Java for C++ Programmers

Contents

Introduction

The purpose of these notes is to help students in Computer Sciences 537 (Introduction to Operating Systems) at the University of Wisconsin - Madision learn enough Java to do the course projects. The Computer Sciences Department is in the process of converting most of its classes from C++ to Java as the principal langauge for programming projects. CS 537 was the first course to make the switch, Fall term, 1996. At that time vitually all the students had heard of Java and none had used it. Over the last few years more and more of our courses were converted to Java. Finally last year (1998-99), the introductory programming prerequisites for this course, CS 302 and CS 367, were taught in Java. Nonetheless, many students are unfamiliar with Java, having learned how to program from earlier versions of 302 and 367, or from courses at other institutions.

Applications *vs* Applets

The first thing you have to decide when writing a Java program is whether you are writing an *application* or an *applet*. An applet is piece of code designed to display a part of a document. It is run by a browser (such as Netscape Navigator or Microsoft Internet Explorer) in response to an `<applet>` tag in the document. We will not be writing any applets in this course.

An application is a stand-alone program. All of our programs will be applications.

Java was orginally designed to build active, multimedia, interactive environments, so its standard runtime library has lots of features to aid in creating user interfaces. There are standard classes to create scrollbars, pop-up menus, etc. There are special facilities for manipulating URL's and network connections. We will not be using any of these features. On the other hand, there is one thing operating systems and user interfaces have in common: They both require multiple, cooperating threads of control. We will be using those features in this course.

JavaScript

You may have heard of JavaScript. JavaScript is an addition to HTML (the language for writing Web pages) that supports creation of ``subroutines''. It has a syntax that looks sort of like Java, but otherwise it has very little to do with Java. I have heard one very good analogy: JavaScript is to Java as the C Shell (csh) is to C.

The Java API

The Java langauge is actually rather small and simple - an order of magnitude smaller and simpler than C++, and in some ways, even smaller and simpler than 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 really need, you can learn about it a little at a time, and there is excellent, browsable, on-line documentation. These libraries are grouped into *packages*. One set of about 60 packgages, called the *Java 2 Platform API* comes bundled with the language (API stands for "Application Programming Interface"). You will probably only use classes from three of these packages:

- java.lang contains things like character-strings, that are essentially "built in" to the langauge.
- java.io contains support for input and output, and
- java.util contains some handy data structures such as lists and hash tables.

A First Example

Large parts of Java are identical to C++. For example, the following procedure, which sorts an array of integers using insertion sort, is exactly the same in C++ or Java.[1]

```
/** Sort the array a[] in ascending order
 ** using an insertion sort.
 */
void sort(int a[], int size) {
    for (int i = 1; i < size; i++) {
        // a[0..i-1] is sorted
        // insert a[i] in the proper place
        int x = a[i];
        int j;
        for (j = i-1; j >=0; --j) {
            if (a[j] <= x)
                break;
            a[j+1] = a[j];
        }
        // now a[0..j] are all <= x
        // and a[j+2..i] are > x
        a[j+1] = x;
    }
}
```

Note that the syntax of control structures (such as `for` and `if`), assignment statements, variable declarations, and comments are all the same in Java as in C++.

To test this procedure in a C++ program, we might use a ``main program'' like this:

```
#include <iostream.h>
#include <stdlib.h>
extern "C" int random();

/** Test program to test sort */
int main(int argc, char *argv[]) {
    if (argc != 2) {
        cerr << "usage: sort array-size" << endl;
        exit(1);
    }
    int size = atoi(argv[1]);
    int *test = new int[size];
    for (int i = 0; i < size; i++)
        test[i] = random() % 100;
    cout << "before" << endl;
    for (int i = 0; i < size; i++)
```

```
        cout << " " << test[i];
    cout << endl;

    sort(test, size);

    cout << "after" << endl;
    for (int i = 0; i < size; i++)
        cout << " " << test[i];
    cout << endl;
    return 0;
}
```

A Java program to test the sort procedure is different in a few ways. Here is a complete Java program using the `sort` procedure.

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

class SortTest {
    /** Sort the array a[] in ascending order
     ** using an insertion sort.
     */
    static void sort(int a[], int size) {
        for (int i = 1; i < size; i++) {
            // a[0..i-1] is sorted
            // insert a[i] in the proper place
            int x = a[i];
            int j;
            for (j = i-1; j >=0; --j) {
                if (a[j] <= x)
                    break;
                a[j+1] = a[j];
            }
            // now a[0..j] are all <= x
            // and a[j+2..i] are > x
            a[j+1] = x;
        }
    }

    /** Test program to test sort */
    public static void main(String argv[]) {
        if (argv.length != 1) {
            System.out.println("usage: sort array-size");
```

```
            System.exit(1);
        }
        int size = Integer.parseInt(argv[0]);
        int test[] = new int[size];
        Random r = new Random();

        for (int i = 0; i < size; i++)
            test[i] = (int)(r.nextFloat() * 100);
        System.out.println("before");
        for (int i = 0; i < size; i++)
            System.out.print(" " + test[i]);
        System.out.println();

        sort(test, size);

        System.out.println("after");
        for (int i = 0; i < size; i++)
            System.out.print(" " + test[i]);
        System.out.println();

        System.exit(0);
    }
}
```

A copy of this program is available in `~cs537-1/public/examples/SortTest.java`. To try it out, create a new directory and copy the example to a file named `SortTest.java` in that directory or visit with your web browser and use the *Save As...* option from the *File* menu. The file *must* be called `SortTest.java`!

```
mkdir test1
cd test1
cp ~cs537-1/public/examples/SortTest.java SortTest.java
javac SortTest.java
java SortTest 10
```

(The C++ version of the program is also available in `~cs537-1/public/examples/sort.cc`).

The `javac` command invokes the Java compiler on the source file `SortTest.java`. If all goes well, it will create a file named `SortTest.class`, which contains code for the Java virtual machine. The `java` command invokes the Java interpreter to run the code for class `SortTest`. Note that the first parameter is `SortTest`, not `SortTest.class` or `SortTest.java` because it is the name of a *class*, not a file.

There are several things to note about this program. First, Java has no ``top-level'' or ``global'' variables or functions. A Java program is *always* a set of `class` definitions. Thus, we had to make `sort` and `main` member functions (called ``methods'' in Java) of a class, which we called `SortTest`.

Second, the `main` function is handled somewhat differently in Java from C++. In C++, the first function to be executed is always a function called `main`, which has two arguments and returns an integer value. The return value is the ``exit status'' of the program; by convention, a status of zero means ``normal termination'' and anything else means something went wrong. The first argument is the number of words on the command-line that invoked the program, and the second argument is a an array of character strings (denoted `char *argv[]` in C++) containing those words. If we invoke the program by typing

```
sort 10
```

we will find that `argc==2`, `argv[0]=="sort"`, and `argv[1]=="10"`.

In Java, the first thing executed is the method called `main` of the indicated class (in this case `SortTest`). The `main` method does not return any value (it is of type `void`). For now, ignore the words ```public static`'' preceding `void`. We will return to these later. The `main` method takes only one parameter, an array of strings (denoted `String argv[]` in Java). This array will have one element for each word on the command line *following* the name of the class being executed. Thus in our example call,

```
java SortTest 10
```

`argv[0] == "10"`. There is no separate argument to tell you how many words there are, but in Java, you can tell how big *any* array is by using `length`. In this case `argv.length == 1`, meaning `argv` contains only one word.

The third difference to note is the way I/O is done in Java. `System.out` in Java is roughly equivalent to `cout` in C++ (or `stdout` in C), and

```
System.out.println(whatever);
```

is (even more) roughly equivalent to

```
cout << whatever << endl;
```

Our C++ program used three functions from the standard library, `atoi`, `random`, and `exit`. `Integer.parseInt` does the same thing as `atoi`: It converts the character-string "10" to the integer value ten, and `System.exit(1)` does the same thing as `exit(1)`: It immediately terminates the program, returning an exit status of 1 (meaning something's wrong). The library class `Random` defines random-number generators. The statement `Random r = new Random()` create an instance of this class, and `r.nextFloat()` uses it to generate a floating point number between 0 and 1. The *cast* `(int)` means

the same thing in Java as in C++. It converts its floating-point argument to an integer, throwing away the fraction.

Finally, note that the `#include` directives from C++ have been replaced by `import` declarations. Although they have roughly the same effect, the mechanisms are different. In C++, `#include <iostream.h>` pulls in a source file called `iostream.h` from a source library and compiles it along with the rest of the program. `#include` is usually used to include files containing declarations of library functions and classes, but the file could contain any C++ source code whatever. The Java declaration `import java.util.Random` imports the pre-compiled class `Random` from a *package* called `java.util`. The next section explains more about packages.

Names, Packages, and Separate Compilation

As in C or C++, case is significant in identifiers in Java. Aside from a few reserved words, like **if**, **while**, etc., the Java langauge places no restrictions on what names you use for functions, variables, classes, etc. However, there is a standard naming convention, which all the standard Java libraries follow, and which you *must* follow in this class.

- Names of classes are in MixedCase starting with a capital letter. If the most natural name for the class is a phrase, start each word with a capital letter, as in StringBuffer.
- Names of "constants" (see below) are ALL_UPPER_CASE. Separate words of phrases with underscores as in MIN_VALUE.
- Other names (functions, variables, reserved words, etc.) are in lower case or mixedCase, starting with a lower-case letter.

A more extensive set of guidelines is included in the Java Language Specification.

Simple class defintions in Java look rather like class definitions in C++ (although, as we shall see later, there are important differences).

```
class Pair { int x, y; }
```

Each class definition should go in a separate file, and the name of the source file must be exactly the same (including case) as the name of the class, with "`.java`" appended. For example, the definition of `Pair` must go in file `Pair.java`. The file is compiled as shown above and produces a `.class` file. There are exceptions to the rule that requires a separate source file for each class. In particular, class definitions may be nested. However, this is an advanced feature of Java, and you should *never nest class definitions* unless you known what you're doing!

There is a large set of predefined classes, grouped into *packages*. The full name of one of these predefined classes includes the name of the package as prefix. We already saw the class `java.util.Random`. The `import` statement allows you to omit the package name from one of these classes. Because the SortTest program starts with

```
import java.util.Random;
```

we can write

```
Random r = new Random();
```

rather than

```
java.util.Random r = new java.util.Random();
```

You can import all the classes in a package at once with a notation like

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

The package `java.lang` is special; every program behaves as if it started with

```
import java.lang.*;
```

whether it does or not. You can define your own packages, but defining packages is an advanced topic beyond the scope of what's required for this course.

The `import` statement doesn't really "import" anything. It just introduces a convenient abbreviation for a fully-qualified class name. When a class needs to use another class, all it has to do is use it. The Java compiler will know that it is supposed to be a class by the way it is used, will import the appropriate `.class` file, and will even compile a `.java` file if necessary. (That's why it's important for the name of the file to match the name of the class). For example, here is a simple program that uses two classes:

```
class HelloTest {
    public static void main(String[] args) {
        Hello greeter = new Hello();
        greeter.speak();
    }
}
class Hello {
    void speak() {
        System.out.println("Hello World!");
    }
}
```

Put each class in a separate file (`HelloTest.java` and `Hello.java`). Then try this:

```
javac HelloTest.java
java Hello
```

You should see a cheery greeting. If you type `ls` you will see that you have both `HelloTest.class` and `Hello.class` even though you only asked to compiled `HelloTest.java`. The Java compiler figured out that class `HelloTest` uses class `Hello` and automatically compiled it. Try this to learn more about what's going on:

```
rm -f *.class
javac -verbose HelloTest.java
java Hello
```

Values, Objects, and Pointers

It is sometimes said that Java doesn't have pointers. That is not true. In fact, objects can *only* be referenced with pointers. More precisely, variables can hold primitive values (such as integers or floating-point numbers) or *references* (pointers) to objects. A variable cannot hold an object, and you cannot make a pointer to a primitive value. Since you don't have a choice, Java doesn't have a special notation like C++ does to indicate when you want to use a pointer.

There are exactly eight primitive types in Java, `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. Most of these are similar to types with the same name in C++. We mention only the differences.

A `boolean` value is either `true` or `false`. You cannot use an `integer` where a `boolean` is required (e.g. in an `if` or `while` statement) nor is there any automatic conversion between `boolean` and `integer`.

A `char` value is 16 bits rather than 8 bits, as it is in C or C++, to allow for all sorts of international alphabets. As a practical matter, however, you are unlikely to notice the difference. The `byte` type is an 8-bit signed integer (like `signed char` in C or C++).

A `short` is 16 bits and an `int` is 32 bits, just as in C or C++ on most modern machines (in C++ the size is machine-dependent, but in Java it is guaranteed to be 32 bits). A Java `long` is *not* the same as in C++; it is 64 bits long--twice as big as a normal `int`--so it can hold any value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The types `float` and `double` are just like in C++: 32-bit and 64-bit floating point.

As in C++, *objects* are *instances* of *classes*. There is no prefix `*` or `&` operator or infix `->` operator.

As an example, consider the class declaration (which is the same in C++ and in Java)

```
class Pair { int x, y; }
```

C++	Java
`Pair default;`	`Pair default = new Pair();`
`Pair *p, *q, *r;`	`Pair p, q, r;`
`default.x = 0;`	`default.x = 0;`
`p = new Pair;`	`p = new Pair();`
`p -> y = 5;`	`p.y = 5;`
`q = p;`	`q = p;`
`r = &default;`	*not possible*

As in C or C++, arguments to a Java procedure are passed ``by value'':

```
void f() {
    int n = 1;
    Pair p = new Pair();
    p.x = 2; p.y = 3;
    System.out.println(n);     // prints 1
    System.out.println(p.x);   // prints 2
    g(n,p);
    System.out.println(n);     // still prints 1
    System.out.println(p.x);   // prints 100
}
void g(int num, Pair ptr) {
    System.out.println(num);   // prints 1
    num = 17;                  // changes only the local copy
    System.out.println(num);   // prints 17

    System.out.println(ptr.x);// prints 2
    ptr.x = 100;               // changes the x field of caller's Pair
    ptr = null;                // changes only the local ptr
}
```

The formal parameters `num` and `ptr` are local variables in the procedure `g` initialized with *copies* of the values of `n` and `p`. Any changes to `num` and `ptr` affect only the copies. However, since `ptr` and `p` point to the same object, the assignment to `ptr.x` in `g` changes the value of `p.x`.

Unlike C++, Java has no way of declaring *reference* parameters, and unlike C++ or C, Java has no way of creating a pointer to a (non-object) value, so you can't do something like this

```
/* C or C++ */
void swap1(int *xp, int *yp) {
    int tmp;
    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}
int foo = 10, bar = 20;
swap1(&foo, &bar);  /* now foo==20 and bar==10 */

// C++ only
void swap2(int &xp, int &yp) {
    int tmp;
    tmp = xp;
    xp = yp;
    yp = tmp;
}
int this_one = 88, that_one = 99;
swap2(this_one, that_one);  // now this_one==99 and that_one==88
```

You'll probably miss reference parameters most in situations where you want a procedure to return more than one value. As a work-around you can return an object or array or pass in a pointer to an object. See Section 2.6 on page 36 of the Java book for more information.

Garbage Collection

New objects are create by the `new` operator in Java just like C++ (except that an argument list is required after the class name, even if the constructor for the class doesn't take any arguments so the list is empty). However, there is no `delete` operator. The Java system automatically deletes objects when no references to them remain. This is a much more important convenience than it may at first seem. `delete` operator is extremely error-prone. Deleting objects too early can lead to *dangling* reference, as in

```
p = new Pair();
// ...
```

```
q = p;
// ... much later
delete p;
q -> x  = 5; // oops!
```

while deleting them too late (or not at all) can lead to *garbage*, also known as a *storage leak*.

Static, Final, Public, and Private

Just as in C++, it is possible to restrict access to members of a class by declaring them *private*, but the syntax is different.
In C++:

```
class C {
    private:
        int i;
        double d;
    public:
        int j;
        void f() { /*...*/ }
}
```

In Java:

```
class C {
    private    int i;
    public     int j;
    private    double d;
    public     void f() { /* ... */ }
}
```

As in C++, private members can only be accessed from inside the bodies of methods (function members) of the class, not ``from the outside." Thus if `x` is an instance of `C`, `x.i` is not legal, but `i` can be accessed from the body of `x.f()`. (`protected` is also supported; it means the same thing as it does in C++). The default (if neither `public` nor `private` is specified) is that a member can be accessed from anywhere in the same *package*, giving a facility rather like ``friends" in C++. You will probably be putting all your classes in one package, so the default is essentially `public`, but you should not rely on this default. *In this course, **every** member must be declared `public`, `protected`, or `private`.*

The keyword `static` also means the same thing in Java as C++, which not what the word implies: Ordinary members have one copy per instance,

whereas a `static` member has only one copy, which is shared by all instances. In effect, a `static` member lives in the class itself, rather than instances.

```
class C {
    int x = 1;  // by the way, this is ok in Java but not C++
    static int y = 1;
    void f(int n) { x += n; }
    static int g() { return ++y; }
}
C p = new C();
C q = new C();
p.f(3);
q.f(5);
System.out.println(p.x);  // prints 4
System.out.println(q.x);  // prints 6
System.out.println(C.y);  // prints 1
System.out.println(p.y);  // means the same thing
System.out.println(C.g());// prints 2
System.out.println(q.g());// prints 3
```

Static members are often used instead of global variables and functions, which do not exist in Java. For example,

```
Math.tan(x);            // tan is a static method of class Math
Math.PI;                // a static "field" of class Math with value 3.14159...
Integer.parseInt("10"); // used in the sorting example
```

The keyword `final` is roughly equivalent to `const` in C++: `final` fields cannot be changed. It is often used in conjunction with `static` to defined named constants.

```
class Card {
    public int suit = CLUBS;     // default
    public final static int CLUBS = 1;
    public final static int DIAMONDS = 2;
    public final static int HEARTS = 3;
    public final static int SPADES = 4;
}
Card c = new Card();
c.suit = Card.SPADES;
```

Each `Card` has its own suit. The value `CLUBS` is shared by all instances of `Card` so it only needs to be stored once, but since it's `final`, it doesn't need to be stored at all!

Arrays

In Java, arrays are objects. Like all objects in Java, you can only point to them, but unlike a C++ variable, which is treated like a pointer to the first element of the array, a Java array variable points to the whole object. There is no way to point to a particular slot in an array.

Each array has a read-only (final) field `length` that tells you how many elements it has. The elements are numbered starting at zero as in C++: `a[0] ... a[a.length-1]`. Once you create an array (using `new`), you can't change its size. If you need more space, you have to create a new (larger) array and copy over the elements (but see the library class `Vector` below).

```
int x = 3;    // a value
int[] a;      // a pointer to an array object; initially null
int a[];      // means exactly the same thing (for compatibility with C)
a = new int[10]; // now a points to an array object
a[3] = 17;    // accesses one of the slots in the array
a = new int[5]; // assigns a different array to a
              // the old array is inaccessible (and so
              // is garbage-collected)
int[] b = a; // a and b share the same array object
System.out.println(a.length); // prints 5
```

Strings

Since you can make an array of anything, you can make an an array of `char` or an an array of `byte`, but Java has something much better: the type `String`. The `+` operator is *overloaded* on Strings to mean concatenation. What's more, you can concatenate *anything* with a string; Java automatically converts it to a string. Built-in types such as numbers are converted in the obvious way. Objects are converted by calling their `toString()` methods. Library classes all have `toString` methods that do something reasonable. You should do likewise for all classes you define. This is great for debugging.

```
String s = "hello";
String t = "world";
System.out.println(s + ", " + t);       // prints "hello, world"
System.out.println(s + "1234");         // "hello1234"
System.out.println(s + (12*100 + 34));  // "hello1234"
System.out.println(s + 12*100 + 34);    // "hello120034" (why?)
System.out.println("The value of x is " + x);  // will work for any x
```

```
System.out.println("System.out = " + System.out);
            // "System.out = java.io.PrintStream@80455198"
String numbers = "";
for (int i=0; i<5; i++)
    numbers += " " + i;
System.out.println(numbers);            // " 1 2 3 4 5"
```

Strings have lots of other useful operations:

```
String s = "whatever", t = "whatnow";
s.charAt(0);             // 'w'
s.charAt(3);             // 't'
t.substring(4);          // "now" (positions 4 through the end)
t.substring(4,6);        // "no"  (positions 4 and 5, but not 6)
s.substring(0,4);        // "what" (positions 0 through 3)
t.substring(0,4);        // "what"
s.compareTo(t);          // a value less than zero
                         // s precedes t in "lexicographic"
                         // (dictionary) order
t.compareTo(s);          // a value greater than zero (t follows s)
t.compareTo("whatnow"); // zero
t.substring(0,4) == s.substring(0,4);
                         // false (they are different String objects)
t.substring(0,4).equals(s.substring(0,4));
                         // true (but they are both equal to "what")
t.indexOf('w');          // 0
t.indexOf('t');          // 3
t.indexOf("now");        // 4
t.lastIndexOf('w');      // 6
t.endsWith("now");       // true
```

and more.

You can't modify a string, but you can make a string variable point to a new string (as in `numbers += " " + i;`). See `StringBuffer` if you want a string you can scribble on.

Constructors and Overloading

A constructor is like in C++: a method with the same name as the class. If a constructor has arguments, you supply corresponding values when using

new. Even if it has no arguments, you still need the parentheses (unlike C++). There can be multiple constructors, with different numbers or types of arguments. The same is true for other methods. This is called *overloading*. Unlike C++, you cannot overload operators. The operator `+' is overloaded for strings and (various kinds of) numbers, but user-defined overloading is not allowed.

```
class Pair {
    int x, y;
    Pair(int u, int v) {
        x = u;  // the same as this.x = u
        y = v;
    }
    Pair(int x) {
        this.x = x;  // not the same as x = x!
        y = 0;
    }
    Pair() {
        x = 0;
        y = 0;
    }
}
class Test {
    public static void main(String[] argv) {
        Pair p1 = new Pair(3,4);
        Pair p2 = new Pair();  // same as new Pair(0,0)
        Pair p3 = new Pair;    // error!
    }
}
```

NB: The bodies of the methods have to be defined *in line* right after their headers as shown above. You have to write

```
class Foo {
    double square(double d) { return d*d; }
};
```

rather than

```
class Foo {
    double square(double);
};
double Foo::square(double d) { return d*d; }
    // ok in C++ but not in Java
```

Inheritance, Interfaces, and Casts

In C++, when we write

```
class Derived : public Base { ... }
```

we mean two things:

- A `Derived` can do anything a `Base` can, and perhaps more.
- A `Derived` does things the way a `Base` does them, unless specified otherwise.

The first of these is called *interface inheritance* or *subtyping* and the second is called *method inheritance*. In Java, they are specified differently.

Method inheritance is specified with the keyword `extends`.

```
class Base {
    int f() { /* ... */ }
    void g(int x) { /* ... */ }
}
class Derived extends Base {
    void g(int x) { /* ... */ }
    double h() { /* ... */ }
}
```

Class `Derived` has three methods: `f`, `g`, and `h`. The method `Derived.f()` is implemented in the same way (the same executable code) as `Base.f()`, but `Derived.g()` *overrides* the implementation of `Base.g()`. We call `Base` the *super class* of `Derived` and `Derived` a *subclass* of `Base`. Every class (with one exception) has exactly one super class (single inheritance). If you leave out the `extends` specification, Java treats it like ``extends Object''. The primordial class `Object` is the lone exception -- it does not extend anything. All other classes extend `Object` either directly or indirectly. `Object` has a method `toString`, so every class has a method `toString`; either it inherits the method from its super class or it overrides it.

Interface inheritance is specified with `implements`. A class implements an *Interface*, which is like a class, except that the methods don't have bodies. Two examples are given by the built-in interfaces`Runnable`and `Enumeration.`

```
interface Runnable {
    void run();
}
interface Enumeration {
```

```
    Object nextElement();
    boolean hasMoreElements();
}
```

An object is `Runnable` if it has a method named `run` that is public[2] and has no arguments or results. To be an `Enumeration`, a class has to have a public method `nextElement()` that returns an `Object` and a public method `hasMoreElements` that returns a `boolean`. A class that claims to `implement` these interfaces has to either inherit them (via `extends`) or define them itself.

```
class Words extends StringTokenizer implements Enumeration, Runnable {
    public void run() {
        for (;;) {
            String s = nextToken();
            if (s == null) {
                return;
            }
            System.out.println(s);
        }
    }
    Words(String s) {
        super(s);
        // perhaps do something else with s as well
    }
}
```

The class `Words` needs methods `run`, `hasMoreElements`, and `nextElement` to meet its promise to implement interfaces `Runnable` and `Enumeration`. It inherits implementations of `hasMoreElements` and `nextElement` from `StringTokenizer`, but it has to give its own implementation of `run`. The `implements` clause tells users of the class what they can expect from it. If `w` is an instance of `Words`, I know I can write

```
w.run();
```

or

```
if (w.hasMoreElements()) ...
```

A class can only extend one class, but it can implement any number of interfaces.

By the way, constructors are not inherited. The call `super(s)` in class `Words` calls the constructor of `StringTokenizer` that takes one `String` argument. If you don't explicitly call `super`, Java automatically calls the super class constructor with no arguments (such a constructor must exist in this

case). Note the call `nextToken()` in `Words.run`, which is short for `this.nextToken()`. Since `this` is an instance of `Words`, it has a `nextToken` method -- the one it inherited from `StringTokenizer`.

A *cast* in Java looks just like a cast in C++: It is a type name in parentheses preceding an expression. We have already seen an example of a cast used to convert between primitive types. A cast can also be used to convert an object reference to a super class or subclass. For example,

```
Words w = new Words("this is a test");
Object o = w.nextElement();
String s = (String)o;
System.out.println("The first word has length " + s.length());
```

We know that `w.nextElement()` is ok, since `Words` implements the interface `Enumeration`, but all that tells us is that the value returned has type `Object`. We cannot call `o.length()` because class `Object` does not have a `length` method. In this case, however, we know that `o` is not just any kind of `Object`, but a `String` in particular. Thus we *cast* `o` to type `String`. If we were wrong about the type of `o` we would get a run-time error. If you are not sure of the type of an object, you can test it with `instanceof` (note the lower case `o'), or find out more about it with the method `Object.getClass()`

```
if (o instanceof String) {
    n = ((String)o).length();
} else {
    System.err.println("Bad type " + o.getClass().getName());
}
```

Exceptions

A Java program should never ``core dump," no matter how buggy it is. If the compiler excepts it and something goes wrong at run time, Java throws an *exception*. By default, an exception causes the program to terminate with an error message, but you can also *catch* an exception.

```
try {
    // ...
    foo.bar();
    // ...
    a[i] = 17;
    // ...
}
catch (IndexOutOfBoundsException e) {
    System.err.println("Oops: " + e);
}
```

The `try` statement says you're interested in catching exceptions. The `catch` clause (which can only appear after a `try`) says what to do if an `IndexOutOfBoundsException`occurs anywhere in the `try` clause. In this case, we print an error message. The `toString()` method of an exception generates a string containing information about what went wrong, as well as a call trace. Because we caught this exception, it will not terminate the program. If some other kind of exception occurs (such as divide by zero), the exception will be thrown back to the caller of this function and if that function doesn't catch it, it will be thrown to that function's caller, and so on back to the `main` function, where it will terminate the program if it isn't caught. Similarly, if the function `foo.bar` throws an `IndexOutOfBoundsException` and doesn't catch it, we will catch it here.

The `catch` clause actually catches `IndexOutOfBoundsException` or any of its subclasses, including `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`, and others. An `Exception` is just another kind of object, and the same rules for inheritance hold for exceptions as any other king of class.

You can define and throw your own exceptions.

```
class SytaxError extends Exception {
    int lineNumber;
    SytaxError(String reason, int line) {
        super(reason);
        lineNumber = line;
    }
    public String toString() {
        return "Syntax error on line " + lineNumber + ": " + getMessage();
    }
}
class SomeOtherClass {
    public void parse(String line) throws SyntaxError {
        // ...
        if (...)
            throw new SyntaxError("missing comma", currentLine);
        //...
    }
    public void parseFile(String fname) {
        //...
        try {
            // ...
            nextLine = in.readLine();
            parse(nextLine);
            // ...
        }
        catch (SyntaxError e) {
            System.err.println(e);
```

```
            }
        }
    }
```

Each function must declare in its header (with the keyword `throws`) all the exceptions that may be thrown by it or any function it calls. It doesn't have to declare exceptions it catches. Some exceptions, such as `IndexOutOfBoundsException`, are so common that Java makes an exception for them (sorry about that pun) and doesn't require that they be declared. This rule applies to `RuntimeException` and its subclasses. You should never define new subclasses of `RuntimeException`.

There can be several `catch` clauses at the end of a `try` statement, to catch various kinds of exceptions. The first one that ``matches'' the exception (i.e., is a super class of it) is executed. You can also add a `finally` clause, which will *always* be executed, no matter how the program leaves the `try` clause (whether by falling through the bottom, executing a `return`, `break`, or `continue`, or throwing an exception).

Threads

Java lets you do several things at once by using *threads*. If your computer has more than one CPU, it may actually run two or more threads simultaneously. Otherwise, it will switch back and forth among the threads at times that are unpredictable unless you take special precautions to control it.

There are two different ways to create threads. I will only describe one of them here.

```
Thread t = new Thread(command); //
t.start();  // t start running command, but we don't wait for it to finish
// ... do something else (perhaps start other threads?)
// ... later:
t.join();  // wait for t to finish running command
```

The constructor for the built-in class `Thread`takes one argument, which is any object that has a method called `run`. This requirment is specified by requiring that `command` *implement* the `Runnable` interface described earlier. (More precisely, `command` must be an instance of a class that `implements Runnable`). The way a thread ``runs'' a command is simply by calling its `run()` method. It's as simple as that!

In project 1, you are supposed to run each command in a separate thread. Thus you might declare something like this:

```
class Command implements Runnable {
    String theCommand;
    Command(String c) {
```

```
            theCommand = c;
        }
        public void run() {
            // Do what the command says to do
        }
    }
```

You can parse the command string either in the constructor or at the start of the `run()` method.

The main program loop reads a command line, breaks it up into commands, runs all of the commands concurrently (each in a separate thread), and waits for them to all finish before issuing the next prompt. In outline, it may look like this.

```
    for (;;) {
        System.out.print("% "); System.out.flush();
        String line = inputStream.readLine();
        int numberOfCommands = // count how many comands there are on the line
        Thread t[] = new Thread[numberOfCommands];
        for (int i=0; i<numberOfCommands; i++) {
            String c = // next command on the line
            t[i] = new Thread(new Command(c));
            t[i].start();
        }
        for (int i=0; i<numberOfCommands; i++) {
            t[i].join();
        }
    }
```

This main loop is in the `main()` method of your main class. It is not necessary for that class to implement `Runnable`.

Although you won't need it for project 1, the next project will require to to synchronize thread with each other. There are two reasons why you need to do this: to prevent threads from interferring with each other, and to allow them to cooperate. You use `synchronized` methods to prevent interference, and the built-in methods `Object.wait()`, `Object.notify()`, `Object.notifyAll()`, and `Thread.yield()` to support cooperation.

Any method can be preceded by the word `synchronized` (as well as `public`, `static`, etc.). The rule is:

No two threads may be executing `synchronized` methods of the same object at the same time.

The Java system enforces this rule by associating a *monitor lock* with each object. When a thread calls a `synchronized` method of an object, it tries to

grab the object's monitor lock. If another thread is holding the lock, it waits until that thread releases it. A thread releases the monitor lock when it leaves the `synchronized` method. If one `synchronized` method of a calls contains a call to another, a thread may have the same lock ``multiple times.'' Java keeps track of that correctly. For example,

```
class C {
    public synchronized void f() {
        // ...
        g();
        // ...
    }
    public synchronized void g() { /* ... */ }
}
```

If a thread calls `C.g()` ``from the outside'', it grabs the lock before executing the body of `g()` and releases it when done. If it calls `C.f()`, it grabs the lock on entry to `f()`, calls `g()` without waiting, and only releases the lock on returning from `f()`.

Sometimes a thread needs to wait for another thread to do something before it can continue. The methods `wait()` and `notify()`, which are defined in class `Object` and thus inherited by all classes, are made for this purpose. They can only be called from within `synchronized` methods. A call to `wait()` releases the monitor lock and puts the calling thread to sleep (i.e., it stops running). A subsequent call to `notify` on the same object wakes up a sleeping thread and lets it start running again. If more than one thread is sleeping, one is chosen arbitrarily[3] no threads are sleeping in this object, `notify()` does nothing. The awakened thread has to wait for the monitor lock before it starts; it competes on an equal basis with other threads trying to get into the monitor. The method `notifyAll` is similar, but wakes up *all* threads sleeping in the object.

```
class Buffer {
    private Queue q;
    public synchronized void put(Object o) {
        q.enqueue(o);
        notify();
    }
    public synchronized Object get() {
        while (q.isEmpty())
            wait();
        return q.dequeue();
    }
}
```

This class solves the so-call ``producer-consumer'' problem (it assumes the `Queue` class has been defined elsewhere). ``Producer'' threads somehow create objects and put them into the buffer by calling `Buffer.put()`, while ``consumer'' threads remove objects from the buffer (using `Buffer.get()`)

and do something with them. The problem is that a consumer thread may call `Buffer.get()` only to discover that the queue is empty. By calling `wait()` it releases the monitor lock and goes to sleep so that producer threads can call `put()` to add more objects. Each time a producer adds an object, it calls `notify()` just in case there is some consumer waiting for an object.

This example is not correct as it stands (and the Java compiler will reject it). The `wait()` method can throw an `InterruptedException` exception, so the `get()` method must either catch it or declare that it `throws InterruptedException` as well. The simplest solution is just to catch the exception and ignore it:

```
class Buffer {
    private Queue q;
    public synchronized void put(Object o) {
        q.enqueue(o);
        notify();
    }
    public synchronized Object get() {
        while (q.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return q.dequeue();
    }
}
```

The method printStackTrace() prints some information about the exception, including the line number where it happened. It is a handy thing to put in a `catch` clause if you don't know what else to put there. *Never* use an empty catch clause. If you violate this rule, you will live to regret it!

There is also a version of `Object.wait()` that takes an integer parameter. The call `wait(n)` will return after `n` milliseconds if nobody wakes up the thread with `notify` or `notifyAll` sooner.

You may wonder why `Buffer.get()` uses `while (q.isEmpty())` rather than `if (q.isEmpty())`. In this particular case, either would work. However, in more complicated situations, a sleeping thread might be awakened for the ``wrong'' reason. Thus it is always a good idea when you wake up to recheck the condition that made to decide to go to sleep before you continue.

Input and Output

Input/Output, as described in Chapter 12 of the Java book, is not as complicated as it looks. You can get pretty far just writing to `System.out`(which is of type `PrintStream`) with methods `println`and `print`. For input, you probably want to wrap the standard input `System.in`in a `BufferedReader`, which provides the handy method `readLine()`

```
BufferedReader in =
    new BufferedReader(new InputStreamReader(System.in));
for(;;) {
    String line = in.readLine();
    if (line == null) {
        break;
    }
    // do something with the next line
}
```

If you want to read from a file, rather than from the keyboard (standard input), you can use `FileReader`, probably wrapped in a `BufferedReader`.

```
BufferedReader in =
    new BufferedReader(new FileReader("somefile"));
for (;;) {
    String line = in.readLine();
    if (line == null) {
        break;
    }
    // do something with the next line
}
```

Similarly, you can use `new PrintWriter(new FileOutputStream("whatever"))` to write to a file.

Other Goodies

The library of pre-defined classes has several other handy tools. See the online manual , particularly java.lang and java.util for more details.

Integer, Character, etc.

Java makes a big distinction between values (integers, characters, etc.) and objects. Sometimes you need an object when you have a value (the next paragraph has an example). The classes `Integer`, `Character`, etc. serve as convenient wrappers for this purpose. For example, `Integer i = new Integer(3)` creates a version of the number 3 wrapped up as an object. The value can be retrieved as `i.intValue`. These classes also serve as

convenient places to define utility functions for manipulating value of the given types, often as `static` methods or defined constants.

```
int i = Integer.MAX_VALUE;          // 2147483648, the largest possible int
int i = Integer.parseInt("123");    // the int value 123
String s = Integer.toHexString(123);// "7b" (123 in hex)
double x = Double.parseDouble("123e-2");
                                    // the double value 1.23
Character.isDigit('3')              // true
Character.isUpperCase('a')          // false
Character.toUpperCase('a')          // 'A'
```

Vector

A `Vector` is like an array, but it grows as necessary to allow you to add as many elements as you like. Unfortunately, there is only one kind of `Vector`--a vector of `Object`. Thus you can insert objects of any type into it, but when you take objects out, you have to use a cast to recover the original type.[4]

```
Vector v = new Vector();     // an empty vector
for (int i=0; i<100; i++)
    v.add(new Integer(i));
// now it contains 100 Integer objects

// print their squares
for (int i=0; i<100; i++) {
    Integer member = (Integer)(v.get(i));
    int n = member.intValue();
    System.out.println(n*n);
}

// another way to do that
for (Iterator i = v.iterator(); i.hasNext(); ) {
    int n = ((Integer)(i.next())).intValue();
    System.out.println(n*n);
}
v.set(5, "hello");           // like v[5] = "hello"
Object o = v.get(3);         // like o = v[3];
v.add(6, "world");           // set v[6] = "world" after first shifting
                             // element v[7], v[8], ... to the right
                             // to make room
v.remove(3);                 // remove v[3] and shift v[4], ... to the
                             // left to fill in the gap
```

Elements of a `Vector` must be objects, not values. That means you can put a `String` or an instance of a user-defined class into a `Vector`, but if you want to put an integer, floating-point number, or character into `Vector`, you have to wrap it:

```
v.add(47);                      // WRONG!
sum += v.get(i);                // WRONG!
v.add(new Integer(47));         // right
sum += ((Integer)v.get(i)).intValue();
                                // ugly, but right
```

The class `Vector` is implemented using an ordinary array that is generally only partially filled. If `Vector` runs out of space, it allocates a bigger array and copies over the elements. There are a variety of additional methods, not shown here, that let you give the implementation advice on how to manage the extra space more efficiently. For example, if you know that you are not going to add any more elements to `v`, you can call `v.trimToSize()` to tell the system to repack the elements into an array just big enough to hold them.

Don't forget to `import java.util.Vector;` or `import java.util.*;` .

Maps and Sets

The interface Map[5] represents a table mapping *keys* to *values*. It is sort of like an array or `Vector`, except that the ``subscripts'' can be *any* objects, rather than non-negative integers. Since `Map` is an interface rather than a class you cannot create instances of it, but you can create instances of the class `HashMap`, which implements `Map` using a hash table.

```
Map table = new HashMap();          // an empty table
table.put("seven", new Integer(7));   // key is the String "seven";
                                    // value is an Integer object
table.put("seven", 7);              // WRONG! (7 is not an object)
Object o = table.put("seven", new Double(7.0));
                                    // binds "seven" to a double object
                                    // and returns the previous value
int n = ((Integer)o).intValue();    // n = 7
table.containsKey("seven");         // true
table.containsKey("twelve");        // false

// print out the contents of the table
for (Iterator i = table.keySet().iterator(); i.hasNext(); ) {
    Object key = i.next();
    System.out.println(key + " -> " + table.get(key));
}
```

```
o = table.get("seven");          // get current binding (a Double)
o = table.remove("seven");       // get current binding and remove it
table.clear();                   // remove all bindings
```

Sometimes, you only care whether a particular key is present, not what it's mapped to. You could always use the same object as a value (or use null), but it would be more efficient (and, more importantly, clearer) to use a Set.

```
System.out.println("What are your favorite colors?");
BufferedReader in =
    new BufferedReader(new InputStreamReader(System.in));
Set favorites = new HashSet();
try {
    for (;;) {
        String color = in.readLine();
        if (color == null) {
            break;
        }
        if (!favorites.add(color)) {
            System.out.println("you already told me that");
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
int n = favorites.size();
if (n == 1) {
    System.out.println("your favorite color is:");
} else {
    System.out.println("your " + n + " favorite colors are:");
}
for (Iterator i = favorites.iterator(); i.hasNext(); ) {
    System.out.println(i.next());
}
```

StringTokenizer

A `StringTokenizer` is handy in breaking up a string into words separated by white space (or other separator characters). The following example is from the Java book:

```
String str = "Gone, and forgotten";
```

```
StringTokenizer tokens = new StringTokenizer(str, " ,");
while (tokens.hasMoreTokens())
    System.out.println(tokens.nextToken());
```

It prints out

```
Gone
and
forgotten
```

The second arguement to the constructor is a String containing the characters that such be considered separators (in this case, space and comma). If it is omitted, it defaults to space, tab, return, and newline (the most common ``white-space'' characters).

There is a much more complicated class `StreamTokenizer` for breaking up an input stream into tokens. Many of its features seem to be designed to aid in parsing the Java langauge itself (which is not a surprise, considering that the Java compiler is written in Java).

Other Utilities

The random-number generator `Random` was presented above. See Chapter 13 of the Java book for information about other handy classes.

---

[1]Throughout this tutorial, examples in C++ are shown in `green` and examples in Java are shown in `blue`. This example could have been in either `green`*or*blue!

[2] All the members of an `Interface` are implicitly public. You can explicitly declare them to be `public`, but you don't have to, and you shouldn't.

[3] as a practical matter, it's probably the one that has been sleeping the longest, but you can't depend on that

[4]Interface Iterator was introduced with Java 1.2. It is a somewhat more convenient version of the older interface `Enumeration` discussed earlier.

[5]Interfaces `Map` and `Set` were introduced with Java 1.2. Earier versions of the API contained only `Hashtable`, which is similar toHashMap.

---

solomon@cs.wisc.edu

Wed Sep 1 13:18:17 CDT 1999