Java Utility Classes

This handout was written by Julie Zelenski.

This handout briefly describes a few of the standard Java built-in classes (`Strings`, `Arrays`, `Vector`, `Hashtable`, `Math`, `PrintStream`, etc.) that you will find useful and handy when programming in Java. We won't have much time in lecture to cover these basic classes, but they are fairly straightforward and you won't find it difficult to pick up what you need to know. If you want more detail than this handout, refer to the complete on-line documentation for all classes in the Java packages at:

```
http://java.sun.com/products/jdk/1.1/docs/api/packages.html.
```

Strings

Unlike in C, Java strings are not character pointers or null-terminated arrays of characters, but are objects of the `String` and `StringBuffer` classes. The `String` class is immutable, once its characters and length are established, its contents cannot change. It is fast and efficient and used for all simple string needs: string constants, string manipulations (substring, concat, etc.), and temporary conversions to strings (such as converting an integer to a string to print it out). The `String` class contains constructors to create a `String` object from an array of characters, an array of bytes, a `StringBuffer` object, etc. You can also directly create a `String` object from a string literal (enclosed in double quotes) as a convenience.

```
char[] chars = {'a', 'b', 'c'};
      // shorthand to allocate and initialize array
String s, t;

s = new String(chars);           // s is now the string "abc"
t = "Hello";
```

The `String` class contains a wealth of methods to operate on strings, such as `length`, `charAt` (note that you cannot access individual characters with subscripts, since `Strings` are not arrays!), `equals`, `startsWith`, `endsWith`, `hashCode` (used to produce a hash value from a string) , `indexOf` (overloaded to search for chars or strings), `substring`, `concat`, `replace`, and conversions to string from built-in types (`boolean`, `int`, `double`, etc.). You will find its functionality similar to that of the `strlib` from CS106, but note that the operations are not functions, but methods, which you send to a `String` object as the receiver. For example, to compare two `String` objects, you send one `String` the `equals` or `compareTo` message passing the other `String` as an argument. The methods which return strings (`concat`, `substring`, etc.) don't change the `String` you send the message to, they create new `String` objects and return those (in a familiar LISP-like theme). In general, Java does not allow operator overloading, except in the case of allowing `+` to be used to concatenate two `String` objects into a new `String`. Some basic `String` usage:

```
String s, t;
int position;

s = "abc" + "def";
for (int i = 0; i < s.length(); i++)
   System.out.println(s.charAt(i)): // prints characters line by line

position = s.indexOf('d');    // find first occurrence of d in the string
t = s.substring(0, position); // make a new string that is a piece of this one
if (s.equals("CS107"))        // test for equality (case-sensitive)
   s = "Something else";
```

The `StringBuffer` class is a resizable, changeable sequence of characters. `StringBuffers` have editing methods such as `setLength`, `setCharAt`, `append`, `insert`, and `reverse` that change the receiving object and its characters rather than create new `Strings`. It is primarily used to build up character sequences and then `toString` is used to create `Strings` from them. For example, this

```
"I have " + 3 + " apples."
```

is compiled to

```
new StringBuffer().append("I have ").append(3).append("apples.").toString();
```

For our purposes, you will probably only need to use the `String` class, not the `StringBuffer`. You can refer to the JavaSoft docs to learn more about the details about features and methods of either class.

Arrays

Syntactically, Java arrays look a lot like C arrays, but the resemblance is only superficial— the underlying implementation is quite different. For starters, arrays and pointers are not the same thing in Java, and there is no pointer math. Java arrays are objects in their own right, know their own length, and do bounds-checking on all element accesses.

When you declare an array, you always leave the size unspecified, and you use the new operator to create an array of the length you need:

```
int[] numbers; // you can put the brackets on either type or name, actually
               // either way, it starts out as null

numbers = new int[100];   // create new array object of 100 integers
```

At this point, `numbers` is an array of 100 integers. You can access elements using the familiar C subscript notation and you can access the array's `length` field to retrieve the number of elements in the array. Elements are indexed from `0` to `length-1`, any attempt

to read or write an element out of bounds will raise an `IndexOutOfBoundsException` at runtime.

```
for (int i = 0; i < numbers.length; i++)
   numbers[i] = i;
```

The length field is like a final data member, you cannot change its value. Once an array has been allocated of a certain length, its length does not change.
You also can declare, allocate, and initialize a Java array in one shorthand step with this syntax borrowed from C:

```
int[] numbers = {1, 10, 4, 25, 5};     // allocates int array of 5 elements
```

You can declare arrays of any base type—either one of the basic primitive types (`int`, `char`, `boolean`, etc.) as well as any class type (`Object`, `String`, `Point`, etc.) When an array is first allocated, all of the elements are cleared (so each integer element would be zero, boolean elements would be false, object references would be null, etc.) Not allowing any new variables to have random uninitialized contents is part of Java's strategy to ensure safety and portability. Note that when you declare and allocate an array of objects, it merely makes space for the array itself, but each of the elements will be a `null` reference. If you reference one of these array elements before it has been assigned a valid object reference, you will get a null pointer exception at runtime. For example, in the following buggy code:

```
Point[] points;

points = new Point[5];     // make space for array, but no objects
allocated
points[0].translate(10, 10);     // will raise a null pointer exception!
```

What we need to do is to loop and allocate each Point object and assign into the array:

```
points = new Point[5];     // make space for array, but no objects
allocated
for (int i = 0; i < points.length; i++)
   points[i] = new Point();      // allocate each Point object

points[0].translate(10, 10);     // now this is just fine
```

Since arrays themselves are objects and thus implemented as references, assigning one array to another makes only a shallow copy and changes made to one array are seen by both:

```
int[] numbers, times;   // putting bracket on int means both variables are
arrays

numbers = new int[100];
times = numbers;
times[4] = 45;       // both times[4] & numbers[4] are same element
```

To handle arrays of more than one dimension, Java uses a strategy of building "arrays of arrays" rather than using multi-dimensional arrays. For example, this declaration:

```
int[][] grid;
```

says that `grid` is an array in which the elements are themselves arrays of integers. We can allocate the storage for all the arrays (both inner and outer) with one call to `new`:

```
grid  = new int[5][10]; // allocates 5 rows by 10 columns of integers
```

This creates a grid which stores 5 "rows" by 10 "columns". Unlike C, this is not implemented a contiguous block of 50 integers— `grid` itself is an array, which has 5 elements, each are of type integer array, each contain 10 elements. When we access `grid[0][0]`, it retrieves the first element from `grid`, which is an array itself, and then it retrieves the first element from that array, which is an integer. The expression `grid[0]` refers to the array of 10 integer elements that represents the first row, which you can manipulate as a usual array. Here's a loop that assigns all the integers in the `grid` by iterating over the outer array, and then over the inner arrays:

```
for (int row = 0; row < grid.length; row++)        // iterate over all rows
   for (int col = 0; col < grid[row].length; col++) // iterate over columns
      grid[row][col] = 100;
```

One advantage of this design is that the individual rows don't all have to be of the same length, as they usually would in C, since each row is an independent array. To build something that takes advantage of this, we would first allocate the outermost array of rows, and then separately allocate the inner arrays to different lengths as desired:

```
int[][] grid;

grid  = new int[5][];      // allocates 5 rows, no columns yet, each is
null
   for (int row = 0; row < grid.length; row++)  // iterate over rows
      grid[row] = new int[row];
```

Just like all Java arrays, arrays of arrays also know their lengths and always bounds-check all element accesses.

Vector

Arrays are fine if you know the length of the collection in advance and the length never changes, but for many uses that is too restrictive. The `java.util` package includes a `Vector` object that is close in functionality to our beloved `DArray`. It has methods such as `size` (to return the number elements), `elementAt`, `setElementAt`, `addElement`, `insertElementAt`, `indexOf`, `contains`, `removeElement`, and more. It handles all of the memory management using a chunk allocation strategy very similar to that of the `DArray`.

In order for any collection abstraction to be truly useful, it needs to work for elements of any type. For our C `DArray`, the client stated the size on creation and worked with `void *`, while the implementation resorted to pointer arithmetic and `memcpy` to copy values into the collection. Not an ideal strategy for client or implementor as I'm sure you well remember!

How does the Java `Vector` class accomplish generic behavior? A `Vector` is stated to hold things of `Object` type. Because all objects are implemented as references (pointers), they are all the same size. Since all classes must eventually derive from `Object`, and any subclass is compatible with its superclass, this means a `Vector` can hold any type of object. And since almost everything in Java is an object, this works out just fine. For the few excluded built-ins (`int`, `char`, etc.) that are not objects, there are object wrappers (`Integer`, `Boolean`, `Double`, etc.) that serve to convert a basic built-in into an object form.

```
   Vector numbers;

   numbers = new Vector();                 // creates an empty Vector object
   for (int i = 0; i < 100; i++)
      numbers.addElement(new Integer(i)); // creates Integer object from
                                          number,
                                          // adds to vector
```

The return type from `elementAt` is declared to be of `Object` type. If you need to use the returned object and send it messages more specific to its class, you will first need to cast it to the proper class. Although the cast syntax in Java looks like the unsafe static cast mechanism from C (which we have come to rightly view with suspicion), Java casts are a safe runtime entity. If you attempt to incompatibly cast from one type to another, an exception is raised.

```
   Vector shapes = new Vector();

   shapes.addElement(new Circle(100, 100)); // add some Shape objects
(subclasses)
   shapes.addElement(new Rectangle(100, 100));
   shapes.addElement(new Triangle(100, 100));

   for (int i = 0; i < shapes.size(); i++)
      ((Shape)shapes.elementAt(i)).drawSelf();
```

How does the `Vector` deal with the problem of type-safety? You don't state anything about the type of objects being stored in a given `Vector`, and in fact, all elements don't even have to be of the same type. We might have various types of `Shape` objects or even unrelated objects in the vector. Mixing in different type objects doesn't cause any problems if that's what you wanted to do. But what if it were truly a mistake, let's say you accidentally add a `String` into that vector of `Shape` objects? You don't have any problems when adding it, since a `String` object is compatible with the base `Object` type expected by the `Vector`. But when you pull that element out and cast it to a `Shape`, it will fail with a runtime cast exception. In some sense this is similar to the list from LISP, which doesn't rely on compile-time type homogeneity to enforce type safety, but

instead uses runtime type information to stop any attempt to mis-use or misinterpret the contents you earlier stored.

How does the `Vector`'s approach compare to the `void*` strategy of the `DArray` from a client perspective? How does it compare to the list of LISP? For those of you acquainted with C++, how does this strategy compare to generic behavior accomplished through use of templates?

One interesting question to ponder: how does the `contains` method know how whether an element is equal to the key? (there are no function pointers in Java at this time...) Check out the on-line docs and learn how it works!

Packages

One thing to note is that the `Vector` class is defined in the `java.util` package. The classes `String`, `Math`, `Thread`, etc., are in the `java.lang` package which is always visible without taking any special steps, but you need to do things a little differently to get to things outside the standard package. One way is to use the fully qualified class name like this:

```
java.util.Vector shapes = new java.util.Vector();
```

The other is include an "`import`" line that brings in the needed class or package so that you can refer to the class by the name `Vector` without the package qualifier. You can `import` just the `Vector` class, like this:

```
import java.util.Vector;
```

Or you can import all the classes in a package using the wildcard '`*`' character, which brings in short names for all the classes within the `java.util` package:

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

Hashtable

The `java.util` package also contains another quite useful class, the `Hashtable`. The `Hashtable` allows you to store data tagged by a key that is hashed for fast retrieval. It resolves collisions by chaining each bucket in a linked list. It is built on a fairly clever implementation that knows how to increase the number of buckets and re-hash when the table load gets too high.

The interesting methods for this object are `put`, `get`, and `remove`. When entering and retrieving entries, both the key and data are declared to be of `Object` data, so any object will be accepted. The object used as the key must respond to the `hashCode` and `equals` method in order to hash it to its own bucket and compare to find it later. The `Object` class has default implementations for both of these methods using the pointer value of the `Object`. Often, you need to override them and implement versions appropriate for

your class— for example, the `String` class overrides these to provide versions more appropriate for `Strings`.

The return type from `get` will need to be appropriately casted (just as above when accessing elements in the `Vector`). A code excerpt that shows you how might store bank accounts using the owner's name (a `String`) as the key:

```
Hashtable table = new Hashtable();  // creates an empty table
Account[] accts;

// imagine code here to allocate array of Accounts,
// set up with data about owner & balance

for (int i = 0; i < accts.length; i++)
   table.put(accts[i].ownerName(), accts[i]);
         // store each account under name in table

Account jerry = (Account) table.get("Jerry Cain");
         // look up account by String name
         // cast result to Account object
```

Enumeration

There are no function pointers in Java and so there is not a "map" operation for `Vector` or `Hashtable`. Instead, you must write client-side loops that iterate over the elements to get this effect. The `Vector` and `Hashtable` classes share a common interface for enumeration. The idea of an "`iterator`" or "`enumerator`" is gaining favor as the tidy way for a data structure to allow a client to step through and access all the elements. You create an `Enumeration` by asking the `Vector` or `Hashtable` for its `elements`, then loop while it returns `true` on `hasMoreElements`. The `nextElement` method retrieves each element in turn, e.g., to print all elements of a `Vector v`:

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;)
   System.out.println(e.nextElement());
```

The same exact loop could be used to print all the elements of `Hashtable`, just change the object whom you send the `elements` message to.

Interfaces

The `Enumeration` type above is something called an **interface** in Java. An interface is like an abstract class— it establishes a set of methods that a class can claim to respond to. (The idea of interfaces was pilfered from the "protocol" concept in NeXT's Objective-C, by the way.)

A class may only have one superclass, however it may implement as many interfaces as it likes. It must provide implementations for all the messages in the interface. Unlike inheritance, subscribing to an interface does not inherit any implementation for the methods, it just brings in prototypes which the class must implement. In the literature,

this is described as "subclassing vs. subtyping". Subclassing means inheriting both an interface and its implementation. Subtyping means inheriting just the interface without any code, which is appropriate for classes that do not share implementation structure.

You need this feature when you have classes in unrelated parts of the hierarchy that all need to respond (polymorphically) to some message. Interfaces are an excellent language feature—they allow you to do almost all of the interesting things multiple inheritance enabled, but without most of multiple inheritance's problems.

```
public interface Colorable {
   public abstract void setColor(Color c);
   public abstract Color getColor();
}

public class Square extends Shape implements Colorable {
   ....
   public void setColor() { /*change color however Squares do it */ }
   ...
}

public class ColoredString extends String implements Colorable {
   ...
   public void setColor() { /*change color however Colored Strings do it */
}
   ...
}


Colorable[] things;  // contains a heterogeneous mess of objects
                     // which all implement the Colorable interface
...
for (int i = 0; i < things.length; i++) {
   things[i].setColor(Color.red);  // polymorphism without relying on
inheritance
}
```

Math

Given there are no "functions" in Java and all newly-defined types must be classes, this sometimes leads to awkward problems for the things which don't fit the object-oriented model. One such example is the set of mathematical operations: exponentiation, square root, sine, cosine, logarithms, etc. — where do those go?

Java provides a Math class in the standard `java.lang` package that offers these operations defined as **static** methods. Static methods are often called "class methods"— they are methods you send to the class itself, not to a particular instance of the class. We don't create a `Math` object to send it the `sqrt` message, we directly send the `sqrt` message to the `Math` class itself, like this:

```
System.out.println("Square root of 2 is " + Math.sqrt(2));
```

Yes, this is a bit hokey, but that's the way Java does it. One static method of the `Math` class you might find useful is `random` which returns a `double` in the range [0.0 to 1.0) which you can then scale to a random integer from some interval.

PrintStream

Something you use all the time without even quite noticing it is the `PrintStream` class. The `public` variable `out` of the `System` class is an object of class `PrintStream`, it is the "`stdout`" of the Java world. The two most frequently used methods are `print` and `println`. They both operate in the same way, the only difference concerns whether they add a new line to the end of what is printed (`println` does and `print` doesn't).

Both `print` and `println` take exactly one argument and are multiply overloaded to allow you to pass any type of parameter as that argument, be it a primitive type like int or char or an object of any class. When passed an object, the object is sent a `toString` message (see discussion in `Object` base class below) to get a `String` representation to print out.

Rather than allow variables arguments like C's `printf` (which is notoriously unsafe), you construct combined expressions to print by repeated string concatenation, usually via the `+` operator. Each operand to `+` is first converted to a `String` and then concatenated together, the final `String` result is handed to the `PrintStream` to output:

```
System.out.println("A string " + Math.sqrt(2) + " and more " + obj);
```

Object base class

All classes in Java eventually derive from `Object`, the simplest of all classes. There is not much functionality defined in this class, but it's worth nothing a few methods with simple implementations intended to be overridden by subclasses. The `equals` method returns a `boolean` result on whether the passed `Object` is equal to the receiver. The default implementation just uses pointer equality as the test. Similarly, the default `hashCode` method produces a `hashCode` based on the hashing the pointer. And finally, the `toString` method is sent to an `Object` to get a `String` representation of its data, usually in order to print it out. The default implementation just produces a `String` of the class name and pointer address of the object.

Your classes will inherit the simple versions from `Object`, which may be sufficient, but any subclass is welcome to override any of these methods and provide more appropriate versions for a class. For example, `String` overrides `equals` to compare the characters in the two `String`s, not just looking at the pointers.