryUsingComparator(key);

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

**throw** **new** NullPointerException();

Comparable<? **super** K> k = (Comparable<? **super** K>) key;

Entry<K,V> p = root;

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

**int** cmp = k.compareTo(p.key);

**if** (cmp < 0)

p = p.left;

**else** **if** (cmp > 0)

p = p.right;

**else**

**return** p;

}

**return** **null**;

}

/\*\*

\* Version of getEntry using comparator. Split off from getEntry

\* for performance. (This is not worth doing for most methods,

\* that are less dependent on comparator performance, but is

\* worthwhile here.)

\*/

**final** Entry<K,V> getEntryUsingComparator(Object key) {

K k = (K) key;

Comparator<? **super** K> cpr = comparator;

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

Entry<K,V> p = root;

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

**int** cmp = cpr.compare(k, p.key);

**if** (cmp < 0)

p = p.left;

**else** **if** (cmp > 0)

p = p.right;

**else**

**return** p;

}

}

**return** **null**;

}

/\*\*

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

\* If the map previously contained a mapping for the key, the old

\* value is replaced.

\*

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

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

\*

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\* **@throws** ClassCastException if the specified key cannot be compared

\* with the keys currently in the map

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

\* and this map uses natural ordering, or its comparator

\* does not permit null keys

\*/

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

Entry<K,V> t = root;

**if** (t == **null**) {

// TBD:

// 5045147: (coll) Adding null to an empty TreeSet should

// throw NullPointerException

//

// compare(key, key); // type check

root = **new** Entry<K,V>(key, value, **null**);

size = 1;

modCount++;

**return** **null**;

}

**int** cmp;

Entry<K,V> parent;

// split comparator and comparable paths

Comparator<? **super** K> cpr = comparator;

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

**do** {

parent = t;

cmp = cpr.compare(key, t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

}

**else** {

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

**throw** **new** NullPointerException();

Comparable<? **super** K> k = (Comparable<? **super** K>) key;

**do** {

parent = t;

cmp = k.compareTo(t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

}

Entry<K,V> e = **new** Entry<K,V>(key, value, parent);

**if** (cmp < 0)

parent.left = e;

**else**

parent.right = e;

fixAfterInsertion(e);

size++;

modCount++;

**return** **null**;

}

/\*\*

\* Removes the mapping for this key from this TreeMap if present.

\*

\* **@param** key key for which mapping should be removed

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\* **@throws** ClassCastException if the specified key cannot be compared

\* with the keys currently in the map

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

\* and this map uses natural ordering, or its comparator

\* does not permit null keys

\*/

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

Entry<K,V> p = getEntry(key);

**if** (p == **null**)

**return** **null**;

V oldValue = p.value;

deleteEntry(p);

**return** oldValue;

}

/\*\*

\* Fields initialized to contain an instance of the entry set view

\* the first time this view is requested. Views are stateless, so

\* there's no reason to create more than one.

\*/

**private** **transient** EntrySet entrySet = **null**;

**private** **transient** KeySet<K> navigableKeySet = **null**;

**private** **transient** NavigableMap<K,V> descendingMap = **null**;

/\*\*

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

\* The set's iterator returns the keys in ascending order.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation), the results of

\* the iteration are undefined. The set supports element removal,

\* which removes the corresponding mapping from the map, via the

\* <tt>Iterator.remove</tt>, <tt>Set.remove</tt>,

\* <tt>removeAll</tt>, <tt>retainAll</tt>, and <tt>clear</tt>

\* operations. It does not support the <tt>add</tt> or <tt>addAll</tt>

\* operations.

\*/

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

**return** navigableKeySet();

}

/\*\*

\* **@since** 1.6

\*/

**public** NavigableSet<K> navigableKeySet() {

KeySet<K> nks = navigableKeySet;

**return** (nks != **null**) ? nks : (navigableKeySet = **new** KeySet(**this**));

}

/\*\*

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

\* The collection's iterator returns the values in ascending order

\* of the corresponding keys.

\* The collection is backed by the map, so changes to the map are

\* reflected in the collection, and vice-versa. If the map is

\* modified while an iteration over the collection is in progress

\* (except through the iterator's own <tt>remove</tt> operation),

\* the results of the iteration are undefined. The collection

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Collection.remove</tt>, <tt>removeAll</tt>,

\* <tt>retainAll</tt> and <tt>clear</tt> operations. It does not

\* support the <tt>add</tt> or <tt>addAll</tt> operations.

\*/

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

Collection<V> vs = values;

**return** (vs != **null**) ? vs : (values = **new** Values());

}

/\*\*

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

\* The set's iterator returns the entries in ascending key order.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation, or through the

\* <tt>setValue</tt> operation on a map entry returned by the

\* iterator) the results of the iteration are undefined. The set

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Set.remove</tt>, <tt>removeAll</tt>, <tt>retainAll</tt> and

\* <tt>clear</tt> operations. It does not support the

\* <tt>add</tt> or <tt>addAll</tt> operations.

\*/

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

EntrySet es = entrySet;

**return** (es != **null**) ? es : (entrySet = **new** EntrySet());

}

// Red-black mechanics

**private** **static** **final** **boolean** *RED* = **false**;

**private** **static** **final** **boolean** *BLACK* = **true**;

/\*\*

\* Node in the Tree. Doubles as a means to pass key-value pairs back to

\* user (see Map.Entry).

\*/

**static** **final** **class** Entry<K,V> **implements** Map.Entry<K,V> {

K key;

V value;

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

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

Entry<K,V> parent;

**boolean** color = *BLACK*;

/\*\*

\* Make a new cell with given key, value, and parent, and with

\* <tt>null</tt> child links, and BLACK color.

\*/

Entry(K key, V value, Entry<K,V> parent) {

**this**.key = key;

**this**.value = value;

**this**.parent = parent;

}

/\*\*

\* Returns the key.

\*

\* **@return** the key

\*/

**public** K getKey() {

**return** key;

}

/\*\*

\* Returns the value associated with the key.

\*

\* **@return** the value associated with the key

\*/

**public** V getValue() {

**return** value;

}

/\*\*

\* Replaces the value currently associated with the key with the given

\* value.

\*

\* **@return** the value associated with the key before this method was

\* called

\*/

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

V oldValue = **this**.value;

**this**.value = value;

**return** oldValue;

}

**public** **boolean** equals(Object o) {

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

**return** **false**;

Map.Entry<?,?> e = (Map.Entry<?,?>)o;

**return** valEquals(key,e.getKey()) && valEquals(value,e.getValue());

}

**public** **int** hashCode() {

**int** keyHash = (key==**null** ? 0 : key.hashCode());

**int** valueHash = (value==**null** ? 0 : value.hashCode());

**return** keyHash ^ valueHash;

}

**public** String toString() {

**return** key + "=" + value;

}

}

}

# HashMap

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits nullvalues and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

**Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

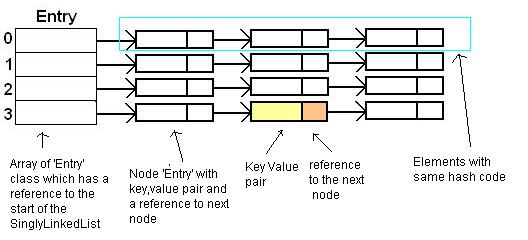
Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a[ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

**Working**

HashMap works on the principal of hashing. It stores values in the form of key,value pair and to access a value you need to provide the key.  
For efficient use of HashMap the 'key' element should implement equals() and hashcode() method. equals() method define that two objects are meaningfully equal. hashcode() helps HashMap to arrange elements separately in a bucket. So elements with same hascode are kept in the same bucket together.  
So when we want to fetch a element using **get(K key),**HashMap first identifies the bucket in which all elements of the same hascode as the hashcode of the 'key' passed are present. Than it uses the equals() method to identify the actual object present in the bucket.  
  
Lets see how HashMap implements this logic internally.  
  
[](http://1.bp.blogspot.com/-wlGPaH0RYHA/UhxiVVVKU4I/AAAAAAAAATU/maIrHVaeym0/s1600/HashMap.JPG)For fast access to a value HashMap places a element (both key and value) in a[SinglyLinkedList](http://www.java-redefined.com/2013/08/data-structure-singly-linked-list.html)(Bucket). All the elements that have the same hascode are placed in the same SinglyLinkedList. The number of SinglyLinkedList(buckets) depends upon how many objects are present with different hashcode. To hold these buckets together a array is used. The size of the array is initially defined to 12. And it changes as new elements with different hascodes are added. Lets see the pictorial view.  
The structure of the 'Entry' class used above.

class Entry {

K key;

V value;

Entry next;

int hash;

}

HashMap also has some more variables which define the initial size of the array.  
  
DEFAULT\_LOAD\_FACTOR = 0.75f;  
DEFAULT\_INITIAL\_CAPACITY = 16;  
  
Entry[] table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

**public** **class** HashMap<K,V>

**extends** AbstractMap<K,V>

**implements** Map<K,V>, Cloneable, Serializable

{

/\*\*

\* The default initial capacity - MUST be a power of two.

\*/

**static** **final** **int** *DEFAULT\_INITIAL\_CAPACITY* = 16;

/\*\*

\* The maximum capacity, used if a higher value is implicitly specified

\* by either of the constructors with arguments.

\* MUST be a power of two <= 1<<30.

\*/

**static** **final** **int** *MAXIMUM\_CAPACITY* = 1 << 30;

/\*\*

\* The load factor used when none specified in constructor.

\*/

**static** **final** **float** *DEFAULT\_LOAD\_FACTOR* = 0.75f;

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

**transient** Entry[] table;

/\*\*

\* The number of key-value mappings contained in this map.

\*/

**transient** **int** size;

/\*\*

\* The next size value at which to resize (capacity \* load factor).

\* **@serial**

\*/

**int** threshold;

/\*\*

\* The load factor for the hash table.

\*

\* **@serial**

\*/

**final** **float** loadFactor;

/\*\*

\* The number of times this HashMap has been structurally modified

\* Structural modifications are those that change the number of mappings in

\* the HashMap or otherwise modify its internal structure (e.g.,

\* rehash). This field is used to make iterators on Collection-views of

\* the HashMap fail-fast. (See ConcurrentModificationException).

\*/

**transient** **volatile** **int** modCount;

/\*\*

\* Initialization hook for subclasses. This method is called

\* in all constructors and pseudo-constructors (clone, readObject)

\* after HashMap has been initialized but before any entries have

\* been inserted. (In the absence of this method, readObject would

\* require explicit knowledge of subclasses.)

\*/

**void** init() {

}

/\*\*

\* Constructs an empty <tt>HashMap</tt> with the specified initial

\* capacity and load factor.

\*

\* **@param** initialCapacity the initial capacity

\* **@param** loadFactor the load factor

\* **@throws** IllegalArgumentException if the initial capacity is negative

\* or the load factor is nonpositive

\*/

**public** HashMap(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

**if** (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

**if** (loadFactor <= 0 || Float.isNaN(loadFactor))

**throw** **new** IllegalArgumentException("Illegal load factor: " +

loadFactor);

// Find a power of 2 >= initialCapacity

**int** capacity = 1;

**while** (capacity < initialCapacity)

capacity <<= 1;

**this**.loadFactor = loadFactor;

threshold = (**int**)(capacity \* loadFactor);

table = **new** Entry[capacity];

init();

}

/\*\*

\* Applies a supplemental hash function to a given hashCode, which

\* defends against poor quality hash functions. This is critical

\* because HashMap uses power-of-two length hash tables, that

\* otherwise encounter collisions for hashCodes that do not differ

\* in lower bits. Note: Null keys always map to hash 0, thus index 0.

\*/

**static** **int** hash(**int** h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

**return** h ^ (h >>> 7) ^ (h >>> 4);

}

/\*\*

\* Returns index for hash code h.

\*/

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

}

/\*\*

\* 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==null ? k==null :

\* key.equals(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.

\*

\* **@see** #put(Object, Object)

\*/

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

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

**return** getForNullKey();

**int** hash = *hash*(key.hashCode());

**for** (Entry<K,V> e = table[indexFor(hash, table.length)];

e != **null**;

e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))

**return** e.value;

}

**return** **null**;

}

/\*\*

\* Offloaded version of get() to look up null keys. Null keys map

\* to index 0. This null case is split out into separate methods

\* for the sake of performance in the two most commonly used

\* operations (get and put), but incorporated with conditionals in

\* others.

\*/

**private** V getForNullKey() {

**for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {

**if** (e.key == **null**)

**return** e.value;

}

**return** **null**;

}

/\*\*

\* 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.

\*/

**public** **boolean** containsKey(Object key) {

**return** getEntry(key) != **null**;

}

/\*\*

\* Returns the entry associated with the specified key in the

\* HashMap. Returns null if the HashMap contains no mapping

\* for the key.

\*/

**final** Entry<K,V> getEntry(Object key) {

**int** hash = (key == **null**) ? 0 : *hash*(key.hashCode());

**for** (Entry<K,V> e = table[indexFor(hash, table.length)];

e != **null**;

e = e.next) {

Object k;

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**return** e;

}

**return** **null**;

}

/\*\*

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

\* If the map previously contained a mapping for the key, the old

\* value is replaced.

\*

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

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

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\*/

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

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

**return** putForNullKey(value);

**int** hash = *hash*(key.hashCode());

**int** i = indexFor(hash, table.length);

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

**return** **null**;

}

/\*\*

\* Offloaded version of put for null keys

\*/

**private** V putForNullKey(V value) {

**for** (Entry<K,V> e = table[0]; e != **null**; e = e.next) {

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

V oldValue = e.value;

e.value = value;

e.recordAccess(**this**);

**return** oldValue;

}

}

modCount++;

addEntry(0, **null**, value, 0);

**return** **null**;

}

/\*\*

\* Rehashes the contents of this map into a new array with a

\* larger capacity. This method is called automatically when the

\* number of keys in this map reaches its threshold.

\*

\* If current capacity is MAXIMUM\_CAPACITY, this method does not

\* resize the map, but sets threshold to Integer.MAX\_VALUE.

\* This has the effect of preventing future calls.

\*

\* **@param** newCapacity the new capacity, MUST be a power of two;

\* must be greater than current capacity unless current

\* capacity is MAXIMUM\_CAPACITY (in which case value

\* is irrelevant).

\*/

**void** resize(**int** newCapacity) {

Entry[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity == *MAXIMUM\_CAPACITY*) {

threshold = Integer.*MAX\_VALUE*;

**return**;

}

Entry[] newTable = **new** Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (**int**)(newCapacity \* loadFactor);

}

/\*\*

\* Transfers all entries from current table to newTable.

\*/

**void** transfer(Entry[] newTable) {

Entry[] src = table;

**int** newCapacity = newTable.length;

**for** (**int** j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

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

src[j] = **null**;

**do** {

Entry<K,V> next = e.next;

**int** i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

} **while** (e != **null**);

}

}

}

/\*\*

\* 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 mappings to be stored in this map

\* **@throws** NullPointerException if the specified map is null

\*/

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

**int** numKeysToBeAdded = m.size();

**if** (numKeysToBeAdded == 0)

**return**;

/\*

\* Expand the map if the map if the number of mappings to be added

\* is greater than or equal to threshold. This is conservative; the

\* obvious condition is (m.size() + size) >= threshold, but this

\* condition could result in a map with twice the appropriate capacity,

\* if the keys to be added overlap with the keys already in this map.

\* By using the conservative calculation, we subject ourself

\* to at most one extra resize.

\*/

**if** (numKeysToBeAdded > threshold) {

**int** targetCapacity = (**int**)(numKeysToBeAdded / loadFactor + 1);

**if** (targetCapacity > *MAXIMUM\_CAPACITY*)

targetCapacity = *MAXIMUM\_CAPACITY*;

**int** newCapacity = table.length;

**while** (newCapacity < targetCapacity)

newCapacity <<= 1;

**if** (newCapacity > table.length)

resize(newCapacity);

}

**for** (Iterator<? **extends** Map.Entry<? **extends** K, ? **extends** V>> i = m.entrySet().iterator(); i.hasNext(); ) {

Map.Entry<? **extends** K, ? **extends** V> e = i.next();

put(e.getKey(), e.getValue());

}

}

/\*\*

\* Removes the mapping for the specified key from this map if present.

\*

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

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\*/

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

Entry<K,V> e = removeEntryForKey(key);

**return** (e == **null** ? **null** : e.value);

}

/\*\*

\* Removes and returns the entry associated with the specified key

\* in the HashMap. Returns null if the HashMap contains no mapping

\* for this key.

\*/

**final** Entry<K,V> removeEntryForKey(Object key) {

**int** hash = (key == **null**) ? 0 : *hash*(key.hashCode());

**int** i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

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

Entry<K,V> next = e.next;

Object k;

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k)))) {

modCount++;

size--;

**if** (prev == e)

table[i] = next;

**else**

prev.next = next;

e.recordRemoval(**this**);

**return** e;

}

prev = e;

e = next;

}

**return** e;

}

/\*\*

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

\* specified value.

\*

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

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

\* specified value

\*/

**public** **boolean** containsValue(Object value) {

**if** (value == **null**)

**return** containsNullValue();

Entry[] tab = table;

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

**for** (Entry e = tab[i] ; e != **null** ; e = e.next)

**if** (value.equals(e.value))

**return** **true**;

**return** **false**;

}

}

# Hashtable

This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).

The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming. No rehash operations will *ever* occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

Hashtable<String, Integer> numbers

= new Hashtable<String, Integer>();

numbers.put("one", 1);

numbers.put("two", 2);

numbers.put("three", 3);

To retrieve a number, use the following code:

Integer n = numbers.get("two");

if (n != null) {

System.out.println("two = " + n);

}

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are *fail-fast*: if the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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. The Enumerations returned by Hashtable's keys and elements methods are *not* fail-fast.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

As of the Java 2 platform v1.2, this class was retrofitted to implement the [Map](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Map.java#Map) interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Hashtable is synchronized.

**public** **class** Hashtable<K,V>

**extends** Dictionary<K,V>

**implements** Map<K,V>, Cloneable, java.io.Serializable {

/\*\*

\* The hash table data.

\*/

**private** **transient** Entry[] table;

/\*\*

\* The total number of entries in the hash table.

\*/

**private** **transient** **int** count;

/\*\*

\* The table is rehashed when its size exceeds this threshold. (The

\* value of this field is (int)(capacity \* loadFactor).)

\*

\* **@serial**

\*/

**private** **int** threshold;

/\*\*

\* The load factor for the hashtable.

\*

\* **@serial**

\*/

**private** **float** loadFactor;

/\*\*

\* The number of times this Hashtable has been structurally modified

\* Structural modifications are those that change the number of entries in

\* the Hashtable or otherwise modify its internal structure (e.g.,

\* rehash). This field is used to make iterators on Collection-views of

\* the Hashtable fail-fast. (See ConcurrentModificationException).

\*/

**private** **transient** **int** modCount = 0;

/\*\*

\* Constructs a new, empty hashtable with the specified initial

\* capacity and the specified load factor.

\*

\* **@param** initialCapacity the initial capacity of the hashtable.

\* **@param** loadFactor the load factor of the hashtable.

\* **@exception** IllegalArgumentException if the initial capacity is less

\* than zero, or if the load factor is nonpositive.

\*/

**public** Hashtable(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

**if** (loadFactor <= 0 || Float.*isNaN*(loadFactor))

**throw** **new** IllegalArgumentException("Illegal Load: "+loadFactor);

**if** (initialCapacity==0)

initialCapacity = 1;

**this**.loadFactor = loadFactor;

table = **new** Entry[initialCapacity];

threshold = (**int**)(initialCapacity \* loadFactor);

}

/\*\*

\* Constructs a new, empty hashtable with a default initial capacity (11)

\* and load factor (0.75).

\*/

**public** Hashtable() {

**this**(11, 0.75f);

}

/\*\*

\* Returns an enumeration of the keys in this hashtable.

\*

\* **@return** an enumeration of the keys in this hashtable.

\* **@see** Enumeration

\* **@see** #elements()

\* **@see** #keySet()

\* **@see** Map

\*/

**public** **synchronized** Enumeration<K> keys() {

**return** **this**.<K>getEnumeration(KEYS);

}

/\*\*

\* Returns an enumeration of the values in this hashtable.

\* Use the Enumeration methods on the returned object to fetch the elements

\* sequentially.

\*

\* **@return** an enumeration of the values in this hashtable.

\* **@see** java.util.Enumeration

\* **@see** #keys()

\* **@see** #values()

\* **@see** Map

\*/

**public** **synchronized** Enumeration<V> elements() {

**return** **this**.<V>getEnumeration(VALUES);

}

/\*\*

\* Tests if some key maps into the specified value in this hashtable.

\* This operation is more expensive than the {@link #containsKey

\* containsKey} method.

\*

\* <p>Note that this method is identical in functionality to

\* {@link #containsValue containsValue}, (which is part of the

\* {@link Map} interface in the collections framework).

\*

\* **@param** value a value to search for

\* **@return** <code>true</code> if and only if some key maps to the

\* <code>value</code> argument in this hashtable as

\* determined by the <tt>equals</tt> method;

\* <code>false</code> otherwise.

\* **@exception** NullPointerException if the value is <code>null</code>

\*/

**public** **synchronized** **boolean** contains(Object value) {

**if** (value == **null**) {

**throw** **new** NullPointerException();

}

Entry tab[] = table;

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

**for** (Entry<K,V> e = tab[i] ; e != **null** ; e = e.next) {

**if** (e.value.equals(value)) {

**return** **true**;

}

}

}

**return** **false**;

}

/\*\*

\* Returns true if this hashtable maps one or more keys to this value.

\*

\* <p>Note that this method is identical in functionality to {@link

\* #contains contains} (which predates the {@link Map} interface).

\*

\* **@param** value value whose presence in this hashtable is to be tested

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

\* specified value

\* **@throws** NullPointerException if the value is <code>null</code>

\* **@since** 1.2

\*/

**public** **boolean** containsValue(Object value) {

**return** contains(value);

}

/\*\*

\* Tests if the specified object is a key in this hashtable.

\*

\* **@param** key possible key

\* **@return** <code>true</code> if and only if the specified object

\* is a key in this hashtable, as determined by the

\* <tt>equals</tt> method; <code>false</code> otherwise.

\* **@throws** NullPointerException if the key is <code>null</code>

\* **@see** #contains(Object)

\*/

**public** **synchronized** **boolean** containsKey(Object key) {

Entry tab[] = table;

**int** hash = key.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {

**if** ((e.hash == hash) && e.key.equals(key)) {

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* 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.equals(k))},

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

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

\*

\* **@param** key the key whose associated value is to be returned

\* **@return** the value to which the specified key is mapped, or

\* {@code null} if this map contains no mapping for the key

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

\* **@see** #put(Object, Object)

\*/

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

Entry tab[] = table;

**int** hash = key.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {

**if** ((e.hash == hash) && e.key.equals(key)) {

**return** e.value;

}

}

**return** **null**;

}

/\*\*

\* Increases the capacity of and internally reorganizes this

\* hashtable, in order to accommodate and access its entries more

\* efficiently. This method is called automatically when the

\* number of keys in the hashtable exceeds this hashtable's capacity

\* and load factor.

\*/

**protected** **void** rehash() {

**int** oldCapacity = table.length;

Entry[] oldMap = table;

**int** newCapacity = oldCapacity \* 2 + 1;

Entry[] newMap = **new** Entry[newCapacity];

modCount++;

threshold = (**int**)(newCapacity \* loadFactor);

table = newMap;

**for** (**int** i = oldCapacity ; i-- > 0 ;) {

**for** (Entry<K,V> old = oldMap[i] ; old != **null** ; ) {

Entry<K,V> e = old;

old = old.next;

**int** index = (e.hash & 0x7FFFFFFF) % newCapacity;

e.next = newMap[index];

newMap[index] = e;

}

}

}

/\*\*

\* Maps the specified <code>key</code> to the specified

\* <code>value</code> in this hashtable. Neither the key nor the

\* value can be <code>null</code>. <p>

\*

\* The value can be retrieved by calling the <code>get</code> method

\* with a key that is equal to the original key.

\*

\* **@param** key the hashtable key

\* **@param** value the value

\* **@return** the previous value of the specified key in this hashtable,

\* or <code>null</code> if it did not have one

\* **@exception** NullPointerException if the key or value is

\* <code>null</code>

\* **@see** Object#equals(Object)

\* **@see** #get(Object)

\*/

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

// Make sure the value is not null

**if** (value == **null**) {

**throw** **new** NullPointerException();

}

// Makes sure the key is not already in the hashtable.

Entry tab[] = table;

**int** hash = key.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entry<K,V> e = tab[index] ; e != **null** ; e = e.next) {

**if** ((e.hash == hash) && e.key.equals(key)) {

V old = e.value;

e.value = value;

**return** old;

}

}

modCount++;

**if** (count >= threshold) {

// Rehash the table if the threshold is exceeded

rehash();

tab = table;

index = (hash & 0x7FFFFFFF) % tab.length;

}

// Creates the new entry.

Entry<K,V> e = tab[index];

tab[index] = **new** Entry<K,V>(hash, key, value, e);

count++;

**return** **null**;

}

/\*\*

\* Removes the key (and its corresponding value) from this

\* hashtable. This method does nothing if the key is not in the hashtable.

\*

\* **@param** key the key that needs to be removed

\* **@return** the value to which the key had been mapped in this hashtable,

\* or <code>null</code> if the key did not have a mapping

\* **@throws** NullPointerException if the key is <code>null</code>

\*/

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

Entry tab[] = table;

**int** hash = key.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entry<K,V> e = tab[index], prev = **null** ; e != **null** ; prev = e, e = e.next) {

**if** ((e.hash == hash) && e.key.equals(key)) {

modCount++;

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

prev.next = e.next;

} **else** {

tab[index] = e.next;

}

count--;

V oldValue = e.value;

e.value = **null**;

**return** oldValue;

}

}

**return** **null**;

}

**private** <T> Enumeration<T> getEnumeration(**int** type) {

**if** (count == 0) {

**return** Collections.emptyEnumeration();

} **else** {

**return** **new** Enumerator<T>(type, **false**);

}

}

**private** <T> Iterator<T> getIterator(**int** type) {

**if** (count == 0) {

**return** Collections.emptyIterator();

} **else** {

**return** **new** Enumerator<T>(type, **true**);

}

}

/\*\*

\* Compares the specified Object with this Map for equality,

\* as per the definition in the Map interface.

\*

\* **@param** o object to be compared for equality with this hashtable

\* **@return** true if the specified Object is equal to this Map

\* **@see** Map#equals(Object)

\* **@since** 1.2

\*/

**public** **synchronized** **boolean** equals(Object o) {

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

**return** **true**;

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

**return** **false**;

Map<K,V> t = (Map<K,V>) o;

**if** (t.size() != size())

**return** **false**;

**try** {

Iterator<Map.Entry<K,V>> i = entrySet().iterator();

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

Map.Entry<K,V> e = i.next();

K key = e.getKey();

V value = e.getValue();

**if** (value == **null**) {

**if** (!(t.get(key)==**null** && t.containsKey(key)))

**return** **false**;

} **else** {

**if** (!value.equals(t.get(key)))

**return** **false**;

}

}

} **catch** (ClassCastException unused) {

**return** **false**;

} **catch** (NullPointerException unused) {

**return** **false**;

}

**return** **true**;

}

/\*\*

\* Returns the hash code value for this Map as per the definition in the

\* Map interface.

\*

\* **@see** Map#hashCode()

\* **@since** 1.2

\*/

**public** **synchronized** **int** hashCode() {

/\*

\* This code detects the recursion caused by computing the hash code

\* of a self-referential hash table and prevents the stack overflow

\* that would otherwise result. This allows certain 1.1-era

\* applets with self-referential hash tables to work. This code

\* abuses the loadFactor field to do double-duty as a hashCode

\* in progress flag, so as not to worsen the space performance.

\* A negative load factor indicates that hash code computation is

\* in progress.

\*/

**int** h = 0;

**if** (count == 0 || loadFactor < 0)

**return** h; // Returns zero

loadFactor = -loadFactor; // Mark hashCode computation in progress

Entry[] tab = table;

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

**for** (Entry e = tab[i]; e != **null**; e = e.next)

h += e.key.hashCode() ^ e.value.hashCode();

loadFactor = -loadFactor; // Mark hashCode computation complete

**return** h;

}

}

# LinkedList

Linked list implementation of the List interface. Implements all optional list operations, and permits all elements (including null). In addition to implementing the List interface, the LinkedList class 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, queue, or double-ended queue.

The class implements the Deque interface, providing first-in-first-out queue operations for add, poll, along with other stack and deque operations.

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

**Note that this implementation is not synchronized.** If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using theCollections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new LinkedList(...));

The iterators returned by this class's iterator and listIterator methods 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[ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

**Working**

Linked List is a Doubly Linked List as desribed here [Double Link List](http://www.java-redefined.com/2013/08/data-structure-doubly-link-list.html)  
With the Node called as **Entry** class having structure as

class Entry {

E element;

Entry next;

Entry previous;

}

LinkedList class also has a instance variable of type 'Entry' called 'header'. This is the 'start' element or node of the list.  
  
**add(E element )**  
 Every Time we call add(var);  a new instance of 'Entry' class is created and attached at the end of the list.  
   
**add(var,positon)**  
Inserts the specified element at the specified position in this list.  
Shifts the element currently at that position (if any) and any subsequent elements to the right.  
   
**get(int index)**  
It iterates through the list and returns the element. This is very expensive and time consuming as opposed to**ArraList.get(int index)**

**public** **class** LinkedList<E>

**extends** AbstractSequentialList<E>

**implements** List<E>, Deque<E>, Cloneable, java.io.Serializable

{

**private** **transient** Entry<E> header = **new** Entry<E>(**null**, **null**, **null**);

**private** **transient** **int** size = 0;

/\*\*

\* Constructs an empty list.

\*/

**public** LinkedList() {

header.next = header.previous = header;

}

/\*\*

\* Constructs a list containing the elements of the specified

\* collection, in the order they are returned by the collection's

\* iterator.

\*

\* **@param** c the collection whose elements are to be placed into this list

\* **@throws** NullPointerException if the specified collection is null

\*/

**public** LinkedList(Collection<? **extends** E> c) {

**this**();

addAll(c);

}

/\*\*

\* Returns the first element in this list.

\*

\* **@return** the first element in this list

\* **@throws** NoSuchElementException if this list is empty

\*/

**public** E getFirst() {

**if** (size==0)

**throw** **new** NoSuchElementException();

**return** header.next.element;

}

/\*\*

\* Returns the last element in this list.

\*

\* **@return** the last element in this list

\* **@throws** NoSuchElementException if this list is empty

\*/

**public** E getLast() {

**if** (size==0)

**throw** **new** NoSuchElementException();

**return** header.previous.element;

}

/\*\*

\* Returns <tt>true</tt> if this list contains the specified element.

\* More formally, returns <tt>true</tt> if and only if this list contains

\* at least one element <tt>e</tt> such that

\* <tt>(o==null&nbsp;?&nbsp;e==null&nbsp;:&nbsp;o.equals(e))</tt>.

\*

\* **@param** o element whose presence in this list is to be tested

\* **@return** <tt>true</tt> if this list contains the specified element

\*/

**public** **boolean** contains(Object o) {

**return** indexOf(o) != -1;

}

/\*\*

\* Returns the number of elements in this list.

\*

\* **@return** the number of elements in this list

\*/

**public** **int** size() {

**return** size;

}

/\*\*

\* Appends the specified element to the end of this list.

\*

\* <p>This method is equivalent to {@link #addLast}.

\*

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

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

\*/

**public** **boolean** add(E e) {

addBefore(e, header);

**return** **true**;

}

/\*\*

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

\* if it is present. If this 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).

\*

\* **@param** o element to be removed from this list, if present

\* **@return** <tt>true</tt> if this list contained the specified element

\*/

**public** **boolean** remove(Object o) {

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

**for** (Entry<E> e = header.next; e != header; e = e.next) {

**if** (e.element==**null**) {

remove(e);

**return** **true**;

}

}

} **else** {

**for** (Entry<E> e = header.next; e != header; e = e.next) {

**if** (o.equals(e.element)) {

remove(e);

**return** **true**;

}

}

}

**return** **false**;

}

// Positional Access Operations

/\*\*

\* Returns the element at the specified position in this list.

\*

\* **@param** index index of the element to return

\* **@return** the element at the specified position in this list

\* **@throws** IndexOutOfBoundsException {@inheritDoc}

\*/

**public** E get(**int** index) {

**return** entry(index).element;

}

/\*\*

\* Replaces the element at the specified position in this list with the

\* specified 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) {

Entry<E> e = entry(index);

E oldVal = e.element;

e.element = element;

**return** oldVal;

}

/\*\*

\* Inserts the specified element at the specified position in this list.

\* Shifts the element currently at that position (if any) and any

\* subsequent elements to the right (adds one to their indices).

\*

\* **@param** index index at which the specified element is to be inserted

\* **@param** element element to be inserted

\* **@throws** IndexOutOfBoundsException {@inheritDoc}

\*/

**public** **void** add(**int** index, E element) {

addBefore(element, (index==size ? header : entry(index)));

}

/\*\*

\* Removes the element at the specified position in this list. Shifts any

\* subsequent elements to the left (subtracts one from their indices).

\* Returns the element that was removed from the list.

\*

\* **@param** index the index of the element to be removed

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

\* **@throws** IndexOutOfBoundsException {@inheritDoc}

\*/

**public** E remove(**int** index) {

**return** remove(entry(index));

}

// Search Operations

/\*\*

\* Returns the index of the first occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\* More formally, returns the lowest index <tt>i</tt> such that

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

\* or -1 if there is no such index.

\*

\* **@param** o element to search for

\* **@return** the index of the first occurrence of the specified element in

\* this list, or -1 if this list does not contain the element

\*/

**public** **int** indexOf(Object o) {

**int** index = 0;

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

**for** (Entry e = header.next; e != header; e = e.next) {

**if** (e.element==**null**)

**return** index;

index++;

}

} **else** {

**for** (Entry e = header.next; e != header; e = e.next) {

**if** (o.equals(e.element))

**return** index;

index++;

}

}

**return** -1;

}

/\*\*

\* Returns the index of the last occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\* More formally, returns the highest index <tt>i</tt> such that

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

\* or -1 if there is no such index.

\*

\* **@param** o element to search for

\* **@return** the index of the last occurrence of the specified element in

\* this list, or -1 if this list does not contain the element

\*/

**public** **int** lastIndexOf(Object o) {

**int** index = size;

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

**for** (Entry e = header.previous; e != header; e = e.previous) {

index--;

**if** (e.element==**null**)

**return** index;

}

} **else** {

**for** (Entry e = header.previous; e != header; e = e.previous) {

index--;

**if** (o.equals(e.element))

**return** index;

}

}

**return** -1;

}

// Queue operations.

/\*\*

\* Retrieves, but does not remove, the head (first element) of this list.

\* **@return** the head of this list, or <tt>null</tt> if this list is empty

\* **@since** 1.5

\*/

**public** E peek() {

**if** (size==0)

**return** **null**;

**return** getFirst();

}

/\*\*

\* Retrieves, but does not remove, the head (first element) of this list.

\* **@return** the head of this list

\* **@throws** NoSuchElementException if this list is empty

\* **@since** 1.5

\*/

**public** E element() {

**return** getFirst();

}

/\*\*

\* Retrieves and removes the head (first element) of this list

\* **@return** the head of this list, or <tt>null</tt> if this list is empty

\* **@since** 1.5

\*/

**public** E poll() {

**if** (size==0)

**return** **null**;

**return** removeFirst();

}

/\*\*

\* Retrieves and removes the head (first element) of this list.

\*

\* **@return** the head of this list

\* **@throws** NoSuchElementException if this list is empty

\* **@since** 1.5

\*/

**public** E remove() {

**return** removeFirst();

}

/\*\*

\* Adds the specified element as the tail (last element) of this list.

\*

\* **@param** e the element to add

\* **@return** <tt>true</tt> (as specified by {@link Queue#offer})

\* **@since** 1.5

\*/

**public** **boolean** offer(E e) {

**return** add(e);

}

/\*\*

\* Pushes an element onto the stack represented by this list. In other

\* words, inserts the element at the front of this list.

\*

\* <p>This method is equivalent to {@link #addFirst}.

\*

\* **@param** e the element to push

\* **@since** 1.6

\*/

**public** **void** push(E e) {

addFirst(e);

}

/\*\*

\* Pops an element from the stack represented by this list. In other

\* words, removes and returns the first element of this list.

\*

\* <p>This method is equivalent to {@link #removeFirst()}.

\*

\* **@return** the element at the front of this list (which is the top

\* of the stack represented by this list)

\* **@throws** NoSuchElementException if this list is empty

\* **@since** 1.6

\*/

**public** E pop() {

**return** removeFirst();

}

/\*\*

\* Returns a list-iterator of the elements in this list (in proper

\* sequence), starting at the specified position in the list.

\* Obeys the general contract of <tt>List.listIterator(int)</tt>.<p>

\*

\* The list-iterator is <i>fail-fast</i>: if the list is structurally

\* modified at any time after the Iterator is created, in any way except

\* through the list-iterator's own <tt>remove</tt> or <tt>add</tt>

\* methods, the list-iterator will throw a

\* <tt>ConcurrentModificationException</tt>. 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.

\*

\* **@param** index index of the first element to be returned from the

\* list-iterator (by a call to <tt>next</tt>)

\* **@return** a ListIterator of the elements in this list (in proper

\* sequence), starting at the specified position in the list

\* **@throws** IndexOutOfBoundsException {@inheritDoc}

\* **@see** List#listIterator(int)

\*/

**public** ListIterator<E> listIterator(**int** index) {

**return** **new** ListItr(index);

}

/\*\*

\* **@since** 1.6

\*/

**public** Iterator<E> descendingIterator() {

**return** **new** DescendingIterator();

}

/\*\* Adapter to provide descending iterators via ListItr.previous \*/

**private** **class** DescendingIterator **implements** Iterator {

**final** ListItr itr = **new** ListItr(size());

**public** **boolean** hasNext() {

**return** itr.hasPrevious();

}

**public** E next() {

**return** itr.previous();

}

**public** **void** remove() {

itr.remove();

}

}

/\*\*

\* Returns an array containing all of the elements in this list

\* in proper sequence (from first to last element).

\*

\* <p>The returned array will be "safe" in that no references to it are

\* maintained by this list. (In other words, this method must allocate

\* a new array). The caller is thus free to modify the returned array.

\*

\* <p>This method acts as bridge between array-based and collection-based

\* APIs.

\*

\* **@return** an array containing all of the elements in this list

\* in proper sequence

\*/

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

Object[] result = **new** Object[size];

**int** i = 0;

**for** (Entry<E> e = header.next; e != header; e = e.next)

result[i++] = e.element;

**return** result;

}

/\*\*

\* Returns an array containing all of the elements in this list in

\* proper sequence (from first to last element); the runtime type of

\* the returned array is that of the specified array. If the list fits

\* in the specified array, it is returned therein. Otherwise, a new

\* array is allocated with the runtime type of the specified array and

\* the size of this list.

\*

\* <p>If the list fits in the specified array with room to spare (i.e.,

\* the array has more elements than the list), the element in the array

\* immediately following the end of the list is set to <tt>null</tt>.

\* (This is useful in determining the length of the list <i>only</i> if

\* the caller knows that the list does not contain any null elements.)

\*

\* <p>Like the {@link #toArray()} method, this method acts as bridge between

\* array-based and collection-based APIs. Further, this method allows

\* precise control over the runtime type of the output array, and may,

\* under certain circumstances, be used to save allocation costs.

\*

\* <p>Suppose <tt>x</tt> is a list known to contain only strings.

\* The following code can be used to dump the list into a newly

\* allocated array of <tt>String</tt>:

\*

\* <pre>

\* String[] y = x.toArray(new String[0]);</pre>

\*

\* Note that <tt>toArray(new Object[0])</tt> is identical in function to

\* <tt>toArray()</tt>.

\*

\* **@param** a the array into which the elements of the list are to

\* be stored, if it is big enough; otherwise, a new array of the

\* same runtime type is allocated for this purpose.

\* **@return** an array containing the elements of the list

\* **@throws** ArrayStoreException if the runtime type of the specified array

\* is not a supertype of the runtime type of every element in

\* this list

\* **@throws** NullPointerException if the specified array is null

\*/

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

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

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

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

**int** i = 0;

Object[] result = a;

**for** (Entry<E> e = header.next; e != header; e = e.next)

result[i++] = e.element;

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

a[size] = **null**;

**return** a;

}

**private** **class** ListItr **implements** ListIterator<E> {

**private** Entry<E> lastReturned = header;

**private** Entry<E> next;

**private** **int** nextIndex;

**private** **int** expectedModCount = modCount;

ListItr(**int** index) {

**if** (index < 0 || index > size)

**throw** **new** IndexOutOfBoundsException("Index: "+index+

", Size: "+size);

**if** (index < (size >> 1)) {

next = header.next;

**for** (nextIndex=0; nextIndex<index; nextIndex++)

next = next.next;

} **else** {

next = header;

**for** (nextIndex=size; nextIndex>index; nextIndex--)

next = next.previous;

}

}

}

**private** **static** **class** Entry<E> {

E element;

Entry<E> next;

Entry<E> previous;

Entry(E element, Entry<E> next, Entry<E> previous) {

**this**.element = element;

**this**.next = next;

**this**.previous = previous;

}

}

**private** Entry<E> addBefore(E e, Entry<E> entry) {

Entry<E> newEntry = **new** Entry<E>(e, entry, entry.previous);

newEntry.previous.next = newEntry;

newEntry.next.previous = newEntry;

size++;

modCount++;

**return** newEntry;

}

**private** E remove(Entry<E> e) {

**if** (e == header)

**throw** **new** NoSuchElementException();

E result = e.element;

e.previous.next = e.next;

e.next.previous = e.previous;

e.next = e.previous = **null**;

e.element = **null**;

size--;

modCount++;

**return** result;

}

/\*\*

\* **@since** 1.6

\*/

**public** Iterator<E> descendingIterator() {

**return** **new** DescendingIterator();

}

/\*\* Adapter to provide descending iterators via ListItr.previous \*/

**private** **class** DescendingIterator **implements** Iterator {

**final** ListItr itr = **new** ListItr(size());

**public** **boolean** hasNext() {

**return** itr.hasPrevious();

}

**public** E next() {

**return** itr.previous();

}

**public** **void** remove() {

itr.remove();

}

}

}

# LinkedHashMap

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (*insertion-order*). Note that insertion order is not affected if a key is *re-inserted* into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

This implementation spares its clients from the unspecified, generally chaotic ordering provided by [HashMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap) (and [Hashtable](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Hashtable.java#Hashtable)), without incurring the increased cost associated with [TreeMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeMap.java#TreeMap). It can be used to produce a copy of a map that has the same order as the original, regardless of the original map's implementation:

void foo(Map m) {

Map copy = new LinkedHashMap(m);

...

}

This technique is particularly useful if a module takes a map on input, copies it, and later returns results whose order is determined by that of the copy. (Clients generally appreciate having things returned in the same order they were presented.)

A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). This kind of map is well-suited to building LRU caches. Invoking the put or get method results in an access to the corresponding entry (assuming it exists after the invocation completes). The putAll method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. *No other methods generate entry accesses.* In particular, operations on collection-views do *not* affect the order of iteration of the backing map.

The [removeEldestEntry(java.util.Map.Entry)](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/LinkedHashMap.java#LinkedHashMap.removeEldestEntry%28java.util.Map.Entry%29) method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map.

This class provides all of the optional Map operations, and permits null elements. Like HashMap, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a LinkedHashMap requires time proportional to the *size* of the map, regardless of its capacity. Iteration over a HashMap is likely to be more expensive, requiring time proportional to its *capacity*.

A linked hash map has two parameters that affect its performance: *initial capacity* and *load factor*. They are defined precisely as for HashMap. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than forHashMap, as iteration times for this class are unaffected by capacity.

**Note that this implementation is not synchronized.** If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using theCollections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new LinkedHashMap(...));

A structural modification is any operation that adds or deletes one or more mappings or, in the case of access-ordered linked hash maps, affects iteration order. In insertion-ordered linked hash maps, merely changing the value associated with a key that is already contained in the map is not a structural modification. **In access-ordered linked hash maps, merely querying the map with get is a structural modification.**)

The iterators returned by the iterator method of the collections returned by all of this class's collection view methods are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#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.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

**public** **class** LinkedHashMap<K,V>

**extends** HashMap<K,V>

**implements** Map<K,V>

{

**private** **static** **final** **long** *serialVersionUID* = 3801124242820219131L;

/\*\*

\* The head of the doubly linked list.

\*/

**private** **transient** Entry<K,V> header;

/\*\*

\* The iteration ordering method for this linked hash map: <tt>true</tt>

\* for access-order, <tt>false</tt> for insertion-order.

\*

\* **@serial**

\*/

**private** **final** **boolean** accessOrder;

/\*\*

\* Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance

\* with the specified initial capacity and load factor.

\*

\* **@param** initialCapacity the initial capacity

\* **@param** loadFactor the load factor

\* **@throws** IllegalArgumentException if the initial capacity is negative

\* or the load factor is nonpositive

\*/

**public** LinkedHashMap(**int** initialCapacity, **float** loadFactor) {

**super**(initialCapacity, loadFactor);

accessOrder = **false**;

}

/\*\*

\* Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance

\* with the specified initial capacity and a default load factor (0.75).

\*

\* **@param** initialCapacity the initial capacity

\* **@throws** IllegalArgumentException if the initial capacity is negative

\*/

**public** LinkedHashMap(**int** initialCapacity) {

**super**(initialCapacity);

accessOrder = **false**;

}

/\*\*

\* Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance

\* with the default initial capacity (16) and load factor (0.75).

\*/

**public** LinkedHashMap() {

**super**();

accessOrder = **false**;

}

/\*\*

\* Constructs an empty <tt>LinkedHashMap</tt> instance with the

\* specified initial capacity, load factor and ordering mode.

\*

\* @param initialCapacity the initial capacity

\* @param loadFactor the load factor

\* @param accessOrder the ordering mode - <tt>true</tt> for

\* access-order, <tt>false</tt> for insertion-order

\* @throws IllegalArgumentException if the initial capacity is negative

\* or the load factor is nonpositive

\*/

**public** LinkedHashMap(**int** initialCapacity,

**float** loadFactor,

**boolean** accessOrder) {

**super**(initialCapacity, loadFactor);

**this**.accessOrder = accessOrder;

}

/\*\*

\* Called by superclass constructors and pseudoconstructors (clone,

\* readObject) before any entries are inserted into the map. Initializes

\* the chain.

\*/

**void** init() {

header = **new** Entry<K,V>(-1, **null**, **null**, **null**);

header.before = header.after = header;

}

/\*\*

\* Transfers all entries to new table array. This method is called

\* by superclass resize. It is overridden for performance, as it is

\* faster to iterate using our linked list.

\*/

**void** transfer(HashMap.Entry[] newTable) {

**int** newCapacity = newTable.length;

**for** (Entry<K,V> e = header.after; e != header; e = e.after) {

**int** index = indexFor(e.hash, newCapacity);

e.next = newTable[index];

newTable[index] = e;

}

}

/\*\*

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

\* specified value.

\*

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

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

\* specified value

\*/

**public** **boolean** containsValue(Object value) {

// Overridden to take advantage of faster iterator

**if** (value==**null**) {

**for** (Entry e = header.after; e != header; e = e.after)

**if** (e.value==**null**)

**return** **true**;

} **else** {

**for** (Entry e = header.after; e != header; e = e.after)

**if** (value.equals(e.value))

**return** **true**;

}

**return** **false**;

}

/\*\*

\* 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==null ? k==null :

\* key.equals(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) {

Entry<K,V> e = (Entry<K,V>)getEntry(key);

**if** (e == **null**)

**return** **null**;

e.recordAccess(**this**);

**return** e.value;

}

/\*\*

\* LinkedHashMap entry.

\*/

**private** **static** **class** Entry<K,V> **extends** HashMap.Entry<K,V> {

// These fields comprise the doubly linked list used for iteration.

Entry<K,V> before, after;

Entry(**int** hash, K key, V value, HashMap.Entry<K,V> next) {

**super**(hash, key, value, next);

}

/\*\*

\* Removes this entry from the linked list.

\*/

**private** **void** remove() {

before.after = after;

after.before = before;

}

/\*\*

\* Inserts this entry before the specified existing entry in the list.

\*/

**private** **void** addBefore(Entry<K,V> existingEntry) {

after = existingEntry;

before = existingEntry.before;

before.after = **this**;

after.before = **this**;

}

/\*\*

\* This method is invoked by the superclass whenever the value

\* of a pre-existing entry is read by Map.get or modified by Map.set.

\* If the enclosing Map is access-ordered, it moves the entry

\* to the end of the list; otherwise, it does nothing.

\*/

**void** recordAccess(HashMap<K,V> m) {

LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;

**if** (lm.accessOrder) {

lm.modCount++;

remove();

addBefore(lm.header);

}

}

**void** recordRemoval(HashMap<K,V> m) {

remove();

}

}

**private** **abstract** **class** LinkedHashIterator<T> **implements** Iterator<T> {

Entry<K,V> nextEntry = header.after;

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

/\*\*

\* The modCount value that the iterator believes that the backing

\* List should have. If this expectation is violated, the iterator

\* has detected concurrent modification.

\*/

**int** expectedModCount = modCount;

**public** **boolean** hasNext() {

**return** nextEntry != header;

}

**public** **void** remove() {

**if** (lastReturned == **null**)

**throw** **new** IllegalStateException();

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

LinkedHashMap.**this**.remove(lastReturned.key);

lastReturned = **null**;

expectedModCount = modCount;

}

Entry<K,V> nextEntry() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

**if** (nextEntry == header)

**throw** **new** NoSuchElementException();

Entry<K,V> e = lastReturned = nextEntry;

nextEntry = e.after;

**return** e;

}

}

/\*\*

\* This override alters behavior of superclass put method. It causes newly

\* allocated entry to get inserted at the end of the linked list and

\* removes the eldest entry if appropriate.

\*/

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

createEntry(hash, key, value, bucketIndex);

// Remove eldest entry if instructed, else grow capacity if appropriate

Entry<K,V> eldest = header.after;

**if** (removeEldestEntry(eldest)) {

removeEntryForKey(eldest.key);

} **else** {

**if** (size >= threshold)

resize(2 \* table.length);

}

}

/\*\*

\* This override differs from addEntry in that it doesn't resize the

\* table or remove the eldest entry.

\*/

**void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {

HashMap.Entry<K,V> old = table[bucketIndex];

Entry<K,V> e = **new** Entry<K,V>(hash, key, value, old);

table[bucketIndex] = e;

e.addBefore(header);

size++;

}

/\*\*

\* Returns <tt>true</tt> if this map should remove its eldest entry.

\* This method is invoked by <tt>put</tt> and <tt>putAll</tt> after

\* inserting a new entry into the map. It provides the implementor

\* with the opportunity to remove the eldest entry each time a new one

\* is added. This is useful if the map represents a cache: it allows

\* the map to reduce memory consumption by deleting stale entries.

\*

\* <p>Sample use: this override will allow the map to grow up to 100

\* entries and then delete the eldest entry each time a new entry is

\* added, maintaining a steady state of 100 entries.

\* <pre>

\* private static final int MAX\_ENTRIES = 100;

\*

\* protected boolean removeEldestEntry(Map.Entry eldest) {

\* return size() > MAX\_ENTRIES;

\* }

\* </pre>

\*

\* <p>This method typically does not modify the map in any way,

\* instead allowing the map to modify itself as directed by its

\* return value. It <i>is</i> permitted for this method to modify

\* the map directly, but if it does so, it <i>must</i> return

\* <tt>false</tt> (indicating that the map should not attempt any

\* further modification). The effects of returning <tt>true</tt>

\* after modifying the map from within this method are unspecified.

\*

\* <p>This implementation merely returns <tt>false</tt> (so that this

\* map acts like a normal map - the eldest element is never removed).

\*

\* **@param** eldest The least recently inserted entry in the map, or if

\* this is an access-ordered map, the least recently accessed

\* entry. This is the entry that will be removed it this

\* method returns <tt>true</tt>. If the map was empty prior

\* to the <tt>put</tt> or <tt>putAll</tt> invocation resulting

\* in this invocation, this will be the entry that was just

\* inserted; in other words, if the map contains a single

\* entry, the eldest entry is also the newest.

\* **@return** <tt>true</tt> if the eldest entry should be removed

\* from the map; <tt>false</tt> if it should be retained.

\*/

**protected** **boolean** removeEldestEntry(Map.Entry<K,V> eldest) {

**return** **false**;

}

}

# ConcurrentHashMap

A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as [java.util.Hashtable](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Hashtable.java#Hashtable), and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do *not* entail locking, and there is *not* any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset. For aggregate operations such as putAlland clear, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do *not* throw[java.util.ConcurrentModificationException](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ConcurrentModificationException.java#ConcurrentModificationException). However, iterators are designed to be used by only one thread at a time.

The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Because placement in hash tables is essentially random, the actual concurrency will vary. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention. But overestimates and underestimates within an order of magnitude do not usually have much noticeable impact. A value of one is appropriate when it is known that only one thread will modify and all others will only read. Also, resizing this or any other kind of hash table is a relatively slow operation, so, when possible, it is a good idea to provide estimates of expected table sizes in constructors.

This class and its views and iterators implement all of the *optional* methods of the [java.util.Map](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Map.java#Map) and [java.util.Iterator](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Iterator.java#Iterator) interfaces.

Like [java.util.Hashtable](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Hashtable.java#Hashtable) but unlike [java.util.HashMap](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap), this class does *not* allow null to be used as a key or value.

This class is a member of the Java Collections Framework

**public** **class** ConcurrentHashMap<K, V> **extends** AbstractMap<K, V>

**implements** ConcurrentMap<K, V>, Serializable {

**private** **static** **final** **long** *serialVersionUID* = 7249069246763182397L;

/\*

\* The basic strategy is to subdivide the table among Segments,

\* each of which itself is a concurrently readable hash table.

\*/

/\* ---------------- Constants -------------- \*/

/\*\*

\* The default initial capacity for this table,

\* used when not otherwise specified in a constructor.

\*/

**static** **final** **int** *DEFAULT\_INITIAL\_CAPACITY* = 16;

/\*\*

\* The default load factor for this table, used when not

\* otherwise specified in a constructor.

\*/

**static** **final** **float** *DEFAULT\_LOAD\_FACTOR* = 0.75f;

/\*\*

\* The default concurrency level for this table, used when not

\* otherwise specified in a constructor.

\*/

**static** **final** **int** *DEFAULT\_CONCURRENCY\_LEVEL* = 16;

/\*\*

\* The maximum capacity, used if a higher value is implicitly

\* specified by either of the constructors with arguments. MUST

\* be a power of two <= 1<<30 to ensure that entries are indexable

\* using ints.

\*/

**static** **final** **int** *MAXIMUM\_CAPACITY* = 1 << 30;

/\*\*

\* The maximum number of segments to allow; used to bound

\* constructor arguments.

\*/

**static** **final** **int** *MAX\_SEGMENTS* = 1 << 16; // slightly conservative

/\*\*

\* Number of unsynchronized retries in size and containsValue

\* methods before resorting to locking. This is used to avoid

\* unbounded retries if tables undergo continuous modification

\* which would make it impossible to obtain an accurate result.

\*/

**static** **final** **int** *RETRIES\_BEFORE\_LOCK* = 2;

/\* ---------------- Fields -------------- \*/

/\*\*

\* Mask value for indexing into segments. The upper bits of a

\* key's hash code are used to choose the segment.

\*/

**final** **int** segmentMask;

/\*\*

\* Shift value for indexing within segments.

\*/

**final** **int** segmentShift;

/\*\*

\* The segments, each of which is a specialized hash table

\*/

**final** Segment<K,V>[] segments;

**transient** Collection<V> values;

/\* ---------------- Small Utilities -------------- \*/

/\*\*

\* Applies a supplemental hash function to a given hashCode, which

\* defends against poor quality hash functions. This is critical

\* because ConcurrentHashMap uses power-of-two length hash tables,

\* that otherwise encounter collisions for hashCodes that do not

\* differ in lower or upper bits.

\*/

**private** **static** **int** hash(**int** h) {

// Spread bits to regularize both segment and index locations,

// using variant of single-word Wang/Jenkins hash.

h += (h << 15) ^ 0xffffcd7d;

h ^= (h >>> 10);

h += (h << 3);

h ^= (h >>> 6);

h += (h << 2) + (h << 14);

**return** h ^ (h >>> 16);

}

/\*\*

\* Returns the segment that should be used for key with given hash

\* **@param** hash the hash code for the key

\* **@return** the segment

\*/

**final** Segment<K,V> segmentFor(**int** hash) {

**return** segments[(hash >>> segmentShift) & segmentMask];

}

/\*\*

\* Segments are specialized versions of hash tables. This

\* subclasses from ReentrantLock opportunistically, just to

\* simplify some locking and avoid separate construction.

\*/

**static** **final** **class** Segment<K,V> **extends** ReentrantLock **implements** Serializable {

/\*

\* Segments maintain a table of entry lists that are ALWAYS

\* kept in a consistent state, so can be read without locking.

\* Next fields of nodes are immutable (final). All list

\* additions are performed at the front of each bin. This

\* makes it easy to check changes, and also fast to traverse.

\* When nodes would otherwise be changed, new nodes are

\* created to replace them. This works well for hash tables

\* since the bin lists tend to be short. (The average length

\* is less than two for the default load factor threshold.)

\*

\* Read operations can thus proceed without locking, but rely

\* on selected uses of volatiles to ensure that completed

\* write operations performed by other threads are

\* noticed. For most purposes, the "count" field, tracking the

\* number of elements, serves as that volatile variable

\* ensuring visibility. This is convenient because this field

\* needs to be read in many read operations anyway:

\*

\* - All (unsynchronized) read operations must first read the

\* "count" field, and should not look at table entries if

\* it is 0.

\*

\* - All (synchronized) write operations should write to

\* the "count" field after structurally changing any bin.

\* The operations must not take any action that could even

\* momentarily cause a concurrent read operation to see

\* inconsistent data. This is made easier by the nature of

\* the read operations in Map. For example, no operation

\* can reveal that the table has grown but the threshold

\* has not yet been updated, so there are no atomicity

\* requirements for this with respect to reads.

\*

\* As a guide, all critical volatile reads and writes to the

\* count field are marked in code comments.

\*/

**private** **static** **final** **long** *serialVersionUID* = 2249069246763182397L;

/\*\*

\* The number of elements in this segment's region.

\*/

**transient** **volatile** **int** count;

/\*\*

\* Number of updates that alter the size of the table. This is

\* used during bulk-read methods to make sure they see a

\* consistent snapshot: If modCounts change during a traversal

\* of segments computing size or checking containsValue, then

\* we might have an inconsistent view of state so (usually)

\* must retry.

\*/

**transient** **int** modCount;

/\*\*

\* The table is rehashed when its size exceeds this threshold.

\* (The value of this field is always <tt>(int)(capacity \*

\* loadFactor)</tt>.)

\*/

**transient** **int** threshold;

/\*\*

\* The per-segment table.

\*/

**transient** **volatile** HashEntry<K,V>[] table;

/\*\*

\* The load factor for the hash table. Even though this value

\* is same for all segments, it is replicated to avoid needing

\* links to outer object.

\* **@serial**

\*/

**final** **float** loadFactor;

Segment(**int** initialCapacity, **float** lf) {

loadFactor = lf;

setTable(HashEntry.<K,V>newArray(initialCapacity));

}

@SuppressWarnings("unchecked")

**static** **final** <K,V> Segment<K,V>[] newArray(**int** i) {

**return** **new** Segment[i];

}

/\*\*

\* Sets table to new HashEntry array.

\* Call only while holding lock or in constructor.

\*/

**void** setTable(HashEntry<K,V>[] newTable) {

threshold = (**int**)(newTable.length \* loadFactor);

table = newTable;

}

/\*\*

\* Returns properly casted first entry of bin for given hash.

\*/

HashEntry<K,V> getFirst(**int** hash) {

HashEntry<K,V>[] tab = table;

**return** tab[hash & (tab.length - 1)];

}

/\*\*

\* Reads value field of an entry under lock. Called if value

\* field ever appears to be null. This is possible only if a

\* compiler happens to reorder a HashEntry initialization with

\* its table assignment, which is legal under memory model

\* but is not known to ever occur.

\*/

V readValueUnderLock(HashEntry<K,V> e) {

lock();

**try** {

**return** e.value;

} **finally** {

unlock();

}

}

/\* Specialized implementations of map methods \*/

V get(Object key, **int** hash) {

**if** (count != 0) { // read-volatile

HashEntry<K,V> e = getFirst(hash);

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

**if** (e.hash == hash && key.equals(e.key)) {

V v = e.value;

**if** (v != **null**)

**return** v;

**return** readValueUnderLock(e); // recheck

}

e = e.next;

}

}

**return** **null**;

}

**boolean** containsKey(Object key, **int** hash) {

**if** (count != 0) { // read-volatile

HashEntry<K,V> e = getFirst(hash);

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

**if** (e.hash == hash && key.equals(e.key))

**return** **true**;

e = e.next;

}

}

**return** **false**;

}

**boolean** containsValue(Object value) {

**if** (count != 0) { // read-volatile

HashEntry<K,V>[] tab = table;

**int** len = tab.length;

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

**for** (HashEntry<K,V> e = tab[i]; e != **null**; e = e.next) {

V v = e.value;

**if** (v == **null**) // recheck

v = readValueUnderLock(e);

**if** (value.equals(v))

**return** **true**;

}

}

}

**return** **false**;

}

**boolean** replace(K key, **int** hash, V oldValue, V newValue) {

lock();

**try** {

HashEntry<K,V> e = getFirst(hash);

**while** (e != **null** && (e.hash != hash || !key.equals(e.key)))

e = e.next;

**boolean** replaced = **false**;

**if** (e != **null** && oldValue.equals(e.value)) {

replaced = **true**;

e.value = newValue;

}

**return** replaced;

} **finally** {

unlock();

}

}

V replace(K key, **int** hash, V newValue) {

lock();

**try** {

HashEntry<K,V> e = getFirst(hash);

**while** (e != **null** && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue = **null**;

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

oldValue = e.value;

e.value = newValue;

}

**return** oldValue;

} **finally** {

unlock();

}

}

V put(K key, **int** hash, V value, **boolean** onlyIfAbsent) {

lock();

**try** {

**int** c = count;

**if** (c++ > threshold) // ensure capacity

rehash();

HashEntry<K,V>[] tab = table;

**int** index = hash & (tab.length - 1);

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

**while** (e != **null** && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

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

oldValue = e.value;

**if** (!onlyIfAbsent)

e.value = value;

}

**else** {

oldValue = **null**;

++modCount;

tab[index] = **new** HashEntry<K,V>(key, hash, first, value);

count = c; // write-volatile

}

**return** oldValue;

} **finally** {

unlock();

}

}

**void** rehash() {

HashEntry<K,V>[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity >= *MAXIMUM\_CAPACITY*)

**return**;

/\*

\* Reclassify nodes in each list to new Map. Because we are

\* using power-of-two expansion, the elements from each bin

\* must either stay at same index, or move with a power of two

\* offset. We eliminate unnecessary node creation by catching

\* cases where old nodes can be reused because their next

\* fields won't change. Statistically, at the default

\* threshold, only about one-sixth of them need cloning when

\* a table doubles. The nodes they replace will be garbage

\* collectable as soon as they are no longer referenced by any

\* reader thread that may be in the midst of traversing table

\* right now.

\*/

HashEntry<K,V>[] newTable = HashEntry.newArray(oldCapacity<<1);

threshold = (**int**)(newTable.length \* loadFactor);

**int** sizeMask = newTable.length - 1;

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

// We need to guarantee that any existing reads of old Map can

// proceed. So we cannot yet null out each bin.

HashEntry<K,V> e = oldTable[i];

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

HashEntry<K,V> next = e.next;

**int** idx = e.hash & sizeMask;

// Single node on list

**if** (next == **null**)

newTable[idx] = e;

**else** {

// Reuse trailing consecutive sequence at same slot

HashEntry<K,V> lastRun = e;

**int** lastIdx = idx;

**for** (HashEntry<K,V> last = next;

last != **null**;

last = last.next) {

**int** k = last.hash & sizeMask;

**if** (k != lastIdx) {

lastIdx = k;

lastRun = last;

}

}

newTable[lastIdx] = lastRun;

// Clone all remaining nodes

**for** (HashEntry<K,V> p = e; p != lastRun; p = p.next) {

**int** k = p.hash & sizeMask;

HashEntry<K,V> n = newTable[k];

newTable[k] = **new** HashEntry<K,V>(p.key, p.hash,

n, p.value);

}

}

}

}

table = newTable;

}

/\*\*

\* Remove; match on key only if value null, else match both.

\*/

V remove(Object key, **int** hash, Object value) {

lock();

**try** {

**int** c = count - 1;

HashEntry<K,V>[] tab = table;

**int** index = hash & (tab.length - 1);

HashEntry<K,V> first = tab[index];

HashEntry<K,V> e = first;

**while** (e != **null** && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue = **null**;

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

V v = e.value;

**if** (value == **null** || value.equals(v)) {

oldValue = v;

// All entries following removed node can stay

// in list, but all preceding ones need to be

// cloned.

++modCount;

HashEntry<K,V> newFirst = e.next;

**for** (HashEntry<K,V> p = first; p != e; p = p.next)

newFirst = **new** HashEntry<K,V>(p.key, p.hash,

newFirst, p.value);

tab[index] = newFirst;

count = c; // write-volatile

}

}

**return** oldValue;

} **finally** {

unlock();

}

}

**void** clear() {

**if** (count != 0) {

lock();

**try** {

HashEntry<K,V>[] tab = table;

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

tab[i] = **null**;

++modCount;

count = 0; // write-volatile

} **finally** {

unlock();

}

}

}

}

/\* ---------------- Public operations -------------- \*/

/\*\*

\* Creates a new, empty map with the specified initial

\* capacity, load factor and concurrency level.

\*

\* **@param** initialCapacity the initial capacity. The implementation

\* performs internal sizing to accommodate this many elements.

\* **@param** loadFactor the load factor threshold, used to control resizing.

\* Resizing may be performed when the average number of elements per

\* bin exceeds this threshold.

\* **@param** concurrencyLevel the estimated number of concurrently

\* updating threads. The implementation performs internal sizing

\* to try to accommodate this many threads.

\* **@throws** IllegalArgumentException if the initial capacity is

\* negative or the load factor or concurrencyLevel are

\* nonpositive.

\*/

**public** ConcurrentHashMap(**int** initialCapacity,

**float** loadFactor, **int** concurrencyLevel) {

**if** (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

**throw** **new** IllegalArgumentException();

**if** (concurrencyLevel > *MAX\_SEGMENTS*)

concurrencyLevel = *MAX\_SEGMENTS*;

// Find power-of-two sizes best matching arguments

**int** sshift = 0;

**int** ssize = 1;

**while** (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

segmentShift = 32 - sshift;

segmentMask = ssize - 1;

**this**.segments = Segment.*newArray*(ssize);

**if** (initialCapacity > *MAXIMUM\_CAPACITY*)

initialCapacity = *MAXIMUM\_CAPACITY*;

**int** c = initialCapacity / ssize;

**if** (c \* ssize < initialCapacity)

++c;

**int** cap = 1;

**while** (cap < c)

cap <<= 1;

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

**this**.segments[i] = **new** Segment<K,V>(cap, loadFactor);

}

/\*\*

\* Creates a new, empty map with the specified initial capacity

\* and load factor and with the default concurrencyLevel (16).

\*

\* **@param** initialCapacity The implementation performs internal

\* sizing to accommodate this many elements.

\* **@param** loadFactor the load factor threshold, used to control resizing.

\* Resizing may be performed when the average number of elements per

\* bin exceeds this threshold.

\* **@throws** IllegalArgumentException if the initial capacity of

\* elements is negative or the load factor is nonpositive

\*

\* **@since** 1.6

\*/

**public** ConcurrentHashMap(**int** initialCapacity, **float** loadFactor) {

**this**(initialCapacity, loadFactor, *DEFAULT\_CONCURRENCY\_LEVEL*);

}

/\*\*

\* Creates a new, empty map with a default initial capacity (16),

\* load factor (0.75) and concurrencyLevel (16).

\*/

**public** ConcurrentHashMap() {

**this**(*DEFAULT\_INITIAL\_CAPACITY*, *DEFAULT\_LOAD\_FACTOR*, *DEFAULT\_CONCURRENCY\_LEVEL*);

}

/\*\*

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

\* map contains more than <tt>Integer.MAX\_VALUE</tt> elements, returns

\* <tt>Integer.MAX\_VALUE</tt>.

\*

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

\*/

**public** **int** size() {

**final** Segment<K,V>[] segments = **this**.segments;

**long** sum = 0;

**long** check = 0;

**int**[] mc = **new** **int**[segments.length];

// Try a few times to get accurate count. On failure due to

// continuous async changes in table, resort to locking.

**for** (**int** k = 0; k < *RETRIES\_BEFORE\_LOCK*; ++k) {

check = 0;

sum = 0;

**int** mcsum = 0;

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

sum += segments[i].count;

mcsum += mc[i] = segments[i].modCount;

}

**if** (mcsum != 0) {

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

check += segments[i].count;

**if** (mc[i] != segments[i].modCount) {

check = -1; // force retry

**break**;

}

}

}

**if** (check == sum)

**break**;

}

**if** (check != sum) { // Resort to locking all segments

sum = 0;

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

segments[i].lock();

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

sum += segments[i].count;

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

segments[i].unlock();

}

**if** (sum > Integer.*MAX\_VALUE*)

**return** Integer.*MAX\_VALUE*;

**else**

**return** (**int**)sum;

}

/\*\*

\* 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.equals(k)},

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

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

\*

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

\*/

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

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).get(key, hash);

}

/\*\*

\* Tests if the specified object is a key in this table.

\*

\* **@param** key possible key

\* **@return** <tt>true</tt> if and only if the specified object

\* is a key in this table, as determined by the

\* <tt>equals</tt> method; <tt>false</tt> otherwise.

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

\*/

**public** **boolean** containsKey(Object key) {

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).containsKey(key, hash);

}

/\*\*

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

\* specified value. Note: This method requires a full internal

\* traversal of the hash table, and so is much slower than

\* method <tt>containsKey</tt>.

\*

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

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

\* specified value

\* **@throws** NullPointerException if the specified value is null

\*/

**public** **boolean** containsValue(Object value) {

**if** (value == **null**)

**throw** **new** NullPointerException();

// See explanation of modCount use above

**final** Segment<K,V>[] segments = **this**.segments;

**int**[] mc = **new** **int**[segments.length];

// Try a few times without locking

**for** (**int** k = 0; k < *RETRIES\_BEFORE\_LOCK*; ++k) {

**int** sum = 0;

**int** mcsum = 0;

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

**int** c = segments[i].count;

mcsum += mc[i] = segments[i].modCount;

**if** (segments[i].containsValue(value))

**return** **true**;

}

**boolean** cleanSweep = **true**;

**if** (mcsum != 0) {

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

**int** c = segments[i].count;

**if** (mc[i] != segments[i].modCount) {

cleanSweep = **false**;

**break**;

}

}

}

**if** (cleanSweep)

**return** **false**;

}

// Resort to locking all segments

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

segments[i].lock();

**boolean** found = **false**;

**try** {

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

**if** (segments[i].containsValue(value)) {

found = **true**;

**break**;

}

}

} **finally** {

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

segments[i].unlock();

}

**return** found;

}

/\*\*

\* Legacy method testing if some key maps into the specified value

\* in this table. This method is identical in functionality to

\* {@link #containsValue}, and exists solely to ensure

\* full compatibility with class {@link java.util.Hashtable},

\* which supported this method prior to introduction of the

\* Java Collections framework.

\* **@param** value a value to search for

\* **@return** <tt>true</tt> if and only if some key maps to the

\* <tt>value</tt> argument in this table as

\* determined by the <tt>equals</tt> method;

\* <tt>false</tt> otherwise

\* **@throws** NullPointerException if the specified value is null

\*/

**public** **boolean** contains(Object value) {

**return** containsValue(value);

}

/\*\*

\* Maps the specified key to the specified value in this table.

\* Neither the key nor the value can be null.

\*

\* <p> The value can be retrieved by calling the <tt>get</tt> method

\* with a key that is equal to the original key.

\*

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

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

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>

\* **@throws** NullPointerException if the specified key or value is null

\*/

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

**if** (value == **null**)

**throw** **new** NullPointerException();

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).put(key, hash, value, **false**);

}

/\*\*

\* {@inheritDoc}

\*

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

\* or <tt>null</tt> if there was no mapping for the key

\* **@throws** NullPointerException if the specified key or value is null

\*/

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

**if** (value == **null**)

**throw** **new** NullPointerException();

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).put(key, hash, value, **true**);

}

/\*\*

\* Removes the key (and its corresponding value) from this map.

\* This method does nothing if the key is not in the map.

\*

\* **@param** key the key that needs to be removed

\* **@return** the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>

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

\*/

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

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).remove(key, hash, **null**);

}

/\*\*

\* {@inheritDoc}

\*

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

\*/

**public** **boolean** remove(Object key, Object value) {

**int** hash = *hash*(key.hashCode());

**if** (value == **null**)

**return** **false**;

**return** segmentFor(hash).remove(key, hash, value) != **null**;

}

/\*\*

\* {@inheritDoc}

\*

\* **@throws** NullPointerException if any of the arguments are null

\*/

**public** **boolean** replace(K key, V oldValue, V newValue) {

**if** (oldValue == **null** || newValue == **null**)

**throw** **new** NullPointerException();

**int** hash = *hash*(key.hashCode());

**return** segmentFor(hash).replace(key, hash, oldValue, newValue);

}

/\*\*

\* ConcurrentHashMap list entry. Note that this is never exported

\* out as a user-visible Map.Entry.

\*

\* Because the value field is volatile, not final, it is legal wrt

\* the Java Memory Model for an unsynchronized reader to see null

\* instead of initial value when read via a data race. Although a

\* reordering leading to this is not likely to ever actually

\* occur, the Segment.readValueUnderLock method is used as a

\* backup in case a null (pre-initialized) value is ever seen in

\* an unsynchronized access method.

\*/

**static** **final** **class** HashEntry<K,V> {

**final** K key;

**final** **int** hash;

**volatile** V value;

**final** HashEntry<K,V> next;

HashEntry(K key, **int** hash, HashEntry<K,V> next, V value) {

**this**.key = key;

**this**.hash = hash;

**this**.next = next;

**this**.value = value;

}

@SuppressWarnings("unchecked")

**static** **final** <K,V> HashEntry<K,V>[] newArray(**int** i) {

**return** **new** HashEntry[i];

}

}

}

# Object Class

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

**public** **class** Object {

**private** **static** **native** **void** registerNatives();

**static** {

*registerNatives*();

}

/\*\*

\* Returns the runtime class of this {@code Object}. The returned

\* {@code Class} object is the object that is locked by {@code

\* static synchronized} methods of the represented class.

\*

\* <p><b>The actual result type is {@code Class<? extends |X|>}

\* where {@code |X|} is the erasure of the static type of the

\* expression on which {@code getClass} is called.</b> For

\* example, no cast is required in this code fragment:</p>

\*

\* <p>

\* {@code Number n = 0; }<br>

\* {@code Class<? extends Number> c = n.getClass(); }

\* </p>

\*

\* **@return** The {@code Class} object that represents the runtime

\* class of this object.

\* **@see** Class Literals, section 15.8.2 of

\* <cite>The Java&trade; Language Specification</cite>.

\*/

**public** **final** **native** Class<?> getClass();

/\*\*

\* Returns a hash code value for the object. This method is

\* supported for the benefit of hash tables such as those provided by

\* {@link java.util.HashMap}.

\* <p>

\* The general contract of {@code hashCode} is:

\* <ul>

\* <li>Whenever it is invoked on the same object more than once during

\* an execution of a Java application, the {@code hashCode} method

\* must consistently return the same integer, provided no information

\* used in {@code equals} comparisons on the object is modified.

\* This integer need not remain consistent from one execution of an

\* application to another execution of the same application.

\* <li>If two objects are equal according to the {@code equals(Object)}

\* method, then calling the {@code hashCode} method on each of

\* the two objects must produce the same integer result.

\* <li>It is <em>not</em> required that if two objects are unequal

\* according to the {@link java.lang.Object#equals(java.lang.Object)}

\* method, then calling the {@code hashCode} method on each of the

\* two objects must produce distinct integer results. However, the

\* programmer should be aware that producing distinct integer results

\* for unequal objects may improve the performance of hash tables.

\* </ul>

\* <p>

\* As much as is reasonably practical, the hashCode method defined by

\* class {@code Object} does return distinct integers for distinct

\* objects. (This is typically implemented by converting the internal

\* address of the object into an integer, but this implementation

\* technique is not required by the

\* Java<font size="-2"><sup>TM</sup></font> programming language.)

\*

\* **@return** a hash code value for this object.

\* **@see** java.lang.Object#equals(java.lang.Object)

\* **@see** java.lang.System#identityHashCode

\*/

**public** **native** **int** hashCode();

/\*\*

\* Indicates whether some other object is "equal to" this one.

\* <p>

\* The {@code equals} method implements an equivalence relation

\* on non-null object references:

\* <ul>

\* <li>It is <i>reflexive</i>: for any non-null reference value

\* {@code x}, {@code x.equals(x)} should return

\* {@code true}.

\* <li>It is <i>symmetric</i>: for any non-null reference values

\* {@code x} and {@code y}, {@code x.equals(y)}

\* should return {@code true} if and only if

\* {@code y.equals(x)} returns {@code true}.

\* <li>It is <i>transitive</i>: for any non-null reference values

\* {@code x}, {@code y}, and {@code z}, if

\* {@code x.equals(y)} returns {@code true} and

\* {@code y.equals(z)} returns {@code true}, then

\* {@code x.equals(z)} should return {@code true}.

\* <li>It is <i>consistent</i>: for any non-null reference values

\* {@code x} and {@code y}, multiple invocations of

\* {@code x.equals(y)} consistently return {@code true}

\* or consistently return {@code false}, provided no

\* information used in {@code equals} comparisons on the

\* objects is modified.

\* <li>For any non-null reference value {@code x},

\* {@code x.equals(null)} should return {@code false}.

\* </ul>

\* <p>

\* The {@code equals} method for class {@code Object} implements

\* the most discriminating possible equivalence relation on objects;

\* that is, for any non-null reference values {@code x} and

\* {@code y}, this method returns {@code true} if and only

\* if {@code x} and {@code y} refer to the same object

\* ({@code x == y} has the value {@code true}).

\* <p>

\* Note that it is generally necessary to override the {@code hashCode}

\* method whenever this method is overridden, so as to maintain the

\* general contract for the {@code hashCode} method, which states

\* that equal objects must have equal hash codes.

\*

\* **@param** obj the reference object with which to compare.

\* **@return** {@code true} if this object is the same as the obj

\* argument; {@code false} otherwise.

\* **@see** #hashCode()

\* **@see** java.util.HashMap

\*/

**public** **boolean** equals(Object obj) {

**return** (**this** == obj);

}

/\*\*

\* Creates and returns a copy of this object. The precise meaning

\* of "copy" may depend on the class of the object. The general

\* intent is that, for any object {@code x}, the expression:

\* <blockquote>

\* <pre>

\* x.clone() != x</pre></blockquote>

\* will be true, and that the expression:

\* <blockquote>

\* <pre>

\* x.clone().getClass() == x.getClass()</pre></blockquote>

\* will be {@code true}, but these are not absolute requirements.

\* While it is typically the case that:

\* <blockquote>

\* <pre>

\* x.clone().equals(x)</pre></blockquote>

\* will be {@code true}, this is not an absolute requirement.

\* <p>

\* By convention, the returned object should be obtained by calling

\* {@code super.clone}. If a class and all of its superclasses (except

\* {@code Object}) obey this convention, it will be the case that

\* {@code x.clone().getClass() == x.getClass()}.

\* <p>

\* By convention, the object returned by this method should be independent

\* of this object (which is being cloned). To achieve this independence,

\* it may be necessary to modify one or more fields of the object returned

\* by {@code super.clone} before returning it. Typically, this means

\* copying any mutable objects that comprise the internal "deep structure"

\* of the object being cloned and replacing the references to these

\* objects with references to the copies. If a class contains only

\* primitive fields or references to immutable objects, then it is usually

\* the case that no fields in the object returned by {@code super.clone}

\* need to be modified.

\* <p>

\* The method {@code clone} for class {@code Object} performs a

\* specific cloning operation. First, if the class of this object does

\* not implement the interface {@code Cloneable}, then a

\* {@code CloneNotSupportedException} is thrown. Note that all arrays

\* are considered to implement the interface {@code Cloneable} and that

\* the return type of the {@code clone} method of an array type {@code T[]}

\* is {@code T[]} where T is any reference or primitive type.

\* Otherwise, this method creates a new instance of the class of this

\* object and initializes all its fields with exactly the contents of

\* the corresponding fields of this object, as if by assignment; the

\* contents of the fields are not themselves cloned. Thus, this method

\* performs a "shallow copy" of this object, not a "deep copy" operation.

\* <p>

\* The class {@code Object} does not itself implement the interface

\* {@code Cloneable}, so calling the {@code clone} method on an object

\* whose class is {@code Object} will result in throwing an

\* exception at run time.

\*

\* **@return** a clone of this instance.

\* **@exception** CloneNotSupportedException if the object's class does not

\* support the {@code Cloneable} interface. Subclasses

\* that override the {@code clone} method can also

\* throw this exception to indicate that an instance cannot

\* be cloned.

\* **@see** java.lang.Cloneable

\*/

**protected** **native** Object clone() **throws** CloneNotSupportedException;

/\*\*

\* Returns a string representation of the object. In general, the

\* {@code toString} method returns a string that

\* "textually represents" this object. The result should

\* be a concise but informative representation that is easy for a

\* person to read.

\* It is recommended that all subclasses override this method.

\* <p>

\* The {@code toString} method for class {@code Object}

\* returns a string consisting of the name of the class of which the

\* object is an instance, the at-sign character `{@code @}', and

\* the unsigned hexadecimal representation of the hash code of the

\* object. In other words, this method returns a string equal to the

\* value of:

\* <blockquote>

\* <pre>

\* getClass().getName() + '@' + Integer.toHexString(hashCode())

\* </pre></blockquote>

\*

\* **@return** a string representation of the object.

\*/

**public** String toString() {

**return** getClass().getName() + "@" + Integer.*toHexString*(hashCode());

}

/\*\*

\* Wakes up a single thread that is waiting on this object's

\* monitor. If any threads are waiting on this object, one of them

\* is chosen to be awakened. The choice is arbitrary and occurs at

\* the discretion of the implementation. A thread waits on an object's

\* monitor by calling one of the {@code wait} methods.

\* <p>

\* The awakened thread will not be able to proceed until the current

\* thread relinquishes the lock on this object. The awakened thread will

\* compete in the usual manner with any other threads that might be

\* actively competing to synchronize on this object; for example, the

\* awakened thread enjoys no reliable privilege or disadvantage in being

\* the next thread to lock this object.

\* <p>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. A thread becomes the owner of the

\* object's monitor in one of three ways:

\* <ul>

\* <li>By executing a synchronized instance method of that object.

\* <li>By executing the body of a {@code synchronized} statement

\* that synchronizes on the object.

\* <li>For objects of type {@code Class,} by executing a

\* synchronized static method of that class.

\* </ul>

\* <p>

\* Only one thread at a time can own an object's monitor.

\*

\* **@exception** IllegalMonitorStateException if the current thread is not

\* the owner of this object's monitor.

\* **@see** java.lang.Object#notifyAll()

\* **@see** java.lang.Object#wait()

\*/

**public** **final** **native** **void** notify();

/\*\*

\* 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

\* {@code wait} methods.

\* <p>

\* The awakened threads will not be able to proceed until the current

\* thread relinquishes the lock on this object. The awakened threads

\* will compete in the usual manner with any other threads that might

\* be actively competing to synchronize on this object; for example,

\* the awakened threads enjoy no reliable privilege or disadvantage in

\* being the next thread to lock this object.

\* <p>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* **@exception** IllegalMonitorStateException if the current thread is not

\* the owner of this object's monitor.

\* **@see** java.lang.Object#notify()

\* **@see** java.lang.Object#wait()

\*/

**public** **final** **native** **void** notifyAll();

/\*\*

\* Causes the current thread to wait until either another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object, or a

\* specified amount of time has elapsed.

\* <p>

\* The current thread must own this object's monitor.

\* <p>

\* This method causes the current thread (call it <var>T</var>) to

\* place itself in the wait set for this object and then to relinquish

\* any and all synchronization claims on this object. Thread <var>T</var>

\* becomes disabled for thread scheduling purposes and lies dormant

\* until one of four things happens:

\* <ul>

\* <li>Some other thread invokes the {@code notify} method for this

\* object and thread <var>T</var> happens to be arbitrarily chosen as

\* the thread to be awakened.

\* <li>Some other thread invokes the {@code notifyAll} method for this

\* object.

\* <li>Some other thread {@linkplain Thread#interrupt() interrupts}

\* thread <var>T</var>.

\* <li>The specified amount of real time has elapsed, more or less. If

\* {@code timeout} is zero, however, then real time is not taken into

\* consideration and the thread simply waits until notified.

\* </ul>

\* The thread <var>T</var> is then removed from the wait set for this

\* object and re-enabled for thread scheduling. It then competes in the

\* usual manner with other threads for the right to synchronize on the

\* object; once it has gained control of the object, all its

\* synchronization claims on the object are restored to the status quo

\* ante - that is, to the situation as of the time that the {@code wait}

\* method was invoked. Thread <var>T</var> then returns from the

\* invocation of the {@code wait} method. Thus, on return from the

\* {@code wait} method, the synchronization state of the object and of

\* thread {@code T} is exactly as it was when the {@code wait} method

\* was invoked.

\* <p>

\* A thread can also wake up without being notified, interrupted, or

\* timing out, a so-called <i>spurious wakeup</i>. While this will rarely

\* occur in practice, applications must guard against it by testing for

\* the condition that should have caused the thread to be awakened, and

\* continuing to wait if the condition is not satisfied. In other words,

\* waits should always occur in loops, like this one:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait(timeout);

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* (For more information on this topic, see Section 3.2.3 in Doug Lea's

\* "Concurrent Programming in Java (Second Edition)" (Addison-Wesley,

\* 2000), or Item 50 in Joshua Bloch's "Effective Java Programming

\* Language Guide" (Addison-Wesley, 2001).

\*

\* <p>If the current thread is {@linkplain java.lang.Thread#interrupt()

\* interrupted} by any thread before or while it is waiting, then an

\* {@code InterruptedException} is thrown. This exception is not

\* thrown until the lock status of this object has been restored as

\* described above.

\*

\* <p>

\* Note that the {@code wait} method, as it places the current thread

\* into the wait set for this object, unlocks only this object; any

\* other objects on which the current thread may be synchronized remain

\* locked while the thread waits.

\* <p>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* **@param** timeout the maximum time to wait in milliseconds.

\* **@exception** IllegalArgumentException if the value of timeout is

\* negative.

\* **@exception** IllegalMonitorStateException if the current thread is not

\* the owner of the object's monitor.

\* **@exception** InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\* **@see** java.lang.Object#notify()

\* **@see** java.lang.Object#notifyAll()

\*/

**public** **final** **native** **void** wait(**long** timeout) **throws** InterruptedException;

/\*\*

\* Causes the current thread to wait until another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object, or

\* some other thread interrupts the current thread, or a certain

\* amount of real time has elapsed.

\* <p>

\* This method is similar to the {@code wait} method of one

\* argument, but it allows finer control over the amount of time to

\* wait for a notification before giving up. The amount of real time,

\* measured in nanoseconds, is given by:

\* <blockquote>

\* <pre>

\* 1000000\*timeout+nanos</pre></blockquote>

\* <p>

\* In all other respects, this method does the same thing as the

\* method {@link #wait(long)} of one argument. In particular,

\* {@code wait(0, 0)} means the same thing as {@code wait(0)}.

\* <p>

\* The current thread must own this object's monitor. The thread

\* releases ownership of this monitor and waits until either of the

\* following two conditions has occurred:

\* <ul>

\* <li>Another thread notifies threads waiting on this object's monitor

\* to wake up either through a call to the {@code notify} method

\* or the {@code notifyAll} method.

\* <li>The timeout period, specified by {@code timeout}

\* milliseconds plus {@code nanos} nanoseconds arguments, has

\* elapsed.

\* </ul>

\* <p>

\* The thread then waits until it can re-obtain ownership of the

\* monitor and resumes execution.

\* <p>

\* As in the one argument version, interrupts and spurious wakeups are

\* possible, and this method should always be used in a loop:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait(timeout, nanos);

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* **@param** timeout the maximum time to wait in milliseconds.

\* **@param** nanos additional time, in nanoseconds range

\* 0-999999.

\* **@exception** IllegalArgumentException if the value of timeout is

\* negative or the value of nanos is

\* not in the range 0-999999.

\* **@exception** IllegalMonitorStateException if the current thread is not

\* the owner of this object's monitor.

\* **@exception** InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\*/

**public** **final** **void** wait(**long** timeout, **int** nanos) **throws** InterruptedException {

**if** (timeout < 0) {

**throw** **new** IllegalArgumentException("timeout value is negative");

}

**if** (nanos < 0 || nanos > 999999) {

**throw** **new** IllegalArgumentException(

"nanosecond timeout value out of range");

}

**if** (nanos >= 500000 || (nanos != 0 && timeout == 0)) {

timeout++;

}

wait(timeout);

}

/\*\*

\* Causes the current thread to wait until another thread invokes the

\* {@link java.lang.Object#notify()} method or the

\* {@link java.lang.Object#notifyAll()} method for this object.

\* In other words, this method behaves exactly as if it simply

\* performs the call {@code wait(0)}.

\* <p>

\* The current thread must own this object's monitor. The thread

\* releases ownership of this monitor and waits until another thread

\* notifies threads waiting on this object's monitor to wake up

\* either through a call to the {@code notify} method or the

\* {@code notifyAll} method. The thread then waits until it can

\* re-obtain ownership of the monitor and resumes execution.

\* <p>

\* As in the one argument version, interrupts and spurious wakeups are

\* possible, and this method should always be used in a loop:

\* <pre>

\* synchronized (obj) {

\* while (&lt;condition does not hold&gt;)

\* obj.wait();

\* ... // Perform action appropriate to condition

\* }

\* </pre>

\* This method should only be called by a thread that is the owner

\* of this object's monitor. See the {@code notify} method for a

\* description of the ways in which a thread can become the owner of

\* a monitor.

\*

\* **@exception** IllegalMonitorStateException if the current thread is not

\* the owner of the object's monitor.

\* **@exception** InterruptedException if any thread interrupted the

\* current thread before or while the current thread

\* was waiting for a notification. The <i>interrupted

\* status</i> of the current thread is cleared when

\* this exception is thrown.

\* **@see** java.lang.Object#notify()

\* **@see** java.lang.Object#notifyAll()

\*/

**public** **final** **void** wait() **throws** InterruptedException {

wait(0);

}

/\*\*

\* Called by the garbage collector on an object when garbage collection

\* determines that there are no more references to the object.

\* A subclass overrides the {@code finalize} method to dispose of

\* system resources or to perform other cleanup.

\* <p>

\* The general contract of {@code finalize} is that it is invoked

\* if and when the Java<font size="-2"><sup>TM</sup></font> virtual

\* machine has determined that there is no longer any

\* means by which this object can be accessed by any thread that has

\* not yet died, except as a result of an action taken by the

\* finalization of some other object or class which is ready to be

\* finalized. The {@code finalize} method may take any action, including

\* making this object available again to other threads; the usual purpose

\* of {@code finalize}, however, is to perform cleanup actions before

\* the object is irrevocably discarded. For example, the finalize method

\* for an object that represents an input/output connection might perform

\* explicit I/O transactions to break the connection before the object is

\* permanently discarded.

\* <p>

\* The {@code finalize} method of class {@code Object} performs no

\* special action; it simply returns normally. Subclasses of

\* {@code Object} may override this definition.

\* <p>

\* The Java programming language does not guarantee which thread will

\* invoke the {@code finalize} method for any given object. It is

\* guaranteed, however, that the thread that invokes finalize will not

\* be holding any user-visible synchronization locks when finalize is

\* invoked. If an uncaught exception is thrown by the finalize method,

\* the exception is ignored and finalization of that object terminates.

\* <p>

\* After the {@code finalize} method has been invoked for an object, no

\* further action is taken until the Java virtual machine has again

\* determined that there is no longer any means by which this object can

\* be accessed by any thread that has not yet died, including possible

\* actions by other objects or classes which are ready to be finalized,

\* at which point the object may be discarded.

\* <p>

\* The {@code finalize} method is never invoked more than once by a Java

\* virtual machine for any given object.

\* <p>

\* Any exception thrown by the {@code finalize} method causes

\* the finalization of this object to be halted, but is otherwise

\* ignored.

\*

\* **@throws** Throwable the {@code Exception} raised by this method

\*/

**protected** **void** finalize() **throws** Throwable { }

}

# Java (JVM) Memory Model

## Permanent Generation

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

## Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

## Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

## Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

## Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

## Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

|  |  |
| --- | --- |
| VM Switch | VM Switch Description |
| -Xms | For setting the initial heap size when JVM starts |
| -Xmx | For setting the maximum heap size. |
| -Xmn | For setting the size of the Young Generation, rest of the space goes for Old Generation. |
| -XX:PermGen | For setting the initial size of the Permanent Generation memory |
| -XX:MaxPermGen | For setting the maximum size of Perm Gen |
| -XX:SurvivorRatio | For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8. |
| -XX:NewRatio | For providing ratio of old/new generation sizes. The default value is 2. |

### Java Garbage Collection Types

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.

Serial GC is useful in client-machines such as our simple stand alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

1. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using -XX:ParallelGCThreads=n JVM option.

Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

1. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
2. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

CMS collector on young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can’t afford longer pause times. We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it’s long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”.

# Java Heap Memory vs Stack Memory Difference

Java Heap Memory

Heap memory is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

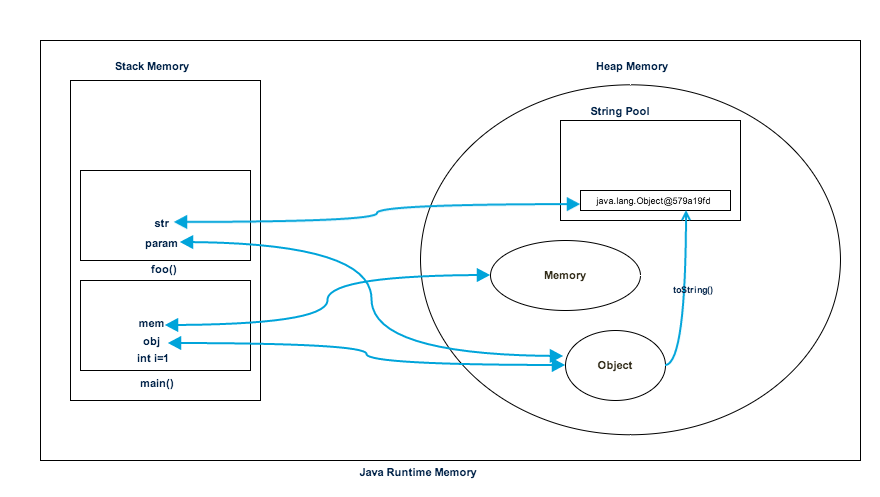
Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.  
Stack memory size is very less compared to Heap memory.

Let’s understand the Heap and Stack memory usage with a simple program.

|  |  |
| --- | --- |
| Memory.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.journaldev.test;    public class Memory {        public static void main(String[] args) { // Line 1          int i=1; // Line 2          Object obj = new Object(); // Line 3          Memory mem = new Memory(); // Line 4          mem.foo(obj); // Line 5      } // Line 9        private void foo(Object param) { // Line 6          String str = param.toString(); //// Line 7          System.out.println(str);      } // Line 8    } |

Below image shows the Stack and Heap memory with reference to above program and how they are being used to store primitive, Objects and reference variables.



Let’s go through the steps of execution of the program.

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the [String Pool](http://www.journaldev.com/797/what-is-java-string-pool) in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

# Class Loader

Class loading is the process of loading the class from byte code ( class file).

Class loader needs to find the bytecode at a particular location to memory before initialization.

Class loader will load the class, memory will be allocated loaded class in method area ( class area).

## Class loading principles

* 1. Delegation principle
  2. Visibility principle
  3. Uniqueness principle

Delegation – for loading a particular class, the class loader will check whether the class is already loaded in cache, if not then they will delegate the request to their parent class loader.

Visibility- classes loaded by parent class loader will be visible to child class loaders but classes loaded by child class loaders will not be visible to parent class loader.

Uniqueness – if a class is loaded by a particular class loader, then the child class loader will never load the same class again.

## Types of class loaders

* 1. Bootstrap class loader
  2. Extension class loader
  3. Application class loader
  4. Custom class loader ( inherit any of the class loaders, implement loadclass or findclass )

Boot strap class loader, loads classes from class library of java and jre. It is written in native language and not part of java api. So it cant be accessed or displayed. So getclassloader() will return null.

Extention class loader, load classes with additional functionality to java. Like for mysql. Classes loaded from extension loader ext folder of jre/lib folder of jdk or jre.

Application class loader, loads class from classpath , it is a static inner class in launcher.

**Improving Performance of java application**

1. Use bit shift operator for multiplying and divide by 2 , computers are very fast with bitwise operation.  
  
2. Use stringbuffer in place of string if you are doing lots of string manipulation it will reduce memory by avoiding creating lots of string garbage. If you are using java5 then consider stringbuilder but that is not synchronized so beware.  
  
3. try to make variable , class , method final whenever possible that’s allow compiler to do lots of optimization e.g. compile time binding so you will get faster output.  
  
4. static methods are bonded compile time while non static methods are resolved at runtime based on object type so static method will be faster than non static.

5. don't call methods in for loop for checking condition e.g. length() size() etc.  
instead of doing this , use modified version  
for(int i=0; i<vector.size();i++)

int size = vector.size();

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

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

**Bitwise and Bit Shift Operators**

~ Unary bitwise complement

<< Signed left shift

>> Signed right shift

>>> Unsigned right shift

& Bitwise AND

^ Bitwise exclusive OR

| Bitwise inclusive OR

*Left shift: value << num*

**// Left shifting as a quick way to multiply by 2.**

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative

|  |  |  |
| --- | --- | --- |
| **Shift Operators** | | |
| **Operator** | **Use** | **Description** |
| << | op1 << op2 | Shifts bits of op1 left by distance op2; fills with 0 bits on the right side |
| >> | op1 >> op2 | Shifts bits of op1 right by distance op2; fills with highest (sign) bit on the left side |
| >>> | op1 >>> op2 | Shifts bits of op1 right by distance op2; fills with 0 bits on the left side |

Shifting is basically taking the binary equivalent of a number and moving the bit pattern left or right.

**Left Shift Operator <<**

This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right. For example,

* Consider 12 whose binary equivalent is 00000000 00000000 00000000 00001100 (binary for 32 bits). Remember that the type operands (such as *byte* and *short*) are implicitly promoted to the *int* type before any shift operators are applied. Let's shift it left 2 times.

00000000 00000000 00000000 00001100 << 2(times)

When you shift left ( << ) the void left behind by the shift is filled by zero's. This is the result of (12<<2):

00000000 00000000 00000000 00110000 (48)

* Consider -12 whose binary equivalent is 11111111 11111111 11111111 11110100 (binary for 32 bits). Let's shift it left 2 times.

11111111 11111111 11111111 11110100 << 2(times)

When you shift left ( << ) the void left behind by the shift is filled by zero's. This is the result of (-12<<2):

11111111 11111111 11111111 11010000 (-48)

Let's do -12<<28 and the result is

01000000 00000000 00000000 00000000 (1073741824)

**Right Shift Operator >>**

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.

* Consider 12 whose binary equivalent is 00000000 00000000 00000000 00001100 (binary for 32 bits). Remember that the type operands (such as *byte* and *short*) are implicitly promoted to the *int* type before any shift operators are applied. Let's shift it right 2 times.

00000000 00000000 00000000 00001100 >> 2(times)

When shifting right ( >> ) the leftmost bits exposed by the right shift are filled in with previous contents of the leftmost bit. This is the result of (12>>2):

00000000 00000000 00000000 00000011 (3)

* Consider -12 whose binary equivalent is 11111111 11111111 11111111 11110100 (binary for 32 bits). Let's shift it right 2 times.

11111111 11111111 11111111 11110100 >> 2(times)

When shifting right ( >> ) the leftmost bits exposed by the right shift are filled in with previous contents of the leftmost bit. This is the result of (-12>>2):

11111111 11111111 11111111 11111101 (-3)

Note that right shifting ( >> ) always preserves the sign of the original number i.e. to say that a negative number will stay negative while a positive number will stay positive after a right shift ( >> ).

**Unsigned Right Shift Operator >>>**

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

When shifting right ( >> ) the leftmost bits exposed by the right shift are filled in with previous contents of the leftmost bit. BUT, the >>> unsigned right shift operator always fill zero's and only zero's no matter positive and negative number. For example,

* Let's do 00000000 00000000 00000000 00001100 >>> 2 (12>>>2) and the result is 00000000 00000000 00000000 00000011 (3).
* Let's do 11111111 11111111 11111111 11110100 >>> 2 (-12>>>2) and the result is 00111111 11111111 11111111 11111101 (1073741821).