JAVA Programming Manual

What is Java?

Java (with a capital J) is a high-level, third generation programming language, like C, Fortran, Smalltalk, Perl, and many others. You can use Java to write computer applications that crunch numbers, process words, play games, store data or do any of the thousands of other things computer software can do.

Compared to other programming languages, Java is most similar to C. However although Java shares much of C's syntax, it is not C. Knowing how to program in C or, better yet, C++, will certainly help you to learn Java more quickly, but you don't need to know C to learn Java. Unlike C++ Java is not a superset of C. A Java compiler won't compile C code, and most large C programs need to be changed substantially before they can become Java programs.

What's most special about Java in relation to other programming languages is that it lets you write special programs called *applets* that can be downloaded from the Internet and played safely within a web browser. Traditional computer programs have far too much access to your system to be downloaded and executed willy-nilly. Although you generally trust the maintainers of various ftp archives and bulletin boards to do basic virus checking and not to post destructive software, a lot still slips through the cracks. Even more dangerous software would be promulgated if any web page you visited could run programs on your system. You have no way of checking these programs for bugs or for out-and-out malicious behavior before downloading and running them.

Java solves this problem by severely restricting what an applet can do. A Java applet cannot write to your hard disk without your permission. It cannot write to arbitrary addresses in memory and thereby introduce a virus into your computer. It should not crash your system.

Java is a Platform

Java (with a capital J) is a platform for application development. A platform is a loosely defined computer industry buzzword that typically means some combination of hardware and system software that will mostly run all the same software. For instance PowerMacs running System 7.5 would be one platform. DEC Alphas running Windows NT would be another.

There's another problem with distributing executable programs from web pages. Computer programs are very closely tied to the specific hardware and operating system they run. A Windows program will not run on a computer that only runs DOS. A Mac application can't run on a Unix workstation. VMS code can't be executed on an IBM mainframe, and so on. Therefore major commercial applications like Microsoft Word or Netscape have to be written almost independently for all the different platforms they run on. Netscape is one of the most cross-platform of major applications, and it still only runs on a minority of platforms.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it

produces a special format called *byte code*. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. This byte code fragment means the same thing on a Solaris workstation as it does on a Macintosh PowerBook. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program java (with a little j). Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java.

All these pieces, the javac compiler, the java interpreter, the Java programming language, and more are collectively referred to as Java.

Java is Simple

Java was designed to make it much easier to write bug free code. According to Sun's Bill Joy, shipping C code has, on average, one bug per 55 lines of code. The most important part of helping programmers write bug-free code is keeping the language simple.

Java has the bare bones functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features. Despite its simplicity Java has considerably more functionality than C, primarily because of the large class library.

Because Java is simple, it is easy to read and write. Obfuscated Java isn't nearly as common as obfuscated C. There aren't a lot of special cases or tricks that will confuse beginners.

About half of the bugs in C and C++ programs are related to memory allocation and deallocation. Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and deallocation. The C library memory allocation functions `malloc()` and `free()` are gone as are C++'s destructors.

Java is an excellent teaching language, and an excellent choice with which to learn programming. The language is small so it's easy to become fluent. The language is interpreted so the compile-run-link cycle is much shorter. The runtime environment provides automatic memory allocation and garbage collection so there's less for the programmer to think about. Java is object-oriented unlike Basic so the beginning programmer doesn't have to unlearn bad programming habits when moving into real world projects. Finally, it's very difficult (if not quite impossible) to write a Java program that will crash your system, something that you can't say about any other language.

Java is Object-Oriented

Object oriented programming is the catch phrase of computer programming in the 1990's. Although object oriented programming has been around in one form or another since the Simula language was invented in the 1960's, it's really begun to take hold in modern GUI environments like Windows, Motif and the Mac. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods.

Object oriented programming is alleged to have a number of advantages including:

- Simpler, easier to read programs
- More efficient reuse of code
- Faster time to market
- More robust, error-free code

In practice object-oriented programs have been just as slow, expensive and buggy as traditional non-object-oriented programs. In large part this is because the most popular object-oriented language is C++. C++ is a complex, difficult language that shares all the obfuscation of C while sharing none of C's efficiencies. It is possible in practice to write clean, easy-to-read Java code. In C++ this is almost unheard of outside of programming textbooks.

Java is Platform Independent

Java was designed to not only be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor architectures Java is compiled to an intermediate form called byte-code. A Java program never really executes natively on the host machine. Rather a special native program called the Java interpreter reads the byte code and executes the corresponding native machine instructions. Thus to port Java programs to a new platform all that is needed is to port the interpreter and some of the library routines. Even the compiler is written in Java. The byte codes are precisely defined, and remain the same on all platforms.

The second important part of making Java cross-platform is the elimination of undefined or architecture dependent constructs. Integers are always four bytes long, and floating point variables follow the IEEE 754 standard for computer arithmetic exactly. You don't have to worry that the meaning of an integer is going to change if you move from a Pentium to a PowerPC. In Java everything is guaranteed.

However the virtual machine itself and some parts of the class library must be written in native code. These are not always as easy or as quick to port as pure Java programs. This is why for example, there's not yet a version of Java 1.2 for the Mac.

Java is Safe

Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious.

This required the elimination of many features of C and C++. Most notably there are no pointers in Java. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted runtime environment. Furthermore Java has strong typing. Variables must be declared, and variables do not change types when you aren't looking. Casts are strictly limited to casts between types that make sense. Thus you can cast an int to a long or a byte to a short but not a long to a boolean or an int to a String.

Java implements a robust exception handling mechanism to deal with both expected and unexpected errors. The worst that an applet can do to a host system is bring down the runtime environment. It cannot bring down the entire system.

Most importantly Java applets can be executed in an environment that prohibits them from introducing viruses, deleting or modifying files, or otherwise destroying data and crashing the host computer. A Java enabled web browser checks the byte codes of an applet to verify that it doesn't do anything nasty before it will run the applet.

However the biggest security problem is not hackers. It's not viruses. It's not even insiders erasing their hard drives and quitting your company to go to work for your competitors. No, the biggest security issue in computing today is bugs. Regular, ordinary, non-malicious unintended bugs are responsible for more data loss and lost productivity than all other factors combined. Java, by making it easier to write bug-free code, substantially improves the security of all kinds of programs.

Java is High Performance

Java byte codes can be compiled on the fly to code that rivals C++ in speed using a "just-in-time compiler." Several companies are also working on native-machine-architecture compilers for Java. These will produce executable code that does not require a separate interpreter, and that is indistinguishable in speed from C++.

While you'll never get that last ounce of speed out of a Java program that you might be able to wring from C or Fortran, the results will be suitable for all but the most demanding applications.

As of May, 1999, the fastest VM, IBM's Java 1.1 VM for Windows, is very close to C++ on CPU-intensive operations that don't involve a lot of disk I/O or GUI work; C++ is itself only a few percent slower than C or Fortran on CPU intensive operations.

It is certainly possible to write large programs in Java. The HotJava browser, the Java Workshop integrated development environment and the javac compiler are large programs that are written entirely in Java.

Java is Multi-Threaded

Java is inherently multi-threaded. A single Java program can have many different threads executing independently and continuously. Three Java applets on the same page can run together with each getting equal time from the CPU with very little extra effort on the part of the programmer.

This makes Java very responsive to user input. It also helps to contribute to Java's robustness and provides a mechanism whereby the Java environment can ensure that a malicious applet doesn't steal all of the host's CPU cycles.

Unfortunately multithreading is so tightly integrated with Java, that it makes Java rather difficult to port to architectures like Windows 3.1 or the PowerMac that don't natively support preemptive multi-threading.

There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs. Nonetheless, in simple programs it's possible to leave multi-threading alone and normally be OK.

Java is Dynamic(ly linked)

Java does not have an explicit link phase. Java source code is divided into .java files, roughly one per each class in your program. The compiler compiles these into .class files containing byte code. Each .java file generally produces exactly one .class file.

(There are a few exceptions we'll discuss later in the semester, non-public classes and inner classes).

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file you're compiling depends on other, non-compiled files the compiler will try to find them and compile them as well. The compiler is quite smart, and can handle circular dependencies as well as methods that are used before they're declared. It also can determine whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of differing classes that it's never seen before without recompilation.

Furthermore, Java .class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large runtime libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

Java is Garbage Collected

You do not need to explicitly allocate or deallocate memory in Java. Memory is allocated as needed, both on the stack and the heap, and reclaimed by the *garbage collector* when it is no longer needed. There's no `malloc()`, `free()`, or destructor methods.

There are constructors and these do allocate memory on the heap, but this is transparent to the programmer.

Most Java virtual machines use an inefficient, mark and sweep garbage collector.

The Hello World Application

```
class HelloWorld {

  public static void main (String args[]) {

    System.out.println("Hello World!");

  }

}
```

Hello World is very close to the simplest program imaginable. When you successfully compile and run it, it prints the words "Hello World!" on your display. Although it doesn't teach very much programming, it gives you a chance to learn the mechanics of typing and compiling code. The goal of this program is not to learn how to print words to the terminal. It's to learn how to type, save and compile a program. This is often a non-trivial procedure, and there are a lot of things that can go wrong even if your source code is correct.

To write the code you need a text editor. You can use any text editor like Notepad, Brief, emacs or vi. Personally I use BBEdit on the Mac and TextPad on Windows.

You should not use a word processor like Microsoft Word or WordPerfect since these save their files in a proprietary format and not in pure ASCII text. If you absolutely must use one of these, be sure to tell it to save your files as pure text. Generally this will require using Save As... rather

than Save. If you have an integrated development environment like BlueJ 1.0 or Borland JBuilder, that will include a text editor you can use to edit Java source code. It will probably change your words various colors and styles for no apparent reason. Don't worry about this yet. As long as the text is correct you'll be fine.

When you've chosen your text editor, type or copy the above program into a new file. For now type it exactly as it appears here. Like C and unlike Fortran, Java is case sensitive so `System.out.println` is not the same as `system.out.println`. `CLASS` is not the same as `class`, and so on.

However, white space is not significant except inside string literals. The exact number of spaces or tabs you use doesn't matter.

Save this code in a file called `HelloWorld.java`. Use exactly that name including case. Congratulations! You've written your first Java program.

Saving files on Windows

Some Windows text editors including Notepad add a three letter ".txt" extension to all the files they save without telling you. Thus you can unexpectedly end up with a file called "HelloWorld.java.txt." This is wrong and will not compile. If your editor has this problem, you should get a better editor. However in the meantime enclose the filename in double quotes in the Save dialog box to make editor save the file with exactly the name you want.

Compiling and Running Hello World

To make sure your Java environment is correctly configured, bring up a command-line prompt and type

```
javac nofile.java
```

If your computer responds with

```
error: Can't read: nofile.java
```

you're ready to begin. If, on the other hand, it responds

```
javac: Command not found
```

or something similar, then you need make sure you have the Java environment properly installed and your PATH configured.

Assuming that Java is properly installed on your system there are three steps to creating a Java program:

1. writing the code
2. compiling the code
3. running the code

Under Unix, compiling and running the code looks like this:

```
% javac HelloWorld.java
% java HelloWorld
Hello World
%
```

Under Windows, it's similar. You just do this in a DOS shell.

```
C:> javac HelloWorld.java
C:> java HelloWorld
Hello World
C:>
```

Notice that you use the `.java` extension when compiling a file, but you do not use the `.class` extension when running a file.

On the Mac, you compile files by dragging and dropping them onto the compiler.

For IDEs, consult your product documentation.

for loops

```
class Count {

  public static void main (String args[]) {

    int i;

    for (i = 0; i < 50; i=i+1) {
      System.out.println(i);
    }

  }

}
```

Example.

Variable declaration inside for loop.

```
class Count {

  public static void main (String args[]) {

    for (int i = 0; i < 50; i = i+1) {
      System.out.println(i);
    }

  }

}
```

Increment and decrement operators

Java has ++ and -- operators like C.

```
class Count {

  public static void main (String args[]) {

    for (int i = 0; i < 50; i++) {
      System.out.println(i);
    }

  }

}
```

decrement operators

```
class Count {

  public static void main (String args[]) {

    for (int i = 50; i > 0; i--) {
      System.out.println(i);
    }

  }

}
```

Print statements

```
class PrintArgs {

  public static void main (String args[]) {

    for (int i = 0; i < args.length; i++) {
      System.out.println(args[i]);
    }

  }

}
% java PrintArgs Hello there!
Hello
there!
```

`System.out.println()` prints its arguments followed by a platform dependent line separator (carriage return (ASCII 13, \r) and a linefeed (ASCII 10, \n) on Windows, linefeed on Unix, carriage return on the Mac)

`System.err.println()` prints on standard err instead.

You can concatenate arguments to `println()` with a plus sign (+), e.g.

```
System.out.println("There are " + args.length + " command line arguments");
```

Using `print()` instead of `println()` does not break the line. For example,

```
System.out.print("There are ");
System.out.print(args.length);
System.out.print(" command line arguments");
System.out.println();
```

`System.out.println()` breaks the line and flushes the output. In general nothing will actually appear on the screen until there's a line break character.

Fibonacci Numbers

```
class Fibonacci {

  public static void main (String args[]) {

    int low = 1;
    int high = 0;

    System.out.println(low);
    while (high < 50) {
      System.out.println(high);
      int temp = high;
      high = high + low;
      low = temp;
    }

  }

}
```

Example

Addition

while loop

Relations

Variable declarations and assignments

Variables and Data Types

There are eight primitive data types in Java:

- **`boolean`**
- `byte`

- `short`
- **`int`**
- *`long`*
- *`float`*
- **`double`**
- **`char`**

However there are only seven kinds of literals, and one of those is not a primitive data type:

- `boolean`: true or false
- `int`: 89, -945, 37865
- `long`: 89L, -945L, 5123567876L
- `float`: 89.5f, -32.5f,
- `double`: 89.5, -32.5, 87.6E45
- `char`: 'c', '9', 't'
- `String`: "This is a string literal"

There are no `short` or `byte` literals.

Strings are a *reference* or *object* type, not a primitive type. However the Java compiler has special support for strings so this sometimes appears not to be the case.

```
class Variables {

  public static void main (String args[]) {

    boolean b = true;
    int low = 1;
    long high = 76L;
    long middle = 74;
    float pi = 3.1415292f;
    double e = 2.71828;
    String s = "Hello World!";

  }

}
```

Comments

Comments in Java are identical to those in C++. Everything between /* and */ is ignored by the compiler, and everything on a single line after // is also thrown away. Therefore the following program is, as far as the compiler is concerned, identical to the first HelloWorld program.

```
// This is the Hello World program in Java
class HelloWorld {

  public static void main (String args[]) {
```

```
    /* Now let's print the line Hello World */
    System.out.println("Hello World!");

  } // main ends here

} // HelloWorld ends here
```

The `/* */` style comments can comment out multiple lines so they're useful when you want to remove large blocks of code, perhaps for debugging purposes. `//` style comments are better for short notes of no more than a line. `/* */` can also be used in the middle of a line whereas `//` can only be used at the end. However putting a comment in the middle of a line makes code harder to read and is generally considered to be bad form.

Comments evaluate to white space, not nothing at all. Thus the following line causes a compiler error:

```
int i = 78/* Split the number in two*/76;
```

Java turns this into the illegal line

```
int i = 78 76;
```

not the legal line

```
int i = 7876;
```

This is also a difference between K&R C and ANSI C.

Command line arguments

```
class printArgs {

  public static void main (String args[]) {

    for (int i = 0; i < args.length; i++) {
      System.out.println(args[i]);
    }

  }

}
```

The name of the class is **not** included in the argument list

Command line arguments are passed in an array of Strings. The first array component is the zeroth.

For example, consider this invocation:

```
% java printArgs Hello There
```

`args[0]` is the string `"Hello"`. `args[1]` is the string `"There"`. `args.length` is 2.

All command line arguments are passed as String values, never as numbers. Later you'll learn how to convert Strings to numbers.

Points

What is a class?

Fields say what an object is

Methods say what an object does

```
class TwoDPoint {

  double x;
  double y;

}
```

To compile this class, put it in a file called TwoDPoint.java and type:

```
% javac TwoDPoint.java
```

What does this produce?

Is this a complete program? Can you run it?

Other Examples:

- ThreeDPoint
- Student
- Angle

Objects

What is an object.

Create objects with the `new` keyword followed by a constructor. For example, the following program creates a `TwoDPoint` object and prints its fields:

```
class OriginPrinter {

  public static void main(String[] args) {

    TwoDPoint origin; // only declares, does not allocate

    // The constructor allocates and usually initializes the object
    origin = new TwoDPoint();

    // set the fields
    origin.x = 0.0;
    origin.y = 0.0;

    // print the two-d point
```

```
    System.out.println("The origin is at " + origin.x + ", " + origin.y);

  }  // end main

} // end OriginPrinter
```
The . is the *member access separator*.

A constructor invocation with `new` is required to allocate an object. There is no C++ like static allocation.

To compile this class, put it in a file called OriginPrinter.java in the same directory as TwoDPoint.java and type:

```
% javac OriginPrinter.java
```
What does this produce?

Is this a complete program now? Can you run it?

Multiple Objects

In general there will be more than one object in any given class. Reference variables are used to distinguish between different objects of the same class.

For example, the following program creates two two-d point objects and prints their fields:

```
class TwoPointPrinter {

  public static void main(String[] args) {

    TwoDPoint origin; // only declares, does not allocate
    TwoDPoint one; // only declares, does not allocate

    // The constructor allocates and usually initializes the object
    origin = new TwoDPoint();
    one = new TwoDPoint();

    // set the fields
    origin.x = 0.0;
    origin.y = 0.0;
    one.x = 1.0;
    one.y = 0.0;

    // print the two-d points
    System.out.println("The origin is at " + origin.x + ", " + origin.y);
    System.out.println("One is at " + one.x + ", " + one.y);

  }  // end main

} // end TwoPointPrinter
```

`one` and `origin` are two different reference variables pointing to two different point objects. It's not enough to identify a variable as a member of a class like `x` or `y` in the example above. You have to specify which object in the class you're referring to.

Multiple Objects

It is possible for two different reference variables to point to the same object.

When an object is no longer pointed to by any reference variable (including references stored deep inside the runtime or class library) it will be marked for garbage collection.

For example, the following program declares two TwoDPoint reference variables, creates one two-d point object, and assigns that object to both variables. The two variables are equal.

```
class EqualPointPrinter {

  public static void main(String[] args) {

    TwoDPoint origin1; // only declares, does not allocate
    TwoDPoint origin2; // only declares, does not allocate

    // The constructor allocates and usually initializes the object
    origin1 = new TwoDPoint();
    origin2 = origin1;

    // set the fields
    origin1.x = 0.0;
    origin1.y = 0.0;


    // print
    System.out.println("origin1 is at " + origin1.x + ", " + origin1.y);
    System.out.println("origin2 is at " + origin2.x + ", " + origin2.y);

  }  // end main

} // end EqualPointPrinter
```

`origin1` and `origin2` are two different reference variables referring to the same point object.

Static Fields

Static or class fields belong to a class, not to an object

```
class Point {

    double x;
    double y;
    static double xorigin = 0.0;
    static double yorigin = 0.0;
```

```
}
System.out.println("The origin is at (" + Point.xorigin + ", "
 + Point.yorigin + ")");
```
You access class variables with the name of the class rather than a reference variable.

Methods

Methods say what an object does.

```
class TwoDPoint {

    double x;
    double y;

    void print() {
      System.out.println(this.x + "," + this.y);
    }

  }

}
```
Notice that you use the Java keyword `this` to reference a field from inside the same class.

```
TwoDPoint origin = new TwoDPoint();
origin.x = 0.0;
origin.y = 0.0;
origin.print();
```
noun-verb instead of verb-noun; that is subject-verb instead of verb-direct object.

subject-verb-direct object(s) is also possible.

Passing Arguments to Methods

```
class TwoDPoint {

    double x;
    double y;

    void print() {
      System.out.println("(" + this.x + "," + this.y + ")");
    }

    void print(int n) {
      for (int i = 0; i < n; i++) {
        System.out.println("(" + this.x + "," + this.y + ")");
      }
    }

  }

}
```

To use this class, you might have some lines like these in a separate class and file:

```
TwoDPoint origin = new TwoDPoint();
origin.x = 0.0;
origin.y = 0.0;
origin.print(10);
```

Note that there are two different `print()` methods. One takes an argument. One doesn't. As long as the argument lists can disambiguate the choice, this is allowed. This is called *overloading*.

Also note, that the `System.out.println()` we've been using all along is an overloaded method.

`main(String[] args)` is a non-overloaded method that has an array of strings as arguments.

Returning values from methods

```
class TwoDPoint {

    double x;
    double y;

    void print() {
      System.out.println("(" + this.x + "," + this.y + ")");
    }

    String getAsString() {
      return "(" + this.x + "," + this.y + ")";
    }

  }

}
TwoDPoint origin = new TwoDPoint();
origin.x = 0.0;
origin.y = 0.0;
String s = origin.getAsString();
System.out.println(s);
```

Better yet,

```
TwoDPoint origin = new TwoDPoint();
origin.x = 0.0;
origin.y = 0.0;
System.out.println(origin.getAsString());
```

setter methods

Also known as *mutator* methods, setter methods just set the value of a field (often private) in a class.

```
class TwoDPoint {
```

```
    double x;
    double y;

    String getAsString() {
      return "(" + this.x + "," + this.y + ")";
    }

    void setX(double value) {
      this.x = value;
    }

    void setY(double value) {
      this.y = value;
    }

  }

}
TwoDPoint origin = new TwoDPoint();
origin.setX(0.0);
origin.setY(0.0);
System.out.println(origin.getAsString());
```

getter methods

Also known as *accessor* methods, getter methods just return the value of a field in a class.

```
class TwoDPoint {

    double x;
    double y;

    String getAsString() {
      return "(" + this.x + "," + this.y + ")";
    }

    void setX(double value) {
      this.x = value;
    }

    void setY(double value) {
      this.y = value;
    }

    double getX() {
      return this.x;
    }

    double getY() {
      return this.y;
    }

  }
```

```
}
TwoDPoint origin = new TwoDPoint();
origin.setX(0.0);
origin.setY(0.0);
System.out.println("The x coordinate is " + origin.getX());
```

Constructors

Constructors create new instances of a class, that is objects. Constructors are special methods that have the same name as their class and no return type. For example,

```
class TwoDPoint {

    double x;
    double y;

    TwoDPoint(double xvalue, double yvalue) {
      this.x = xvalue;
      this.y = yvalue;
    }

    String getAsString() {
      return "(" + this.x + "," + this.y + ")";
    }

    void setX(double value) {
      this.x = value;
    }

    void setY(double value) {
      this.y = value;
    }

    double getX() {
      return this.x;
    }

    double getY() {
      return this.y;
    }

  }

}
```

Constructors are used along with the `new` keyword to produce an object in the class (also called an instance of the class):

```
TwoDPoint origin = new TwoDPoint(0.0, 0.0);
System.out.println("The x coordinate is " + origin.getX());
```

Shadowing field names and `this`

By making use of the `this` keyword, you can even use the same name for arguments to the constructor (or any other method) as you use for field names. For example,

```
class TwoDPoint {

    double x;
    double y;

    TwoDPoint(double x, double y) {
      this.x = x;
      this.y = y;
    }

    String getAsString() {
      return "(" + this.x + "," + this.y + ")";
    }

    void setX(double x) {
      this.x = x;
    }

    void setY(double y) {
      this.y = y;
    }

    double getX() {
      return this.x;
    }

    double getY() {
      return this.y;
    }

  }

}
```

Inside a method, a declaration of a variable or argument with the same name as a field *shadows* the field. You can refer to the field by prefixing its name with `this`.

Arrays

An array is a collection of variables of the same type.

The `args[]` array of a `main()` method is an array of Strings.

Consider a class which counts the occurrences of the digits 0-9. For example you might wish to test the randomness of a random number generator. If a random number generator is truly random, all digits should occur with equal frequency over a sufficiently long period of time.

You will do this by creating an array of ten ints called `ndigit`. The zeroth component of `ndigit` will track the number of zeros; the first component will track the numbers of ones and so forth.

The RandomTest program below tests Java's random number generator to see if it produces apparently random numbers.

```
import java.util.Random;

class RandomTest {

  public static void main (String args[]) {

    int[] ndigits = new int[10];
    double x;
    int n;

    Random myRandom = new Random();

    // Initialize the array
    for (int i = 0; i < 10; i++) {
      ndigits[i] = 0;
    }

    // Test the random number generator a whole lot
    for (long i=0; i < 100000; i++) {
      // generate a new random number between 0 and 9
      x = myRandom.nextDouble() * 10.0;
      n = (int) x;
      //count the digits in the random number
      ndigits[n]++;
    }

    // Print the results
    for (int i = 0; i < 10; i++) {
      System.out.println(i+": " + ndigits[i]);
    }
  }

}
```

Below is one possible output from this program. If you run it your results should be slightly different. After all this is supposed to be random. These results are pretty much what you would expect from a reasonably random generator. If you have a fast CPU and some time to spare, try bringing the number of tests up to a billion or so, and see if the counts for the different digits get any closer to each other.

```
% javac RandomTest.java
% java RandomTest
0: 10171
1: 9724
2: 9966
3: 10065
4: 9989
5: 10132
6: 10001
7: 10158
8: 9887
9: 9907
%
```

There are three `for` loops in this program, one to initialize the array, one to perform the desired calculation, and a final one to print out the results. This is quite common in code that uses arrays.

Exercises

1. Get Hello World to work.
2. Personalize the Hello World program with your name so that it tells you Hello rather than the somewhat generic "World."
3. Write a program that produces the following output:

```
Hello World!
It's been nice knowing you.
Goodbye world!
```

8. Write a program that prints all the integers between 0 and 36.
9. Imagine you need to open a standard combination dial lock but don't know the combination and don't have a pair of bolt cutters. Write a program that prints all possible combinations so you can print them on a piece of paper and check off each one as you try it. Assume the numbers on the dial range from zero to thirty-six and three numbers in sequence are needed to open the lock.
10. Suppose the lock isn't a very good one and any number that's no more than two away from the correct number in each digit will also work. In other words if the combination is 17-6-32, then 18-5-31, 19-4-32, 15-8-33 and many other combinations will also open the lock. Write a program that prints out a minimal list of combinations you would need to try to guarantee opening the lock.
11. Write a program that randomly fills a 10 component array, then prints the largest and smallest values in the array.
12. Hand in the first page of a print out of the documentation for the `java.text.DecimalFormat` class.
13. Hand in a screen shot of your web browser's Bookmarks or Shortcuts *menu* showing a bookmark for the Java class library documentation.
14. Install jEdit. Hand in a screen shot of the program showing the hello worls source code.
15. Sign up for an account on utopia.
16. Go to http://utopia.poly.edu and follow the instructions for signing up for a home page.

Primitive Data Types in Java

Java's primitive data types are very similar to those of C. They include boolean, byte, short, int, long, float, double, and char. The boolean type has been added. However the implementation of the data types has been substantially cleaned up in several ways.

1. Where C and C++ leave a number of issues to be machine and compiler dependent (for instance the size of an `int`) Java specifies everything.

2. Java prevents casting between arbitrary variables. Only casts between numeric variables and between sub and superclasses of the same object are allowed.
3. All numeric variables in Java are signed.

`sizeof` isn't necessary in Java because all sizes are precisely defined. For instance, an `int` is always 4 bytes. This may not seem to be adequate when dealing with objects that aren't base data types. However even if you did know the size of a particular object, you couldn't do anything with it anyway. You cannot convert an arbitrary object into bytes and back again.

Java's Primitive Data Types

`boolean`
: 1-bit. May take on the values `true` and `false` only.

  `true` and `false` are defined constants of the language and are not the same as `True` and `False`, `TRUE` and `FALSE`, zero and nonzero, 1 and 0 or any other numeric value. Booleans may not be cast into any other type of variable nor may any other variable be cast into a boolean.

`byte`
: 1 signed byte (two's complement). Covers values from -128 to 127.

`short`
: 2 bytes, signed (two's complement), -32,768 to 32,767

`int`
: 4 bytes, signed (two's complement). -2,147,483,648 to 2,147,483,647. Like all numeric types ints may be cast into other numeric types (byte, short, long, float, double). When *lossy* casts are done (e.g. int to byte) the conversion is done modulo the length of the smaller type.

`long`
: 8 bytes signed (two's complement). Ranges from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

`float`
: 4 bytes, IEEE 754. Covers a range from 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative).

  Like all numeric types floats may be cast into other numeric types (`byte`, `short`, `long`, `int`, `double`). When *lossy* casts to integer types are done (e.g. `float` to `short`) the fractional part is truncated and the conversion is done modulo the length of the smaller type.

`double`
: 8 bytes IEEE 754. Covers a range from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).

`char`
: 2 bytes, unsigned, Unicode, 0 to 65,535

Chars are not the same as bytes, ints, shorts or Strings.

Java Operators

An operator is a symbol that operates on one or more arguments to produce a result. The Hello World program is so simple it doesn't use any operators, but almost all other programs you write will.

Operator	Purpose
+	addition of numbers, concatenation of Strings
+=	add and assign numbers, concatenate and assign Strings
-	subtraction
-=	subtract and assign
*	multiplication
*=	multiply and assign
/	division
/=	divide and assign
|	bitwise OR
|=	bitwise OR and assign
^	bitwise XOR
^=	bitwise XOR and assign
&	bitwise AND
&=	bitwise AND and assign
%	take remainder
%=	take remainder and assign
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!	boolean NOT
!=	not equal to
++	increment by one
--	decrement by one
>>	shift bits right with sign extension
>>=	shift bits right with sign extension and assign
<<	shift bits left
<<=	shift bits left and assign
>>>	unsigned bit shift right

`>>>=`	unsigned bit shift right and assign
`&&`	boolean AND
`||`	boolean OR
`==`	boolean equals
`=`	assignment
`~`	bitwise NOT
`?:`	conditional
`instanceof`	type checking

White Space

White space consists mostly of the space character that you produce by hitting the space bar on your keyboard and that is commonly used to separate words in sentences. There are four other white space characters in Java, the horizontal tab, the form feed, the carriage return, and the linefeed. Depending on your platform, when you hit the return or enter key, you get either a carriage return (the Mac), a linefeed (Unix) or both (DOS, Windows, VMS). This produces a hard line break in the source code text.

Outside of `String` literals Java treats all white space and runs of white space (more than one white space character in immediate succession) the same. It's used to separate tokens, and one space is as good as seven spaces, a tab and two carriage returns. Exactly which white space characters you use is primarily a result of what's convenient for human beings reading the code. The compiler doesn't care.

Inside `String` and character literals the only white space permitted is the space character. Carriage returns, tabs, line feeds and form feeds must be inserted with special escape sequences like `\r`, `\t`, `\f`, and `\n`. You cannot break a String across a line like this:

```
String poem = "Mary had a little lamb
whose fleece was white as snow
and everywhere that Mary went
the lamb was sure to go.";
```

Instead you must use `\n` and the string concatenation operator, `+`, like this:

```
String poem = "Mary had a little lamb\n" +
"whose fleece was white as snow\n" +
"and everywhere that Mary went\n" +
"the lamb was sure to go.";
```

Note that you can break a statement across multiple lines, you just can't break a `String` literal.

Also note that `\n` only works on Unix. You should probably use `System.getProperty("line.separator")` instead to return the proper line separator string for the platform your program is running on.

Java does not have all the escape sequences C has. Besides those already mentioned it has only \b for backspace, \\ for the backslash character itself.

There are also \u escapes that let you include any Unicode character.

Literals

Literals are pieces of Java source code that mean exactly what they say. For instance `"Hello World!"` is a String literal and its meaning is the words Hello World!

The string `"Hello World!"` looks like it's several things; but to the compiler it's just one thing, a `String`. This is similar to how an expression like 1,987,234 may be seven digits and two commas but is really just one number.

The double quote marks tell you this is a string literal. A string is an ordered collection of characters (letters, digits, punctuation marks, etc.). Although the String may have meaning to a human being reading the code, the computer sees it as no more than a particular set of letters in a particular order. It has no concept of the meaning of the characters. For instance it does not know that "two" + "two" is "four." In fact the computer thinks that "two" + "two" is "twotwo"

The quote marks show where the string begins and ends. However the quote marks themselves are not a part of the string. The value of this string is Hello World!, not "Hello World!" You can change the output of the program by changing Hello World to some other line of text.

A string in a Java program has no concept of italics, bold face, font family or other formatting. It cares only about the characters that compose it. Even if you're using an editor like NisusWriter that lets you format text files, **"Hello World!"** is identical to *"Hello World!"* as far as Java is concerned.

`char` literals are similar to string literals except they're enclosed in single quotes and must have exactly one character. For example `'c'` is a `char` literal that means the letter c.

`true` and `false` are boolean literals that mean true and false.

Numbers can also be literals. `34` is an `int` literal and it means the number thirty-four. `1.5` is a `double` literal. `45.6`, `76.4E8` (76.4 times 10 to the 8th power) and -32.0 are also `double` literals.

`34L` is a `long` literal and it means the number thirty-four. `1.5F` is a `float` literal. `45.6f`, `76.4E8F` and -32.0F are also `float` literals.

Identifiers in Java

Identifiers are the names of variables, methods, classes, packages and interfaces. Unlike literals they are not the things themselves, just ways of referring to them. In the HelloWorld program, `HelloWorld`, `String`, `args`, `main` and `System.out.println` are identifiers.

Identifiers must be composed of letters, numbers, the underscore _ and the dollar sign `$`. Identifiers may only begin with a letter, the underscore or a dollar sign.

Each variable has a name by which it is identified in the program. It's a good idea to give your variables mnemonic names that are closely related to the values they hold. Variable names can include any alphabetic character or digit and the underscore _. The main restriction on the names you can give your variables is that they cannot contain any white space. You cannot begin a variable name with a number. It is important to note that as in C but not as in Fortran or Basic, all variable names are case-sensitive. `MyVariable` is not the same as `myVariable`. There is no limit to the length of a Java variable name. The following are legal variable names:

- `MyVariable`
- `myvariable`
- `MYVARIABLE`
- `x`
- `i`
- `_myvariable`
- `$myvariable`
- `_9pins`
- `andros`
- `ανδροϛ`
- `OReilly`
- `This_is_an_insanely_long_variable_name_that_just_keeps_going_and_going_and_going_and_well_you_get_the_idea_The_line_breaks_arent_really_part_of_the_variable_name_Its_just_that_this_variable_name_is_so_ridiculously_long_that_it_won't_fit_on_the_page_I_cant_imagine_why_you_would_need_such_a_long_variable_name_but_if_you_do_you_can_have_it`

T he following are not legal variable names.

- `My Variable // Contains a space`
- `9pins // Begins with a digit`
- `a+c // The plus sign is not an alphanumeric character`
- `testing1-2-3 // The hyphen is not an alphanumeric character`
- `O'Reilly // Apostrophe is not an alphanumeric character`
- `OReilly_&_Associates // ampersand is not an alphanumeric character`

Tip: How to Begin a Variable Name with a Number

If you want to begin a variable name with a digit, prefix the name you'd like to have (e.g. 8ball) with an underscore, e.g. _8ball. You can also use the underscore to act like a space in long variable names.

Keywords

Keywords are identifiers like `public`, `static` and `class` that have a special meaning inside Java source code and outside of comments and Strings. Four keywords are used in Hello World, `public`, `static`, `void` and `class`.

Keywords are reserved for their intended use and cannot be used by the programmer for variable or method names.

There are fifty reserved keywords in Java 1.1 (51 in Java 1.2). The forty-eight that are actually used in are listed below. Don't worry if the purposes of the keywords seem a little opaque at this point. They will all be explained in much greater detail later.

Keywords Used in Java 1.1

Keyword	Purpose
`abstract`	declares that a class or method is abstract
`boolean`	declares a boolean variable or return type
`break`	prematurely exits a loop
`byte`	declares a byte variable or return type
`case`	one case in a switch statement
`catch`	handle an exception
`char`	declares a character variable or return type
`class`	signals the beginning of a class definition
`continue`	prematurely return to the beginning of a loop
`default`	default action for a switch statement
`do`	begins a do while loop
`double`	declares a double variable or return type
`else`	signals the code to be executed if an if statement is not true
`extends`	specifies the class which this class is a subclass of
`final`	declares that a class may not be subclassed or that a field or method may not be overridden
`finally`	declares a block of code guaranteed to be executed
`float`	declares a floating point variable or return type
`for`	begins a for loop
`if`	execute statements if the condition is true
`implements`	declares that this class implements the given interface
`import`	permit access to a class or group of classes in a package

`instanceof`	tests whether an object is an instanceof a class
`int`	declares an integer variable or return type
`interface`	signals the beginning of an interface definition
`long`	declares a long integer variable or return type
`native`	declares that a method is implemented in native code
`new`	allocates a new object
`package`	defines the package in which this source code file belongs
`private`	declares a method or member variable to be private
`protected`	declares a class, method or member variable to be protected
`public`	declares a class, method or member variable to be public
`return`	returns a value from a method
`short`	declares a short integer variable or return type
`static`	declares that a field or a method belongs to a class rather than an object
`super`	a reference to the parent of the current object
`switch`	tests for the truth of various possible cases
`synchronized`	Indicates that a section of code is not thread-safe
`this`	a reference to the current object
`throw`	throw an exception
`throws`	declares the exceptions thrown by a method
`transient`	This field should not be serialized
`try`	attempt an operation that may throw an exception
`void`	declare that a method does not return a value
`volatile`	Warns the compiler that a variable changes asynchronously
`while`	begins a while loop

Two other keywords, `const` and `goto`, are reserved by Java but are not actually implemented. This allows compilers to produce better error messages if these common C++ keywords are improperly used in a Java program.

Java 1.2 adds the `strictfp` keyword to declare that a method or class must be run with exact IEEE 754 semantics.

`true` and `false` appear to be missing from this list. In fact, they are not keywords but rather boolean literals. You still can't use them as a variable name though.

Separators in Java

Separators help define the structure of a program. The separators used in HelloWorld are parentheses, `( )`, braces, `{ }`, the period, `.`, and the semicolon, `;`. The table lists the six Java separators (nine if you count opening and closing separators as two).

Separator

	Purpose
( )	Encloses arguments in method definitions and calling; adjusts precedence in arithmetic expressions; surrounds cast types and delimits test expressions in flow control statements
{ }	defines blocks of code and automatically initializes arrays
[ ]	declares array types and dereferences array values
;	terminates statements
,	separates successive identifiers in variable declarations; chains statements in the test, expression of a for loop
.	Selects a field or method from an object; separates package names from sub-package and class names
:	Used after loop labels

Addition of Integers in Java

```
class AddInts {

  public static void main (String args[]) {

    int i = 1;
    int j = 2;
    int k;

    System.out.println("i is " + i);
    System.out.println("j is " + j);

    k = i + j;
    System.out.println("i + j is " + k);

    k = i - j;
    System.out.println("i - j is " + k);

  }

}
```

Here's what happens when you run AddInts:

```
% javac AddInts.java
% java AddInts
i is 1
j is 2
i + j is 3
i - j is -1
```

Addition of doubles in Java

Doubles are treated much the same way, but now you get to use decimal points in the numbers. This is a similar program that does addition and subtraction on doubles.

```
class AddDoubles {

  public static void main (String args[]) {

    double x = 7.5;
    double y = 5.4;
    double z;


    System.out.println("x is " + x);
    System.out.println("y is " + y);

    z = x + y;
    System.out.println("x + y is " + z);

    z = x - y;
    System.out.println("x - y is " + z);

  }

 }
```

Here's the result:

```
% javac AddDoubles.java
% java AddDoubles
x is 7.5
y is 5.4
x + y is 12.9
x - y is 2.0999999999999996
```

Multiplication and division in Java

Of course Java can also do multiplication and division. Since most keyboards don't have the times and division symbols you learned in grammar school, Java uses `*` to mean multiplication and `/` to mean division. The syntax is straightforward as you see below.

```
class MultiplyDivide {

  public static void main (String args[]) {

    int i = 10;
    int j = 2;
    int k;

    System.out.println("i is " + i);
    System.out.println("j is " + j);

    k = i/j;
    System.out.println("i/j is " + k);
    k = i * j;
```

```
    System.out.println("i * j is " + k);

  }

}
```

Here's the result:

```
% javac MultiplyDivide.java
% java MultiplyDivide
i is 10
j is 2
i/j is 5
i * j is 20
%
```

Floats and doubles are multiplied and divided in exactly the same way. When faced with an inexact integer division, Java rounds the result down. For instance dividing 10 by 3 produces 3.

Unexpected Quotients

2/3 = 0

3/2 = 1

1/0 = ArithmeticException

0/0 = ArithmeticException

1.0/0.0 = Inf

1.0/0 = Inf

0.0/0.0 = NaN

-1.0/0.0 = -Inf

Inf + 1 = Inf

Inf + Inf = Inf

Inf - Inf = NaN

Inf/Inf = NaN

NaN + anything = NaN

NaN - anything = NaN

NaN * anything = NaN

NaN - anything = NaN

NaN < NaN is false

NaN > NaN is false

NaN <= NaN is false

NaN >= NaN is false

NaN == NaN is false

NaN != NaN is true

The Remainder or Modulus Operator in Java

Java has one important arithmetical operator you may not be familiar with, `%`, also known as the modulus or remainder operator. The `%` operator returns the remainder of two numbers. For instance `10 % 3` is 1 because 10 divided by 3 leaves a remainder of 1. You can use `%` just as you might use any other more common operator like `+` or `-`.

```
class Remainder {

  public static void main (String args[]) {

    int i = 10;
    int j = 3;
    int k;

    System.out.println("i is " + i);
    System.out.println("j is " + j);

    k = i%j;
    System.out.println("i%j is " + k);
  }

}
```

Here's the output:

```
% javac Remainder.java
% java Remainder
i is 10
j is 3
i%j is 1
%
```

Perhaps surprisingly the remainder operator can be used with floating point values as well. It's surprising because you don't normally think of real number division as producing remainders. However there are rare times when it's useful to ask exactly how many times does 1.5 go into 5.5

and what's left over? The answer is that 1.5 goes into 5.5 three times with one left over, and it's that one which is the result of 5.5 % 1.5 in Java.

Operator Precedence in Java

It's possible to combine multiple arithmetic expressions in one statement. For instance the following line adds the numbers one through five:

```
int m = 1 + 2 + 3 + 4 + 5;
```

A slightly more interesting example: the following program calculates the energy equivalent of an electron using Einstein's famous formula $E = mc^2$.

```
class mc2 {
  public static void main (String args[]) {

    double mass = 9.1096E-25;
    double c = 2.998E8;
    double E = mass * c * c;
    System.out.println(E);
  }
}
```

Here's the output:

```
% javac mc2.java
% java mc2
8.18771e-08
%
```

This is all very obvious. However if you use different operators on the same line it's not always clear what the result will be. For instance consider the following code fragment:

```
int n = 1 - 2 * 3 - 4 + 5;
```

Is n equal to -2? You might think so if you just calculate from left to right. However if you compile this in a program and print out the result you'll find that Java thinks n is equal to -4. Java got that number because it performs all multiplications before it performs any additions or subtractions. If you like you can think of the calculation Java did as being:

```
int n = 1 - (2 * 3) - 4 + 5;
```

This is an issue of order of evaluation. Within the limited number of operators you've learned so far here is how Java calculates:

1. *, /, % Do all multiplications, divisions and remainders from left to right.
2. +, - Do additions and subtractions from left to right.
3. = Assign the right-hand side to the left-hand side

Parentheses in Java

Sometimes the default order of evaluation isn't what you want. For instance, the formula to change a Fahrenheit temperature to a Celsius temperature is C = (5/9) (F - 32) where C is degrees Celsius and F is degrees Fahrenheit. You must subtract 32 from the Fahrenheit temperature before you multiply by 5/9, not after. You can use parentheses to adjust the order much as they are used in the above formula. The next program prints a table showing the conversions from Fahrenheit and Celsius between zero and three hundred degrees Fahrenheit every twenty degrees.

```
// Print a Fahrenheit to Celsius table

class FahrToCelsius  {

  public static void main (String args[]) {

    double fahr, celsius;
    double lower, upper, step;

    // lower limit of temperature table
    lower = 0.0;

    // upper limit of temperature table
    upper = 300.0;

    // step size
    step  = 20.0;

    fahr = lower;
    while (fahr <= upper) {
      celsius = (5.0 / 9.0) * (fahr-32.0);
      System.out.println(fahr + " " + celsius);
      fahr = fahr + step;
    }

  }

}
```

Parentheses in Java

As usual here's the output:

```
% javac FahrToCelsius.java
% java FahrToCelsius
0 -17.7778
20 -6.66667
40 4.44444
60 15.5556
80 26.6667
100 37.7778
120 48.8889
140 60
160 71.1111
180 82.2222
200 93.3333
```

```
220 104.444
240 115.556
260 126.667
280 137.778
300 148.889
%
```

This program is a little more involved than the previous examples. Mostly it's stuff you've seen before though so a line by line analysis isn't necessary. The line to be concerned with is

```
celsius = (5.0 / 9.0) * (fahr-32.0);
```

This is a virtual translation of the formula C = (5/9)(F - 32) with the single change that a * was added because Java does not implicitly multiply items in parentheses. The parentheses are used just as they are in regular algebra, to adjust the precedence of terms in a formula. In fact the precedence of operations that use the basic arithmetic operators (+, -, *, /) is exactly the same as you learned in high school algebra.

Remember, you can always use parentheses to change the order of evaluation. Everything inside the parentheses will be calculated before anything outside of the parentheses is calculated. If you're in doubt it never hurts to put in extra parentheses to clear up the order in which terms will be evaluated.

Mixing Data Types

As well as combining different operations, you can mix and match different numeric data types on the same line. The program below uses both ints and doubles, for example.

```
class IntAndDouble {

  public static void main (String args[]) {

    int i = 10;
    double x = 2.5;
    double k;

    System.out.println("i is " + i);
    System.out.println("x is " + x);

    k = i + x;
    System.out.println("i + x is " + k);
    k = i * x;
    System.out.println("i * x is " + k);
    k = i - x;
    System.out.println("i - x is " + k);
    k = x - i;
    System.out.println("x - i is " + k);
    k = i / x;
    System.out.println("i / x is " + k);
    k = x / i;
    System.out.println("x / i is " + k);
```

```
  }

}
```

This program produces the following output:

```
% java IntAndDouble
i is 10
x is 2.5
i + x is 12.5
i * x is 25
i - x is 7.5
x - i is -7.5
i / x is 4
x / i is 0.25
%
```

Mixing Data Types

Order can make a differenc when data types are mixed. For example,

1 / 2 * 3.5 = 0.0
3.5 * 1 / 2 = 1.75
3.5 / 2 = 1.75

You cannot assume that the usual mathematical laws of commutativity apply when mixing data types, especially integer and floating point types.

1.0 / 2 * 3.5 = 1.75
3.5 * 1.0 / 2 = 1.75
1 / 2.0 * 3.5 = 1.75
3.5 * 1.0 / 2.0 = 1.75

Arithmetic Promotion and Binary Operations

An `int` divided by an `int` is an int, and a `double` divided by a `double` is a `double`, but what about an `int` divided by a `double` or a `double` divided by an `int`? When doing arithmetic on unlike types Java tends to widen the types involved so as to avoid losing information. After all 3 * 54.2E18 will be a perfectly valid `double` but much too big for any `int`.

The basic rule is that if either of the variables in a binary operation (addition, multiplication, subtraction, addition, remainder) are doubles then Java treats both values as doubles. If neither value is a `double` but one is a `float`, then Java treats both values as floats. If neither is a float or

a double but one is a `long`, then Java treats both values as longs. Finally if there are no doubles, floats or longs, then Java treats both values as an `int`, even if there aren't any ints in the equation. Therefore the result will be a `double`, `float`, `long` or `int` depending on the types of the arguments.

Arithmetic Promotion, Assignments, and Casting

In an assignment statement, i.e. if there's an equals sign, Java compares the type of the left hand side to the final type of the right hand side. It won't change the type of the left hand side, but it will check to make sure that the value it has (`double`, `float`, `int` or `long`) on the right hand side can fit in the type on the left hand side. Anything can fit in a `double`. Anything except a `double` can fit in a `float`. Any integral type can fit in a `long`, but a `float` or a `double` can't, and ints, shorts, and bytes can fit inside ints. If the right hand side can fit inside the left hand side, the assignment takes place with no further ado.

Assigning `long` values to `int` variables or `double` values to `float` variables can be equally troublesome. In fact it's so troublesome the compiler won't let you do it unless you tell it you really mean it with a cast. When it's necessary to force a value into a particular type, use a cast. To cast a variable or a literal or an expression to a different data type just precede it with the type in parentheses. For instance:

```
int i = (int) 9.0/4.0;
```

A cast lets the compiler know that you're serious about the conversion you plan to make.

When a value is cast down before assignment, series of operations takes place to chop the right hand side down to size. For a conversion between a floating point number and an int or a long, the fractional part of the floating point number is truncated (rounded toward zero). This produces an integer. If the integer is small enough to fit in the left hand side, the assignment is completed. On the other hand if the number is too large, then the integer is set to the largest possible value of its type. If the floating point number is too small the integer is set to the smallest possible value of its type.

This can be a nasty bug in your code. It can also be hard to find since everything may work perfectly 99 times out of a hundred and only on rare occasions will the rounding become a problem. However when it does there will be no warning or error message. You need to be very careful when assigning floating point values to integer types.

Converting Strings to Numbers

When processing user input it is often necessary to convert a `String` that the user enters into an `int`. The syntax is straightforward. It requires using the static `Integer.valueOf(String s)` and `intValue()` methods from the `java.lang.Integer` class. To convert the `String "22"` into the `int 22` you would write

```
int i = Integer.valueOf("22").intValue();
```

Doubles, floats and longs are converted similarly. To convert a String like "22" into the long value 22 you would write

```
long l = Long.valueOf("22").longValue();
```

To convert `"22.5"` into a `float` or a `double` you would write:

```
double x = Double.valueOf("22.5").doubleValue();
float y = Float.valueOf("22.5").floatValue();
```

The various `valueOf()` methods are relatively intelligent and can handle plus and minus signs, exponents, and most other common number formats. However if you pass one something completely non-numeric like `"pretty in pink,"` it will throw a `NumberFormatException`. You haven't learned how to handle exceptions yet, so try to avoid passing theses methods non-numeric data.

You can now rewrite the E = mc2 program to accept the mass in kilograms as user input from the command line. Many of the exercises will be similar.

```
class Energy {
  public static void main (String args[]) {

    double mass;
    double c = 2.998E8;  // meters/second
    double E;

    mass = Double.valueOf(args[0]).doubleValue();
    E = mass * c * c;
    System.out.println(E + " Joules");
  }
}
```

Here's the output:

```
% javac Energy.java
% java Energy 0.0456
4.09853e+15 Joules
%
```

The char data type in Java

A char is a single character, that is a letter, a digit, a punctuation mark, a tab, a space or something similar. A char literal is a single one character enclosed in single quote marks like this

```
char myCharacter = 'g';
```

Some characters are hard to type. For these Java provides escape sequences. This is a backslash followed by an alphanumeric code. For instance `'\n'` is the newline character. `'\t'` is the tab character. `'\\'` is the backslash itself. The following escape sequences are defined:

\b backspace
\t tab
\n linefeed
\f formfeed
\r carriage return
\" double quote, "
\' single quote, '
\\ backslash, \

The double quote escape sequence is used mainly inside strings where it would otherwise terminate the string. For instance

```
System.out.println("And then Jim said, \"Who's at the door?\"");
```

It isn't necessary to escape the double quote inside single quotes. The following line is legal in Java

```
char doublequote = '"';
```

Unicode

Java uses the Unicode character set. Unicode is a two-byte character code set that has characters representing almost all characters in almost all human alphabets and writing systems around the world including English, Arabic, Chinese and more.

Unfortunately many operating systems and web browsers do not handle Unicode. For the most part Java will properly handle the input of non-Unicode characters. The first 128 characters in the Unicode character set are identical to the common ASCII character set. The second 128 characters are identical to the upper 128 characters of the ISO Latin-1 extended ASCII character set. It's the next 65,280 characters that present problems.

You can refer to a particular Unicode character by using the escape sequence \u followed by a four digit hexadecimal number. For example

\u00A9 © The copyright symbol

\u0022 " The double quote
\u00BD 1/2 The fraction 1/2
\u0394 Δ The capital Greek letter delta
\u00F8 ø A little o with a slash through it

You can even use the full Unicode character sequence to name your variables. However chances are your text editor doesn't handle more than basic ASCII very well. You can use Unicode escape sequences instead like this

```
String Mj\u00F8lner = "Hammer of Thor";
```

but frankly this is way more trouble than it's worth.

Java Flow Control

- `if`
- `else`
- `else if`
- `while`
- `for`
- `do while`
- `switch case`
- `break`
- `continue`

`goto` is a reserved word. It is not implemented.

We'll talk about exception handling later.

The if statement in Java

All but the most trivial computer programs need to make decisions. They test a condition and operate differently based on the outcome of the test. This is quite common in real life. For instance you stick your hand out the window to test if it's raining. If it is raining then you take an umbrella with you. If it isn't raining then you don't.

All programming languages have some form of an `if` statement that tests conditions. In the previous code you should have tested whether there actually were command line arguments before you tried to use them.

Arrays have lengths and you can access that length by referencing the variable `arrayname.length` You test the length of the `args` array as follows.

```
// This is the Hello program in Java

class Hello {

    public static void main (String args[]) {

      if (args.length > 0) {
        System.out.println("Hello " + args[0]);
      }
  }

}
```

`System.out.println(args[0])` was wrapped in a conditional test, `if (args.length > 0) { }`. The code inside the braces, `System.out.println(args[0])`, now gets executed if and only if the length of the args array is greater than zero.

The arguments to a conditional statement like `if` must be a boolean value, that is something that evaluates to true or false. Integers are not permissible.

In Java numerical greater than and lesser than tests are done with the > and < operators respectively. You can test whether a number is less than or equal to or greater than or equal to another number with the <= and >= operators.

Testing for Equality

Testing for equality is a little trickier. You would expect to test if two numbers are equal by using the = sign. However the = sign has already been used as an assignment operator that sets the value of a variable. Therefore a new symbol is needed to test for equality. Java borrows C's double equals sign, ==, for this purpose.

It's not uncommon for even experienced programmers to write == when they mean = or vice versa. In fact this is a very common cause of errors in C programs. Fortunately in Java, you are not allowed to use == and = in the same places. Therefore the compiler can catch your mistake and make you fix it before you can run the program.

However there is one way you can still get into trouble:

```
boolean b = true;
 if (b = false) {
   System.out.println("b is false");
 }
```

To avoid this, some programmers get in the habit of writing condition tests like this:

```
boolean b = true;
 if (false = b) {
   System.out.println("b is false");
```

```
      }
```

Since you can't assign to a literal, this causes a compiler error if you misuse the = sign when you mean to write ==.

The else statement in Java

```
// This is the Hello program in Java

class Hello {

    public static void main (String args[]) {

      if (args.length > 0) {
        System.out.println("Hello " + args[0]);
      }
      else {
        System.out.println("Hello whoever you are.");
      }
  }

}
```

Else If

`if` statements are not limited to two cases. You can combine an `else` and an `if` to make an `else if` and test a whole range of mutually exclusive possibilities. For instance, here's a version of the Hello program that handles up to four names on the command line:

```
// This is the Hello program in Java

class Hello {

    public static void main (String args[]) {

      if (args.length == 0) {
        System.out.println("Hello whoever you are");
      }
      else if (args.length == 1) {
        System.out.println("Hello " + args[0]);
      }
      else if (args.length == 2) {
        System.out.println("Hello " + args[0] + " " + args[1]);
      }
      else if (args.length == 3) {
        System.out.println("Hello " + args[0] + " " + args[1] + " " +
args[2]);
      }
      else if (args.length == 4) {
        System.out.println("Hello " + args[0] +
          " " + args[1] + " " args[2] + " " + args[3]);
      }
```

```
      else {
        System.out.println("Hello " + args[0] + " " + args[1] + " " args[2]
         + " " + args[3] + " and all the rest!");
      }

  }

}
```

You can see that this gets mighty complicated mighty quickly. No experienced Java programmer would write code like this. There is a better solution and you'll explore it in the next section.

The while loop in Java

```
// This is the Hello program in Java

 class Hello {

   public static void main (String args[]) {

     System.out.print("Hello ");   // Say Hello
     int i = 0;  // Declare and initialize loop counter
     while (i < args.length) { // Test and Loop
       System.out.print(args[i]);
       System.out.print(" ");
       i = i + 1;     // Increment Loop Counter
     }
     System.out.println();  // Finish the line
  }

}
```

The for loop in Java

```
// This is the Hello program in Java

 class Hello {

   public static void main (String args[]) {

     System.out.print("Hello ");   // Say Hello
     for (int i = 0; i < args.length; i = i + 1) { // Test and Loop
       System.out.print(args[i]);
       System.out.print(" ");
     }
     System.out.println();  // Finish the line
  }

}
```

Multiple Initializers and Incrementers

Sometimes it's necessary to initialize several variables before beginning a for loop. Similarly you may want to increment more than one variable. Java lets you do this by placing a comma between the different initializers and incrementers like this:

```
for (int i = 1,  j = 100;  i < 100;  i = i+1, j = j-1)  {
    System.out.println(i + j);
}
```

You can't, however, include multiple test conditions, at least not with commas. The following line is illegal and will generate a compiler error.

```
for (int i = 1,  j = 100;  i <= 100, j > 0;  i = i-1, j = j-1) {
```

To include multiple tests you need to use the boolean logic operators && and || which will be discussed later.

The `do while` loop in Java

```
// This is the Hello program in Java

 class Hello {

   public static void main (String args[]) {

     int i = -1;
     do {
       if (i == -1) System.out.print("Hello ");
       else {
         System.out.print(args[i]);
         System.out.print(" ");
       }
       i = i + 1;
     } while (i < args.length);
     System.out.println();  // Finish the line
  }

}
```

Booleans

Booleans are named after George Boole, a nineteenth century logician. Each boolean variable has one of two values, true or false. These are not the same as the Strings "true" and "false". They are not the same as any numeric value like 1 or 0. They are simply true and false. Booleans are not numbers; they are not Strings. They are simply booleans.

Boolean variables are declared just like any other variable.

```
boolean test1 = true;
boolean test2 = false;
```

Note that true and false are reserved words in Java. These are called the Boolean literals. They are case sensitive. True with a capital T is not the same as true with a little t. The same is true of False and false.

Relational Operators

Java has six relational operators that compare two numbers and return a boolean value. The relational operators are <, >, <=, >=, ==, and !=.

`x < y`	Less than	True if x is less than y, otherwise false.
`x > y`	Greater than	True if x is greater than y, otherwise false.
`x <= y`	Less than or equal to	True if x is less than or equal to y, otherwise false.
`x >= y`	Greater than or equal to	True if x is greater than or equal to y, otherwise false.
`x == y`	Equal	True if x equals y, otherwise false.
`x != y`	Not Equal	True if x is not equal to y, otherwise false.

Here are some code snippets showing the relational operators.

```
boolean test1 = 1 < 2;  // True. One is less that two.
boolean test2 = 1 > 2;  // False. One is not greater than two.
boolean test3 = 3.5 != 1;  // True. One does not equal 3.5
boolean test4 = 17*3.5 >= 67.0 - 42; //True. 59.5 is greater than 5
boolean test5 = 9.8*54 <= 654; // True. 529.2 is less than 654
boolean test6 = 6*4 == 3*8; // True. 24 equals 24
boolean test7 = 6*4 <= 3*8; // True. 24 is less than or equal to 24
boolean test8 = 6*4 < 3*8; // False. 24 is not less than 24
```

This, however, is an unusual use of booleans. Almost all use of booleans in practice comes in conditional statements and loop tests. You've already seen several examples of this. Earlier you saw this

```
if (args.length > 0) {
  System.out.println("Hello " + args[0]);
}
```

`args.length > 0` is a boolean value. In other words it is either `true` or it is `false`. You could write

```
boolean test = args.length > 0;
if (test) {
  System.out.println("Hello " + args[0]);
}
```

instead. However in simple situations like this the original approach is customary. Similarly the condition test in a `while` loop is a boolean. When you write `while (i < args.length)` the `i < args.length` is a boolean.

Relational Operator Precedence

Whenever a new operator is introduced you have to ask yourself where it fits in the precedence tree. If you look back at the example in the last section, you'll notice that it was implicitly assumed that the arithmetic was done before the comparison. Otherwise, for instance

```
boolean test8 = 6*4 < 3*8; // False. 24 is not less than 24
```

4 < 3 returns false which would then be multiplied by six and eight which would generate a compile time error because you can't multiply booleans. Relational operators are evaluated after arithmetic operators and before the assignment operator. == and != have slightly lower precedences than <, >, <= and >=. Here's the revised order:

1. *, /, % Do all multiplications, divisions and remainders from left to right.
2. +, - Next do additions and subtractions from left to right.
3. <, >, >=, <= Then any comparisons for relative size.
4. ==, != Then do any comparisons for equality and inequality
5. = Finally assign the right-hand side to the left-hand side

For example,

```
boolean b1 = 7 > 3 == true;
boolean b2 = true == 7 > 3;
b = 7 > 3;
```

Testing Objects for Equality

<, >, <= and >= can only be used with numbers and characters. They cannot be used with Strings, booleans, arrays or other compound types since there's no well-defined notion of order for these objects. Is true greater than false? Is "My only regret is that I have but one life to give for my country" greater than "I have a dream"?

Equality is a little easier to test however. `true` is equal to `true` and `true` is not equal to `false`. Similarly "My only regret is that I have but one life to give for my country" is not equal to "I have a dream." However you might be surprised if you ran this program:

```
class JackAndJill {

  public static void main(String args[]) {

    String s1 = new String("Jack went up the hill.");
    String s2 = new String("Jack went up the hill.");

    if ( s1 == s2 ) {
      System.out.println("The strings are the same.");
    }

    else if ( s1 != s2 ) {
      System.out.println("The strings are not the same.");
    }
  }
}
```

The result is

```
The strings are not the same.
```

Testing for Equality with `equals()`

That's not what you expected. To compare strings or any other kind of object you need to use the `equals(Object o)` method from `java.lang.String`. Below is a corrected version that works as expected. The reasons for this odd behavior go fairly deep into Java and the nature of object data types like strings.

```
class JackAndJill {

  public static void main(String args[]) {

    String s1 = new String("Jack went up the hill.");
    String s2 = new String("Jack went up the hill.");

    if ( s1.equals(s2) ) {
      System.out.println("The strings are the same.");
    }
    else {
      System.out.println("The strings are not the same.");
    }
  }
}
```

Break

A `break` statement exits a loop before an entry condition fails. For example, in this variation on the `CountWheat` program an error message is printed, and you break out of the `for` loop if `j` becomes negative.

```
class CountWheat  {

  public static void main (String args[]) {

    int total = 0;

    for (int square=1, int grains = 1; square <= 64; square++) {
      grains *= 2;
      if (grains <= 0) {
        System.out.println("Error: Overflow");
        break;
      }
      total += grains;
      System.out.print(total + "\t  ");
      if (square % 4 == 0) System.out.println();
    }
    System.out.println("All done!");

  }
```

```
}
```
Here's the output:
```
% javac CountWheat.java
% java CountWheat
2            6           14              30
62           126         254             510
1022         2046        4094            8190
16382        32766       65534           131070
262142       524286      1048574         2097150
4194302      8388606     16777214        33554430
67108862     134217726   268435454       536870910
1073741822   2147483646      Error: Overflow
All done!
%
```
The most common use of `break` is in `switch` statements.

Continue

It is sometimes necessary to exit from the middle of a loop. Sometimes you'll want to start over at the top of the loop. Sometimes you'll want to leave the loop completely. For these purposes Java provides the `break` and `continue` statements.

A `continue` statement returns to the beginning of the innermost enclosing loop without completing the rest of the statements in the body of the loop. If you're in a `for` loop, the counter is incremented. For example this code fragment skips even elements of an array

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

  if (m[i] % 2 == 0) continue;
  // process odd elements...

}
```
The `continue` statement is rarely used in practice, perhaps because most of the instances where it's useful have simpler implementations. For instance, the above fragment could equally well have been written as
```
for (int i = 0; i < m.length; i++) {

  if (m[i] % 2 != 0) {
    // process odd elements...

  }

}
```
There are only seven uses of `continue` in the entire Java 1.0.1 source code for the java packages.

Labeled Loops

Normally inside nested loops `break` and `continue` exit the innermost enclosing loop. For example consider the following loops.

```
for (int i=1; i < 10; i++) {
  for (int j=1; j < 4; j++) {
    if (j == 2) break;
    System.out.println(i + ", " + j);
  }
}
```

This code fragment prints

```
1, 1
2, 1
3, 1
4, 1
5, 1
6, 1
7, 1
8, 1
9, 1
```

because you break out of the innermost loop when j is two. However the outermost loop continues.

To break out of both loops, label the outermost loop and indicate that label in the break statement like this:

```
iloop: for (int i=1; i < 3; i++) {
  for (int j=1; j < 4; j++) {
    if (j == 2) break iloop;
    System.out.println(i + ", " + j);
  }
}
```

This code fragment prints

```
1, 1
```

and then stops because `j` is two and the outermost loop is exited.

The switch statement in Java

Switch statements are shorthands for a certain kind of `if` statement. It is not uncommon to see a stack of `if` statements all relate to the same quantity like this:

```
if (x == 0) doSomething0();
else if (x == 1) doSomething1();
else if (x == 2) doSomething2();
else if (x == 3) doSomething3();
else if (x == 4) doSomething4();
else doSomethingElse();
```

Java has a shorthand for these types of multiple `if` statements, the `switch-case` statement. Here's how you'd write the above using a `switch-case`:

```
switch (x) {
  case 0:
    doSomething0();
    break;
```

```
  case 1:
    doSomething1();
    break;
  case 2:
    doSomething2();
    break;
  case 3:
    doSomething3();
    break;
  case 4:
    doSomething4();
    break;
  default:
    doSomethingElse();
}
```

In this fragment `x` must be a variable or expression that can be cast to an `int` without loss of precision. This means the variable must be or the expression must return an `int`, `byte`, `short` or `char`. `x` is compared with the value of each the `case` statements in succession until one matches. This fragment compares `x` to literals, but these too could be variables or expressions as long as the variable or result of the expression is an `int`, `byte`, `short` or `char`. If no cases are matched, the default action is triggered.

Once a match is found, all subsequent statements are executed until the end of the `switch` block is reached or you break out of the block. This can trigger decidedly unexpected behavior. Therefore it is common to include the `break` statement at the end of each case block. It's good programming practice to put a `break` after each one unless you explicitly want all subsequent statements to be executed.

It's important to remember that the `switch` statement doesn't end when one case is matched and its action performed. The program then executes all statements that follow in that `switch` block until specifically told to break.

The ? : operator in Java

The value of a variable often depends on whether a particular boolean expression is or is not true and on nothing else. For instance one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) {
  max = a;
}
else {
  max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of `if-else` that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

`(a > b) ? a : b;` is an expression which returns one of two values, `a` or `b`. The condition, `(a > b)`, is tested. If it is true the first value, `a`, is returned. If it is false, the second value, `b`, is returned. Whichever value is returned is dependent on the conditional test, `a > b`. The condition can be any expression which returns a boolean value.

The ? : operator in Java

The conditional operator only works for assigning a value to a variable, using a value in a method invocation, or in some other way that indicates the type of its second and third arguments. For example, consider the following

```
if (name.equals("Rumplestiltskin")) {
  System.out.println("Give back child");
}
else {
  System.out.println("Laugh");
}
```

This may **not** be written like this:

```
name.equals("Rumplestiltskin")
 ? System.out.println("Give back child")
 : System.out.println("Laugh");
```

First of all, both the second and third arguments are void. Secondly, no assignment is present to indicate the type that is expected for the second and third arguments (though you know void must be wrong).

The first argument to the conditional operator must have or return boolean type and the second and third arguments must return values compatible with the value the entire expression can be expected to return. You can never use a `void` method as an argument to the `? :` operator.

Logical Operators in Java

The relational operators you've learned so far (<, <=, >, >=, !=) are sufficient when you only need to check one condition. However what if a particular action is to be taken only if several conditions are true? You can use a sequence of `if` statements to test the conditions, as follows:

```
if (x == 2) {
  if (y != 2) {
     System.out.println("Both conditions are true.");
   }
}
```

This, however, is hard to write and harder to read. It only gets worse as you add more conditions. Fortunately, Java provides an easy way to handle multiple conditions: the logic operators. There are three logic operators, `&&`, `||` and `!`.

`&&` is logical and. `&&` combines two boolean values and returns a boolean which is true if and only if both of its operands are true. For instance

```
boolean b;
b = 3 > 2 && 5 < 7; // b is true
b = 2 > 3 && 5 < 7; // b is now false
```

`||` is logical or. `||` combines two boolean variables or expressions and returns a result that is true if either or both of its operands are true. For instance

```
boolean b;
b = 3 > 2 || 5 < 7; // b is true
b = 2 > 3 || 5 < 7; // b is still true
b = 2 > 3 || 5 > 7; // now b is false
```

The last logic operator is `!` which means not. It reverses the value of a boolean expression. Thus if `b` is true `!b` is false. If `b` is false `!b` is true.

```
boolean b;
b = !(3 > 2); // b is false
b = !(2 > 3); // b is true
```

These operators allow you to test multiple conditions more easily. For instance the previous example can now be written as

```
if (x == 2 && y != 2) {
  System.out.println("Both conditions are true.");
}
```

That's a lot clearer.

The Order of Evaluation of Logic Operators

When Java sees a `&&` operator or a `||`, the expression on the left side of the operator is evaluated first. For example, consider the following:

```
boolean b, c, d;
b = !(3 > 2); // b is false
c = !(2 > 3); // c is true
d = b && c; // d is false
```

When Java evaluates the expression `d = b && c;`, it first checks whether `b` is true. Here `b` is false, so `b && c` must be false regardless of whether `c` is or is not true, so Java doesn't bother checking the value of `c`.

On the other hand when faced with an `||` Java short circuits the evaluation as soon as it encounters a true value since the resulting expression must be true. This short circuit evaluation is less important in Java than in C because in Java the operands of `&&` and `||` must be booleans which are unlikely to have side effects that depend on whether or not they are evaluated. Still it's possible to force them. For instance consider this code.

```
boolean b = (n == 0) || (m/n > 2);
```

Even if `n` is zero this line will never cause a division by zero, because the left hand side is always evaluated first. If `n` is zero then the left hand side is true and there's no need to evaluate the right

hand side. Mathematically this makes sense because `m/0` is in some sense infinite which is greater than two.

This isn't a perfect solution though because `m` may be 0 or it may be negative. If `m` is negative and `n` is zero then `m/n` is negative infinity which is less than two. And if `m` is also zero, then `m/n` is very undefined.

The proper solution at this point depends on your problem. Since real world quantities aren't infinite, when infinities start popping up in your programs, nine times out of ten it's a sign that you've lost too much precision. The remaining times are generally signals that you've left out some small factor in your physical model that would remove the infinity.

Therefore if there's a real chance your program will have a divide by zero error think carefully about what it means and how you should respond to it. If, upon reflection, you decide that what you really want to know is whether `m/n` is finite and greater than zero you should use a line like this

```
boolean b = (n != 0) && (m/n > 0);
```

Avoiding Short Circuits

If you want all of your boolean expressions evaluated regardless of the truth value of each, then you can use `&` and `|` instead of `&&` and `||`. However make sure you use these only on boolean expressions. Unlike `&&` and `||`, `&` and `|` also have a meaning for numeric types which is completely different from their meaning for booleans.

Precedence

Finally let's add the `&&`, `||`, `&`, `|` and `?` operators to the precedence table

1. *, /, % Multiplicative operators
2. +, - Additive operators
3. <, >, >=, <= Relational operators
4. ==, != Then do any comparisons for equality and inequality
5. & Bitwise and
6. | Bitwise or
7. && Logical and
8. || Logical or
9. ? : Conditional operator
10. = Assignment operator

Declaring Arrays

Like all other variables in Java, an array must have a specific type like `byte`, `int`, `String` or `double`. Only variables of the appropriate type can be stored in an array. One array cannot store both ints and Strings, for instance.

Like all other variables in Java an array must be declared. When you declare an array variable you suffix the type with `[]` to indicate that this variable is an array. Here are some examples:

```
int[] k;
float[] yt;
String[] names;
```

This says that `k` is an array of ints, `yt` is an array of floats and `names` is an array of Strings. In other words you declare an array like you declare any other variable except that you append brackets to the end of the type.

You also have the option to append the brackets to the variable instead of the type.

```
int k[];
float yt[];
String names[];
```
The choice is primarily one of personal preference. You can even use both at the same time like this
```
int[] k[];
float[] yt[];
String[] names[];
```

Creating Arrays

Declaring arrays merely says what kind of values the array will hold. It does not create them. Java arrays are objects, and like any other object you use the `new` keyword to create them. When you create an array, you must tell the compiler how many components will be stored in it. Here's how you'd create the variables declared on the previous page:

```
k = new int[3];
yt = new float[7];
names = new String[50];
```

The numbers in the brackets specify the length of the array; that is, how many slots it has to hold values. With the lengths above `k` can hold three ints, `yt` can hold seven floats and `names` can hold fifty Strings. This step is sometimes called allocating the array since it sets aside the memory the array requires.

Initializing Arrays

Individual components of an array are referenced by the array name and by an integer which represents their position in the array. The numbers you use to identify them are called subscripts or indexes into the array.

Subscripts are consecutive integers beginning with 0. Thus the array `k` above has components `k[0]`, `k[1]`, and `k[2]`. Since you start counting at zero there is no `k[3]`, and trying to access it will throw an `ArrayIndexOutOfBoundsException`. You can use array components wherever you'd use a similarly typed variable that wasn't part of an array. For example this is how you'd store values in the arrays above:

```
k[0] = 2;
k[1] = 5;
k[2] = -2;
yt[17] = 7.5f;
names[4] = "Fred";
```

This step is called initializing the array or, more precisely, initializing the components of the array. Sometimes the phrase "initializing the array" is used to mean when you initialize all the components of the array.

For even medium sized arrays, it's unwieldy to specify each component individually. It is often helpful to use for loops to initialize the array. Here is a loop which fills an array with the squares of the numbers from 0 to 100.

```
float[] squares;
squares = new float[101];

for (int i=0; i <= 100; i++) {
   squares[i] = i*i;
}
```

Two things you should note about this code fragment:

1. Watch the fenceposts! Since array subscripts begin at zero you need 101 components if you want to include the square of 100.
2. Although i is an int, it is promoted to a float when it is stored in squares, since squares is declared to be an array of floats.

System.arraycopy()

Although copying an array isn't particularly difficult, it is an operation which benefits from a native implementation. Therefore `java.lang.System` includes a static `System.arraycopy()` method you can use to copy one array to another.

```
public static void arraycopy(Object source, int sourcePosition,
```

`Object destination, int destinationPosition, int numberOfElements)`
`System.arraycopy()` copies `numberOfElements` elements from from the array `source`, beginning with the element at `sourcePosition`, to the array `destination` starting at `destinationPosition`. The `destination` array must already exist when `System.arraycopy()` is called. The method does not create it. The `source` and `destination` arrays must be of the same type.

For example,

```
int[] unicode = new int[65536];
 for (int i = 0; i < unicode.length; i++) {
   unicode[i] = i;
 }
 int[] latin1 = new int[256];
 System.arraycopy(unicode, 0, latin1, 0, 256);
```

Multi-Dimensional Arrays

So far all these arrays have been one-dimensional. That is, a single number could locate any value in the array. However sometimes data is naturally represented by more than one number. For instance a position on the earth requires a latitude and a longitude.

The most common kind of multidimensional array is the two-dimensional array. If you think of a one-dimensional array as a column of values, you can think of a two-dimensional array as a table of values like this

	c0	c1	c2	c3
r0	0	1	2	3
r1	1	2	3	4
r2	2	3	4	5
r3	3	4	5	6
r4	4	5	6	7

Here we have an array with five rows and four columns. It has twenty total elements. However we say it has dimension five by four, not dimension twenty. This array is not the same as a four by five array like this one:

	c0	c1	c2	c3	c4
r0	0	1	2	3	4
r1	1	2	3	4	5
r2	2	3	4	5	6
r3	3	4	5	6	7

We need to use two numbers to identify a position in a two-dimensional array. These are the element's row and column positions. For instance if the above array is called `J` then `J[0][0]` is 0, `J[0][1]` is 1, `J[0][2]` is 2, `J[0][3]` is 3, `J[1][0]` is 1, and so on.

Here's how the elements in a four by five array called `M` are referred to:

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]
M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]

Declaring, Allocating and Initializing Two Dimensional Arrays

Two dimensional arrays are declared, allocated and initialized much like one dimensional arrays. However you have to specify two dimensions rather than one, and you typically use two nested for loops to fill the array.

This example fills a two-dimensional array with the sum of the row and column indexes

```
class FillArray {

  public static void main (String args[]) {

    int[][] matrix;
    matrix = new int[4][5];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        matrix[row][col] = row+col;
      }
    }

  }

}
```

Of course the algorithm you use to fill the array depends completely on the use to which the array is to be put. The next example calculates the identity matrix for a given dimension. The identity matrix of dimension N is a square matrix which contains ones along the diagonal and zeros in all other positions.

```
class IDMatrix {

  public static void main (String args[]) {
```

```
    double[][] id;
    id = new double[4][4];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 4; col++) {
        if (row != col) {
          id[row][col]=0.0;
        }
        else {
          id[row][col] = 1.0;
        }
      }
    }

  }

}
```

In two-dimensional arrays `ArrayIndexOutOfBoundsException`s occur whenever you exceed the maximum column index or row index.

You can also declare, allocate, and initialize a a two-dimensional array at the same time by providing a list of the initial values inside nested brackets. For instance the three by three identity matrix could be set up like this:

```
double[][] ID3 = {
  {1.0, 0.0, 0.0},
  {0.0, 1.0, 0.0},
  {0.0, 0.0, 1.0}
};
```

The spacing and the line breaks used above are purely for the programmer. The compiler doesn't care. The following works equally well:

```
double[][] ID3 = {{1.0, 0.0, 0.0},{0.0, 1.0, 0.0},{0.0, 0.0, 1.0}};
```

Even Higher Dimensions

You don't have to stop with two dimensional arrays. Java permits arrays of three, four or more dimensions. However chances are pretty good that if you need more than three dimensions in an array, you're probably using the wrong data structure. Even three dimensional arrays are exceptionally rare outside of scientific and engineering applications.

The syntax for three dimensional arrays is a direct extension of that for two-dimensional arrays. The program below declares, allocates and initializes a three-dimensional array. The array is filled with the sum of its indexes.

```
class Fill3DArray {
```

```
  public static void main (String args[]) {

    int[][][] M;
    M = new int[4][5][3];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        for (int ver=0; ver < 3; ver++) {
          M[row][col][ver] = row+col+ver;
        }
      }
    }

  }

}
```

You need the additional nested `for` loop to handle the extra dimension. The syntax for still higher dimensions is similar. Just add another pair of brackets and another dimension.

Unbalanced Arrays

Like C Java does not have true multidimensional arrays. Java fakes multidimensional arrays using arrays of arrays. This means that it is possible to have unbalanced arrays. An unbalanced array is a multidimensional array where the dimension isn't the same for all rows. In most applications this is a horrible idea and should be avoided.

Exercises

1. Sales tax in New York City is 8.25%. Write a program that accepts a price on the command line and prints out the appropriate tax and total purchase price.
2. Modify the sales tax program to accept an arbitrary number of prices, total them, calculate the sales tax and print the total amount.
3. Write a program that reads two numbers from the command line, the number of hours worked by an employee and their base pay rate. Then output the total pay due.
4. Modify the previous program to meet the U.S. Dept. of Labor's requirement for time and a half pay for hours over forty worked in a given week.
5. Add warning messages to the payroll program if the pay rate is less than the minimum wage ($5.15 an hour as of 1998) or if the employee worked more than the number of hours in a week.
6. Write a program that reads an integer n from the command line and calculates n! (n factorial).
7. There are exactly 2.54 centimeters to an inch. Write a program that takes a number of inches from the command line and converts it to centimeters.
8. Write the inverse program that reads a number of centimeters from the command line and converts it to inches.

9. There are 454 grams in a pound and 1000 grams in a kilogram. Write programs that convert pounds to kilograms and kilograms to pounds. Read the number to be converted from the command line. Can you make this one program instead of two?
10. The equivalent resistance of resistors connected in series is calculated by adding the resistances of the individual resistors. Write a program that accepts a series of resistances from the command line and outputs the equivalent resistance.

What is Object Oriented Programming?

In classic, procedural programming you try to make the real world problem you're attempting to solve fit a few, pre-determined data types: integers, floats, Strings, and arrays perhaps. In object oriented programming you create a model for a real world system. Classes are programmer-defined types that model the parts of the system.

A *class* is a programmer defined type that serves as a blueprint for instances of the class. You can still have ints, floats, Strings, and arrays; but you can also have cars, motorcycles, people, buildings, clouds, dogs, angles, students, courses, bank accounts, and any other type that's important to your problem.

Classes specify the data and behavior possessed both by themselves and by the objects built from them. A class has two parts: the fields and the methods. Fields describe what the class is. Methods describe what the class does.

Using the blueprint provided by a class, you can create any number of objects, each of which is called an *instance* of the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. For example, all humans have eye color but the color of each human's eyes can be different from others.

On the other hand, objects have the same methods as all other objects in the class except in so far as the methods depend on the value of the fields and arguments to the method.

This dichotomy is reflected in the runtime form of objects. Every object has a separate block of memory to store its fields, but the bytes in the actual methods are shared between all objects in a class.

Another common analogy is that a class is to an object as a cookie cutter is to a cookie. One cookie cutter can make many cookies. There may be only one class, but there can be many objects in that class. Each object is an instance of one class.

Example 1: The Car Class

Suppose you need to write a traffic simulation program that watches cars going past an intersection. Each car has a speed, a maximum speed, and a license plate that uniquely identifies it. In traditional programming languages you'd have two floating point and one string variable for each car. With a class you combine these into one thing like this.

```
class Car {

  String licensePlate; // e.g. "New York 543 A23"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour

}
```

These variables (`licensePlate`, `speed` and `maxSpeed`) are called the *member variables*, *instance variables*, or *fields* of the class.

Fields tell you what a class is and what its properties are.

An *object* is a specific instance of a class with particular values (possibly mutable) for the fields. While a class is a general blueprint for objects, an instance is a particular object.

Note the use of comments to specify the units. That's important. A unit confusion between pounds and newtons led to the loss of NASA's $94 million Mars Climate Orbiter. (Believe it or not that's a cheap mission by NASA standards. If you're rich enough that you don't have to worry about losing $94 million worth of work, you don't have to put comments in your source code. Everybody else has to use comments.)

How would you write an `Angle` class?

Constructing objects with `new`

```
class Car {

  String licensePlate; // e.g. "New York 543 A23"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour

}
```

To instantiate an object in Java, use the keyword `new` followed by a call to the class's constructor. Here's how you'd create a new `Car` variable called `c`:

```
  Car c;
  c = new Car();
```

The first word, `Car`, declares the type of the variable `c`. Classes are types and variables of a class type need to be declared just like variables that are ints or doubles.

The equals sign is the assignment operator and `new` is the construction operator.

Finally notice the `Car()` method. The parentheses tell you this is a method and not a data type like the `Car` on the left hand side of the assignment. This is a constructor, a method that creates a new instance of a class. You'll learn more about constructors shortly. However if you do nothing, then the compiler inserts a default constructor that takes no arguments.

This is often condensed into one line like this:

```
  Car c = new Car();
```

The Member Access Separator .

```
class Car {

  String licensePlate; // e.g. "New York 543 A23"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour

}
```

Once you've constructed a car, you want to do something with it. To access the fields of the car you use the `.` separator. The `Car` class has three fields

- `licensePlate`
- `speed`
- `maxSpeed`

Therefore if `c` is a `Car` object, `c` has three fields as well:

- `c.licensePlate`
- `c.speed`
- `c.maxSpeed`

You use these just like you'd use any other variables of the same type. For instance:

```
    Car c = new Car();

    c.licensePlate = "New York A45 636";
    c.speed = 70.0;
    c.maxSpeed = 123.45;

    System.out.println(c.licensePlate + " is moving at " + c.speed +
      "kilometers per hour.");
```

The `.` separator selects a specific member of a `Car` object by name.

Using a `Car` object in a different class

```
class Car {

  String licensePlate; // e.g. "New York 543 A23"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour

}
```

The next program creates a new car, sets its fields, and prints the result:

```
class CarTest {

  public static void main(String args[]) {

    Car c = new Car();

    c.licensePlate = "New York A45 636";
    c.speed = 70.0;
    c.maxSpeed = 123.45;

    System.out.println(c.licensePlate + " is moving at " + c.speed +
      "kilometers per hour.");
  }

}
```

This program requires not just the `CarTest` class but also the `Car` class. To make them work together put the `Car` class in a file called Car.java. Put the `CarTest` class in a file called CarTest.java. Put both these files in the same directory. Then compile both files in the usual way. Finally run `CarTest`. For example,

```
% javac Car.java
% javac CarTest.java
% java CarTest
New York A45 636 is moving at 70.0 kilometers per hour.
```

Note that `Car` does not have a `main()` method so you cannot run it. It can exist only when called by other programs that do have `main()` methods.

Many of the applications you write from now on will use multiple classes. It is customary in Java to put every class in its own file. Next week, you'll learn how to use packages to organize your commonly used classes in different directories. For now keep all your .java source code and .class byte code files in one directory.

Initializing Fields

Fields can (and often should) be initialized when they're declared, just like local variables.

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed     = 120.0; // in kilometers per hour

}
```

The next program creates a new car and prints it:

```
class CarTest2 {

  public static void main(String[] args) {

    Car c = new Car();

    System.out.println(c.licensePlate + " is moving at " + c.speed +
      "kilometers per hour.");
  }

}
```

For example,

```
% javac Car.java
% javac CarTest.java
% java CarTest
 is moving at 0.0 kilometers per hour.
```

Methods

Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class *is*. Methods say what a class *does*. The fields and methods of a class are collectively referred to as the *members of the class*.

The classes you've encountered up till now have mostly had a single method, `main()`. However, in general classes can have many different methods that do many different things. For instance the `Car` class might have a method to make the car go as fast as it can. For example,

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed;    = 120.0; // in kilometers per hour
```

```
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }

}
```

The fields are the same as before, but now there's also a method called `floorIt()`. It begins with the Java keyword `void` which is the return type of the method. Every method must have a return type which will either be `void` or some data type like `int`, `byte`, `float`, or `String`. The return type says what kind of the value will be sent back to the calling method when all calculations inside the method are finished. If the return type is `int`, for example, you can use the method anywhere you use an `int` constant. If the return type is `void` then no value will be returned.

`floorIt` is the name of this method. The name is followed by two empty parentheses. Any arguments passed to the method would be passed between the parentheses, but this method has no arguments. Finally an opening brace (`{`) begins the body of the method.

There is one statement inside the method

```
this.speed = this.maxSpeed;
```

Notice that within the `Car` class the field names are prefixed with the keyword `this` to indicate that I'm referring to fields in the current object.

Finally the `floorIt()` method is closed with a `}` and the class is closed with another `}`.

Question: what are some other methods this class might need? Or, another way of putting it, what might you want to do with a `Car` object?

Invoking Methods

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed;    = 120.0; // in kilometers per hour

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }



}
```

Outside the `Car` class, you call the `floorIt()` method just like you reference fields, using the name of the object you want to accelerate to maximum and the `.` separator as demonstrated below

```
class CarTest3 {

  public static void main(String args[]) {

    Car c = new Car();

    c.licensePlate = "New York A45 636";
    c.maxSpeed = 123.45;

    System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");

    c.floorIt();

    System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");

  }

}
```

The output is:

```
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
```

The `floorIt()` method is completely enclosed within the `Car` class. Every method in a Java program must belong to a class. Unlike C++ programs, Java programs cannot have a method hanging around in global space that does everything you forgot to do inside your classes.

Implied `this`

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed;    = 120.0; // in kilometers per hour

  void floorIt() {
    speed = maxSpeed;
  }

}
```

Within the `Car` class, you don't absolutely need to prefix the field names with `this.` like `this.licensePlate` or `this.speed`. Just `licensePlate` and `speed` are sufficient. The `this.` may be implied. That's because the `floorIt()` method must be called by a specific instance of the `Car` class, and this instance knows what its data is. Or, another way of looking at it, the every object has its own `floorIt()` method.

For clarity, I will use an explicit `this` today, and I recommend you do so too, at least initially. As you become more comfortable with Java, classes, references, and OOP, you will be able to leave out the `this` without fear of confusion. Most real-world code does not use an explicit `this`.

Member Variables vs. Local Variables

```
class Car {

  String licensePlate = "";    // member variable
  double speed;       = 0.0;   // member variable
  double maxSpeed;    = 120.0; // member variable

  boolean isSpeeding() {
    double excess;    // local variable
    excess = this.maxSpeed - this.speed;
    if (excess < 0) return true;
    else return false;
  }

}
```

Until now all the programs you've seen quite simple in structure. Each had exactly one class. This class had a single method, `main()`, which contained all the program logic and variables. The variables in those classes were all local to the `main()` method. They could not be accessed by anything outside the `main()` method. These are called *local variables*.

This sort of program is the amoeba of Java. Everything the program needs to live is contained inside a single cell. It's quite an efficient arrangement for small organisms, but it breaks down when you want to design something bigger or more complex.

The `licensePlate`, `speed` and `maxSpeed` variables of the `Car` class, however, belong to a `Car` object, not to any individual method. They are defined outside of any methods but inside the class and are used in different methods. They are called *member variables* or *fields*.

Member variable, *instance variable*, and *field* are different words that mean the same thing. *Field* is the preferred term in Java. *Member variable* is the preferred term in C++.

A *member* is not the same as a member variable or field. Members include both fields and methods.

Passing Arguments to Methods

It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the

implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

To do this you need to be able to send information into the `Car` class. This is done by passing arguments. For example, to allow other objects to change the value of the `speed` field in a `Car` object, the `Car` class could provide an `accelerate()` method. This method does not allow the car to exceed its maximum speed, or to go slower than 0 kph.

```
 void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }
```

The first line of the method is called its *signature*. The signature

```
void accelerate(double deltaV)
```

indicates that `accelerate()` returns no value and takes a single argument, a `double` which will be referred to as `deltaV` inside the method.

`deltaV` is a purely *formal* argument.

Java passes method arguments by value, not by reference.

Passing Arguments to Methods, An Example

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed;    = 120.0; // in kilometers per hour

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }

  void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
```

```
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }

}
```

```
class CarTest4 {

  public static void main(String args[]) {

    Car c = new Car();

    c.licensePlate = "New York A45 636";
    c.maxSpeed = 123.45;

    System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c.licensePlate + " is moving at " + c.speed +
       " kilometers per hour.");
    }

  }

}
```

Here's the output:

```
utopia% java CarTest4
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 40.0 kilometers per hour.
New York A45 636 is moving at 50.0 kilometers per hour.
New York A45 636 is moving at 60.0 kilometers per hour.
New York A45 636 is moving at 70.0 kilometers per hour.
New York A45 636 is moving at 80.0 kilometers per hour.
New York A45 636 is moving at 90.0 kilometers per hour.
New York A45 636 is moving at 100.0 kilometers per hour.
New York A45 636 is moving at 110.0 kilometers per hour.
New York A45 636 is moving at 120.0 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
```

Setter Methods

Setter methods, also known as mutator methods, merely set the value of a field to a value specified by the argument to the method. These methods almost always return `void`.

One common idiom in setter methods is to use `this.`*`name`* to refer to the field and give the argument the same name as the field. For example,

```
class Car {

  String licensePlate; // e.g. "New York A456 324"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour

  // setter method for the license plate property
  void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  // setter method for the maxSpeed property
  void setMaximumSpeed(double maxSpeed) {
    if (maxSpeed > 0) this.maxSpeed = maxSpeed;
    else this.maxSpeed = 0.0;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }

  void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }

}
```

Using Setter Methods, An Example

```
class CarTest5 {

  public static void main(String args[]) {

    Car c = new Car();

    c.setLicensePlate("New York A45 636");
    c.setMaximumSpeed(123.45);

    System.out.println(c.licensePlate + " is moving at " + c.speed +
```

```
     " kilometers per hour.");

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c.licensePlate + " is moving at " + c.speed +
       " kilometers per hour.");
    }

  }

}
```

Here's the output:

```
utopia% java CarTest5
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 40.0 kilometers per hour.
New York A45 636 is moving at 50.0 kilometers per hour.
New York A45 636 is moving at 60.0 kilometers per hour.
New York A45 636 is moving at 70.0 kilometers per hour.
New York A45 636 is moving at 80.0 kilometers per hour.
New York A45 636 is moving at 90.0 kilometers per hour.
New York A45 636 is moving at 100.0 kilometers per hour.
New York A45 636 is moving at 110.0 kilometers per hour.
New York A45 636 is moving at 120.0 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
```

Returning Values From Methods

It's often useful to have a method return a value to the class that called it. This is accomplished by the `return` keyword at the end of a method and by declaring the data type that is returned by the method at the beginning of the method.

For example, the following `getLicensePlate()` method returns the current value of the `licensePlate` field in the `Car` class.

```
 String getLicensePlate() {
    return this.licensePlate;
  }
```

A method like this that merely returns the value of an object's field or property is called a *getter* or *accessor* method.

The signature `String getLicensePlate()` indicates that `getLicensePlate()` returns a value of type `String` and takes no arguments. Inside the method the line

```
return this.licensePlate;
```

returns the `String` contained in the `licensePlate` field to whoever called this method. It is important that the type of value returned by the return statement match the type declared in the method signature. If it does not, the compiler will complain.

Returning Multiple Values From Methods

It is not possible to return more than one value from a method. You cannot, for example, return the `licensePlate`, `speed` and `maxSpeed` fields from a single single method. You could combine them into an object of some kind and return the object. However this would be poor object oriented design.

The right way to solve this problem is to define three separate methods, `getSpeed()`, `getMaxSpeed()`, and `getLicensePlate()`, each of which returns its respective value. For example,

```
class Car {

  String licensePlate = "";    // e.g. "New York 543 A23"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed;    = 120.0; // in kilometers per hour

  // getter (accessor) methods
  String getLicensePlate() {
    return this.licensePlate;
  }

  double getMaxSpeed() {
    return this.maxSpeed;
  }

  double getSpeed() {
    return this.speed;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }

  void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }
```

```
  }

}
```

Using Getter Methods, An Example

```
class CarTest6 {

  public static void main(String args[]) {

    Car c = new Car();

    c.setLicensePlate("New York A45 636");
    c.setMaximumSpeed(123.45);

    System.out.println(c.getLicensePlate() + " is moving at "
     + c.getSpeed() + " kilometers per hour.");

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c.getLicensePlate() + " is moving at "
       + c.getSpeed() + " kilometers per hour.");
    }

  }

}
```

There's no longer any direct access to fields!

Here's the output:

```
utopia% java CarTest6
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 40.0 kilometers per hour.
New York A45 636 is moving at 50.0 kilometers per hour.
New York A45 636 is moving at 60.0 kilometers per hour.
New York A45 636 is moving at 70.0 kilometers per hour.
New York A45 636 is moving at 80.0 kilometers per hour.
New York A45 636 is moving at 90.0 kilometers per hour.
New York A45 636 is moving at 100.0 kilometers per hour.
New York A45 636 is moving at 110.0 kilometers per hour.
New York A45 636 is moving at 120.0 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
```

Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line
`Car c = new Car();`
`Car()` is the constructor. A constructor has the same name as the class.

If no constructor exists Java provides a generic one that takes no arguments (a *noargs constructor*), but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called `Car()`.

Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.

The following method is a constructor that initializes license plate to an empty string, speed to zero, and maximum speed to 120.0.

```
Car() {
   this.licensePlate = "";
   this.speed  = 0.0;
   this.maxSpeed = 120.0;
 }
```

Better yet, you can create a constructor that accepts three arguments and use those to initialize the fields as below.

```
Car(String licensePlate, double speed, double maxSpeed) {

   this.licensePlate = licensePlate;
   this.speed  = speed;
   if (maxSpeed > 0) this.maxSpeed = maxSpeed;
   else this.maxSpeed = 0.0;
   if (speed > this.maxSpeed) this.speed = this.maxSpeed;
   if (speed < 0) this.speed = 0.0;
   else this.speed = speed;

 }
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified:

```
Car(String licensePlate, double maxSpeed) {

   this.licensePlate = licensePlate;
   this.speed  = 0.0;
   if (maxSpeed > 0) this.maxSpeed = maxSpeed;
   else this.maxSpeed = 0.0;

 }
```

Constructors

Here's the complete class:

```
class Car {

  String licensePlate; // e.g. "New York A456 324"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour

  Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed  = 0.0;
    if (maxSpeed > 0) this.maxSpeed = maxSpeed;
    else this.maxSpeed = 0.0;

  }

  // getter (accessor) methods
  String getLicensePlate() {
    return this.licensePlate;
  }

  double getMaxSpeed() {
    return this.maxSpeed;
  }

  double getSpeed() {
    return this.speed;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt() {
    this.speed = this.maxSpeed;
  }

  void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }

}
```

Notice that I've taken out several things:

- the initialization of the fields
- the setter methods

Using Constructors

The next program uses the constructor to initialize a car rather than setting the fields directly.

```
class CarTest7 {

  public static void main(String args[]) {

    Car c = new Car("New York A45 636", 123.45);

    System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() +
     " kilometers per hour.");

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed()
       + " kilometers per hour.");
    }

  }

}
```

You no longer need to know about the fields `licensePlate`, `speed` and `maxSpeed`. All you need to know is how to construct a new car and how to print it.

You may ask whether the `setLicensePlate()` method is still needed since it's now set in a constructor. The general answer to this question depends on the use to which the `Car` class is to be put. The specific question is whether a car's license plate may need to be changed after the `Car` object is created.

Some classes may not change after they're created; or, if they do change, they'll represent a different object. The most common such class is `String`. You cannot change a string's data. You can only create a new `String` object. Such objects are called *immutable*.

Constraints

One of the reasons to use constructors and setter methods rather than directly accessing fields is to enforce constraints. For instance, in the `Car` class it's important to make sure that the speed is always less than or equal to the maximum speed and that both speed and maximum speed are greater than or equal to zero.

You've already seen one example of this in the `accelerate()` method which will not accelerate a car past its maximum speed.

```
  void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
```

```
      this.speed = this.maxSpeed;
    }
    if (this.speed < 0.0) {
      this.speed = 0.0;
    }

  }
```

You can also insert constraints like that in the constructor. For example, this `Car` constructor makes sure that the maximum speed is greater than or equal to zero:

```
 Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed  = 0.0;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
    }

  }
```

Access Protection

Global variables are a classic cause of bugs in most programming languages. Some unknown function can change the value of a variable when the programmer isn't expecting it to change. This plays all sorts of havoc.

Most OOP languages including Java allow you to protect variables from external modification. This allows you to guarantee that your class remains consistent with what you think it is as long as the methods of the class themselves are bug-free. For example, int the Car class we'd like to make sure that no block of code in some other class is allowed to make the speed greater than the maximum speed. We want a way to make the following illegal:

```
 Car c = new Car("New York A234 567", 100.0);
  c.speed = 150.0;
```

This code violates the constraints we've placed on the class. We want to allow the compiler to enforce these constraints.

A class presents a picture of itself to the world. (This picture is sometimes called an *interface*, but the word *interface* has a more specific meaning in Java.) This picture says that the class has certain methods and certain fields. Everything else about the class including the detailed workings of the class's methods is hidden. As long as the picture the class shows to the world doesn't change, the programmer can change how the class implements that picture. Among other advantages this allows the programmer to change and improve the algorithms a class uses without worrying that some piece of code depends in unforeseen ways on the details of the algorithm used. This is called encapsulation.

Another way to think about encapsulation is that a class signs a contract with all the other classes in the program. This contract says that a class has methods with unambiguous names which take particular types of arguments and return a particular type of value. The contract may also say that a class has fields with given names and of a given type. However the contract does not specify how the methods are implemented. Furthermore, it does not say that there aren't other private fields and methods which the class may use. A contract guarantees the presence of certain methods and fields. It does not exclude all other methods and fields. This contract is implemented through access protection. Every class, field and method in a Java program is defined as either public, private, protected or unspecified.

You're closer to your immediate family (your parents and your children) than you are to your cousins. You're closer to your cousins than to the general public at large, but there are some things you don't tell anybody. Furthermore, your family is not my family.

Examples of Access Protection

This is how the `Car` class would probably be written in practice. Notice that all the fields are now declared `private`, and they are accessed only through `public` methods. This is the normal pattern for all but the simplest classes.

```
public class Car {

  private String licensePlate; // e.g. "New York A456 324"
  private double speed;        // kilometers per hour
  private double maxSpeed;     // kilometers per hour

  public Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed  = 0.0;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
    }

  }

  // getter (accessor) methods
  public String getLicensePlate() {
    return this.licensePlate;
  }

  public double getSpeed() {
    return this.speed;
  }

  public double getMaxSpeed() {
    return this.maxSpeed;
```

```
  }

  // setter method for the license plate property
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  public void floorIt() {
    this.speed = this.maxSpeed;
  }

  public void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }

}
```

In many cases there will also be `private`, `protected` and default access methods as well. Collectively these are called *non-public* methods.

In many cases, the fields may be protected or default access. However public fields are rare. This allows programmers to change the implementation of a class while still maintaining the same contract with the outside world.

Dynamic vs static linking.

Examples of Access Protection

Now let's try to directly access the fields from another class and see what happens:

```
class CarTest8 {

  public static void main(String args[]) {

    Car c = new Car("New York A45 636", 100.0);

    c.licensePlate = "New York A45 636";
    c.speed = 0.0;
    c.maxSpeed = 123.45;

    System.out.println(c.licensePlate + " is moving at " + c.speed +
```

```
     " kilometers per hour.");

    c.floorIt();

    System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");

  }

}
```

Here's what happens when you try to compile it against the revised `Car` class:

```
% javac Car.java
% javac CarTest8.java
CarTest8.java:7: Variable licensePlate in class Car not accessible from class
CarTest8.
    c.licensePlate = "New York A45 636";
     ^
CarTest8.java:8: Variable speed in class Car not accessible from class
CarTest8.
    c.speed = 0.0;
     ^
CarTest8.java:9: Variable maxSpeed in class Car not accessible from class
CarTest8.
    c.maxSpeed = 123.45;
     ^
CarTest8.java:11: Variable licensePlate in class Car not accessible from class
CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed +
                       ^
CarTest8.java:11: Variable speed in class Car not accessible from class
CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed +
                                                           ^
CarTest8.java:16: Variable licensePlate in class Car not accessible from class
CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed +
                       ^
CarTest8.java:16: Variable speed in class Car not accessible from class
CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed +
                                                           ^
7 errors
%
```

The Four Levels of Access Protection

Any two different Java objects have one of four relations to each other. The four relations are:

- The objects are in the same class.
- One object is a subclass of the other object's class.
- The objects are in the same package.
- None of the above. (Both objects are members of the general public.)

These relationships are not mutually exclusive. One object can be a subclass of another object in the same package, for example.

You can define which of your class's members, that is its fields and its methods, are accessible to other objects in each of these four groups, relative to the current class.

If you want any object at all to be able to call a method or change a field, declare it `public`.

If you want only objects in the same class to be able to get or set the value of a field or invoke a method, declare it `private`.

If you want access restricted to subclasses and members of the same package, declare it `protected`.

Finally, to restrict access only to objects in the same package, use no access declaration at all. This is called "package" or "default" access, but it has no keyword. The `default` keyword means something else entirely.

Can anyone remember what?

By default, all classes you write are in the same package. However, they are in different packages from the Java classes like `System` or `Applet`.

The public fields and methods of an object can be accessed from anywhere the object itself can be seen. Anyone can touch an object's public members. They should be kept to a minimum. Public fields should relate very closely to the core functionality of the class. They should not show intimate details of the inner workings of the class. Except in very simple instances fields should probably not be public.

The private fields and methods of an object can only be accessed by the object itself and by other objects of the same class (siblings). An object may touch its sibling's private parts. A sibling is an object in the same class but which is not the same object.

The Three Benefits of Access Protection

Access protection has three main benefits:

1. It allows you to enforce constraints on an object's state.
2. It provides a simpler client interface. Client programmers don't need to know everything that's in the class, only the public parts.

3. It separates interface from implementation, allowing them to vary independently. For instance consider making the `licensePlate` field of `Car` an instance of a new `LicensePlate` class instead of a `String`.

Changing the Implementation

Suppose the `Car` class needs to be used in a simulation of New York City traffic in which each actual car on the street is represented by one `Car` object. That's a lot of cars. As currently written each car object occupies approximately 60 bytes of memory (depending mostly on the size of the license plate string. We can knock off eight bytes per car by using floats instead of doubles, but the interface can stay the same:

```
public class Car {

  private String licensePlate; // e.g. "New York A456 324"
  private float speed;        // kilometers per hour
  private float maxSpeed;     // kilometers per hour

  public Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed = 0.0F;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = (float) maxSpeed;
    }
    else {
      maxSpeed = 0.0F;
    }

  }

  // getter (accessor) methods
  public String getLicensePlate() {
    return this.licensePlate;
  }

  public double getSpeed() {
    return this.speed;
  }

  public double getMaxSpeed() {
    return this.maxSpeed;
  }

  // setter method for the license plate property
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  public void floorIt() {
```

```
    this.speed = this.maxSpeed;
  }

  public void accelerate(double deltaV) {

     this.speed = this.speed + (float) deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0F;
     }

  }

}
```

Since the interface is the same, no other classes that depend on this class need to change or even be recompiled. We might save even more by using a custom `LicensePlate` class that only allowed one-byte ASCII characters instead of two byte Unicode characters.

What should be public? What should be private?

As a rule of thumb:

- Classes are public.
- Fields are private.
- Constructors are public.
- Getter and setter methods are public.
- Other methods must be decided on a case-by-case basis.

All of these rules may be freely violated if you have a reason for doing so. These are simply the defaults that handle 90% of the cases.

Further Examples

Money

What is money?

What is the essential information a program needs to know about money? How can these be represented as fields in a `Money` class?

Is this the only way to represent Money?

What should the constructor for `Money` look like?

What should the `toString()` method for Money look like?

Do you need any getter or setter methods? (Can money change its state or should it be immutable?)

What do you want to do with money? What methods should you write?

Angles

Complex Numbers

What is a complex number?

What are the essential parts of a complex number? How can these be represented as fields in a `ComplexNumber` class?

Is this the only way to represent a complex number?

What should the constructor for a `ComplexNumber` look like?

What should the `toString()` method for a ComplexNumber look like?

Do you need any `get()` or `set()` methods? (Can a complex number change its state or should it be immutable?)

What do you want to do with a complex number? What methods should you write?

Exercises

1. Implement the `Money` class discussed in class. This class should represent a dollar and cents amount with 0-99 cents and the cents being the same sign as the dollars. The class should at a minimum have getter methods that return the dollars and cents, a `toString()` method, all reasonable constructors, addition and subtraction methods, and a `main()` method that provides a thorough test of all the methods in the class.

   This should not be a hard problem. 90% of it was done for you in the third class.

2. Implement the Angle class discussed in class. This class should represent a mathematical angle with a guaranteed value between 0 and 360 degrees; that is, 0 <= degrees < 360. (Note that 0 and 360 are not symmetric. 0 is valid value while 360 is not.) The class should at a minimum have getter methods that return the radians and the degrees, sine, cosine, tangent, secant, cotangent, and cosecant methods, a `toString()` method, all reasonable constructors, and a `main()` method that provides a thorough test of all the methods in the class.
3. Implement the complex number class discussed in the lecture. At a minimum it should have a constructor, a `toString()` method, and methods to add, subtract, and multiply two complex numbers, and to return the real and imaginary parts.

   You may wish to attempt to implement division, absolute value, and argument methods as well. If so you will need to look ahead a little to learn about `java.lang.Math`. In particular, you'll need the trigonometric and square root functions.

Overloading

Overloading is when the same method or operator can be used on many different types of data. For instance the + sign is used to add ints as well as concatenate strings. The plus sign behaves differently depending on the type of its arguments. Therefore the plus sign is inherently overloaded.

Methods can be overloaded as well. `System.out.println()` can print a `double`, a `float`, an `int`, a `long`, or a `String`. You don't do anything different depending on the type of number you want the value of. Overloading takes care of it.

Programmer-defined classes can overload methods as well. To do this simply write two methods with the same name but different argument lists. For instance last week you saw several different versions of the `Car` constructor, one that took three arguments and one that took two arguments, and one that took no arguments. You can use all of these in a single class, though here I only use two because there really aren't any good default values for `licensePlate` and `maxSpeed`. On the other hand, 0 is a perfectly reasonable default value for `speed`.

```
public class Car {

  private String licensePlate; // e.g. "New York A456 324"
  private double speed;        // kilometers per hour
  private double maxSpeed;     // kilometers per hour

  // constructors
  public Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed  = 0.0;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
```

```
    else {
      maxSpeed = 0.0;
    }

  }

  public Car(String licensePlate, double speed, double maxSpeed) {

    this.licensePlate = licensePlate;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
    }

    if (speed < 0.0) {
      speed = 0.0;
    }

    if (speed <= maxSpeed) {
      this.speed = speed;
    }
    else {
      this.speed = maxSpeed;
    }

  }

  // other methods...

}
```

Normally a single identifier refers to exactly one method or constructor. When as above, one identifier refers to more than one method or constructor, the method is said to be *overloaded*. You could argue that this should be called identifier overloading rather than method overloading since it's the identifier that refers to more than one method, not the method that refers to more than one identifier. However in common usage this is called method overloading.

Which method an identifier refers to depends on the signature. The signature is the number, type, and order of the arguments passed to a method. The signature of the first constructor in the above program is `Car(String, double)`. The signature of the second method is `Car(String, double, double)`. Thus the first version of the `Car()` constructor is called when there is one String argument followed by one double argument and the second version is used when there is one String argument followed by two double arguments.

If there are no arguments to the constructor, or two or three arguments that aren't the right type in the right order, then the compiler generates an error because it doesn't have a method whose signature matches the requested method call. For example

```
Error:    Method Car(double) not found in class Car.
Car.java  line 17
```

`this` in constructors

It is often the case that overloaded methods are essentially the same except that one supplies default values for some of the arguments. In this case, your code will be easier to read and maintain (though perhaps *marginally* slower) if you put all your logic in the method that takes the most arguments, and simply invoke that method from all its overloaded variants that merely fill in appropriate default values.

This technique should also be used when one method needs to convert from one type to another. For instance one variant can convert a `String` to an `int`, then invoke the variant that takes the int as an argument.

This straight-forward for regular methods, but doesn't quite work for constructors because you can't simply write a method like this:

```
  public Car(String licensePlate, double maxSpeed) {

    Car(licensePlate, 0.0, maxSpeed);

  }
```

Instead, to invoke another constructor in the same class from a constructor you use the keyword `this` like so:

```
  public Car(String licensePlate, double maxSpeed) {

    this(licensePlate, 0.0, maxSpeed);

  }
```

Must this be the first line of the constructor?

For example,

```
public class Car {

  private String licensePlate; // e.g. "New York A456 324"
  private double speed;        // kilometers per hour
  private double maxSpeed;     // kilometers per hour

  // constructors
  public Car(String licensePlate, double maxSpeed) {

    this(licensePlate, 0.0, maxSpeed);

  }

  public Car(String licensePlate, double speed, double maxSpeed) {

    this.licensePlate = licensePlate;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
```

```
    }

    if (speed < 0.0) {
      speed = 0.0;
    }

    if (speed <= maxSpeed) {
      this.speed = speed;
    }
    else {
      this.speed = maxSpeed;
    }

  }

  // other methods...

}
```

This approach saves several lines of code. In also means that if you later need to change the constraints or other aspects of construction of cars, you only need to modify one method rather than two. This is not only easier; it gives bugs fewer opportunities to be introduced either through inconsistent modification of multiple methods or by changing one method but not others.

Operator Overloading

Some object oriented languages, notably C++, allow you to not only overload methods but also operators like + or -. This is very useful when dealing with user defined mathematical classes like complex numbers where + and - have well-defined meanings.

However most non-mathematical classes do not have obvious meanings for operators like + and - . Experience has shown that operator overloading is a large contributor to making multi-person programming projects infeasible. Therefore Java does not support operator overloading.

Inheritance

Code reusability is claimed to be a key advantage of object-oriented languages over non-object-oriented languages. Inheritance is the mechanism by which this is achieved. An object can inherit the variables and methods of another object. It can keep those it wants, and replace those it doesn't want.

For example, let us also expand the `Car` class so that a car also has a make, a model, a year, a number of passengers it can carry, four wheels, either two or four doors. That class might look like this:

```
public class Car {

  private String licensePlate;      // e.g. "New York A456 324"
  private double speed;             // kilometers per hour
  private double maxSpeed;          // kilometers per hour
  private String make;              // e.g. "Ford"
  private String model;             // e.g. "Taurus"
  private int    year;              // e.g. 1997, 1998, 1999, 2000, 2001, etc.
  private int    numberPassengers;  // e.g. 4
  private int    numberWheels = 4;  // all cars have four wheels
  private int    numberDoors;       // e.g. 4


  // constructors
  public Car(String licensePlate, double maxSpeed,
   String make, String model, int year, int numberOfPassengers,
   int numberOfDoors) {

    this(licensePlate, 0.0, maxSpeed, make, model, year,
     numberOfPassengers, numberOfDoors);

  }

  public Car(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {

    this(licensePlate, speed, maxSpeed, make, model, year,
     numberOfPassengers, 4);

  }

  public Car(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers,
   int numberOfDoors) {

    // I could add some more constraints like the
    // number of doors being positive but I won't
    // so that this example doesn't get too big.
    this.licensePlate = licensePlate;
    this.make = make;
    this.model = model;
    this.year = year;
    this.numberPassengers = numberOfPassengers;
    this.numberDoors = numberOfDoors;

    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
    }

    if (speed < 0.0) {
      speed = 0.0;
    }

    if (speed <= maxSpeed) {
      this.speed = speed;
```

```
  }
  else {
    this.speed = maxSpeed;
  }

}


// getter (accessor) methods
public String getLicensePlate() {
  return this.licensePlate;
}

public String getMake() {
  return this.make;
}

public String getModel() {
  return this.model;
}

public int getYear() {
  return this.year;
}

public int getNumberOfPassengers() {
  return this.numberPassengers;
}

public int getNumberOfWheels() {
  return this.numberWheels;
}

public int getNumberOfDoors() {
  return this.numberDoors;
}

public double getMaxSpeed() {
  return this.speed;
}

public double getSpeed() {
  return this.maxSpeed;
}

// setter method for the license plate property
public void setLicensePlate(String licensePlate) {
  this.licensePlate = licensePlate;
}

// accelerate to maximum speed
// put the pedal to the metal
public void floorIt() {
  this.speed = this.maxSpeed;
}

public void accelerate(double deltaV) {
```

```
    this.speed = this.speed + deltaV;
    if (this.speed > this.maxSpeed) {
      this.speed = this.maxSpeed;
    }
    if (this.speed <  0.0) {
      this.speed = 0.0;
    }

  }

}
```

Obviously this doesn't exhaust everything there is to say about a car. Which properties you choose to include in your class depends on your application.

Inheritance: the Superclass

In this example you begin by defining a more general `MotorVehicle` class.

```
public class MotorVehicle {

  protected String licensePlate;      // e.g. "New York A456 324"
  protected double speed;             // kilometers per hour
  protected double maxSpeed;          // kilometers per hour
  protected String make;              // e.g. "Harley-Davidson", "Ford"
  protected String model;             // e.g. "Fatboy", "Taurus"
  protected int    year;              // e.g. 1998, 1999, 2000, 2001, etc.
  protected int    numberPassengers;  // e.g. 4


  // constructors
  public MotorVehicle(String licensePlate, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {
    this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
  }

  public MotorVehicle(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {

    // I could add some more constraints like the
    // number of doors being positive but I won't
    // so that this example doesn't get too big.
    this.licensePlate = licensePlate;
    this.make = make;
    this.model = model;
    this.year = year;
    this.numberPassengers = numberOfPassengers;

    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
```

```
  }

  if (speed < 0.0) {
    speed = 0.0;
  }

  if (speed <= maxSpeed) {
    this.speed = speed;
  }
  else {
    this.speed = maxSpeed;
  }

}


// getter (accessor) methods
public String getLicensePlate() {
  return this.licensePlate;
}

public String getMake() {
  return this.make;
}

public String getModel() {
  return this.model;
}

public int getYear() {
  return this.year;
}

public int getNumberOfPassengers() {
  return this.numberPassengers;
}

public int getNumberOfPassengers() {
  return this.numberWheels;
}

public double getMaxSpeed() {
  return this.speed;
}

public double getSpeed() {
  return this.maxSpeed;
}

// setter method for the license plate property
protected void setLicensePlate(String licensePlate) {
  this.licensePlate = licensePlate;
}

// accelerate to maximum speed
// put the pedal to the metal
public void floorIt() {
  this.speed = this.maxSpeed;
```

```
  }

  public void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed < 0.0) {
       this.speed = 0.0;
     }

  }

}
```

The `MotorVehicle` class has all the characteristics shared by motorcycles and cars, but it leaves the number of wheels unspecified, and it doesn't have a `numberDoors` field since not all motor vehicles have doors. It also makes the fields and the `setLicensePlate()` method `protected` instead of `private` and `public`.

Inheritance: the Motorcycle subclass

The `MotorVehicle` class has all the characteristics shared by motorcycles and cars, but it leaves the number of wheels unspecified, and it doesn't have a `numberDoors` field since not all motor vehicles have doors.

Next you define two subclasses of `MotorVehicle`, one for cars and one for motorcycles. To do this you use the keyword `extends`.

```
public class Motorcycle extends MotorVehicle {

  protected int numberWheels = 2;


  // constructors
  public Motorcycle(String licensePlate, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {
    this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
  }

  public Motorcycle(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {

    // invoke superclass constructor
    super(licensePlate, speed, maxSpeed, make, model, year,
     numberOfPassengers);
  }

  public int getNumberOfWheels() {
    return this.numberWheels;
```

```
  }

}
```

Inheritance: The Car subclass

```
public class Car extends MotorVehicle {

  protected int numberWheels = 4;
  protected int numberDoors;


  // constructors
  public Car(String licensePlate, double maxSpeed,
   String make, String model, int year, int numberOfPassengers,
   int numberOfDoors) {
    this(licensePlate, maxSpeed, make, model, year, numberOfPassengers,
     numberOfDoors);
  }

  public Car(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {
    this(licensePlate, speed, maxSpeed, make, model, year,
     numberOfPassengers, 4);
  }

  public Car(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers,
   int numberOfDoors) {
    super(licensePlate, speed, maxSpeed, make, model,
     year, numberOfPassengers);
    this.numberDoors = numberOfDoors;
  }

  public int getNumberOfWheels() {
    return this.numberWheels;
  }

  public int getNumberOfDoors() {
    return this.numberDoors;
  }

}
```

It may look like these classes aren't as complete as the earlier ones, but that's incorrect. `Car` and `Motorcycle` each inherit the members of their superclass, `MotorVehicle`. Since a `MotorVehicle` has a make, a model, a year, a speed, a maximum speed, a number of passengers, cars and motorcycles also have makes, models, years, speeds, maximum speeds, and numbers of passengers. They alse have all the public methods the superclass has. They do not have the same constructors, though they can invoke the superclass constructor through the `super` keyword, much as a constructor in the same class can be invoked with the `this` keyword.

Subclasses and Polymorphism

Possibly move this later after Inheritance is finished

`Car` and `Motorcycle` are *subclasses* of `MotorVehicle`. If you instantiate a `Car` or a `Motorcycle` with `new`, you can use that object anywhere you can use a `MotorVehicle`, because cars are motor vehicles. Similarly you can use a `Motorcycle` anywhere you can use a `MotorVehicle`. This use of a subclass object in place of a superclass object is the beginning of *polymorphism*. I'll say more about polymorphism later.

The converse is not true. Although all cars are motor vehicles, not all motor vehicles are cars. Some are motorcycles. Therefore if a method expects a `Car` object you shouldn't give it a `MotorVehicle` object instead.

Note that I said you shouldn't give a method that expects a `Car` a `MotorVehicle`. I didn't say you couldn't. Objects can be cast into their subclasses. This is useful when using data structures like `Vector`s that only handle generic objects. It's up to the programmer to keep track of what kind of object is stored in a `Vector`, and to use it accordingly.

The proper choice of classes and subclasses is a skill learned primarily through experience. There are often different ways to define classes.

`toString()` as example of polymorphism

toString() Methods

Print methods are common in some languages, but most Java programs operate differently. You can use `System.out.println()` to print any object. However for good results your class should have a `toString()` method that formats the object's data in a sensible way and returns a string. Otherwise all that's printed is the name of the class which is normally not what you want. For example, a good `toString()` method for the `Car` class might be

```
 public String toString() {
    return  (this.licensePlate + " is moving at " + this.speed
     + "kph and has a maximum speed of " + this.maxSpeed +"kph.");
  }
```

Using `toString()` Methods

Below is a version of `CarTest` that uses `toString()` and `System.out.println()` instead of printing the fields directly and thus works with the new `Car` class that makes its fields private.

```
class CarTest5 {

  public static void main(String args[]) {

    Car c = new Car("New York A45 636", 123.45);
    System.out.println(c);

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c);
    }

  }

}
```

Rules for toString() Methods

`toString()` methods should return a single line of text that does not contain any carriage returns or linefeeds.

`toString()` methods are primarily for debugging.

`toString()` should not do a lot of fancy processing. `toString()` methods should be quick.

The string returned by `toString()` should contain the name of the class, and names and values of the fields that represent the state of the object, unless there are an excessive number of such fields, in which case only the most important should be returned.

A better `Car toString()` method would be:

```
 public String toString() {
    return "[Car: plate=" + this.licensePlate
     + " speed=" + this.speed +  + "MaxSpeed=" + this.maxSpeed +"]");
  }
```

These rules are conventions, not requirements of the language.

Multilevel Inheritance

The `Car-Motorcycle-MotorVehicle` example showed single-level inheritance. There's nothing to stop you from going further. You can define subclasses of cars for compacts, station wagons, sports coupes and more. For example, this class defines a compact as a car with two doors:

```
public class Compact extends Car {

  // constructors
  public Compact(String licensePlate, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {
    this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
  }

  public Compact(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers) {
    super(licensePlate, speed, maxSpeed, make, model,
     year, numberOfPassengers, 2);
  }

}
```

`Compact` not only inherits from its immediate superclass, `Car`, but also from `Car`'s superclass, `MotorVehicle`. Thus the `Compact` class also has a make, a model, a year and so on. There's no limit to this chain of inheritance, though getting more than four or five classes deep makes code excessively complex.

Multiple Inheritance

Some object oriented languages, notably C++, allow a class to inherit from more than one unrelated class. This is called multiple inheritance and is different from the multi-level inheritance in this section. Most of the things that can be accomplished via multiple inheritance in C++ can be handled by interfaces in Java.

Overriding Methods

Suppose that one day you've just finished your `Car` class. It's been plugged into your traffic simulation which is chugging along merrily simulating traffic. Then your pointy haired boss rolls in the door, and tells you that he needs the `Car` class to not accelerate past the 70 miles per hour (pointy haired bosses rarely understand the metric system) even if the car's a Ferrari with a maximum speed in excess of 200 miles per hour.

What are you going to do? Your first reaction may be to change the class that you already wrote so that it limits the speed of all the cars. However you're using that class elsewhere and things will break if you change it.

You could create a completely new class in a different file, either by starting from scratch or by copying and pasting. This would work, but it would mean that if you found a bug in the `Car` class now you'd have to fix it in two files. And if you wanted to add new methods to the `Car` class, you'd have to add them in two files. Still this is the best you could do if you were writing in C or some other traditional language.

Overriding Methods: The Solution

The object oriented solution to this problem is to define a new class, call it `SlowCar`, which inherits from `Car` and imposes the additional constraint that a car may not go faster than 70 mph (112.65 kph).

To do this you'll need to adjust the two places that speed can be changed, the constructor and the `accelerate()` method. The constructor has a different name because all constructors are named after their classes but the `accelerate()` method must be *overridden*. This means the subclass has a method with the same signature as the method in the superclass.

```
public class SlowCar extends Car {

  private static final double speedLimit = 112.65408; // kph == 70 mph

  public SlowCar(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers, int numDoors)
{

    super(licensePlate, speed, maxSpeed, make, model, year,
     numberOfPassengers, numDoors);
    if (speed > speedLimit) this.speed = speedLimit;

  }

  public void accelerate(double deltaV) {

     double speed = this.speed + deltaV;

     if (speed > this.maxSpeed) {
       speed = this.maxSpeed;
     }

     if (speed > speedLimit) {
       speed = speedLimit;
     }

     if (speed < 0.0) {
       speed = 0.0;
     }

     this.speed = speed;

  }

}
```

The first thing to note about this class is what it doesn't have, `getSpeed()`, `getLicensePlate()`, `getMaximumSpeed()`, `setLicensePlate()` methods or `speed`, `maxSpeed` and `numDoors` fields. All of these are provided by the superclass `Car`. Nothing about them has changed so they don't need to be repeated here.

Next look at the `accelerate()` method. This is different than the `accelerate()` method in `Car`. It imposes the additional constraint. Since the `speed` and `maxSpeed` fields from `Car` are protected, they're accessible from this subclass.

The constructor is a little more complicated. First note that if you're going to use a non-default constructor, that is a constructor with arguments, you do need to write a constructor for the subclass, even if it's just going to do the exact same thing as the matching constructor in the superclass. You cannot simply inherit `Car`'s constructor because that constructor is named `Car()` and this one must be named `SlowCar()`.

The constructor needs to set the value of name, url, and description. However they're not accessible from the subclass. Instead they are set by calling the superclass's constructor using the keyword `super`. When `super` is used as a method in the first non-blank line of a constructor, it stands for the constructor of this class's superclass.

The immediate superclass's constructor *will* be called in the first non-blank line of the subclass's constructor. If you don't call it explicitly, then Java will call it for you with no arguments. It's a compile time error if the immediate superclass doesn't have a constructor with no arguments and you don't call a different constructor in the first line of the subclass's constructor.

The use of the ternary operator in the constructor call is unusual. However, it's necessary to meet the compiler's requirement that the invocation of `super` be the first line in the subclass constructor. Otherwise this could be written more clearly using only `if-else`.

Adding Methods

A subclass isn't restricted to changing the behavior of its superclass. It can also add completely new methods and fields that are not shared with the superclass. For instance if a class represents a user of a multi-user database, the user class might start off with no access. Secretaries could be given read-only access. Therefore the secretary class would have additional methods to read data that aren't part of the user class.

Data entry clerks might be allowed to read the database and to create new records but not to modify old records. Therefore the data entry class would inherit from the secretary class and have methods to allow adding new records. A supervisor class would inherit from the data entry class and also have methods that allows modification of existing records. Finally a database manager would inherit from supervisor, but have new methods that allowed database managers to change the structure of the database.

As a practical example, recall that the `Car` class added a `numberDoors` field and a `getNumberOfDoors()` method that the `MotorVehicle` class didn't have.

Class or static Members

A method or a field in a Java program can be declared `static`. This means the member belongs to the class rather than to an individual object.

If a variable is static, then when any object in the class changes the value of the variable, that value changes for all objects in the class.

For example, suppose the `Car` class contained a `speedLimit` field that was set to 112 kph (70 mph). This would be the same for all cars. If it changed (by act of Congress) for one car, it would have to change for all cars. This is a typical static field.

Methods are often static is if they neither access nor modify any of the instance (non-static) fields of a class and they do not invoke any non-static methods in the class. This is common in calculation methods like a square root method that merely operate on their arguments and return a value. One way of thinking of it is that a method should be static if it neither uses nor needs to use `this`.

Class or static Members

Below is a `Car` class with such a `speedLimit` field and `getSpeedLimit()` method.

```
public class Car {

  private String licensePlate; // e.g. "New York A456 324"
  private double speed;        // kilometers per hour
  private double maxSpeed;     // kilometers per hour
  private static double speedLimit = 112.0;  // kilometers per hour

  public Car(String licensePlate, double maxSpeed) {

    this.licensePlate = licensePlate;
    this.speed  = 0.0;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = maxSpeed;
    }
    else {
      maxSpeed = 0.0;
    }

  }

  // getter (accessor) methods
  public static double getSpeedLimit() {
    return speedLimit;
  }

  public boolean isSpeeding() {
    return this.speed > speedLimit;
  }

  public String getLicensePlate() {
```

```
    return this.licensePlate;
  }

  public double getMaxSpeed() {
    return this.maxSpeed;
  }

  public double getSpeed() {
    return this.speed;
  }

  // setter method for the license plate property
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  public void floorIt() {
    this.speed = this.maxSpeed;
  }

  public void accelerate(double deltaV) {

     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed) {
       this.speed = this.maxSpeed;
     }
     if (this.speed <  0.0) {
       this.speed = 0.0;
     }

  }

}
```

Invoking static methods

If a method or field is declared `static`, you access it by using the class name rather than a reference to a particular instance of the class. Therefore instead of writing

```
Car c = new Car("New York", 89.7);
 double maximumLegalSpeed = c.getSpeedLimit();
```

you just write

```
double maximumLegalSpeed = Car.getSpeedLimit();
```

There does not even have to be an instance of a class in order to invoke a static method in the class.

Static methods may not call non-static methods or members of the same class directly. Rather they must specify which instance of the class they are referring to. Trying to call a non-static method or member is a very common compile time error. The specific error message generated by the javac compiler will look something like this

```
Error:    Can't make static reference to method void floorIt() in class Car.
```

Of course the names and signature will be changed to match the specific method and class.

`main()` methods for testing

The Java Class Library

Java contains an extensive library of pre-written classes you can use in your programs. These classes are divided into groups called *packages*.

The Java 1.1 packages

- `java.applet`
- `java.awt`
- `java.awt.datatransfer`
- `java.awt.event`
- `java.awt.image`
- `java.awt.peer`
- `java.beans`
- `java.io`
- `java.lang`
- `java.lang.reflect`
- `java.math`
- `java.net`
- `java.rmi`
- `java.rmi.dgc`
- `java.rmi.registry`
- `java.rmi.server`
- `java.security`
- `java.security.acl`
- `java.security.interfaces`
- `java.sql`
- `java.text`
- `java.util`
- `java.util.zip`

Each package defines a number of classes, interfaces, exceptions, and errors.

Packages can be split into sub-packages. for example, the `java.lang` package has a sub-package called `java.lang.reflect`. These are really completely different packages. A class in a sub-package has no more access to a class in the parent package (or vice versa) than it would to a class in a completely different package.

The java.net package

Each package defines a number of classes, interfaces, exceptions, and errors. For example, the java.net package contains these, interfaces, classes, and exceptions:

Interfaces in java.net

- `ContentHandlerFactory`
- `FileNameMap`
- `SocketImplFactory`
- `URLStreamHandlerFactory`

Classes in java.net

- `ContentHandler`
- `DatagramPacket`
- `DatagramSocket`
- `DatagramSocketImpl`
- `HttpURLConnection`
- `InetAddress`
- `MulticastSocket`
- `ServerSocket`
- `Socket`
- `SocketImpl`
- `URL`
- `URLConnection`
- `URLEncoder`
- `URLStreamHandler`

Exceptions in java.net

- `BindException`
- `ConnectException`
- `MalformedURLException`
- `NoRouteToHostException`
- `ProtocolException`
- `SocketException`
- `UnknownHostException`
- `UnknownServiceException`

Documentation for the class library

Javasoft provides a large amount of documentation for the classes, interface's and exceptions in the class library. If you've installed the JDK on your own PC, you'll find this in your JDK directory in docs/api/packages.html You can load that file into a web browser, and peruse the entire library.

If you have not installed the JDK, you can find it online at

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

Reading the documentation for a class in the class library

For example, let's suppose you want to use the `URL` class in the `java.net` package. By reading the documentation for the class you discover that it has the following public methods and constructors:

```
public URL(String protocol, String host, int port, String file)
 throws MalformedURLException
public URL(String protocol, String host, String file)
 throws MalformedURLException
public URL(String spec) throws MalformedURLException
public URL(URL context, String spec) throws MalformedURLException
public int getPort()
public String getFile()
public String getProtocol()
public String getHost()
public String getRef()
public boolean equals(Object obj)
public int hashCode()
public boolean sameFile(URL other)
public String toString()
public URLConnection openConnection() throws IOException
public final InputStream openStream() throws IOException
public static synchronized void
 setURLStreamHandlerFactory(URLStreamHandlerFactory factory)
```

You use this class just like you'd use any other class with these methods that happens to be named `java.net.URL`.

Using a class from the class library

You use the `java.net.URL` class just like you'd use any other class with these methods that happens to be named `java.net.URL`. For example,

```
public class URLSplitter {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        java.net.URL u = new java.net.URL(args[i]);
        System.out.println("Protocol: " + u.getProtocol());
        System.out.println("Host: " + u.getHost());
        System.out.println("Port: " + u.getPort());
```

```
        System.out.println("File: " + u.getFile());
        System.out.println("Ref: " + u.getRef());
      }
      catch (java.net.MalformedURLException e) {
        System.err.println(args[i] + " is not a valid URL");
      }
    }

  }

}
```

Here's the output:

```
% java SplitURL http://www.poly.edu
Protocol: http
Host: www.poly.edu
Port: -1
File: /
Ref: null
```

Importing Classes

Fully qualifed names like `java.net.URL` are not the most convenient thing to have to type. You can use the shorter class names like `URL` without the `java.net` part if you first import the class by adding an `import` statement at the top of the file. For example,

```
import java.net.URL;
import java.net.MalformedURLException;

public class URLSplitter {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        System.out.println("Protocol: " + u.getProtocol());
        System.out.println("Host: " + u.getHost());
        System.out.println("Port: " + u.getPort());
        System.out.println("File: " + u.getFile());
        System.out.println("Ref: " + u.getRef());
      }
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a valid URL");
      }
    }

  }
}
```

Package Imports

Instead of importing each class you need individually, you can import an entire package by replacing the class name with an asterisk (*) like this:

```
import java.net.*;

public class URLSplitter {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        System.out.println("Protocol: " + u.getProtocol());
        System.out.println("Host: " + u.getHost());
        System.out.println("Port: " + u.getPort());
        System.out.println("File: " + u.getFile());
        System.out.println("Ref: " + u.getRef());
      }
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a valid URL");
      }
    }

  }
}
```

This does not affect the final compiled code at all. However compilation will take a little longer. In general the time you save in not having to recover from error messages about missing classes more than makes up for the time you lose in compilation.

Name Conflicts when importing packages

It is possible that you will try to import a package that contains classes that have the same name as a class in your own source code. In general you should try to avoid this, especially when the conflict is with a class in the java packages.

The two classes may or may not mean the same thing. In general you should probably rename the class in your own source code rather than in the package. However if the two classes with conflicting names are related functionally, it may make sense to implement your class as a subclass of the class in the package.

There are a couple of name conflicts in the class library. The worst offenders are `java.util.List` and `java.awt.List`. The second worst offenders are `java.sql.Date` and `java.util.Date`, though this case is somewhat mitigated because `java.sql.Date` is a subclass of `java.util.Date`. If you need to use one or both of the conflictingly named classes as well as importing the other package, you simply have to use the full package qualified name, inconvenient as it may be to type.

You don't need to import java.lang.*

There is one exception to the import rule. All classes in the `java.lang` package are imported by default. Thus you do not need to `import java.lang.*;` to use them without fully qualifed names.

Consider the `System.out.println()` method we've been using since the first day of class.

`System` is really the `java.lang.System` class. This class has a `public static` field called `out` which is an instance of the `java.io.PrintStream` class. So when you write `System.out.println()`, you're really calling the `println()` method of the `out` field of the `java.lang.System` class.

The java.lang package

Each package defines a number of classes, interfaces, exceptions, and errors. For example, in Java 1.1 the `java.lang` package contains these:

Interfaces in java.lang

- `Cloneable`
- `Runnable`

Classes in java.lang

- `Boolean`
- `Byte`
- `Character`
- `Class`
- `ClassLoader`
- `Compiler`
- `Double`
- `Float`
- `Integer`
- `Long`
- `Math`
- `Number`
- `Object`
- `Process`
- `Runtime`
- `SecurityManager`

- Short
- String
- StringBuffer
- System
- Thread
- ThreadGroup
- Throwable
- Void

Exceptions in java.lang

- ArithmeticException
- ArrayIndexOutOfBoundsException
- ArrayStoreException
- ClassCastException
- ClassNotFoundException
- CloneNotSupportedException
- Exception
- IllegalAccessException
- IllegalArgumentException
- IllegalMonitorStateException
- IllegalStateException
- IllegalThreadStateException
- IndexOutOfBoundsException
- InstantiationException
- InterruptedException
- NegativeArraySizeException
- NoSuchFieldException
- NoSuchMethodException
- NullPointerException
- NumberFormatException
- RuntimeException
- SecurityException
- StringIndexOutOfBoundsException

Errors in java.lang

- AbstractMethodError
- ClassCircularityError
- ClassFormatError
- Error
- ExceptionInInitializerError
- IllegalAccessError
- IncompatibleClassChangeError
- InstantiationError
- InternalError
- LinkageError
- NoClassDefFoundError
- NoSuchFieldError

- `NoSuchMethodError`
- `OutOfMemoryError`
- `StackOverflowError`
- `ThreadDeath`
- `UnknownError`
- `UnsatisfiedLinkError`
- `VerifyError`
- `VirtualMachineError`

java.lang.Object

The `java.lang.Object` class is the ultimate superclass of all objects If a class does not explicitly extend a class, then the compiler assumes it extends `java.lang.Object`.

There is one exception. Guesses?

The Methods of java.lang.Object

`java.lang.Object` provides a number of methods that are common to all objects. `toString()` is the most common such method. Since the default `toString()` method only produces the name of the class, you should override it in all classes you define.

- `public Object()`
- `public final Class getClass()`
- `public int hashCode()`
- `public boolean equals(Object obj)`
- `protected Object clone() throws CloneNotSupportedException`
- `public String toString()`
- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait(long timeout) throws InterruptedException`
- `public final void wait(long timeout, int nanos) throws InterruptedException`
- `public final void wait() throws InterruptedException`
- `protected void finalize() throws Throwable`

toString() Methods

Print methods are common in some languages, but most Java programs operate differently. You can use `System.out.println()` to print any object. However for good results your class should have a `toString()` method that formats the object's data in a sensible way and returns a string.

Otherwise all that's printed is the name of the class which is normally not what you want. For example, a good `toString()` method for the `Car` class might be

```
 public String toString() {
    return  (this.licensePlate + " is moving at " + this.speed
     + "kph and has a maximum speed of " + this.maxSpeed +"kph.");
  }
```

Using `toString()` Methods

Below is a version of `CarTest` that uses `toString()` and `System.out.println()` instead of printing the fields directly and thus works with the new `Car` class that makes its fields private.

```
class CarTest5 {

  public static void main(String args[]) {

    Car c = new Car("New York A45 636", 123.45);
    System.out.println(c);

    for (int i = 0; i < 15; i++) {
      c.accelerate(10.0);
      System.out.println(c);
    }

  }

}
```

Rules for toString() Methods

`toString()` methods should return a single line of text that does not contain any carriage returns or linefeeds.

`toString()` methods are primarily for debugging.

`toString()` should not do a lot of fancy processing. `toString()` methods should be quick.

The string returned by `toString()` should contain the name of the class, and names and values of the fields that represent the state of the object, unless there are an excessive number of such fields, in which case only the most important should be returned.

A better `Car toString()` method would be:

```
 public String toString() {
    return "[Car: plate=" + this.licensePlate
```

```
    + " speed=" + this.speed +  + "MaxSpeed=" + this.maxSpeed +"]");
  }
```

These rules are conventions, not requirements of the language.

The equals() method

The `equals()` method of `java.lang.Object` acts the same as the == operator; that is, it tests for object identity rather than object equality. The implicit contract of the `equals()` method, however, is that it tests for equality rather than identity. Thus most classes will override `equals()` with a version that does field by field comparisons before deciding whether to return true or false.

To elaborate, an object created by a `clone()` method (that is a copy of the object) should pass the `equals()` test if neither the original nor the clone has changed since the clone was created. However the clone will fail to be == to the original object.

For example, here is an `equals()` method you could use for the `Car` class. Two cars are equal if and only if their license plates are equal, and that's what this method tests for.

```
public boolean equals(Object o) {

    if (o instanceof Car) {
      Car c = (Car) o;
      if (this.licensePlate.equals(c.licensePlate)) return true;
    }
    return false;

  }
```

This example is particularly interesting because it demonstrates the impossibility of writing a useful generic `equals()` method that tests equality for any object. It is not sufficient to simply test for equality of all the fields of two objects. It is entirely possible that some of the fields may not be relevant to the test for equality as in this example where changing the speed of a car does not change the actual car that's referred to.

Be careful to avoid this common mistake when writing `equals()` methods:

```
public boolean equals(Car c) {

    if (o instanceof Car) {
      Car c = (Car) o;
      if (this.licensePlate.equals(c.licensePlate)) return true;
    }
    return false;

  }
```

The `equals()` method must allow tests against any object of any class, not simply against other objects of the same class (`Car` in this example.)

You do not need to test whether `o` is `null`. `null` is never an instance of any class. `null instanceof Object` returns `false`.

The hashCode() method of java.lang.Object

Anytime you override `equals()` you should also override `hashCode()`. The `hashCode()` method should ideally return the same `int` for any two objects that compare equal and a different int for any two objects that don't compare equal, where equality is defined by the `equals()` method. This is used as an index by the `java.util.Hashtable` class.

In the `Car` example equality is determined exclusively by comparing license plates; therefore only the `licesePlate` field is used to determine the hash code. Since `licensePlate` is a `String`, and since the `String` class has its own `hashCode()` method, we can sponge off of that.

```
public int hashCode() {

   return this.licensePlate.hashCode();

 }
```

Other times you may need to use the bitwise operators to merge hash codes for multiple fields. There are also a variety of useful methods in the type wrapper classes (`java.lang.Double`, `java.lang.Float`, etc.) that convert primitive data types to integers that share the same bit string. These can be used to hash primitive data types.

java.lang.Math

The Java class library is huge. We will not cover it all today. In fact, the remaining eight classes will focus mostly on the class library. However, I do want to take this opportunity to look briefly at one useful class in the library, `java.lang.Math`. This is a class which contains static methods for performing many standard mathematical operations like square roots and cosines. You will need it for many of this weeks exercises.

The `Math` class contains several dozen static methods. Recall that to use a static method from a class, you just prefix its name with the name of the class followed by a period. For instance

```
double x = Math.sqrt(9.0);
```

Y ou never need to instantiate the `Math` class directly. (In fact you can't. The `Math()` constructor is declared private.)

Examples of java.lang.Math Methods

Here is an example program that exercises most of the routines in `java.lang.Math`. If your high school math is a little rusty, don't worry if you don't remember the exact meaning of logarithms or cosines. Just know that they're here in Java if you need them.

```
public class MathLibraryExample {

  public static void main(String args[]) {

    int i = 7;
    int j = -9;
    double x = 72.3;
    double y = 0.34;

    System.out.println("i is " + i);
    System.out.println("j is " + j);
    System.out.println("x is " + x);
    System.out.println("y is " + y);

    // The absolute value of a number is equal to
    // the number if the number is positive or
    // zero and equal to the negative of the number
    // if the number is negative.

    System.out.println("|" + i + "| is " + Math.abs(i));
    System.out.println("|" + j + "| is " + Math.abs(j));
    System.out.println("|" + x + "| is " + Math.abs(x));
    System.out.println("|" + y + "| is " + Math.abs(y));

    // Truncating and Rounding functions

    // You can round off a floating point number
    // to the nearest integer with round()
     System.out.println(x + " is approximately " + Math.round(x));
     System.out.println(y + " is approximately " + Math.round(y));

    // The "ceiling" of a number is the
    // smallest integer greater than or equal to
    // the number. Every integer is its own
    // ceiling.
     System.out.println("The ceiling of " + i + " is " + Math.ceil(i));
     System.out.println("The ceiling of " + j + " is " + Math.ceil(j));
     System.out.println("The ceiling of " + x + " is " + Math.ceil(x));
     System.out.println("The ceiling of " + y + " is " + Math.ceil(y));

     // The "floor" of a number is the largest
     // integer less than or equal to the number.
     // Every integer is its own floor.
     System.out.println("The floor of " + i + " is " + Math.floor(i));
     System.out.println("The floor of " + j + " is " + Math.floor(j));
```

```
 System.out.println("The floor of " + x + " is " + Math.floor(x));
 System.out.println("The floor of " + y + " is " + Math.floor(y));

 // Comparison operators

 // min() returns the smaller of the two arguments you pass it
 System.out.println("min(" + i + "," + j + ") is " + Math.min(i,j));
 System.out.println("min(" + x + "," + y + ") is " + Math.min(x,y));
 System.out.println("min(" + i + "," + x + ") is " + Math.min(i,x));
 System.out.println("min(" + y + "," + j + ") is " + Math.min(y,j));

 // There's a corresponding max() method
 // that returns the larger of two numbers
 System.out.println("max(" + i + "," + j + ") is " + Math.max(i,j));
 System.out.println("max(" + x + "," + y + ") is " + Math.max(x,y));
 System.out.println("max(" + i + "," + x + ") is " + Math.max(i,x));
 System.out.println("max(" + y + "," + j + ") is " + Math.max(y,j));

 // The Math library defines a couple
 // of useful constants:
 System.out.println("Pi is " + Math.PI);
 System.out.println("e is " + Math.E);
 // Trigonometric methods
// All arguments are given in radians

// Convert a 45 degree angle to radians
double angle = 45.0 * 2.0 * Math.PI/360.0;
System.out.println("cos(" + angle + ") is " + Math.cos(angle));
System.out.println("sin(" + angle + ") is " + Math.sin(angle));

 // Inverse Trigonometric methods
 // All values are returned as radians

double value = 0.707;

System.out.println("acos(" + value + ") is " + Math.acos(value));
System.out.println("asin(" + value + ") is " + Math.asin(value));
System.out.println("atan(" + value + ") is " + Math.atan(value));

// Exponential and Logarithmic Methods

// exp(a) returns e (2.71828...) raised
// to the power of a.
System.out.println("exp(1.0) is " + Math.exp(1.0));
System.out.println("exp(10.0) is " + Math.exp(10.0));
System.out.println("exp(0.0) is " +  Math.exp(0.0));

// log(a) returns  the natural
// logarithm (base e) of a.
System.out.println("log(1.0) is " + Math.log(1.0));
System.out.println("log(10.0) is " + Math.log(10.0));
System.out.println("log(Math.E) is " + Math.log(Math.E));

// pow(x, y) returns the x raised
// to the yth power.
System.out.println("pow(2.0, 2.0) is " + Math.pow(2.0,2.0));
System.out.println("pow(10.0, 3.5) is " + Math.pow(10.0,3.5));
System.out.println("pow(8, -1) is " + Math.pow(8,-1));
```

```
    // sqrt(x) returns the square root of x.
    for (i=0; i < 10; i++) {
      System.out.println(
       "The square root of " + i + " is " + Math.sqrt(i));
    }


    // Finally there's one Random method
    // that returns a pseudo-random number
    // between 0.0 and 1.0;

    System.out.println("Here's one random number: " + Math.random());
    System.out.println("Here's another random number: " + Math.random());

  }

}
```

java.lang.Math

Here's the output from the math library example

```
i is 7
j is -9
x is 72.3
y is 0.34
|7| is 7
|-9| is 9
|72.3| is 72.3
|0.34| is 0.34
72.3 is approximately 72
0.34 is approximately 0
The ceiling of 7 is 7
The ceiling of -9 is -9
The ceiling of 72.3 is 73
The ceiling of 0.34 is 1
The floor of 7 is 7
The floor of -9 is -9
The floor of 72.3 is 72
The floor of 0.34 is 0
min(7,-9) is -9
min(72.3,0.34) is 0.34
min(7,72.3) is 7
min(0.34,-9) is -9
max(7,-9) is 7
max(72.3,0.34) is 72.3
max(7,72.3) is 72.3
max(0.34,-9) is 0.34
Pi is 3.14159
e is 2.71828
cos(0.785398) is 0.707107
sin(0.785398) is 0.707107
acos(0.707) is 0.785549
asin(0.707) is 0.785247
```

```
atan(0.707) is 0.615409
exp(1.0) is 2.71828
exp(10.0) is 22026.5
exp(0.0) is 1
log(1.0) is 0
log(10.0) is 2.30259
log(Math.E) is 1
pow(2.0, 2.0) is 4
pow(10.0, 3.5) is 3162.28
pow(8, -1) is 0.125
The square root of 0 is 0
The square root of 1 is 1
The square root of 2 is 1.41421
The square root of 3 is 1.73205
The square root of 4 is 2
The square root of 5 is 2.23607
The square root of 6 is 2.44949
The square root of 7 is 2.64575
The square root of 8 is 2.82843
The square root of 9 is 3
Here's one random number: 0.820582
Here's another random number: 0.866157
```

java.util.Date

A `Date` object represents a precise moment in time, down to the millisecond. Dates are represented as a `long` that counts the number of milliseconds since midnight, January 1, 1970, Greenwich Meantime.

Does this have a year 2000 problem? If so in what year?

To create a `Date` object for the current date and time use the noargs `Date()` constructor like this:

```
Date now = new Date();
```

To create a `Date` object for a specific time, pass the number of milliseconds since midnight, January 1, 1970, Greenwich Meantime to the constructor, like this:

```
Date midnight_jan2_1970 = new Date(24L*60L*60L*1000L);
```

You can return the number of milliseconds in the `Date` as a `long`, using the `getTime()` method. For example, to time a block of code, you might do this

```
Date d1 = new Date();
// timed code goes here
Date d2 = new Date();
long elapsed_time = d2.getTime() - d1.getTime();
System.out.println("That took " + elapsed_time + " milliseconds");
```

You can change a `Date` by passing the new date as a number of milliseconds since midnight, January 1, 1970, GMT, to the `setTime()` method, like this:

```
Date midnight_jan2_1970 = new Date();
midnight_jan2_1970.setTime(24L*60L*60L*1000L);
```

The `before()` method returns true if this Date is before the Date argument, false if it's not. For example

```
if (midnight_jan2_1970.before(new Date())) {
```

The `after()` method returns true if this `Date` is after the Date argument, false if it's not. For example

```
if (midnight_jan2_1970.after(new Date())) {
```

The `Date` class also has the usual `hashCode()`, `equals()`, and `toString()` methods.

java.util.Calendar

The `Calendar` class converts a time in milliseconds since midnight, January 1, 1970, Greenwich Mean Time, (that is a `Date` object), into days, minutes, hours, and seconds according to the local calendar.

java.util.Random

The `java.util.Random` class allows you to create objects that produce pseudo-random numbers with uniform or gaussian distributions according to a linear congruential formula with a 48-bit seed.

The algorithm used is good enough for games. I wouldn't use it for cryptography.

You can choose the seed or you can let Java pick one based on the current time.

```
Random r = new Random(109876L);
int i = r.nextInt();
int j = r.nextInt();
long l = r.nextLong();
float f = r.nextFloat();
double d = r.nextDouble();
int k = r.nextGaussian();
```

The `nextInt()`, `nextLong()`, and `nextBytes()` methods all cover their respective ranges with equal likelihood. For example, to simulate a six-sided die; that is to generate a random integer between 1 and 6, you might write

```
Random r = new Random();
int die = r.nextInt();
die = Math.abs(die);
die = die % 6;
die += 1;
```

```
System.out.println(die);
```
The `nextGaussian()` method returns a pseudo-random, Gaussian distributed, double value with mean 0.0 and standard deviation 1.0.

The `nextBytes()` method fills a byte[] array with random bytes. For example,

```
byte[] ba = new byte[1024];
Random r = new Random();
r.nextBytes(ba);
for (int i = 0; i < ba.length; i++) {
  System.out.println(ba[i]);
}
```

java.lang.String

Strings are objects. Specifically they're instances of the class `java.lang.String`. This class has many methods that are useful for working with strings.

Internally Java Strings are arrays of Unicode characters. For example the String `"Hello"` is a five element array. Like arrays, Strings begin counting at 0. Thus in the String `"Hello"` `'H'` is character 0, `'e'` is character 1, and so on.

0	1	2	3	4
H	e	l	l	o

Constructors

```
public String()
public String(String value)
public String(char value[])
public String(char value[], int offset, int count)
public String(byte bytes[], int offset, int length, String enc)
 throws UnsupportedEncodingException
public String(byte bytes[], String enc) throws UnsupportedEncodingException
public String(byte bytes[], int offset, int length)
public String(byte bytes[])
public String(StringBuffer buffer)
```

index methods

```
public int length()
public int indexOf(int ch)
public int indexOf(int ch, int fromIndex)
public int lastIndexOf(int ch)
public int lastIndexOf(int ch, int fromIndex)
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
```

```
public int lastIndexOf(String str)
public int lastIndexOf(String str, int fromIndex)
```

valueOf() methods

```
public static String valueOf(char data[])
public static String valueOf(char data[], int offset, int count)
public static String copyValueOf(char data[], int offset, int count)
public static String copyValueOf(char data[])
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
public static String valueOf(double d)
```

substring() methods

```
public char charAt(int index)
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)
public byte[] getBytes(String enc) throws UnsupportedEncodingException
public byte[] getBytes()
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
public String concat(String str)
public char[] toCharArray()
```

Beginnings are inclusive. Ends are exclusive.

comparisons

```
public boolean equals(Object anObject)
public boolean equalsIgnoreCase(String anotherString)
public int compareTo(String anotherString)
public boolean regionMatches(int toffset, String other, int ooffset, int len)
public boolean regionMatches(boolean ignoreCase, int toffset,
 String other, int ooffset, int len)
public boolean startsWith(String prefix, int toffset)
public boolean startsWith(String prefix)
public boolean endsWith(String suffix)
```

Modifying Strings

```
public String replace(char oldChar, char newChar)
public String toLowerCase(Locale locale)
public String toLowerCase()
public String toUpperCase(Locale locale)
public String toUpperCase()
public String trim()
```

The `final` keyword

The `final` keyword is used in several different contexts as a modifier meaning that what it modifies cannot be changed in some sense.

final classes

You will notice that a number of the classes in Java library are declared `final`, e.g.
`public final class String`
This means this class will not be subclassed, and informs the compiler that it can perform certain optimizations it otherwise could not. It also provides some benefit in regard to security and thread safety.

The compiler will not let you subclass any class that is declared `final`. You probably won't want or need to declare your own classes `final` though.

final methods

You can also declare that methods are `final`. A method that is declared `final` cannot be overridden in a subclass. The syntax is simple, just put the keyword `final` after the access specifier and before the return type like this:
`public final String convertCurrency()`

final fields

You may also declare fields to be `final`. This is not the same thing as declaring a method or class to be `final`. When a field is declared `final`, it is a constant which will not and cannot change. It can be set once (for instance when the object is constructed, but it cannot be changed after that.) Attempts to change it will generate either a compile-time error or an exception (depending on how sneaky the attempt is).

Fields that are both `final`, `static`, and `public` are effectively named constants. For instance a physics program might define `Physics.c`, the speed of light as

```
public class Physics {

  public static final double c = 2.998E8;


}
```

In the `SlowCar` class, the `speedLimit` field is likely to be both `final` and `static` though it's `private`.

```
public class SlowCar extends Car {

  private final static double speedLimit = 112.65408; // kph == 70 mph

  public SlowCar(String licensePlate, double speed, double maxSpeed,
   String make, String model, int year, int numberOfPassengers, int numDoors)
{
    super(licensePlate,
     (speed < speedLimit) ? speed : speedLimit,
```

```
      maxSpeed, make, model, year, numberOfPassengers, numDoors);
  }

  public void accelerate(double deltaV) {

      double speed = this.speed + deltaV;

      if (speed > this.maxSpeed) {
        speed = this.maxSpeed;
      }

      if (speed > speedLimit) {
        speed = speedLimit;
      }

      if (speed < 0.0) {
        speed = 0.0;
      }

      this.speed = speed;

  }

}
```

final arguments

Finally, you can declare that method arguments are `final`. This means that the method will not directly change them. Since all arguments are passed by value, this isn't absolutely required, but it's occasionally helpful.

What can be declared `final` in the `Car` and `MotorVehicle` classes?

abstract

Java allows methods and classes to be declared `abstract`. An abstract method is not actually implemented in the class. It is merely declared there. The body of the method is then implemented in subclasses of that class. An abstract method must be part of an abstract class. You create abstract classes by adding the keyword `abstract` after the access specifier, e.g.
`public abstract class MotorVehicle`
Abstract classes cannot be instantiated. It is a compile-time error to try something like
`MotorVehicle m = new MotorVehicle();`
when `MotorVehicle` has been declared to be abstract. `MotorVehicle` is actually a pretty good example of the sort of class that might be abstract. You're unlikely to be interested in a generic motor vehicle. Rather you'll have trucks, motorcycles, cars, go-carts and other subclasses of `MotorVehicle`, but nothing that is only a `MotorVehicle`.

An abstract method provides a declaration but no implementation. In other words, it has no method body. Abstract methods can only exist inside abstract classes and interfaces. For example, the `MotorVehicle` class might have an abstract `fuel()` method:

```
public abstract void fuel();
```

`Car` would override/implement this method with a `fuel()` method that filled the gas tank with gasoline. `EighteenWheelerTruck` might override this method with a `fuel()` method that filled its gas tank with diesel. `ElectricCar` would override/implement this method with a `fuel()` method that plugged into the wall socket.

Interfaces

Interfaces are the next level of abstraction. An interface is like a class with nothing but `abstract` methods and `final`, `static` fields. All methods and fields of an interface must be public.

However, unlike a class, an interface can be added to a class that is already a subclass of another class. Furthermore an interface can apply to members of many different classes. For instance you can define an `Import` interface with the single method `calculateTariff()`.

```
public interface Import {

  public double calculateTariff();

}
```

You might want to use this interface on many different classes, cars among them but also for clothes, food, electronics and moore. It would be inconvenient to make all these objects derive from a single class. Furthermore, each different type of item is likely to have a different means of calculating the tariff. Therefore you define an `Import` interface and declare that each class implements `Import`.

The syntax is simple. `Import` is declared public so that it can be accessed from any class. It is also possible to declare that an interface is protected so that it can only be implemented by classes in a particular package. However this is very unusual. Almost all interfaces will be public. No interface may be private because the whole purpose of an Interface is to be inherited by other classes.

The `interface` keyword takes the place of the `class` keyword. Line 3 looks like a classic method definition. It's public (as it must be). It's abstract, also as it must be. And it returns a `double`. The method's name is `calculateTariff()` and it takes no arguments. The difference between thisa and a method in a class is that there is no method body. That remains to be created in each class that implements the interface.

You can declare many different methods in an interface. These methods may be overloaded. An interface can also have fields, but if so they must be `final` and `static` (in other words constants).

Implementing Interfaces

To actually use this interface you create a class that includes a `public double calculateTariff()` method and declare that the class implements `Import`. For instance here's one such class:

```
public class Car extends MotorVehicle implements Import {

  int numWheels = 4;

  public double calculateTariff() {
    return this.price * 0.1;
  }

}
```

One of the advantages of interfaces over classes is that a single class may implement more than one interface. For example, this `Car` class implements three interfaces: `Import`, `Serializable`, and `Cloneable`

```
import java.io.*;


public class Car extends MotorVehicle
 implements Import, Serializable, Cloneable {

  int numWheels = 4;

  public double calculateTariff() {
    return this.price * 0.1;
  }

}
```

`Serializable` and `Cloneable` are marker interfaces from the class library that only add a type to a class, but do not declare any additional methods.

Implementing the Cloneable Interface

The `java.lang.Object` class contains a `clone()` method that returns a bitwise copy of the current object.

```
protected native Object clone() throws CloneNotSupportedException
```

Not all objects are cloneable. It particular only instances of classes that implement the `Cloneable` interface can be cloned. Trying to clone an object that does not implement the `Cloneable` interface throws a `CloneNotSupportedException`.

For example, to make the `Car` class cloneable, you simply declare that it implements the `Cloneable` interface. Since this is only a *marker interface*, you do not need to add any methods to the class.

```
public class Car extends MotorVehicle implements Cloneable {

  // ...

}
```

For example

```
Car c1 = new Car("New York A12 345", 150.0);
Car c2 = c1.clone();
```

Most classes in the class library do not implement `Cloneable` so their instances are not cloneable.

Most of the time, clones are *shallow copies*. In other words if the object being cloned contains a reference to another object A, then the clone contains a reference to the same object A, not to a clone of A. If this isn't the behavior you want, you can override `clone()` yourself.

You may also override `clone()` if you want to make a subclass uncloneable, when one of its superclasses does implement `Cloneable`. In this case simply use a `clone()` method that throws a `CloneNotSupportedException`. For example,

```
 public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException("Can't clone a SlowCar");
    // never get here
    return this;
  }
```

You may also want to override `clone()` to make it `public` instead of `protected`. In this case, you can simply fall back on the superclass implementation. For example,

```
 public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
```

Wrapping Your Own Packages

Java does not limit you to using only the system supplied packages. You can write your own as well. You write packages just like you write any other Java program. Make sure you follow these rules:

1. There must be no more than one public class per file.
2. All files in the package must be named classname.java where classname is the name of the single public class in the file.
3. At the very top of each file in the package, before any import statements, put the statement

```
package myPackage;
```

To use the package in other programs, compile the .java files as usual and then move the resulting .class files into the appropriate subdirectory of one of the directories referenced in your CLASSPATH environment variable. For instance if /home/elharo/classes is in your CLASSPATH and your package is called `package1`, then you would make a directory called `package1` in /home/elharo/classes and then put all the .class files in the package in /home/elharo/classes/package1.

For example,

```
package com.macfaq.net;

import java.net.*;

public class URLSplitter {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        System.out.println("Protocol: " + u.getProtocol());
        System.out.println("Host: " + u.getHost());
        System.out.println("Port: " + u.getPort());
        System.out.println("File: " + u.getFile());
        System.out.println("Ref: " + u.getRef());
      }
      catch (MalformedURLException e) {
        System.err.println(args[i] + " is not a valid URL");
      }
    }

  }

}
% javac -d /home/elharo/classes URLSplitter.java
```

The -d flag to the compiler tells it to create the necessary directories such as elharo/net in the specified directory. In this example, URLSplitter.class is placed in /home/elharo/classes/com/macfaq/net. You can use the usual shell syntax such as . for the current directory or ~ for your home directory.

Naming Packages

As you saw earlier name space conflicts arise if two pieces of code declare the same name. Java keeps track of what belongs to each package internally. Thus if Sun decides to put a `Website` class in java.lang in Java 2.0, it won't conflict with a `Website` class defined in an `http` package. Since they're in different packages Java can tell them apart. Just as you tell John Smith apart from John Jones by their last names, so too does Java tell two Website classes apart by their package names.

However this scheme breaks down if two different classes share the same package name as well as class name. In a world in which many different packages can be downloaded from many different sites, all landing in the user's CLASSPATH, it's not unthinkable that two different people might write packages called `http` with a class called `Website`.

To ensure that package names don't conflict with each other, Sun asks that you prefix all your packages with a reversed domain name. Thus if your domain name is poly.edu, your package names should begin with `edu.poly`. Within that domain you are responsible for making sure that no one else writes a package that conflicts with yours.

If you don't have a personal domain name, only an account with an Internet service provider, then add your username to this as well. For instance under this scheme the package prefix for foo@utopia.poly.edu would be `edu.poly.utopia.foo`.

This is primarily of interest to applet writers, not applet users. If you're surfing the net and you load one applet from MIT that has a `http.Website` class and another from Polytechnic that has a different `http.Website` class, Java can still tell them apart because before it runs any package it downloads off the net it prefixes everything with the site from which it got it. In other words it sees `edu.poly.www.http.Website` and `edu.mit.www.http.Website`. It's only when you download a package manually and install it in one directory in your CLASSPATH and install another package elsewhere in your CLASSPATH that real name conflicts can arise if packages aren't carefully prefixed with a domain or email address.

JAR archives

JAR archives are ZIP archives with a different extension. They contain a hierarchy of files and directories. In essence a JAR file can take the place of a directory containing its contents. This has many uses including the distribution of related classes as a single file.

The JDK includes a `jar` program modeled after the Unix `tar` program that will bundle up files and directories into a JAR archive. For example, suppose your *homework* directory contains the file *Trivia.class*.

Now supose that Trivia is in the package `edu.poly.utopia.eharold.games`. Also suppose that the directory */home/users/eharold/homework/* contains (in various sub-directories) all the files and directories necessary to run the program `edu.poly.utopia.eharold.games.Trivia`. You can pack up everyhting in the edu package into a JAR archive like this:

```
% cd /home/users/eharold/homework
% jar cvf eharold.jar edu
```

The *edu* directory and all its contents are recursively stored in the archive named eharold.jar. The name of the archive is unimportant, only its contents.

This archive may now be added to the class path like this, or in whatever fashion you normally add directories to the class path on your platform of choice:

```
% setenv CLASSPATH $CLASSPATH:eharold.jar
```

The main thing to note here is that the JAR file is now taking the place of an entire directory hierarchy.

In Java 1.2 you can also place it in the *ext* directory of your *jre/lib* directory.

Either way, all the classes in the archive will be added to the class path and will be accessible to your programs.

Runnable JAR archives

You can run a program stored in the JAR archive that has a `main()` method like this:

```
% java -cp eharold.jar MainClassName
```

You must use the fully package qualified name. For example,

```
% java -cp eharold.jar edu.poly.utopia.eharold.games.Trivia
```

The `-cp` flag adds the jar file to the class path.

In Java 1.2 you can add a `Main-Class` attribute to a JAR file's manifest so that the person who runs the program does not need to know the name of the name of the class with the `main()` method. This attribute has the following form

```
Main-Class: edu.poly.utopia.eharold.games.Trivia
```

Put this line into a file called (for example) *MyManifest.txt*. Then use this command line to package the JAR:

```
% jar cvmf MyManifest.txt eharold.jar edu
```

This will copy data from the file *MyManifest.txt* into the JAR's own manifest file, and add the directory *edu* to the archive.

To run the program packaged in the jar file *eharold.jar* you simply type:

```
% java -jar eharold.jar
```

Java will look inside the JAR archive's manifest to find which class's `main()` method it should run.

Inner Classes

In Java 1.1 and later, you can define an *inner class*. This is a class whose body is defined inside another class, referred to as the *top-level class*. For example

```
public class Queue {

  Element back = null;

  public void add(Object o) {

    Element e = new Element();
    e.data = o;
    e.next = back;
    back = e;

  }

  public Object remove() {

    if (back == null) return null;
    Element e = back;
    while (e.next != null) e = e.next;
    Object o = e.data;
    Element f = back;
    while (f.next != e) f = f.next;
    f.next = null;
    return o;

  }

  public boolean isEmpty() {
    return back == null;
  }

  // Here's the inner class
  class Element {

    Object data = null;
    Element next = null;

  }

 }
```

Inner classes may also contain methods. They may not contain static members.

Inner classes in a class scope can be public, private, protected, final, abstract.

Inner classes can also be used inside methods, loops, and other blocks of code surrounded by braces (`{}`). Such a class is not a member, and therefore cannot be declared public, private, protected, or static.

The inner class has access to all the methods and fields of the top-level class, even the private ones.

The inner class's short name may not be used outside its scope. If you absolutely must use it, you can use the fully qualified name instead. (For example, `Queue$Element`) However, if you need to do this you should almost certainly have made it a top-level class instead or at least an inner class within a broader scope.

Inner classes are most useful for the adapter classes required by 1.1 AWT and JavaBeans event handling protocols. They allow you to implement callbacks. In most other languages this would be done with function pointers.

Exceptions

- What is an exception?
- try-catch
- finally
- The different kinds of exceptions
- Multiple catch clauses
- The throws clause
- Throwing exceptions
- Writing your own exception classes

What is an Exception?

Why use exceptions instead of return values?

1. Forces error checking
2. Cleans up your code by separating the normal case from the exceptional case. (The code isn't littered with a lot of `if-else` blocks checking return values.)
3. Low overhead for non-exceptional case

Traditional programming languages set flags or return bad values like -1 to indicate problems. Programmers often don't check these values.

Java throws `Exception` objects to indicate a problem. These cannot be ignored.

What is an Exception?

Consider this program:

```
public class HelloThere {

  public static void main(String[] args) {

    System.out.println("Hello " + args[0]);

  }

}
```

Suppose it's run like this:

```
% java HelloThere
```

Notice that's there's no `args[0]`. Here's what you get:

```
% java HelloThere
java.lang.ArrayIndexOutOfBoundsException: 0
        at HelloThere.main(HelloThere.java:5)
```

This is **not** a crash. The virtual machine exits normally. All memory is cleaned up. All resources are released.

try-catch

```
try-catch
public class HelloThere {

  public static void main(String[] args) {

    try {
      System.out.println("Hello " + args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Hello Whoever you are.");
    }

  }

}
```

What can you do with an exception once you've caught it?

1. Fix the problem and try again.
2. Do something else instead.
3. Exit the application with `System.exit()`
4. Rethrow the exception.
5. Throw a new exception.
6. Return a default value (in a non-void method).
7. Eat the exception and return from the method (in a void method).
8. Eat the exception and continue in the same method (Rare and dangerous. Be very careful if you do this. Novices almost always do this for the wrong reasons. Do **not** simply to avoid dealing with the exception. Generally you should only do this if you can logically guarantee that the exception will never be thrown or if the statements inside the `try` block do not need to be executed correctly in order for the following code to run.)

Printing an error message by itself is generally **not** an acceptable response to an exception.

The `finally` keyword

```
public class HelloThere {

  public static void main(String[] args) {

    try {
      System.out.println("Hello " + args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Hello Whoever you are.");
    }
    finally {
      System.out.println("How are you?");
    }


  }

}
```

The different kinds of exceptions

Checked Exceptions
: Environmental error that cannot necessarily be detected by testing; e.g. disk full, broken socket, database unavailable, etc.

Errors
: Virtual machine error: class not found, out of memory, no such method, illegal access to private field, etc.

Runtime Exceptions

Programming errors that should be detected in testing: index out of bounds, null pointer, illegal argument, etc.

Checked exceptions must be handled at compile time. Runtime exceptions do not need to be. Errors often cannot be.

The `Throwable` class hierarchy

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
           |
           +----java.lang.Exception
                   |
                   +----java.io.IOException
                   |
                   +----java.lang.RuntimeException
                           |
                           +----java.lang.ArithmeticException
                           |
                           +----java.lang.ArrayIndexOutOfBoundsException
                           |
                           +----java.lang.IllegalArgumentException
                           |
                           +----java.lang.NumberFormatException
```

Almost all the code is in the `java.lang.Throwable` class. Almost all of its subclasses only provide new constructors that change the message of the exception.

Catching multiple exceptions

```
public class HelloThere {

  public static void main(String[] args) {

    int repeat;

    try {
      repeat = Integer.parseInt(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      // pick a default value
      repeat = 1;
    }
    catch (NumberFormatException e) {
      // print an error message
      System.err.println("Usage: java HelloThere repeat_count" );
      System.err.println(
       "where repeat_count is the number of times to say Hello" );
```

```
      System.err.println("and given as an integer like 1 or 7" );
      return;
    }

    for (int i = 0; i < repeat; i++) {
      System.out.println("Hello");
    }

  }

}
```

Catching multiple exceptions

If multiple blocks match the exception type, the first block that matches the type of the exception catches it.

```
public class HelloThere {

  public static void main(String[] args) {

    int repeat;

    try {
      // possible NumberFormatException and ArrayIndexOutOfBoundsException
      repeat = Integer.parseInt(args[0]);

      // possible ArithmeticException
      int n = 2/repeat;

      // possible StringIndexOutOfBoundsException
      String s = args[0].substring(5);
    }
    catch (NumberFormatException e) {
      // print an error message
      System.err.println("Usage: java HelloThere repeat_count" );
      System.err.println(
       "where repeat_count is the number of times to say Hello" );
      System.err.println("and given as an integer like 1 or 7" );
      return;
    }
    catch (ArrayIndexOutOfBoundsException e) {
      // pick a default value
      repeat = 1;
    }
    catch (IndexOutOfBoundsException e) {
      // ignore it
    }
    catch (Exception e) {
      // print an error message and exit
      System.err.println("Unexpected exception");
      e.printStackTrace();
      return;
    }
```

```
    for (int i = 0; i < repeat; i++) {
      System.out.println("Hello");
    }

  }

}
```

It's rare to catch a generic `Error` or `Throwable` because it's really hard to clean up after them in the general case.

The `throws` keyword

Rather than explicitly catching an exception you can declare that your method throws the exception. This passes the repsonsibility to handle it to the method that invokes your method. This is done with the `throws` keyword. For example,

```
 public static void copy(InputStream in, OutputStream out)
   throws IOException {

    byte[] buffer = new byte[256];
    while (true) {
      int bytesRead = in.read(buffer);
      if (bytesRead == -1) break;
      out.write(buffer, 0, bytesRead);
    }

  }
```

A single method may have the potential to throw more than one type of exception. In this case the exception clases are just separated by commas. For example,

```
public BigDecimal divide(BigDecimal val, int roundingMode) throws
ArithmeticException, IllegalArgumentException
```

You can declare that your method throws runtime exceptions though you do not have to. The main use of this is as documentation for the programmer. It can also be useful in white box testing.

Throwing Exceptions

The `throw` keyword

```
public class Clock {

  int hours;   // 1-12
  int minutes; // 0-59
  int seconds; // 0-59

  public Clock(int hours, int minutes, int seconds) {
```

```
    if (hours < 1 || hours > 12) {
      throw new IllegalArgumentException("Hours must be between 1 and 12");
    }
    if (minutes < 0 || minutes > 59) {
      throw new IllegalArgumentException("Minutes must be between 0 and 59");
    }
    if (seconds < 0 || seconds > 59) {
      throw new IllegalArgumentException("Seconds must be between 0 and 59");
    }

    this.hours   = hours;
    this.minutes = minutes;
    this.seconds = seconds;

  }

  public Clock(int hours, minutes) {
    this(hours, minutes, 0);
  }

  public Clock(int hours) {
    this(hours, 0\, 0);
  }


}
```

Writing Exception Subclasses

Most exception subclasses inherit all their functionality from the superclass. Each subclass mainly serves as a marker for a different kind of exception. However it only rarely provides new methods or fields. Thus most of the time the only methods you need to implement are the constructors. There should be one noargs constructor and one constructor that takes a `String` message as an argument. These will mostly just invoke the matching superclass constructor.

```
public class ClockException extends Exception {

  public ClockException(String message) {
    super(message);
  }

  public ClockException() {
    super();
  }

}
```

Exception Methods

Mostly exceptions just serve as signals. They tend not to have a lot of methods of their own, and those they have are rarely invoked directly. The two most commonly used are `toString()` and `printStackTrace()`.

```
public String getMessage()
public String getLocalizedMessage()
public String toString()
public void printStackTrace()
public void printStackTrace(PrintStream s)
public void printStackTrace(PrintWriter s)
public native Throwable fillInStackTrace()
```

All of these are inherited from `java.lang.Throwable` as are pretty much all other methods in most exception classes.

Exercises

1. Although mathematicians prefer to work in radians, most scientists and engineers find it easier to think in degrees. Write sine, cosine and tangent methods that accept their arguments in degrees.
2. Write the corresponding set of inverse trigonometric methods that return their values in degrees instead of radians.
3. The math library is missing secant, cosecant and cotangent methods. Write them.
4. The math library lacks a log10 method for taking the common logarithm. Write one.
5. Computer scientists often use a log2 (log base 2). java.lang.Math doesn't have one of those either. Write it.
6. Put all the methods in the previous five exercises into a package and class of your own creation. Be sure to choose sensible, easy-to-understand, hard-to-confuse, names for all packages, classes, and methods. Declare methods and fields static, final, and/or abstract when appropriate.
7. A simple model for the growth of bacteria with an unlimited supply of nutrients says that after t hours an initial population of p0 will have grown to p0 * e to the 1.4t. Write a Java application that calculates the growth of a colony of bacteria. As usual get the value of p0 and t from the command line.
8. Modify the bacteria growth program so that the time can be input in minutes. Note that the formula still requires a time in hours.
9. Complete the `ComplexNumber` class discussed in last week's class.
10. Define a reasonably named package for financial classes. Place last week's `Money` class in this package.
11. Add an overloaded constructor to the `Money` class that only takes the number of dollars.
12. Add an overloaded constructor to the `Money` class that takes no arguments and initializes the object to $0.00.
13. Add an `equals()` method to the `Money` class.
14. Define an exception class called `MoneyOverflowException` which can be thrown when an operation with `Money` results in an over flow. Place this class in the same finance package.

15. Rewrite the methods in the `Money` class so that they recognize overflow and throw a `MoneyOverFlowException` if it occurs.
16. Use the classes in the `java.math` package to eliminate the possibility of overflow in the `Money` class.
17. Rewrite the two logistic equation problems from Week 2 using the `java.math.BigDecimal` class to provide 20 decimal digits of precision. Some hints:
    - Invocation of the `setScale()` method with every iteration is necessary to keep the value of population from overflawing the memory of the computer.
    - You may need to use `compareTo()` and the `Comparable` interface instead of either `==` or `equals()`.

HTML in 10 minutes

HTML is the HyperText Markup Language.

HTML files are text files featuring semantically tagged elements.

HTML filenames are suffixed with .htm or .html.

Here's a simple HTML file:

```
<html>
<head>
<title>My First HTML Document</title>
</head>
<body>
<h1>A level one heading</h1>

Hello there. This is <STRONG>very</STRONG> important.

</body>
</html>
```

Look at this file in your web browser.

Elements that are enclosed in angle brackets like `<html>`, `<head>`, and `<title>` are called *tags*. Tags are case insensitive. `<html>` means the same thing as `<HTML>` as `<Html>` `<HtMl>`.

Most tags are matched with closing tags, and affect the text contained between them. The closing tag is the same as the opening tag except for a / after the opening angle bracket. For example, `</html>`, `</head>`, and `</title>` are closing tags. The text in between `<title>` and `</title>`, *My First HTML Document* in the above example, is the title of the page.

As you can see from the above example tags may, in general, nest. However they may not overlap (though some browsers can handle this).

Some tags have *attributes*. An attribute is a name, followed by an = sign, followed by the value. For example, to make a centered `H1` heading, use the `ALIGN` attribute with value `center`; i.e.

```
<h1 align="center">A level one heading</h1>
```

Attributes are also case-insensitive. The double quotes around the value are optional unless the value contains embedded white space.

For more information about HTML see Larry Aronson and Joseph Lowry's *The HTML 3.2 Manual of Style* (Ziff-Davis Press, 1997) or the NCSA Beginner's Guide to HTML.

URLs in 10 minutes

URL stands for uniform resource locator. A URL is a pointer to a particular resource on the Internet at a particular location. For example `http://metalab.unc.edu/javafaq/course/week5/exercises.html` and `ftp://ftp.macfaq.com/pub/macfaq/` are both URLs.

A URL specifies the protocol used to access a server (e.g., ftp, http), the name of the server, and the location of a file on that server. A typical URL looks like `http://metalab.unc.edu/javafaq/books.html`. This specifies that there is a file called books.html in a directory called javafaq on the server metalab.unc.edu, and that this file can be accessed via the http protocol. The full syntax is:

protocol://hostname[:port]/path/filename#section

The protocol, also sometimes called the scheme, is generally one of these

file
: a file on your local disk

ftp
: an FTP server

http
: a World Wide Web server

gopher
: a Gopher server

mailto
: an email address

news
: a Usenet newsgroup

telnet
: a connection to a Telnet-based service

WAIS
: a WAIS server

A few other protocols are occasionally encountered including rmi (remote method invocation) and https (secure http).

The parts of a URL

The hostname part of the URL should be a valid Internet hostname like www.ora.com or shock.njit.edu. It can also be an IP address like 204.29.207.217 or 128.235.252.184.

The port number is optional. It's not necessary if the service is running on the default port, 80 for http servers.

The path points to a particular directory on the specified server. The path is relative to the document root of the server, not necessarily to the root of the file system on the server. In general a server, especially one open to the public, does not show its entire file system to clients. Rather it shows only the contents of a specified directory. This directory is called the server root, and all paths and filenames are relative to it. Thus on a Unix workstation all files that are available to the public may be in /var/public/html, but to somebody connecting from a remote machine this directory looks like the root of the file system.

The filename points to a particular file in the directory specified by the path. It is often omitted in which case it is left to the server's discretion what file, if any, to send. Many servers will send an index file for that directory, often called index.html or Welcome.html. Others will send a list of the files in the directory. Others may send an error message.

Section is used to reference a named anchor in an HTML document. Some documents refer to the section part of the URL as a "fragment." A named anchor is created in HTML document with a name tag like this

```
<A NAME="xtocid1902914">Comments</A>
```

A URL that points to this name, includes not only the filename, but also the named anchor separated from the rest of the URL by a # like this
`http://metalab.unc.edu/javafaq/javafaq.html#xtocid1902914`
For more detailed information about URLs, see

- W3C
- A Beginner's Guide to URLs
- RFC 1738
- RFC 1808

Links in 10 minutes

To make a hypertext link, you surround the text you want to be linked with <A></A> tags. Inside the <A> tag place an HREF attribute whose value is the URL you want to link too. For example,

```
Make sure you get the
<A HREF="http://metalab.unc.edu/javafaq/course/week5/exercises.html">
  exercises
</A>
from the web site.
```

Most browsers will underline the linked text, and color it blue or purple, but this is a presentation decision left up to the browser. The browser also provides the user with a means to activate the link (normally by clicking on the linked text). When the user activates the link, the browsers attempts to load the page specified in the link. If the URL in the anchor is not an HTTP URL, then the browser will take whatever action is appropriate, perhaps saving a file for an ftp URL or launching an email program for a mailto URL.

Relative URLs

When a web browser reads an HTML document, it has a great deal of information about the document. This includes the protocol used to retrieve the document, the name of the host where the document lives, and the path to that document on the host. Most of this is likely to be the same for many of the URLs in that document. Relative URLs inherit the protocol, hostname, and path of their parent document rather than respecifying it in each <A HREF> tag. Thus if any piece of the URL is missing, it is assumed to be the same as that of the document in which the URL is found. Such a URL is called a *relative URL*. In contrast, a completely specified URL is called an absolute URL. For example, suppose while browsing
`http://metalab.unc.edu/javafaq/books.html` you click on this hyperlink:

```
<a href="javafaq.html">the FAQ</A>
```

Your browser cuts `books.html` off the end of

```
http://metalab.unc.edu/javafaq/books.html
```

to get

```
http://metalab.unc.edu/javafaq/
```

Then it attaches `javafaq.html` onto the end of `http://metalab.unc.edu/javafaq/` to get

```
http://metalab.unc.edu/javafaq/javafaq.html
```

If the relative link begins with a `/`, then it is relative to the document root instead of relative to the current file. Thus if while browsing `http://metalab.unc.edu/javafaq/books.html` you clicked on this hyperlink:

```
<a href="/boutell/faq/www_faq.html">
```

Your browser would throw away `/javafaq/javafaq.html` and attach `/boutell/faq/www_faq.html` to the end of `http://metalab.unc.edu` to get `http://metalab.unc.edu/boutell/faq/www_faq.html`.

Relative URLs have a number of advantages. First and least they save a little typing. More importantly relative URLs allow entire trees of HTML documents to be moved or copied from one site to another without breaking all the internal links.

Hello World: The Applet

The reason people are excited about Java as more than just another OOP language is because it allows them to write interactive applets on the web. Hello World isn't a very interactive program, but let's look at a webbed version.

```
import java.applet.Applet;
import java.awt.Graphics;


public class HelloWorldApplet extends Applet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }

}
```

The applet version of HelloWorld is a little more complicated than the HelloWorld application, and it will take a little more effort to run it as well.

First type in the source code and save it into file called HelloWorldApplet.java. Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created. Now you need to create an HTML file that will include your applet. The following simple HTML file will do.

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>

<BODY>
This is the applet:<P>
<applet code="HelloWorldApplet.class" width="150" height="50">
</applet>
</BODY>
</HTML>
```

Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file. When you've done that, load the HTML file into a Java enabled browser like Internet Explorer

4.0 or Sun's applet viewer included with the JDK. You should see something like below, though of course the exact details depend on which browser you use.

If you're using the JDK 1.1 to compile your program, you should use the applet viewer, HotJava, Internet Explorer 4.0 or later, or Netscape 4.0.6 or later on Windows and Unix to view the applet. Netscape Navigator 4.0.5 and earlier and 3.x versions of Internet Explorer do not support Java 1.1. Furthermore, no Mac version of Navigator supports Java 1.1.

If the applet compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the .class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorld.html. Also make sure that you're using a version of Netscape or Internet Explorer which supports Java. Not all versions do.

In any case Netscape's Java support is less than the perfect so if you have trouble with an applet, the first thing to try is loading it into Sun's Applet Viewer instead. If the Applet Viewer has a problem, then chances are pretty good the problem is with the applet and not with the browser.

What is an Applet?

According to Sun "An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application....The `Applet` class provides a standard interface between applets and their environment."

Four definitions of applet:

- A small application
- A secure program that runs inside a web browser
- A subclass of `java.applet.Applet`
- An instance of a subclass of `java.applet.Applet`

```
public class Applet extends Panel
java.lang.Object
   |
   +----java.awt.Component
           |
           +----java.awt.Container
                   |
```

```
                +----java.awt.Panel
                        |
                        +----java.applet.Applet
```

The APPLET HTML Tag

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. The <APPLET> tag is similar to the <IMG> tag. Like <IMG> <APPLET> references a source file that is not part of the HTML page on which it is embedded. IMG elements do this with the SRC attribute. APPLET elements do this with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document. Thus if you're browsing http://metalab.unc.edu/javafaq/index.html and that page references an applet with CODE="Animation.class", then the file Animation.class should be at the URL http://metalab.unc.edu/javafaq/animation.class.

For reasons that remain a mystery to HTML authors everywhere if the applet resides somewhere other than the same directory as the page it lives on, you don't just give a URL to its location. Rather you point at the CODEBASE. The CODEBASE attribute is a URL that points at the directory where the .class file is. The CODE attribute is the name of the .class file itself. For instance if on the HTML page of the previous section you had written

```
<APPLET CODE="HelloWorldApplet.class" CODEBASE="classes"
WIDTH=200 HEIGHT=200>
</APPLET>
```

then the browser would have tried to find HelloWorldApplet.class in the classes directory in the same directory as the HTML page that included the applet. On the other hand if you had written

```
<APPLET CODE="HelloWorldApplet.class"
CODEBASE="http://www.foo.bar.com/classes" WIDTH=200 HEIGHT=200>
</APPLET>
```

then the browser would try to retrieve the applet from http://www.foo.bar.com/classes/HelloWorldApplet.class regardless of where the HTML page was.

In short the applet viewer will try to retrieve the applet from the URL given by the formula (CODEBASE + "/" + code). Once this URL is formed all the usual rules about relative and absolute URLs apply.

You can leave off the .class extension and just use the class name in the CODE attribute. For example,

```
<APPLET CODE="HelloWorldApplet"
CODEBASE="http://www.foo.bar.com/classes" WIDTH=200 HEIGHT=200>
</APPLET>
```

If the applet is in a non-default package, then the full package qualified name must be used. For example,

```
<APPLET CODE="com.macfaq.greeting.HelloWorldApplet"
CODEBASE="http://www.foo.bar.com/classes" WIDTH=200 HEIGHT=200>
```

```
</APPLET>
```
In this case the browser will look for http://www.foo.bar.com/classes/com/macfaq/greeting/HelloWorldApplet.class so the directory structure on the server should also mirror the package hierarchy.

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the applet. These numbers are specified in pixels and are required.

Spacing Preferences

The <APPLET> tag has several attributes to define how it is positioned on the page.

The ALIGN attribute defines how the applet's rectangle is placed on the page relative to other elements. Possible values include LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM and ABSBOTTOM. This attribute is optional.

You can specify an HSPACE and a VSPACE in pixels to set the amount of blank space between an applet and the surrounding text. The HSPACE and VSPACE attributes are optional.

```
<applet code="HelloWorldApplet.class"
CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10>
</APPLET>
The ALIGN, HSPACE, and VSPACE attributes are identical to the attributes of
the same name used by the <IMG> tag.
```

Alternate Text

The <APPLET> has an ALT attribute. An ALT attribute is used by a browser that understands the APPLET tag but for some reason cannot play the applet. For instance, if you've turned off Java in Netscape Navigator 3.0, then the browser should display the ALT text. Note that I said it **should**, not that it **does**. The ALT tag is optional.

```
<applet code="HelloWorldApplet.class"
CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10
ALT="Hello World!">
</APPLET>
```
ALT is not used by browsers that do not understand <APPLET> at all. For that purpose <APPLET> has been defined to require a closing tag, </APPLET>. All raw text between the opening and closing <APPLET> tags is ignored by a Java capable browser. However a non-Java capable browser will ignore the <APPLET> tags instead and read the text between them. For example the following HTML fragment says Hello to people both with and without Java capable browsers.

```
<applet code="HelloWorldApplet.class"
```

```
CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10
ALT="Hello World!">
Hello World!<P>
</APPLET>
```

Naming Applets

You can give an applet a name by using the `NAME` attribute of the `APPLET` tag. This allows communication between different applets on the same Web page.

```
<applet code="HelloWorldApplet.class" Name=Applet1
CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10
ALT="Hello World!">
Hello World!<P>
</APPLET>
```

JAR Archives

HTTP 1.0 uses a separate connection for each request. When you're downloading many small files, the time required to set up and tear down the connections can be a significant fraction of the total amount of time needed to load a page. It would be better if you could load all the HTML documents, images, applets, and sounds a page needed in one connection.

One way to do this without changing the HTTP protocol, is to pack all those different files into a single archive file, perhaps a zip archive, and just download that.

We aren't quite there yet. Browsers do not yet understand archive files, but in Java 1.1 applets do. You can pack all the images, sounds, and .class files an applet needs into one JAR archive and load that instead of the individual files. Applet classes do not have to be loaded directly. They can also be stored in JAR archives. To do this you use the `ARCHIVES` attribute of the `APPLET` tag

```
<APPLET CODE=HelloWorldApplet WIDTH=200 HEIGHT=100 ARCHIVES="HelloWorld.jar">
<hr>
Hello World!
<hr>
</APPLET>
```

In this example, the applet class is still `HelloWorldApplet`. However, there is no HelloWorldApplet.class file to be downloaded. Instead the class is stored inside the archive file HelloWorld.jar.

Sun provides a tool for creating JAR archives with its JDK 1.1. For example,

```
% jar cf HelloWorld.jar *.class
```

This puts all the .class files in the current directory in the file named "HelloWorld.jar". The syntax of the jar command is deliberately similar to the Unix tar command.

The OBJECT Tag

HTML 4.0 deprecates the `<APPLET>` tag. Instead you are supposed to use the `<OBJECT>` tag. For the purposes of ewbedding applets, the `<OBJECT>` tag is used almost exactly like the `<APPLET>` tag except that the `class` attribute becomes the `classid` attribute. For example,

```
<OBJECT classid="MyApplet.class"
CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10>
</OBJECT>
```

The `<OBJECT>` tag is also used to embed ActiveX controls and other kinds of active content, and it has a few additional attributes to allow it to do that. However, for the purposes of Java you don't need to know about these.

The `<OBJECT>` tag is supported by Netscape ???? and later and Internet Explorer ???? and later. It is not supported by earlier versions of those browsers so `<APPLET>` is unlikely to disappear anytime soon.

You can support both by placing an `<APPLET>` element inside an `<OBJECT>` element like this:

```
<OBJECT classid="MyApplet.class" width=200 height=200>
<APPLET code="MyApplet.class" width=200 height=200>
</APPLET>
</OBJECT>
```

Browsers that understand `<OBJECT>` will ignore its content while browsers that don't will display its content.

`PARAM` elements are the same for `<OBJECT>` as for `<APPLET>`.

For the complete story, you can read about the `<OBJECT>` tag in the HTML 4.0 specification.

Finding an Applet's Size

When running inside a web browser the size of an applet is set by the height and width attributes and cannot be changed by the applet. Many applets need to know their own size. After all you don't want to draw outside the lines. :-)

Retrieving the applet size is straightforward with the `getSize()` method. `java.applet.Applet` inherits this method from `java.awt.Component`. `getSize()` returns a `java.awt.Dimension`

object. A `Dimension` object has two public `int` fields, `height` and `width`. Below is a simple applet that prints its own dimensions.

```
import java.applet.*;
import java.awt.*;


public class SizeApplet extends Applet {

  public void paint(Graphics g) {

    Dimension appletSize = this.getSize();
    int appletHeight = appletSize.height;
    int appletWidth = appletSize.width;

    g.drawString("This applet is " + appletHeight +
      " pixels high by " + appletWidth + " pixels wide.",
      15, appletHeight/2);

  }

}
```

Note how the applet's height is used to decide where to draw the text. You'll often want to use the applet's dimensions to determine how to place objects on the page. The applet's width wasn't used because it made more sense to left justify the text rather than center it. In other programs you'll have occasion to use the applet width too.

Passing Parameters to Applets

Parameters are passed to applets in NAME=VALUE pairs in `<PARAM>` tags between the opening and closing `APPLET` tags. Inside the applet, you read the values passed through the `PARAM` tags with the `getParameter()` method of the `java.applet.Applet` class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

```
import java.applet.*;
import java.awt.*;

public class DrawStringApplet extends Applet {

  privte String defaultMessage = "Hello!";

  public void paint(Graphics g) {

    String inputFromPage = this.getParameter("Message");
    if (inputFromPage == null) inputFromPage = defaultMessage;
```

```
    g.drawString(inputFromPage, 50, 25);

  }

}
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
This is the applet:<P>
<APPLET code="DrawStringApplet.class" width="300" height="50">
<PARAM name="Message" value="Howdy, there!">
This page will be very boring if your
browser doesn't understand Java.
</APPLET>
</BODY>
</HTML>
```

Of course you are free to change "Howdy, there!" to a "message" of your choice. You only need to change the HTML, not the Java source code. PARAMs let you customize applets without changing or recompiling the code.

This applet is very similar to the HelloWorldApplet. However rather than hardcoding the message to be printed it's read into the variable `inputFromPage` from a PARAM in the HTML.

You pass `getParameter()` a string that names the parameter you want. This string should match the name of a `<PARAM>` tag in the HTML page. `getParameter()` returns the value of the parameter. All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.

The `<PARAM>` HTML tag is also straightforward. It occurs between `<APPLET>` and `</APPLET>`. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the value of the PARAM as a String. Both should be enclosed in double quote marks if they contain white space.

An applet is not limited to one PARAM. You can pass as many named PARAMs to an applet as you like. An applet does not necessarily need to use all the PARAMs that are in the HTML. Additional PARAMs can be safely ignored.

Processing An Unknown Number Of Parameters

Most of the time you have a fairly good idea of what parameters will and won't be passed to your applet. However some of the time there will be an undetermined number of parameters. For instance Sun's imagemap applet passes each "hot button" as a parameter. Different imagemaps have different numbers of hot buttons. Another applet might want to pass a series of URL's to different sounds to be played in sequence. Each URL could be passed as a separate parameter.

Or perhaps you want to write an applet that displays several lines of text. While it would be possible to cram all this information into one long string, that's not too friendly to authors who want to use your applet on their pages. It's much more sensible to give each line its own <PARAM> tag. If this is the case, you should name the tags via some predictable and numeric scheme. For instance in the text example the following set of <PARAM> tags would be sensible:

```
<PARAM name="Line1" value="There once was a man from Japan">
<PARAM name="Line2" value="Whose poetry never would scan">
<PARAM name="Line3" value="When asked reasons why,">
<PARAM name="Line4" value="He replied, with a sigh:">
<PARAM name="Line5" value="I always try to get as many
syllables into the last line as I can.">
```

The program below displays this limerick. Lines are accumulated into an array of strings called `poem`. A `for` loop fills the array with the different lines of poetry. There are 101 spaces in the array, but since you won't normally need that many, an `if` clause tests to see whether the attempt to get a parameter was successful by checking to see if the line is `null`. As soon as one fails, the loop is broken. Once the loop is finished `num_lines` is decremented by one because the last line the loop tried to read wasn't there.

The `paint()` method loops through the poem array and prints each String on the screen, incrementing the `y` position by fifteen pixels each step so you don't draw one line on top of the other.

Processing An Unknown Number Of Parameters

```
import java.applet.*;
import java.awt.*;

public class PoetryApplet extends Applet {

  private String[] poem = new String[101];
  private int numLines;

  public void init() {
```

```
    String nextline;

    for (numLines = 1; numLines < poem.length; numLines++) {
      nextline = this.getParameter("Line" + numLines);
      if (nextline == null) break;
      poem[numLines] = nextline;
    }
    numLines--;

  }

  public void paint(Graphics g) {

    int y = 15;

    for (int i=1; i <= numLines; i++) {
      g.drawString(poem[i], 5, y);
      y += 15;
    }

  }

}
```

Here's the applet:

You might think it would be useful to be able to process an arbitrary list of parameters without knowing their names in advance, if nothing else so you could return an error message to the page designer. Unfortunately there's no way to do it in Java 1.0 or 1.1. It may appear in future versions.

Applet Security

The possibility of surfing the Net, wandering across a random page, playing an applet and catching a virus is a fear that has scared many uninformed people away from Java. This fear has also driven a lot of the development of Java in the direction it's gone. Earlier I discussed various security features of Java including automatic garbage collection, the elimination of pointer arithmetic and the Java interpreter. These serve the dual purpose of making the language simple for programmers and secure for users. You can surf the web without worrying that a Java applet will format your hard disk or introduce a virus into your system.

In fact both Java applets and applications are much safer in practice than code written in traditional languages. This is because even code from trusted sources is likely to have bugs. However Java programs are much less susceptible to common bugs involving memory access than are programs written in traditional languages like C. Furthermore the Java runtime environment provides a fairly robust means of trapping bugs before they bring down your system.

Most users have many more problems with bugs than they do with deliberately malicious code. Although users of Java applications aren't protected from out and out malicious code, they are largely protected from programmer errors.

Applets implement additional security restrictions that protect users from malicious code too. This is accomplished through the `java.lang.SecurityManager` class. This class is subclassed to provide different security environments in different virtual machines. Regrettably implementing this additional level of protection does somewhat restrict the actions an applet can perform. Let's explore exactly what an applet can and cannot do.

Applet Security

The possibility of surfing the Net, wandering across a random page, playing an applet and catching a virus is a fear that has scared many uninformed people away from Java. This fear has also driven a lot of the development of Java in the direction it's gone. Earlier I discussed various security features of Java including automatic garbage collection, the elimination of pointer arithmetic and the Java interpreter. These serve the dual purpose of making the language simple for programmers and secure for users. You can surf the web without worrying that a Java applet will format your hard disk or introduce a virus into your system.

In fact both Java applets and applications are much safer in practice than code written in traditional languages. This is because even code from trusted sources is likely to have bugs. However Java programs are much less susceptible to common bugs involving memory access than are programs written in traditional languages like C. Furthermore the Java runtime environment provides a fairly robust means of trapping bugs before they bring down your system. Most users have many more problems with bugs than they do with deliberately malicious code. Although users of Java applications aren't protected from out and out malicious code, they are largely protected from programmer errors.

Applets implement additional security restrictions that protect users from malicious code too. This is accomplished through the `java.lang.SecurityManager` class. This class is subclassed to provide different security environments in different virtual machines. Regrettably implementing this additional level of protection does somewhat restrict the actions an applet can perform. Let's explore exactly what an applet can and cannot do.

Who Can an Applet Talk To?

By default an applet can only open network connections to the system from which the applet was downloaded. This system is called the *codebase*. An applet cannot talk to an arbitrary system on the Internet. Any communication between different client systems must be mediated through the server.

The concern is that if connections to arbitrary hosts were allowed, then a malicious applet might be able to make connections to other systems and launch network based attacks on other machines in an organization's internal network. This would be an especially large problem

because the machine's inside a firewall may be configured to trust each other more than they would trust any random machine from the Internet. If the internal network is properly protected by a firewall, this might be the only way an external machine could even talk to an internal machine. Furthermore arbitrary network connections would allow crackers to more easily hide their true location by passing their attacks through several applet intermediaries.

HotJava, Sun's applet viewer, and Internet Explorer (but not Netscape) let you grant applets permission to open connections to any system on the Internet, though this is not enabled by default.

How much CPU time does an applet get?

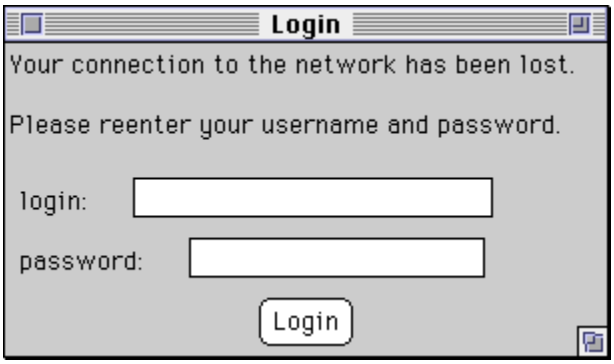
One of the few legitimate concerns about hostile applets is excessive use of CPU time. It is possible on a non-preemptively multitasking system (specifically the Mac) to write an applet that uses so much CPU time in a tight loop that it effectively locks up the host system. This is not a problem on preemptively multitasking systems like Solaris and Windows NT. Even on those platforms, though, it is possible for an applet to force the user to kill their web browser, possibly losing accumulated bookmarks, email and other work.

It's also possible for an applet to use CPU time for purposes other than the apparent intent of the applet. For instance, a popular applet could launch a Chinese lottery attack on a Unix password file. A popular game applet could launch a thread in the background which tried a random assortment of keys to break a DES encrypted file. If the key was found, then a network connection could be opened to the applet server to send the decrypted key back. The more popular the applet was the faster the key would be found. The ease with which Java applets are decompiled would probably mean that any such applet would be discovered, but there really isn't a way to prevent it from running in the first place.

User Security Issues and Social Engineering

Contrary to popular belief most computer break-ins by external hackers don't happen because of great knowledge of operating system internals and network protocols. They happen because a hacker went digging through a company's garbage and found a piece of paper with a password written on it, or perhaps because they talked to a low-level bureaucrat on the phone, convinced this person they were from the local data processing department and that they needed him or her to change their password to "DEBUG."

This is sort of attack is called social engineering. Java applets introduce a new path for social engineering. For instance imagine an applet that pops up a dialog box that says, "You have lost your connection to the network. Please enter your username and password to reconnect." How many people would blindly enter their username and password without thinking? Now what if the box didn't really come from a lost network connection but from a hacker's applet? And instead of reconnecting to the network (a connection that was never lost in the first place) the username and password was sent over the Internet to the cracker? See the problem?



Preventing Applet Based Social Engineering Attacks

To help prevent this, Java applet windows are specifically labeled as such with an ugly bar that says: "Warning: Applet Window" or "Unsigned Java Applet Window." The exact warning message varies from browser to browser but in any case should be enough to prevent the more obvious attacks on clueless users. It still assumes the user understands what "Unsigned Java Applet Window" means and that they shouldn't type their password or any sensitive information in such a window. User education is the first part of any real security policy.

Content Issues

Some people claim that Java is insecure because it can show the user erotic pictures and play flatulent noises. By this standard the entire web is insecure. Java makes no determination of the content of an applet. Any such determination would require artificial intelligence and computers far more powerful than what we have today.

The Basic Applet Life Cycle

1. The browser reads the HTML page and finds any `<APPLET>` tags.
2. The browser parses the `<APPLET>` tag to find the `CODE` and possibly `CODEBASE` attribute.
3. The browser downloads the .class file for the applet from the URL found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a `java.lang.Class` object.

5. The browser instantiates the applet class to form an applet object. This requires the applet to have a noargs constructor.
6. The browser calls the applet's `init()` method.
7. The browser calls the applet's `start()` method.
8. While the applet is running, the browser passes any events intended for the applet, e.g. mouse clicks, key presses, etc., to the applet's `handleEvent()` method. Update events are used to tell the applet that it needs to repaint itself.
9. The browser calls the applet's `stop()` method.
10. The browser calls the applet's `destroy()` method.

The Basic Applet Life Cycle

All applets have the following four methods:

```
public void init();
public void start();
public void stop();
public void destroy();
```

They have these methods because their superclass, `java.applet.Applet`, has these methods. (It has others too, but right now I just want to talk about these four.)

In the superclass, these are simply do-nothing methods. For example,

```
public void init() {}
```

Subclasses may override these methods to accomplish certain tasks at certain times. For instance, the `init()` method is a good place to read parameters that were passed to the applet via `<PARAM>` tags because it's called exactly once when the applet starts up. However, they do not have to override them. Since they're declared in the superclass, the Web browser can invoke them when it needs to without knowing in advance whether the method is implemented in the superclass or the subclass. This is a good example of polymorphism.

init(), start(), stop(), and destroy()

The `init()` method is called exactly once in an applet's life, when the applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and set up the user interface. Most applets have `init()` methods.

The `start()` method is called at least once in an applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets you write will not have explicit `start()` methods and will merely inherit one from their superclass. A `start()` method is often used to start any threads the applet will need while it runs.

The `stop()` method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's `start()` method will be called if at some later point the browser returns to the page containing the applet. In some cases the `stop()` method may be called multiple times in an applet's life. Many applets you write will not have explicit `stop()` methods and will merely inherit one from their superclass. Your applet should use the `stop()` method to pause any running threads. When your applet is stopped, it *should* not use any CPU cycles.

The `destroy()` method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final clean-up. For example, an applet that stores state on the server might send some data back to the server before it's terminated. many applets will not have explicit `destroy()` methods and just inherit one from their superclass.

For example, in a video applet, the `init()` method might draw the controls and start loading the video file. The `start()` method would wait until the file was loaded, and then start playing it. The `stop()` method would pause the video, but not rewind it. If the `start()` method were called again, the video would pick up where it left off; it would not start over from the beginning. However, if `destroy()` were called and then `init()`, the video would start over from the beginning.

In the JDK's appletviewer, selecting the Restart menu item calls `stop()` and then `start()`. Selecting the Reload menu item calls `stop()`, `destroy()`, and `init()`, in that order. (Normally the byte codes will also be reloaded and the HTML file reread though Netscape has a problem with this.)

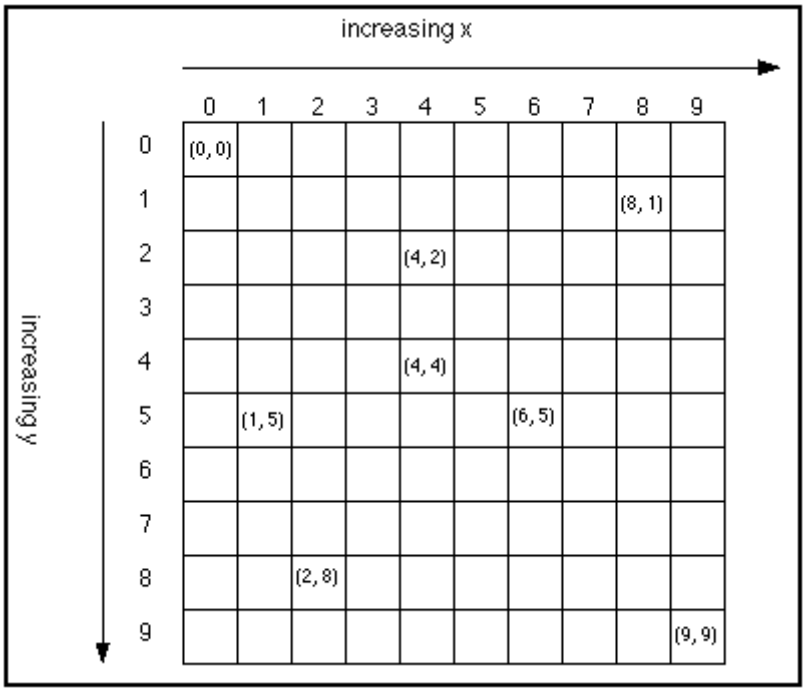
The applet `start()` and `stop()` methods are not related to the similarly named methods in the `java.lang.Thread` class.

Your own code may occasionally invoke `start()` and `stop()`. For example, it's customary to stop playing an animation when the user clicks the mouse in the applet and restart it when they click the mouse again.

Your own code can also invoke `init()` and `destroy()`, but this is normally a bad idea. Only the environment should call `init()` and `destroy()`.

The Coordinate System

Java uses the standard, two-dimensional, computer graphics coordinate system. The first visible pixel in the upper left-hand corner of the applet canvas is (0, 0). Coordinates increase to the right and down.



Graphics Objects

In Java all drawing takes place via a `Graphics` object. This is an instance of the class `java.awt.Graphics`.

Initially the `Graphics` object you use will be the one passed as an argument to an applet's `paint()` method. Later you'll see other `Graphics` objects too. Everything you learn today about drawing in an applet transfers directly to drawing in other objects like Panels, Frames, Buttons, Canvases and more.

Each `Graphics` object has its own coordinate system, and all the methods of `Graphics` including those for drawing Strings, lines, rectangles, circles, polygons and more. Drawing in Java starts with particular `Graphics` object. You get access to the `Graphics` object through the `paint(Graphics g)` method of your applet. Each draw method call will look like `g.drawString("Hello World", 0, 50)` where `g` is the particular `Graphics` object with which you're drawing.

For convenience's sake in this lecture the variable `g` will always refer to a preexisting object of the `Graphics` class. As with any other method you are free to use some other name for the particular `Graphics` context, `myGraphics` or `appletGraphics` perhaps.

Drawing Lines

Drawing straight lines with Java is easy. Just call
`g.drawLine(x1, y1, x2, y2)`
where `(x1, y1)` and `(x2, y2)` are the endpoints of your lines and `g` is the `Graphics` object you're drawing with.

This program draws a line diagonally across the applet.

```
import java.applet.*;
import java.awt.*;

public class SimpleLine extends Applet {

 public void paint(Graphics g) {

    g.drawLine(0, 0, this.getSize().width, this.getSize().height);

 }

}
```

Here's the result

Drawing Rectangles

Drawing rectangles is simple. Start with a `Graphics` object `g` and call its `drawRect()` method:

```
public void drawRect(int x, int y, int width, int height)
```

As the variable names suggest, the first `int` is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

This uses `drawRect()` to draw a rectangle around the sides of an applet.

```
import java.applet.*;
import java.awt.*;

public class RectangleApplet extends Applet {

  public void paint(Graphics g) {

    g.drawRect(0, 0, this.getSize().width - 1, this.getSize().height - 1);

  }
```

```
}
```

Remember that `getSize().width` is the width of the applet and `getSize().height` is its height.

Why was the rectangle drawn only to `getSize().height-1` and `getSize().width-1`?

Remember that the upper left hand corner of the applet starts at (0, 0), not at (1, 1). This means that a 100 by 200 pixel applet includes the points with x coordinates between 0 and 99, not between 0 and 100. Similarly the y coordinates are between 0 and 199 inclusive, not 0 and 200.

There is no separate `drawSquare()` method. A square is just a rectangle with equal length sides, so to draw a square call `drawRect()` and pass the same number for both the `height` and `width` arguments.

Filling Rectangles

The `drawRect()` method draws an open rectangle, a box if you prefer. If you want to draw a filled rectangle, use the `fillRect()` method. Otherwise the syntax is identical.

This program draws a filled square in the center of the applet. This requires you to separate the applet width and height from the rectangle width and height. Here's the code:

```
import java.applet.*;
import java.awt.*;

public class FillAndCenter extends Applet {

  public void paint(Graphics g) {

    int appletHeight = this.getSize().height;
    int appletWidth  = this.getSize().width;
    int rectHeight   = appletHeight/3;
    int rectWidth    = appletWidth/3;
    int rectTop      = (appletHeight - rectHeight)/2;
    int rectLeft     = (appletWidth - rectWidth)/2;

    g.fillRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);

  }

}
```

Clearing Rectangles

It is also possible to clear a rectangle that you've drawn. The syntax is exactly what you'd expect:

```
public abstract void clearRect(int x, int y, int width, int height)
```

This program uses `clearRect()` to blink a rectangle on the screen.

```
import java.applet.*;
import java.awt.*;

public class Blink extends Applet {

  public void paint(Graphics g) {

    int appletHeight = this.getSize().height;
    int appletWidth  = this.getSize().width;
    int rectHeight   = appletHeight/3;
    int rectWidth    = appletWidth/3;
    int rectTop      = (appletHeight - rectHeight)/2;
    int rectLeft     = (appletWidth - rectWidth)/2;

    for (int i=0; i < 1000; i++) {
      g.fillRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
      g.clearRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
    }

  }

}
```

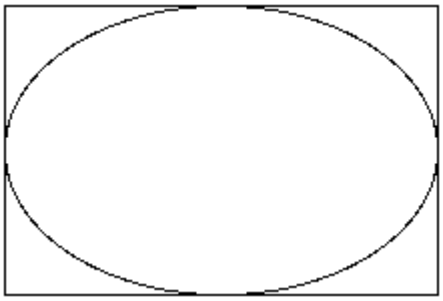
This is not how you should do animation in practice, but this is the best we can do until we introduce threads.

Ovals and Circles

Java has methods to draw outlined and filled ovals. As you'd probably guess these methods are called `drawOval()` and `fillOval()` respectively. As you might not guess they take identical arguments to `drawRect()` and `fillRect()`, i.e.

```
public void drawOval(int left, int top, int width, int height)
public void fillOval(int left, int top, int width, int height)
```

Instead of the dimensions of the oval itself, the dimensions of the smallest rectangle which can enclose the oval are specified. The oval is drawn as large as it can be to touch the rectangle's edges at their centers. This picture may help:



The arguments to `drawOval()` are the same as the arguments to `drawRect()`. The first `int` is the left hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height.

There is no special method to draw a circle. Just draw an oval inside a square.

Java also has methods to draw outlined and filled arcs. They're similar to `drawOval()` and `fillOval()` but you must also specify a starting and ending angle for the arc. Angles are given in degrees. The signatures are:

```
public void drawArc(int left, int top, int width, int height,
 int startangle, int stopangle)
public void fillArc(int left, int top, int width, int height,
 int startangle, int stopangle)
```

The rectangle is filled with an arc of the largest circle that could be enclosed within it. The location of 0 degrees and whether the arc is drawn clockwise or counter-clockwise are currently platform dependent.

Bullseye

This is a simple applet which draws a series of filled, concentric circles alternating red and white, in other words a bullseye.

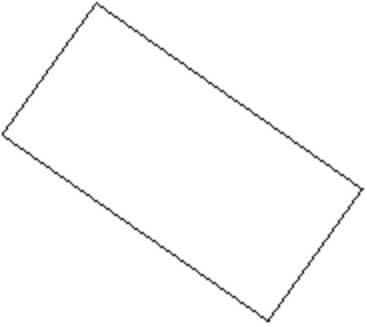
```
import java.applet.*;
import java.awt.*;


public class Bullseye extends Applet {

  public void paint(Graphics g) {

    int appletHeight = this.getSize().height;
    int appletWidth = this.getSize().width;

    for (int i=8; i >= 0; i--) {
      if ((i % 2) == 0) g.setColor(Color.red);
      else g.setColor(Color.white);
```



```
                // Center the rectangle
                int rectHeight = appletHeight*i/8;
                int rectWidth  = appletWidth*i/8;
                int rectLeft   = appletWidth/2  - i*appletWidth/16;
                int rectTop    = appletHeight/2 - i*appletHeight/16;
        g.fillOval(rectLeft, rectTop, rectWidth, rectHeight);
      }

    }

}
```

The .class file that draws this image is only 684 bytes. The equivalent GIF image is 1,850 bytes, almost three times larger.

Almost all the work in this applet consists of centering the enclosing rectangles inside the applet. The lines in bold do that. The first two lines just set the height and the width of the rectangle to the appropriate fraction of the applet's height and width. The next two lines set the position of the upper left hand corner. Once the rectangle is positioned, drawing the oval is easy.

Polygons

In Java rectangles are defined by the position of their upper left hand corner, their height, and their width. However it is implicitly assumed that there is in fact an upper left hand corner. Not all rectangles have an upper left hand corner. For instance consider the rectangle below.

Where is its upper left hand corner? What's been assumed so far is that the sides of the rectangle are parallel to the coordinate axes. You can't yet handle a rectangle that's been rotated at an arbitrary angle.

There are some other things you can't handle either, triangles, stars, rhombuses, kites, octagons and more. To take care of this broad class of shapes Java has a `Polygon` class.

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the `Polygon` class is

```
public Polygon(int[] xpoints, int[] ypoints, int npoints)
```

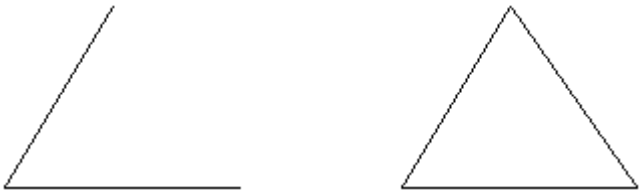
`xpoints` is an array that contains the x coordinates of the polygon. `ypoints` is an array that contains the y coordinates. Both should have the length `npoints`. Thus to construct a 3-4-5 right triangle with the right angle on the origin you would type

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
```

```
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```
To actually draw the polygon you use `java.awt.Graphics`'s `drawPolygon(Polygon p)` method within your `paint()` method like this:
```
g.drawPolygon(myTriangle);
```
You can pass the arrays and number of points directly to the `drawPolygon()` method if you prefer:
```
g.drawPolygon(xpoints, ypoints, xpoints.length);
```
There's also an overloaded `fillPolygon()` method. The syntax is exactly as you expect:
```
g.fillPolygon(myTriangle);
g.fillPolygon(xpoints, ypoints, xpoints.length());
```

Polylines

Java automatically closes the polygons it draws. That is it draws polygons that look like the one on the right rather than the one on the left.



If you don't want your polygons to be closed, you can draw a polyline instead with the `Graphics` class's `drawPolyline()` method

```
public abstract void drawPolyline(int xPoints[], int yPoints[], int nPoints)
```

Loading Images

Polygons, ovals, lines and text cover a lot of ground. The remaining graphic object you need is an image. Images in Java are bitmapped GIF or JPEG files that can contain pictures of just about anything. You can use any program at all to create them as long as that program can save in GIF or JPEG format.

Images displayed by Java applets are retrieved from the web via a URL that points to the image file. An applet that displays a picture must have a URL to the image its going to display. Images can be stored on a web server, a local hard drive or anywhere else the applet can point to via a URL. Make sure you put your images somewhere the person viewing the applet can access them. A file URL that points to your local hard drive may work while you're developing an applet, but it won't be of much use to someone who comes in over the web.

Typically you'll put images in the same directory as either the applet or the HTML file. Though it doesn't absolutely have to be in one of these two locations, storing it there will probably be more convenient. Put the image with the applet .class file if the image will be used for all instances of the applet. Put the applet with the HTML file if different instances of the applet will use different images. A third alternative is to put all the images in a common location and use PARAMs in the HTML file to tell Java where the images are.

If you know the exact URL for the image you wish to load, you can load it with the `getImage()` method:

```
URL imageURL = new URL("http://www.prenhall.com/logo.gif");
java.awt.Image img = this.getImage(imageURL);
```
You can compress this into one line as follows
```
Image img = this.getImage(new URL("http://www.prenhall.com/logo.gif"));
```

The `getImage()` method is provided by `java.applet.Applet`. The `URL` class is provided by `java.net.URL`. Be sure to import it.

Code and Document Bases

If you don't know the exact URL of the image, but you do know its name and that it's in the same directory as the applet, you can use an alternate form of `getImage()` that takes a URL and a filename. Use the applet's `getCodeBase()` method to return the URL to the applet directory like this:
```
Image img = this.getImage(this.getCodeBase(), "test.gif");
```
The `getCodeBase()` method returns a `URL` object that points to the directory where the applet came from.

Finally if the image file is stored in the same directory as the HTML file, use the same `getImage()` method but pass it `getDocumentBase()` instead. This returns a URL that points at the directory which contains the HTML page in which the applet is embedded.

```
Image img = this.getImage(this.getDocumentBase(), "test.gif");
```
If an image is loaded from the Internet, it may take some time for it to be fully downloaded. Most of the time you don't need to worry about this. You can draw the Image as soon as you've connected it to a URL using one of the above methods. Java will update it as more data becomes available without any further intervention on your part.

Load all the images your applet needs in the `init()` method. In particular you do not want to load them in the `paint()` method. If you do they will be reloaded every time your applet repaints itself, and applet performance will be abysmal.

Drawing Images at Actual Size

Once the image is loaded draw it in the `paint()` method using the `drawImage()` method like this
`g.drawImage(img, x, y, io)`
`img` is a member of the `Image` class which you should have already loaded in your `init()` method. `x` is the x coordinate of the upper left hand corner of the image. `y` is the y coordinate of the upper left hand corner of the image. `io` is a member of a class which implements the `ImageObserver` interface. The `ImageObserver` interface is how Java handles the asynchronous updating of an Image when it's loaded from a remote web site rather than directly from the hard drive. java.applet.Applet implements `ImageObserver` so for now just pass the keyword `this` to `drawImage()` to indicate that the current applet is the `ImageObserver` that should be used.

A `paint()` method that does nothing more than draw an Image starting at the upper left hand corner of the applet may look like this

```
public void paint(Graphics g) {
  g.drawImage(img, 0, 0, this);
}
```

This draws the image at the actual size of the picture.

Scaling Images

You can scale an image into a particular rectangle using this version of the `drawImage()` method:
```
public boolean drawImage(Image img, int x, int y, int width,
 int height, ImageObserver io)
```
`width` and `height` specify the size of the rectangle to scale the image into. All other arguments are the same as before. If the scale is not in proportion to the size of the image, it can end up looking quite squashed.

To avoid disproportionate scaling use the image's `getHeight()` and `getWidth()` methods to determine the actual size. Then scale appropriately. For instance this is how you would draw an Image scaled by one quarter in each dimension:

```
g.drawImage(img, 0, 0, img.getWidth(this)/4, img.getHeight(this)/4, this);
```
This program reads a GIF file in the same directory as the HTML file and displays it at a specified magnification. The name of the GIF file and the magnification factor are specified via PARAMs.
```
import java.awt.*;
import java.applet.*;

public class MagnifyImage extends Applet {

  private Image image;
  private int scaleFactor;

  public void init() {
    String filename  = this.getParameter("imagefile");
    this.image       = this.getImage(this.getDocumentBase(), filename);
    this.scaleFactor = Integer.parseInt(this.getParameter("scalefactor"));
```

```
  }

  public void paint (Graphics g) {
    int width    = this.image.getWidth(this);
    int height   = this.image.getHeight(this);
    scaledWidth  = width * this.scaleFactor;
    scaledHeight = height * this.scaleFactor;
    g.drawImage(this.image, 0, 0, scaledWidth, scaledHeight, this);
  }

}
```

This applet is straightforward. The `init()` method reads two PARAMs, one the name of the image file, the other the magnification factor. The `paint()` method calculates the scale and then draws the image.

Scaling Images

You may ask why the scale factor is calculated in the `paint()` method rather than the `init()` method. Some time could be saved by not recalculating the image's height and width every time the image is painted. After all the image size should be constant.

In this application the size of the image doesn't change, and indeed it will be a rare Image that changes size in the middle of an applet. However in the `init()` method the `Image` probably hasn't fully loaded. If instead you were to try the program below, which does exactly that, you'd see that the image wasn't scaled at all. The reason is that in this version the image hasn't loaded when you make the calls to `getWidth()` and `getHeight()` so they both return -1.

```
import java.applet.*;
import java.awt.*;


public class MagnifyImage extends Applet {

  private Image theImage;
  private int   scaledWidth;
  private int   scaledHeight;

  public void init() {

    String filename = this.getParameter("imagefile");
    theImage        = this.getImage(this.getDocumentBase(), filename);
    int scalefactor = Integer.parseInt(this.getParameter("scalefactor"));
    int width       = theImage.getWidth(this);
    int height      = theImage.getHeight(this);
    scaledWidth     =  width  * scalefactor;
    scaledHeight    =  height * scalefactor;

  }

  public void paint (Graphics g) {
```

```
    g.drawImage(theImage, 0, 0, scaledWidth, scaledHeight, this);
  }
}
```

Color

Color is a class in the AWT. Individual colors like red or mauve are instances of this class, `java.awt.Color`. Be sure to import it if you want to use other than the default colors. You create new colors using the same RGB triples that you use to set background colors on web pages. However you use decimal numbers instead of the hex values used by the `bgcolor` tag. For example medium gray is `Color(127, 127, 127)`. Pure white is `Color(255, 255, 255)`. Pure red is `(255, 0, 0)` and so on. As with any variable you should give your colors descriptive names. For instance

```
Color medGray = new Color(127, 127, 127);
Color cream = new Color(255, 231, 187);
Color lightGreen = new Color(0, 55, 0);
```

A few of the most common colors are available by name. These are

- `Color.black`
- `Color.blue`
- `Color.cyan`
- `Color.darkGray`
- `Color.gray`
- `Color.green`
- `Color.lightGray`
- `Color.magenta`
- `Color.orange`
- `Color.pink`
- `Color.red`
- `Color.white`
- `Color.yellow`

Color

Color is not a property of a particular rectangle, string or other thing you may draw. Rather color is a part of the `Graphics` object that does the drawing. To change colors you change the color of your `Graphics` object. Then everything you draw from that point forward will be in the new color, at least until you change it again.

When an applet starts running the color is set to black by default. You can change this to red by calling `g.setColor(Color.red)`. You can change it back to black by calling `g.setColor(Color.black)`. The following code fragment shows how you'd draw a pink String followed by a green one:

```
g.setColor(Color.pink);
g.drawString("This String is pink!", 50, 25);
g.setColor(Color.green);
g.drawString("This String is green!", 50, 50);
```

Remember everything you draw after the last line will be drawn in green. Therefore before you start messing with the color of the pen its a good idea to make sure you can go back to the color you started with. For this purpose Java provides the `getColor()` method. You use it like follows:

```
Color oldColor = g.getColor();
g.setColor(Color.pink);
g.drawString("This String is pink!", 50, 25);
g.setColor(Color.green);
g.drawString("This String is green!", 50, 50);
g.setColor(oldColor);
```

System Colors

In Java 1.1, the `java.awt.SystemColor` class is a subclass of `java.awt.Color` which provides color constants that match native component colors. For example, if you wanted to make the background color of your applet, the same as the background color of a window, you might use this `init()` method:

```
public void paint (Graphics g) {

  g.setColor(SystemColor.control);
  g.fillRect(0, 0, this.getSize().width, this.getSize().height);

}
```

These are the available system colors:

- `SystemColor.desktop // Background color of desktop`
- `SystemColor.activeCaption // Background color for captions`
- `SystemColor.activeCaptionText // Text color for captions`
- `SystemColor.activeCaptionBorder // Border color for caption text`
- `SystemColor.inactiveCaption // Background color for inactive captions`
- `SystemColor.inactiveCaptionText // Text color for inactive captions`
- `SystemColor.inactiveCaptionBorder // Border color for inactive captions`
- `SystemColor.window // Background for windows`
- `SystemColor.windowBorder // Color of window border frame`
- `SystemColor.windowText // Text color inside windows`
- `SystemColor.menu // Background for menus`
- `SystemColor.menuText // Text color for menus`
- `SystemColor.text // background color for text`
- `SystemColor.textText // text color for text`
- `SystemColor.textHighlight // background color for highlighted text`

- SystemColor.textHighlightText // text color for highlighted text
- SystemColor.control // Background color for controls
- SystemColor.controlText // Text color for controls
- SystemColor.controlLtHighlight // Light highlight color for controls
- SystemColor.controlHighlight // Highlight color for controls
- SystemColor.controlShadow // Shadow color for controls
- SystemColor.controlDkShadow // Dark shadow color for controls
- SystemColor.inactiveControlText // Text color for inactive controls
- SystemColor.scrollbar // Background color for scrollbars
- SystemColor.info // Background color for spot-help text
- SystemColor.infoText // Text color for spot-help text

Fonts

You've already seen one example of drawing text in the `HelloWorldApplet` program of the last chapter. You call the `drawString()` method of the `Graphics` object. This method is passed the `String` you want to draw as well as an x and y coordinate. If `g` is a `Graphics` object, then the syntax is

```
g.drawString(String s, int x, int y)
```

The `String` is simply the text you want to draw. The two integers are the x and y coordinates of the lower left-hand corner of the `String`. The `String` will be drawn above and to the right of this point. However letters with descenders like y and p may have their descenders drawn below the line.

Until now all the applets have used the default font, probably some variation of Helvetica though this is platform dependent. However unlike HTML Java does allow you to choose your fonts. Java implementations are guaranteed to have a serif font like Times that can be accessed with the name "Serif", a monospaced font like courier that can be accessed with the name "Mono", and a sans serif font like Helvetica that can be accessed with the name "SansSerif".

The following applet lists the fonts available on the system it's running on. It does this by using the `getFontList()` method from `java.awt.Toolkit`. This method returns an array of strings containing the names of the available fonts. These may or may not be the same as the fonts installed on your system. It's implementation dependent whether or not all the fonts a system has are available to the applet.

```
import java.awt.*;
import java.applet.*;


public class FontList extends Applet {

  private String[] availableFonts;

  public void init () {
```

```
    Toolkit t = Toolkit.getDefaultToolkit();
    availableFonts = t.getFontList();

  }

  public void paint(Graphics g) {

    for (int i = 0; i < availableFonts.length; i++) {
      g.drawString(availableFonts[i], 5, 15*(i+1));
    }
  }

}
```

Choosing Font Faces and Sizes

Choosing a font face is easy. First you create a new `Font` object. Then you call `g.setFont(Font f)`. To instantiate a `Font` object use this constructor:

```
public Font(String name, int style, int size)
```

`name` is the name of the font family, e.g. "Serif", "SansSerif", or "Mono".

`size` is the size of the font in points. In computer graphics a point is considered to be equal to one pixel. 12 points is a normal size font. 14 points is probably better on most computer displays. Smaller point sizes look good on paper printed with a high resolution printer, but not in the lower resolutions of a computer monitor.

`style` is an mnemonic constant from `java.awt.Font` that tells whether the text will be bold, italic or plain. The three constants are `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The program below prints each font in its own face and 14 point bold.

```
import java.applet.*;
import java.awt.*;


public class FancyFontList extends Applet {

  private String[] availableFonts;

  public void init () {

    Toolkit t = Toolkit.getDefaultToolkit();
    availableFonts = t.getFontList();

  }
```

```
  public void paint(Graphics g) {

    for (int i = 0; i < availableFonts.length; i++) {
      Font f = new Font(availableFonts[i], Font.BOLD, 14);
      g.setFont(f);
      g.drawString(availableFonts[i], 5, 15*i + 15);
    }
  }

}
```

FontMetrics

No word wrapping is done when you draw a string in an applet, even if you embed newlines in the string with `\n`. If you expect that a string may not fit in the applet, you should probably use a TextArea Component instead. You'll learn about text areas and other AWT Components next class. However there are times when you will need to concern yourself with how much space a particular string will occupy. You find this out with a `FontMetrics` object. `FontMetrics` allow you to determine the height, width or other useful characteristics of a particular string, character, or array of characters in a particular font.

As an example the following program expands on the DrawString applet. Previously text would run off the side of the page if the string was too long to fit in the applet. Now the string will wrap around if necessary.

In order to tell where and whether to wrap the `String`, you need to measure the string, not its length in characters which can be variable but rather its width and height in pixels. Measurements of this sort on strings clearly depend on the font that's used to draw the string. All other things being equal a 14 point string will be wider than the same string in 12 or 10 point type.

To measure character and string sizes you need to look at the `FontMetrics` of the current font. To get a `FontMetrics` object for the current `Graphics` object you use the `java.awt.Graphics.getFontMetrics()` method. From `java.awt.FontMetrics` you'll need `fm.stringWidth(String s)` to return the width of a string in a particular font, and `fm.getLeading()` to get the appropriate line spacing for the font. There are many more methods in `java.awt.FontMetrics` that let you measure the heights and widths of specific characters as well as ascenders, descenders and more, but these three methods will be sufficient for this program.

Finally you'll need the `StringTokenizer` class from `java.util` to split up the `String` into individual words. However you do need to be careful lest some annoying beta tester (or, worse yet, end user) tries to see what happens when they feed the word antidisestablishmentarianism or supercalifragilisticexpialidocious into an applet that's 50 pixels across.

```
import java.applet.*;
import java.awt.*;
import java.util.*;
```

```
public class WrapTextApplet extends Applet {

  private String inputFromPage;

  public void init() {
    this.inputFromPage = this.getParameter("Text");
  }

  public void paint(Graphics g) {

    int line = 1;
    int linewidth = 0;
    int margin = 5;
    StringBuffer sb = new StringBuffer();
    FontMetrics fm = g.getFontMetrics();
    StringTokenizer st = new StringTokenizer(inputFromPage);

    while (st.hasMoreTokens()) {
      String nextword = st.nextToken();
      if (fm.stringWidth(sb.toString() + nextword) + margin <
        this.getSize().width) {
        sb.append(nextword);
        sb.append(' ');
      }
      else if (sb.length() == 0) {
        g.drawString(nextword, margin, line*fm.getHeight());
        line++;
      }
      else {
        g.drawString(sb.toString(), margin, line*fm.getHeight());
        sb = new StringBuffer(nextword + " ");
        line++;
      }

    }
    if (sb.length() > 0) {
      g.drawString(sb.toString(), margin, line*fm.getHeight());
      line++;
    }

  }

}
```

Exercises

1. Write an applet that randomly places a designer specified number of rectangles of random sizes and colors at least partially inside the applet's visible area. Be sure to intelligently

handle the case where the designer does not provide proper initialization parameters for the applet.

Make the applet available on a Web page. The Web page should thoroughly test the capabilities of the applet. Include a link to full source code for the applet. Hand in the source code for the applet, sample HTML files for the applet, screenshots of the running applet, and a URL where the applet can be viewed.

2. Write an applet that draws a graph of p(n) vs. n where p(n) is the nth iteration of the logistic equation (p(i) = r*p(i-1)*(1.0-p(i-1)). (You only need to draw the graph. You do not need to draw axes or legends or anything of that nature.) Use `PARAM` tags to specify the rate, the initial population, the x and y scale factors and the number of iterations to track. Note that it is not necessary to achieve convergence. Thus this problem is a little different from previous ones involving the logistic equation.

   Make the applet available on a Web page. Include at least three instances of the applet on the page: one with a rate between 1 and 2, one with a rate between 2 and 3, and one with a rate between 3 and 4. Include a link to full source code for the applet. Hand in the source code for the applet, sample HTML files for the applet, screenshots of the running applet, and a URL where the applet can be viewed.

3. Write an applet that reads an indefinite number of strings from `PARAM` tags and draws them.
4. Allow the Web page designer to specify the font face, size, and style of each string in the applet from problem 2. Be sure to handle the case of the font not being available on the client.
5. Also let the designer pick the color of each string.
6. Finally, let the Web page designer specify the position of each string. Thus you should hand in an applet that allows the designer to specify the font, size, style, color, and position of an indefinite number of strings.
7. Make the applet available on a Web page. The Web page should thoroughly test the capabilities of the applet. Include a link to full source code for the applet. Hand in the source code for the applet, sample HTML files for the applet, screenshots of the running applet, and a URL where the applet can be viewed.