*OCP Java SE 7 Programmer II*

*7. Assertions and Java 7 Exceptions*

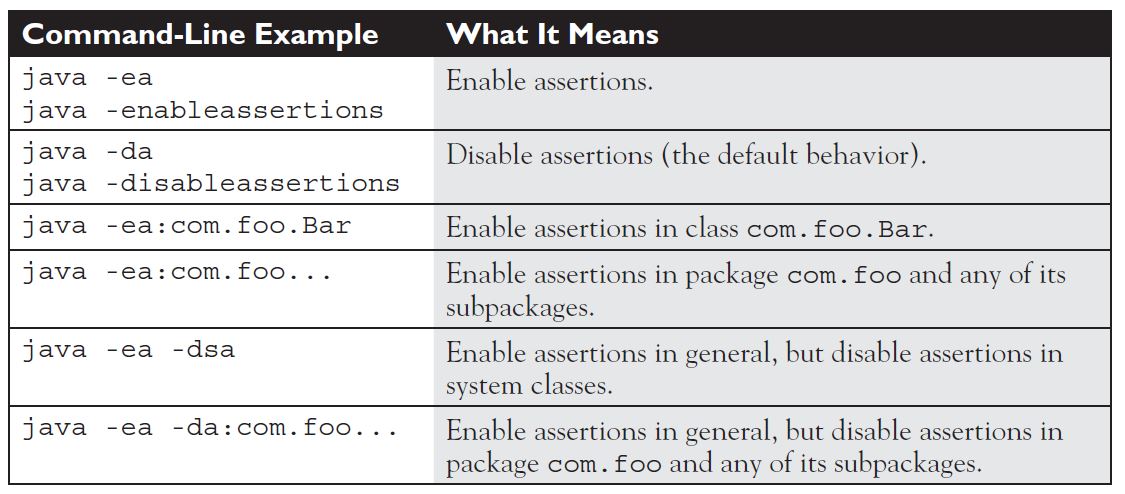
***Assertions:***

Prior to java 1.4 assert is not a keyword in java, so we can use as identifier also.

int assert = getInitialValue();//**compiles.**

From java 1.4 onwards assert is keyword, so we can’t use as identifier.

int assert = getInitialValue();//**not compiles.**



* Don’t Use Assertions to Validate Arguments to a public Method.
* Don't Use Assertions to Validate Command-Line Arguments.
* Do Use Assertions to Validate Arguments to a private Method.
* Do Use Assertions, Even in public Methods, to Check for Cases That You Know Are Never, Ever Supposed to Happen.

switch(x) {

case 1: y = 3; break;

case 2: y = 9; break;

case 3: y = 27; break;

default: assert false; // we're never supposed to get here!

}

* Don't Use assert Expressions That Can Cause Side Effects!

The following would be a very bad idea:

public void doStuff() {

assert (modifyThings());

// continues on

}

public boolean modifyThings() {

y = x++;

return true;

}

assert expressions aren't guaranteed to always run, so you don't want your code to behave differently depending on whether assertions are enabled.

***Working with Java 7 Exception Handling***

You can't use the variable name multiple times in a multi-catch. The following won't compile:

catch(Exception1 e1 | Exception2 e2)3

With multi-catch, order doesn’t matter. The following two snippets are equivalent to each other:

catch(SQLException | IOException e) // these two statements are equivalent

catch(IOException | SQLException e)

With multi-catch, you have to make sure a given exception can only match one type. The following will not compile:

catch(FileNotFoundException | IOException e)

catch(IOException | FileNotFoundException e)

You'll get a compiler error that looks something like:

The exception FileNotFoundException is already caught by the alternative IOException

Remember, multi-catch is only for exceptions in different inheritance hierarchies. To make sure this is clear, what do you think happens with the following code:

catch(IOException | Exception e)

That’s right. It won’t compile because IOException is a subclass of Exception. Which means it is redundant and the compiler won’t accept it.

catch (SQLException | IOException e) {

e = new IOException();// **won’t compile.**

}

Since multi-catch uses multiple types, there isn't a clearly defined type for the variable that you can set. Java solves this by making the catch parameter final when that happens.

catch (SQLException | IOException e) {

log(e);

throw e;

}

Lucky for us, Java 7 helps us out here as well with a new feature.

catch (Exception e) {

log(e);

throw e;

}

In Java 7, } catch (Exception e) { doesn't really catch ANY Exception subclass. The code may say that, but the compiler is translating for you. The compiler says, "Well, I know it can't be just any exception because the throws clause won't let me. I'll pretend the developer meant to only catch SQLException and IOException. After all, if any others show up, I'll just fail compilation on throw e; just like I used to in Java 6." Tricky, isn't it?

*8. String Processing, Data Formatting, Resource Bundles*

Date d1 = new Date(1\_000\_000\_000\_000L);

System.out.println("1st date " + d1.toString());

Calendar c = Calendar.getInstance();

c.setTime(d1); // #1

if(Calendar.SUNDAY == c.getFirstDayOfWeek()) // #2

System.out.println("Sunday is the first day of the week");

System.out.println("trillionth milli day of week is " + c.get(Calendar.DAY\_OF\_WEEK)); // #3

c.add(Calendar.MONTH, 1); // #4

Date d2 = c.getTime(); // #5

System.out.println("new date " + d2.toString() );

**Output:**

1st date Sat Sep 08 19:46:40 MDT 2001

Sunday is the first day of the week

trillionth milli day of week is 7

new date Mon Oct 08 19:46:40 MDT 2001

The other Calendar method you should know for the exam is the roll() method. The roll() method acts like the add() method, except that when a part of a Date gets incremented or decremented, larger parts of the Date will not get incremented or decremented. Hmmm… for instance:

// assume c is October 8, 2001

c.roll(Calendar.MONTH, 9); // notice the year in the output

Date d4 = c.getTime();

System.out.println("new date " + d4.toString() );

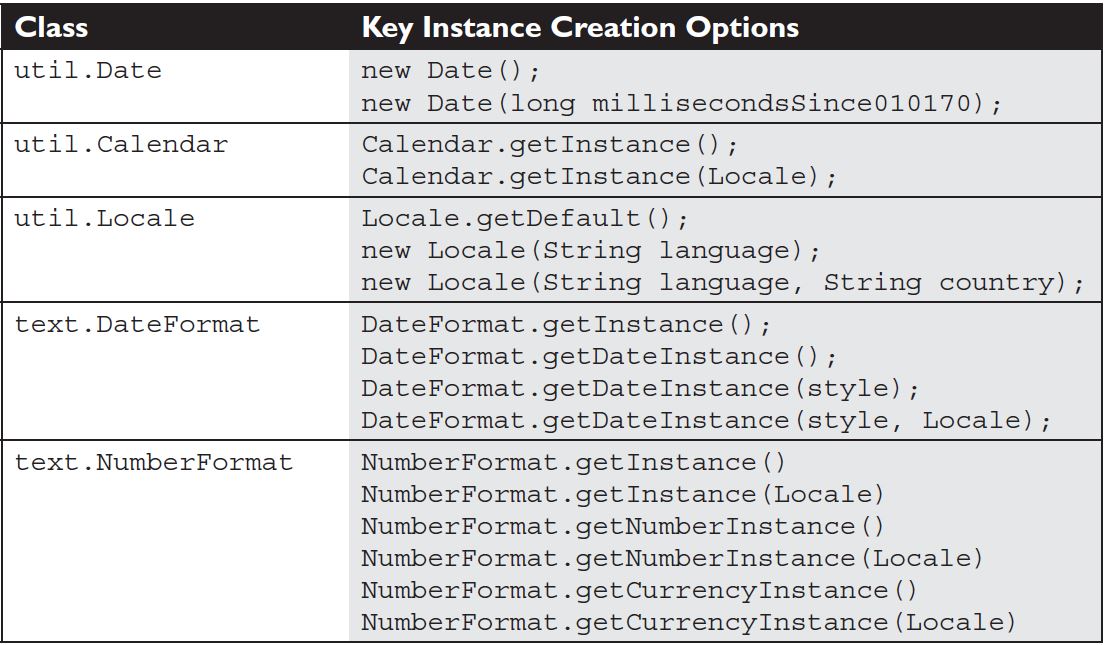
**The output would be something like this:**

new date Fri Jul 08 19:46:40 MDT 2001

Notice that the year did not change, even though we added nine months to an October date. In a similar fashion, invoking roll() with HOUR won't change the date, the month, or the year.

DateFormat.format() to convert Date into a String.

DateFormat.parse() to convert String into a Date.



%[arg\_index$][flags][width][.precision]conversion char

* **arg\_index** An integer followed directly by a $, this indicates which argument should be printed in this position.
* **flags** While many flags are available, for the exam, you'll need to know:

- Left-justify this argument

+ Include a sign (+ or -) with this argument

0 Pad this argument with zeroes

, Use locale-specific grouping separators (i.e., the comma in 123,456)

( Enclose negative numbers in parentheses

* **width** This value indicates the minimum number of characters to print. (If you want nice, even columns, you'll use this value extensively.)
* **precision** For the exam, you'll only need this when formatting a floating point number, and in the case of floating-point numbers, precision indicates the number of digits to print after the decimal point.

conversion The type of argument you'll be formatting. You'll need to know:

b boolean

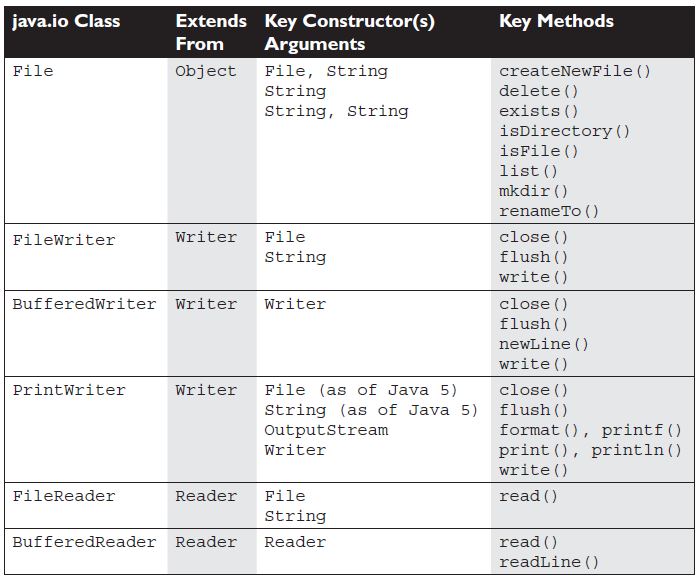
c char

d integer

f floating point

s string

* The Calendar methods you should understand are add(), which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and roll(), which works like add() but doesn't increment a date's bigger pieces. (For example, adding ten months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
* DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
* The DateFormat.format() method is used to create strings containing properly formatted dates.
* Remember that metacharacters and strings don't mix well unless you remember to "escape" them properly. For instance, String s = "\\d";.



1. Invoke the createNewFile() method on a File object. For example:

File file = new File("foo"); // no file yet

file.createNewFile();

1. Create a Writer or a Stream. Specifically, create a FileWriter, a PrintWriter, or a FileOutputStream. Whenever you create an instance of one of these classes, you automatically create a file.

The readPassword() method of Console class doesn't return a string; it returns a character array. Here's the reason for this: Once you've got the password, you can verify it and then absolutely remove it from memory. If a string was returned, it could exist in a pool somewhere in memory, and perhaps some nefarious hacker could find it.

Path p6 = Paths.*get*("tmp", "file1.txt");

If the program is run from the root, it is the one in /tmp/file1.txt. If the program is run from /tmp, it is the one in /tmp/tmp/file1.txt.

Path path1 = Paths.get("/java/source");

Path path2 = Paths.get("/java/source/directory");

Path file = Paths.get("/java/source/directory/Program.java");

Files.createDirectory(path1); // create first level of directory

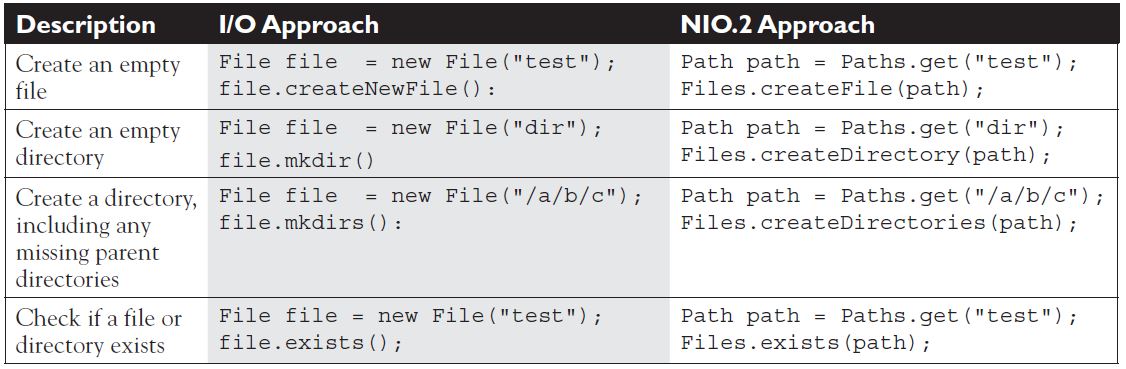
Files.createDirectory(path2); // create second level of directory

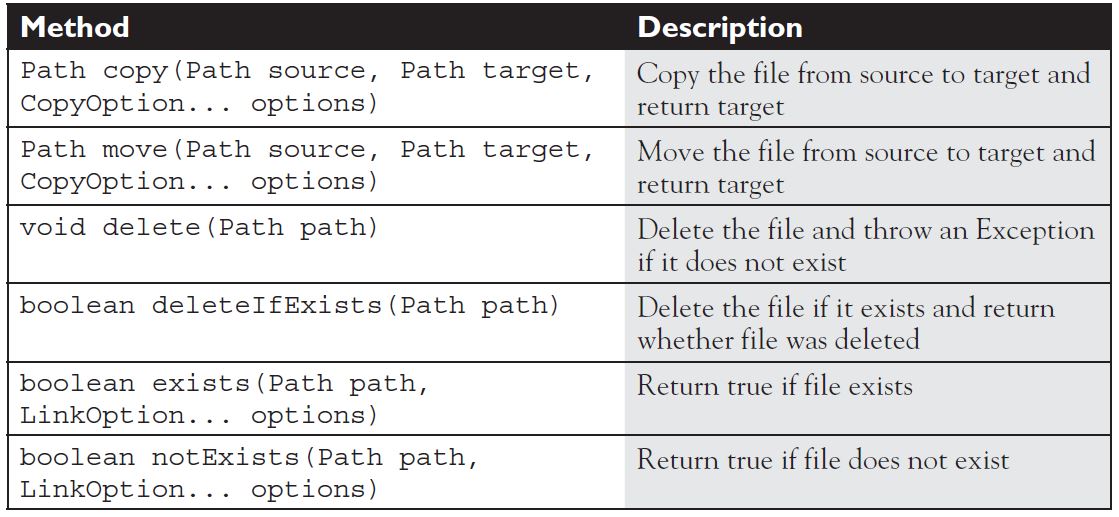
Files.createFile(file); // create file

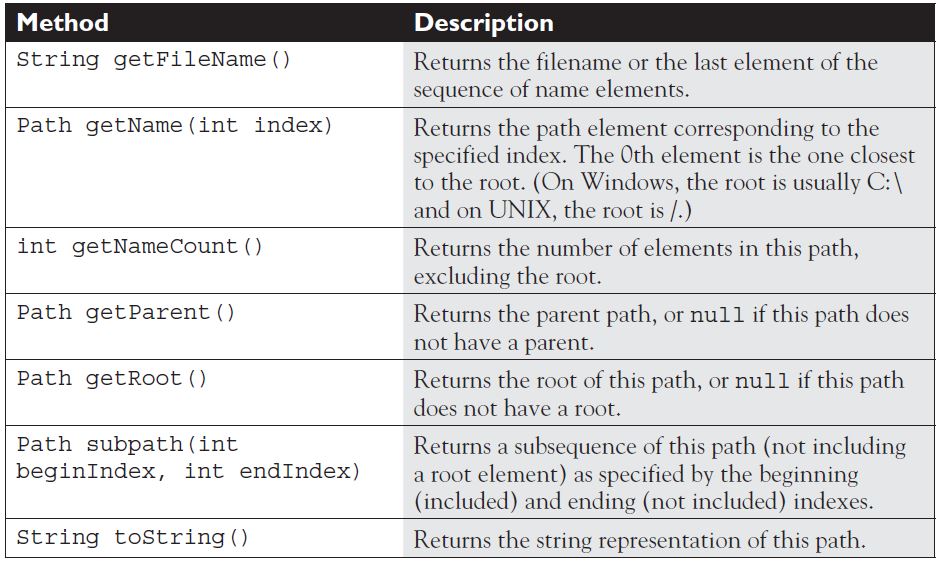
Or we could create all the directories in one go:

Files.createDirectories(path2); // create all levels of directories

Files.createFile(file); // create file







In the world of globs, one asterisk means "match any character except for a directory boundary." Two asterisks means "match any character, including a directory boundary."

Path path = Paths.get("/com/java/One.java");

matches(path, "glob:\*.java"); // false

matches(path, "glob:\*\*/\*.java"); // true

matches(path, "glob:\*"); // false

matches(path, "glob:\*\*"); // true

Path path1 = Paths.get("One.java");

Path path2 = Paths.get("One.ja^a");

matches(path1, "glob:\*.????"); // true

matches(path1, "glob:\*.???"); // false

matches(path2, "glob:\*.????"); // true

matches(path2, "glob:\*.???"); // false

Globs also provide a nice way to match multiple patterns. Suppose we want to match anything that begins with the names Kathy or Bert:

Path path1 = Paths.get("Bert-book");

Path path2 = Paths.get("Kathy-horse");

matches(path1, "glob:{Bert\*,Kathy\*}"); // true

matches(path2, "glob:{Bert,Kathy}\*"); // true

matches(path1, "glob:{Bert,Kathy}"); // false

The first glob shows we can put wildcards inside braces to have multiple glob expressions. The second glob shows that we can put common wildcards outside the braces to share them. The third glob shows that without the wildcard, we will only match the literal strings "Bert" and "Kathy."

The basic flow of WatchService stays the same, regardless of what you want to do:

1. Create a new WatchService

2. Register it on a Path listening to one or more event types

3. Loop until you are no longer interested in these events

4. Get a WatchKey from the WatchService

5. Call key.pollEvents and do something with the events

6. Call key.reset to look for more events

**Serialization (OCP 7 Objective 7.2)**

* The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
* A class must implement Serializable before its objects can be serialized.
* The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.
* If you mark an instance variable transient, it will not be serialized even though the rest of the object’s state will be.
* You can supplement a class’s automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding calls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.
* If a superclass implements Serializable, then its subclasses do automatically.
* If a superclass doesn’t implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its super constructor(s).
* DataInputStream and DataOutputStream aren’t actually on the exam, in spite of what the Oracle objectives say.
* It’s okay for a class to implement Serializable even if its superclass doesn’t. However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don’t run on deserialized classes that implement Serializable.
* PathMatcher and WatchService use FileSystem-specific implementations.
* Know what the following expressions mean for globs: \*, \*\*, ?, and {a,b}.
* You can change the flow of a file visitor by returning one of the FileVisitResult constants: CONTINUE, SKIP\_SUBTREE, SKIP\_SIBLINGS, or TERMINATE.
* An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked private.
* To instantiate an inner class, you must have a reference to an instance of the outer class.

MyOuter.MyInner inner = new MyOuter().new MyInner();

* A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked final.
* instantiating a static nested class requires using both the outer and nested class names as follows:

BigOuter.Nested n = new BigOuter.Nested();

* The only modifiers you can apply to a method-local inner class are abstract and final. (Never both at the same time, though.)
* We should create static inner class object only in static methods of outer class.
* We should not call non static inner class from static method of outer class.
* Don’t use the transient variables while writing hashcode() method, because in deserialization default values will be constructed.

Here's what could happen using code like the preceding example:

1. Give an object some state (assign values to its instance variables).
2. Put the object in a HashMap, using the object as a key.
3. Save the object to a file using serialization without altering any of its state.
4. Retrieve the object from the file through deserialization.
5. Use the deserialized (brought back to life on the heap) object to get the object out of the HashMap.

* Don’t use Arrays.sort(arrayToSort, Comparator) to sort primitive array.

String [] sa = {"one", "two", "three", "four"};

Arrays.sort(sa);

for(String s : sa)

System.out.print(s + " ");

System.out.println("\none = " + Arrays.binarySearch(sa,"one"));

System.out.println("now reverse sort");

ReSortComparator rs = new ReSortComparator();

Arrays.sort(sa,rs);

for(String s : sa)

System.out.print(s + " ");

System.out.println("\none = "+ Arrays.binarySearch(sa,"one"));

System.out.println("one = "+ Arrays.binarySearch(sa,"one",rs));

}

static class ReSortComparator

implements Comparator<String> {

public int compare(String a, String b) {

return b.compareTo(a);

}

**Output:**

four one three two

one = 1

now reverse sort

two three one four

one = -1

one = 2

The Arrays.asList() method copies an array into a List. The API says, "Returns a fixed-size list backed by the specified array. (Changes to the returned list 'write through' to the array.)" When you use the asList() method, the array and the List become joined at the hip**. When you update one of them, the other is updated automatically**.

String[] sa = {"one", "two", "three", "four"};

List sList = Arrays.asList(sa); // make a List

System.out.println("size " + sList.size());

System.out.println("idx2 " + sList.get(2));

sList.set(3,"six"); // change List

sa[1] = "five"; // change array

for(String s : sa)

System.out.print(s + " ");

System.out.println("\nsl[1] " + sList.get(1));

**Output:**

size 4

idx2 three

one five three six

sl[1] five

Now let's take a look at the toArray() method. There's nothing too fancy going on with the toArray() method; it comes in two flavors: one that returns a new Object array, and one that uses the array you send it as the destination array:

List<Integer> iL = new ArrayList<Integer>();

for(int x=0; x<3; x++)

iL.add(x);

Object[] oa = iL.toArray(); // create an Object array

Integer[] ia2 = new Integer[3];

ia2 = iL.toArray(ia2); // create an Integer array

If you change ia2 it won’t reflect changes to IL. If you change IL it wont’ reflect changes to ia2.

lowerKey() & lower(): Returns the element less than the given element.

higherKey() & higher(): Returns the element greater than the given element.

ceilingKey() & ceiling(): returns the element greater than *or equal to* the given element.

For the exam, you'll need to understand the basics just explained, plus a few more details about three methods from TreeSet—headSet(), subSet(), and tailSet() and three methods from TreeMap headMap(), subMap(), and tailMap().

TreeSet<Integer> times = new TreeSet<Integer>();

times.add(1205); // add some departure times

times.add(1505);

times.add(1545);

times.add(1830);

times.add(2010);

times.add(2100);

TreeSet<Integer> subset = new TreeSet<Integer>();

subset = (TreeSet)times.headSet(1600);

System.out.println("J5 - last before 4pm is: " + subset.last());

TreeSet<Integer> sub2 = new TreeSet<Integer>();

sub2 = (TreeSet)times.tailSet(2000);

System.out.println("J5 - first after 8pm is: " + sub2.first());

// Java 6 version using the new lower() and higher() methods

System.out.println("J6 -last before 4pm is:"+**times.lower(1600)**);

System.out.println("J6-first after 8pm is:"+**times.higher(2000)**);

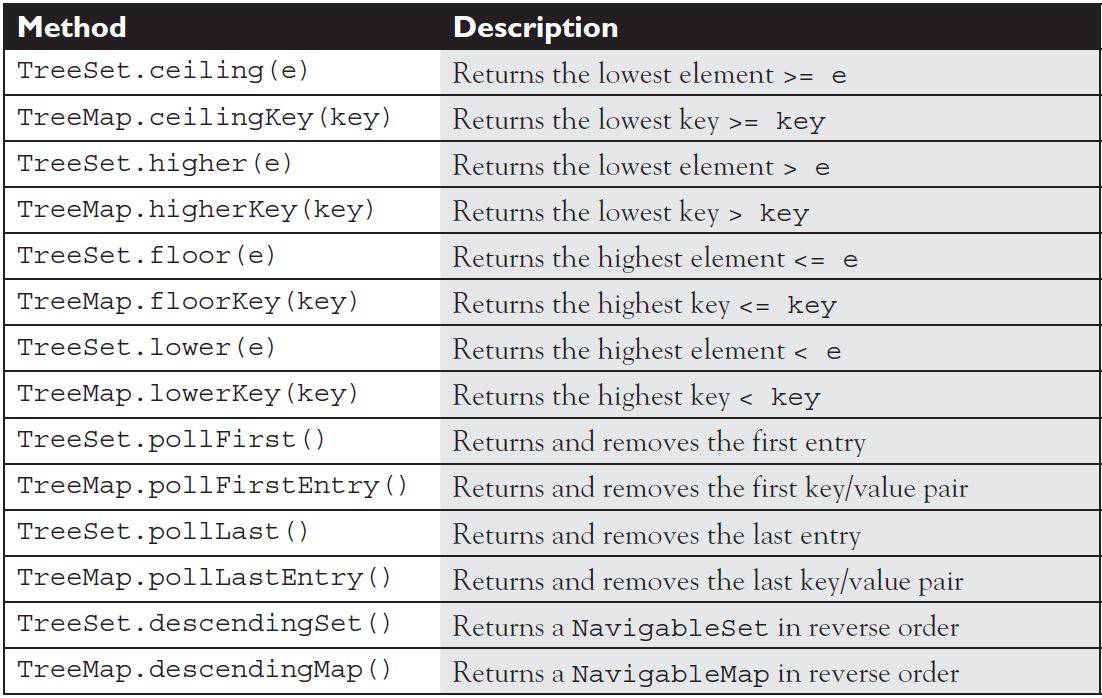
**Output:**

J5 - last before 4pm is: 1545

J5 - first after 8pm is: 2010

J6 - last before 4pm is: 1545

J6 - first after 8pm is: 2010



**Polling**

In the case of TreeSet, pollFirst() returns and removes the first entry in the set, and pollLast() returns and removes the last. Similarly, TreeMap now provides pollFirstEntry() and pollLastEntry() to retrieve and remove key/value pairs.

**Descending Order**

Also new to Java 6 for TreeSet and TreeMap are methods that return a collection in the reverse order of the collection on which the method was invoked. The important methods for the exam are TreeSet.descendingSet() and TreeMap.descendingMap().

**Backed Collections**

Some of the classes in the java.util package support the concept of "backed collections." We'll use a little code to help explain the idea:

TreeMap<String, String> map = new TreeMap<String, String>();

map.put("a", "ant"); map.put("d", "dog"); map.put("h", "horse");

SortedMap<String, String> submap;

submap = map.subMap("b", "g"); // #1 create a backed collection

System.out.println(map + " " + submap); // #2 show contents

map.put("b", "bat"); // #3 add to original

submap.put("f", "fish"); // #4 add to copy

map.put("r", "raccoon"); // #5 add to original - out of range

// submap.put("p", "pig"); // #6 add to copy - out of range

System.out.println(map + " " + submap); // #7 show final contents

**Output:**

{a=ant, d=dog, h=horse} {d=dog}

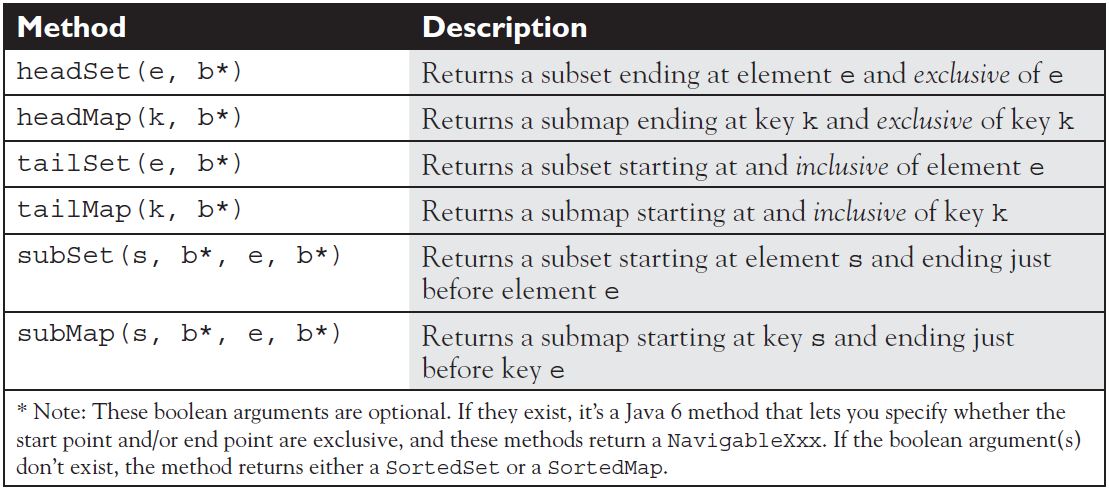
{a=ant, b=bat, d=dog, f=fish, h=horse, r=raccoon} {b=bat, d=dog, f=fish}

When we add key/value pairs to either the original TreeMap or the partial-copy SortedMap, the new entries were automatically added to the other collection sometimes.

we can add new entries to either collection within the range of the copy, and the new entries will show up in both collections. In addition, we can add a new entry to the original collection, even if it's outside the range of the copy. In this case, the new entry will show up only in the original it won't be added to the copy because it's outside the copy's range.

**Tip:**

Let’s say that you’ve created a backed collection using either a tailXxx() or subXxx() method. Typically in these cases, the original and copy collections have different “first” elements. For the exam, it’s important that you remember that the pollFirstXxx() methods will always remove the first entry from the collection on which they’re invoked, but they will remove an element from the other collection only if it has the same value. So it’s most likely that invoking pollFirstXxx() on the copy will remove an entry from both collections, but invoking pollFirstXxx() on the original will remove only the entry from the original.



PriorityQueue follows FIFO order. In addition, a PriorityQueue can be ordered using a Comparator, which lets you define any ordering you want. Queues have a few methods not found in other collection interfaces: peek(), poll(), and offer().

class PQsort

implements Comparator<Integer> {

public int compare(Integer one, Integer two) {

return two - one; // unboxing and **inverse sort**

}

}

public static void main(String[] args) {

int[] ia = {1,5,3,7,6,9,8 }; // unordered data

PriorityQueue<Integer> pq1 = new PriorityQueue<Integer>(); // use natural order

for(int x : ia) // load queue

pq1.offer(x);

for(int x : ia) // review queue

System.out.print(pq1.poll() + " ");

System.out.println("");

PQsort pqs = new PQsort(); // get a Comparator

PriorityQueue<Integer> pq2 =

new PriorityQueue<Integer>(10,pqs); // use Comparator

for(int x : ia) // load queue

pq2.offer(x);

System.out.println("size " + pq2.size());

System.out.println("peek " + pq2.peek());

System.out.println("size " + pq2.size());

System.out.println("poll " + pq2.poll());

System.out.println("size " + pq2.size());

for(int x : ia) // review queue

System.out.print(pq2.poll() + " ");

**Output:**

1 3 5 6 7 8 9

size 7

peek 9

size 7

poll 9

size 6

8 7 6 5 3 1 null

String[] sa = {">ff<", "> f<", ">f <", ">FF<" }; // ordered?

PriorityQueue<String> pq3 = new PriorityQueue<String>();

for(String s : sa)

pq3.offer(s);

for(String s : sa)

System.out.print(pq3.poll() + " ");

***Output:***

> f< >FF< >f < >ff<

If you remember that spaces sort before characters and that uppercase letters sort before lowercase characters, you should be good to go for the exam.

You're also free to put boundaries on the type you declare. For example, if you want to restrict the makeArrayList() method to only Number or its subtypes (Integer, Float, and so on), you would say.

public <T extends Number> void makeArrayList(T t)

public void makeList(T t) { }// legal method

public class Radio {

public <T> Radio(T t) { } // legal constructor

}

You can declare a class with a name that is the same as the type parameter placeholder:

class X { public <X> X(X x) { } }

One of the most common mistakes programmers make when creating generic classes or methods is to use a <?> in the wildcard syntax rather than a type variable <T>, <E>, and so on. This code might look right, but isn't:

public class NumberHolder<? extends Number> { }

While the question mark works when declaring a reference for a variable, it does NOT work for generic class and method declarations. This code is not legal:

public class NumberHolder<?> { ? aNum; } // NO!

But if you replace the <?> with a legal identifi er, you're good:

public class NumberHolder<T> { T aNum; } // Yes

* equals(), hashCode(), and toString() are public.
* If you don't override equals(), your objects won't be useful hashing keys.
* If you don't override equals(), different objects can't be considered equal.
* Strings and wrappers override equals() and make good hashing keys.
* When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
* Highlights of the equals() contract:
* Reflexive: x.equals(x) is true.
* Symmetric: If x.equals(y) is true, then y.equals(x) must be true.
* Transitive: If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
* Consistent: Multiple calls to x.equals(y) will return the same result.
* Null: If x is not null, then x.equals(null) is false.
* If x.equals(y) is true, then x.hashCode() == y.hashCode() is true.
* HashMap, HashSet, Hashtable, LinkedHashMap, and LinkedHashSet use hashing.
* Highlights of the hashCode() contract:
* Consistent: Multiple calls to x.hashCode() return the same integer.
* If x.equals(y) is true, x.hashCode() == y.hashCode() is true.
* If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.
* Transient variables aren't appropriate for equals() and hashCode().
* **ArrayList** Fast iteration and fast random access.
* **Vector** It's like a slower ArrayList, but it has synchronized methods.
* **LinkedList** Good for adding elements to the ends, i.e., stacks and queues.
* **HashSet** Fast access, assures no duplicates, provides no ordering.
* **LinkedHashSet** No duplicates; iterates by insertion order.
* **TreeSet** No duplicates; iterates in sorted order.
* **HashMap** Fastest updates (key/values); allows one null key, many null values.
* **Hashtable** Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
* **LinkedHashMap** Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
* **TreeMap** A sorted map.
* **PriorityQueue** A to-do list ordered by the elements' priority.
* Sorting can be in natural order or via a Comparable or many Comparators.
* To be searched, an array or List must first be sorted.
* A sort() method. Sort using a Comparator or sort using natural order.
* A binarySearch() method. Search a presorted array or List.
* Arrays.asList() creates a List from an array and links them together.
* Collections.reverse() reverses the order of elements in a List.
* Collections.reverseOrder() returns a Comparator that sorts in reverse.
* Lists and Sets have a toArray() method to create arrays.
* You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type-safe collection.
* Polymorphic assignments apply only to the base type, not the generic type parameter. You can say

List<Animal> aList = new ArrayList<Animal>(); // yes

You can't say

List<Animal> aList = new ArrayList<Dog>(); // no

* When using a wildcard List<? extends Dog>, the collection can be accessed but not modified.
* When using a wildcard List<?>, any generic type can be assigned to the reference, but for access only—no modifications.
* List<Object> refers only to a List<Object>, while List<?> or List<? extends Object> can hold any type of object, but for access only.
* Declaration conventions for generics use T for type and E for element.
* The generics type identifier can be used in class, method, and variable declarations:

class Foo<t> { } // a class

T anInstance; // an instance variable

Foo(T aRef) {} // a constructor argument

void bar(T aRef) {} // a method argument

T baz() {} // a return type

* You can declare a generic method using a type not defined in the class:

public <T> void makeList(T t) { }

This is NOT using T as the return type. This method has a void return type, but to use T within the argument, you must declare the <T>, which happens before the return type.

Just because a thread's sleep() expires and it wakes up, this does not mean it will return to running! Remember, when a thread wakes up, it simply goes back to the runnable state.

You can put sleep() code anywhere, since *all* code is being run by *some* thread. When the executing code hits a sleep() call, it puts the currently running thread to sleep.

Don't rely on thread priorities when designing your multithreaded application. Because thread scheduling priority behavior is not guaranteed, use thread priorities as a way to improve the efficiency of your program, but just be sure your program doesn't depend on that behavior for correctness.

When the wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately.

* You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a IllegalThreadStateException.
* There is no guarantee that the order in which threads were started determines the order in which they'll run.
* There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns, regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.)
* A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
* A dead thread cannot be started again.
* A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
* The sleep() method is a static method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.
* The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels some levels may be treated as effectively equal.
* The yield() method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.
* The closest thing to a guarantee is that at any given time, when a thread is running, it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
* While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
* static methods can be synchronized using the lock from the java.lang.Class instance representing that class.
* There is also a variation of the tryLock method that allows you to specify an amount of time you are willing to wait to acquire the lock.

boolean locked = lock.tryLock(3, TimeUnit.SECONDS);

* Another benefit of the tryLock method is deadlock avoidance.

Object o1 = new Object();

Object o2 = new Object();

synchronized(o1) {

// thread A could pause here

synchronized(o2) {

// work

}

}

You should never acquire the locks in the opposite order because it could lead to deadlock.

synchronized(o2) {

// thread B gets stuck here

synchronized(o1) {

// work

}

}

Looking at a similar example using a ReentrantLock, start by creating two locks:

Lock l1 = new ReentrantLock();

Lock l2 = new ReentrantLock();

Next, you acquire both locks in thread A:

boolean aq1 = l1.tryLock();

boolean aq2 = l2.tryLock();

try{

if (aq1 && aq2) {

// work

}

} finally {

if (aq2) l2.unlock(); // don't unlock if not locked

if (aq1) l1.unlock();

}

If a thread attempts to release a lock that it does not own, an IllegalMonitorStateException will be thrown.

Now in thread B, the locks are obtained in the reverse order in which thread A obtained them. With traditional locking, using synchronized code blocks and attempting to obtain locks in the reverse order could lead to deadlock.

boolean aq2 = l2.tryLock();

boolean aq1 = l1.tryLock();

try{

if (aq1 && aq2) {

// work

}

} finally {

if (aq1) l1.unlock();

if (aq2) l2.unlock();

}

Now, even if thread A was only in possession of the l1 lock, there is no possibility that thread B could block because we use the non-blocking tryLock method. Using this technique, you can avoid deadlocking scenarios, but you must deal with the possibility that both locks could not be acquired. Using a simple loop, you can repeatedly attempt to obtain both locks until successful (Note: This approach is CPU intensive; we'll look at a better solution next):

loop2:

while (true) {

boolean aq2 = l2.tryLock();

boolean aq1 = l1.tryLock();

try {

if (aq1 && aq2) {

// work

break loop2;

}

} finally {

if (aq2) l2.unlock();

if (aq1) l1.unlock();

}

}

**Livelock:** It is remotely possible that this example could lead to livelock. Imagine if thread A always acquires lock1 at the same time that thread B acquires lock2. Each thread's attempt to acquire the second lock would always fail, and you'd end up repeating forever, or at least until you were lucky enough to have one thread fall behind the other. You can avoid livelock in this scenario by introducing a short random delay with Thread.sleep(int) any time you fail to acquire both locks.

**Condition**

A Condition provides the equivalent of the traditional wait, notify, and notifyAll methods. The traditional wait and notify methods allow developers to implement an await/signal pattern. You use an await/signal pattern when you would use locking.

Lock lock = new ReentrantLock();

Condition blockingPoolA = lock.newCondition();

blockingPoolA.**await()**;

blockingPoolA.**signalAll()**;

blockingPoolA.**signal()**;

One advantage of a Condition over the traditional wait/notify operations is that multiple Conditions can exist for each Lock. A Condition is effectively a waiting/ blocking pool for threads.

Lock lock = new ReentrantLock();

Condition blockingPoolA = lock.newCondition();

Condition blockingPoolB = lock.newCondition();

A for-each loop uses an Iterator when executing, so it is safe to use with a copy-on-write collection, unlike a traditional for loop.

for(Object o : collection) {} // use this

for(int i = 0; i < collection.size(); i++) {} // not this

ConcurrentHashMap, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet.

ConcurrentSkipListMap and ConcurrentSkipListSet are sorted. ConcurrentSkipListMap keys and ConcurrentSkipListSet elements require the use of the Comparable or Comparator interfaces to enable ordering.

Because a LinkedTransferQueue implements the BlockingQueue, TransferQueue, and Queue interfaces, it can be used to showcase all the different methods that can be used to add and remove elements using the various types of queues. Creating a LinkedTransferQueue is easy. Because LinkedTransferQueue is not bound by size, a limit to the number of elements CANNOT be supplied to its constructor.

TransferQueue<Integer> tq = new LinkedTransferQueue<>(); // not bounded

**ThreadLocalRandom**

The first Callable example used a java.util.concurrent.ThreadLocalRandom. ThreadLocalRandom is a new way in Java 7 to create random numbers. Math. random() and shared Random instances are thread-safe, but suffer from contention when used by multiple threads. A ThreadLocalRandom is unique to a thread and will perform better because it avoids any contention. ThreadLocalRandom also provides several convenient methods such as nextInt(int,int) that allow you to specify the range of possible values returned.

The ReentrantLock class provides the basic Lock implementation. Commonly used methods are lock(), unlock(), isLocked(), and tryLock(). Calling lock() increments a counter and unlock() decrements the counter. A thread can only obtain the lock when the counter is zero.

add and remove of CopyOnWriteArrayList will modify the list and cause a new internal array to be created.

**ArrayBlockingQueue :** The add method will throw an IllegalStateException if the queue is full. The two offer methods will return false if the queue is full. Only the put method will block until space becomes available.

**public ResultSet getResultSet() throws SQLException** If the boolean value from the execute() method returns true, then there is a result set. To get the result set, as shown earlier, call the getResultSet() method on the Statement object. Then you can process the ResultSet object.

ResultSet rs = stmt.getResultSet();

**public int getUpdateCount() throws SQLException** If the boolean value from the execute() method returns false, then there is a row count, and this method will return the number of rows affected. A return value of –1 indicates that there are no results.

int numRows = stmt.getUpdateCount();

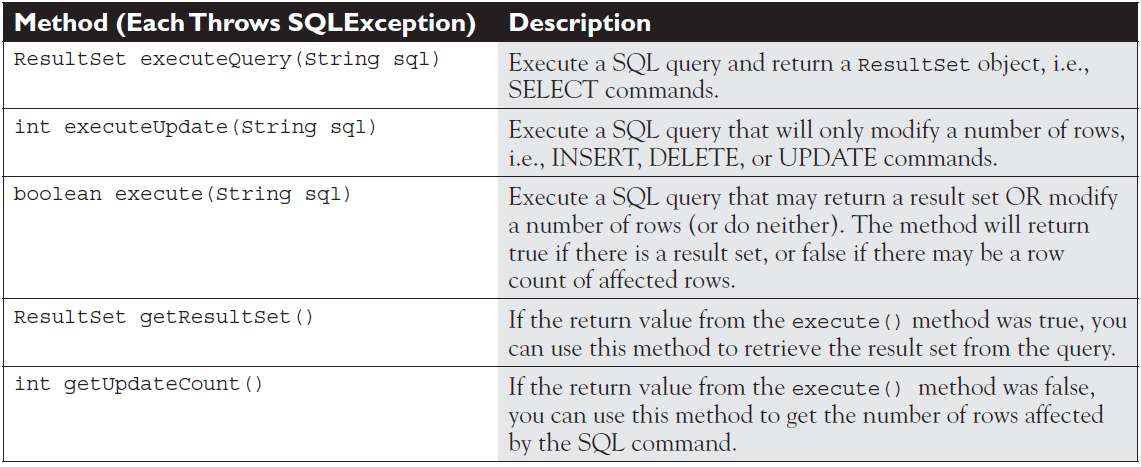
if (numRows == -1) {

out.println("No results");

} else {

out.println(numRows + " rows affected.");

}



Using ResultSetMetaData, you can get important information about the results returned from the query, including the number of columns, the table name, the column name, and the column class name the Java class that is used to represent this column when the column is returned as an Object.

String query = "SELECT AuthorID FROM Author";

ResultSet rs = stmt.executeQuery(query);

ResultSetMetaData rsmd = rs.getMetaData();

rs.next();

int colCount = rsmd.getColumnCount(); // How many columns in this

// ResultSet?

out.println("Column Count: " + colCount);

for (int i = 1; i <= colCount; i++) {

out.println("Table Name: " + rsmd.getTableName(i));

out.println("Column Name: " + rsmd.getColumnName(i));

out.println("Column Size: " + rsmd.getColumnDisplaySize(i));

}

* **TYPE\_FORWARD\_ONLY** The default value for a ResultSet the cursor moves forward only through a set of results.
* **TYPE\_SCROLL\_INSENSITIVE** A cursor position can be moved in the result forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data the database itself are not reflected in the result set. In other words, the result set does not have to "keep state" with the database. This type is generally supported by databases.
* **TYPE\_SCROLL\_SENSITIVE** A cursor can be changed in the results forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data are reflected in the open result set. As you can imagine, this is difficult to implement, and is therefore not implemented in a database or JDBC driver very often.
* **CONCUR\_READ\_ONLY** This is the default value for result set concurrency. Any open result set is read-only and cannot be modified or changed.
* **CONCUR\_UPDATABLE** A result set can be modified through the ResultSet methods while the result set is open.
* The default ResultSet is not updatable (read-only), and the cursor moves forward only.
* A ResultSet that is scrollable and updatable can be modified, and the cursor can be positioned anywhere within the ResultSet.
* ResultSetMetaData can be used to dynamically discover the number of columns and their type returned in a ResultSet.
* ResultSetMetaData does not have a row count method. To determine the number of rows returned, the ResultSet must be scrollable.
* When a Connection is closed, all of the related Statements and ResultSets are closed.
* PreparedStatements are precompiled and can increase efficiency for frequently used SQL queries.
* PreparedStatement is a good way to avoid SQL injection attacks.
* PreparedStatement setXXXX methods are indexed from 1, not 0.
* CallableStatements are executed using a stored procedure on the database.