1

Notes from

Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

http://www.wellho.net/net/whcotnl.html

for details.

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and disctibute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

http://www.wellho.net/course/index.html

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

http://www.wellho.net/course/otc.html

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

http://www.wellho.net
graham@wellho.net technical contact
lisa@wellho.net administration contact

Our full postal address is

404 The Spa
Melksham
Wiltshire
UK SN12 6QL

Phone +44 (0) 1225 708225
Fax +44 (0) 1225 707126

2

Fundamental classes

The java.util package contains many powerful tools for computer scientists, and is an essential element of most Java applications. You'll learn when and how to use stacks, hash tables and many other utility classes. You'll learn how to sort, how to make system calls (and why you shouldn't!), and how to find out more about your environment.

The fundamental packages. 4

Data wrappers. 5

java.lang.Math 9

External low-level calls. 10

Utility objects to hold multiple simple objects 15

The StringTokenizer 20

Collections 22

Sorting 26

Java is a simple language; there are just eight primitive data types:
byte
boolean
char
double
float
int
long
short

So how come you can do so much with Java? Through classes of objects and the methods associated with them – both classes provided as a standard part of the Java Runtime Environment (JRE), and classes provided as extras with the Enterprise Edition or from other bundles or third parties.

2.1 The fundamental packages

java.lang

java.lang contains the classes that are most central to the language itself; it's a broad rather than a deep hierarchy, which means that there's a lot of classes in the package, but they tend to be independent of one another.

Firstly, **java.lang** contains the data wrapper classes – immutable class wrappers around the primitive types which allow you to treat primitives as if they're objects.[1]

Strings and StringBuffers are a part of **java.lang**, as are the Math (and from Java 1.3 StrictMath) classes. All of these classes (except StringBuffer) implement the Comparable interface, which provides sorting and searching capabilities.

System, Runtime and Process classes provide low-level methods and an API for performing low-level system functions and running external processes.

java.lang also includes the Thread class, and Throwables and exception handling; these are major topics (along with Strings and StringBuffers).

java.util

One of the fundamental packages that you'll use in all but the very simplest of Java applications is the java utility package **java.util** has been with us since the beginning of Java; it was one of the eight original packages in Java release 1.0. There have been a number of additions for Java2, and some of those additions have superseded older objects for certain applications.

The utilities in **java.util** provide the more complex programming structures that you'll want to use in any professionally written classes, but aren't in the basic language. The majority of utility classes are concerned with collections,[2] but the package also includes classes for date and time work, resource bundle handling, parsing a string (StringTokenizer) and the Timer API that was introduced at Java 1.3.

Java.util also includes sub-packages **java.util.jar** and **java.util.zip**.

Other fundamental packages

The following packages might also be considered to be fundamental in certain applications:

java.io	Classes and interfaces for input and output
java.math	Packages for arbitrary precision arithmetic (do NOT confuse with **java.lang.Math** ;-))
java.net	Network access and programming

1 great for use in collections

2 objects which can be used to hold groups of objects in different ways to an array, which is the only built in structure for a group of objects

java.text	Formatting of numbers, dates, times, etc
java.security	A number of packages relating to security

2.2 Data wrappers

What's the difference between a primitive and an object? A primitive is a variable into which you can save an information of one of eight pre-defined types directly, whereas an object is held in a variable which is a reference[1] to a piece or a number of pieces of information.

Primitives are accessed directly through basic facilities provided in Java, whereas objects are accessed through the object interface. Here's a teaching example that copies and compares primitives and objects, showing you how their behaviour differs:

```
public class Obj_v_prim {

// Object v Primitive - a comparison example

public static void main (String [] args) {

   float sample = 16.5f;
   Float Sam2 = new Float(16.5f);

   System.out.println("Primitive is "+sample);
   System.out.println("Object is "+Sam2);

// copy and compare - appreciate the difference!

   float sam_copy = sample;
   Float Sam2_copy = Sam2;

   float another = 16.5f;
   Float Another2 = new Float(16.5f);

   System.out.println("\nDoing == comparisons");

   if (sample == sam_copy) {
               System.out.println("sample and sam_copy are equal");
   } else {
               System.out.println("sample and sam_copy differ");
   }

   if (sample == another) {
               System.out.println("sample and another are equal");
   } else {
               System.out.println("sample and another differ");
   }

   if (Sam2 == Sam2_copy) {
               System.out.println("Sam2 and Sam2_copy are equal");
   } else {
               System.out.println("Sam2 and Sam2_copy differ");
   }

   if (Sam2 == Another2) {
               System.out.println("Sam2 and Another2 are equal");
   } else {
```

1 a pointer or a memory address under the hood

```
            System.out.println("Sam2 and Another2 differ");
    }

    System.out.println("\nDoing \".equals\" comparisons");

    if (Sam2.equals(Sam2_copy)) {
            System.out.println("Sam2 and Sam2_copy are equal");
    } else {
            System.out.println("Sam2 and Sam2_copy differ");
    }

    if (Sam2.equals(Another2)) {
            System.out.println("Sam2 and Another2 are equal");
    } else {
            System.out.println("Sam2 and Another2 differ");
    }

    }
}
```

This test code starts off by defining a primitive **float** and an object of type **Float**:[1]

```
float sample = 16.5f;
Float Sam2 = new Float(16.5f);
```

The text program then prints out the values of the variables:

```
System.out.println("Primitive is "+sample);
System.out.println("Object is "+Sam2);
```

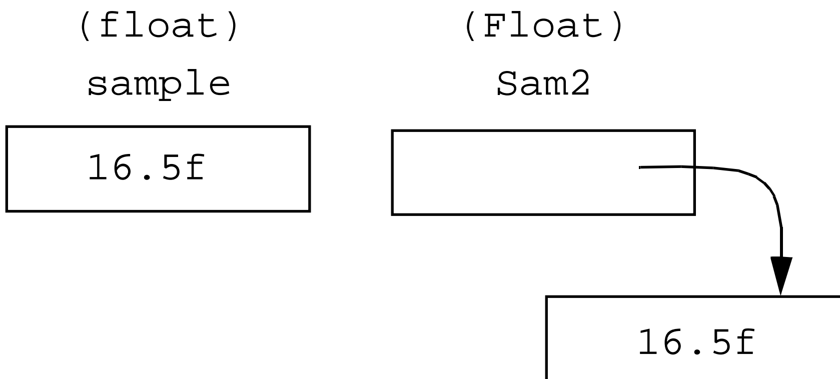


Figure 1 Diagram of our test code

The results they come up with are the same, but you should be aware that's because the **toString** method on class **Float** turns the number that the object contains into a string, and not because the variable themselves are the same.

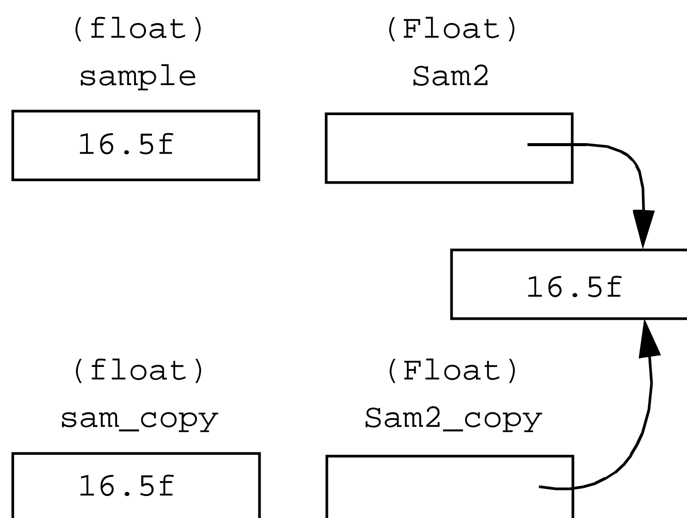
The test code then copies each of the variables.

```
 float sam_copy = sample;
Float Sam2_copy = Sam2;
```

Our result is shown in Figure 2.

[1] note the significance of the capitalisation here!

Figure 2 The test code has copied each of the variables



Notice that we now have the number 16.5 stored three times in memory; twice in the two primitives, but only once in an object as *Sam2_copy* and *Sam2* are both references to the same object.

We then set up two more variables containing 16.5 as well. `another` is a primitive-like sample, and *Another2* is an object like *Sam2*. The capitalisation of the variable names here is just a good convention, whereas it's vital in the type declaration (**`float`** v **`Float`**).

```
float another = 16.5f;
Float Another2 = new Float(16.5f);
```

Our test program then goes on to use "**`==`**" to compare the various variables that we have, and it reports:

```
Doing == comparisons
sample and sam_copy are equal
sample and another are equal
Sam2 and Sam2_copy are equal
Sam2 and Another2 differ
```

With the primitives, all the variables themselves contain the value 16.5 so they're all reported as being equal. With the objects, *Sam2* and *Sam2_copy* contain references to the same **`Float`** object, so report as being equal. *Another2* contains a reference to a different **`Float`** object, and so is reported to differ. It doesn't matter to the **`==`** operator that both of the float objects happen to contain the same number; it only cares that they are different objects.

If you need to compare the contents of "data wrapper" type objects (**`Boolean`**, **`Character`**, **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`** or **`Double`**), use the "equals" method on the object. Although this method compares references for most objects,[1] it's overridden for the data wrappers to compare the contents of the object. Thus:

```
System.out.println("\nDoing \".equals\" comparisons");

if (Sam2.equals(Sam2_copy)) {
          System.out.println("Sam2 and Sam2_copy are equal");
```

[1] that's how it's defined in the Object base class

```
} else {
            System.out.println("Sam2 and Sam2_copy differ");
}

if (Sam2.equals(Another2)) {
            System.out.println("Sam2 and Another2 are equal");
} else {
            System.out.println("Sam2 and Another2 differ");
}
```

yields the results:

```
Doing ".equals" comparisons
Sam2 and Sam2_copy are equal
Sam2 and Another2 are equal
```

Why use data wrappers?

Remember how an array can only hold primitives of one type, but it can hold a whole mixture of objects? Data wrappers give you the ability to hold a whole mixture of information in arrays, and also in other collections as we'll see later in this module. There are also times that it's to your advantage to be able to handle numbers and characters through the more sophisticated object interface rather than as simple primitives.

Microsoft's C# has dispensed with primitives and treats everything as an object. It's an interesting discussion as to whether this is a good idea, or a step too far!

Other methods on Float objects

You can construct a **`Float`** object from a String or from a **`float`** or double primitive. For example:

```
Float V2 = null;
String err = null;
try {
            V2 = new Float(getParameter("temperature"));
} catch (NumberFormatException e) {
            err = "Not a valid numeric format";
} catch (ServletException e) {
            err = "Temperature field not provided";
}
```

You can copy the value of a **`Float`** object into a primitive for further onward use in your program:

```
float tval = V2.floatValue();
if (tval < 20.0f) { .....
```

As from Java 1.2, a number of other methods were added, including a static **`parseFloat`** so the code from the example above could have read:

```
float tval = Float.parseFloat(getParameter("temperature"));
```

(You would still need the code to catch the exceptions, of course.)

There are other methods available too; have a quick glance at the reference material for further details.

Other methods on other data wrapper objects

There are similar methods on other data wrapper objects, but they don't always follow the same exact pattern, and some of their histories vary. Check the manual.

For example, I can write:

```
int tval = Integer.parseInt(getParameter("temperature"));
```

and you'll spot straight away the class in named **`Integer`** [1] but the method is called **`parseInt`** (and not **`parseInteger`**). You won't spot that **`parseInt`** was available as long ago as Java release 1.0, even though **`parseFloat`** only came in at Java 1.2

2.3 java.lang.Math

The Math class defines a number of constants and static methods for trigonometric, exponentation and other mathematical operations. Most of them work on double parameters and return double types. Do note that these methods use the underlying C libraries of the computer that's hosting the Java Runtime Environment and are not implemented in Java, so they're fast but results may not exactly match between platforms.

```
public class Mathop {

public static void main(String [] args) {

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

            try {
            double uservalue = Double.parseDouble(args[i]);
            System.out.println("Input value "+uservalue);

            double uvr = Math.toRadians(uservalue);
            System.out.println("That's "+uvr+" radians");

            double cosine = Math.cos(uvr);
            double primary = Math.acos(cosine);
            System.out.println("Cosine is "+cosine+
                        " which has a primary angle of "+uvr);

            } catch (NumberFormatException e) {
                        System.out.println("Not a number ["+args[i]+"]");

            } finally {
                        System.out.println();
            }
  }
}

}
```

[1] not `Int` as you would expect

```
$ java Mathop -45 0 zero 60 90 200
Input value -45.0
That's -0.7853981633974483 radians
Cosine is 0.7071067811865476 which has a primary angle of -0.7853981633974483

Input value 0.0
That's 0.0 radians
Cosine is 1.0 which has a primary angle of 0.0

Not a number [zero]

Input value 60.0
That's 1.0471975511965976 radians
Cosine is 0.5000000000000001 which has a primary angle of 1.0471975511965976

Input value 90.0
That's 1.5707963267948966 radians
Cosine is 6.123031769111886E-17 which has a primary angle of 1.5707963267948966

Input value 200.0
That's 3.490658503988659 radians
Cosine is -0.9396926207859084 which has a primary angle of 3.490658503988659

$
```

Figure 3 Running public class Mathop

Other methods available include:

ceil next whole number above
floor next whole number below
round nearest whole number (returns an **int** or **long**)
abs number without sign (make negatives positive)
pow raise to power
max maximum of two numbers
min minimum of two numbers
exp exponential
log natural logarithm
sqrt square root
tan tangent

and other trigonometry functions.

Do note that method for secant and cosecant, etc., are not provided in this fundamental class.

2.4 External low-level calls

The System class – miscellaneous features

The System class provides a low-level interface to system facilities and properties. The examples earlier in this module have already used **System.out**, and you also have:

System.err an output print stream for error messages
System.in an Input stream that you'll use for keyboard input

The **getProperty** method lets you get information about the Java Virtual Machine that you're running on, and the underlying operating system.

```
public class About {

public static void main (String [] args) {

  String [] what = {
              "java.home",
              "java.class.path",
              "java.version",
              "java.vendor",
              "java.vm.version",   // Java 1.2, many more like this
              "os.name",
              "os.arch",
              "os.version",
              "file.separator",
              "path.separator",
              "line.separator",
              "user.name",
              "user.home",
              "user.dir" };

  for (int i=0; i<what.length; i++) {
              String current = what[i];
              StringBuffer line = new StringBuffer(current);
              while (line.length() < 20) line.append(" ");
              String result =
                          line.append(System.getProperty(current)).toString();
              System.out.println(
                          (result.replace('\n','\u00ac').replace('\r','\u00ad')));
              }
  }
}
```

On the system on which this module was written, here are the results:

```
$ java About
java.home           /home/javatools/jdk1.3/jre
java.class.path     .:/home/javatools/jdk1.3/lib/tools.jar:/home/javatools/j2sdkee1.3/lib/j2ee.j
home/graham/mysqlx/mm.mysql-2.0.8:/home/javatools/jakarta-regexp-1.2/jakarta-regesxp-1.2.jar
java.version        1.3.0
java.vendor         Sun Microsystems Inc.
java.vm.version     1.3.0
os.name             Linux
os.arch             i386
os.version          2.2.16-22
file.separator      /
path.separator      :
line.separator      ¨
user.name           graham
user.home           /home/graham
user.dir            /home/graham/java2
$
```

Figure 4 Running public class About

Also in System, there's an `exit` method (which takes a single integer parameter – a return status) to exit from a Java Virtual Machine.

The System class – garbage collection

Within the Java Virtual Machine, memory is allocated dynamically; in other words, as objects are created, the memory that is required to hold them is set aside,

and when they are destroyed the memory is released.

That's not the whole story. If Java went around cleaning up every time that a piece of memory was released, it would be grossly inefficient. It would be rather like calling up the garbage collector to come 'round whenever you finished a can of Coke, instead of waiting for his visit once or twice a week to collect garbage in bulk.

Java's "Garbage Collector" is run from time to time, usually at a time decided by the Virtual Machine itself. Not only does it collate all the memory that's free, but it also runs the **finalize** method on any objects which have been released.

Here's a very simple class which we can store things in:

```
public class Thing {

public static int number_of_things = 0;
public String what;

public Thing (String what) {
  this.what = what;
  number_of_things++;
  }

protected void finalize ()  {
  number_of_things--;
  }

}
```

Now, let's create a number of things, throw some of them away, and keep counting how many things we have:

```
public class Handbag {

public static void main (String [] args) {

  Thing [] Ihave = new Thing[args.length];

  for (int i=0; i<args.length; i++) {
              Ihave[i] = new Thing(args[i]);
              }

  System.out.println ("I have "+Thing.number_of_things+
                          " things in my handbag");
  Ihave[0] = null;
  System.out.println ("I now have "+Thing.number_of_things+
                          " things in my handbag");
  System.out.println ("And now there are "+Thing.number_of_things+
                          " things in my handbag");
  Ihave[1] = null;
  System.out.println ("now there are "+Thing.number_of_things+
                          " things in my handbag");

  System.out.println ("and at the end there are "+Thing.number_of_things+
                          " things in my handbag");

}

}
```

We would hope that if we start off with four things in the handbag, that will decrease to three, and then two things, as we drop the first and second things. Alas:

Figure 5 Running public class Handbag

```
$ java Handbag "Credit Cards" keys money pen
I have 4 things in my handbag
I now have 4 things in my handbag
And now there are 4 things in my handbag
now there are 4 things in my handbag
and at the end there are 4 things in my handbag
$
```

We can force finalizations methods to be run using the **runFinalization** method, and we can force garbage collection (with an automatic **runFinalization** first) using the **gc** method of the System class.

```
public class Mypocket {

public static void main (String [] args) {

   Thing [] Ihave = new Thing[args.length];

   for (int i=0; i<args.length; i++) {
               Ihave[i] = new Thing(args[i]);
               }

   System.out.println ("I have "+Thing.number_of_things+
                            " things in my pocket");
   Ihave[0] = null;
   System.out.println ("I now have "+Thing.number_of_things+
                            " things in my pocket");
   System.gc();
   System.out.println ("And now there are "+Thing.number_of_things+
                            " things in my pocket");
   Ihave[1] = null;
   System.out.println ("now there are "+Thing.number_of_things+
                            " things in my pocket");
   System.runFinalization();
   System.gc();   // Odd that I need this - to come back to?
   System.out.println ("and at the end there are "+Thing.number_of_things+
                            " things in my pocket");
}
}
```

Figure 6 Running public class Mypocket

```
$ java Mypocket coin hankie sweet hole
I have 4 things in my pocket
I now have 4 things in my pocket
And now there are 3 things in my pocket
now there are 3 things in my pocket
and at the end there are 2 things in my pocket
$
```

A word of caution here: If you find that you have to use the **gc** and **runFinalization** methods in your program, you may have a poor design. Really, finalize methods are there to flush buffers to disk, etc., and shouldn't contain code on which a continuing application relies. You may find them useful if you're timing

things, or if you have an application in which you want a degree of control over the garbage collector.

If you are timing, the **getTimeMillis()** method returns you the number of milliseconds that have elapsed since 1st January 1970 (as a long), so that you can perform comparisons and get an idea of how things are going. A **gc()** call before you get the timing will help even out your timing tests, as it helps prevent distortion of statistics caused by an occasional garbage collection.

The Runtime and Process classes

The **Runtime** class actually underlies the **System** and **Process** classes, and the only method in it that's called directly is often the **exec** method. The **exec** method starts a new process running externally to the Java Virtual Machine.

Here's an example that attempts to report on how much disk space you have free:

```
import java.io.*;

public class Discfree {

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

   String [] Command = null;

   if (System.getProperty("os.name").equals("Linux")) {
               Command = new String[1];
               Command[0] = "df";
               }
   if (System.getProperty("os.name").equals("Solaris")) {
               Command = new String[2];
               Command[0] = "df";
               Command[1] = "-k";
               }
   if (Command == null) {
               System.out.println("Can't find free space on this OS");
               System.exit(1);
               }

   Process Findspace = Runtime.getRuntime().exec(Command);

   BufferedReader Resultset = new BufferedReader(
                          new InputStreamReader (
                          Findspace.getInputStream()));

   String line;
   while ((line = Resultset.readLine()) != null) {
               System.out.println(line);
               }
   }
}
```

```
$ java Discfree
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda3             11087504   8320380   2203904  80% /
/dev/hda1              3244412   1767104   1477308  55% /c
/dev/hda5              3780112   1060100   2720012  29% /d
$
```

Figure 7 Running public class Discfree on a Linux system

That will work nicely on Solaris too (and other 'nix operating systems should fit in well too) but it would need some modifications under Windows.

A word of caution on system classes

One of the key features of Java is that it's portable. In other words that it will run the same way whatever operating system it's running on, under whatever virtual machine. This feature can be lost to your Java applications if you use the low-level classes described in this section of the notes.

- **`Runtime.exec()`** calls rely on other software that's on the computer on which you are running you Java Virtual Machine. It's no good calling a Linux program such as `df` if you're likely have your code on a Windows system.
- **`System.exit`** (and many other System methods) aren't available in applets or Servlets, and many of the system properties have been disabled in applets to avoid security problems such as the owner of a web site being able to find the user's login name.

2.5 Utility objects to hold multiple simple objects

If you want to hold a number of objects in a single composite object, we call it a collection. The Java language itself (without any additional classes) supports arrays, which can hold primitives or objects. The **`java.util`** package adds a whole further series of classes which can be used to hold multiple objects in various arrangements and with various facilities.

Before we go on to look at these various collection classes, do note one important limitation – they hold multiple OBJECTS; if you want to hold multiple primitives in a collection, you need to use the data type wrappers that we looked at earlier in this module.

The first collection objects we look at – the Vector, Stack and Hashtable (and the enumeration interface) – have been available from the beginnings of Java. A complete Collection framework was added at Java 1.2; we'll look at those after we've had a brief look at the older classes.

Vectors

A vector is a container for a number of objects that can be referenced by their index number (position) within the container. You set up a vector with a constructor (of course), and you can then add and examine elements using methods such as **`addElement`** and **`elementAt`**. The **`size`** method lets you find out how many elements there are in your vector.

```
import java.util.*;

/** Using a vector*/

public class holt
   {
   public static void main(String[] args)
      {
      System.out.println("Shortened pack");
      Vector play = new Vector();
      play.addElement(new suspect_card_9(3));
      for (int k=21;k<53;k++)
         play.addElement (new playing_card_9((k==27)?0:k));

// And print out cards!
     for (int k=0;k<play.size();k++)
          {
```

```
            card_9 current = (card_9) play.elementAt(k);
            System.out.println(current.getcardname());
        }
    }
}
```

We need to import the java.util classes:

```
import java.util.*;
```

We declare a new object of class vector:

```
Vector play = new Vector();
```

We add elements to that vector. As we have not specified where in the vector elements are to be added, each will automatically be added on to the end.

```
play.addElement(new suspect_card_9(3));
for (int k=21;k<53;k++)
   play.addElement (new playing_card_9((k==27)?0:k));
```

We can find out how many elements there are in our vector by using the `size` method:

```
for (int k=0;k<play.size();k++)
```

And we can get specific elements back by using the **`elementAt`** method. Since a vector can hold objects of any class, we must cast our returned element to the correct class.

```
card_9 current = (card_9) play.elementAt(k);
```

There are many more methods available in the vector class.

To replace or insert elements:

`insertElementAt(`*`obj,posn`*`)`	extra element not on end
`setElementAt (`*`obj,posn`*`)`	replacing an element

To retrieve specific elements:

`firstElement()`
`lastElement()`

To remove objects from a vector:

`removeElementAt(`*`posn`*`)`	by position
`removeElement(`*`obj`*`)`	by object; returns flag
`removeAllElements()`	

To search for an object by position:

`indexOf(`*`obj`*`)`

You'll notice that a vector is like a more flexible version of an array, with an object oriented interface and without the restriction on extension. Elements can be removed in from the middle of a vector, and the remaining elements simply move up to fill the gaps. To allow for all these extra facilities, the internal arrangement of the memory of a vector uses a “link list”, so that it can be dynamically expanded[1] and elements removed and added efficiently anywhere. But this means that if you want to step through the data many times, efficiency drops a little.

So you have one further method that lets you copy a vector into an array:

copyInto(*array*)

Stacks

Java stack objects are derived from vectors. In other words, they have vector capabilities plus.

In computing and Java terms, a stack is a pile of objects put onto a pile for future (re)use. Like any pile of objects, any new objects are added to the top (**push**) and it's the top object that is passed back to you when you ask (**pop**).

Stacks are sometimes referred to as “first in, last out” or FILO stores, since the first object pushed onto a stack remains there until all subsequently pushed objects have been popped.

<u>Methods available for stack objects</u>:

push(*obj*)	to stack an object
pop()	to remove the top item and return it
peek()	to return a copy of the top item but not remove it
empty()	true if and only if the stack is empty

Storing our cards on a stack:

```
import java.util.*;

public class outmarsh
   {
   public static void main(String[] args)
      {
      System.out.println("Shortened pack");
      Stack play = new Stack();
      play.push(new suspect_card_9(3));
      for (int k=21;k<53;k++)
         play.push(new playing_card_9((k==27)?0:k));

// And print out cards!

     while (! play.empty())
          {
          card_9 current = (card_9) play.pop();
          System.out.println(current.getcardname());
       }
   }
}
```

Will mean they print in reverse order!

1 you didn't have to give a size, did you?

Hashes

Arrays have elements which are numbered from zero up, as do vectors. Stacks are a sub-class of vectors, so they number in the same way.

What if we want to hold a table of information but instead of having the objects in it numbered 0,1,2,3 we want to use some other key sequence?

For example, we might want to have a table of "Cluedo" suspect cards (in a new class) where the key is the name of the person, and the members are the room in which that person is presently located.

We can use a class called **`Hashtable`** to store information of this type. In a Hashtable, each element has two objects associated with it:

- The key (the moral equivalent of the array subscript 0, 1, 2 etc.)
- The data item (which can be an object of any type, just as in a regular array)

Let's actually set up a hash table for three of the suspects:

```
import java.util.*;

public class hinton {
   public static void main(String[] args)
      {
      Hashtable suspects = new Hashtable();
      suspects.put("White","Kitchen");
      suspects.put("Green","Library");
      suspects.put("Plum","Billiard Room");
```

Objects of any class can be used as the key and can be held in the hashtable. However, if you are using an object of your own class as the key, you need to provide a method called **`hashCode`**.[1]

The first line of executable code here creates a new instance of an object in class **`Hashtable`**, and the three following lines use the `put` method to add data.

You can put as many objects as you like into a hash table.

If you give a new key, a new entry will be added to the table. If you reuse a key, the old object will be replaced.

The `put` method added objects to the Hashtable. The **`get`** method can be used for getting them back out:

```
System.out.print ("Who are you looking for? ");
      String who = WellHouseInput.readLine();

      String where = (String) suspects.get(who);

      if (where == null)
         {
            System.out.println ("Don't know where "+who+" is");
         } else {
            System.out.println ("The person "+who+" is in the "+where);
         }
```

1 See Java documentation

Figure 8 Using the get method

```
$ java hinton
Who are you looking for? White
The person White is in the Kitchen
$ java hinton
Who are you looking for? Yellow
Don't know where Yellow is
$
```

Note the return of null if there is no element matching the key in the hash.

With arrays and the other utility objects, you can use a **size** method to tell you how many entries you have, and then a **for** loop (or similar) to step through them. In the case of Hashtables, though, there is no algorithm which will step through all the keys; you have to use the Enumeration interface.

Let's modify the **hinton.java** class so that an empty string entered lists out all people and where they are located.

```
Enumeration names = suspects.keys();
    while (names.hasMoreElements()) {
    String surname = (String) names.nextElement();
    String where = (String) suspects.get(surname);
    System.out.println(surname+" is in the "+where);
    }
```

Figure 9 Outputting a list if an empty string is entered

```
$ java hinton
Who are you looking for?
Plum is in the Billiard Room
Green is in the Library
White is in the Kitchen
$
```

If you wish to enumerate through the elements rather than the keys, your can use the **elements()** method.

Note that the enumerated elements may be returned to you in any order. You cannot say they are “alphabetic” nor “in the order they were put” nor give any other description.

If you are going to use complete objects within your own classes as keys, you must provide a method **hashCode** and a method **equals**. See the Java documentation for further details.

We'll come back to look at the Collection classes added in Java 2 shortly, but let's have a look at the Enumerations and StringTokenizer first, as these are of great use in setting up and passing through collections.

Enumerations

Used by many of the utility classes as a way of returning each element in turn. Three methods to consider:

- **hasMoreElements**
- **elements**
- **nextElement**

Let's rewrite the printing loop of *holt.java* to use enumeration.

This is what we start with:

```
for (int k=0;k<play.size();k++)
     {
     card_9 current = (card_9) play.elementAt(k);
     System.out.println(current.getcardname());
   }
```

And with enumeration it changes to:

```
Enumeration loop_through = play.elements();

while (loop_through.hasMoreElements())
     {
     card_9 current = (card_9) loop_through.nextElement();
     System.out.println(current.getcardname());
   }
```

The operation does not change.

2.6 The StringTokenizer

There's often a requirement to split down a string into its component elements, or "tokens", such as splitting a sentence into words or a line of data into fields. You could roll your own class and methods, but better to use the **StringTokenizer** class.

The simplest way to use a **StringTokenizer** is to construct one to parse a string (passed as a parameter to the constructor), then use enumeration methods to handle the individual tokens. For example:

```
import java.util.*;
public class Strtok {

public static void main(String [] args) {

  String Demo = "This is a string that we want to tokenize";

  StringTokenizer Tok = new StringTokenizer(Demo);
  int n=0;

  while (Tok.hasMoreElements())
             System.out.println("" + ++n +": "+Tok.nextElement());
  }
}
```

```
$ java Strtok
1: This
2: is
3: a
4: string
5: that
6: we
7: want
8: to
9: tokenize
$
```

Figure 10 Using a StringTokenizer

Alternative constructors allow you to specify a string of characters that can be used as separators, and to choose whether you wish to return the separator itself as a token. The **nextToken** method (similar to **nextElement**) allows you to specify a new delimiter string if you need to.

The **StringTokenizer** class is fundamental to much of the String handling that you'll need to do in Java.

The NCSA standard file format for web server access logs is nasty. Delimiters vary through each access record; some fields are always quoted and others are never quoted.

Here's a sample line from an access log file. Let's extract the URL called, the number of bytes returned to the caller, and the name of the program calling us up (In this example, the visitor is Google's robot; for a human visitor, we would be told which browser and version they are using)

```
crawl5.googlebot.com - - [02/Sep/2001:01:11:36 -0700] "GET /book/0-7357-0949-1.html HTTP/1.0" 200 6081 "
" "Googlebot/2.1 (+http://www.googlebot.com/bot.html)"
```

And the code:

```
import java.util.*;
import java.io.*; // for test program only

public class Access {

// Access Log file (NCSA format) analysis

String Host;
String Time;
String Request;
String Method;
String URL;
int status;
int size = 0;
String Referer;
String UserAgent;

// crawl5.googlebot.com - - [02/Sep/2001:01:11:36 -0700] "GET /book/0-7357-0949-1.html HTTP/1.0" 200 6081
"-" "Googlebot/2.1 (+http://www.googlebot.com/bot.html)"

public Access (String Data) {

  StringTokenizer Splitter = new StringTokenizer(Data," \t");
  String skip;

  Host = Splitter.nextToken();
  skip = Splitter.nextToken();
  skip = Splitter.nextToken("[");
  Time = Splitter.nextToken(" \t");
  skip = Splitter.nextToken("\"");

  Request = Splitter.nextToken();
  skip = Splitter.nextToken(" \t");

  status = Integer.parseInt(Splitter.nextToken(" \t"));

  try {
```

```
              size = Integer.parseInt(Splitter.nextToken(" \t"));
  } catch (Exception e) {
              size = 0;
  }

  skip = Splitter.nextToken("\"");
  Referer = Splitter.nextToken();
  skip = Splitter.nextToken();
  UserAgent = Splitter.nextToken();

  // Use another String Tokenizer on the HTTP Request

  Splitter = new StringTokenizer(Request);
  Method = Splitter.nextToken();
  URL = Splitter.nextToken();

  }

// Test code

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

  BufferedReader Source = new BufferedReader(
                         new FileReader( args[0]));

  String Line = Source.readLine();

  Access Earliest = new Access(Line);

  System.out.println("Access to web page "+Earliest.URL);
  System.out.println("Information returned (in bytes) "+Earliest.size);
  System.out.println("Called by "+Earliest.UserAgent);


  }

}
```

Figure 11 Running an access log file

```
$ java Access accesslog.txt
Access to web page /book/0-7357-0949-1.html
Information returned (in bytes) 6081
Called by Googlebot/2.1 (+http://www.googlebot.com/bot.html)
$
```

2.7 Collections

As from Java 1.2, a whole range of additional collection objects have been added to *java.util* which to some extent supersede the **`Hash`**, **`Vector`** and **`Stack`** classes.

ArrayLists

`ArrayLists` are an alternative to vectors. Let's see an example of one in use to hold a whole series of Access objects of the sort that we've just defined in our **`StringTokenizer`** example...

```
import java.util.*;
import java.io.*;

public class Arlist {

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

   BufferedReader Source = new BufferedReader(
                           new FileReader( args[0]));

   ArrayList Hits = new ArrayList();
   String Line;

   while ((Line = Source.readLine()) != null) {
               Hits.add(new Access(Line));
               }

   int nindex = 0;

   for (int i=0; i<Hits.size(); i++) {
               if ((((Access)Hits.get(i)).URL).equals("/index.html")) nindex++;
               }

   System.out.println("/index.html accessed "+nindex+
               " out of a total of "+Hits.size()+" accesses");

   }
}
```

The **add** method adds an object to the end of the **ArrayList** (with an integer parameter as well, you can insert an element into the middle of a list), and the **get** method retrieves an element. The **size** method tells you the number of elements in the **ArrayList**.

HashSets

A **HashSet** is a collection into which you can place a number of objects, but only one instance of each object is kept. Although a "hashing" technique is used internally, a Hashset is not a key and value pair in the same way that the **HashTable** that we've seen already, nor the **HashMap** that we'll be seeing shortly.

When might I use a **HashSet**? If I wanted to get a list of all the unique hosts that have visited my web site (from the accesslog file) then I can simply store the hosts into a **HashSet** and use an Iterator to report on all the elements. The removal of duplicates has been taken care of for me!

```
import java.util.*;
import java.io.*;

public class Hset {
public static void main(String [] args) throws IOException {

   BufferedReader Source = new BufferedReader(
                           new FileReader( args[0]));

   HashSet Hosts = new HashSet();
   String Line;
```

```
    while ((Line = Source.readLine()) != null) {
                Hosts.add((new Access(Line)).Host);
                }

    Iterator It = Hosts.iterator();
    int n=0;
    while (It.hasNext()) {
                System.out.println("" + ++n +": "+It.next());
                }
    }
}
```

```
$ java Hset accesslog.txt
1: ham.ulcc.wwwcache.ja.net
2: ip5.d-fd.com
3: ezspider305.directhit.com
4: ac8a2a48.ipt.aol.com
5: ppp-4-35.5800-7.access.uk.worldonline.com
6: cache-mtc-al06.proxy.aol.com
7: 216-21-202-114.spectrumdsl.net
8: inetgw.il.neustar.com
9: violet.d48.lilly.com
10: sv.us.ircache.net
(etc)
256: 194.216.208.232
257: wwwcache.lanl.gov
258: wfp2.almaden.ibm.com
259: adsl-64-175-33-48.dsl.pltn13.pacbell.net
260: host217-32-157-185.hg.mdip.bt.net
261: j4017.inktomisearch.com
262: cache1.uwn.unsw.edu.au
$
```

Figure 12 Running a HashSet

The sample data file we used here contained 3,817 accesses, so each host called up more than 10 pages from our web server.[1]

Iterators and general Collection interfaces

There are times that we want to step through all the elements of a collection, be it a **HashSet**, an **ArrayList** (or a **LinkedList** or a **HashMap**...) and we can do so using the **iterator** interface. There's an example of an Iterator in the previous example. Its mandatory methods are

hasNext() which returns a boolean
next() which returns an object

The **List** interface is implemented on ArrayLists, Vectors, and several other classes; we've seen some of the methods it defines in use already. This interface means that we can use common code to handle any conforming collection object. Methods include

add(*Object Toadd*)
add(*int position, Object Toadd*)
addAll(*Collection Toadd*)
clear()
contains(*Object Totest*)
get(*int position*)

[1] sounds impressive until you realise that each image is a separate hit

indexOf(*Object Tofind*)
isEmpty()
remove(*Object Toscrap*)
remove(*int position*)
size()
toArray()

HashMaps

HashMaps are very similar to HashTables, except that they are not synchronized, so they’re faster but may not be thread-safe. It's suggested that you use HashMaps unless you require Java 1.0 or 1.1 compatibility, or you're writing a threaded application where several threads may access the Hash at the same time.

Here's a sample program that uses a HashMap to count the number of accesses from each host computer in the Access log file that we've been using as an example. Note once again that we've had to store an Integer Object in the HashMap, as collection objects can't store **`int`** (or any other) primitives.

```
import java.util.*;
import java.io.*;

public class Hmap {

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

  BufferedReader Source = new BufferedReader(
                          new FileReader( args[0]));

  HashMap HostCount = new HashMap();
  String Line;

  while ((Line = Source.readLine()) != null) {
             String Host = (new Access(Line)).Host;

             int was = 0;
             Object Got;
             if ((Got = HostCount.get(Host)) != null)
                         was = (((Integer)Got).intValue());
             HostCount.put(Host, new Integer(was+1));
  }

  Set HostKeys = HostCount.keySet();
  Iterator It = HostKeys.iterator();
  while (It.hasNext()) {
             String HostNow = (String)(It.next());
             System.out.println(HostNow + " - " + HostCount.get(HostNow));
             }
  }
}
```

```
$ java Hmap accesslog.txt
ham.ulcc.wwwcache.ja.net - 2
ip5.d-fd.com - 10
ezspider305.directhit.com - 9
ac8a2a48.ipt.aol.com - 1
ppp-4-35.5800-7.access.uk.worldonline.com - 25
cache-mtc-al06.proxy.aol.com - 1
216-21-202-114.spectrumdsl.net - 1
inetgw.il.neustar.com - 16
(etc)
hector.access.one.net - 13
nachiket.vsnl.net.in - 1
194.216.208.232 - 11
wwwcache.lanl.gov - 14
wfp2.almaden.ibm.com - 498
adsl-64-175-33-48.dsl.pltn13.pacbell.net - 1
host217-32-157-185.hg.mdip.bt.net - 2
j4017.inktomisearch.com - 1
cache1.uwn.unsw.edu.au - 1
$
```

Figure 13 Running a HashMap

2.8 Sorting

If you try to make practical use of the previous examples, you'll discover that you really want your output sorted. If there are two or three records to be listed out, then the order doesn't really matter, but many more and sorting becomes vital.

Basic sorting in Java

Prior to Java 1.2, if you wanted to sort you had to "roll your own". Not particularly hard, but sometimes long-winded and you would have had a lot of work to do if you wanted to write an efficient sort routine.

In Java 1.2, **Arrays.sort** and **Collections.sort**[1] will sort your composite objects.

Let's sort the hosts from our HashMap example. We've chosen something slightly awkward to sort, as HashMaps can't be sorted, so we've got a Set of keys out from it. Now, a Set object can't be sorted either (for a class to be sort-able, all the elements need to be in memory; alas, that doesn't happen with a Set), so we have to put all the elements into... well, we chose a Vector. At last - something we can sort!

```
import java.util.*;
import java.io.*;

public class Hmapsort {

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

  BufferedReader Source = new BufferedReader(
                       new FileReader( args[0]));

  HashMap HostCount = new HashMap();
  String Line;

  while ((Line = Source.readLine()) != null) {
            String Host = (new Access(Line)).Host;
```

1 those are static methods that can be used on arrays and some utility classes that are collections

```
                        int was = 0;
                        Object Got;
                        if ((Got = HostCount.get(Host)) != null)
                                    was = (((Integer)Got).intValue());
                        HostCount.put(Host, new Integer(was+1));
            }

            Set HostKeys = HostCount.keySet();

            Iterator It = HostKeys.iterator();
            Vector HKeys = new Vector();
            while (It.hasNext()) {
                        HKeys.add((String)(It.next()));
                        }

            Collectionssort(HKeys);

            for (int i=0;i<HKeys.size();i++) {
                        String K = (String)(HKeys.elementAt(i));
                        Integer V = (Integer)(HostCount.get(K));
                        System.out.println(K + " - " + V);
                        }
            }
}
```

Figure 14 Running public class Hmapsort

```
$ java Hmapsort accesslog.txt
130-127-83-233.generic.clemson.edu - 1
132.146.145.128 - 3
141.161.46.81 - 19
144.137.82.201 - 1
146.2wrongs.com - 5
155.136.219.101 - 1
160.129.202.150 - 1
160.83.32.14 - 10
192.75.23.73 - 11
(etc)
webcachewp1b.cache.pol.co.uk - 11
wfp2.almaden.ibm.com - 498
wiseone.z-tel.com - 1
wobbly-bob.leeds.wwwcache.ja.net - 2
wsp01.instinet.com - 1
wwwcache.lanl.gov - 14
xenosaur.excite.com - 1
ying.crosskeys.com - 1
zzz-063255229058.splitrock.net - 1
$
```

Comparator classes

Let's say you don't like the default sort. Our example above is sorted alphabetically by the host computer name, but in practice you might want to sort by the top-level domain or by the number of hits.

You can define your own sort routine by defining a Comparator class – a class that implements the comparator interface – and passing that comparator class across to sort.

Here's an example that sorts by the number of hits:

```
import java.util.*;
import java.io.*;

public class Hms2 {
public static HashMap HostCount;
public static void main(String [] args) throws IOException {

  BufferedReader Source = new BufferedReader(
                        new FileReader( args[0]));
  HostCount = new HashMap();
  String Line;

  while ((Line = Source.readLine()) != null) {
             String Host = (new Access(Line)).Host;

             int was = 0;
             Object Got;
             if ((Got = HostCount.get(Host)) != null)
                        was = (((Integer)Got).intValue());
             HostCount.put(Host, new Integer(was+1));
  }

  Set HostKeys = HostCount.keySet();
  Iterator It = HostKeys.iterator();
  Vector HKeys = new Vector();
  while (It.hasNext()) {
             HKeys.add((String)(It.next()));
             }

  Collectionssort(HKeys, new Bynum());

  for (int i=0;i<HKeys.size();i++) {
             String K = (String)(HKeys.elementAt(i));
             Integer V = (Integer)(HostCount.get(K));
             System.out.println(K + " - " + V);
             }
  }

public static int getCount(String Key) {
             Integer Got = (Integer)(HostCount.get(Key)) ;
             return (Got.intValue());
  }
}
```

And here's the comparator class:

```
public class Bynum implements java.util.Comparator {

  public int compare(Object one, Object two) {
  return (Hms2.getCount((String)one) - Hms2.getCount((String)two));
  }
}
```

Figure 15 Running a comparator class

```
$ java Hms2 accesslog.txt
ac8a2a48.ipt.aol.com - 1
cache-mtc-al06.proxy.aol.com - 1
216-21-202-114.spectrumdsl.net - 1
violet.d48.lilly.com - 1
sv.us.ircache.net - 1
(etc)
node1.orchestream.com - 67
crawl5.googlebot.com - 80
crawl3.googlebot.com - 98
63.73.169.11 - 98
crawl4.googlebot.com - 127
crawl1.googlebot.com - 132
cache-haw.cableinet.co.uk - 137
wfp2.almaden.ibm.com - 498
marvin.northernlight.com - 519
cr013r01.sac2.fastsearch.net - 621
$
```

You'll notice that our comparator class makes a callback to the main class (*Hms2*) to determine the number of hits for one of the objects held in the HashMap, since all the comparator has been given is the Key. If our sort was based on the key (for example, if we were sorting by top-level domain) then this callback would not be necessary.

You'll also notice that the HashMap is now a static class variable in *Hms2*, necessary so that it's available to the **getCount** callback method, and not simply a local variable in main.

The Comparable interface

If you don't want to write your own Comparator class, you can also have a class of objects which implements the comparable interface; String, wrapper classes, and Date classes already implement this interface so that sorting by date (for example) is made easy.

The comparable interface requires that you define two methods:

compareTo()
equals()

Classes that implement this interface are said to have a “natural order” and default sorting uses this order. There's nothing to stop you defining another Comparator if you want to sort them differently.

Here's an example using our access log records, sorting them by the number of bytes returned, largest first:

```
import java.util.*;
import java.io.*; // for test program only

public class Acsort implements Comparable{

// Access Log file (NCSA format) analysis

String Host;
String Time;
String Request;
String Method;
String URL;
```

```
int status;
int size = 0;
String Referer;
String UserAgent;

// crawl5.googlebot.com - - [02/Sep/2001:01:11:36 -0700] "GET /book/0-7357-0949-1.html HTTP/1.0" 200 6081
"-" "Googlebot/2.1 (+http://www.googlebot.com/bot.html)"

public Acsort (String Data) {

  StringTokenizer Splitter = new StringTokenizer(Data," \t");
  String skip;

  Host = Splitter.nextToken();
  skip = Splitter.nextToken();
  skip = Splitter.nextToken("[");
  Time = Splitter.nextToken(" \t");
  skip = Splitter.nextToken("\"");

  Request = Splitter.nextToken();
  skip = Splitter.nextToken(" \t");

  status = Integer.parseInt(Splitter.nextToken(" \t"));

  try {
            size = Integer.parseInt(Splitter.nextToken(" \t"));
  } catch (Exception e) {
            size = 0;
  }

  skip = Splitter.nextToken("\"");
  Referer = Splitter.nextToken();
  skip = Splitter.nextToken();
  UserAgent = Splitter.nextToken();

  // Use another String Tokenizer on the HTTP Request

  Splitter = new StringTokenizer(Request);
  Method = Splitter.nextToken();
  URL = Splitter.nextToken();

  }

public int compareTo(Object second) {
  return (((Acsort)(second)).size - size);
  }

public boolean equals(Acsort second) {
  int diff = size - second.size;
  return (diff == 0);
  }

// Test code

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

  BufferedReader Source = new BufferedReader(
                         new FileReader( args[0]));
```

```
String Line;
Vector Actable = new Vector();

while ((Line = Source.readLine()) != null) {
            Actable.add(new Acsort(Line));
            }

Collections.sort(Actable);

for (int i=0; i<20; i++) {
            Acsort current = (Acsort)(Actable.get(i));
            System.out.println ("URL "+current.URL+
                        " from "+current.Host+
                        " was "+current.size+ " bytes");
            }

}

}
```

```
$ java Acsort accesslog.txt
URL /net/search.php4?search=C%2B%2B from febris.mcc.wwwcache.ja.net was 108734 bytes
URL /resources/questions.html from 203.197.108.181 was 44383 bytes
URL /resources/questions.html from inetgw.il.neustar.com was 44383 bytes
URL /resources/questions.html from 195-23-189-183.nr.ip.pt was 44383 bytes
URL /resources/questions.html from cip114.ccc-cable.net was 44383 bytes
URL /resources/questions.html from cr013r01.sac2.fastsearch.net was 44383 bytes
URL /net/search.php4?search=flights+to+munich from webcachew01c.cache.pol.co.uk was 43613 bytes
URL /wellimg/gje.jpg from cache-haw.cableinet.co.uk was 37527 bytes
URL /wellimg/gje.jpg from smtp.deltaimpact.co.uk was 37527 bytes
URL /wellimg/gje.jpg from 24-168-195-156.ff.cox.rr.com was 37527 bytes
URL /wellimg/gje.jpg from cache-haw.cableinet.co.uk was 37527 bytes
URL /resources/library.html from 194.73.75.22 was 34333 bytes
URL /resources/library.html from node1.orchestream.com was 34333 bytes
URL /resources/library.html from 63.73.169.11 was 34333 bytes
URL /resources/library.html from crawl1.googlebot.com was 34333 bytes
URL /resources/library.html from wfp2.almaden.ibm.com was 34333 bytes
URL /resources/library.html from salt.mcc.wwwcache.ja.net was 34333 bytes
URL /resources/library.html from twocats.demon.co.uk was 34333 bytes
URL /resources/library.html from ppp-4-35.5800-7.access.uk.worldonline.com was 34333 bytes
URL /resources/library.html from cr013r01.sac2.fastsearch.net was 34333 bytes
$
```

Figure 16 Running public class Acsort

Exercise

Write a class of objects of type Person. For each Person, you'll have a name and a job title, a constructor that takes two parameters, and a two accessor methods.

Write a second class. This one should have a main method which:

a) Creates a HashMap of objects of type person. The keys will be strings – the name of the host computer that a person is using – and the values held in the HashMap will be the person objects.

b) Uses an Iterator to loop through the HashMap, printing out each person's name, job title and computer name.

Here's some sample data:

Graham Ellis	Trainer	dupiaza
John Jones	Team Leader	vindaloo
Juanita Hamilton	Support Specialist	balti
Kathy Codd	Systems Analyst	samosa

3

License

These notes are distributed under the **Well House Consultants Open Training Notes License.** *Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - http://www.wellho.net - for further details.

Change log
Original Version, Well House Consultants, 2004

```
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
Updated by: ______________________ on ____________________
```

License Ends.