*OCP Java SE 7 Programmer II*

*7. Assertions and Java 7 Exceptions*

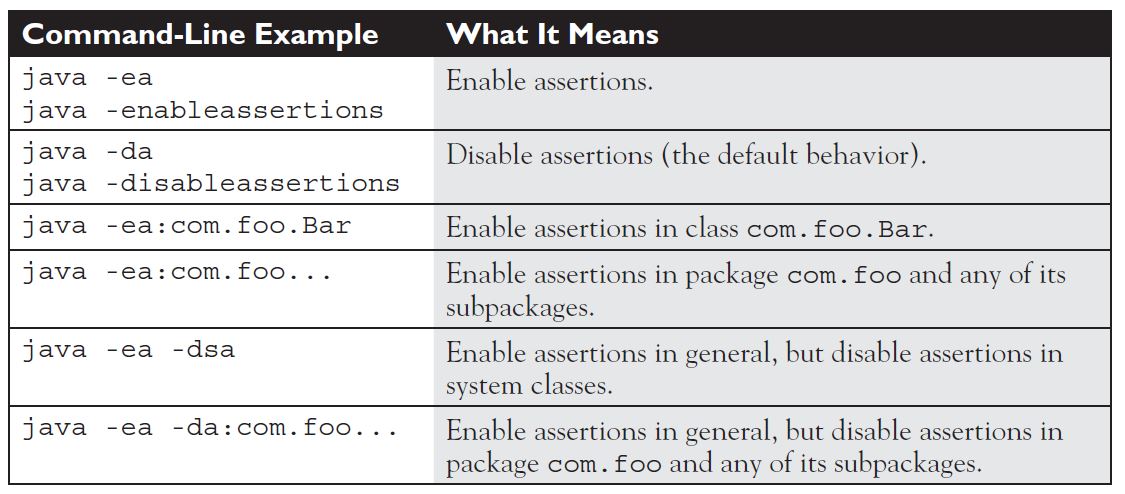
***Assertions:***

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() );

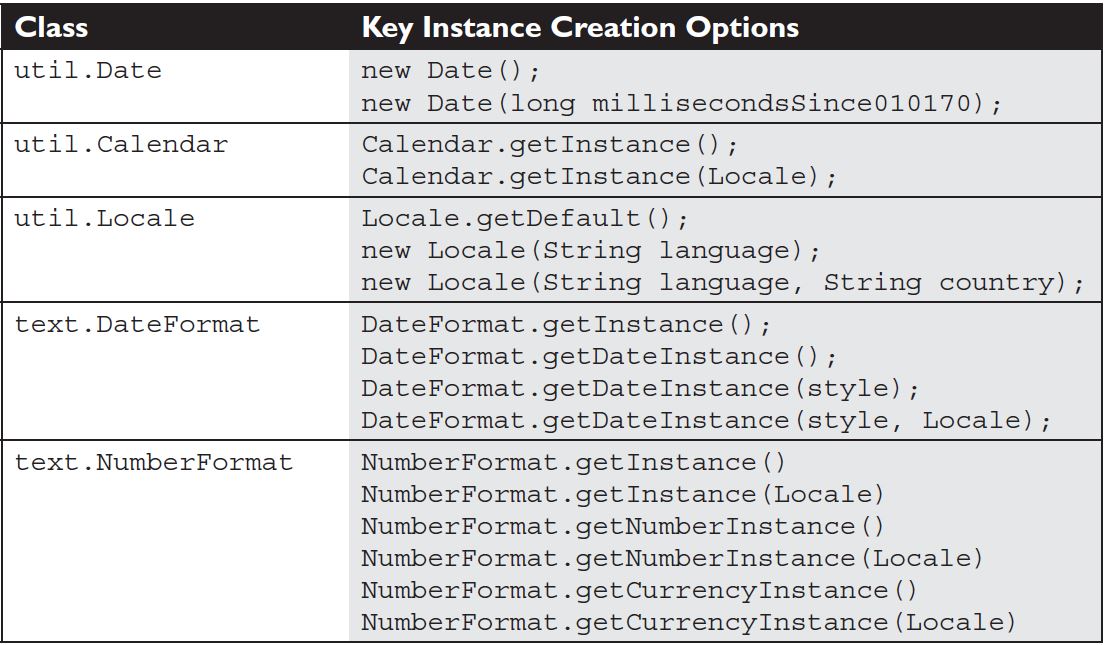
**The output would be something like this:**

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

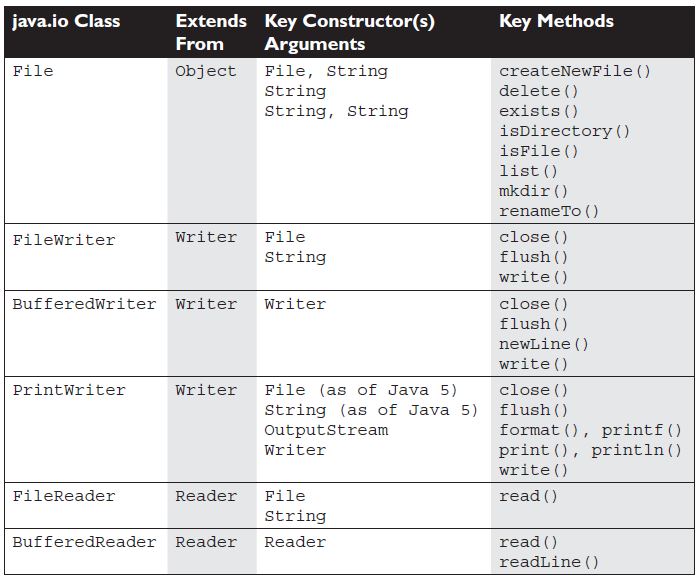
c char

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

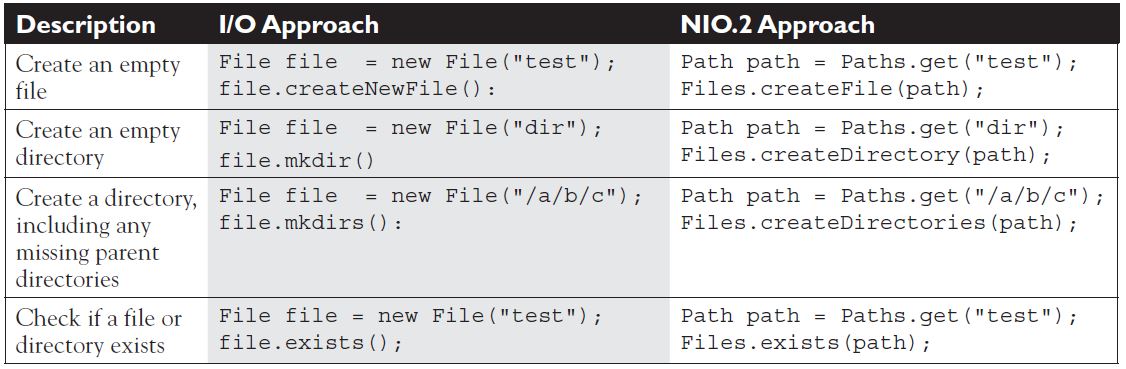
Files.createDirectory(path2); // create second level of directory

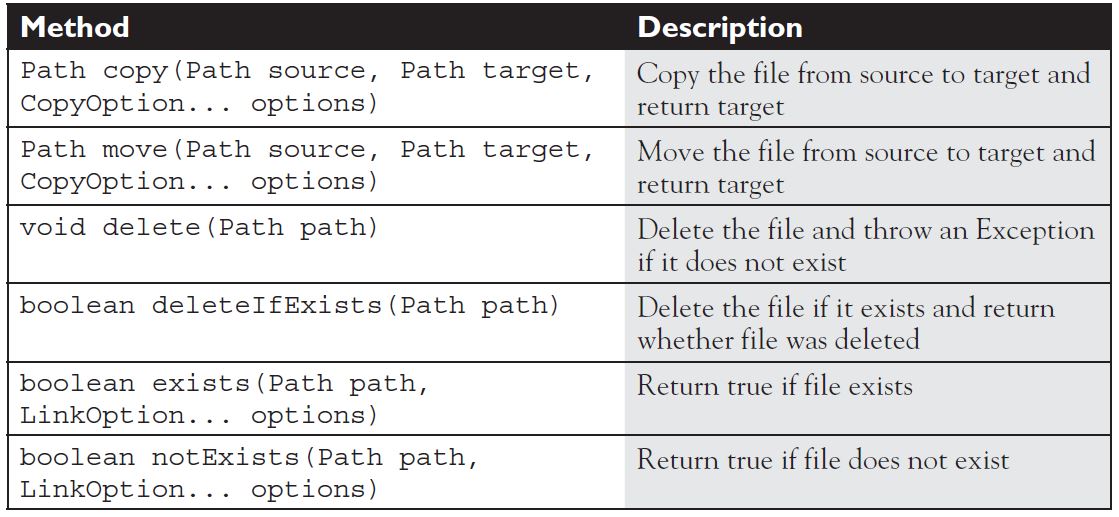
Files.createFile(file); // create file

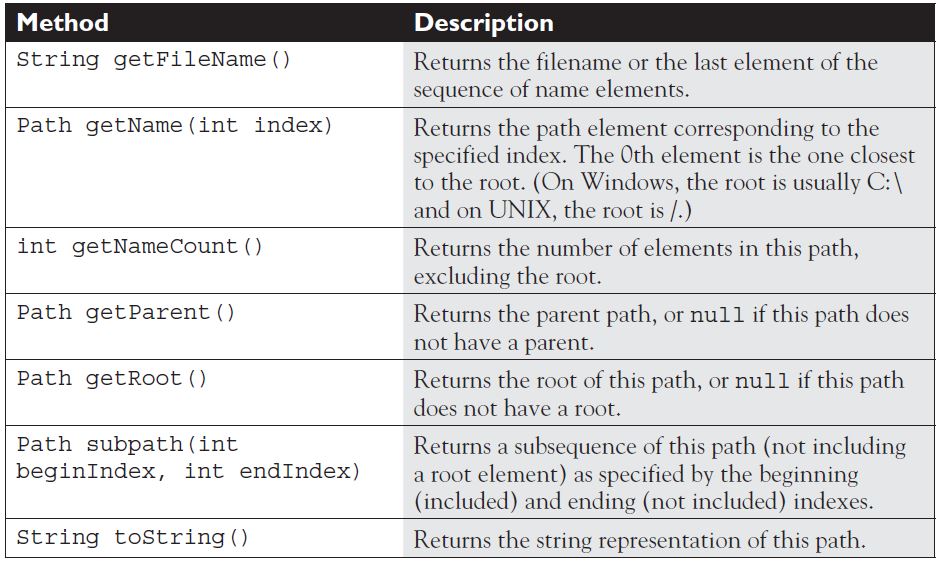
Or we could create all the directories in one go:

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.