**Kaithe S.------ 10.Development**

Compile java program

javac [options] [source files]

Both the [options] and the [source files] are optional parts of the command, and both allow multiple entries.

The following are both legal javac commands:

javac -help

javac -classpath com:. -g Foo.java Bar.java

Whenever you specify multiple options and/or files they should be separated by spaces.

By default, the compiler puts a .class file in the same directory as the .java source file.

The -d option lets you tell the compiler in which directory to put the .class file(s) it generates (d is for destination).

javac -d ../classes com/wickedlysmart/MyClass.java

Because MyClass.java is in a package,

the compiler knew to put the resulting .class file into the

classes/com/wickedlysmart directory.

The last thing about -d that you'll need to know for the exam is that if the

destination directory you specify doesn't exist, you'll get a compiler error.

java:5: error while writing MyClass: classes/MyClass.class (No

such file or directory)

**Run the java program**

java [options] class [args]

[options] and [args] parts of the java command are optional, and they

can both have multiple values.

You must specify exactly one class file to execute, and you don't specify

the .class extension on the command line.

java -DmyProp=myValue MyClass x 1

this command can be read as "Create a *system*

*property* called myProp and set its value to myValue. Then launch the file named

MyClass.class and send it two String *arguments* whose values are x and 1."

System Properties

java.util.Properties that can be used to access a

system's persistent information such as the current versions of the operating system,

the Java compiler, and the Java virtual machine.

you can also add and retrieve your own properties:

import java.util.\*;

public class TestProps {

public static void main(String[] args) {

Properties p = System.getProperties();

p.setProperty("myProp", "myValue");

p.list(System.out);

}

}

If this file is compiled and invoked as follows:

java -DcmdProp=cmdVal TestProps

Two name=value properties were added

to the system's properties: myProp=myValue was added via the setProperty

method, and cmdProp=cmdVal was added via the -D option at the command line.

When using the -D option, if your value contains white space the entire value

should be placed in quotes like this:

java -DcmdProp="cmdVal take 2" TestProps

when you use -D, the name=value pair must follow

*immediately*, no spaces allowed.

The getProperty() method is used to retrieve a single property. It can be

invoked with a single argument (a String that represents the name (or key)), or it

can be invoked with two arguments, (a String that represents the name (or key),

and a default String value to be used as the property if the property does not already

exist). In both cases, getProperty() returns the property as a String.

Command Line Arguments:

public class CmdArgs {

public static void main(String[] args) {

int x = 0;

for(String s : args)

System.out.println(x++ + " element = " + s);

}

}

compiled and then invoked as follows

java CmdArgs x 1

the output will be

0 element = x

1 element = 1

The following are all legal declarations for main():

static public void main(String[] args)

public static void main(String... x)

static public void main(String bang\_a\_gong[])

if javac needed access to java.util.HashMap then the java command will need

to find java.util.HashMap as well.

Both java and javac use the same basic search algorithm:

1. They both have the same list of places (directories) they search, to look

for classes.

2. They both search through this list of directories in the same order.

3. As soon as they find the class they're looking for, they stop searching for that

class. In the case that their search lists contain two or more files with the

same name, the first file found will be the file that is used.

4. The first place they look is in the directories that contain the classes that

come standard with J2SE.

5. The second place they look is in the directories defined by classpaths.

6. Classpaths should be thought of as "class search paths." They are lists of

directories in which classes might be found.

7. There are two places where classpaths can be declared:

A classpath can be declared as an operating system environment variable.

The classpath declared here is used by default, whenever java or javac are

invoked.

A classpath can be declared as a command-line option for either java or

javac. *Classpaths declared as command-line options override the classpath declared*

*as an environment variable, but they persist only for the length of the invocation.*

Classpaths consist of a variable number of directory locations, separated by

delimiters.

For Unix-based operating systems, forward slashes(/) are used to

construct directory locations, and the separator is the colon (:).

For Windows-based operating systems, back slashes(\) are used to

construct directory locations, and the separator is the semicolon (;).

The way to tell java or

javac to search in the current directory is to add a dot (.) to the classpath

classpaths are searched from left to right.

Use one class in another class:

import com.foo.MyClass; // either import will work

import com.foo.\*;

public class Another {

void go() {

MyClass m1 = new MyClass(); // alias name

com.foo.MyClass m2 = new com.foo.MyClass(); // or using fully qualified name

m1.hi();

m2.hi();

}

}

**Relative and Absolute Paths**

Unix begins with a forward

slash (/) (on Windows it would be something like c:\). The leading slash indicates

that this path is starting from the root directory of the system. Because it's starting

from the root, it doesn't *matter* what the current directory is—*a directory's absolute*

*path is always the same*. A *relative* path is one that does NOT start with a slash.

JAR stands for Java Archive. JAR files

are used to compress data (similar to ZIP files) and to archive data.

cd ws

jar -cf MyJar.jar myApp

The jar command will create a JAR file called MyJar.jar and it will contain the

myApp directory and myApp's entire subdirectory tree and files

You can look at the

contents of the JAR file with the next command

jar -tf MyJar.jar

***The jar command creates the*** META-INF ***directory automatically.***

***The jar command creates the*** MANIFEST.MF ***fi le automatically.***

***The jar command won’t place any of your fi les in*** META-INF/***.***

compile UseStuff.java in the test directory, and UseStuff.java needs access

to a class contained in myApp.jar.

cd test

javac -classpath ws/myApp.jar UseStuff.java

when using a classpath, the last directory in the path must be the

super-directory of the *root* directory for the package.

cd test

javac -classpath ws UseStuff.java

***importing multiple***

***packages—just remember, an*** import ***statement can import only a single package.***

***you can’t say*** import java.\*;

import java.util.\*; ***you are saying "Use the short name for all of the***

***classes in the*** java.util ***package.***

jre/lib/ext. If you put JAR files into the ext

subdirectory, java and javac can find them, and use the class files they contain.

You don't have to mention these subdirectories in a classpath statement – not recommended for software development.

import statements have is that they save typing and they can make your code

easier to read.

Static imports can

be used when you want to use a class's static members.

Before static imports:

public class TestStatic {

public static void main(String[] args) {

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.toHexString(42));

}

}

After static imports:

import static java.lang.System.out;//1 static import on the System class out object

import static java.lang.Integer.\*; //2 static imports of ALL the static members

public class TestStaticImport {

public static void main(String[] args) {

out.println(MAX\_VALUE); // 3

out.println(toHexString(42)); // 4

}

}

Disadvantage:

Make the code a little harder to read.

Watch out for ambiguously named static members. For instance, if you do

a static import for both the Integer class and the Long class, referring to

MAX\_VALUE will cause a compiler error, since both Integer and Long have

a MAX\_VALUE constant, and Java won't know which MAX\_VALUE you're referring

to.

6. Strings, I/O,Formatting,and Parsing

In Java, each character in a string is a 16-bit Unicode character. Because Unicode characters are 16 bits a rich, international set of characters is easily represented in Unicode.

In Java, strings are objects.

you can create an instance of a String with the new keyword, as follows:

String s = new String();

String s = new String("abcdef");

String s = "abcdef";

they all create a new String object, with a value of "abcdef", and assign it to a reference variable s.

String s2 = s; // refer s2 to the same String as s

Once you have assigned a String a value, that value can never change— it's immutable,

String object is immutable, its reference variable is not.

s = s.concat(" more stuff"); // the concat() method 'appends'

// a literal to the end

VM creates a new String object, gave it the value "abcdef more stuff" and made s refer to it.

There are now three String objects, because the literal argument to concat, " more stuff", is itself a new String object. But we have references only to "abcdef" (referenced by s2) and "abcdef more stuff"(referenced by s).

String x = "Java";

x.concat(" Rules!");

System.out.println("x = " + x); // the output is "x = Java"

The VM creates a second String object with the value "Java Rules!" but nothing refers to it. The second String object is instantly lost. The reference variable x still refers to the original String with the value "Java".

x.toUpperCase();

System.out.println("x = " + x); // the output is still:

// x = Java

x.replace('a', 'X');

System.out.println("x = " + x); // the output is still:

// x = Java

String x = "Java";

x = x.concat(" Rules!"); // Now we're assigning the

// new String to x

System.out.println("x = " + x); // the output will be:

// x = Java Rules!

JVM sets aside a special area of memory called the "String constant pool."

Why making String objects immutable?

If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.

If someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked final.

String s = "abc"; // creates one String object and one

// reference variable

"abc" will go in the pool and s will refer to it.

String s = new String("abc"); // creates two objects,

// and one reference variable

we used the new keyword, Java will create a new String object in normal (nonpool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool.

Important String methods:

**charAt()** Returns the character located at the specified index

**concat()** Appends one String to the end of another ( "+" also works)

■ **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case

■ **length()** Returns the number of characters in a String

■ **replace()** Replaces occurrences of a character with a new character

■ **substring()** Returns a part of a String

■ **toLowerCase()** Returns a String with uppercase characters converted

■ **toString()** Returns the value of a String

■ **toUpperCase()** Returns a String with lowercase characters converted

■ **trim()** Removes whitespace from the ends of a String

StringBuilder class is not thread safe. In other words, its methods are not synchronized.

StringBuffer class is thread safe. In other words, its methods are synchronized.

StringBuffer sb = new StringBuffer("abc");

sb.append("def");

System.out.println("sb = " + sb); // output is "sb = abcdef"

StringBuilder sb = new StringBuilder("abc");

sb.append("def").reverse().insert(3, "---");

System.out.println( sb ); // output is "fed---cba"

Each example needed only a single StringXxx object to execute.

result = method1().method2().method3();

Start with leftmost method and the result we have to use for the right methods.

String x = "abc";

String y = x.concat("def").toUpperCase().replace('C','x');

//chained methods

System.out.println("y = " + y); // result is "y = ABxDEF"

file I/O,

console I/O,

thread I/O,

high-performance I/O,

byte-oriented I/O,

character-oriented I/O,

I/O filtering and wrapping,

serialization,

I/O classes that’s need to understand for the exam:

1. File:

* An abstract representation of file and directory pathnames.
* The File class isn't used to actually read or write data.
* used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.

1. **FileReader**

* This class is used to read character files.
* Its read() methods are low level methods, allowing you to read single characters, the whole stream of characters, or a fixed number of characters.
* FileReaders are usually *wrapped* by higher-level objects such as BufferedReaders

1. BufferedReader

* BufferedReaders read relatively large chunks of data from a file at once, and keep this data in a buffer.
* BufferedReader provides more convenient methods such as readLine(), that allow you to get the next line of characters from a file.

1. FileWriter

* This class is used to write to character files.
* Its write()methods allow you to write character(s) or Strings to a file.
* FileWriters are usually *wrapped* by higher-level Writer objects such as BufferedWriters or PrintWriters

1. BufferedWriter

* BufferedWriters write relatively large chunks of data to a file at once,
* The BufferedWriter class also provides a newLine()method to create platform-specific line separators automatically.

1. PrintWriter

* PrintWriter in can be used instead of FileWriter and/or a BufferedWriter
* New methods like format(), printf(), and append()make PrintWriters very flexible and powerful.

1. Console

* This new, Java 6 convenience class provides methods to read input from the console and write formatted output to the console.

***Stream classes are used to read and write bytes***

***Readers and Writers are used to read and write characters***

***Since all of the fi le I/O on the exam is related to characters, if you see API class names containing the word "Stream", for instance DataOutputStream, then the question is probably about serialization, or something unrelated to the actual I/O objective.***

***Creating a file, writing into the file and Reading from the file:\***

**import** java.io.File;

**public** **class** Test2 {

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

File file = **new** File("D:\test2.txt");

}

}

When you make a new instance of the class File, *you're not yet*

*making an actual file, you're just creating a filename*. Once you have a File *object*, there

are several ways to make an actual file.

***Creating a file***

**import** java.io.File;

**import** java.io.IOException;

**public** **class** Test2 {

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

**try** {

File file = **new** File("D:/test2.txt");

System.*out*.println(file.exists());// false

**boolean** newFile = **false**;

newFile = file.createNewFile();

System.*out*.println(newFile);// true

System.*out*.println(file.exists());// true

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**boolean exists()**

Return true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

**boolean createNewFile() throws** [**IOException**](eclipse-javadoc:%E2%98%82=Test/C:%5C/apps%5C/Java%5C/jdk1.6.0_37%5C/jre%5C/lib%5C/rt.jar%3Cjava.io(File.class%E2%98%83File~createNewFile%E2%98%82IOException)

Return true if the named file does not exist and was successfully created; false if the named file already exists

FileWriter

**package** com.test;

**import** java.io.File;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** Test3 {

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

**try** {

File file = **new** File("test3.txt");

FileWriter fw = **new** FileWriter(file); // Constructs a FileWriter object given a File object.

fw.write("Name: Suraj Kumar"); //Writes a string.

fw.flush(); //Flushes the stream.

fw.close(); //Closes the stream, flushing it first.

//Once the stream has been closed, further write() or flush() invocations will cause an IOException to be thrown.

//Closing a previously closed stream has no effect.

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

FileWriter fw = new FileWriter(file) did three things:

a. It created a FileWriter reference variable, fw.

b. It created a FileWriter object, and assigned it to fw.

c. It created an actual empty file out on the disk (and you can prove it).

**package** com.test;

**import** java.io.File;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**public** **class** Test3 {

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

File file = **new** File("D:/test3.txt");

**char**[] in = **new** **char**[50];

**try** {

FileWriter fw = **new** FileWriter(file); // Constructs a FileWriter object given a File object.

fw.write("Name: Suraj Kumar"); //Writes a string.

fw.flush(); //Flushes the stream.

fw.close(); //Closes the stream, flushing it first.

//Once the stream has been closed, further write() or flush() invocations will cause an IOException to be thrown.

//Closing a previously closed stream has no effect.

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

FileReader fr = **new** FileReader(file); // Creates a new FileReader, given the File to read from

**int** size = fr.read(in); // Reads characters into an array. This method will block until some input is available,

// an I/O error occurs, or the end of the stream is reached.

System.*out*.println("size = " + size); // size = 20

**for** (**char** c : in) {

System.*out*.print(c); // Name: Suraj Kumar

}

fr.close();

} **catch** (IOException ioe) {

ioe.printStackTrace();

}

}

}

When we were reading data back in, we put it into a character array. It

being an array and all, we had to declare its size beforehand, so we'd have

been in trouble if we hadn't made it big enough! We could have read the

data in one character at a time, looking for the end of file after each

read()

Because of these limitations, we should use higher-level I/O classes like BufferedWriter or BufferedReader in combination with FileWriter or FileReader.

Java's entire I/O system was designed around the idea of using several classes in

combination. Combining I/O classes is sometimes called *wrapping* and sometimes

called *chaining*.

File file = new File("fileWrite2.txt"); // create a File object AND

// open "fileWrite2.txt"

FileReader fr = new FileReader(file); // create a FileReader to get

// data from 'file'

BufferedReader br = new BufferedReader(fr); // create a BufferReader to

// get its data from a Reader

String data = br.readLine(); // read some data

***Creating a Directory***

first we create a Directory (like File object) object, then we create an actual directory using the following mkdir() method:

File myDir = new File("mydir"); // create an object

myDir.mkdir(); // create an actual directory

Once you've got a directory, you put files into it, and work with those files:

File myFile = new File(myDir, "myFile.txt");

myFile.createNewFile();

This code is making a new file in a subdirectory

you could write some data to the file myFile:

PrintWriter pw = new PrintWriter(myFile);

pw.println("new stuff");

pw.flush();

pw.close();

Be careful when you're creating new directories! As we've seen, constructing a

Writer or a Stream will often create a file for you automatically if one doesn't exist,

but that's not true for a directory:

File myDir = new File("mydir");

// myDir.mkdir(); // call to mkdir() omitted!

File myFile = new File(

myDir, "myFile.txt");

myFile.createNewFile(); // exception if no mkdir!

This will generate an exception something like

java.io.IOException: No such file or directory

***Create a directory, create a file, write in the file, and read from the file***

**package** com.test;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**public** **class** Test4 {

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

File myDir = **new** File("D:/myCreatedByJavaProg");

System.*out*.println(myDir.mkdir()); // true if and only if the directory was created; false otherwise

// first time return true otherwise false

File myFile = **new** File(myDir, "test4.txt");

**try** {

PrintWriter pw = **new** PrintWriter(myFile);

pw.println("Name: Suraj Kumar");

pw.println("Designation: SSE");

pw.println("Project: MNIS (CPDv2)");

pw.flush();

pw.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**try** {

/\*FileReader fr = new FileReader(myFile);

BufferedReader br = new BufferedReader(fr);

\*/

// Wrapped the FileReader with BufferedReader

BufferedReader br = **new** BufferedReader(**new** FileReader(**new** File("D:/myCreatedByJavaProg/test4.txt")));

String s = **null**;

**try** {

**while** ((s = br.readLine()) != **null**) {

System.*out*.println(s);

}

br.close();

} **catch** (IOException e) {

e.printStackTrace();

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

}

}

* When there is no more data to read, readLine() returns a null—this is our signal to stop reading the file.
* When reading a file, no flushing is required, so you won't even find a flush() method in a Reader kind of class.

***Rename a file, delete the file, and delete the directory***

**package** com.test;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**public** **class** Test5 {

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

// create a file

File fdir = **new** File("D:/myDir");

fdir.mkdir();

File file1 = **new** File(fdir, "myTest1.txt");

**try** {

file1.createNewFile();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// write into file

**try** {

PrintWriter pw = **new** PrintWriter(file1);

pw.println("Name: Suraj Kumar");

pw.flush();

pw.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// rename the file

System.*out*.println(file1.renameTo(**new** File(fdir, "newFile.txt")));// true

// Retruns: true if and only if the

// renaming succeeded; false otherwise

// The rename operation might not be able to move a file from one

// filesystem to another, it might not be atomic, and it

// might not succeed if a file with the destination abstract

// pathname already exists. The return value should always

// be checked to make sure that the rename operation was

// successful.

// delete a directory

System.*out*.println(fdir.delete());// false

// Returns: true if and only if the file or directory is successfully

// deleted; false otherwise

// Deletes the file or directory denoted by this abstract

// pathname.Deletes the file or directory denoted by this abstract

// pathname. If this pathname denotes a directory, then the directory

// must be empty in order to be deleted.

// delete a file

System.*out*.println(file1.delete());// false

System.*out*.println(**new** File(fdir, "newFile.txt").delete());// true

// Returns: true if and only if the file or directory is successfully

// deleted; false otherwise

System.*out*.println(fdir.delete());// true

}

}

***Deleting the Directory***

**import** java.io.File;

**public** **class** Test6 {

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

File fdir = **new** File("D:/newDir");

fdir.mkdir();

System.*out*.println(fdir.exists());// true

System.*out*.println(fdir.delete());// true

}

}

*Search files and Directories*

**package** com.test;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** Test7 {

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

File pf = **new** File("D:/newDir");

File sf = **new** File("D:/newDir/sonDir");

File sf2 = **new** File("D:/newDir/sonDir2");

// System.out.println(sf.mkdirs()); // true

// create newDir and sonDir

System.*out*.println(pf.mkdir());// true

System.*out*.println(sf.mkdir());// true

System.*out*.println(sf2.mkdir());// true

File fsf2 = **null**;

**try** {

**new** File(pf, "newFileinP.txt").createNewFile();

**new** File(sf, "newFileInS.txt").createNewFile();

fsf2 = **new** File(sf2, "newFileInS2.txt");

fsf2.createNewFile();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

String[] searchList = **new** String[10];

searchList = pf.list();

**for** (String sl : searchList) {

System.*out*.println(sl);

}

/\*

\* newFileinP.txt

\* sonDir

\* sonDir2

\*/

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

searchList = **new** String[10];

searchList = sf.list();

List<String> l = Arrays.*asList*(sf.list());

System.*out*.println(l);

**for** (String sl : searchList) {

System.*out*.println(sl);

}

/\*newFileInS.txt\*/

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

searchList = **new** String[10];

searchList = sf2.list();

**for** (String sl : searchList) {

System.*out*.println(sl);

}

/\*newFileInS2.txt\*/

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

searchList = **new** String[20];

searchList = **new** File("D:/").list();

**for** (String sl : searchList) {

System.*out*.println(sl);

}

/\* $RECYCLE.BIN

apps

codebase

cpdv2 war

docs

log

log.rar

myCreatedByJavaProg

newDir

softwares

System Volume Information

test2.txt

test3.txt

workspace

\*/

System.*out*.println(fsf2.getPath());//D:\newDir\sonDir2\newFileInS2.txt

System.*out*.println(fsf2.getParent());//D:\newDir\sonDir2

}

}

java.io.console

- New to Java 6 is the java.io.Console class

- you'll typically have access to a console object, to which you can get a reference by invoking System.console().

- it's possible for your Java program to be running in an environment that doesn't have access to a console object, so be sure that your invocation of System.console() actually returns a valid console reference and not null.

- The Console class makes it easy to accept input from the command line, both

echoed and nonechoed (such as a password), and makes it easy to write formatted

output to the command line.

* The readLine method returns a string containing whatever the user keyed in—that's pretty intuitive. However, the readPassword method doesn't return a string: it returns a character array.

import java.io.Console;

public class NewConsole {

public static void main(String[] args) {

Console c = System.console(); // #1: get a Console

char[] pw;

pw = c.readPassword("%s", "pw: "); // #2: return a char[]

for(char ch: pw)

c.format("%c ", ch); // #3: format output

c.format("\n");

MyUtility mu = new MyUtility();

while(true) {

name = c.readLine("%s", "input?: "); // #4: return a String

c.format("output: %s \n", mu.doStuff(name));

}

}

}

class MyUtility { // #5: class to test

String doStuff(String arg1) {

// stub code

return "result is " + arg1;

}

}

Serialization

* one to serialize objects and write them to a stream, and a second to read the stream and deserialize objects.

ObjectOutputStream.writeObject() // serialize and write

ObjectInputStream.readObject() // read and deserialize

* The java.io.ObjectOutputStream and java.io.ObjectInputStream classes are considered to be *higher*-level classes in the java.io package, and need to wrap them around *lower*-level classes, such as java.io.FileOutputStream and java.io.FileInputStream.

import java.io.\*;

class Cat **implements Serializable { } // 1**

public class SerializeCat {

public static void main(String[] args) {

**Cat c = new Cat(); // 2**

try {

FileOutputStream fs = new FileOutputStream("testSer.ser");

ObjectOutputStream os = new ObjectOutputStream(fs);

**os.writeObject(c); // 3**

os.close();

} catch (Exception e) { e.printStackTrace(); }

try {

FileInputStream fis = new FileInputStream("testSer.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

**c = (Cat) ois.readObject(); // 4**

ois.close();

} catch (Exception e) { e.printStackTrace(); }

}

}

* Serializable is a *marker* interface; it has no methods to implement.
* writeObject() performs two tasks: it serializes the object, and then it writes the serialized object to a file
* Create a FileOutputStream to write the object and wrap the FileOutputStream in an ObjectOutputStream,
* de-serialize the object by invoking the readObject() method. The readObject() method returns an Object, so we have to cast the deserialized object back to object type.

import java.io.\*;

public class SerializeDog {

public static void main(String[] args) {

Collar c = new Collar(3);

Dog d = new Dog(c, 5);

System.out.println("before: collar size is "

+ d.getCollar().getCollarSize());

try {

FileOutputStream fs = new FileOutputStream("testSer.ser");

ObjectOutputStream os = new ObjectOutputStream(fs);

os.writeObject(d);

os.close();

} catch (Exception e) { e.printStackTrace(); }

try {

FileInputStream fis = new FileInputStream("testSer.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

d = (Dog) ois.readObject();

ois.close();

} catch (Exception e) { e.printStackTrace(); }

System.out.println("after: collar size is "

+ d.getCollar().getCollarSize());

}

}

class Dog implements Serializable {

private Collar theCollar;

private int dogSize;

public Dog(Collar collar, int size) {

theCollar = collar;

dogSize = size;

}

public Collar getCollar() { return theCollar; }

}

class Collar implements Serializable {

private int collarSize;

public Collar(int size) { collarSize = size; }

public int getCollarSize() { return collarSize; }

}

This produces the output:

before: collar size is 3

after: collar size is 3

Remember, the most common reason to implement writeObject() and

readObject() is when you have to save some part of an object's state manually. If

you choose, you can write and read ALL of the state yourself, but that's very rare.

So, when you want to do only a *part* of the serialization/deserialization yourself, you

MUST invoke the defaultReadObject() and defaultWriteObject() methods

to do the rest.

class Dog implements Serializable {

**transient private Collar theCollar; // we can't serialize this**

private int dogSize;

public Dog(Collar collar, int size) {

theCollar = collar;

dogSize = size;

}

public Collar getCollar() { return theCollar; }

**private void writeObject(ObjectOutputStream os)** {

// throws IOException { // 1

try {

**os.defaultWriteObject();** // 2

os.writeInt(theCollar.getCollarSize()); // 3

} catch (Exception e) { e.printStackTrace(); }

}

**private void readObject(ObjectInputStream is) {**

// throws IOException, ClassNotFoundException { // 4

try {

**is.defaultReadObject();** // 5

theCollar = new Collar(is.readInt()); // 6

} catch (Exception e) { e.printStackTrace(); }

}

}

why wouldn't *all* Java classes be serializable?

Why isn't class Object serializable? There are some things in Java that simply cannot

be serialized because they are runtime specific. Things like streams, threads, runtime,

etc. and even some GUI classes (which are connected to the underlying OS) cannot

be serialized.

***If a superclass is Serializable, then according to normal Java interface***

***rules, all subclasses of that class automatically implement Serializable implicitly.***

when an object

is constructed using new (as opposed to being deserialized), the following things

happen (in this order):

1. All instance variables are assigned default values.

2. The constructor is invoked, which immediately invokes the superclass

constructor (or another overloaded constructor, until one of the overloaded

constructors invokes the superclass constructor).

3. All superclass constructors complete.

4. Instance variables that are initialized as part of their declaration are assigned

their initial value (as opposed to the default values they're given prior to

the superclass constructors completing).

5. The constructor completes.

when an object is deserialized we do NOT want any

of the normal initialization to happen. We don't want the constructor to run, and

we don't want the explicitly declared values to be assigned. We want only the values

saved as part of the serialized state of the object to be reassigned.

class Foo implements Serializable {

int num = 3;

void changeNum() { num = 10; }

}

Obviously if you serialize a Foo instance *after* the changeNum() method runs,

the value of the num variable should be 10. When the Foo instance is deserialized,

you want the num variable to still be 10! You obviously don't want the initialization

(in this case, the assignment of the value 3 to the variable num) to happen.

Of course if you have variables marked transient, they will not be restored to

their original state (unless you implement readObject()), but will instead be given

the default value for that data type. In other words, even if you say

class Bar implements Serializable {

transient int x = 42;

}

when the Bar instance is deserialized, the variable x will be set to a value of 0.

Object references marked transient will always be reset to null, regardless of

whether they were initialized at the time of declaration in the class.

If you are a serializable class, but your superclass is NOT serializable, then any

instance variables you INHERIT from that superclass will be reset to the values they

were given during the original construction of the object. This is because the nonserializable

class constructor WILL run!

import java.io.\*;

class SuperNotSerial {

public static void main(String [] args) {

Dog d = new Dog(35, "Fido");

System.out.println("before: " + d.name + " "

+ d.weight);

try {

FileOutputStream fs = new FileOutputStream("testSer.ser");

ObjectOutputStream os = new ObjectOutputStream(fs);

os.writeObject(d);

os.close();

} catch (Exception e) { e.printStackTrace(); }

try {

FileInputStream fis = new FileInputStream("testSer.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

d = (Dog) ois.readObject();

ois.close();

} catch (Exception e) { e.printStackTrace(); }

System.out.println("after: " + d.name + " "

+ d.weight);

}

}

class Dog extends Animal implements Serializable {

String name;

Dog(int w, String n) {

weight = w; // inherited

name = n; // not inherited

}

}

class Animal { // not serializable !

int weight = 42;

}

which produces the output:

before: Fido 35

after: Fido 42

The key here is that because Animal is not serializable, when the Dog was

deserialized, the Animal constructor ran and reset the Dog's inherited weight variable.

***If you serialize a collection or an array, every element must be***

***serializable! A single non-serializable element will cause serialization to fail. Note also***

***that while the collection interfaces are not serializable, the concrete collection classes in***

***the Java API are.***

**Serialization Is Not for Statics**

Static variables are purely CLASS variables.

Static variables cannot be serialized.

**Dates**

1. **java.util.Date:**

* Most of this class's methods have been deprecated
* use this class to bridge between the Calendar and DateFormat class.

1. **java.util.Calendar:**

* Convert and manipulate dates and times. For instance, if you want to add a month to a given date or find out what day of the week January 1, 3000 falls on,

1. **java.text.DateFormat:**

* to format dates not only providing various styles such as "01/01/70" or "January 1, 1970," but also to format dates for numerous locales around the world

1. **java.text.NumberFormat:**

* to format numbers and currencies for locales around the world

1. **java.util.Locale**

* With the help of the Locale class you'll be able to convert a date like "10/10/2005" to "Segunda-feira, 10 de Outubro de 2005" in no time.

if you want to do date formatting for a specific locale, you need to create

your Locale object before your DateFormat object, because you'll need your Locale

object as an argument to your DateFormat factory method.

**Use Case Steps**

Get the current date and time.

1. Create a Date: Date d = new Date();

2. Get its value: String s = d.toString();

Get an object that lets you perform date and time calculations in your locale.

1. Create a Calendar:

Calendar c = Calendar.getInstance();

2. Use c.add(...) and c.roll(...) to perform date and time manipulations.

Get an object that lets u perform date and time calculations in a different locale.

1. Create a Locale:

Locale loc = new Locale(language); or

Locale loc = new Locale(language, country);

1. Create a Calendar for that locale:

Calendar c = Calendar.getInstance(loc);

3. Use c.add(...) and c.roll(...) to perform date and time manipulations.

Get an object that lets you perform date and time calculations, and then format it for output in different locales with different date styles.

1. Create a Calendar:

Calendar c = Calendar.getInstance();

2. Create a Locale for each location:

Locale loc = new Locale(...);

3. Convert your Calendar to a Date:

Date d = c.getTime();

4. Create a DateFormat for each Locale:

DateFormat df = DateFormat.getDateInstance

(style, loc);

5. Use the format() method to create formatted dates:

String s = df.format(d);

Get an object that lets you format numbers or currencies across many different locales.

1. Create a Locale for each location:

Locale loc = new Locale(...);

2. Create a NumberFormat:

NumberFormat nf = NumberFormat.getInstance(loc);

-or- NumberFormat nf = NumberFormat.getCurrencyInstance(loc);

3. Use the format() method to create formatted output:

String s = nf.format(someNumber);

the date and time is stored as a primitive long.

Specifically, the long holds the number of milliseconds (you know, 1000 of these per

second), between the date being represented and January 1, 1970.

**package** com.test;

**import** java.util.Date;

**public** **class** Test9 {

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

Date date = **new** Date(1000000000000L);

System.*out*.println(date.toString());

}

}

Sun Sep 09 07:16:40 IST 2001

**package** com.test;

**import** java.util.Date;

**public** **class** Test9 {

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

Date d1 = **new** Date(1000000000000L);

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

d1.setTime(d1.getTime() + 3600000); // 3600000 millis / hour

System.*out*.println("new time " + d1.toString());

}

}

1st date Sun Sep 09 07:16:40 IST 2001

new time Sun Sep 09 08:16:40 IST 2001

Date now = new Date();

now.getTime();

you'll get a number somewhere

between one trillion and two trillion.

**Calendar Class**

The Calendar class is designed to make date manipulation easy.

Calendar class is an abstract class.

In order to create a Calendar instance, you have to use one of the overloaded

getInstance() static factory methods:

Calendar cal = Calendar.getInstance();

your Calendar reference

variable is actually referring to an instance of a concrete subclass of Calendar. You

can't know for sure what subclass you'll get (java.util.GregorianCalendar is what you'll almost certainly get), but it won't matter to you.

7. Generics and

Collections

**Methods of Class Objectd D**

**escription**

public boolean equals (Object obj) Decides whether two objects are meaningfully equivalent.

void finalize() Called by garbage collector when the garbage collector sees that

the object cannot be referenced.

public int hashCode() Returns a hashcode int value for an object, so that the object can

be used in Collection classes that use hashing, including Hashtable,

HashMap, and HashSet.

final void notify() Wakes up a thread that is waiting for this object’s lock.

final void notifyAll() Wakes up *all* threads that are waiting for this object’s lock.

final void wait() Causes the current thread to wait until another thread calls

notify() or notifyAll() on this object.

public String toString() Returns a “text representation” of the object.

**The toString() Method:**

Override toString() when you want to read something meaningful about the objects of your class.

Code can call toString() on your object

When you pass an object reference to the System.out.println() method, the object's toString() method is called

**when you don't override the toString()**

public class HardToRead {

public static void main (String [] args) {

HardToRead h = new HardToRead();

System.out.println(h);

}

}

Running the HardToRead class gives us the lovely and meaningful,

% java HardToRead

HardToRead@a47e0

The preceding output is what you get when you don't override the toString()

method of class Object. It gives you the class name (at least that's meaningful)

followed by the @ symbol, followed by the unsigned hexadecimal representation of

the object's hashcode.

**when you override the toString()**

public class BobTest {

public static void main (String[] args) {

Bob f = new Bob("GoBobGo", 19);

System.out.println(f);

}

}

class Bob {

int shoeSize;

String nickName;

Bob(String nickName, int shoeSize) {

this.shoeSize = shoeSize;

this.nickName = nickName;

}

public String toString() {

return ("I am a Bob, but you can call me " + nickName +

". My shoe size is " + shoeSize);

}

}

This ought to be a bit more readable:

% java BobTest

I am a Bob, but you can call me GoBobGo. My shoe size is 19

**Overriding equals()**

comparing two object references using the == operator evaluates to true only when both references refer to the same object

(because == simply looks at the bits in the variable, and they're either identical or

they're not).

String class and the wrapper classes have overridden

the equals() method (inherited from class Object), so that you could compare

two different objects (of the same type) to see if their contents are meaningfully

equivalent.

**Difference between == and equals()**

When you really need to know if two references are identical, use ==. But when

you need to know if the objects themselves (not the references) are equal, use the

equals() method.

----------

For some classes, you might decide that two objects can never be equal. So in this case you might not ever need, or want, to override the equals() method.

if you don't override a class's equals()

method, you won't be able to use those objects as a key in a hashtable and you

probably won't get accurate Sets, such that there are no conceptual duplicates.

The equals() method in class Object uses only the == operator for comparisons,

so unless you override equals(), two objects are considered equal only if the two

references refer to the same object.

The bottom line is this: if you want objects of your class to be used as keys for a hashtable, then you must override equals() so that two different instances can be considered the same.

The String and wrapper classes work well as keys in hashtables—they override the equals() method.

So rather than using the actual

car instance as the key into the car/owner pair, you could simply use a String that

represents the unique identifier for the car.

**Rules to override the equals() method**

1. public boolean equals(Object o) {

2. if ((o instanceof Moof) && (((Moof)o).getMoofValue()

== this.moofValue)) {

3. return true;

4. } else {

5. return false;

6. }

7. }

in line 1 we are

indeed declaring a valid override of the equals() method we inherited from Object.

Line 2 is where all the action is.

First, be sure that the object being tested is of the correct type! so you need to do an instanceof test on it. The instanceof test just to be sure that you could cast the object argument to the

correct type, if the object doesn't pass the instanceof test, then you'll get a runtime ClassCastException.

Without the cast, you can't compile because the compiler would see the object referenced by o as simply, well, an Object. And since the Object class doesn't have a getMoofValue() method,

the compiler would squawk (technical term).

reason to appreciate the short circuit && operator—if the instanceof test

fails, we'll never get to the code that does the cast, so we're always safe at runtime

with the following:

if ((o instanceof Moof) && (((Moof)o).getMoofValue()

== this.moofValue)) {

return true;

} else {

return false;

}

**Contract specified in the equals() method**

A Java contract is a set of rules that should be followed, otherwise won’t get correct implementation.

If you don't follow the contract, your code may still compile and run, but your code (or

someone else's) may break at runtime in some unexpected way.

■ It is **reflexive**. For any reference value x, x.equals(x) should return true.

■ It is **symmetric**. For any reference values x and y, x.equals(y) should

return true if and only if y.equals(x) returns true.

■ It is **transitive**. For any reference values x, y, and z, if x.equals(y) returns

true and y.equals(z) returns true, then x.equals(z) must return true.

■ It is **consistent**. For any reference values x and y, multiple invocations of

x.equals(y) consistently return true or consistently return false, provided

no information used in equals comparisons on the object is modified.

■ For any non-null reference value x, x.equals(null) should return false.

rule of thumb should be, if you override equals(), override hashCode() as well.

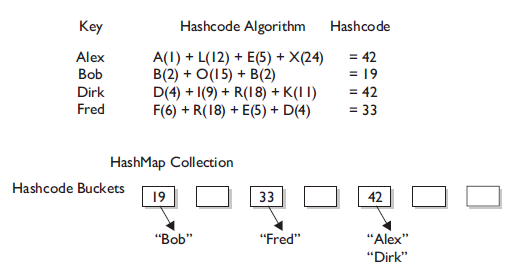
**Overriding hashCode()**

Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be *stored* in the collection, and the hashcode is used again to help *locate* the object in the collection.

***Hashing retrieval is a two-step process.***

***1. Find the right bucket (using*** hashCode()***)***

***2. Search the bucket for the right element (using*** equals() ***).***

****

So for efficiency, your goal is to have the data distributed as evenly as possible across all buckets. Ideally, you might have just one data per bucket so that whensomeone asked for a data you could simply calculate the hashcode and just grab theone data from the correct bucket.

The least efficient (but still functional) hashcode generator would return the same hashcode

(say, 42) regardless of the name, so that all the papers landed in the same bucket while the others stood empty.

When you put an object in a collection that uses hashcodes, the

collection uses the hashcode of the object to decide in which bucket/slot the object should land. Then when you want to fetch that object (or, for a hashtable, retrieve

the associated value for that object), you have to give the collection a reference to

an object that the collection compares to the objects it holds in the collection. As

long as the object (stored in the collection, like a paper in the bucket) you're trying

to search for has the same hashcode as the object you're using for the search (the

name you show to the person working the buckets), then the object will be found.

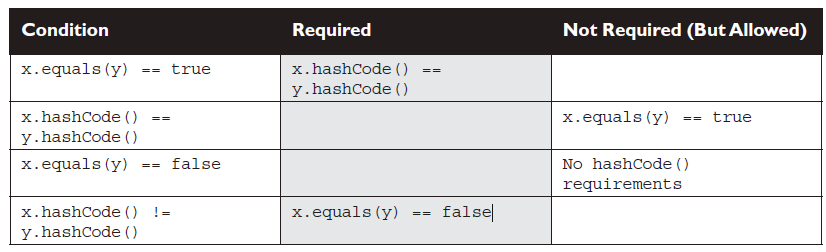
the default hashcode method in class Object virtually always comes up with a unique number

for each object, even if the equals() method is overridden in such a way that two

or more objects are considered equal. It doesn't matter how equal the objects are if

their hashcodes don't reflect that. So one more time: If two objects are equal, their

hashcodes must be equal as well.

****

**What happens if you include a transient variable in your hashCode() method?**

class SaveMe implements Serializable{

transient int x;

int y;

SaveMe(int xVal, int yVal) {

x = xVal;

y = yVal;

}

public int hashCode() {

return (x ^ y); // Legal, but not correct to

// use a transient variable

}

public boolean equals(Object o) {

SaveMe test = (SaveMe)o;

if (test.y == y && test.x == x) { // Legal, not correct

return true;

} else {

return false;

}

}

}

The object in the collection and the supposedly same object brought

back to life are no longer identical after serializing and deserializing. The object's transient variable will come back with a default value rather than the value the variable had at the time it

was saved (or put into the HashMap). So the new hashcode calculation will give a different hashcode, and the equals() method fails as well since x is used to determine object equality.

**Bottom line: Don't use transient variables to determine hashcodes or equality.**

**Collections:**

There are a few basic operations you'll normally use with collections:

■ Add objects to the collection.

■ Remove objects from the collection.

■ Find out if an object (or group of objects) is in the collection.

■ Retrieve an object from the collection (without removing it).

■ Iterate through the collection, looking at each element (object) one

after another.

which collection to choose based on a stated requirement.

The core interfaces you need to know for the

exam (and life in general) are the following nine:

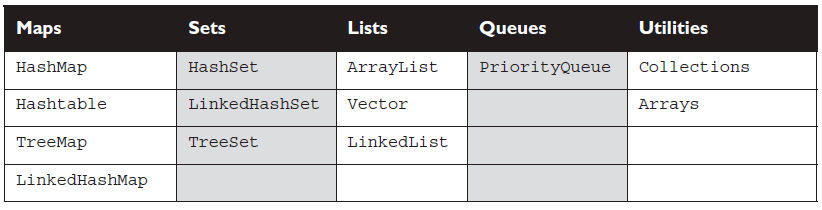
Collection Set SortedSet

List Map SortedMap

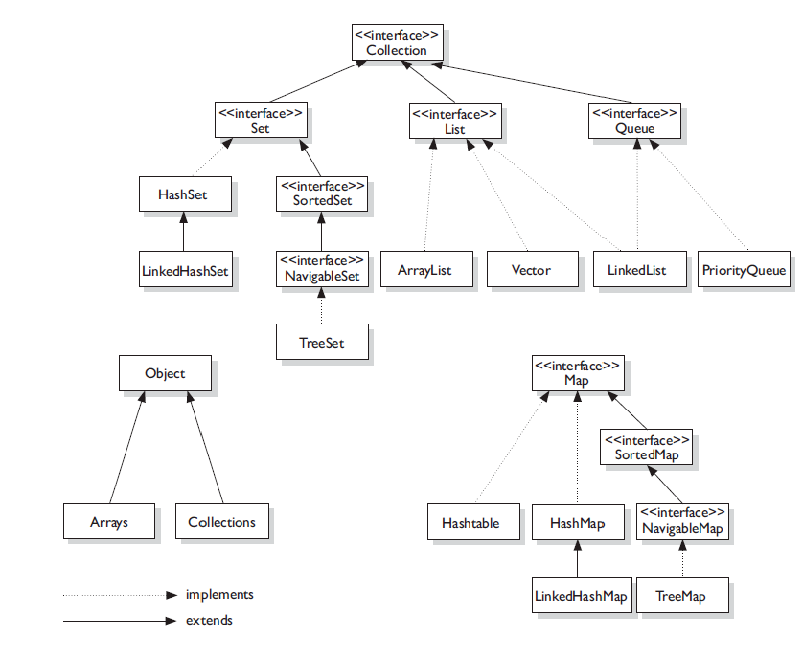
Queue NavigableSet NavigableMap

The core concrete implementation classes you need to know for the exam are the

following 13 (there are others, but the exam doesn't specifically cover them):



None of the Map-related classes and interfaces extend from Collection.



***Collections is a class, with static utility methods, while Collection***

***is an interface with declarations of the methods common to most collections***

***including*** add()***,*** remove()***,*** contains()***,*** size()***, and*** iterator()***.***

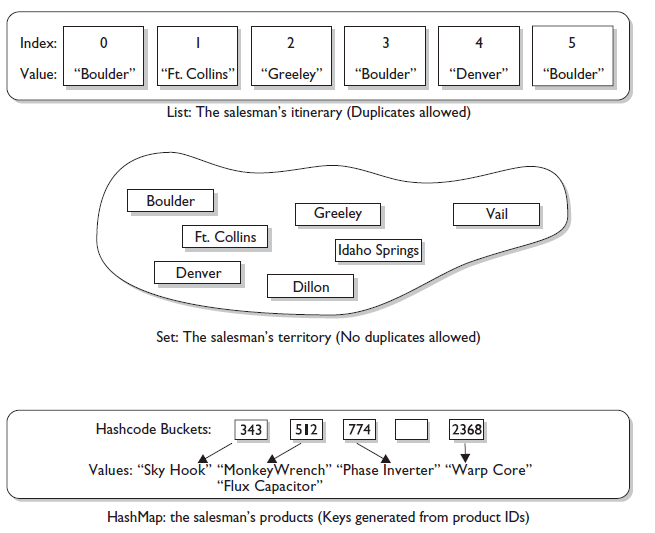
Collections come in four basic flavors:

■ **Lists** *Lists* of things (classes that implement List).

■ **Sets** *Unique* things (classes that implement Set).

■ **Maps** Things with a *unique* ID (classes that implement Map).

■ **Queues** Things arranged by the order in which they are to be processed.



An implementation class can be unsorted and unordered, ordered but unsorted, or

both ordered and sorted. But an implementation can never be sorted but unordered,

because sorting is a specific type of ordering, as you'll see in a moment.

**Ordered** When a collection is ordered, it means you can iterate through the

collection in a specific (not-random) order.

Hashtable collection is not ordered.

ArrayList, however, keeps the order established by the elements' index position (just like an array).

LinkedHashSet keeps the order established by insertion, so the last element inserted is the last element in the LinkedHashSet (as opposed to an ArrayList, where you can insert an element at

a specific index position).

**Sorted** A *sorted* collection means that the order in the collection is determined

according to some rule or rules, known as the sort order.

Most commonly, the sort order used is something called the *natural* order.

For a collection of String objects, then, the natural order is alphabetical. For

Integer objects, the natural order is by numeric value—1 before 2, and so on. And

for Foo objects, the natural order is…um…we don't know. There is no natural

order for Foo unless or until the Foo developer provides one, through an interface

(*Comparable*)that defines how instances of a class can be compared to one another

* **Difference between index ordering and insertion ordering – my question**

All three List implementations are ordered by index position—a position that

you determine either by setting an object at a specific index or by adding it without

specifying position, in which case the object is added to the end.

non-lists don't have

is a set of methods related to the index. Those key methods include things like

get(int index), indexOf(Object o), add(int index, Object obj)

**ArrayList** Think of this as a growable array. It gives you fast iteration and fast random access.

an ordered collection (by index), but not sorted.

ArrayList now implements

the new RandomAccess interface—a marker interface (meaning it has no methods)

that says, "this list supports fast (generally constant time) random access."

Choose

this over a LinkedList when you need fast iteration but aren't as likely to be doing a

lot of insertion and deletion.

**Vector**

Vector is basically the same as an ArrayList, but Vector methods are synchronized

for thread safety.

You'll normally want to use ArrayList instead of Vector

because the synchronized methods add a performance hit you might not need.

And

if you do need thread safety, there are utility methods in class Collections that can

help. Vector is the only class other than ArrayList to implement RandomAccess.

**LinkedList** A LinkedList is ordered by index position, like ArrayList, except

that the elements are doubly-linked to one another.

a LinkedList may iterate more slowly than an ArrayList,

but it's a good choice when you need fast insertion and deletion.

As of Java 5, the

LinkedList class has been enhanced to implement the java.util.Queue interface. As

such, it now supports the common queue methods: peek(), poll(), and offer().

**Set Interface**

A Set cares about uniqueness—it doesn't allow duplicates.

equals() method determines whether two objects are identical

**HashSet** A HashSet is an unsorted, unordered Set.

It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get.

Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

**LinkedHashSet** A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

Use this class instead of HashSet when you care about the iteration order.

When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

**TreeSet**

The TreeSet is one of two sorted collections (the other being TreeMap).

Optionally, you can

construct a TreeSet with a constructor that lets you give the collection your own

rules for what the order should be (rather than relying on the ordering defined by

the elements' class) by using a Comparable or Comparator. As of Java 6, TreeSet

implements NavigableSet.

**Map Interface**

A Map cares about unique identifiers. You map a unique key (the ID) to a specific

value, where both the key and the value are, of course, objects.

Maps rely on the equals() method to determine whether

two keys are the same or different.

**HashMap** The HashMap gives you an unsorted, unordered Map.

When you

need a Map and you don't care about the order (when you iterate through it), then

HashMap is the way to go;

the

keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient

your hashCode() implementation, the better access performance you'll get.

HashMap allows one null key and multiple null values in a collection.

**Hashtable**

Hashtable is the synchronized counterpart to HashMap.

while HashMap lets you have null values as well as one null key, a

Hashtable doesn't let you have anything that's null.

**LinkedHashMap**

LinkedHashMap collection maintains insertion order (or, optionally, access order).

Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

**TreeMap**

TreeMap is a sorted Map.

sorted by the natural order of the elements.

TreeMap lets you define a custom sort order (via a

Comparable or Comparator) when you construct a TreeMap, that specifies how the

elements should be compared to one another when they're being ordered. As of

Java 6, TreeMap implements NavigableMap.

**Queue Interface**

Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

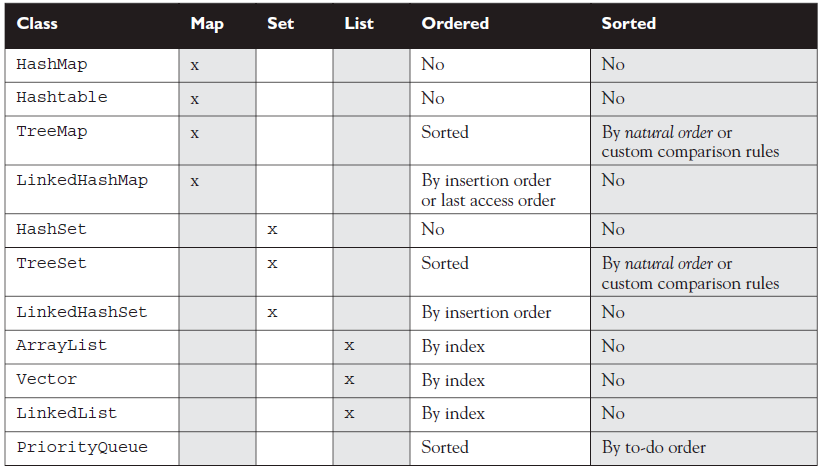
**PriorityQueue**

A PriorityQueue's elements are ordered

either by natural ordering (in which case the elements that are sorted first will be

accessed first) or according to a Comparator. In either case, the elements' ordering

represents their relative priority.



**ArrayList Basics**

Some of the advantages ArrayList has over arrays are

■ It can grow dynamically.

■ It provides more powerful insertion and search mechanisms than arrays.

you'll typically want to instantiate an ArrayList polymorphically like this:

List myList = new ArrayList();

As of Java 5 you'll want to say

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

This kind of declaration follows the object oriented programming principle of

"coding to an interface", and it makes use of generics.

In many ways, ArrayList<String> is similar to a String[] in that it declares a

container that can hold only Strings, but it's more powerful than a String[].

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

String s = "hi";

test.add("string");

test.add(s);

test.add(s+s);

System.out.println(test.size());

System.out.println(test.contains(42));

System.out.println(test.contains("hihi"));

test.remove("hi");

System.out.println(test.size());

which produces

3

false

true

2

**In general, collections can hold Objects but not primitives.**

Prior to Java 5, you had to wrap a primitive by hand before you could put

it into a collection. With Java 5, primitives still have to be wrapped, but autoboxing

takes care of it for you.

List myInts = new ArrayList(); // pre Java 5 declaration

myInts.add(new Integer(42)); // had to wrap an int

As of Java 5 we can say

myInts.add(42); // autoboxing handles it!

**Sorting Arrays and Collections**

Both collections and arrays can be sorted and searched using methods in the API.

**Sorting Collections**

ArrayList doesn't give you any way to sort its contents,

but the java.util.Collections class does

import java.util.\*;

class TestSort1 {

public static void main(String[] args) {

**ArrayList<String> stuff = new ArrayList<String>(); // #1**

stuff.add("Denver");

stuff.add("Boulder");

stuff.add("Vail");

stuff.add("Aspen");

stuff.add("Telluride");

System.out.println("unsorted " + stuff);

**Collections.sort(stuff); // #2**

System.out.println("sorted " + stuff);

}

}

This produces something like this:

unsorted [Denver, Boulder, Vail, Aspen, Telluride]

sorted [Aspen, Boulder, Denver, Telluride, Vail]

----

ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();

populateList(); // adds the file data to the ArrayList

System.out.println(dvdList);

Collections.sort(dvdlist);

Oops!, you get something like this:

TestDVD.java:13: cannot find symbol

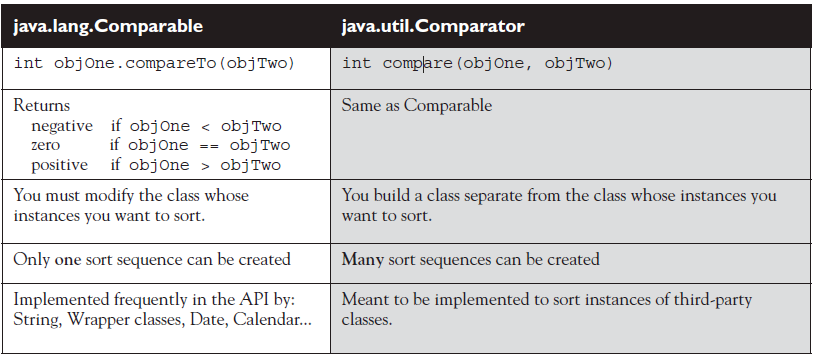
symbol : method sort(java.util.ArrayList<DVDInfo>)

location: class java.util.Collections

Collections.sort(dvdlist);

**Use of Comparator and Comparable interface**

Comparing Comparable to Comparator



**Sorting with the Arrays Class**

using the java.util.Arrays class to sort arrays.

■ Arrays.sort(arrayToSort)

■ Arrays.sort(arrayToSort, Comparator)

sort() methods for both the Collections class and

the Arrays class are static methods, and that they alter the objects they are sorting,

instead of returning a different sorted object.

***whenever you want to sort an array***

***or a collection, the elements inside must all be*** *mutually comparable.* ***In other words, if you***

***have an*** *Object[]* ***and you put*** *Cat* ***and*** *Dog* ***objects into it, you won’t be able to sort***

***it. In general, objects of different types should be considered NOT mutually comparable,***

***unless specifi cally stated otherwise.***

Collections class and the Arrays class both provide methods that allow you to search for a specific element.

When searching through collections or arrays, the following rules apply:

■ Searches are performed using the binarySearch() method.

■ Successful searches return the int index of the element being searched.

■ Unsuccessful searches return an int index that represents the *insertion point*.

The insertion point is the place in the collection/array where the element

would be inserted to keep the collection/array properly sorted.

tive return values and 0 indicate successful searches, the binarySearch()

method uses negative numbers to indicate insertion points. Since 0 is a valid

result for a successful search, the first available insertion point is -1. Therefore,

the actual insertion point is represented as (-(insertion point) -1). For

instance, if the insertion point of a search is at element 2, the actual insertion

point returned will be -3.

■ The collection/array being searched must be sorted before you can search it.

■ If you attempt to search an array or collection that has not already been

sorted, the results of the search will not be predictable.

■ If the collection/array you want to search was sorted in natural order, it *must*

be searched in natural order. (Usually this is accomplished by NOT sending

a Comparator as an argument to the binarySearch() method.)

■ If the collection/array you want to search was sorted using a Comparator, it

*must* be searched using the same Comparator, which is passed as the second

argument to the binarySearch() method. Remember that Comparators

cannot be used when searching arrays of primitives.

Let's take a look at a code sample that exercises the binarySearch() method:

import java.util.\*;

class SearchObjArray {

public static void main(String [] args) {

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

Arrays.sort(sa); // #1

for(String s : sa)

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

System.out.println("\none = "

+ Arrays.binarySearch(sa,"one")); // #2

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

ReSortComparator rs = new ReSortComparator(); // #3

Arrays.sort(sa,rs);

for(String s : sa)

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

System.out.println("\none = "

+ Arrays.binarySearch(sa,"one")); // #4

System.out.println("one = "

+ Arrays.binarySearch(sa,"one",rs)); // #5

}

static class ReSortComparator

implements Comparator<String> { // #6

public int compare(String a, String b) {

return b.compareTo(a); // #7

}

}

}

which produces something like this:

four one three two

one = 1

now reverse sort

two three one four

one = -1

one = 2

Here's what happened:

Line 1 Sort the sa array, alphabetically (the natural order).

Line 2 Search for the location of element "one", which is 1.

Line 3 Make a Comparator instance. On the next line we re-sort the array using

the Comparator.

Line 4 Attempt to search the array. We didn't pass the binarySearch()

method the Comparator we used to sort the array, so we got an incorrect

(undefined) answer.

Line 5 Search again, passing the Comparator to binarySearch(). This time

we get the correct answer, 2

Line 6 We define the Comparator; it's okay for this to be an inner class.

Line 7 By switching the use of the arguments in the invocation of

compareTo(), we get an inverted sort.

**Converting Arrays to Lists to Arrays**

Iterator methods

■ **boolean hasNext()** Returns true if there is at least one more element in

the collection being traversed. Invoking hasNext() does NOT move you to

the next element of the collection.

■ **Object next()** This method returns the next object in the collection,

AND moves you forward to the element after the element just returned.

import java.util.\*;

class Dog {

public String name;

Dog(String n) { name = n; }

}

class ItTest {

public static void main(String[] args) {

List<Dog> d = new ArrayList<Dog>();

Dog dog = new Dog("aiko");

d.add(dog);

d.add(new Dog("clover"));

d.add(new Dog("magnolia"));

**Iterator<Dog> i3 = d.iterator(); // make an iterator**

while (i3.hasNext()) {

**Dog d2 = i3.next(); // cast not required**

System.out.println(d2.name);

}

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

System.out.println("get1 " + d.get(1).name);

System.out.println("aiko " + d.indexOf(dog));

d.remove(2);

Object[] oa = d.toArray();

for(Object o : oa) {

Dog d2 = (Dog)o;

System.out.println("oa " + d2.name);

}

}

}

This produces

aiko

clover

magnolia

size 3

get1 clover

aiko 0

oa aiko

oa clover

declared the Iterator like this:

Iterator i3 = d.iterator(); // make an iterator

But then we would have had to cast the returned value:

Dog d2 = (Dog)i3.next();

**Using Sets**

Sets are used when you don't want any duplicates in your collection.

If you attempt to add an element to a set that already exists in the set, the duplicate

element will not be added, and the add() method will return false.

HashSets tend to be very fast because, as we discussed earlier, they use hashcodes.

import java.util.\*;

class SetTest {

public static void main(String[] args) {

boolean[] ba = new boolean[5];

**// insert code here**

ba[0] = s.add("a");

ba[1] = s.add(new Integer(42));

ba[2] = s.add("b");

ba[3] = s.add("a");

ba[4] = s.add(new Object());

for(int x=0; x<ba.length; x++)

System.out.print(ba[x] + " ");

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

for(Object o : s)

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

}

}

If you insert the following line of code you'll get output something like this:

Set s = new HashSet(); // insert this code

true true true false true

a java.lang.Object@e09713 42 b

The order of objects printed in the second for loop is not predictable: HashSets do not guarantee any ordering. Also, notice that the fourth invocation of add() failed, because it attempted to insert a duplicate entry (a String with the value a) into the Set.

If you insert this line of code you'll get something like this:

Set s = new TreeSet(); // insert this code

Exception in thread "main" java.lang.ClassCastException: java.

lang.String

at java.lang.Integer.compareTo(Integer.java:35)

at java.util.TreeMap.compare(TreeMap.java:1093)

at java.util.TreeMap.put(TreeMap.java:465)

at java.util.TreeSet.add(TreeSet.java:210)

The issue is that whenever you want a collection to be sorted, its elements must

be mutually comparable. Remember that unless otherwise specified, objects of

different types are not mutually comparable.

**Using Maps**

any classes that you

use as a part of the keys for that map must override the hashCode() and equals()

methods.

User Defined Exception

**package** com.crimsonlogic.mnis.cpdv2.account.validator;

/\*\*

\* **@author** surajkumar

\*

\*/

**public** **class** AccountsException **extends** RuntimeException {

**private** **static** **final** **long** *serialVersionUID* = 4738942954095992162L;

**public** AccountsException() {

**super**();

}

**public** AccountsException(**final** String message, **final** Throwable cause) {

**super**(message, cause);

}

**public** AccountsException(**final** String message) {

**super**(message);

}

**public** AccountsException(**final** Throwable cause) {

**super**(cause);

}

}

**if** (stringIsNullOrEmpty(user.getUsername())) {

**throw** **new** AccountsException("UsernameRequired");

}

**if** (stringLengthIsInvalid(user.getUsername(), 7, 20)) {

**throw** **new** AccountsException("UsernameInvalidLength");

}

**if** (!(Pattern.*matches*("[\\p{Alnum}-\_]\*", user.getUsername()))) {

**throw** **new** AccountsException("UsernameInvalidCharacters");

}

**try**{

userService.updateUserAtEditUser(user);

**if** (user.getPassword() != **null** && !user.getPassword().equals("")){

//update user in the ldap server.

userService.modifyUserInLdap(user.getUsername(), user.getPassword());

}

getPageFeedbackPanel().info(

getString("feedback.info.success.editUser"));

} **catch**(AccountsException ae){

FeedbackPanel pageFeedbackPanel = getPageFeedbackPanel();

pageFeedbackPanel.error(getString(ae.getMessage()));

}