Multithreading

Defining, Instantiating and Starting Threads.

1. You can create a Thread by extending the Thread class and overriding the public void run() method.
2. You can also create a Thread by calling the Thread constructor that accepts a Runnable instance. The Runnable object is called the target of the thread.
3. You can call the start() method only once on the thread. If you call it more than once you will get an IllegalStateException.
4. You can create multiple Thread objects with the same Runnable instance as the target.
5. When a Thread object is created, it does not become a new thread of execution until its start() method is called. When a thread has been created but not yet started, it is still in the new state and is not considered Alive.

Transitioning between thread states.

1. When a new thread is started, it will always enter the Runnable state.
2. Thread scheduler can move the thread back and forth between and the running state and the runnable state.
3. For a typical single processor machine, there will always be only 1 thread running at a time, although many thread would be in the runnable state.
4. There is no guarantee that the order in which threads are started will determine the order in which threads run.
5. There is no guarantee that thread that started first will finish first.
6. There is no guarantee that threads will take turns in a fair way. Using the sleep() method can prevent 1 thread from hogging the running process while another thread starves. Though yield() method works well in most cases to make sure that threads take turns in a fair way.
7. Threads may enter blocked/waiting state on calling sleep(), join() or wait() method.
8. A running thread may enter blocked / waiting state if it cannot acquire lock for a synchronized block of code.
9. When the sleep / wait is over, thread will move from blocked/waiting to runnable state.
10. A dead thread cannot be started again.

Concurrent access problems and synchronized threads

1. Synchronized methods prevent more than 1 thread from accessing an object's critical code simultaneously.
2. You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
3. To synchronize a block of code, you must specify an object whose lock you want to synchronize on.
4. While only 1 thread can be accessing synchronized code of a single instance, multiple threads might be accessing its unsynchronized code.
5. When a thread goes to sleep, its lock will be unavailable to other threads.
6. Static methods can be synchronized, using the lock from java.lang.Class instance representing that class.

Communicating objects with wait, notify and notifyAll

1. Wait(), notify() and notifyAll() must all be called from within a synchronized context, else it will throw IllegalMonitorStateException.
2. A thread invokes wait(), notify() and notifyAll() from a particular object and the thread must hold the lock for that object.
3. Wait() method says, there is nothing for me to do, so put me in a waiting state and call me back using notify() when there is something for which I care.
4. Notify() method will put a single waiting thread in to runnable state.
5. Notify() method cannot specify which thread to notify.
6. NotifyAll() works in the same way as notify().

Sleep, yield and join

1. Sleeping is used to delay execution for a period of time and no locks are released when a thread goes to sleep.
2. A sleeping thread is guaranteed to sleep at least for the time specified in the argument to the sleep method, unless it is interrupted. But there is no guarantee as to when the newly awakened thread will return back to running.
3. Sleep() is a static method of the thread class, which causes currently executing thread to go to sleep. One thread cannot tell another thread to sleep.

   ```
   Class SomeClass extendsThread{
           Public void run(){
                   Thread t = Thread.currentThread();
                   t.sleep(1000);
                   Thread.sleep(1000);
                   this.sleep(1000);
           }
   }
   ```

   all 3 sleep calls above are equivalent, all will cause current thread to sleep for a second.

Thread vs Runnable

1. Implementing Runnable allows you to extend other classes as well.
2. A key benefit of Runnable is that you are architecturally separating the “job” from the “runner”.
3. In older versions of Java (1.4 and less) if you create a Thread and not call it’s start method, you are unintentionally creating a memory leak. This issue is fixed in 1.5 but not in lower versions. The issue is that, at construction time, a Thread is added to a list of references in an internal thread table. It won't get removed from that list until its start() method has completed. As long as that reference is there, it won't get garbage collected.
4. Inheriting all Thread methods are additional overhead just for representing a Task which can be done easily with Runnable.
5. Yes: implements `Runnable` is the preferred way to do it, IMO. You're not really specialising the thread's behaviour. You're just giving it something to run. That means composition is the *philosophically* "purer" way to go.

Use Countdown latch when the main thread has to wait for one or more threads to finish processing before it can start. Classical example using CountdownLatch is a server side core Java application which uses services architecture, where multiple services are provided by multiple threads and application cannot start processing until all services have started successfully.
One advantage / disadvantage of countdown latch is that it is not reusable once count reaches 0.

The `java.util.concurrent.ConcurrentMap` interface represents a Map which is capable of handling concurrent access (puts and gets) to it.
The `ConcurrentMap` has a few extra atomic methods in addition to the methods it inherits from its superinterface, `java.util.Map`.
Since `ConcurrentMap` is an interface, you need to use one of its implementations in order to use it. The `java.util.concurrent` package contains the following implementations of the `ConcurrentMap` interface:
ConcurrentHashMap
The `ConcurrentHashMap` is very similar to the `java.util.HashTable` class, except that `ConcurrentHashMap` offers better concurrency than `HashTable` does. `ConcurrentHashMap` does not lock the `Map` while you are reading from it. Additionally, `ConcurrentHashMap` does not lock the entire `Map` when writing to it. It only locks the part of the `Map` that is being written to, internally.

Another difference is that `ConcurrentHashMap` does not throw `ConcurrentModificationException` if the `ConcurrentHashMap` is changed while being iterated. The `Iterator` is not designed to be used by more than one thread though.

Synchronized HashMap:
Each method is synchronized using an object level lock. So the get and put methods on synchMap acquire a lock. Locking the entire collection is a performance overhead. While one thread holds on to the lock, no other thread can use the collection.

ConcurrentHashMap was introduced in JDK 5.
There is no locking at the object level,The locking is at a much finer granularity. For a ConcurrentHashMap, the locks **may be at a hashmap bucket level**.
The effect of lower level locking is that you **can have concurrent readers and writers** which is not possible for synchronized collections. This leads to much more **scalability**.
ConcurrentHashMap does not throw a **ConcurrentModificationException** if one thread tries to modify it while another is iterating over it.

Difference between HashMap and ConcurrentHashMap

1. HashMap is not thread safe and cannot be used in a multithreaded environment without external synchronization. ConcurrentHashMap is thread safe and can be used in multithreaded environment without external synchronization.
2. HashMap can be synchronized by using Collections.synchronizedMap(myMap). The resultant map will be similar to HashTable where the entire map would be locked for each get / put operation. ConcurrentHashMap uses a ReadWriteReentrant lock, which allows reads to be performed by multiple threads as long as no writes are being performed, uses a pair of associated locks, 1 for reading and another for writing. Further it locks only a part of the Map say a bucket, so other parts of the Map are available to other threads for reading / writing.
3. ConcurrentHashMap does not throw a ConcurrentModification exception when the map the structurally modified while the iteration is in progress. However iterators and enumerators returned by CHM are designed to be used by only 1 thread. Map will not reflect any modifications made after the iterator is constructed.
4. CHM provides better scalabiity in multi-.. that synchronized HM whereres HM provides slightly better scalability in singlethreadedenv.

http://www.ibm.com/developerworks/java/library/j-jtp08223/index.html

Executors.newCachedThreadPool : Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to `execute` will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using `ThreadPoolExecutor` constructors.
In general, if ClassName.method is a static method of ClassName, and x is an expression whose type is ClassName, then you can use x.method() and it will be the same as calling ClassName.method(). It doesn't matter what the value of x is; the value is discarded. **It will work even if x is null.**

--> java.util.concurrent.ThreadPoolExecutor

An `ExecutorService` that executes each submitted task using one of possibly several pooled threads, normally configured using `Executors` factory methods.
Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each `ThreadPoolExecutor` also maintains some basic statistics, such as the number of completed tasks.
To be useful across a wide range of contexts, this class provides many adjustable parameters an d extensibility hooks. However, programmers are urged to use the more convenient `Executors` factory methods `Executors.newCachedThreadPool` (unbounded thread pool, with automatic thread reclamation), `Executors.newFixedThreadPool` (fixed size thread pool) and `Executors.newSingleThreadExecutor` (single background thread), that preconfigure settings for the most common usage scenarios. Otherwise, use the following guide when manually configuring and tuning this class:
Core and maximum pool sizes
A `ThreadPoolExecutor` will automatically adjust the pool size (see `getPoolSize`) according to the bounds set by corePoolSize (see `getCorePoolSize`) and maximumPoolSize (see `getMaximumPoolSize`). When a new task is submitted in method `execute(Runnable)`, and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle. If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full. By setting corePoolSize and maximumPoolSize the same, you create a fixed-size thread pool. By setting maximumPoolSize to an essentially unbounded value such as `Integer.MAX_VALUE`, you allow the pool to accommodate an arbitrary number of concurrent tasks. Most typically, core and maximum pool sizes are set only upon construction, but they may also be changed dynamically using `setCorePoolSize` and `setMaximumPoolSize`.
On-demand construction
By default, even core threads are initially created and started only when new tasks arrive, but this can be overridden dynamically using method `prestartCoreThread` or `prestartAllCoreThreads`. You probably want to prestart threads if you construct the pool with a non-empty queue.
Creating new threads
New threads are created using a `ThreadFactory`. If not otherwise specified, a `Executors.defaultThreadFactory` is used, that creates threads to all be in the same `ThreadGroup` and with the same `NORM_PRIORITY` priority and non-daemon status. By supplying a different ThreadFactory, you can alter the thread's name, thread group, priority, daemon status, etc. If a `ThreadFactory` fails to create a thread when asked by returning null from `newThread`, the executor will continue, but might not be able to execute any tasks. Threads should possess the "modifyThread" `RuntimePermission`. If worker threads or other threads using the pool do not possess this permiss ion, service may be degraded: configuration changes may not take effect in a timely manner, and a shutdown pool may remain in a state in which termination is possible but not completed.
Keep-alive times
If the pool currently has more than corePoolSize thre ads, excess threads will be terminated if they have been idle for more than the keepAliveTime (see `getKeepAliveTime(TimeUnit)`). This provides a means of reducing resource consumption when the pool is not being actively used. If the pool becomes more active later, new t hreads will be constructed. This parameter can also be changed dynamically using method `setKeepAliveTime(long, TimeUnit)`. Using a value of `Long.MAX_VALUE` `TimeUnit.NANOSECONDS` effectively disables idle threads from ever terminating prior to shut down. By default, the keep-alive policy applies only when there are more than corePoolSize threads. But method `allowCoreThreadTimeOut(boolean)` can be used to apply this time-out policy to core threads as well, so long as the keepAliveTime value is non-zero.
Queuing
Any `BlockingQueue` may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:
If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.

If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.
If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.
There are three general strategies for queuing:
Direct handoffs. A good default choice for a work queue is a `SynchronousQueue` that hands off tasks to threads without otherwise holding them. Here, an attempt to queue a task will fail if no threads are immediately available to run it, so a new thread will be constructed. This policy avoids lockups when handling sets of requests that might have internal dependencies. Direct handoffs generally require unbounded maximumPoolSizes to avoid rejection of new submitted tasks. This in turn admits the possibility of unbounded thread growth when commands continue to arrive on average faster than they can be processed.
Unbounded queues. Using an unbounded queue (for example a `LinkedBlockingQueue` without a predefined capacity) will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created. (And the value of the maximumPoolSize therefore doesn't have any effect.) This may be appropriate when each task is completely independent of others, so tasks cannot affect each others execution; for example, in a web page server. While this style of queuing can be useful in smoothing out transient bursts of requests, it admits the possibility of unbounded work queue growth when commands continue to arrive on average faster than they can be processed.
Bounded queues. A bounded queue (for example, an `ArrayBlockingQueue`) helps prevent resource exhaustion when used with finite maximumPoolSizes, but can be more difficult to tune and control. Queue sizes and maximum pool sizes may be traded off for each other: Using large queues and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. If tasks frequently block (for example if they are I/O bound), a system may be able to schedule time for more threads than you otherwise allow. Use of small queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.
Rejected tasks
New tasks submitted in method `execute(Runnable)` will be *rejected* when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the `execute` method invokes the `RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` method of its `RejectedExecutionHandler`. Four predefined handler policies are provided:
In the default `ThreadPoolExecutor.AbortPolicy`, the handler throws a runtime `RejectedExecutionException` upon rejection.
In `ThreadPoolExecutor.CallerRunsPolicy`, the thread that invokes `execute` itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
In `ThreadPoolExecutor.DiscardPolicy`, a task that cannot be executed is simply dropped.
In `ThreadPoolExecutor.DiscardOldestPolicy`, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)
It is possible to define and use other kinds of `RejectedExecutionHandler` classes. Doing so requires some care especially when policies are designed to work only under particular capacity or queuing policies.
Hook methods
This class provides `protected` overridable `beforeExecute(Thread, Runnable)` and `afterExecute(Runnable, Throwable)` methods that are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries. Additionally, method `terminated` can be overridden to perform any special processing that needs to be done once the Executor has fully terminated.
If hook or callback methods throw exceptions, internal worker threads may in turn fail and abruptly terminate.
Queue maintenance

Method `getQueue()` allows access to the work queue for purposes of monitoring and debugging. Use of this method for any other purpose is strongly discouraged. Two supplied methods, `remove(Runnable)` and `purge` are available to assist in storage reclamation when large numbers of queued tasks become cancelled.

Finalization

A pool that is no longer referenced in a program *AND* has no remaining threads will be `shutdown` automatically. If you would like to ensure that unreferenced pools are reclaimed even if users forget to call `shutdown`, then you must arrange that unused threads eventually die, by setting appropriate keep-alive times, using a lower bound of zero core threads and/or setting `allowCoreThreadTimeOut(boolean)`.

Extension example. Most extensions of this class override one or more of the protected hook methods. For example, here is a subclass that adds a simple pause/resume feature:

```
class PausableThreadPoolExecutor extends ThreadPoolExecutor {
   private boolean isPaused;
   private ReentrantLock pauseLock = new ReentrantLock();
   private Condition unpaused = pauseLock.newCondition();

  public PausableThreadPoolExecutor(...) { super(...); }

  protected void beforeExecute(Thread t, Runnable r) {
    super.beforeExecute(t, r);
    pauseLock.lock();
    try {
      while (isPaused) unpaused.await();
    } catch (InterruptedException ie) {
      t.interrupt();
    } finally {
      pauseLock.unlock();
    }
  }

  public void pause() {
    pauseLock.lock();
    try {
      isPaused = true;
    } finally {
      pauseLock.unlock();
    }
  }

  public void resume() {
    pauseLock.lock();
    try {
      isPaused = false;
      unpaused.signalAll();
    } finally {
      pauseLock.unlock();
    }
  }
}}
```

protected void	**afterExecute**(Runnable r, Throwable t) Method invoked upon completion of execution of the given Runnable.
void	**allowCoreThreadTimeOut**(boolean value) Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.
boolean	**allowsCoreThreadTimeOut**() Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive.
boolean	**awaitTermination**(long timeout, TimeUnit unit) Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
protected void	**beforeExecute**(Thread t, Runnable r) Method invoked prior to executing the given Runnable in the given thread.
void	**execute**(Runnable command) Executes the given task sometime in the future.
protected void	**finalize**() Invokes `shutdown` when this executor is no longer referenced and it has no threads.
int	**getActiveCount**() Returns the approximate number of threads that are actively executing tasks.
long	**getCompletedTaskCount**() Returns the approximate total number of tasks that have completed execution.
int	**getCorePoolSize**() Returns the core number of threads.
long	**getKeepAliveTime**(TimeUnit unit) Returns the thread keep-alive time, which is the amount of time that threads in excess of the core pool size may remain idle before being terminated.

`int`	**`getLargestPoolSize`**`()` Returns the largest number of threads that have ever simultaneously been in the pool.
`int`	**`getMaximumPoolSize`**`()` Returns the maximum allowed number of threads.
`int`	**`getPoolSize`**`()` Returns the current number of threads in the pool.
`BlockingQueue<Runnable>`	**`getQueue`**`()` Returns the task queue used by this executor.
`RejectedExecutionHandler`	**`getRejectedExecutionHandler`**`()` Returns the current handler for unexecutable tasks.
`long`	**`getTaskCount`**`()` Returns the approximate total number of tasks that have ever been scheduled for execution.
`ThreadFactory`	**`getThreadFactory`**`()` Returns the thread factory used to create new threads.
`boolean`	**`isShutdown`**`()` Returns `true` if this executor has been shut down.
`boolean`	**`isTerminated`**`()` Returns `true` if all tasks have completed following shut down.
`boolean`	**`isTerminating`**`()` Returns true if this executor is in the process of terminating after `shutdown()` or `shutdownNow()` but has not completely terminated.
`int`	**`prestartAllCoreThreads`**`()` Starts all core threads, causing them to idly wait for work.
`boolean`	**`prestartCoreThread`**`()`

	Starts a core thread, causing it to idly wait for work.
`void`	**`purge`**`()` Tries to remove from the work queue all `Future` tasks that have been cancelled.
`boolean`	**`remove`**`(Runnable task)` Removes this task from the executor's internal queue if it is present, thus causing it not to be run if it has not already started.
`void`	**`setCorePoolSize`**`(int corePoolSize)` Sets the core number of threads.
`void`	**`setKeepAliveTime`**`(long time, TimeUnit unit)` Sets the time limit for which threads may remain idle before being terminated.
`void`	**`setMaximumPoolSize`**`(int maximumPoolSize)` Sets the maximum allowed number of threads.
`void`	**`setRejectedExecutionHandler`**`(RejectedExecutionHandler handler)` Sets a new handler for unexecutable tasks.
`void`	**`setThreadFactory`**`(ThreadFactory threadFactory)` Sets the thread factory used to create new threads.
`void`	**`shutdown`**`()` Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
`List<Runnable>`	**`shutdownNow`**`()` Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
`protected void`	**`terminated`**`()` Method invoked when the Executor has terminated.
`String`	**`toString`**`()`

Returns a string identifying this pool, as well as its state, including indications of run state and estimated worker and task counts.

ThreadPoolExecutor`(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)`

Creates a new `ThreadPoolExecutor` with the given initial parameters and default thread factory and rejected execution handler.

ThreadPoolExecutor`(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)`

Creates a new `ThreadPoolExecutor` with the given initial parameters and default thread factory.

ThreadPoolExecutor`(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory)`

Creates a new `ThreadPoolExecutor` with the given initial parameters and default rejected execution handler.

ThreadPoolExecutor`(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)`

Creates a new `ThreadPoolExecutor` with the given initial parameters.

```
String s = null;
String t = s.format ("%08x", someInteger); // works fine
                                  // (String.format is a static method)
```

Capacity is the number of buckets in the HashTable and initial capacity is the capacity at the time of creation of the HashTable.
Load factor is the measure of how full the hashtable is allowed to get before the capacity is increased dynamically. When the number of entries in the HashTable exceeds the product of load factor and initial capacity, the HashTable is rehashed (internal data structures are rebuilt) so that the HashTable has twice the number of buckets.

HashMap code
http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap.0table

Key differences between Serializable and Externalizable

1. Marker interface: Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().
2. Serialization process: Default Serialization process will be kicked-in for classes implementing Serializable interface. Programmer defined Serialization process will be kicked-in for classes implementing Externalizable interface.
3. Maintenance: If any new Non-Serializable is added to Serializable Object, Serialization process will break at runtime. But it does not happen if you implement Externalizable. That new attribute won't be serialized unless you explicitly write it.
4. Backward Compatibility and Control: If you have to support multiple versions, you can have full control with Externalizable interface. You can support different versions of your object. If you implement Externalizable, it's your responsibility to serialize super class
5. public No-arg constructor: Serializable uses reflection to construct object and does not require no arg constructor. But Externalizable demands public no-arg constructor.

Refer to blog by Hitesh Garg for more details.

In earlier version of Java, reflection was very slow, and so serializing large object graphs (e.g. in client-server RMI applications) was a bit of a performance problem. To handle this situation, the java.io.Externalizable interface was provided, which is like java.io.Serializable but with custom-written mechanisms to perform the marshalling and unmarshalling functions (you need to implement readExternal and writeExternal methods on your class). This gives you the means to get around the reflection performance bottleneck.

In recent versions of Java (1.3 onwards, certainly) the performance of reflection is vastly better than it used to be, and so this is much less of a problem. I suspect you'd be hard-pressed to get a meaningful benefit from Externalizable with a modern JVM.

the transient keyword also closes the gap between the two.

If you only want to serialize part of your object, just set specific fields as transient, marking them as not to be persisted , and implement Serializable.

HashSet is internally represented using a HashMap. It is backed by a transient HashMap. When you add an object to a HashMap, it will add the key

What is CopyOnWrirteArrayList? How is it different from ArrayList?
CopyOnWriteArrayList is a thread safe variant of ArrayList in which all mutative operations like get, set, etc. are performed on a fresh copy of the underlying array.
This is ordinarily too costly, but may be *more* efficient that alternatives when traversal operations vastly outnumber mutations, and you cannot or donot want to synchronize the traversals and yet need to preclude interference among multiple concurrent threads.
The “snapshot” style iterator method uses a reference to the state of the array, at the time the iterator was created. This array never changes for the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException. The iterator will not reflect additions, removals or any other changes since the iterator was created. Element changing operations om the iterator themselves(remove, set or add) are not supported and throw UnsupportedOperationException. Added in JDK 1.5

How does remove method work in HashMap?

```
/**
     * 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;
    }
/**
     * 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);
    }
```

Wrapper classes correlate to primitive types.
They have two main functions:

1. Wrap primitives so that they can be handled like objects.
2. To provide utility methods for primitives. (mostly conversions)

Three most important method families:

1. parseXXX: Accepts a String argument, returns corresponding primitive, throws NumberFormatException.
2. valueOf: Accepts a String argument, returns corresponging wrapper, throws NumberFormatException.
3. xxxValue: Accepts no arguments, returns primitive.

Wrapper constructors can take Strings or primitive, except for Char which only takes char.

4. Radix refers to bases. Octal radix is 8, hexadecimal radix is 16, decimal radix is 10.

Every Enum has a static method called values() which retursn an array of enum values in the order they are declared.
As of 5.0 Java lets you restrict the value of a variable to one of the few pre-defined values. Each value in the pre-defined list is called an enum.

```
System.out.println(test1.getCoffeeSize());
Output: BIG
```

Benefits:

1. Free compile time checking of valid values.
2. Self-documenting code, as you can use auto-complete to see valid enum values.
3. Another, that I find more important, is that you can attach metadata to enum values in Java. For example, you could use an enum to define the set of legal operations for a webservice, and then attach metadata for the type of request and data class:
```
AddItem(HttpMethod.POST, ProductEntry.class),
```
4. You can use enums to effectively implement Singletons.
   Public enum Elvis {
      INSTANCE
   }
   But this approach will cause load time initialization which can be painful to debug. So we are better off with it.
5. Making enum a reference type that can contain fixed set of constants has led to efficient Map implementation like EnumMap and Set implementation like EnumSet (JDK classes).
   From javadoc of EnumMap :
   A specialized Map implementation for use with enum type keys. All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient.
   EnumMap combines richness and type safety of Map with the speed of an array (Effective Java).
6. You can use enums as data values in beans and get validation for free.
```
public class Person
{
    private enum Gender { M, F }

    private String _firstName;
    private String _lastName;
    private Date _birthday;
    private Gender _gender;

    // ...

    public String getGender()
    {
        return _gender.toString();
    }

    public void setGender(String value)
    {
        _gender = Gender.valueOf(value);
    }
}
Person person = new Person();

// this is OK
person.setGender("M");

// this isn't
person.setGender("FILE_NOT_FOUND");
```

The second call throws `IllegalArgumentException`. It would also throw if you passed in a lowercase "m", so you usually need to add some code to your setter:

```
_gender = Gender.valueOf(value.toUpperCase());
```

System.out.println(CoffeeSize.valueOf("8"));
Output: RTE. IllegalArgumentException

Things that cannot be marked static:

1. Constructors.
2. Outer classes.
3. Local variables
4. Interfaces
5. Method local inner classes.
6. Inner class methods and instance variables.

Things that can be marked static:

1. Methods
2. Instance variables
3. Initialization blocks
4. A classes nested within another class but not within another method.

System class cannot be instantiated. It provides a number a useful classes and methods. Among the facilities provided by the System class are standard input, standard output and error output streams, a means of loading files and libraries, access to externally defined properties and environment variables and a utility method for quickly copying a portion of an array.

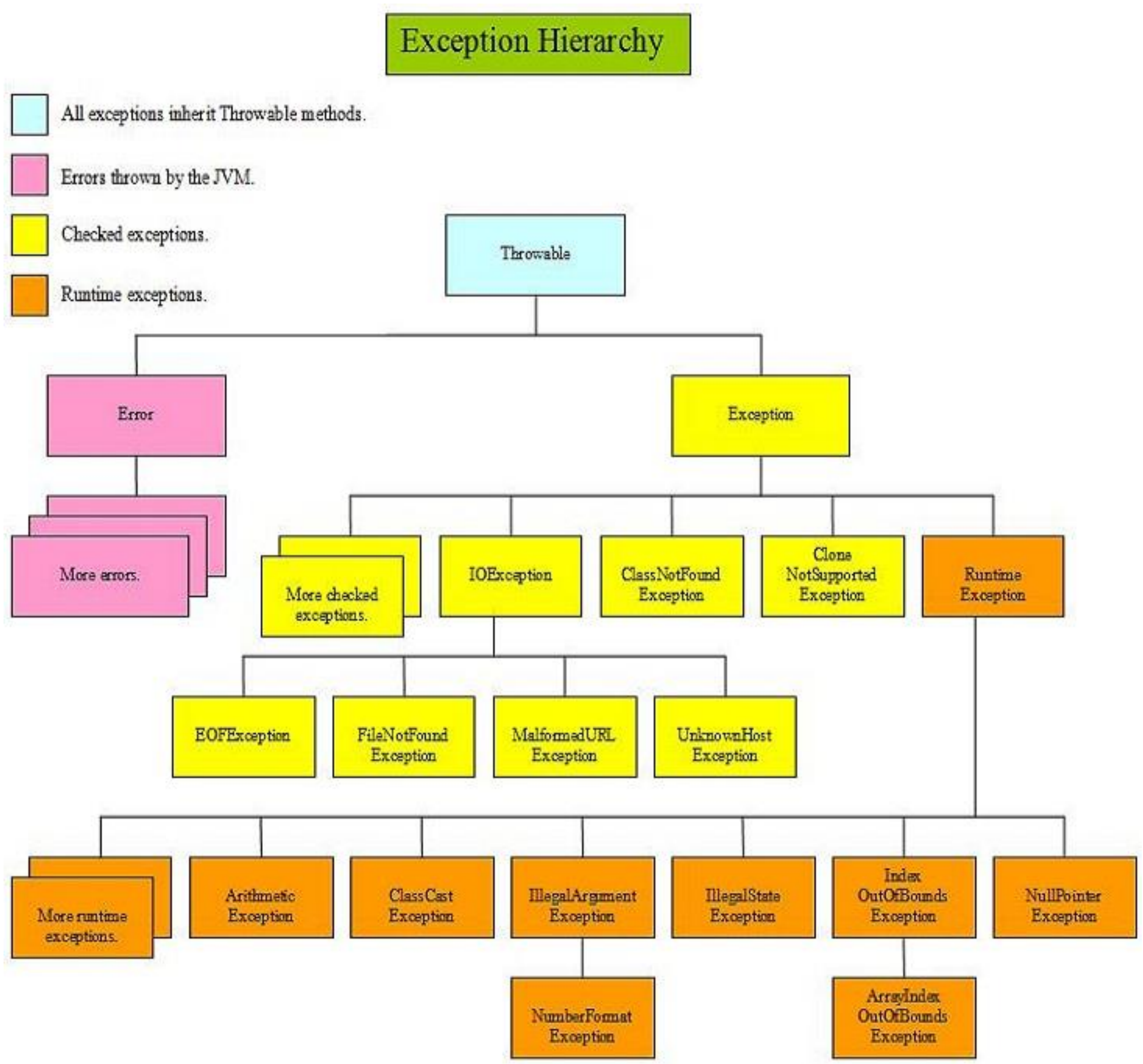
Does constructor create objects?

No. new operator creates object. Calling the constructor is a later step to initialize the instance variables after memory has been allocated to the object.

Does Java support multiple inheritance?
Interface does not facilitate inheritance. Hence implementing multiple interfaces does not make multiple inheritance. Java does not support multiple inheritance.

Difference between == and equals?
== is a operator. Equals() is a method defined in object class. == operator checks if both operands point to the same object in memory. If operands are primitives then it checks if both operands have same value. Equals() method check if objects are logically eqaual. Equals might return true even if both references point to different objects in memory, they can still be logically equivalent. Equals cannot be applied on primitives.

Differences between checked and unchecked exceptions?
Checked exceptions are subclasses of Exception class. If a method throws them, then they must be catched with a try/catch block or method must declare that it throws the checked exception. Else you will get a compiler error.
Unchecked exceptions are subclasses of RuntimeException, you can throw them freely, without having to declare them in the throws clause.



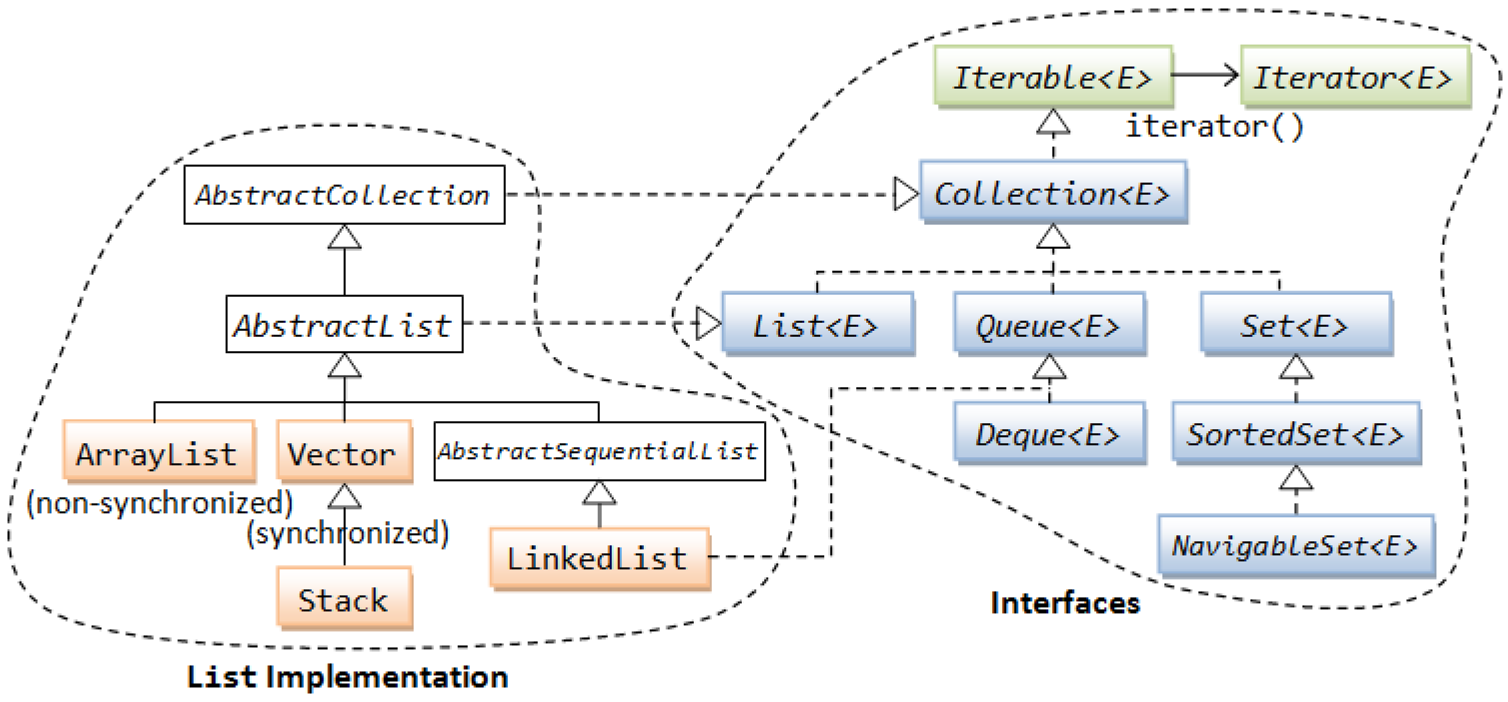
Method Declaration	Description
`public Throwable fillInStackTrace()`	Returns a `Throwable` object, that can be rethrown, which contains a completed *stack trace*.
`public Throwable getCause()`	Returns a `Throwable` object, or `null` if the cause of the exception is nonexistent or unknown.
`public String getLocalizedMessage()`	Returns a *localized* description of the exception.
`public String getMessage()`	Returns a description of the exception.
`public StackTraceElement[] getStackTrace()`	Returns an array of *stack trace* elements, where each

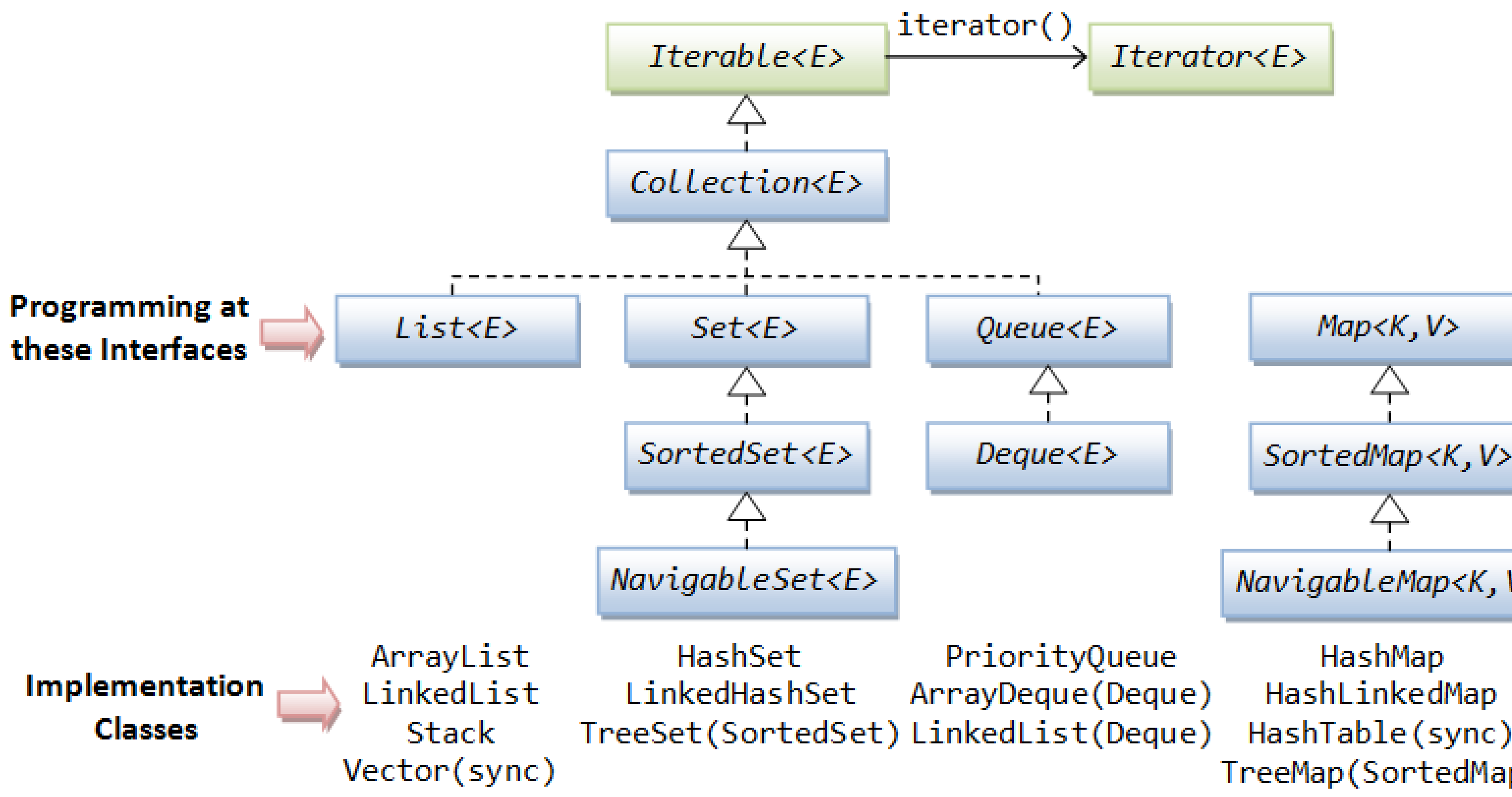
Method Declaration	Description
	element represents one stack frame.
`public Throwable initCause(Throwable cause)`	Set the *cause* of the *throwable*.
`public void printStackTrace()`	Displays the *stack trace* for the exception.
`public void printStackTrace(PrintStream s)`	Sends *stack trace* for the exception to specified stream.
`public void printStackTrace(PrintWriter s)`	Sends *stack trace* for the exception to specified print writer.
`public void setStackTrace(StackTraceElement[] stackTrace)`	Sets *stack trace* returned from `getStackTrace()` and printed by `printStackTrace()` and related methods.
`public String toString()`	Returns a short description of this *throwable* exception.

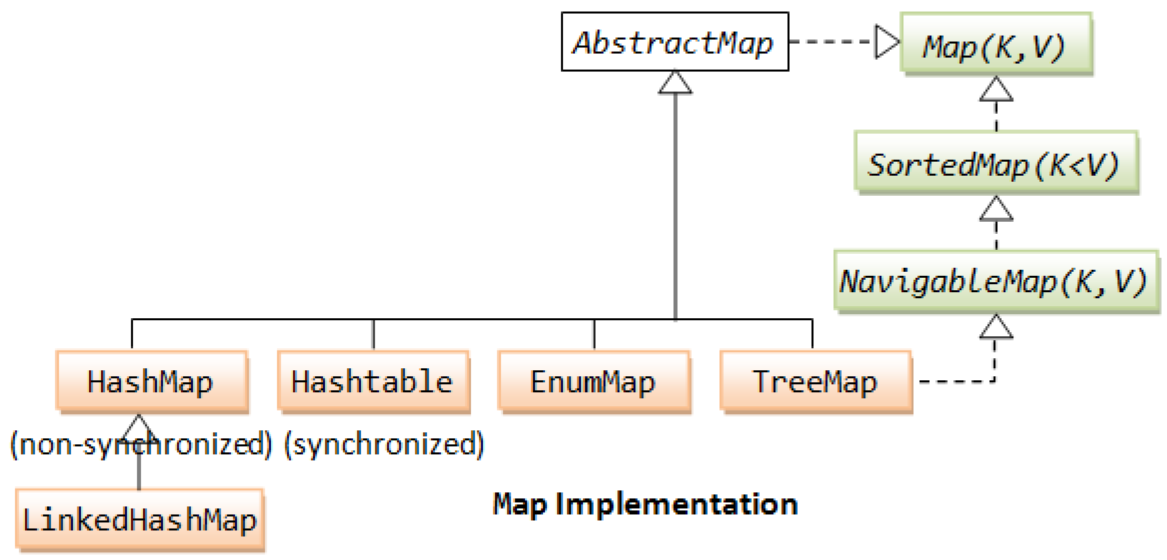
Checked Exception	**Description**
`ClassNotFoundException`	Class requested not found.
`CloneNotSupportedException`	Occurs when an attempt is made to clone an object of a class that doesn't implement the `Cloneable` interface.
`Exception`	Exception and subclasses thereof that an application may want to catch.
`IllegalAccessException`	Attempted access of class refused.
`InstantiationException`	Occurs when an attempt is made to instantiate an object from an abstract class or interface. An example of this exception is shown in the Abstraction lesson when we look at Abstract Classes.
`InterruptedException`	Occurs when a thread has been interrupted by another thread.
`NoSuchFieldException`	Field requested doesn't exist.
`NoSuchMethodException`	Method requested doesn't exist.
`Runtime Exception`	Description
`ArithmeticException`	Arithmetic exception occurred. An example of this exception is shown in the Exception Handling lesson when we look at Using try catch.
`ArrayIndexOutOfBoundsException`	Attempt to access array with an illegal index. An example of this exception is shown in the Arrays lesson when we look at java.lang Array Exceptions.

Checked Exception	Description
`ArrayStoreException`	Attempt to store incompatible type into object array element.
`ClassCastException`	Attempt to cast an object to a subclass of which it is not an instance. An example of this exception is shown in the Generics lesson when we look at a Raw Type/Generic Type Comparison.
`EnumConstantNotPresentException`	Attempt to access enum constant by name that is not contained within the enum type.
`IllegalArgumentException`	Method invoked with an illegal argument.
`IllegalMonitorStateException`	Illegal operation performed on an object monitor.
`IllegalStateException`	Method invoked while an application isn't in the correct state to receive it.
`IllegalThreadStateException`	Attempt to perform an operation on a monitor whilst not owning it.
`IndexOutOfBoundsException`	An index is out of bounds.
`NegativeArraySizeException`	Attempt to create an array with a negative index. An example of this exception is shown in the Arrays lesson when we look at java.lang Array Exceptions.
`NullPointerException`	Attempt to access an object with a null reference. An example of this exception is shown in the Reference Variables lesson when we look at The Heap.
`NumberFormatException`	Invalid attempt to convert the contents of a string to a numeric format. An example of this exception is shown in the Exception Handling lesson when we look at Using try catch finally.
`RuntimeException`	Runtimes Exception and subclasses thereof thrown by the JVM.
`SecurityException`	Thrown when a security violation has occurred.
`StringIndexOutOfBoundsException`	Attempt to access string with an illegal index from a String method. An example of this exception is shown in the String Class lesson when we look at The charAt() Method.
`TypeNotPresentException`	Type not found.

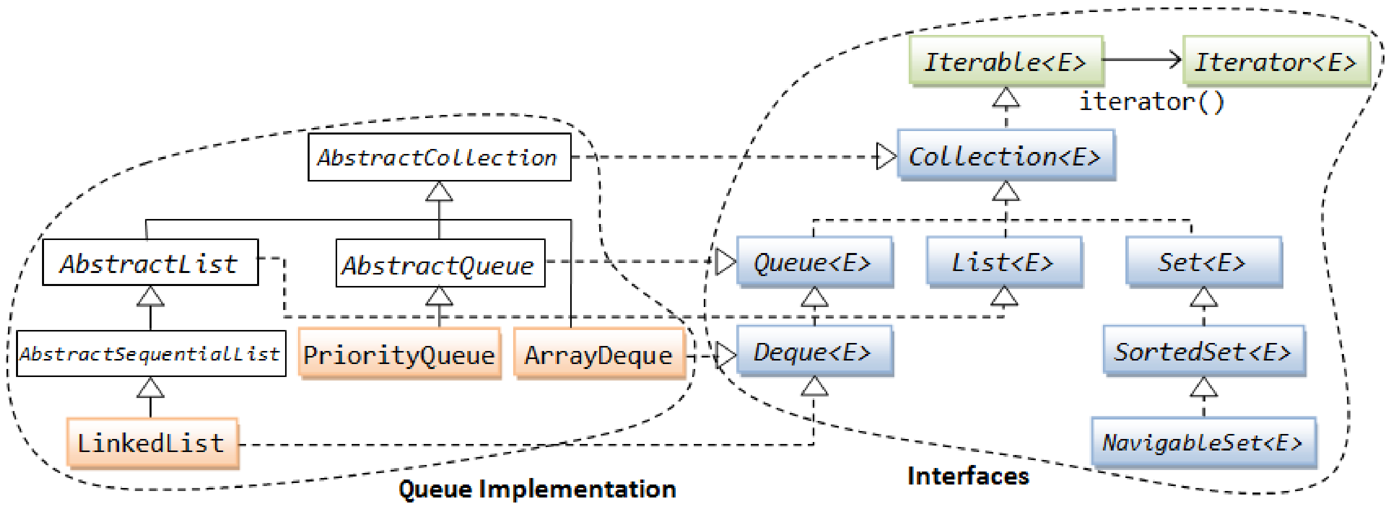
Checked Exception	Description
`UnsupportedOperationException`	Unsupported operation encountered.







Map Implementation



ArrayDeque / Deque does not allow you to insert anywhere in between. List allows you to insert anywhere in between. So does LinkedList.

Deque -

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Local variables live on the stack. Object and instance variables live on the heap.

Where does Java's String constant pool live, stack or heap?

The answer is neither. According to Java Virtual Machine specification, the area for storing String literals is in the runtime constant pool. It is a subset of the method area which is allocated on per class or per interface basis. Method area stores method and instance data, method and constructor code including special methods that are used in class and instance initialization and interface initialization. It is not tied up to the instances at all. It is in the method area. Although logically method area is part of the heap, it is not subjected to garbage collection, as you would expect with other data structures allocated to the heap.

3 ways to represent integers in Java is decimal (base 10), octal (base 8) and hexadecimal (base 16).

Decimal integers use digits 0-9. No prefix or suffix.

Octal integers use digits 0-7. Prefix 0.

Hexadecimal digits use 0-f. a-f is case insensitive. Prefix 0x. Again case insensitive.

```
class Octal {
public static void main(String [] args) {
             int six = 06; // Equal to decimal 6
             int seven = 07; // Equal to decimal 7
             int eight = 010; // Equal to decimal 8
             int nine = 011; // Equal to decimal 9
             System.out.println("Octal 06 = " + six);
             System.out.println("Octal 07 = " + seven);
             System.out.println("Octal 010 = " + eight);
             System.out.println("Octal 011 = " + nine);
      }
}
```

```
Output:
Octal 06 = 6
Octal 07 = 7
Octal 010 = 8
Octal 011 = 9
```

```
        int a = 0x0001;
        int b = 0x7ffffff;
        int c = 0xDeadCafe;
        System.out.println(a + " " + b + " " + c);
```

Output: `1 134217727 -559035650`

All three integer literals (hexadecimal, octal and decimal) are represented as int, but they can also be specified as long by placing the suffix l or L.

Long jo = 0x1234fL;

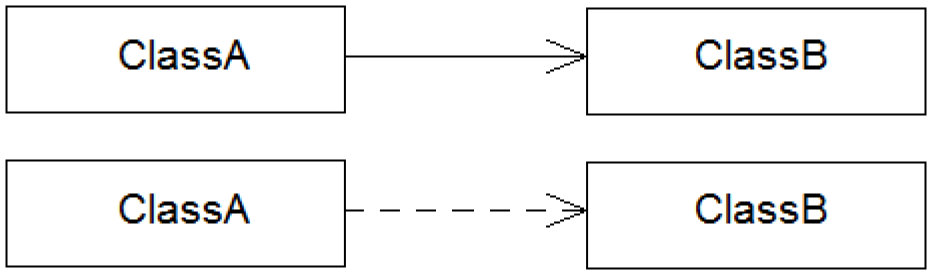
Long so = o12237l;

Floating point literals are defined as double (64 bits) by default. So if you want to assign the floating point literal to a float (32 bits) you must use the suffix F or f. If you don't compiler will complain about a possible loss of precision because you are trying to fit a number into a potentially smaller container (float -32 bits)
Float f = 32.77777 //Compiler error. Possible loss of precision.
Float f = 32.7777f; //No error
Double d = 32.7777D // 'D' is optional.
Double d = 32.7777 //No 'D' suffix but still no error because D is double by default.
Int x = 23,124 //Won't compile because of the comma.

Boolean b = true; //Legal
Boolean b = 0; //Compiler error. Boolean value can only be true to false.
Int x = 1; if(x) {}//Compiler error.

A character literal is represented by the data type char, by a single character in quotes (').
Char a = 'a';
Char b = 'b';
You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u as follows:
Char c = '\u004E'; \\The letter N
Characters are 16 bit unsigned integers under the hood.
That means you can assign a number, integer, such that it will fit into the unsigned 16 bit range (65535 or less).
For example, the following are legal.
Char a = 0x892; //hexadecimal literal
Char b = 982; //integer literal
Char c = (char)70000 // cast is required, 70000 is out of range.

What's the difference between these 2 relationships?



This webpage says enough I think: http://www.classdraw.com/help.htm The following text comes from it but should be enough to understand the difference I think.

So basically the solid line is an association and the dashed/dotted line is a dependency.

Associations can also be unidirectional, where one class knows about the other class and the relationship but the other class does not. Such associations require an open arrowhead to point to the class that is known and only the known class can have a role name and multiplicity. In the example, the Customer class knows about any number of products purchased but the Product class knows nothing about any customer. The multiplicity "0..*" means zero or more.

A dependency is a weak relationship between two classes and is represented by a dotted line. In the example, there is a dependency between Point and LineSegment, because LineSegment's draw() operation uses the Point class. It indicates that LineSegment has to know about Point, even if it has no attributes of that type. This example also illustrates how class diagrams are used to focus in on what is important in the context, as you wouldn't normally want to show such detailed dependencies for all your class operations.

Since my reputation is only 8 I can't place the images itself, but they can still be found on the webpage I menti oned at the start.

[EDIT]

I don't have code examples right here, but how I personally would explain it is as simple as a car and a door.

When a car has a door (or more) it's just a car

Car --- has a --> Door

But when you have a door which can be opened the door class will have a function like

```
public void openDoor(){
this.open();
}
```

To use the function above the car will have to create an instance of the door

```
Class Car(){
Door door1 = new Door();
```

door1.open();
}

In this way you have created a dependency.

So the solid line is just pointing an object(1) to another object(2), but when you start using the object(1) it becomes a dependency.

Reference variable is a sequence of bits which provide a way to reach the object. Reference variable holds the address of the object.

Scope
Static variables have the longest scope. They are created when the class is loaded they stay loaded as long as the class stay s loaded in the Java Virtual Machine.
Instance variables are the next most long lived, they are created when the instance is created and they stay until the instance is removed from memory.
Local variables are next most long lived. They stay alive as long as their method is on the stack. It is possible that they a re alive and still out of scope (If one method calls another, local variables of calling method get out of scope).
Block variables live as long as the code is executing.

Local variables must always be initialized before using, else you will get a compiler error.
Instance variables and array objects need not be initialized, as they get their default values. Default value of all objects is null.

Primitive	Default Value
Int, byte, short, long	0
Float, double	0.0
Char	'\u0000'
Boolean	false

```
public class TimeTravel {
public static void main(String [] args) {
int year; // Local variable (declared but not initialized)
System.out.println("The year is " + year); // Compiler error
}
}
```

Compiling produces output something like this:

```
%javac TimeTravel.java
TimeTravel.java:4: Variable year may not have been initialized.
System.out.println("The year is " + year);
1 error
```

The compiler can't always tell if a variable has been initialized or not. For example, if you initialize variable within a conditional block, compiler knows that the block might not execute and gives you an error.

```
public class TestLocal {
	public static void main(String [] args) {
		int x;
		if (args[0] != null) { // assume you know this will
			// always be true
```

```
                x = 7; // compiler can't tell that this
                // statement will run
            }
            int y = x; // the compiler will choke here
        }
}

//Below code is inside main
Date date = null;
System.out.println(date.toString()); //compiles fine. Eclipse warns - possible null reference.

Date date;
System.out.println(date.toString()); //CE
```

Even local arrays need to be initialized before being used.
Once array is initialized memory is allocated to array elements, they get their default values.

Why is String immutable

1. Synchronization and thread safety: Making String immutable automatically makes it thread safe thereby solving synchronization issues.
2. Security: If strings were mutable, their values could be changed, another piece of code in the application that has reference to that String, would easily change it's value while code that initialized String to a value still expects it's value to be same. This way, database connection parameters, network connection parameters would change before application tries to create a connection. Mutable Strings could cause security issue in Reflection too as paramters there are Strings.
3. Class loading: String object is used for class loading. If Strings were mutable, it would result in wrong class being loaded.
4. Caching: When JVM optimizes your String literals, it sees that if a literal of the same value is already present in the string constant pool, if yes, both references will point to the same String object.
5. Efficiency: Since Strings are immutable, its hashcode will always remain same, so it can be cached, so a HashMap does not need to calculate hashcode() everytime it needs.

That being said, immutability of String only means that you cannot change it using its public API. You can infact bypass normal API using reflection. But it is like making the warranty void.
You can install a SecurityManager to disable reflection, but some libraries like AOP, ORM tools use reflection.

Objects, literals are created after progam runs. Memory allocation for objects happens at runtime. String constant pool exist only at the runtime not at compile time. String constants of a class are created at class load time, not at initialization or first use time.

Because Strings are immutable, String constant pool works. It is not the other way round.

```
class StringTest {
        public static void main(String [] args) {
                String x = "Java"; // Assign a value to x
                String y = x; // Now y and x refer to the same
                // String object
                System.out.println("y string = " + y);
                x = x + " Bean"; // Now modify the object using
                // the x reference
                System.out.println("y string = " + y);
        }
}
```

Output:

```
Y string = Java
Y string = Java
```

HashMap contains an Entry array. Entry is a singly linked list, which contains a key of generic type K, a value of generic ty pe V, int hashcode and Entry reference which refers next node in the list.

```
1. String s = "Fred";
2. String t = s; // Now t and s refer to the same
// String object
3. t.toUpperCase(); // Invoke a String method that changes
// the String

public static void main (String [] args) {
        Dimension d = new Dimension(5,10);
        ReferenceTest rt = new ReferenceTest();
        System.out.println("Before modify() d.height = " + d.height);
        rt.modify(d);
        System.out.println("After modify() d.height = " + d.height);
}
        void modify(Dimension dim) {

        }
 }
```

Output:

```
Before modify() d.height = 10
dim = 11
After modify() d.height = 11
```

Does Java use pass by copy or pass by reference symantics?
Its pass by copy. Pass by copy means creating a copy of that value and passing it. In case of references, Java creates a copy of the bit pattern contained in the reference and passes it.

You can shadow an instance variable by creating a local variable of the same name, either directly or by passing an argument.

```
class Foo {
       static int size = 7;
       static void changeIt(int size) {
              size = size + 200;
              System.out.println("size in changeIt is " + size);
       }
       public static void main (String [] args) {
              Foo f = new Foo();
              System.out.println("size = " + size);
              changeIt(size);
              System.out.println("size after changeIt is " + size);
       }
}
```

Output:

```
size = 7
size in changeIt is 207
size after changeIt is 7
```

int[] key;//legal and recommended
int key [] ;//legal but not recommended due to less readability. Spaces between name and [] legal but bad.
int key[] ;//legal but not recommended due to less readability.

Int[] manager[];//legal. Equivalent to int[][] manager;

JVM does not allocate space unless you create an array object, that's when size matters.
Int[3] keys;//illegal. CE.
Int[] cars = new int[]; //CE. Always give size for JVM to be able to allocate space to hold the objects.

Instance variables of a class are given a value after all super class constructors complete.

Int[] myArray = new int[3][];
An array of objects never holds, objects, instead it holds references to objects.

```
int[] x = new int[5];
x[4] = 2; // OK, the last element is at index 4
x[5] = 3; // Runtime exception. There is no element at index
5!. ArrayIndexOutofBoundsException
int[] z = new int[2];
int y = -3;
z[y] = 4; // Runtime exception. y is a negative number
```

Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
After preceeding 2 lines of code execute, how many objects are there on the heap.
4

Cat[ ][ ] myCats = {{ new Cat("Fluffy"), new Cat("Zeus")},
{new Cat("Bilbo"), new Cat("Legolas"), new Cat("Bert")}}
8 objects

Size must not be specified while creating anonymous arrays. Size is calculated automatically by counting the number of comma separated elements.
New Object[3] {new String("hello"), new String("java")}; //CE.

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
```

Car [] myCars = {new Subaru(), new Car(), new Ferrari()}; //Legal
Objects that pass'Is-A' test for the array type can be assigned to an element of that array.

int[] splats;
int[] dats = new int[4];
char[] letters = new char[5];
splats = dats; // OK, dats refers to an int array
splats = letters; // NOT OK, letters refers to a char array

Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car

int[] blots;

int[][] squeegees = new int[3][];
blots = squeegees; // NOT OK, squeegees is a
// two-d array of int arrays
int[] blocks = new int[6];
blots = blocks; // OK, blocks is an int array

int[][] books = new int[3][];
int[] numbers = new int[6];
int aNumber = 7;
books[0] = aNumber; // NO, expecting an int array not an int
books[0] = numbers; // OK, numbers is an int array

Static initialization blocks execute after the class the loaded. (Only once) Instance initialization blocks execute every tim e a new instance is created, right after all super constructors have completed. If there are more than 1 initialization blocks, they run in the order in which they appear in class file.

```
class Init {
       Init(int x) { System.out.println("1-arg const"); }
       Init() { System.out.println("no-arg const"); }
       static { System.out.println("1st static init"); }
       { System.out.println("1st instance init"); }
       { System.out.println("2nd instance init"); }
       static { System.out.println("2nd static init"); }
       public static void main(String [] args) {
              new Init();
              new Init(7);
       }
}
1st static init
2nd static init
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const
```

There is no way to make the garbage collector run. You can only request garbage collection, but it is not guaranteed. You will hear JVM using concurrent mark sweep algorithm, but there is no guarantee. Specification allows any algorithm to be used. You can make objects eligible for garbage collection in 3 ways:

1. Setting the object reference to null.
2. Reassigning the reference to some other object.
3. Isolating the object.

```
e.g. public class Island {
       private Island island;
       public setIsland(Island island){this.island = island;}
       public void compute(){
              Island i1 = new Island();
              Island i2 = new Island();
              Island i3 = new Island();

              I2.setIsland(i3);
              I1.setIsland(i1);
```

```
                I3.setIsland(i2);

                I1 = null;
                I2 = null;
                I3 = null;

                //do complicated memory intensive stuff
        }
}
```

By the time execution reaches //, 3 island objects have reference variables that refer each other, but their containers have been nulled. So they become eligible for garbage collection.
It is recommended to never use System.gc();
Runtime is a special class that allows us to communicate directly with the JVM. It is a singleton.
You can get its instance by calling Runtime.getRuntime().
Then suggest garbage collector run by calling gc();

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class Object. On the surface this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. The problem is that, as you may have gathered by now, you can't count on the garbage collector to ever delete an object. So, any code that you put into your class's overridden finalize() method is not guaranteed to run. The finalize() method for any given object might run, but you can't count on it, so don't put any essential code into your finalize() method. In fact, we recommend that in general you don't override finalize() at all.
There are a couple of concepts concerning finalize() that you need to remember.

- For any given object, finalize() will be called only once (at most) by the garbage collector.
- Calling finalize() can actually result in saving an object from deletion.

Let's look into these statements a little further. First of all, remember that any code that you can put into a normal method you can put into finalize(). For example, in the finalize() method you could write code that passes a reference to the object in question back to another object, effectively uneligiblizing the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, for this object, finalize() already ran, and it will not run finalize() again.

In every case when an exact match isn't found, JVM uses the method with smallest argument that is wider than the parameter.

```
class EasyOver {
       static void go(int x) { System.out.print("int "); }
       static void go(long x) { System.out.print("long "); }
       static void go(double x) { System.out.print("double "); }
       public static void main(String [] args) {
              byte b = 5;
              short s = 5;
              long l = 5;
              float f = 5.0f;
              go(b);
              go(s);
              go(l);
              go(f);
       }
}
```

```
Int int long double
```
In situations where widening and auto boxing options are available to the compiler, it will choose, widening over auto boxing.
```
class AddBoxing {
       static void go(Integer x) { System.out.println("Integer"); }
       static void go(long x) { System.out.println("long"); }
       public static void main(String [] args) {
              int i = 5;
              go(i); // which go() will be invoked?
       }
}
Long
```
Java 5's designers that most important rule should be pre-existing code should work the way it used to. This keeps the existing code robust. Since widening came later…

```
Class Varargs {
Static void go(int a, int b) { System.out.print("int , int");}
Static void go(byte… b) { System.out.print("byte…");}
Public static void main(String[] args){
       Byte b = 5;
       Varargs.go(5, 5);
}
Int, int
```

1. Widening beats boxing
2. Widening beats var args
3. Boxing beats var args. Var args is a sort of catch all method. Makes sense to use it as a last resort.

```
class BoxOrVararg {
       static void go(Byte x, Byte y)
       { System.out.println("Byte, Byte"); }
       static void go(byte... x) { System.out.println("byte... "); }
       public static void main(String [] args) {
              byte b = 5;
              go(b,b); // which go() will be invoked?
       }
}
Byte, Byte

Widening works for reference variables too
class Animal {static void eat() { } }
       class Dog3 extends Animal {
       public static void main(String[] args) {
              Dog3 d = new Dog3();
              d.go(d); // is this legal ?
       }
       void go(Animal a) { }
}
```
Above code is legal. Because Dog 'IS-A' Animal. **This might not work for wrapper classes. Bytes are not Shorts.**

```
class WidenAndBox {
       static void go(Long x) { System.out.println("Long"); }
       public static void main(String [] args) {
              byte b = 5;
              go(b); // must widen then box - illegal
       }
}
```

```
class BoxAndWiden {
       static void go(Object o) {
       Byte b2 = (Byte) o; // ok - it's a Byte object
       System.out.println(b2);
       }
       public static void main(String [] args) {
              byte b = 5;
              go(b); // can this byte turn into an Object ?
       }
}
```

Strangely enough this code compiles and produces the output:
5.
Compiler can box and widen, but not widen and box.
Why did not the compiler use box and widen when it tried to deal with WidenAndBox class. If it tried to box first then byte would be converted to Byte and ofcourse Byte is not a Long.

```
class Vararg {
       static void wide_vararg(long... x)
       { System.out.println("long..."); }
       static void box_vararg(Integer... x)
       { System.out.println("Integer..."); }
       public static void main(String [] args) {
              int i = 5;
              wide_vararg(i,i); // needs to widen and use var-args
              box_vararg(i,i); // needs to box and use var-args
       }
}
```

This compiles and produces:

```
long...
Integer...
```

1. Primitive widening uses the "smallest" method argument possible.
2. Used individually, boxing and var-args are compatible with overloading.
3. You CANNOT widen from one wrapper type to another. (IS-A fails.)
4. You CANNOT widen and then box. (An int can't become a Long.)
5. You can box and then widen. (An int can become an Object, via Integer.)
6. You can combine var-args with either widening or boxing.

An error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. Applicatons should not declare in its throws clause any subclasses of Error that might be thrown during the execution of the program but not caught, since these Errors are abnormal conditions that should never occur.

All `Throwable`s except subclasses of `java.lang.RuntimeException` or `java.lang.Error` are checked. Properly, in Java, "exceptions" are subclasses of `java.lang.Exception`, "errors" are subclasses of `java.lang.Error` and `java.lang.Throwable` is not usually subclassed directly.

A checked exception is one which must be either handled in a catch clause, or declared as being thrown in the method signature; the compiler enforces this. Generally, one uses checked exceptions for exceptions which should be handled by the calling code, while unchecked exceptions are for conditions which are the result of a programming error and should be fixed by correcting the code.

It is a compile time error if a class is http://docs.oracle.com/javase/specs/jls/se6/html/classes.html#303584 read this later

What is a cookie?
Cookie is a simple text file which is sent by the servlet to the browser, saved by the browser and later sent back to the server.
Cookies identify a user uniquely, so they are used in session management.
Cookie has a name, single value and a set of optional attributes such as path and domain qualifier, maximum age (set in seconds), a comment describing the purpose of the cookie and a version number.
Name of the cookie cannot be changed once created.
Although its value can be changed using the setValue(String) method.
Cookie is created using its constructor public Cookie(String name, String value).
Cookies are sent by the servlet using HttpServletResponse.addCookie(Cookie) method, by adding fields to HTTP Response headers one at a time.
The browser returns cookies to the server by adding fields to the HTTP request headers. Cookies can be retrieved from the request using HttpServletRequest.getCookie() method. It returns the Cookie[]. There can be multiple cookies with same name but different path values.

StringBuffer inherits the equals and hashcode methods from Object class and does not override them.

VisualVM is the profiler that I used. –Xms512M -Xmx1024M BigGridDemo.java

Abstract methods must be overridden by the first concrete subclass.
With respect to overriding:

1. Overriding method must have the same signature as the overridden method.
2. Overriding method must have the same return type as the overridden method. As of Java 5 overriding method can be return a sub class of the super class return type. This is called co-variant return.
3. Overriding method must not have a more restrictive access specifier.
4. Overriding method can use a less restricting access specifier.
5. Overriding method cannot throw broader or more or new checked exceptions.
6. Overriding method can throw narrower or fewer or any unchecked exceptions.

Only inherited methods can be overridden. Private methods cannot be overridden. Final methods cannot be overridden. Trying to do so will throw CE.
Overloading means using the same method name with different arguments. Return type does not play a role in overloading.
Reference type determines which overloaded method will be used at the compile time.
Object type determines which overridden method is used at the runtime.
Polymorphism applied to method overriding and not to method overloading.

When you hit the jsp page directly with url, request.getCookies will return null object.

http://www.pskills.in/servlet/servlet-chaining.jsp

See LinkedHashMap's internal code.

Difference between ArrayList and LinkedList

1. ArrayList needs continuous memory locations, hence elements need to be moved to a bigger space, if new elements are added to a filled array, which is not required for linkedlist.
2. Removal and insertion of elements at a specific position in ArrayList requires trailing elements to be shifted ahead, leading to O(n) complexity, while that for LinkedList is O(1).
3. Random access using index in an ArrayList is faster, O(1) complexity while for LinkedList, entire list needs to be traversed.
4. Though linear search takes limited time for both, binary search using linked list requires creating a new model called binary search tree which is slower but offers constant time for insertion and deletion.
5. For a set of integers you want to sort using quick sort, array list will be faster, for set of large structures you want to sort using selection sort, a linked list will be faster.

What is a class loader? What are the different classloaders used by the JVM?
Class loader is part of the JVM which loads classes and interfaces. Bootstrap, Extension and System are the classloaders used by the JVM. The Bootstrap classloader is a bunch of platform specific of machine instructions that kickoff the whole classloading process. It also loads the code needed to support the basic JRE including classes in java.lang and java.util package. All other classloaders are implemented as Java classes.

What is static import?
By static import we can access static members of a class directly without prefixing it with the class name.
Introduced in Java 5
Import static AnotherConstant.SIZE;
Import static Constants.SIZE; //CE SIZE collides with another import.
Main () { … syso(SIZE); ..}

What are the different segments of memory in Java?
Stack segment – containing local variables
Heap segment – containing objects and instance variables (primitives and objects)
Code segment – containing actual compiled bytecode when loaded

Does gargabe collection guarantee that OutOfMemory will not occur?
No. It is possible that system creates objects faster than they are garbage collected. Also, code might prevent objects from being garbage collected.

Difference between save and persist?
The difference between save and persist is based on the generator class we are using. If the generator class is assigned, then there is no difference because programmer will have to give the primary key value to save in the database.
For any other generator class, say increment, hibernate will assign the primary key value, save and persist will both save the entity in the database normally, but save method will return the primary key id value generated by hibernate and we can see it, persist will never give back the value to the client.
Save is hibernate, persist is part of JPA specification and defined in entityManager.

Difference between merge and update?
Both are used to convert the object from detached state to persistent state. Update will throw exception if the object is already existing in the session. Merge object comes into picture when we try to load the same object again and again from the database.

```
1.          Student current = (Student)session.get(Student.class, 100);
2.          System.out.println("Before merge: " + current.getName());
3.          Student changed = new Student();
4.          changed.setId(100);
5.          changed.setName("Changed new Name");
6.          // session.update(changed); // Throws NonUniqueObjectException
7.          session.merge(changed);
8.          System.out.println("After merge: " + current.getName());
```

Student object is loaded from the database in line 1. New student object is created with same id but different name, at this point if we execute update, we will get NonUniqueObjectException but if we execute merge then name will get changed in the database. Merge occurs when object loaded in the session gets changed.

Thread Pools are useful when you need to limit the number of threads running in your application at the same time. There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack etc.

Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a Blocking Queue which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

Thread pools are often used in multi threaded servers. Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool. The threads in the thread pool will process the requests on the connections concurrently. A later trail will get into detail about implementing multithreaded servers in Java.

Java 5 comes with built in thread pools in the `java.util.concurrent` package, so you don't have to implement your own thread pool. You can read more about it in my text on the java.util.concurrent.ExecutorService. Still it can be useful to know a bit about the implementation of a thread pool anyways.

Here is a simple thread pool implementation. Please note that this implementation uses my own `BlockingQueue` class as explained in my Blocking Queues tutorial. In a real life implementation you would probably use one of Java's built-in blocking queues instead.

```
public class ThreadPool {

    private BlockingQueue taskQueue = null;
    private List<PoolThread> threads = new ArrayList<PoolThread>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new BlockingQueue(maxNoOfTasks);

        for(int i=0; i<noOfThreads; i++){
            threads.add(new PoolThread(taskQueue));
        }
        for(PoolThread thread : threads){
            thread.start();
        }
    }

    public synchronized void  execute(Runnable task) throws Exception{
        if(this.isStopped) throw
            new IllegalStateException("ThreadPool is stopped");

        this.taskQueue.enqueue(task);
    }

    public synchronized void stop(){
        this.isStopped = true;
        for(PoolThread thread : threads){
           thread.doStop();
        }
    }

}
public class PoolThread extends Thread {

    private BlockingQueue taskQueue = null;
    private boolean       isStopped = false;

    public PoolThread(BlockingQueue queue){
        taskQueue = queue;
    }

    public void run(){
        while(!isStopped()){
            try{
                Runnable runnable = (Runnable) taskQueue.dequeue();
                runnable.run();
            } catch(Exception e){
                //log or otherwise report exception,
                //but keep pool thread alive.
            }
        }
    }

    public synchronized void doStop(){
        isStopped = true;
        this.interrupt(); //break pool thread out of dequeue() call.
    }

    public synchronized boolean isStopped(){
        return isStopped;
    }
}
```

The thread pool implementation consists of two parts. A `ThreadPool` class which is the public interface to the thread pool, and a `PoolThread` class which implements the threads that execute the tasks.

To execute a task the method `ThreadPool.execute(Runnable r)` is called with a `Runnable` implementation as parameter. The `Runnable` is enqueued in the blocking queue internally, waiting to be dequeued.

The `Runnable` will be dequeued by an idle `PoolThread` and executed. You can see this in the `PoolThread.run()` method. After execution the `PoolThread` loops and tries to dequeue a task again, until stopped.

To stop the `ThreadPool` the method `ThreadPool.stop()` is called. The stop called is noted internally in the `isStopped` member. Then each thread in the pool is stopped by calling `doStop()` on each thread. Notice how the `execute()` method will throw an `IllegalStateException` if `execute()` is called after `stop()` has been called.

The threads will stop after finishing any task they are currently executing. Notice the `this.interrupt()` call in `PoolThread.doStop()`. This makes sure that a thread blocked in a `wait()` call inside the `taskQueue.dequeue()` call breaks out of the `wait()` call, and leaves the `dequeue()` method call with an `InterruptedException` thrown. This exception is caught in the `PoolThread.run()` method, reported, and then the `isStopped` variable is checked. Since `isStopped` is now true, the `PoolThread.run()` will exit and the thread dies.

Making bean/object singleton does not make it thread safe.
What are annotations?
Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate. Annotations have a number of uses, among them:
• Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
• Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.
• Runtime processing — Some annotations are available to be examined at runtime.

Benefits of immutable objects:

1. Immutable objects are simple: Once the object is created, it will remain in the same state without any extra effort from the programmer. This makes them simple to construct, use and test.
2. Immutable objects are thread safe and can be shared among multiple thread without worrying about synchronization issues.
3. Immutable objects can use static factories that cache frequently used instances to avoid creating new instances when existing ones would do. This reduces memory footprint and garbage collection costs.
4. Since immutable objects can be shared freely, you do not need to provide a copy constructor or a clone method since copies of immutable objects would be equivalent to the original object.
5. They are great building blocks for other objects whether mutable or immutable. It is easier to maintain invariants of a complex object if you know that the underlying object won't change. Thus they make great set elements and map keys. You do not have to worry about their values changing once they are added to the set or map.

Use of finally block and when it does not run?

1. It allows us to wirte clean up code. It is guaranteed to run regardless of exception thrown from try, catch. It allows having return code accidentally bypassed by a return break or continue.
2. If the JVM exits while try or catch is being executed, finally may never run. Likewise if the thread is interrupted or killed while in its try or catch, finally may never run, though the application as a whole continues.

Why are there two Date classes, one in java.util and another in java.sql?

1. Java.util.Date has date and time part while java.sql.Date has date part only. Its complement is java.sql.Time which represents the time part.
2. Java.sql.Date is a subclass of java.util.Date which
   a. Override the toString() method to return Date in the format yyyy-mm-dd.
   b. Adds a static valueOf(String) method which creates a Date object from the String received in format yyyy-mm-dd.
   c. Deprecates the getters and setters for hours, minutes and seconds.

Java.sql.Date is used with JDBC and was intented to not have a time part i.e. hours, minutes, seconds and milliseconds but the class does not enforce this.

When should you override equals and hashcode?
You should always override equals and hashcode, except when

1. Instances of the class are inherently unique – This is true for Thread instances which represent active entities rather than values. The equals implementation provided by Object is perfectly suitable for threads.
2. You do not care whether the class provides a logical equality test - java.util.Random class could have overridden equals method to check if two instances would produces same sequence of random numbers going forwards. But the designers did not think that it clients would need this functionality. Equals implementation provided by Object is perfect...
3. Equals is already defined in the superclass and the superclass definition is appropriate for the subclass. e.g. Set implementations inherit equals definition from AbstractSet and List implementations inherit equals definition from AbstractList and Map from AbstractMap.
4. The class is private or package private and you are certain that its equal method will never be invoked. Eg inner class.

Consider implementing Comparable?
Sorting an array of objects of class implementing Comparable is as simple as Arrays.sort()

Interfaces are slightly slower than abstract classes because JVM has to search the class overriding the method. This does not make a big difference but if you are designing a time critical application then you would not want to leave any stone unturned.

What is the use of marker interfaces?
Marker interfaces are used to provide runtime information about objects. It provides a means to associate metadata with class, where the language does not have explicit support for metadata. A class implements serializable to indicate that its non transient members can be saved to a stream that is wrapped around ObjectOutpuStream.

Out of the box you can use enums to implement singletons, but they cause load time loading, which can be painful to debug, so we are better off with it. Some sites are saying that it is a good thing to use enums for singletons.

Enum constructor can only be private / default.
Enums are not used to hold a fixed set of values. They can contain methods. They can contain constructors. It is not possible to create new enum values at runtime.
Enums cannot implement any interfaces, so they cant be serialized.
Enums are thread safe.
Why main must always be public static void main?
Public: So that interpreter can see it. So that it can be accessible everywhere to every object which may desire to use it for launching the application. Java.exe or javaw.exe can use JNI calls to invoke methods, so they can have it invoked either way irrespective of any access modifier.
Static: No objects exist when main is being called. If it is not static, then you will have to create object to call it.
Void: The value returned by the main method is of no use to the JVM. The only thing that the application would like to communicate to the running process is normal /abnormal termination. This is already possible using System.exit(exitStatus). 0 exit status mean normal termination and non…

When the `intern()` method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.

```
String str = new String("abc");

str.intern();
```
It follows that for any two strings `s` and `t`, `s.intern() == t.intern()` is `true` if and only if `s.equals(t)` is `true`.
Means if s and t both are different string objects and have same character sequence, then calling intern() on both will result in single string pool literal referred by both variables.

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.
Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:
Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 2.1). The next allocation starts at this offset and claims the next portion of the array.
When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

Why we make class abstract: To prevent instantiation. Why? All custom events in Spring must extend ApplicationEvent class. In order for an application to publish an event, it will have to create an instance of that event. In Spring ApplicationEvent class is abstract and generic. It does not make sense to publish a generic event, so ApplicationEvent class is abstract. It also does not include a default constructor, it has a 1 arg constructor accepts Object, forcing the subclass to create a constructor that accepts at least 1 object and passes it to ApplicationEvent using super();

What is marker interface? Name few.
A marker interface is one which does not declare any methods. Serializable and Cloneable.
Runnable has public void run() method, so it is not a marker interface.

Serialization is the process of saving the state of the object / flattening an object. Deserialization is restoring the object.
Transient variables are ignored while serializing the object. While deserializing they get their default values.
ObjectOutputStream.writeObjecy() //serialize and write serialized state to the stream around which ObjectOutputStream is wrapper
ObjectOutputStream.readObject() //read and deserialize
Java's default serialization process takes care of saving the entire object graph of the object. (deep copy). If any object in the object graph does not implement Serializable, then you will get java.io.NotSerializableException.
Sometimes it is not possible to make entire object graph implement Serializable.
You could make Collar serializable by extending Collar class and making subclass serializable, using it in the Dog class, but Collar may be final or subclassing might not be an option due to design constraints or Collar class may itself have some other object which is not Serializable and so on. Transient keyword to the rescue.
To give proper values to the transient variables on deserialization, you can implement methods
```
Private void readObject(ObjectInputStream ois){
        Ois.defaultReadObject(); //handle normal de-serialization
        theCollar = new Collar(Ois.readInt());
}
Private void writeObject(ObjectOutputStream oos){
        Oos.defaultWriteObject(); //handle normal serialization
        Oos.writeInt(theCollar.getCollarSize()); //write extra stuff
```

}
You need to read back the extra stuff in the same order you wrote it, else data would be out of sync.
Most common reason to do this, is you want some part of the object’s state manually.
Not all classes can be serializable because they are runtime specific like threads, streams, runtime, etc. Object class is not serializable for this very reason.
Static variables are never saved as the part of the object, because they do not belong to this object. Serialization is only for objects. Collection interfaces are not serializable but concrete collection classes are. For serialization to work on collections, all objects in that collection must be serializable.
If superclass is serializable then subclass is serializable too.
If superclass is not serializable while subclass is then you won’t get any exception, serialization of subclass would happen and while deserialization superclass variable would get their default values. First non-serializable superclass constructor and all constructors above it will run.

Uses:
Communication: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.
Persistence: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.
Deep Copy: If you need an *exact* replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
Caching: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed.
Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.

10.2 Using Serialization
Serialization is a mechanism built into the core Java libraries for writing a graph of objects into a stream of data. This stream of data can then be programmatically manipulated, and a deep copy of the objects can be made by reversing the process. This reversal is often called deserialization
.
In particular, there are three main uses of serialization:
As a persistence mechanism : If the stream being used is FileOutputStream, then the data will automatically be written to a file.
As a copy mechanism: If the stream being used is ByteArrayOutputStream, then the data will be written to a byte array in memory. This byte array can then be used to create duplicates of the original objects.
As a communication mechanism: If the stream being used comes from a socket, then the data will automatically be sent over the wire to the receiving socket, at which point another program will decide what to do.

How to make a class immutable?
http://www.journaldev.com/129/how-to-create-immutable-class-in-java
1. Declare the class final, so it can’t be extended.
2. Make the fields private so they cannot be accessed from outside the class.
3. Don’t provide setter methods.
4. Initialize all the variables at the time of creating the object performing deep copy.
5. Make all instance variables final to prevent reassignment.
6. Make sure the class methods don’t change the value of fields.

7. From the getter methods, return a copy of the instance variable (if it is an object) rather than returning the object itself.

Is JDK required on each machine to run a program?
JDK is java development kit. It is required for development only. To run a java program you just need a JRE.

Q73. Is it possible to define a method in Java class but provide it's implementation in the code of another language like C?
Ans: Yes, we can do this by use of native methods. In case of native method based development, we define public static methods in our Java class without its implementation and then implementation is done in another language like C separately.

Local variables cannot be static. It gives CE.

Q63. What are the two environment variables that must be set in order to run any Java programs?
Ans: Java programs can be executed in a machine only once following two environment variables have been properly set:
1. PATH variable
2. CLASSPATH variable

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

```
protected Object clone()
                throws CloneNotSupportedException
```

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 `x`, the expression:

```
 x.clone() != x
```

will be true, and that the expression:

```
 x.clone().getClass() == x.getClass()
```

will be `true`, but these are not absolute requirements. While it is typically the case that:

```
 x.clone().equals(x)
```

will be `true`, this is not an absolute requirement.

By convention, the returned object should be obtained by calling `super.clone`. If a class and all of its superclasses (except `Object`) obey this convention, it will be the case that `x.clone().getClass() == x.getClass()`.

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 `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 `super.clone` need to be modified.

It creates a shallow copy.

Constructor cannot be called more than once.
How can we execute code even before the main method is called?
Static blocks are executed when the class is loaded. We can add that code to the class that contains main method.

Try block can be followed by either a finally block or catch block. Any exception thrown from the try block needs to be caught in the catch block or any specific tasks to be performed before code abortion are put in the finally block.

Q20. Does Importing a package imports its sub-packages as well in Java?

Ans: In java, when a package is imported, its sub-packages aren't imported and developer needs to import them separately if required.
For example, if a developer imports a package university.*, all classes in the package named university are loaded but no classes from the sub-package are loaded. To load the classes from its sub-package (say department), developer has to import it explicitly as follows:
Import university.department.*

Q19. What are the performance implications of Interfaces over abstract classes?

Ans: Interfaces are slower in performance as compared to abstract classes as extra indirections are required for interfaces. Another key factor for developers to take into consideration is that any class can extend only one abstract class while a class can implement many interfaces.
Use of interfaces also puts an extra burden on the developers as any time an interface is implemented in a class; developer is forced to implement each and every method of interface.

Difference between Comparable and Comparator?
Comparable is defined in java.lang package which is loaded by default. Comparator is defined in java.util package and is expected to be used a utility.
Comparable defines the natural ordering of the elements and the elements all val ue classes must implement comparable interfaces. String, wrapper classes implement Comparable. Comparator defines additional ordering.
Implementing Comparable will change the current object. While implementing Comparator does not change the current object. The comparator instance is separate from class instance and can be used on demand basis.
Comparable interface has the method public int compareTo(Object object). It returns negative, 0 or positive integer depending on whether this object is smaller than, equal to or greater than the argument. Comparator interface defines method compare(Object o1, Object o2) which returns negative, 0 or positive number depending on whether o1 is smaller than, equal to or greater than o2.
Collection holding instances of any class implementing Comparable can be sorted without providing a Comparator instance using Collections.sort() or Arrays.sort(). **Objects implementing Comparator can be used as TreeSet or TreeMap keys. Same is not applicable to Comparators as comparators separate objects that define sorting logic, they don't modify current obeject.**

Keys in TreeMap and TreeSet must compulsorily implement Comparable interface else you will get java.lang.ClassCastException

Checked Exceptions should be used for expected, but unpreventable errors that are reasonable to recover from. Unchecked Exceptions should be used for everything else.
I'll break this down for you, because most people misunderstand what this means.
Expected but unpreventable: The caller did everything within their power to validate the input parameters, but some condition outside their control has caused the operation to fail. For example, you try reading a file but someone deletes it between the time you check if it exists and the time the read operation begins. By declaring a checked exception, you are telling the caller to anticipate this failure.

Reasonable to recover from: There is no point telling callers to anticipate exceptions that they cannot recover from. If a user attempts to read from an non-existing file, the caller can prompt them for a new filename. On the other hand, if the method fails due to a programming bug (invalid method arguments or buggy method implementation) there is nothing the application can do to fix the problem in mid-execution. The best it can do is log the problem and wait for the developer to fix it at a later time.
Unless the exception you are throwing meets all of the above conditions it should use an Unchecked Exception.
Reevaluate at every level: Sometimes the method catching the checked exception isn't the right place to handle the error. In that case, consider what is reasonable for your own callers. If the exception is expected, unpreventable and reasonable for them to recover from then you should throw a checked exception yourself. If not, you should wrap the exception in an unchecked exception. If you follow this rule you will find yourself converting checked exceptions to unchecked exceptions and vice versa depending on what layer you are in.
For both checked and unchecked exceptions, use the right abstraction level. For example, a code repository with two different implementations (database and filesystem) should avoid exposing implementation-specific details by throwing `SQLException` or `IOException`. Instead, it should wrap the exception in an abstraction that spans all implementations (e.g. `RepositoryException`).

Why FileNotFoundException is CheckedException?

2 down vote accepted

Exceptions always encountered at runtime only, Difference is made when a exception is handled. **Checked or unchecked means whether it is forced to handle at compile time or it will only be identified when it is encountered at runtime.**
If an exception is checked means compiler has way to identify whether the exception can occur or not. and whenever you compile it, you will be forced to handle a checked exception and by doing this chances of runtime exceptions will be reduced.
During file handling, compiler doesn't check whether file is present or not, it just check whether you have handled fileNotFoundException or not, because once you are dealing with a file chances of encountering this exception is very high and you should handle it in your code. for arithmetic Exception there is not way to find it during compile time. and thus it is left unchecked.

Is spring singleton bean thread safe?
No. The two concepts are not even related.
Singletons are about **creation**. This design pattern ensures that only one instance of a class is created.
Thread safety is about **execution**. To quote Wikipedia:
A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.
So eventually thread safety depends on the code and the code only. And this is the reason why Spring beans are not thread safe per se.

CloneTest.java

Null keys and values are not allowed in HashTable and ConcurrentHashMap. HashMap allows 1 null key and any number of

ConcurrentHashMapTest.java

null values.
public class Super

```
{
   int index = 5;
   public void printVal()
   {
      System.out.println(index);
   };
   public static void main(String[] args)
   {
      Super sup = new Sub();
      sup.printVal();
   }
}
class Sub extends Super
{
   int index = 2;
   public void printVal()
   {
      System.out.println(this.index);
   };
} //Output: 2
```

https://dzone.com/articles/java-7-hashmap-vs

MVC architecture
MVC is a software architecture - the structure of the system - that separates domain/application/business (whatever you prefer) logic from the rest of the user interface. It does this by separating the application into three parts: the model, the view, and the controller.
The model manages fundamental behaviors and data of the application. It can respond to requests for information, respond to instructions to change the state of its information, and even to notify observers in event-driven systems when information changes. This could be a database, or any number of data structures or storage systems. In short, it is the data and data-management of the application.
The view effectively provides the user interface element of the application. It'll render data from the model into a form that is suitable for the user interface.
The controller receives user input and makes calls to model objects and the view to perform appropriate actions.

You can reuse CyclicBarrier even if Barrier is broken, but you can not reuse CountDownLatch in Java.

Read more: http://www.java67.com/2012/09/top-10-tricky-java-interview-questions-answers.html#ixzz4oPe2Nn00

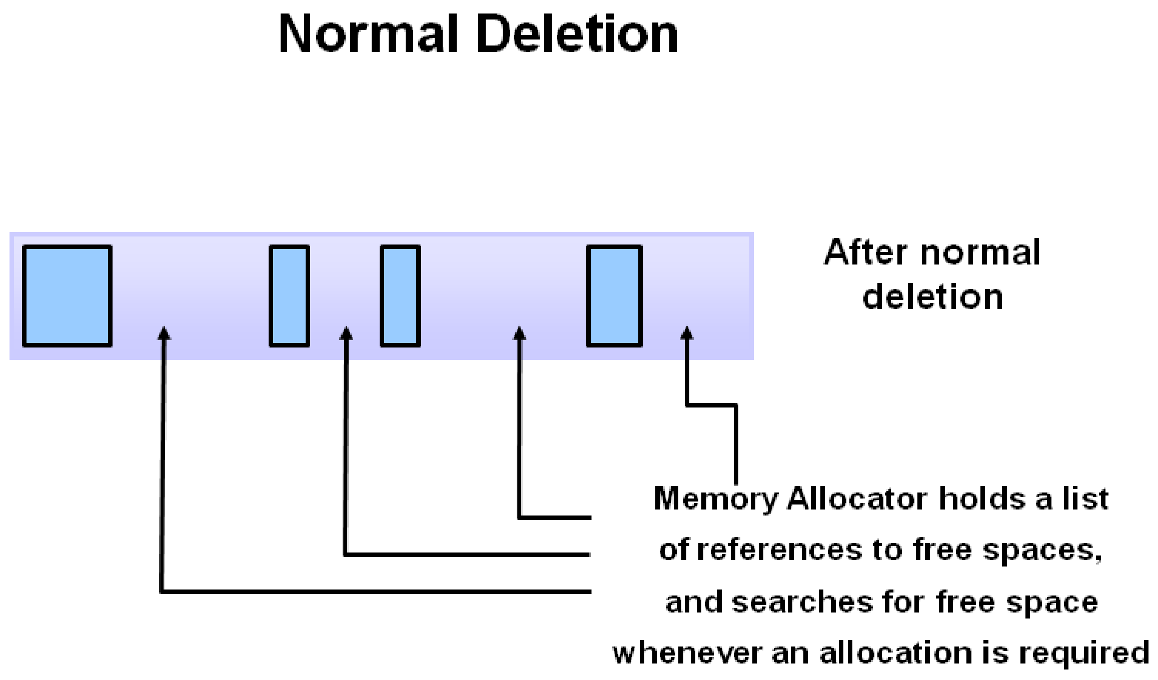
Advantages of locks over synchronized blocks:

1. Locks can be made fair.
2. They can respond to interrupts.
3. They can wait for the lock to become available until a specific time.
4. They make it possible to acquire and release locks in different scopes and in different orders.

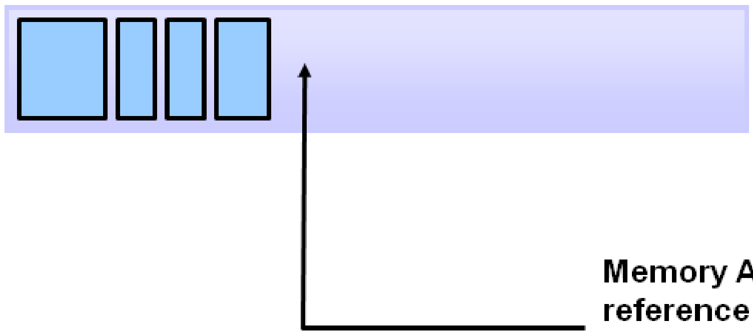
Java development kit

The Java Development Kit (JDK) is a collection of tools for developing Java applications. With JDK you can compile programs and run them in a JVM. JDK also provides tools for packaging and distributing your application.
The JDK and JRE share the application programming interface (API). It makes development easier by providing a set of precompiled libraries for performing many tasks such as dates, calendar, collections, networking, I/O.
JRE is an abstract computing machine.

Automatic garbage collection is the process of looking up an object in the heap memory and identifying which objects are in use and which are not, deleting unused objects. An object in use is one which is reachable by the program, while an unusable is object is an unreachable object.

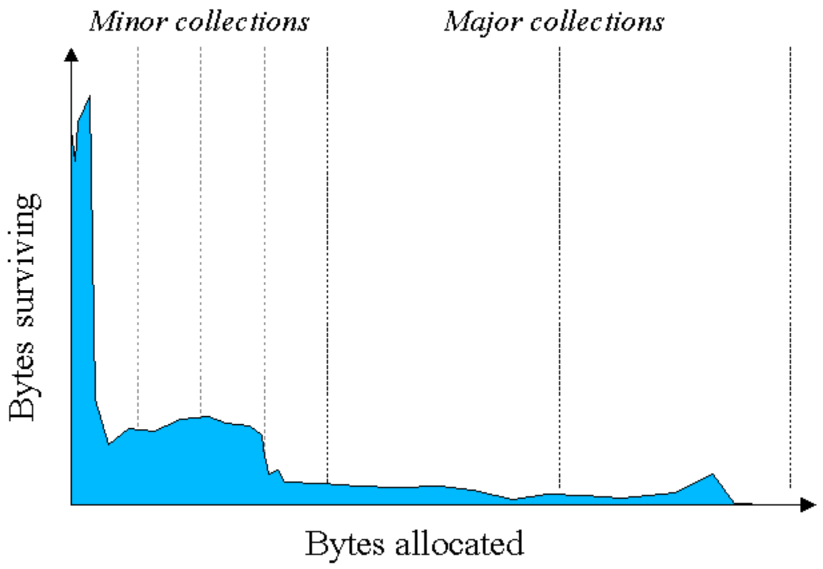

Normal Deletion
After normal deletion
Memory Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required

Deletion with **Compacting**



After normal Deletion with compacting

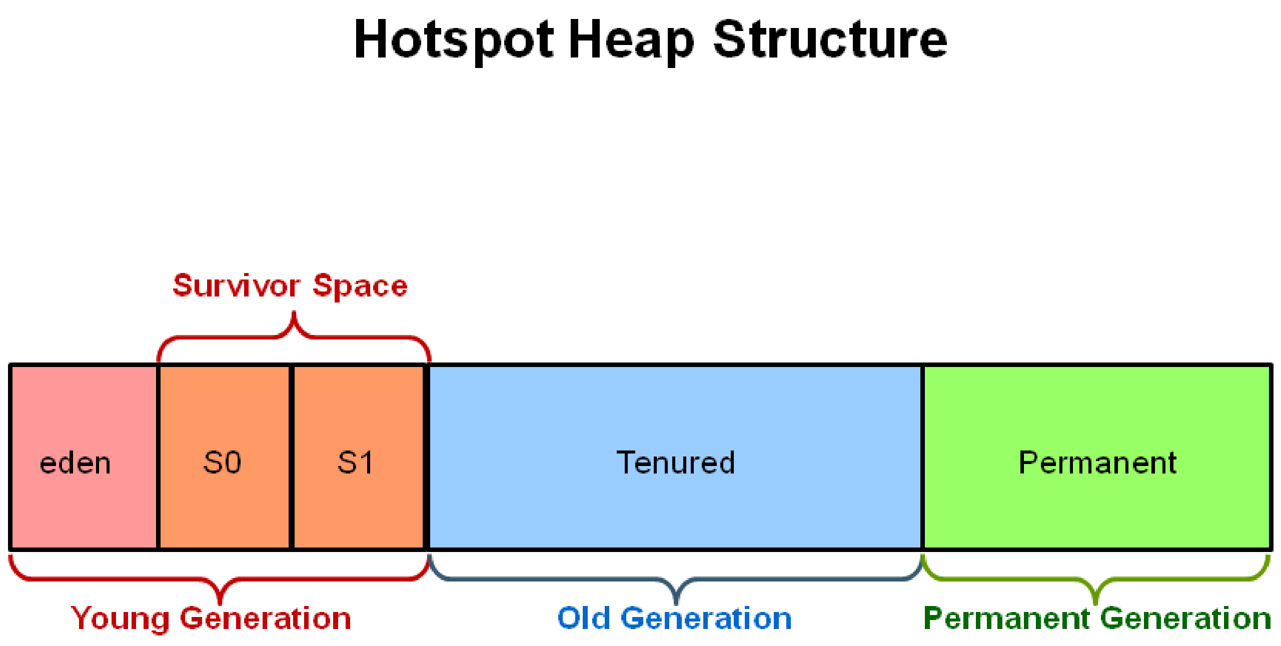
Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.



First step in the process of garbage collection is marking. This is where the garbage collector identifies, which pieces of objects are in use and which are not. This can be a time consuming process, if all the objects in the system must be processed.
Normal deletion process removes unreferenced objects leaving referenced objects and pointers to free space. Memory allocator holds references to free space and searches the free space whenever an allocation is required.
In addition to deletion you can also compact the memory occupied by referenced objects, this makes memory allocation of new objects fast and easy.

Why generational garbage collection?
Having to mark and compact all objects in the heap is inefficient, as the application keeps running, the number of objects increases and it takes longer and longer time for garbage collection. Canonical analysis shows that most objects are short lived, so it makes sense to divide them in the generations based on how long they have been living in the heap and performing garbage collection on only a generation.

Generations in heap:
`



The young generation is where the new objects are allocated and aged. When the young generation fills up, a minor garbage collection occurs. Objects surviving the minor garbage collection are moved to the old generation. Minor garbage is a Stop The World process, this means that JVM stops processing the application while minor garbage collection is in progress. Minor collections can be optimized assuming a high mortality rate. A young generation full of dead objects is garbage collected very quickly.
Objects living in the young generation for a specific time are moved to the old generation.
Old generation is used for storing the long lived objects. Eventually the old generation needs to be garbage collected. This is called a major garbage collection. Even this is a Stop the world event. It is slower because it often involves the live objects. So for responsive application, major garbage collection should be minimized. The time taken for major garbage collection depends on the garbage collector being used.
The Permanent generation contains the metadata required by the JVM to describe the classes and methods used in the application. It is populated at the runtime based on the classes and methods used by the application. It also stores the JVM standard libraries.
Garbage collectors

1. Serial GC

Default for client style machines in Java SE 5 and 6. It uses mark compact collection method. This method moves the referenced objects to the start of the heap, so that new objects are allocated towards the end. This makes it fast to allocate memory quickly. It uses a simgle virtual processor for garbage collection, hence the name.
Use cases:
Applications that do not have a low pause time requirements.
Machine that runs several JVMs.
To enable serial collector use:
-XX:+UseSerialGC
To enable the Serial Collector use:
```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar
c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

2. The Parallel GC
   It uses several threads to perform garbage collection on young generation. It comes in two flavours, one that uses multiple threads on young generation and other that uses multiple threads on both young and old generations.
   ```
   -XX:+UseParallelGC
   -XX:+UseParallelOldGC
   -XX:ParallelGCThreads=<desired number>
   ```
   Usage:
   Applications that do not have a low pause time requirements and where lots of work needs to be done.
3. Concurrent mark sweep
   The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.
   Note: CMS collector on young generation uses the same algorithm as that of the parallel collector.
   To enable the CMS Collector use:
   `-XX:+UseConcMarkSweepGC`
   and to set the number of threads use:
   `-XX:ParallelCMSThreads=<n>`

What is an exception?
Exception is an abnormal condition that occurs during program execution and disrupts the normal flow of the program.

What is exception chaining.?
When an exception is thrown in response to another exception given exception. This gives us complete history of a problem
```
eg. get(int index){
Iterator it = iterator();
while(it.next()){
..
} catch(NoSuchElementException){
throw new ArrayIndexOutOfBoundsException(index);
}
```

Is there a way of throwing a checked exception from a method that does not have throws clause?
Yes we can use generics. We can take advantage of the type erasure performed by the compiler and make it think we are throwing an unchecked exception, when, in fact; we're throwing a checked exception:

```
public <T extends Throwable> T sneakyThrow(Throwable ex) throws T {
   throw (T) ex;
}

public void methodWithoutThrows() {
   this.<RuntimeException>sneakyThrow(new Exception("Checked Exception"));
}
```

Q10. Why would you want to subclass an exception?
If the exception type isn't represented by those that already exist in the Java platform, or if you need to provide more information to client code to treat it in a more precise manner, then you should create a custom exception.
Deciding whether a custom exception should be checked or unchecked depends entirely on the business case. However, as a rule of thumb; if the code using your exception can be expected to recover from it, then create a checked exception otherwise make it unchecked.
Also, you should inherit from the most specific Exception subclass that closely relates to the one you want to throw. If there is no such class, then choose Exception as the parent.

Queue
Difference between pool and remove? Poll retrieves and removes the head of the queue,returns null if the queue is empty. Remove Retrieves and removes the head of the queue and throws **NoSuchElementException** when the queue is empty.
https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html
Add and offer removes both return boolean. Remaning method return the element processed.
Addtion happens at the tail. Removal happens at the head.

PriorityQueue
PriorityQueue **allows** you to **insert duplicates**.
You can construct a priorty queue using another Collection, PriorityQueue, SortedSet or an empty priority queue. When you create an empty priority queue you can give an initial capacity, or use default initial capacity of 11, or specify both initi al capacity and Comparator.
Elements of priority queue are ordered using **natural ordering** of its elements or specified comapator.
When not using a Comparator, **it allows you to insert 1 non comparable element, but gives you a ClassCastException when you insert second element**. If you specify a Comparator, you can insert non-comparable elements.
Not synchronized. If being accessed by multiple threads use PriorityBlockingQueue instead.
Enqueing and dequeing methods(add, remove, offer, poll) provide O(nlogn) time. Linear time fo r remove(Object) and contains(Object). Constant time for retrieval methods peek, element and size.
It is backed by an array.
Keeps the smallest element as per Comparator at the head.
Iterator is not guaranteed to iterate the elements in a particualt order, if you need ordered traversal consider using Arrays.sort(pq.toArray()); It does not throw ConcurrentModificationException either.

What is the difference between fail fast and fail safe iterators?
Fail fast iterator throws ConcurrentModificationException if the collection is modified structurally after the iterator is created in any way, other than iterator's own remove method. Iterator fails quickly and cleanly rather than risking non deterministic behaviour at an undetermined time in future. Fail safe iterators will not throw ConcurrentModificationException because they operate on the copy of underlying data structure, so concurrent modification is not possible.

Note: Fail fast behavior of iterators cannot be guaranteed as it is normally difficult to make any hard gaurantees in the event of concurrent modification by multiple threads. So fail fast behavior should only be used to detect bugs and program should not depend on fail fast behavior for its correctness.

How do you remove an entry from a Collection? and subsequently what is the difference between the remove() method of Collection and remove() method of Iterator, which one you will use while removing elements during iteration?
Collection interface provides a Boolean remove(Object o) method which wil l remove the specified object from the collection and return a Boolean value indicating if the underling collection was modified as a result of this call. Cannot be used while iteration is in progress, as iterator will throw Connc… if it is a fail fast one .
Remove method of the iterator alliows you to remove the last element returned by the iterator, so it can be used while iteration is in progtess.

What is the difference between synchronized collections and concurrent collections?
Concurrent collections offer much better concurrency / performance because they lock only a portion of the underlying data structure unlike synchronized collections that lock the entire data structure.
In ConcHashMap better concurrency is obtained by using a ReadWriteReentrantLock which allows multiple threads to read the map at a time but only one thread to write at a time. It locks only a particular bucket rather entire map. Threads are free to read /write other buckets.
Concurrent collections:

1. ConcurrentHashMap
2. ConcurrentSkipListMap
3. ConcurrentSkipListSet
4. CopyOnWriteArraySet
5. CopyOnWriteArrayList

Synchronized collections:

1. Collections.synchronizedCollection(Collection<T> c)
2. Collections.synchronizedList(List<T> l)
3. Collections.synchronizedSet(Set<T> s)
4. Collections.synchronizedSortedSet(SortedSet<T> s)
5. Collections.synchronizedMap(Map<K,V> m)
6. Collections.synchronizedSortedMap(SortedMap<K,V> m)

Unmofiable collections can be obtained using:

1. Collections.umodifiableCollection(Collection<? Extends T> c)
2. Collections. umodifiableList(List<? Extends T> l)
3. Collections. umodifiableSet(Set<? Extends T> s)
4. Collections. umodifiableSortedSet(SortedSet< T> s)
5. Collections. umodifiableMap(Map<? Extends K, ? Extends V> m)
6. Collections. umodifiableSortedMap(SortedMap<? Extends K, ? Extends V> m)

What is the difference between Enumerator and Iterator?
Enumeration has only 2 methods hasMoreElements() and nextElement(). Iterator has additional methods that allow us to set, remove elements being returned. Iterator methods has shortened names.
An object that implements the Enumerator interface generates a series of elements, one at a time. Successive calls to nextElement() returns the successive element. Its methods allows us to iterate over elements of Vector, keys of hashtable, values of hashtable and to specify inpu streams to SequenceInputStreams.
hasMoreElements() – Returns true if the enumeration has at least 1 more elment and false otherwise.

nextElement() – Returns the next element if the enumeration has at least 1 more element else throws NoSuchElementException.
Iterator allows us to iterate over collections. It has three methods hasNext(), next() and remove().
hasNext() – returns true if the iteration has at least 1 more element, false otherwise.
Next() – returns the next element of the iteration if it has at least 1 more element, else throws NoSuchElementException.
Remove() – Removes from the underlying collection, the last element returned by the iterator, which is an optional operation. It must be called once per call to next else it will throw an IllegalStateException. The behavior of the iterator is unspecified if the underlying collection is modified while iteration is in progress in any way other than this method.

What is difference between RESTful and RESTless?
REST stands of Representational State Transfer. When you have a bunch of uniquely addressable "entities" that you want to make available via Web Application then you can expose REST based web service. REST defines a bunch of stuff about what GET, POST, PUT, DELETE means. The HTTP URL will uniquely identify the entity using either a request parameters or path parameters and HTTP verb (PUT, POST) will identify the action to be perfomed on them. The general semantics for a REST based web service include:

1. PUT – create new entity
2. POST – modify an entity.
3. DELETE – delete existing entity.
4. GET – query an entity.

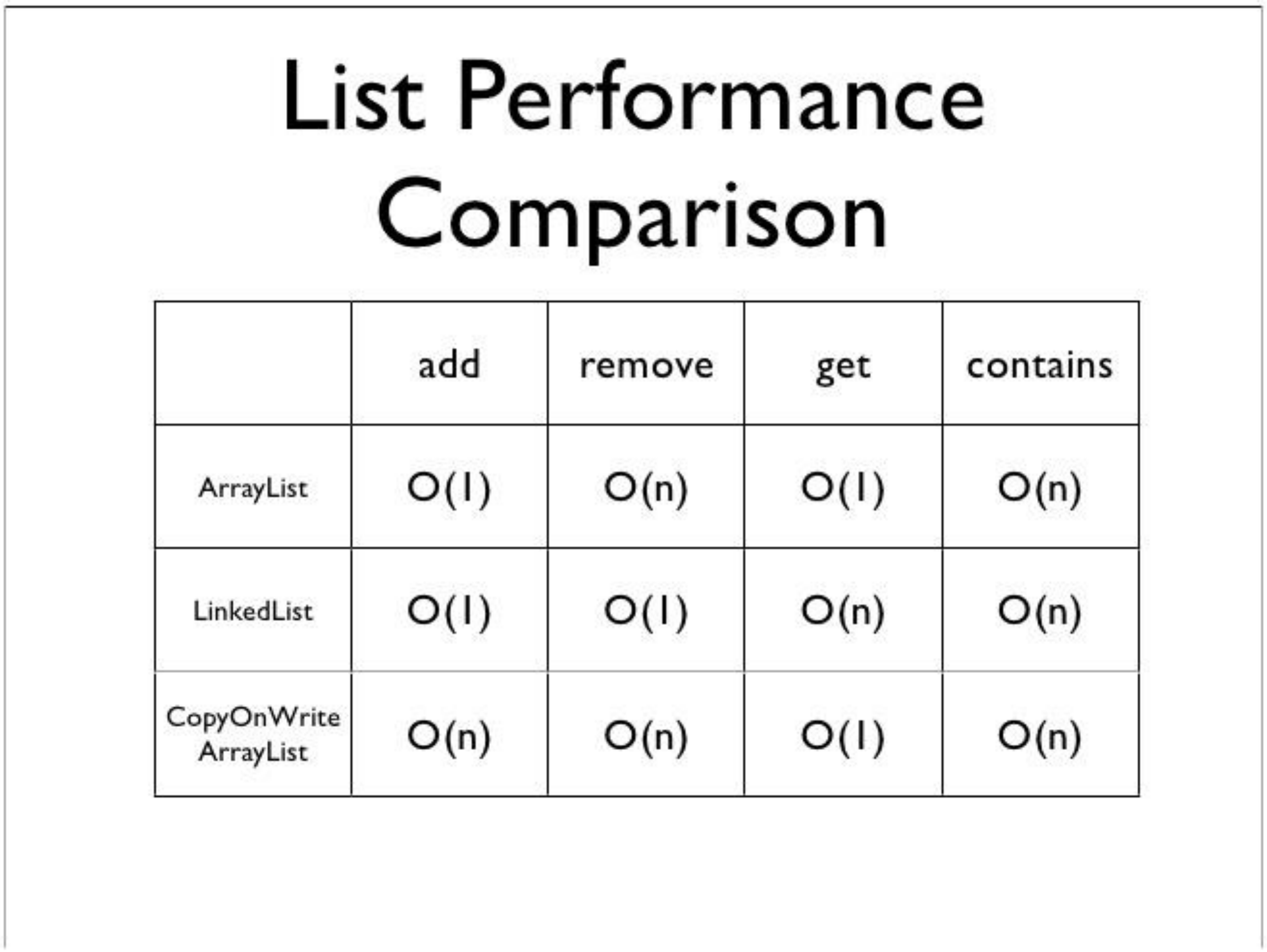
Anything that does not follow the RESTful principles in RESTless.
https://stackoverflow.com/questions/630453/put-vs-post-in-rest?rq=1

Study your OAuth flow.
https://stormpath.com/blog/token-auth-for-java
https://github.com/curityio/example-java-oauth-protected-api/blob/master/oauth-filter/src/main/java/se/curity/examples/oauth/OAuthFilter.java

Filter interview questions with example. Session management, difference between Res.redirect and RD.forward.
Diff between ServletConfig and ServletContext. Core tag library. Session management. Servlet lifecycle. JSP lifecycle. Implicit objects.

InheritanceTest.java

15) What is NavigableMap in Java? What is a benefit over Map?
NavigableMap Map was added in Java 1.6, it adds navigation capability to Map data structure. It provides methods like lowerKey() to get keys which is less than specified key, floorKey() to return keys which is less than or equal to specified key, ceilingKey() to get keys which is greater than or equal to specified key and higherKey() to return keys which is greater specified key from a Map. It also provide similar methods to get entries e.g. lowerEntry(), floorEntry( ), ceilingEntry() and higherEntry(). Apart from navigation methods, it also provides utilities to create sub-Map e.g. creating a Map from entries of an exsiting Map like tailMap, headMap and subMap. headMap() method returns a NavigableMap whose keys are less than specified, tailMap() returns a NavigableMap whose keys are greater than the specified and subMap() gives a NavigableMap between a range, specified by toKey to fromKey.



List Performance Comparison

	add	remove	get	contains
ArrayList	O(1)	O(n)	O(1)	O(n)
LinkedList	O(1)	O(1)	O(n)	O(n)
CopyOnWrite ArrayList	O(n)	O(n)	O(1)	O(n)

```
public class Super {

   public static void main(String[] args) {

      Super s = new Super();
      Sub3 sub3 = (Sub3)s;
      sub3.doo();
   }

   public void foo() {}
}

class Sub3 extends Super {

   public void doo() {
    }
}
Exception in thread "main" java.lang.ClassCastException: test.Super cannot be cast to test.Sub3
	at test.Super.main(Super.java:8)
```

Reference Variable Casting (Objective 5.2)

❑ There are two types of reference variable casting: downcasting and upcasting.

❑ Downcasting: If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.

❑ Upcasting: You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

Dynamic binding means the JVM will decide at runtime which method implementation to invoke based on the class of the object. Polymorphism means you can use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses.