Introduction to Java

Topics in this section include:

- Source code and compilation
- Class files and interpretation
- Applications versus applets
- Java language fundamentals
- User-defined data types with Java
- Java syntax

Overview

Java is a modern, evolutionary computing language that combines an elegant language design with powerful features that were previously available primarily in specialty languages. In addition to the core language components, Java software distributions include many powerful, supporting software libraries for tasks such as database, network, and graphical user interface (GUI) programming. In this section, we focus on the core Java language features.

Java is a true object-oriented (OO) programming language. The main implication of this statement is that in order to write programs with Java, you must work within its object-oriented structure.

Object-oriented languages provide a framework for designing programs that represent real-world entities such as cars, employees, insurance policies, and so on. Representing real-world entities with nonobject-oriented languages is difficult because it's necessary to describe entities such as a truck with rather primitive language constructs such as Pascal's `record`, C's `struct`, and others that represent *data only*.

The *behavior* of an entity must be handled separately with language constructs such as procedures and/or functions, hence, the term procedural programming languages. Given this separation, the programmer must manually associate a data structure with the appropriate procedures that operate on, that is, manipulate, the data.

In contrast, object-oriented languages provide a more powerful `class` construct for representing user-defined entities. The `class` construct supports the creation of *user-defined data types*, such as `Employee`, that represent both the data that describes a particular employee and the manipulation or use of that data.

Java programs employ user-defined data types liberally. Designing nontrivial Java classes requires, of course, a good working knowledge of Java syntax. The following sections, illustrate Java syntax and program design in the context of several Java class definitions.

User-defined Data Types

With Java, every computer program must define one or more user-defined data types via the `class` construct. For example, to create a program that behaves like a dog, we can define a class that (minimally) represents a dog:

```
class Dog {
  void bark() {
    System.out.println("Woof.");
  }
}
```

This user-defined data type begins with the keyword `class`, followed by the name for the data type, in this case, `Dog`, followed by the specification of what it is to be a dog between opening and closing curly brackets. This simple example provides no data fields, only the single behavior of barking, as represented by the method `bark()`.

Methods

A `method` is the object-oriented equivalent of a procedure in nonobject-oriented languages. That is, a *method* is a program construct that provides the mechanism (*method*) for performing some act, in this case, barking. Given an instance of some entity, we invoke behavior with a dot syntax that associates an instance with a method in the class definition:

Method Invocation Syntax
`<instance>.<behavior>()`
`<variable> = <instance>.<behavior>(<arguments>...)`

To elicit a bark from a dog `fido`, for example, the operation would be:

```
fido.bark()
```

Syntactically, Java supports passing data to a method and capturing a value returned from a method, neither of which takes place in the previous invocation.

Java is a strongly typed language, meaning that it expects variables, variable values, return types, and so on to match properly, partly because data types are used to distinguish among multiple methods with the same name. Method return types and parameters are specified during definition:

Method Definition Syntax

`void <method-name>(<arguments>...) {`
`  <statements>...`
`}`

`<return-type> <method-name>(<arguments>...) {`
`  <statements>...`
`}`

Historically, the combination of method name, return type, and argument list is called the *method signature*. With modern OO languages, a class may define multiple methods with the same name, as long as they are distinguishable by their signatures; this practice is called *method overloading*. Java has the restriction that return type does not contribute to the method signature, thus, having two methods with the same names and arguments, but different return types is not possible.

For the current example, the return type of `void` indicates that `bark()` does not compute any value for delivery back it to the invoking program component. Also, `bark()` is invoked without any arguments. In object parlance, invoking a method relative to a particular object (a class instance) is often called *message passing*. In this case, the message contains no supplemental data (no arguments).

For now, if we create an instance of `Dog`, it can bark when provoked, but we have no way of representing data, for example, how many times it will bark, its breed, and so on. Before looking at language constructs that will make the `Dog` data type more versatile, we must consider a mechanical aspect of the Java language, namely, what's necessary to run a program.

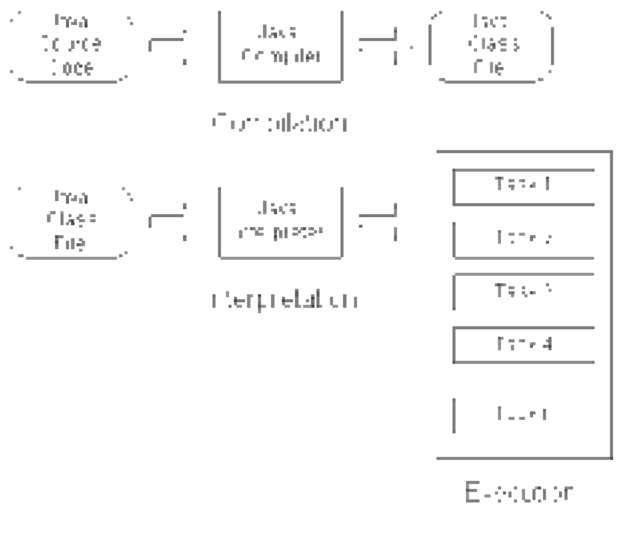
Java Applications

With the Java class and method syntax in hand, we can design a Java program. Java applications consist of one or more classes that define data and behavior. Java applications are translated into a distilled format by a Java compiler. This distilled format is nothing more than a linear sequence of operation-operand(s) tuples:

<operation>	*<operand>*
<operation>	*<operand>*
<operation>	*<operand>*

This stream of data is often called a *bytecode stream*, or simply *Java bytecodes*. The operations in the bytecode stream implement an instruction set for a so-called virtual machine (software-based instruction processor), commonly called a Java virtual machine (JVM). Programs that implement the JVM simply process Java class files, sometimes specific to a particular environment. For example, Java-enabled web browsers such as Netscape Navigator and Internet Explorer include a JVM implementation. Standalone programs that implement the JVM are typically called Java interpreters.

The Java compiler stores this bytecode stream in a so-called class file with the filename extension `.class`. Any Java interpreter can read/process this stream--it "interprets" each operation and its accompanying data (operands). This interpretation phase consists of (1) further translating the distilled Java bytecodes into the machine instructions for the host computer and (2) managing the program's execution. The following diagram illustrates the compilation and execution processes:



Java class files are portable across platforms. Java compilers and interpreters are typically not portable; they are written in a language such as C and compiled to the native machine language for each computer platform. Because Java compilers produce bytecode files that follow a prescribed format and are machine independent, and because any Java interpreter can read and further translate the bytecodes to machine instructions, a Java program will run anywhere--without recompilation.

A class definition such as `Dog` is typically stored in a Java source file with a matching name, in this case, `Dog.java`. A Java compiler processes the source file producing the bytecode class file, in this case, `Dog.class`. In the case of `Dog`, however, this file is not a Java *program*.

A Java program consists of one or more class files, one of which must define a program starting point--`Dog.class` does not. In other words, this starting point is the difference between a class such as `Dog` and a class definition that implements a program. In Java, a program's starting point is defined by a method named `main()`. Likewise, a program must have a well-defined stopping point. In Java, one way to stop a program is by invoking/executing the (system) method `exit()`.

So, before we can do anything exciting, we must have a program that starts and stops cleanly. We can accomplish this with an arbitrary, user-defined data type that provides the `main()` and `exit()` behavior, plus a simple output operation to verify that it actually works:

```
public class SimpleProgram {
  public static void main(String[] args) {
    System.out.println("This is a simple program.");
    System.exit(0);
  }
}
```

The signature for `main()` is invariable; for now, simply define a program entry point following this example--with the modifiers `public` and `static` and the return type `void`. Also, `System` (`java.lang.System`) is a standard class supplied with every Java environment; it defines many utility-type operations. Two examples are illustrated here: (1) displaying data to the standard output device (usually either an IDE window or an operating system command window) and (2) initiating a program exit.

Note that the `0` in the call to `exit()` indicates to the calling program, the Java interpreter, that zero/nothing went wrong; that is, the program is terminating normally, not in an error state.

At this point, we have two class definitions: one, a real-world, user-defined data type `Dog`, and the other, a rather magical class that connects application-specific behavior with the mechanics of starting and stopping a program.

Now is a good time to get acquainted with your Java development environment. If you have an integrated development environment (IDE), it may or may not be file-oriented. With most environments Java source code is stored in a file. One popular exception is IBM's VisualAge for Java, which stores class definitions in a workspace area.

When using an IDE that is file-oriented, note that the filenames and class names must match exactly; in particular, the file and class names are case sensitive. Also, you must work within the rules established for a Java environment with respect to system environment variable settings, and so on. The section Runtime Environments and Class Path Settings includes general information on system settings.

The first magercise is simple but important, because it tests your Java configuration. It includes these basic steps:

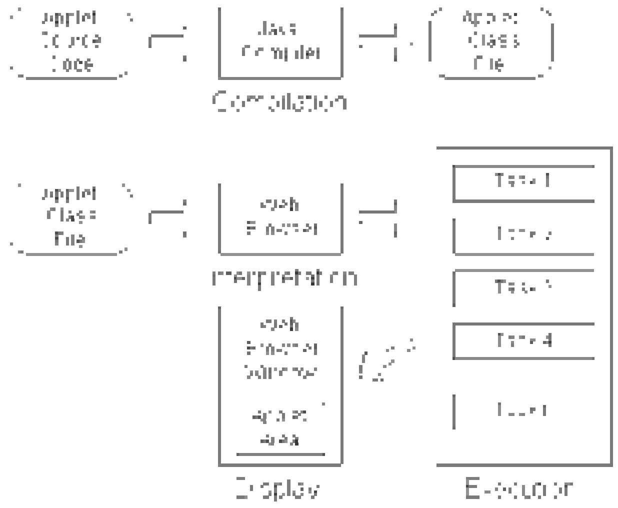
- Write `SimpleProgram` exactly as shown here
- Save it as required by the IDE
- Somewhere in the IDE workspace environment
- Or, in a separate file named `SimpleProgram.java` depending on the IDE
- Build the program
- Execute it
- Observe the output

Magercises occur throughout the course; simply follow the links.

Applications versus Applets

A Java application consists of one or more class files, one of which defines the `main()` method. You can run an application in any environment that provides a Java interpreter, for example, anywhere there's a Java IDE. The Java Runtime Environment (JRE) from Sun provides an interpreter as well, but omits the development-related tools such as the compiler.

A Java applet is not an application; it does not define `main()`. Instead, applets depend on host applications for start-up, windowing, and shut-down operations, typically, a web browser:



Many applets simply render a graphical image in a designated area of the web browser window; others provide a GUI with command buttons that initiate application-specific operations. Applets operate under several security restrictions, which protects users from unknowingly downloading applets that snoop for private data, damage local file systems, and so on.

Applet programming involves many Java concepts that are, by definition, beyond the scope of an introductory section. The final topic in this section includes an introduction to applets (Java Applets).

Java Commenting Syntax

Java supports three types of commenting, illustrated in the following table:

Comment Example	**Description**
`int x; // a comment`	Remainder of line beginning with "//" is a comment area

`/*`
`The variable x is an`
`integer:`
`*/`
`int x;`
	All text between the "/*" and "*/", inclusive, is ignored by compiler

`/**`
`x -- an integer`
`representing the x`
	All text between the "/**" and "*/", inclusive, is ignored by compiler and intended for `javadoc`

`coordinate`
`*/`
`int x;`
	documentation utility

The `javadoc` documentation tool is quite powerful. The standard Java distribution from Sun includes documentation built with `javadoc`; hence, one avenue for learning this tool is to study the HTML documentation alongside the Java source code, which contains the comments that `javadoc` converts into HTML.

Variable Definition and Assignment

Given a user-defined data type such as `Dog`, we would like to create an instance of `Dog` and use it subsequently in a program. Doing so requires both variable definition and assignment operations. A data definition operation specifies a data type and a variable name, and optionally, an initial value:

Data Definition
`<data-type> <variable>;`
`<data-type> <variable-1>, <variable-2>, ..., <variable-n>;`
`<data-type> <variable> = <data-value>;`

The data type may be a primitive, or built-in, type or a user-defined type such as `Dog`. The value may be a literal value or an instance of a user-defined type such as `Dog`. Primitive data types are discussed in Java Data Types.

Several examples of data definitions follow:

Data Definition Examples
`int x;`
`int x = 9;`
`boolean terminate = false;`
`Dog dog = new Dog();`

The `new` operator is described in the next section (Creating Class Instances).

An assignment operation can occur in the following contexts: