Lecture 3: A Quick Introduction to Java

Read: Chapt 1 of Weiss (from 1.4 to the end).

Overview: This lecture will present a very quick description of the Java language (at least the portions that will be needed for the class projects. The Java programming language has a number of nice features, which make it an excellent programming language for implementing data structures. Based on my rather limited experience with Java, I find it to be somewhat cleaner and easier than C++ for data structure implementation (because of its cleaner support for strings, exceptions and threads, its automatic memory management, and its more integrated view of classes). Nonetheless, there are instances where C++ would be better choice than Java for implementation (because of its template capability, its better control over memory layout and memory management).

Java API: The Java language is very small and simple compared to C++. However, it comes with a very large and constantly growing library of utility classes. Fortunately, you only need to know about the parts of this library that you need, and the documentation is all online. The libraries are grouped into *packages*, which altogether are called the "Java 2 Platform API" (application programming interface). Some examples of these packages include

`java.lang`: Contains things like strings, that are essentially built in to the language.

`java.io`: Contains support for input and output, and

`java.util`: Contains some handy data structures such as lists and hash tables.

Simple Types: The basic elements of Java and C++ are quite similar. Both languages use the same basic lexical structure and have the same methods for writing comments. Java supports the following *simple types*, from which more complex types can be constructed.

Integer types: These include `byte` (8-bit), `short` (16-bit), `int` (32-bit), and `long` (64-bit). All are *signed* integers.

Floating point types: These include `float` (32-bit) and `double` (64-bit).

Characters: The `char` type is a 16-bit unicode character (unlike C++ where it behaves essentially like a `byte`.)

Booleans: The `boolean` type is the analogue to C++'s `bool`, either `true` or `false`.

Variables, Literals, and Casting: Variable declarations, literals (constants) and casting work much like they do in C++.

```
int i = 10 * 4;                    // integer assignment
byte j = (byte) i;                 // example of a cast
long l = i;                        // cast not needed when "widening"
float pi1 = 3.14f;                 // floating assignment
double pi2 = 3.141593653;
char c = 'a';                      // character
boolean b = (i < pi1);             // boolean
```

All the familiar arithmetic operators are just as they are in C++.

Arrays: In C++ an array is just a pointer to its first element. There is no index range checking at run time. In Java, arrays have definite lengths and run-time index checking is done. They are indexed starting from 0 just as in C++. When an array is declared, no storage is allocated. Instead, storage is allocated by using the `new` operator, as shown below.

```
int A[];                              // A is an array, initially null
A = new int[4] = {12, 4, -6, 41};     // this allocates and defines array
char B[][] = new char[3][8];          // B is a 2-dim, 3 x 8 array of char
B[0][3] = 'c';
```

In Java you do not need to delete things created by new. It is up to the system to do garbage collection to get rid of objects that you no longer refer to. In Java a 2-dimension array is actually an array of arrays.

String Object: In C++ a string is implemented as an array of characters, terminated by a null character. Java has much better integrated support for strings, in terms of a special String object. For example, each string knows its length. Strings can be concatenated using the "+" operator (which is the only operator overloading supported in Java).

```
String S = "abc";                     // S is the string "abc"
String T = S + "def";                 // T is the string "abcdef"
char c = T[3];                        // c = 'd'
S = S + T;                            // now S = "abcabcdef"
int k = T.length();                   // k = 6
```

Note that strings can be appended to by concatenation (as shown above with S). If you plan on making many changes to a string variable, there is a type `StringBuffer` which provides a more efficient method for "growable" strings.

Java supports automatic casting of simple types into strings, which comes in handy for doing output. For example,

```
int cost = 25;
String Q = "The value is " + cost + " dollars";
```

Would set `Q` to "The value is 25 dollars". If you define your own object, you can define a method `toString()`, which produces a String representation for your object. Then any attempt to cast your object to a string will invoke your method.

There are a number of other string functions that are supported. One thing to watch out for is string comparison. The standard == operator will not do what you might expect. The reason is that a String object (like all objects in Java) behaves much like a pointer or reference does in C++. Comparisons using == test whether the pointers are the same, not whether the contents are the same. The `equals()` or `compareTo()` methods to test the string contents.

Flow Control: All the standard flow-control statements, `if-then-else`, `for`, `while`, `switch`, `continue`, etc., are the same as in C++.

Standard I/O: Input and output from the standard input/output streams are not quite as simple in Java as in C++. Output is done by the `print()` and `println()` methods, each of which prints a string to the standard output. The latter prints an end-of-line character. In Java there are no global variables or functions. Everything is referenced as a member of some object. These functions exist as part of the `System.out` object. Here is an example of its usage.

```
int age = 12;
System.out.println("He is " + age + " years old");
```

Input in Java is much messier. Since our time is limited, we'll refer you to our textbook (by Weiss) for an example of how to use the `java.io.BufferedReader`, `java.io.InputStream-Reader`, `StringTokenizer` to input lines and read them into numeric, string, and character variables.

Classes and Objects: Virtually everything in Java is organized around the notion of hierarchy of objects and classes. As in C++ an *object* is an entity which stores data and provides methods for accessing and modifying the data. An object is an instance of a *class*, which defines the specifics of the object. As in C++ a class consists of *instance variables* and *methods*. Although classes are superficially similar to classes in C++, there are some important differences in how they operate.

File Structure: Java differs from C++ in that there is a definite relationship between files and classes. Normally each Java class resides within its own file, whose name is derived from the name of the class. For example, a class `Student` would be stored in file `Student.java`. Unlike C++, in which class declarations are stored in different files (.h and .cc), in Java everything is stored in the same file and all definitions are made within the class.

Packages: Another difference with C++ is the notion of *packages*. A package is a collection of related classes (e.g. the various classes that make up one data structure). The classes that make up a package are stored in a common directory (with the same name as the package). In C++ you access predefined objects using an `#include` command to load the definitions. In Java this is done by *importing* the desired package. For example, if you wanted to use the `Stack` data structure in the `java.util` library, you would load this using the following.

```
import java.util.Stack;
```

The Dot (.) Operator: Class members are accessed as in C++ using the dot operator. However in Java, objects all behave as if they were pointers (or perhaps more accurately as references that are allowed to be null).

```
class Person {
    String name;
    int    age;
    public Person() {...}          // constructor (details omitted)
}
Person p;                          // p has the value null
p = new Person();                  // p points to a new object
Person q = p;                      // q and p now point to the same object
p = null;                          // q still points to the object
```

Note that unlike C++ the declaration "`Person p`" did *not* allocate an object as it would in C++. It creates a null pointer to a Person. Objects are only created using the "`new`" operator. Also, unlike C++ the statement "`q = p`" does *not* copy the contents of p, but instead merely assigns the pointer q to point to the same object as p. Note that the pointer and reference operators of C++ ("`*`", "`&`", "`->`") do not exist in Java (since everything is a pointer).

While were at it, also note that there is no semicolon (;) at the end of the class definition. This is one small difference between Java and C++.

Inheritance and Subclasses: As with C++, Java supports class inheritance (but only single not multiple inheritance). A class which is derived from another class is said to be a child or *subclass*, and it is said to *extend* the original parent class, or *superclass*.

```
class Person { ... }
class Student extends Person { ... }
```

The Student class inherits the structure of the Person class and can add new instance variables and new methods. If it redefines a method, then this new method *overrides* the old method. Methods behave like virtual member functions in C++, in the sense that each object "remembers" how it was created, even if it assigned to a superclass.

```
class Person {
    public void print() { System.out.println("Person"); }
}
class Student extends Person {
    public void print() { System.out.println("Student"); }
}
Person  P = new Person();
Student S = new Student();
Person  Q = S;
Q.print();
```

What does the last line print? It prints "Student", because Q is assigned to an object that was created as a Student object. This called *dynamic dispatch*, because the system decides at run time which function is to be called.

Variable and Method Modifiers: Getting back to classes, instance variables and methods behave much as they do in C++. The following *variable modifiers* and *method modifiers* determine various characteristics of variables and methods.

`public`: Any class can access this item.

`protected`: Methods in the same package or subclasses can access this item.

`private`: Only this class can access this item.

`static`: There is only one copy of this item, which is shared between all instances of the class.

`final`: This item cannot be overridden by subclasses. (This is allows the compiler to perform optimizations.)

`abstract`: (Only for methods.) This is like a pure virtual method in C++. This method is not defined here (no code is provided), but rather must be defined by a subclass.

Note that Java does not have `const` variables like C++, nor does it have enumerations. A constant is defined as a `static final` variable. (Enumerations can be faked by defining many constants.)

```
class Person {
    static final int TALL   = 0;
    static final int MEDIUM = 1;
    static final int SHORT  = 2;
    ...
}
```

Class Modifiers: Unlike C++, classes can also be associated with modifiers.

(Default:) If no modifier is given then the class is *friendly*, which means that it can be instantiated only by classes in the same package.

`public`: This class can be instantiated or extended by anything in the same package or anything that imports this class.

`final`: This class can have no subclasses.

`abstract`: This class has some abstract methods.

"this" and "super": As in C++, `this` is a pointer to the current object. In addition, Java defines `super` to be a pointer to the object's superclass. The super pointer is nice because it provides a method for explicitly accessing overridden methods in the parent's class and invoking the parent's constructor.

```
class Person {
    int age;
    public Person(int a) { age = a; }       // Person constructor
}
class Student extends Person {
    String major;
    public Student(int a, String m) {       // Student constructor
       super(a);                            // construct Person
       major = m;
    }
}
```

(By the way, this is a dangerous way to initialize a String object, since recall that assignment does not copy contents, it just copies the pointer.)

Functions, Parameters, and Wrappers: Unlike C++ all functions must be part of some class. For example, the main procedure must itself be declared as part of some class, as a public, static, void method. The command line arguments are passed in as an array of strings. Since arrays can determine their own length from the `length()` method, there is no need to indicate the number of arguments. For example, here is what a typical Hello-World program would look like. This is stored in a file `HelloWorld.java`

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World");
        System.out.println("The number of arguments is " + args.length());
    }
}
```

In Java simple types are passed by value and objects are passed by reference. Thus, modifying an object which has been passed in through a parameter of the function has the effect of modifying the original object in the calling routine.

What if you want to pass an integer to a function and have its value modified. Java provides a simple trick to do this. You simply *wrap* the integer with a class, and now it is passed by reference. Java efines wrapper classes `Integer` (for `int`), `Long`, `Character` (for `char`), `Boolean`, etc.

Generic Data Structures without Templates: Java does not have many of the things that C++ has, templates for example. So, how do you define a generic data structure (e.g. a Stack) that can contain objects of various types (int, float, Person, etc.)? In Java every class automatically extends a most generic class called `Object`. We set up our generic stack to hold objects of type `Object`. Since this is a superclass of all classes, we can store any objects from any class in our stack. Of course, to keep our sanity, we should store object of only one type in any given stack. (Since it will be very difficulty to figure out what comes out whenever we pop something off the stack.)

Consider the `ArrayVector` class defined below. Suppose we want to store integers in this vector. An `int` is not an object, but we can use the wrapper class `Integer` to convert an integer into an object.

```
ArrayVector A = new ArrayVector();
A.insertAtRank(0, Integer(999));          // insert 999 at position 0
A.insertAtRank(1, Integer(888));          // insert 888 at position 1
Integer x = (Integer) A.elemAtRank(0);    // accesses 999
```

Note that the ArrayVector contains `Objects`. Thus, when we access something from the data structure, we need to cast it back to the appropriate type.

```
// This is an example of a simple Java class, which implements a vector
// object using an array implementation.  This would be stored in a file
// named "ArrayVector.java".

public class ArrayVector {
    private Object[] a;           // Array storing the elements of the vector
    private int capacity = 16;    // Length of array a
    private int size = 0;         // Number of elements stored in the vector

    // Constructor
    public ArrayVector() { a = new Object[capacity]; }

    // Accessor methods
    public Object elemAtRank(int r) { return a[r]; }

    public int size() { return size; }

    public boolean isEmpty() { return size() == 0; }

    // Modifier methods
    public Object replaceAtRank(int r, Object e) {
        Object temp = a[r];
        a[r] = e;
        return temp;
    }

    public Object removeAtRank(int r) {
        Object temp = a[r];
        for (int i=r; i < size-1; i++)      // Shift elements down
            a[i] = a[i+1];
        size--;
        return temp;
    }

    public void insertAtRank(int r, Object e) {
        if (size == capacity) {             // An overflow
            capacity *= 2;
            Object[] b = new Object[capacity];
            for (int i=0; i < size; i++)
                b[i] = a[i];
            a = b;
        }
        for (int i=size-1; i>=r; i--)       // Shift elements up
            a[i+1] = a[i];
        a[r] = e;
        size++;
   }
}
```