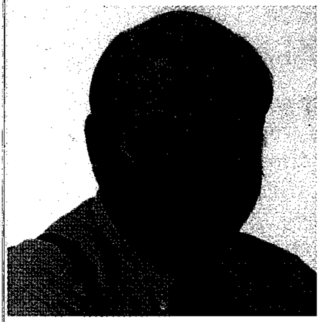
OS/2 Developer

The World of Objects



Your Dog on Java. Any Questions?

BY ROGER SESSIONS

It's hard not to like dogs. You come home after a hard day. You turn on your computer. You type "dog". You always get the cutest little "woof woof".

The only problem I ever have with my dog is when I'm forced to use somebody else's computer. Then when I type "dog", instead of seeing my expected greeting I get:

```
SYS1041: The name specified is
  not recognized as an internal
  or external command, oper-
  able program or batch file.
```

I have thought of placing my *dog.exe* on my personal Web page. I would know then, that, wherever I am, on whatever computer I am using, and whenever I am feeling stressed out I can start up a Web server, download *dog.exe*, and get my reassuring little "woof woof".

This setup would work in those lucky situations that contain the following conditions:

- I am using an IBM compatible machine.
- I am using an OS/2 system instead of a you-know-what system.
- All of the DLLs my dog depends on have been installed.

Wouldn't it be nice if I could download my dog to *any* machine, while running *any* operating system, and know that I could count on my dog doing its little doggy thing?

For that matter, as a dog developer, who has invested thousands of person hours in developing my dog application, wouldn't it be nice to know I could sell it to dog lovers everywhere? And that I wouldn't have to have one version for OS/2, another version for MVS, another version for Macintosh, and so on?

Smell the coffee

The ability to create a single executable and run it on any machine is the most important feature of a hot new programming language called Java. Like any self-respecting program developed in the last 10 years, Java is object-oriented, which makes it fair game for this column. Unlike other recently developed programming languages, Java has received unprecedented interest within the programming community, even before it was released as an official product. If you start getting interested in Java, there are a few web sites you might want to check out. These sites are described in Table 1.

Most of Java's publicity has focused on its ability to run programs within Internet Web browsers; it is within this context that most Java code is currently being written. However, over the long haul, it's the system independence of the program executable that may well prove its most historic feature.

On December 6th of last year, IBM announced that it had licensed Java, and that it "...intends to port Java technology to its OS/2 and AIX operating systems, as well as Microsoft Windows 3.1, and it will make those ports available over the World Wide Web..."

IBM has already fulfilled at least part of this availability promise. At press time for this column, you can set your Web pointer to *http://ncc.hursley.ibm.com/javainfo/*, register as a Java developer, and download the entire beta Java system.

If you do, make sure you install Java in an HPFS partition. Java files use very long path names, and have no option for standard DOS-style 8.3 names.

Let's take a look at some actual Java code and get a feeling for the language.

We will start with an animal interface that defines the **eat** and **setFood** methods. From it, we derive a dog interface that adds on the bark method. The animal interface is defined in the file *animal.java*. The dog interface is defined in *dog.java* (Listing 1).

An interface describes what methods an object supports, but not how those behaviors are implemented. In Listing 1, the first line of both *animal.java* and *dog.java* uses the keyword **interface** to indicate that we are defining interfaces, not implementations. These interface files are used by clients to understand what behaviors are supported by their objects.

An interface can be derived from

Table 1. Java-related Web sites.

http://ncc.hursley.ibm.com/javainfo — The official IBM Java home page.
http://java.sun.com — The official Sun Java Home page.
http://sunsite.unc.edu/javafaq/javafaq.html — A great collection of Frequently Asked Questions about Java.
http://www.gamelan.com — Earthweb's extensive links to Java applets.

Listing 1: Interface files.

animal.java:

```
interface animal {
  public void setfood(String myfood);
  public void eat();
}
```

dog.java:

```
interface dog
extends animal{
  public void bark();
}
```

another interface, as is our dog from our animal in Listing 1. This fact is indicated by use of the keyword extends. We say dog extends animal, meaning that dog will support all the animal behaviors in addition to its own.

In Java, a class is one of many possible implementations of an interface. Listing 2 gives the **animal_implementation** class, an implementation of the animal interface, and **dog_implementation**, **littleDog_implementation**, and **bigDog_implementation** classes—three implementations of the dog interface. We use the keyword **class** to indicate we are showing an implementation, not an interface.

Implementing an interface means to provide implementations of each of the methods defined in a particular interface. Our **dog_implementation** implements our dog interface, which, remember, includes the animal interface. We use the keyword, **implements**, to indicate that a particular class is implementing a particular interface.

Extending a class means deriving a new class from an existing class. The derived class can either add new methods or override existing ones. As shown in Listing 2, our **dog_implementation** extends the **animal_implementation** by adding a bark implementation to the **setFood** and **eat** implementations provided by **animal_implementation**. Our **littleDog_implementation** and **bigDog_implementation** extends the **dog_implementation** by overriding the bark method.

The use of interfaces is optional in Java. Most writers either don't describe it at all or describe it as a poor man's version of multiple inheritance. I think this is unfortunate. The concept of interface descriptions, independent of implementation, is a useful concept. Describing all public classes with well-documented interfaces and distributing these as source files, while distributing implementations only as binary files, would be very natural.

Our client code is shown in Listing 3. In C++, client code is not typically a class. In Java, everything is a class (except for interfaces, which aren't really things at all, but only definitions of things). You run a class.

As you can see in Listing 3, the class test defines one public static method called main. It's defined as public so that it can be invoked publicly. It's defined as static, because it can be invoked without an actual object of the class being instantiated. The name of the method is main, which is the method that is run by default when running a class.

The method **test::main** declares four objects. Pooh is an animal, while Snoopy, Lassie, and Toto are all dogs. Keep in mind that animals are just interfaces. We won't discover what actual implementations will be used until object instantiation time

The instantiations use the **new** directive, as shown in the client code. Through use of this directive, Pooh becomes an **animal_implementation**, a class which supports the animal interface. Snoopy, Lassie, and Toto become

Listing 2: Implementation program.

animal_implementation.java:

```
class animal_implementation
implements animal {
  String food;
  public void setfood(String myfood) {
    food = myfood;
  }
  public void eat() {
    System.out.println(food + " was very nice");
  }
}
```

dog_implementation.java:

```
class dog_implementation
extends animal_implementation
implements dog {
  public void bark() {
    System.out.println("Unknown dog noise");
  }
}
```

littleDog_implementation.java:

```
class littleDog_implementation extends dog_implementation {
  public void bark() {
    System.out.println("woof woof");
  }
}
```

bigDog_implementation.java:

```
class bigDog_implementation extends dog_implementation {
  public void bark() {
    System.out.println("Woof Woof");
    System.out.println("Woof Woof");
  }
}
```

dog_implementation, **bigDog_implementation**, and **littleDog_implementations**, respectively—all classes that support the dog interface.

Java has simplified the coding process with two decisions. The first is lack of support for pointers. The second is automatic memory management.

You can see the impact of both of these decisions in the implementation of the **setFood** method shown in Listing 2. This implementation requires only a single line of code:

```
food = myfood;
```

In C++, this code looks like:

```
if (!food) free(food);
food = (string)
        malloc(strlen
        (myfood)+1);
strcpy(food, myfood);
```

Two out of the three lines of the C++ version are dealing with memory management. Not only is most of the code doing memory management, but it's by far the most complicated code. The ability of Java to automatically manage memory is a considerable simplification.

Not only is the Java code simpler, it's less fragile. The Java version will work fine even if the food variable had never been initialized. The C++ code will fail on any system where food had not been initialized to NULL.

Java is heavily influenced by C++. The main syntax of declaring variables, using operations, and invoking methods is the same in both languages. The major differences, many of which we have already discussed, include the following:

- addition of the concept of interface
- addition of automatic memory management
- lack of support for pointers
- lack of support for multiple inheritance in class definitions, though it is supported in interface definitions
- lack of support for structures, which were considered redundant with classes.

Listing 3: Test program.

```
class test {
  public static void main (String args[]) {
    animal pooh;
    dog snoopy, toto, lassie;

    pooh = new animal_implementation();
    pooh.setfood("honey");
    pooh.eat();
    System.out.println(" ");

    snoopy = new dog_implementation();
    snoopy.setfood("dog food");
    snoopy.eat();
    snoopy.bark();
    System.out.println(" ");

    toto = new littleDog_implementation();
    toto.setfood("little dog food");
    toto.eat();
    toto.bark();
    System.out.println(" ");

    lassie = new bigDog_implementation();
    lassie.setfood("big dog food");
    lassie.eat();
    lassie.bark();
    System.out.println(" ");

  }
}
```

Program output:

```
honey was very nice

dog food was very nice
Unknown dog noise

little dog food was very nice
woof woof

big dog food was very nice
Woof Woof
Woof Woof
```

Similar to C++, Java has full support for polymorphism. (If you are unfamiliar with polymorphism, see "littleDogs, Polymorphism, and Frameworks," February 1996, p. 46.) You can see the polymorphic resolution of the bark method in the client code output shown in Listing 3, where the three different dogs bark differently based on their implementation of bark.

Java is heavily influenced by the Internet. The mindset of Java is to develop programs that, in their executable form, can be downloaded and run on any machine. In other words, Java seeks to create a universal executable.

Java accomplishes this universal executable by running all Java executables through an interpreter. The executables are, in fact, compiled versions of the source code, but they are compiled for the interpreter.

This concept is different than our traditional notion of interpreters. Most interpreters run source code directly. This method achieves universality but at a considerable performance penalty. The Java interpreter runs binary code that has been heavily optimized by the Java compiler.

Unlike most binary code, the binary code produced by the Java compiler is universally standardized. It's exactly the same whether compiled on OS/2 or on Windows 95, because it's produced according to the standards set by the Java Interpreter, not the standards set by an operating system. It's the Java Interpreter that is responsible for getting the binary to run on a particular operating system.

Thus, we get both universality and performance. We get universality because the Java interpreter is responsible for running a program, and any machine with a Java interpreter can run any Java program. We get performance, because the interpreter is running pre-compiled code rather than source code.

We run the Java compiler by typing "javac" and the name of the source file, which must include the extension *.java*. For example, to compile our test class, we type "javac test.java."

The compiler will automatically compile any other files on which this file is dependent. So for our test program, we will automatically compile *animal.java*, *dog.java*, *animal_implementation.java*, and so on.

The compiler generates files with the extension *.class*. In our case, the compile of *test.java* will result in *test.class*, *dog.class*, and so forth.

We can run any of these classes by invoking the Java interpreter and also the name of the class without any extensions. The only class that is meaningful to run is our test class, because it is the only class with a main. We run the test class by typing "java test."

Java is designed to be virus unfriendly. Java programs are intended to be downloaded over the Internet. We are all aware of the danger of downloading viruses. The Java developers wanted to make Java an unlikely carrier for viruses, so two mechanisms are used to protect Java programs.

The first protection mechanism is lack of support for pointers. We have already discussed this in the context of program simplification. Without pointers, writing a program that will wander outside of its address space, and, for example, starts usurping functions of the operating system is very difficult.

The second protection mechanism is through the Java interpreter. Because all Java programs run under control of the interpreter, Java has an opportunity to pre-screen programs before they are allowed to run. Considerable intelligence is being built into the interpreter to ensure that programs don't do things outside the bounds of standard operations, such as update system files.

In summary, lets recap the major Java features that make it attractive to online developers.

- Java is a full object-oriented programming language, supporting the three must-haves of any object-oriented language: encapsulation, inheritance, and polymorphism.
- Java supports separation of interface and implementation, allowing distribution of interface source without distribution of implementation source.
- Java is much easier to program than C++ due to its automatic memory management and lack of support for pointers.
- Java executables are machine-independent and virus-resistant, allowing them to be easily and safely distributed over networks.

So this is your dog on Java: simple, safe, and a dog that will follow you anywhere.

Acknowledgements

The sample code in this article is derived from some code I developed with Simon Nash at IBM. I appreciate his willingness to share his expertise and to let me use this code.

Roger Sessions is president of ObjectWatch Inc., a company specializing in training and consulting in the use of CORBA technologies on IBM platforms. He has spoken at over 30 conferences and has written extensively. His books include Object Persistence: Beyond Object-Oriented Databases, Class Construction in C and C++; Object-Oriented Fundamentals, *and* Reusable Data Structures for C. *Roger also publishes an Internet newsletter called* ObjectWatch on SOM *and can be contacted via e-mail at roger@fc.net.*