Generally, behavior in Java is much more carefully specified and controlled than in C/C++ — this supports portability and security, and reduces ambiguity.

So, some important points from the reading… (you are responsible for everything in the reading! Including stuff I don't talk about here. Some of it won't be important until later in the semester.)

Comments

In addition to the kinds of comments you know from C++ (single-line beginning with // and blocks wrapped with /* … */) Java offers "doccomments" which are intended to be read by automated documentation generators (like **javadoc**). These start with /** and end with */ and may contain lines beginning with @, indicating a special directive. The Java API documentation, for example, is created automatically this way. Among the special directives: @author, @see, @version, @param, @return.

I will expect you to "decorate" your code with doccomments *and* produce a readable version in HTML. Eclipse can do a lot of this work for you,. Check out the wikipedia page http://en.wikipedia.org/wiki/Javadoc for more examples and instructions, and this tutorial http://www.codejava.net/ides/eclipse/how-to-generate-javadoc-in-eclipse for how to take advantage of Eclipse's tools for javadoc generation. Note that other programs out there can read doccomments for a variety of purposes.

Types: Primitive and Reference

As we discussed last time, Java is much more careful about types than C is (C++ is more careful than C, but not as careful as Java). There are two basic kinds of types in Java: *primitive* and *reference*. The primitive types are the ones we think of as "built-in" in C/C++: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, **double**. Each of these has a strictly-specified range of values—definitely independent of the machine on which code might be running. Primitive types are *always* used and passed by value—there is no mechanism analogous to passing pointers in C++.

Everything else in Java is a *reference*. Specifically, anything that's not a primitive value is an *object*, and objects are manipulated through references—a reference is like a 'handle' for an object. References are passed by value, too, but when you know the value of a reference, you can change the value of the object attached to it. Some books say that a reference is "like a pointer in C" but that doesn't give much understanding—references in Java are treated much more carefully. (In C, a pointer can point to pretty much anything, but Java is very sensitive to the type of a reference and will only let it 'handle' a small handful of object types.)

Java also makes it easy to treat primitive values as objects—roughly, it automatically converts primitive values into objects (e.g. of type **Integer** or **Double**), and back, when necessary.

Also note that the == and != operators work slightly differently on primitives and references. When comparing primitives, it's about equality of value (what you would expect). When comparing references, it's about equality of reference—do the two references name the same object? If we want to know whether two *objects* are equal, we have to ask them with the **equals()** method, though the behavior of this method varies, so read the documentation!

If (s.equals(t)) {…}

Garbage Collection

There is nothing like “delete” or “free” in Java!! Instead, the JVM watches to see when a chunk of memory is no longer referenced, and then “collects” it. But it could be that a collected object could still have (say) a file or network connection open; if you have classes like this, you can provide a finalize() method which will be called exactly once before garbage collection (but you have no control over when, exactly, this happens).

Arrays and Strings

So what about arrays, or strings? These are not on the list of primitive types. So yes, they are objects, though the compiler gives us some extra help with them, so sometimes they 'feel' like primitive types. For example, we can declare an array like this:

```
int [] arrayOfInts = new int  { 1, 2, 3, 4 };
```

So we get to use the square-bracket notation we're familiar with from C++, but we also need to use **new** here because we're creating a new object. (And note that **arrayOfInts** is technically a *reference* to an array of integers.) We'll talk more about arrays in a few minutes.

Similarly, the compiler will turn string literals into **String** objects automatically

```
String t = "John said: \"I am the walrus...\"";
```

The + and += operators are also overloaded to work with strings:

```
String quote = "Four score and " + "seven years ago,";
quote +=  " our" + " fathers" +  " brought...";
```

Loops

Java has the three 'usual' loops (**do-while**, **while**, and **for**), but also has an “enhanced” **for** loop that makes it very easy to process the elements of an array (and of some other kinds of collections, which we'll talk about later). For example:

```
int [] arrayOfInts = new int [] { 1, 2, 3, 4 };
int sum = 0;
for( int i : arrayOfInts ) {
        System.out.println( i );
        sum += i;
}
System.out.println(sum);
```

Note that this only lets you process the *values* of the array; changing the contents of the array requires a different approach.

Exceptions

We saw an exception at the end of last class—what were we doing?

We won't need to to work with this much right away, but as we get into writing code that engages heavily with its environment (like, say, the Internet), our code will need to be able to deal with a range of possible error conditions.

Java has very robust error handling (in general, Java code is going to run in a very wide variety of environments, so this robustness is necessary to prevent awkward crashes. The main technique Java uses is *exceptions* (which are also present in C++, though it's harder to use). Conceptually, what is an exception?

I usually think of it as: an error condition that the programmer can anticipate, but which totally disrupts the "normal" flow of execution. Our example, of expecting two `int` values from the user, is a good example. It's pretty easy to predict that the user might not enter a proper int value, but it's *not at all* easy to deal with that happening. That's a classic exception. What other kinds of "errors" might be exceptions?

The main power of this technique is that it allows program control to transfer to error-handling code immediately and unconditionally when an error condition arises. In the lingo, an exception is "thrown" to the error-handling code which "catches" it. As the book says, "Control can be passed a long distance from a deeply nested routine and handled in a single location when that is desirable, or an error can be handled immediately at its source."

Technically, an exception in Java is an instance of the class `java.lang.Exception` or one of its subclasses. These objects aren't generally very complicated; their primary use is to they indicate that something unusual happened; the name of the exception usually indicates something about what it was that happened, sometimes the object will contain a more detailed message about what caused the exception.

One subtype of Exception is an exception to what I'm going to say here: RuntimeExceptions. More on that in a bit.

What's interesting about Exception objects is how they travel. If an error condition arises (one that needs to be handled by special-purpose code), then the code will "throw an exception" with a piece of code like

```
throw new SecurityException( "Access to file : "+ s +" denied.");
```

(that String argument is optional)

Any method that could throw an exception must name the exceptions it throws in its signature:

```
public File openFile( String filename) throws SecurityException { ...
        ...
        throw new SecurityException( "Access to file : "+ s +" denied.");
        ...
}
```

This allows the compiler to make sure that all exceptions that could arise in a program will be dealt with somewhere—a crucial part of robust error-handling!

Throwing exceptions is easy; it's catching and handling them that's more complex. If you are using a method which throws an exception, you have two basic options:

1. You can 'ignore' it, that is, choose not to handle it. In this case, your method must itself say that it throws the kind of exception you're not handling:

```
public void foo (String filename) throws SecurityException {
        .
        .
        .
// the openFile method could throw an exception; but I don't know how to deal with it

        File file = openFile(filename);

        // do stuff with file here
}
```

In that case, any method which calls your method (foo) will itself either let the exception "bubble up" or...

2. "catch" the exception. If your method is the right place to deal with the error condition, then you'll use the try/catch statement. The code which could produce an exception (or exceptions) is put in a "try" block, which is followed by one or more "catch" blocks.

```
try {
        // this could throw an exception; if so, it will get caught below

        openFile(filename);

        // do stuff with file here
}
catch (SecurityException e) {

        // do whatever it takes
}
```

Code in a `try` block has one interesting property: any code after the exception is thrown will *not* be executed.

There's a lot more to this; we'll cover it as necessary. At minimum, you should know to make sure your methods throw whatever exceptions they must; at the very least, all exceptions need to be caught in `main()` even if at the moment the "handling" is clunky.

Let's look at our `InputMismatchException` example…

So, `RuntimeException`s are exceptions that the JVM may throw if your code does something it's not allowed to (like access an array element that doesn't exist, or cast an object in a way it can't be cast) but that couldn't have been caught by the compiler. These are "unchecked" exceptions, which means that you're not required to handle them in your code, though you of course may. So, you don't need to use a `try` block every time you attempt to access an array element, for example.

Arrays in More Detail

As the book says, "Unlike other languages, however, arrays in Java are true, first-class objects. An array is an instance of a special Java array class and has a corresponding type in the type system. This means that to use an array, as with any other object, we first declare a variable of the appropriate type and then use the new operator to create an instance of it."

In particular:

(1) Java creates a new array class every time we declare a new type of array. So

```
int[] arrayOfInts;
```

and

```
double[] arrayOfDoubles;
```

silently cause the compiler to create two classes

(2) Java lets us use [] to access array elements, and lets us use special forms of new to create array objects.

Most of the time, we don't really have to think about arrays as objects, but sometimes it's useful to treat them that way.

We can initialize arrays by specifying their size, e.g.

```
int number;
Strings[] someStrings;
...
someStrings = new String [ number + 2 ];
```

We can also initialize arrays by providing a list of values, eg.

```
int [] primes = { 2, 3, 5, 7, 7+4 };
```

Note that new is not necessary — that's all implicit in the curly braces notation.

We can certainly create arrays of objects, like

```
String [] names  = new String [4];
```

What does this array contain?

A bunch of null references to Strings. We can start populating it, e.g.

```
names[0] = "Jane";
```

So note: declaring the array doesn't allocate storage for the objects "in" the array; only for the *references* to those eventual contents.

This would have the same effect:

```
String names[] = {null, null, null, null};
```

Java arrays have two main differences from C++ arrays:

First, you can get the length (number of elements allocated) with the length member:

```
names.length; // is 4
```

Second, if you try to access a nonexistent element (index less than 0 or >= length), you'll get a runtime exception ArrayOutOfBoundsException — because of Java's emphasis on security, the JVM will simply not allow you to do this.

And remember you can use the “enhanced for” loop to process all the values in an array!

Classes and Objects (Chapter 5)

The basics of working with classes and objects in Java is very similar to C++; you should read Chapter 5 carefully, but *most* of what's there isn't new. Some important exceptions:

Enumerations

These exist in C/C++ too, but they're more useful in Java. An enumeration is simply a list of values, with an order imposed by the order in which you list them:

```
enum Direction {North, South, East, West};
```

You don't need quotes, etc. You can access these as `Direction.North`, `Direction.South`, etc. When you declare an enum, the compiler automatically creates a class—in this example, named `Direction` and gives it `static final` members `North`, `South`, etc.

The compiler will be very picky about these; it will only allow these particular values, ever. And you can use these values in `if/then` or `switch` statements. Even groovier, the compiler provides a few helpful methods. You can get an array of all the possible values by invoking

```
Direction[] directions = Direction.values();
```

(which you can then use in an enhanced for!)

And you can compare the values using

```
Direction dir = Direction.North;
Direction anotherDir = Direction.West;

if ( dir.compareTo( anotherDir) < 0 ) // true
        doSomething();
```

Let’s pause here and look more closely at all the `HelloJava` code from chapter 2 that we briefly played with on the first day of class. You should find that you understand a *lot* of what’s going on, though there will be plenty of gaps right now.

Static and Final

While static methods and variables in Java are very similar to those in C++ (what are they?), Java has one variant

```
public MyClass {
        static final int MYCONSTANT = 347;
```

So the keyword `final` tells the compiler that the value of this variable will *never* be changed, — i.e. it's a constant.

this and 'shadowing'

Like C++, this is used to refer to the current object. (Though note that in C++, this is a pointer; in Java it's a reference.) Mainly this is used in a method where an argument name matches an instance variable.

```
public int set(int size) {
        this.size = size;
```

In Java, we say that the argument name "shadows" the instance variable name — it's a more local scope, so the instance variable is "in the shadow" of the argument, but we can still access it via this.

Local Variables

In general, the compiler initializes variables to sensible values depending on their type (see the book). But local variables—those declared inside a method—are not automatically initialized. The compiler will force you to initialize them before you use them (i.e. they must first be used on the left of an assignment expression.)

Overloading

What is overloading? What can be overloaded?

Methods are overloaded when you write two methods with the same name but different arguments (type and/or number). This is a very useful thing. e.g. the function System.out.print() is heavily overloaded

```
void print(int i);
void print(double d);
void print (String s);
...
```

When you call an overloaded method, the compiler decides which one is the right version to use. (Note that this gets a bit subtle when classes and subclasses are involved—how to pick the "best" one?)

Constructors can also be overloaded:

```
class Date {
        long time;
        Date() {
                time = currentTime();
        }

        Date( String date ) {
                time = parseDate( date );
        }
... }
```

and the compiler will pick the right one.

But: in general, each constructor method will probably be doing essentially the same work, and having to retype that for each version is not only a hassle, it's a recipe for broken code.

```
class Car {
	String model;
	int doors;

	static final int DEFAULT_DOORS = 4;

	Car( String model, int doors ) {
		this.model = model;
		this.doors = doors;
			// other, complicated setup ...
	}

	Car( String model ) {
		// if we don't get a doors parameter, we want to just use DEFAULT_DOORS
		// this.model = model;
		// this.doors = DEFAULT_DOORS;
			// other, complicated setup ...

		this( model, DEFAULT_DOORS);
	}
```

Important! This special use of `this` can *only* happen on the first line of a constructor. Other stuff can happen afterwards.

Assertions

The Java compiler offers us programmers another method for checking for dumb errors while we're writing code—this isn't at all about error handling but rather a way to prevent programming mistakes. In essence, we can "decorate" our code with statements—assertions—about the assumptions we're making at various points, and if those assumptions go wrong, we'll be alerted. As the book says, "Assertions can be used to "sanity check" your code anywhere you believe certain conditions are guaranteed by correct program behavior." So assertions talk about things that should *always* be true; exceptions deal with bad things that need special handling.

Obviously this is *not* the sort of thing you want to have happen in production code, so by default the JVM disables those alerts. But while you're writing/debugging, they can be very useful.

Here's a little example: suppose you're writing a function to calculate the square root (you normally wouldn't have to do this; the function `Math.sqrt` works just fine).

```
double squareRoot (double radicand)
```

What constraints are on the parameter? If it's less than zero, we're going to have trouble, and there's no easy way around it. So this is a good time to to throw an exception in that case:

```
double squareRoot (double radicand) throws NegativeRadicandException {

        if ( radicand < 0 ) {
                throw NegativeRadicandException("Radicand value is " + radicand);
        } else {
                double result = 0.0;

                // calculate stuff

                assert( result > 0.0 ) : "Square root calculated negative result";
                return result;
        }
}
```

So the exception is used to communicate with whoever's using this code — they somehow sent a bad argument — and the assertion is used to help me make sure my code is working properly. (Note that we also need to define the NegativeRadicandException class, which the book illustrates on p 113.)

```
int[] arrayOfInts = new int[4];

int[] arrayOfInts = {1,2,3,4};
```