Tutorials

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

Dr. Munindra Kumar Singh

Assistant Professor

VBS Purvanchal University, Jaunpur

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.

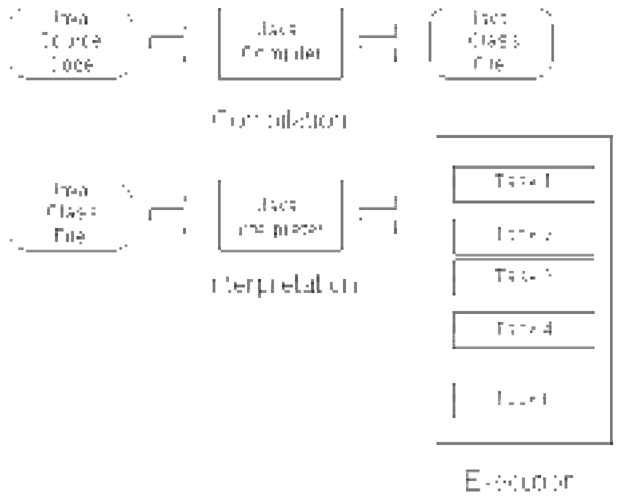
Java Applications

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:

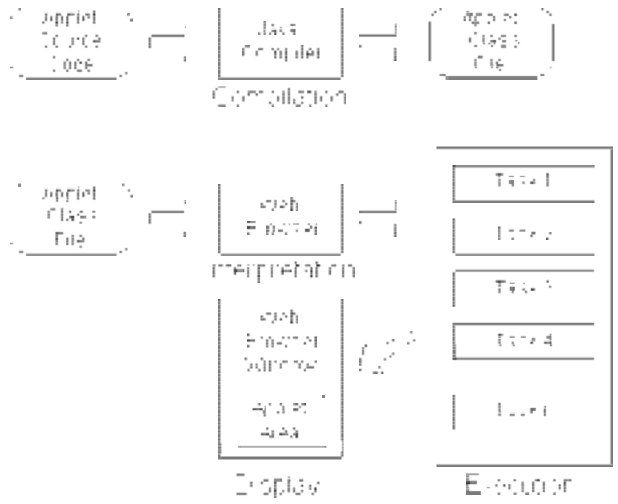
- Write `SimpleProgram` exactly as shown here
- Save it as required by the IDE
- Somewhere in the IDE workspace environment
- Or, in a separate file named `SimpleProgram.java` depending on the IDE
- Build the program
- Execute it
- Observe the output

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:

Assignment Operation
`<data-type> <variable> = <data-value>;`

`<data-type> <variable>;`
`<other-statements>...`
`<variable> = <data-value>;`

The data value to the right of the assignment operator can be a literal value, or any operation that produces a scalar value. Several examples follow:

Assignment Example	**Comment**
`int x = 4;`	Data definition with assignment
`x = 9;`	Assumes prior definition of `x`
`temperature = 21.4;`	Assumes prior definition of `temperature`
`dog = new Dog();`	Assumes prior definition of `dog`

Creating Class Instances

With the capability for starting and stopping a program, plus variable definition and assignment, we can now use the previously developed data type `Dog`. First, we modify `SimpleProgram` to have a more meaningful name, for example, `ADogsLife`:

```
public class ADogsLife {
  public static void main(String[] args) {
    System.exit(0);
  }
}
```

Next, we define the program's behavior in terms of its `main()` method. Specifically, `main()` creates an instance of `Dog` named `dog` (case is significant in Java) and provokes the dog to bark:

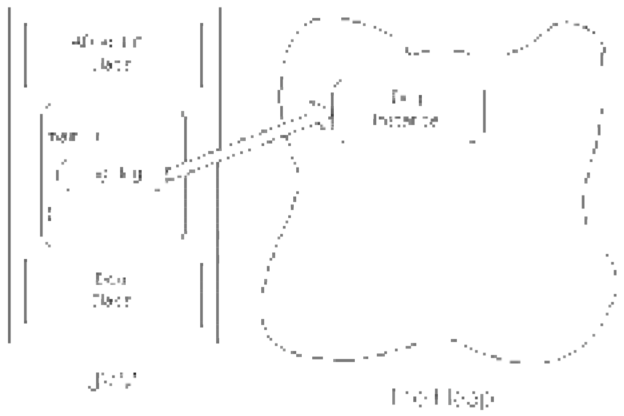
```
public class ADogsLife {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.bark();
    System.exit(0);
  }
}
```

In Java, as with other languages, a program allocates objects dynamically. Java's storage allocation operator is `new`:

Storage Allocation Syntax
`new <data-type>(<arguments>...)`
`<data-type> <variable> = new <data-type>(<arguments>...)`

The `new` operator asks the Java runtime environment to create dynamically (on the fly) an instance of a user-defined data type, for example, "`new Dog()`". You can also associate the instance with a variable for future reference (hence, the term *reference variable*), for example, "`Dog bowwow = new Dog()`". The data type for the reference variable `bowwow` must be specified to the left of the variable name, in this case, "`Dog bowwow`".

Objects receive their storage on/from the heap, which is simply a memory pool area managed by the Java interpreter. The following diagram illustrates the memory allocation for the class files, plus the instance of `Dog` allocated on the heap:



Java Data Types

Java supports a variety of primitive (built-in) data types such as `int` for representing integer data, `float` for representing floating-point values, and others, as well as `class`-defined data types that exist in supporting libraries (Java packages). (All Java primitive data types have lowercase letters.)

The `String` class is defined in `java.lang`, the core package of supplemental class definitions that are fundamental to Java programming. A more complete reference to this class includes the package specification `java.lang.String`. (By convention, class names have mixed-case letters: the first letter of each word is uppercase.)

The Java language has the following primitive types:

Primitive Type	**Description**
`boolean`	`true/false`
`byte`	8 bits
`char`	16 bits (UNICODE)
`short`	16 bits
`int`	32 bits
`long`	64 bits
`float`	32 bits IEEE 754-1985
`double`	64 bits IEEE 754-1985

For an overview of common nonprimitive data types, that is, class definitions provided by the Java environment, see the `java.lang` package within the standard Java distribution documentation, or the documentation supplied with your Java IDE.

Method Overloading

Not all dogs sound alike, however, so to add a little barking variety to the `Dog` implementation, we should define an alternative `bark()` method that accepts a barking sound as a string:

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

  void bark(String barkSound) {
    System.out.println(barkSound);
  }
}
```

This version of `Dog` is legal because even though we have two `bark()` methods, the differing signatures allows the Java interpreter to chose the appropriate method invocation. The method definition syntax "`void bark(String barkSound)`" indicates that this variation of `bark()` accepts an argument of type `String`, referred to within the method as `barkSound`.

As another example of method overloading, consider the program `DogChorus`, which creates two dogs and elicits a different barking behavior for each dog:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    Dog spot = new Dog();
    fido.bark();
    spot.bark("Arf.  Arf.");
    fido.bark("Arf.  Arf.");
    System.exit(0);
  }
}
```

Because `Dog` supports two different barking behaviors, both defined by methods named `bark()`, we can design our program to associate either barking behavior with, say, `fido`. In `DogChorus`, we invoke different barking behaviors for `fido` and `spot`. Note that `fido` changes his bark after hearing `spot`.

Instance Variables

So far, we're defining instances of objects in terms of their behaviors, which in many cases is legitimate, but, in general, user-defined data types incorporate state variables as well. That is, for each instance of `Dog`, it's important to support variability with respect to characteristics such as hair color, weight, and so on. State variables that distinguish one instance of `Dog` from another are called *instance variables*.

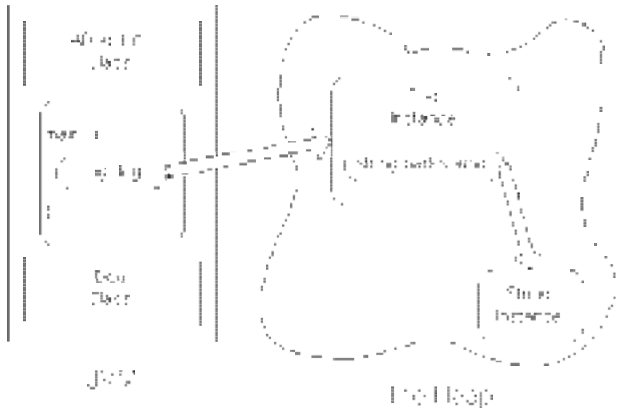
Now suppose we add an instance variable to reflect a particular dog's barking sound; a `String` instance can represent each dog's bark:

```
class Dog {
  String barkSound = new String("Woof.");

  void bark() {
    System.out.println(barkSound);
  }

  void bark(String barkSound) {
    System.out.println(barkSound);
  }
}
```

The definition of `Dog` now includes the instance variable `barkSound`. Each time a new instance of `Dog` is created, that instance will include a reference variable for an instance of `String` representing that particular dog's bark. This *instance* variable is initialized to the default value of `"Woof."`. Consider the line of code

```
String barkSound = new String("Woof.");
```

This statement allocates an instance of `String`, initializes it with the value `"Woof."` (provided in the parentheses following the `String` class name), and stores this data under the reference variable `barkSound`. Note that the reference variable `barkSound` is part of each instance of `Dog`, but that it references an instance of `String` that's allocated on the heap, as is the instance of `Dog`:



Now that the default barking behavior is represented by an instance variable, we can remove the `"Woof."` data from the original `bark()` method, replacing it with a reference to the current value of `barkSound`:

```
void bark() {
  System.out.println(barkSound);
}
```

That is, we've transferred unconditional state data from a method definition into an instance variable that can vary from one dog to the next, and perhaps more importantly, for a particular dog, its value can change dynamically.

Access Methods

In order for the value of an instance variable to vary over time, we must supply a method to change its value; such a method is typically referred to as an *access method*. By convention, a method that's provided simply to affect a change to an instance variable's value begins with the word "set":

```
  void setBark(String barkSound) {
    this.barkSound = barkSound;
  }
```

This method is interesting because it uses two different variables with the same name, `barkSound`. First, the `barkSound` defined as an parameter is the new barking sound. Any unqualified reference to `barkSound` within this method refers to this data passed as an argument. We also have, however, a `barkSound` instance variable for each dog that is instantiated. With Java, we can use the special "instance handle" `this` to refer to the *current* instance of `Dog`. Hence,

```
    this.barkSound = barkSound;
```

replaces the current value of the instance variable (`this.barkSound` with the new value passed as an argument (`barkSound`) to `setBark()`.

To put the `this` variable in perspective, suppose we create an instance of `Dog` referred to as `fido`. Then, if we execute `setBark()` with respect to `fido`, namely,

```
    fido.setBark("Ruff.");
```

the `this` instance in `setBark()` is `fido` and, in particular, `this.barkSound` is the `barkSound` instance variable for the `fido` object.

In the following version of `DogChorus`, we create an object, `fido`, change its barking characteristic from the default `"Woof."` to `"Ruff."`, and then invoke the barking behavior:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    fido.setBark("Ruff.");
    fido.bark();
    System.exit(0);
  }
}
```

With this modification, the characteristics of an object such as `fido` are reflected by both the current values of instance/state variables *and* the available behaviors defined by the methods in `Dog`.

Instance Methods

The type of methods we're designing at present are called *instance methods* because they are invoked relative to a particular instance of a class. For this reason, an instance method can reference an instance variable directly, without the `this` qualifier, as long as there is no variable name conflict, for example,

```
  void bark() {
    System.out.println(barkSound);
  }
```

In this case, the no-argument version of `bark()` references the instance variable `barkSound` directly. As implied by the `setBark()` definition, however, we could also write `bark()` as follows:

```
  void bark() {
    System.out.println(this.barkSound);
  }
```

Here, there are no other variables within (local to) `bark()` named `barkSound`, so these implementations are equivalent.

Conditional Execution

So far, within each method we've used sequential execution only, executing one statement after another. Like other languages, Java provides language constructs for conditional execution, specifically, `if`, `switch`, and the conditional operator `?`.

Conditional Constructs

```
if (<boolean-expression>)
  <statement>...
else
  <statement>...
```

```
switch (<expression>) {
  case <const-expression>:
    <statements>...
    break;

  more-case-statement-break-groups...

  default:
    <statements>...
}
```

```
(<boolean-expression>) ? <if-true-expression>
: <if-false-expression>
```

The more general construct, `if` has the syntax:

```
if (<boolean-expression>)
  <statement>...
```

where <*statement*>... can be one statement, for example,

```
x = 4;
```

or multiple statements grouped within curly brackets (a statement group), for example,

```
{
  x = 4;
  y = 6;
}
```

and <*boolean-expression*> is any expression that evaluates to a `boolean` value, for example,

Boolean Expression	**Interpretation**
`x < 3`	`x` is less than `3`
`x == y`	`x` is equal to `y`
`x >= y`	`x` is greater than or equal to `y`
`x != 10`	`x` is not equal to `10`
<*`variable`*>	*`variable`* is `true`

If the Boolean expression evaluates to `true`, the statement (or statement group) following the `if` clause is executed.

Java also supports an optional `else` clause; the syntax is:

```
if (<boolean-expression>)
  <statements>...
else
  <statements>...
```

If the Boolean expression evaluates to `true`, the statement (or statement group) following the `if` clause is executed; otherwise, the statement (or statement group) following the `else` clause is executed.

Boolean expressions often include one or more Java comparison operators, which are listed in the following table:

Comparison	**Interpretation**

Operator	
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to

Returning to our user-defined type `Dog`, we can add additional state variables for more flexible representation of real-world objects. Suppose we add instance variables `gentle` and `obedienceTrained`, which can be `true` or `false`:

```
class Dog {
  String barkSound = new String("Woof.");
  boolean gentle = true;
  boolean obedienceTrained = false;
  ...
```

In Java, Boolean values are literals (case is significant), and `boolean` variables accept either value. For `gentle` and `obedienceTrained` there are no `new` operators because we're not creating objects--we're creating primitive variables and assigning them the default values `true` and `false`.

Access methods provide the flexibility for modifying these instance variables on a dog-by-dog basis:

```
  void setGentle(boolean gentle) {
    this.gentle = gentle;
  }

  void setObedienceTrained(boolean trained) {
    obedienceTrained = trained;
  }
```

Note that the reference to `obedienceTrained` in `setObedienceTrained()` does not require qualification with `this` because there is no local variable by the same name.

Methods that Return Values

With these new variables, we can provide convenience methods such as

```
boolean isGoodWithChildren() {
  if (gentle == true && obedienceTrained == true)
    return true;
  else
    return false;
}
```

This method introduces additional Java syntax. First, instead of the modifier `void`, `isGoodWithChildren()` provides a return type, in this case, `boolean`. With the replacement of `void` by either a primitive, system-, or user-defined data type, the method must provide a `return` statement for every execution path possible through the method--the Java compiler enforces this "contract."

A return statement has the syntax

```
return <value>;
```

where <*value*> is any expression that evaluates to the appropriate return data type.

For `isGoodWithChildren()`, if the `if` statement's Boolean expression evaluates to `true`, the first `return` executes, which terminates the method execution and returns the result of the evaluation, which in this case is simply the literal value `true`. If the Boolean expression evaluates to `false`, the code block following the `else` clause is evaluated, in this case, a single return statement that returns `false`.

In this example, the Boolean expression is compound--it includes two expressions, each with the `==` comparison operator. The comparative expressions are linked with the *logical and* operator `&&`; hence, the complete expression evaluates to `true` only if both subexpressions evaluate to `true`.

Boolean expressions often include one or more Java logical operators, which are listed in the following table:

Logical Operator	**Interpretation**
`&&`	and ("short-circuit" version)
`&`	and ("full-evaluation" version)

`||`	or ("short-circuit" version)
`|`	or ("full-evaluation" version)

With the "short-circuit" versions, evaluation of subsequent subexpressions is abandoned as soon as a subexpression evaluates to `false` (in the case of `&&`) or `true` (in the case of `||`).

Although `isGoodWithChildren()` provides a full illustration of `if`, including the optional `else` clause, this method can be coded more succinctly. Java, like C, is a syntactically powerful language. First, we can actually remove the `if` because the return values correspond to the Boolean expression, that is, if the Boolean expression evaluates to `true`, return `true`; otherwise, return `false`. The more concise implementation is:

```
boolean isGoodWithChildren() {
  return (gentle == true && obedienceTrained == true);
}
```

One more reduction is possible. Note that each subexpression involves a `boolean` variable compared to the `boolean` literal `true`. In this case, each subexpression can be reduced to the `boolean` variable itself:

```
boolean isGoodWithChildren() {
  return (gentle && obedienceTrained);
}
```

Access Methods Revisited

`Dog` provides `set`-style access methods for modifying an instance variable. At times, it's necessary to retrieve an instance variable's value. Typically, if a class has instance variables that support set operations, they support get operations as well. For each of our set methods, we should code a corresponding get method, for example,

```
boolean getObedienceTrained() {
  return obedienceTrained;
}
```

Note that in the case of `boolean` instance variables such as `obedienceTrained`, some programmers prefer the `is`-style naming convention over the `get`-style and some programmers like to provide both:

```
boolean isObedienceTrained() {
  return obedienceTrained;
  }
```

Note that `isGoodWithChildren()` from the previous section is not really an access method--it does not return (report) an instance variable's value. Instead, it combines higher level, meaningful information with respect to *an instance of* the class `Dog`.

Iterative Execution

Java provides the `while`, `do-while`, and `for` language constructs for iterating over a statement (or statement group) multiple times. `while` is the more general iterative construct; `for` is the more syntactically powerful.

Iterative Constructs

```
while (<boolean-expression>)
  <statements>...
```

```
do
  <statements>...
while (<boolean-expression>)
```

```
for (<init-stmts>...; <boolean-expression>; <exprs>...)
  <statements>...
```

With iteration, we can (to the dismay of our neighbors) make barking behavior repetitive:

```
  void bark(int times) {
    while (times > 0) {
      System.out.println(barkSound);
      times = times - 1;
    }
  }
```

Thus, with yet another `bark()` method, we support the object-oriented task of sending a `Dog` instance the `bark` *message*, accompanied with a *message request* (method argument) for *n* barks, represented in the method definition by the parameter `times`.

`DogChorus` now begins to reflect its name:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    Dog spot = new Dog();
    spot.setBark("Arf.  Arf.");
    fido.bark();
    spot.bark();
    fido.bark(4);
    spot.bark(3);
    new Dog().bark(4); // unknown dog
    System.exit(0);
  }
}
```

`DogChorus` now displays the following:

```
Woof.
Arf.  Arf.
Woof.
Woof.
Woof.
Woof.
Arf.  Arf.
Arf.  Arf.
Arf.  Arf.
Woof.
Woof.
Woof.
Woof.
```

Note the line in the source code with the comment "`// unknown dog`". As we mentioned, Java is a dynamic language, another example of which we illustrate here. An "unnamed" `Dog` is instantiated on the fly (appears suddenly from down the street), joins in the chorus, and then disappears.

That is, with Java we can create an instance of any class on the fly, without assigning it to a reference variable for future use (assuming we need it only once), and use it directly. Furthermore, the Java syntax and order of evaluation for "`new` *`<data-type>`*`()`" is designed so that we can do this without having to group the `new` operation in parentheses.

Java's MultiFunction Operators

We've mentioned that Java is syntactically powerful, like C. Java supports several powerful, multifunction operators described in the following table:

Multifunction Operator	**Interpretation**
`++`	ncrement (by `1`)
`--`	decrement (by `1`)

`+=`	ncrement (by specified value)
`-=`	decrement (by specified value)
`*=`	multiply (by specified value)
`/=`	divide (by specified value)
`&=`	bitwise and (with specified value)
`|=`	bitwise inclusive or (with specified value)
`^=`	bitwise exclusive or (with specified value)
`%=`	nteger remainder (by specified value)

These operators (actually operator combinations) are multifunction operators in the sense that they combine multiple operations: expression evaluation followed by variable assignment. For example, `x++` first evaluates `x`, increments the resulting value by `1`, assigns the result back to `x`, and "produces" the initial value of `x` as the *ultimate evaluation*. In contrast, `++x` first evaluates `x`, increments the resulting value by `1`, assigns the result back to `x`, and produces the updated value of `x` as the ultimate evaluation.

Note that `x++` and `++x` are equivalent in standalone contexts where the only task is to increment a variable by one, that is, contexts where the ultimate evaluation is ignored:

```
int x = 4;
x++; // same effect as ++x
System.out.println("x = " + x);
```

This code produces the output:

```
x = 5
```

In the call to `println()`, the argument is a concatenation of the string `"x = "` and `x` after its conversion to a string. String operations, including the use of `+` for string concatenation, are discussed in the section Strings.

In the following context, the placement of the increment operator is important:

```
int x = 4;
int y = x++;
int z = ++x;
System.out.println(
  "x = " + x + " y = " + y + " z = " + z);
```

This code produces the output:

```
x = 6 y = 4 z = 6
```

The following table includes examples and interpretations:

Multifunction Operator	**Example**	**Pedestrian Equivalent**
`++`	`x++, ++x`	`x = x + 1`
`--`	`x--, --x`	`x = x - 1`
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`&=`	`x &= y`	`x = x & y`
`|=`	`x |= y`	`x = x | y`
`^=`	`x ^= y`	`x = x ^ y`
`%=`	`x %= y`	`x = x % y`

Note that Java restricts bitwise operations with `&`, `|`, and `^` to integer values, which makes sense. Further information on binary operations is available in many introductory computer science texts.

Using the decrement operator we can rewrite the iterative operation in `bark()` somewhat more succinctly as follows:

```
  void bark(int times) {
    while (times > 0) {
      System.out.println(barkSound);
      times--;
    }
  }
```

Further code reduction is possible:

```
  void bark(int times) {
    while (times-- > 0)
      System.out.println(barkSound);
  }
```

In this case, we use the ultimate evaluation of `times` in the `while` construct's Boolean expression that controls iteration (loop continuation). In particular, `times--` decrements the variable but "produces" the initial value of `times` for the expression evaluation proceeding the greater-than comparison operation.

Strings

As mentioned, `String` is a system-defined class--not a primitive--defined in `java.lang`, the core package of supplemental class definitions included with all Java distributions. The `lang` package is considered so essential that no steps are neccessary by the programmer to use its classes. A quick examination of `java.lang.String` shows an extensive number of methods for string manipulation.

The `lang` package also provides a complementary class, `java.lang.StringBuffer`. `String` instances are immutable; that is, they cannot be modified. To use equivalent terminology, `String` operations are *nondestructive*. A programmer simply creates strings, uses them, and when there is no further reference to them, the Java interpreter's garbage collection facility (Java Garbage Collection) recovers the storage space. Most of the string-oriented tasks necessary for normal programming can be accomplished with instances of `String` (which is quite efficient), for example, creating string constants, concatenating strings, and so on.

`StringBuffer`, on the other hand, is more powerful. It includes many methods for destructive string operations, for example, substring and character-level manipulation of strings such as splicing one or more characters into the middle of a string. A good rule of thumb is to use `String` wherever possible, and consider `StringBuffer` only when the functionality provided by `String` is inadequate.

In an earlier section, we performed a common display operation involving strings, namely:

```
System.out.println("x = " + x);
```

This simple line of code demonstrates several string-related issues. First, note that `println()` accepts one argument, which is satisfied by the result of the expression evaluation that includes `+`. In this *context*, `+`performs a string concatenation.

Because `+` is recognized by the Java compiler as a string concatenation operator, the compiler will automatically generate the code to convert any non-`String` operands to `String` instances. In this case, if `x` is an `int` with the value `5`, its value will be converted, generating the string constant `"5"`. The latter is concatenated with `"x = "` producing `"x = 5"`, the single argument to `println()`.

Thus, in the earlier example, we had the code

```
int x = 4;
x++; // same effect as ++x
System.out.println("x = " + x);
```

which produces the output:

```
x = 5
```

Note that you can use this automatic conversion and concatenation anywhere, not just as an argument to a method such as `println()`. This feature is incredibly powerful and convenient, demonstrating once again Java's syntactic power:

```
String waterCoolerGreeting;
if (employee.getAge() > 40) {
  waterCoolerGreeting =
    employee.getFirstName() +
    "!  Wow, what's it like to be " +
    employee.getAge() + "?";
}
else
  waterCoolerGreeting =
    "Hi, " + employee.getFirstName() + "!"
```

Another issue is the use of a double quote-delimited character sequence directly, for example, `"x = "`. Because `String` is a class, the general way to create a string instance is:

```
String prompt = new String("x = ");
```

Note that we have to provide a string in order to create a string! As a convenience for the programmer, Java always recognizes a sequence of characters between double quotes as a string constant; hence, we can use the following short-cut to create a `String` instance and assign it to the reference variable `prompt`:

```
String prompt = "x = ";
String barkSound = "Woof.";
```

One final issue is that automatic conversion to a `String` instance works for objects as well as for integers and other primitives. How? All Java classes are a specialization of the most general Java class `java.lang.Object`, which implies that all classes automatically inherit its `toString()` method. (Class inheritance is beyond the scope of this section, but this one issue is quite interesting and useful now.) During automatic conversions of objects to strings, the Java compiler invokes an object's `toString()` method to do this object-specific conversion.

The `toString()` method inherited from `Object` does very little--a placeholder method really. For every class we design, we can (and should) provide a simple `toString()` method that concatenates together pertinent information for that instance. It is definitely worth the effort to provide `toString()` because it is incredibly useful in debugging operations.

Assuming the existence of several additional instance variables and access methods that distinguish one dog from another, let's define a `toString()` method that returns a collection of descriptive information about an instance of `Dog`:

```
class Dog {
  String barkSound = "Woof.";
  String name = "none";
  String breed = "unknown";
  boolean gentle = true;
  boolean obedienceTrained = false;
  int age = 0;
  ...
  public String toString() {
    return "[name = " + name + "] " +
      "[breed = " + breed + "] " +
      "[age = " + age + "] ";
  }
  ...
```

The test program `TestDogToString` uses the `Dog` instance directly in a display statement:

```
public class TestDogToString {
  public static void main(String[] args) {
    Dog bruno = new Dog();
    bruno.setBark("RRUUFFFF.");
    bruno.setName("Bruno");
    bruno.setBreed("Newfoundland");
    bruno.setAge(14);
    bruno.bark();
    System.out.println(bruno); // automatic conversion
  }
}
```

Running this program displays:

```
RRUUFFFF.
[name = Bruno] [breed = Newfoundland] [age = 14]
```

For now, note that the `toString()` method must have the `public` modifier. Java is a powerful language and many of its features such as inheritance, data and method accessibility, and others are discussed elsewhere. The `toString()` functionality is convenient and warrants early coverage, despite these unaddressed issues. It's a very good idea to include a `toString()` method for every user-defined data type.

Reference Variable Usage

It's common to use the term *reference variable* for any variable that holds a reference to dynamically allocated storage for a class instance, for example, `fido` in the following code:

```
Dog fido = new Dog();
```

In reality, all variables in high-level languages provide a *symbolic reference* to a low-level data storage area. Consider the following code:

```
int x;
Dog fido;
```

Each of these variables represents a data storage area that can hold one scalar value. Using `x` we can store an integer value such as `5` for subsequent retrieval (reference). Using `fido` we can store a data value that is the low-level address (in memory) of a dynamically allocated instance of a user-defined data type. The critical point is that, in both cases, the variable "holds" a scalar value.

In both cases, we can use an assignment operation to store a data value:

```
int x = 5;              // 1.
int y = x;              // 2. x's value also stored in y
Dog fido = new Dog();   // 3.
Dog myDog = fido;       // 4. fido's value also stored
                        //    in myDog
Dog spot = null;        // 5.
```

In the second line, `y` is initialized with the current value of `x`. In the fourth line, `myDog` is initialized with the current value of `fido`. Note, however, that the value in `fido` is not the instance of `Dog`; it is the Java interpreter's "recollection" of where (in memory) it stored the instance of `Dog`. Thus, we can use either reference variable to access this one instance of `Dog`.

With objects, the *context* in which we use the variable determines whether it simply evaluates to the memory address of an object or actually initiates more powerful operations. When the usage involves *dot notation*, for example, `fido.bark()`, the evaluation includes binding the object to the appropriate method from the class definition, that is, invoking a method and performing the implied actions. But, when the usage is something like "`... = fido;`", the evaluation is simply the address.

Consider the expression evaluations that take place within the parentheses in the following code:

```
String sound = "Woof."; // 1.
fido.bark(sound);       // 2. void bark(String barkSound) {...}
int numberBarks = 4;    // 3.
fido.bark(numberBarks); // 4. void bark(int times) {...}
```

In the first line, the `String` instance `"Woof."` is dynamically allocated in storage and its location/address stored in `sound`. In the second line, the evaluation of the argument to `bark()` is simply the scalar value (memory address) stored in the `sound` reference variable because it is more logical to pass along a scalar value than to make another copy of the string instance. That is, the argument is a *copy* of the scalar value held in `sound`.

In the fourth line, the evaluation of the argument to `bark()` is simply the scalar value stored in `numberBarks`. In both cases, the data passed as arguments agree in type with the respective parameters in the method definitions. And, in both cases, the method invocation involves making a copy of the value and passing the copy forward.

In the latter case, this process is generally called *call by value* because the invoked method receives a copy of the *ultimate value* (`4`) from the `int` variable `numberBarks`. When the argument and parameter types are nonprimitive (a defined class), this process is generally called *call by reference* because the invoked method receives a copy of a reference value.

Consider the implication for how the parameters are used in the invoked methods. In the latter case, the `int` parameter `times` is actually modified (decremented) in the method:

```
  void bark(int times) {
    while (times-- > 0)
      System.out.println(barkSound);
   }
```

This modification does not, of course, affect `numberBarks`, which exists in a different context (in the invoking method `main()`), because this method receives a *copy* of the value in `numberBarks`.

In the former case, the `String` parameter `barkSound` is evaluated as the argument to `println()`, but because it is a reference variable, the scalar value is, once again, copied and passed forward in the method invocation chain:

```
  void bark(String barkSound) {
    System.out.println(barkSound);
  }
```

This evaluation of a reference variable is consistent with a previous example from the section Strings:

```
Dog bruno = new Dog();
...
System.out.println(bruno);
```

In this case, the expression evaluation for the argument to `println()` is a reference variable alone (no dot notation), so its scalar value is copied and passed alone. In both cases, the context, that is, the ultimate evaluation within `println()`, requires automatic conversion to a string for display. Ultimately, within one of the `println()` methods (actually, after yet another round of invocations in the case of `bruno`), dot notation is finally applied to the object, in effect, "*`<reference-variable>`*`.toString()`".

There is one important ramification of call-by-reference argument passing. If a method received a reference to an object, it can *potentially* modify the state of that object:

```
class Person {
  ...
  void walkDog(Dog dog) {
    if (dog.barksAtEverything() && dog.tugsAtLeash())
      dog.setGentle(false);
  }
  ...
}
```

Therefore, in designing classes the burden is on the class designer to control which state variables can be modified, and by whom. Instance variable and instance method accessibility issues are discussed elsewhere.

Default Variable Initializations

In Java, variable initialization depends on context. Consider the following:

```
int x;
Dog fido;
```

If `x` and `fido` are instance variables, they are automatically initialized to `0` and `null`, respectively. `null` is a special literal that can be assigned to any reference variable.

In general, if there is no explicit initialization for an instance variable definition, Java automatically initializes the variable to a "zero-like" value, depending on the data type:

Data Type	**Default Initialization Value**
`boolean`	`false`
`byte`	`0`
`char`	`\u0000`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`
`<user-defined-type>`	`null`

Consider the following program:

```
public class TestVariableInit {
  public static void main(String[] args) {
    TestClass tc = new TestClass();
    System.out.println("tc.iStr = " + tc.iStr);
    System.out.println("tc.i = " + tc.i);
  }
}
class TestClass {
  int i; // instance variable
  String iStr; // instance variable
}
```

`TestVariableInit` produces the following output:

```
D:\>java TestVariableInit
```

```
tc.iStr = null
tc.i = 0
```

Several books state that this default initialization applies in all contexts, for example, with variables local to a method (local variables), and that in the case of uninitialized local variables the compiler will generate a warning, or possibly an error, depending on the Java environment.

The fact is that, historically, several Java environments have performed no initialization for local variables and have given no warning or error, in which case the value is garbage. This situation can produce very hard to diagnose runtime errors.

In the case of the Java Development Kit (JDK) from Sun Microsystems, the compiler reports uninitialized local variables as an error--at the point of their reference. Suppose we modify `main()` in `TestVariableInit` as follows:

```
  public static void main(String[] args) {
    TestClass tc = new TestClass();
    String lStr; // local variable
    int i; // local variable
    System.out.println("tc.iStr = " + tc.iStr);
    System.out.println("tc.i = " + tc.i);
    System.out.println("lStr = " + lStr);
    System.out.println("i = " + i);
  }
```

Attempting to compile `TestVariableInit` produces the following output:

```
D:\>javac TestVariableInit.java
TestVariableInit.java:8: Variable lStr may not
have been initialized.
    System.out.println("lStr = " + lStr);
                                   ^
TestVariableInit.java:9: Variable i may not
have been initialized.
    System.out.println("i = " + i);
                                ^
2 errors
```

In this type of situation, the idea that the Java environment guarantees a default value, but simply doesn't allow you to use it, is rather ridiculous. (If a tree falls in the forest, does it make a sound if no one is there to hear it?)

Most programmers would agree that instance variables should be initialized for the sake of readability, and local variables must be initialized for the sake of programmers' sanity everywhere, that is, for wherever and with whatever environment the source code might be compiled today, tomorrow, and after no one remembers who wrote the code.

Arrays

Java provides several classes for managing sets or collections of data, for example, `Vector` (see `java.util.Vector`). And, of course, you can design your own classes.

In addition, Java supports arrays. A Java array is different from a user-defined container object such as a `Vector` instance in the sense that Java provides built-in, language-level syntactic support for arrays, as do many other languages. Although language-level support for arrays increases the complexity of the language definition, it's justified (in the minds of most programmers) because array usage is entrenched in traditional programming.

An array is a linear collection of data elements, each element directly accessible via its index. The first element has index `0`; the last element has index *`n`* `- 1`. It has the form:

Generic Array Object
elements
element type
element 0
element 1
...
element n - 1

The syntax for creating an array object is:

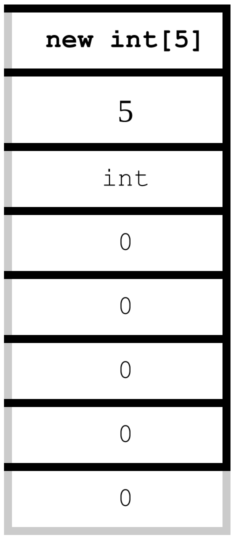
Array Definition
`<data-type>``[]` *`<variable-name>`*`;`

This declaration defines the array object--it does *not* allocate memory for the array object, *nor* does it allocate the elements of the array. Also, you may *not* specify a size within the square brackets.

To allocate an array, use the `new` operator:

```
int[] x = new int[5]; // array of five elements
```

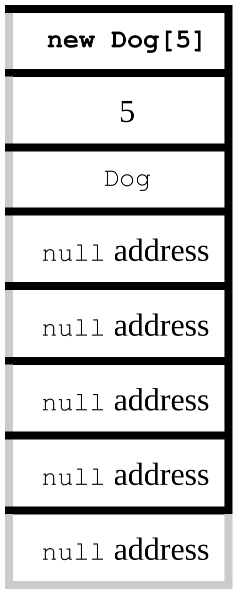
The array `x` of Java primitives has the form:



Consider an array definition for a user-defined type such as `Dog`:

```
Dog[] dog = new Dog[5];
```

This definition creates the array object itself, but not the elements:



Subsequently, you can use the `new` operator to create objects in order to initialize the array elements (which are reference variables):

```
dog[0] = new Dog();
...
dog[4] = new Dog();
```

To create multidimensional arrays, simply create arrays of arrays, for example,

```
T[][] t = new T[10][5];
```

This definition creates ten arrays of references to arrays of references for objects of type `T`. This definition does not allocate memory for instances of `T`.

There is a short-hand notation for defining an array and initializing its elements in one step using comma-separated data values between curly brackets:

Array Definition and Initialization

<data-type>`[]` *<variable-name>* `= {`
  <expression>`,` *<expression>*`, ...`
`};`

The following table provides examples:

Examples of Array Definition and Initialization

`int x = 4;`
`int[] anArray = {3, x, 9, 2};`

`String[] seasons = {"winter", "spring", "summer", "fall"};`

Note that the array size is determined from the number of initializers.

Accessing an undefined array element causes a runtime exception called `ArrayIndexOutOfBoundsException`. Accessing a defined array element that has not yet been assigned to an object results in a runtime exception called `NullPointerException`.

Arrays are useful for enhancing the versatility of our user-defined type `Dog`. Suppose we add an array variable to store a dog's daily diet:

```
class Dog {
  String[] dailyDiet = null;
  String barkSound = "Woof.";
  String name = "none";
  String breed = "unknown";
  ...
```

`dailyDiet` is initialized to `null`, suggesting that there is no legitimate default value. That is, the class definition assumes that an access method will initialize this field, and methods that use this variable should handle a `null` value gracefully.

Next, we provide access methods for setting and getting the diet:

```
  void setDiet(String[] diet)  {
    dailyDiet = new String[diet.length];
    for (int i = 0; i < diet.length; i++)
      dailyDiet[i] = diet[i];
  }

  String[] getDiet() {
    return dailyDiet;
  }
```

`setDiet()` creates an (instance variable) array `dailyDiet` from the array passed as an argument, represented by the parameter `diet`. `setDiet()` uses the `length` variable of the parameter `diet` (available for all arrays) to determine the number of elements. This value appears directly within the "`[]`" brackets following the `new` operator to allocate the required number of elements, each of which can hold a `String` reference variable.

In this situation, we use the `for` construct to iterate over the array. The `for` construct has the syntax:

```
for (<init-stmts>...; <boolean-expression>; <exprs>...)
  <statements>...
```

The iteration control area for the `for` statement has three semicolon-separated components. The first component *<init-stmts>*... is one or more comma-separated initializations, performed once prior to the first iteration. The third component *<exprs>*... is one or more comma-separated expressions, performed after each iteration cycle. The second component is the iteration test condition. As with a `while` statement, this test is performed before each iteration cycle.

The `for` loop control area initializes one index variable `i` to `0`. After each iteration its value is incremented so that it indexes the next element in the array. In the statement group area (the loop body), we copy each reference variable array element from the parameter array to the instance variable array.

The following method demonstrates that the `Dog` class definition must handle `null` values for `dailyDiet` gracefully:

```
  void displayDiet() {
    if (dailyDiet == null) {
      System.out.println(
        "No diet established for " + getName() + ".");
      return;
    }
    else {
      System.out.println(
        "The diet established for " + getName() + " is:");
      for (int i = 0; i < dailyDiet.length; i++)
        System.out.println(dailyDiet[i]);
    }
  }
```

The following program creates a `Dog` instance, establishes its daily diet, and displays the diet:

```
public class DogDiet {
  public static void main(String[] args) {
    Dog fido = new Dog();
    fido.setName("Fido");
    String[] diet = {
      "2 quarts dry food",
      "1 can meat",
      "2 buckets fresh water"
    };
    fido.setDiet(diet);
    fido.displayDiet();
  }
}
```

`DogDiet` produces the following output:

```
D:\>java DogDiet
The diet established for Fido is:
2 quarts dry food
1 can meat
2 buckets fresh water
```

Equality

There is a difference between comparing primitives for equality and comparing two objects for equality. If the value `5` is stored in two different `int` variables, a comparison of the variables for equality will produce the `boolean` value `true`:

```
public class TestIntComparison {
  public static void main(String[] args) {
  int x = 5, y = 5;
  System.out.println(
    "x == y yields " +
    (x == y));
  }
}
```

`TestIntComparison` produces the following output:

```
D:\>java TestIntComparison
x == y yields true
```

The equality operator for primitives compares the values.

On the other hand, the equality operator for objects compares the references not the content of the objects. It asks, "do these references refer to the same object?" Consider a trivial version of `Dog` for illustration purposes that only has a tag number and an age.

```
class Dog {
  int tag;
  int age;
  public void setTag(int t) {tag=t;}
  public void setAge(int a) {age=a;}
}
```

If you have two dogs, even if they have the exact same content, they are not equal using the `==` operator. In the following code fragment, the output will show that `a` and `b` are not equal according to this operator.

```
Dog a = new Dog();
a.setTag(23129);
a.setAge(7);
Dog b = new Dog();
b.setTag(23129);
b.setAge(7);
if ( a==b ) {
  System.out.println("a is equal to b");
}
else {
 System.out.println("a is not equal to b");
}
```

So, how do you compare two objects by value not by reference? Java has a convention that says method `equals()` defines object value equality. There is a definition of `equals()` in class `Object` that is used by default if you do not override it in a subclass. To compare values of dogs `a` and `b`, you would rewrite the comparison above as:

```
if ( a.equals(b) ) {
  System.out.println("a is equals() to b");
}
else {
 System.out.println("a is not equals() to b");
}
```

In this case, the two dogs will still be found equal unless you override `equals()` in `Dog` because `Object.equals()` mimics the `==` operator functionality. The definition of `equals()` in `Dog` is fairly straightforward:

```
class Dog {
  int tag;
  int age;
  public void setTag(int t) {tag=t;}
  public void setAge(int a) {age=a;}
  public boolean equals(Object o) {
    Dog d = (Dog)o;
    if ( tag==d.tag && age==d.age ) {
      return true;
    }
    return false;
  }
}
```

Why does `equals()` define the argument type as `Object` instead of `Dog`? Because you are overriding the definition of `equals()` from the superclass, `Object`, you must repeat the same signature. You expect the argument coming in to be another `Dog`, so you cast the argument to a `Dog` in order to access its fields.

However, since `equals()` is defined in `Dog`, you should check to see that the incoming object is in fact a `Dog` because somebody could have said:

```
fido.equals("blort");
```

The `"blort"` string is a kind of `Object` and, hence, would match the `equals()` signature in `Dog`. A correct version of `equals()` is:

```
public boolean equals(Object o) {
  if ( o instanceof Dog ) {
    Dog d = (Dog)o;
    if ( tag==d.tag && age==d.age ) {
      return true;
    }
  }
  // false if not Dog or contents mismatched
  return false;
}
```

The `instanceof` operator asks whether or not `o` is a kind of `Dog` (which includes subclasses of `Dog`).

Comparison of strings introduces a final wrinkle to comparing objects. Is

```
"abc"=="def"
```

true or false? This is false because they are physically different objects (obvious because they have different content). However, is the following expression

```
"abc"=="abc"
```

true or false? Unfortunately, it depends on the compiler. The compiler is free to optimize those two references to `"abc"` into one object instead of two objects, in which case the expression is true. However, it does not have to perform this optimization--the expression could be false!

Unless you really want to determine if two strings are physically the same object, always use the `equals()` method:

```
boolean b = "abc".equals("def"); // false
boolean c = "abc".equals("abc"); // true
```

Expressiveness

We mentioned that Java is a syntactically powerful language. As an example of Java's expressiveness, as well as Java's expression evaluation order, this section demonstrates how to design a class that supports "chained method evaluation."

Java evaluates expressions from left to right, subject to the standard operator precedence rules. Of course, evaluation order can be controlled with groups of parentheses, as with other languages:

```
int x = (4 + 32) / (2 + 1); // x == 12
```

We mentioned that the `new` operator has a convenient left-to-right evaluation precedence that facilitates one-shot creation of objects, followed by a method invocation to perform some operation:

```
new Dog().bark();
```

The expression evaluation proceeds as follows:

1 `new Dog()` yields an *unnamed* instance of `Dog`

- Call it "`dogWithNoName`"

2 This instance is bound to `bark()` and executed

- "`dogWithNoName.bark()`"

An important point is that Java continues this "evaluation plus binding" strategy in a left-to-right fashion until it consumes the entire expression. Thus, as long as "something" to the left of a dot evaluates to an object and "something" to the right of a dot evaluates to a method, Java will bind the method to the object and perform the operation.

Consider the following program:

```
public class EvalDemo {
  public static void main(String[] args) {
    EvalDemo e = new EvalDemo();
    e.printIt("One, ")
     .printIt("Two, ")
     .printIt("Three.");
  }

  public EvalDemo printIt(String s) {
    System.out.print(s);
    return this;
  }
}
```

It defines the method `printIt()`, which has an interesting design. The return type is the class name, `EvalDemo`, and the method finishes with a `return` statement for the current instance itself, namely, `this`. Thus, given an instance of this class, we can write code such as the following:

```
e.printIt("One, ").printIt("Two, ").printIt("Three.");
```

Each invocation of `printIt()` performs a unit of work, in this case, displaying a message, *and* then again yields (evaluates to) the current object. The expression evaluation then continues with

```
<current-object>.printIt("Two, ").printIt("Three.");
```

and so on, until the entire expression is consumed.

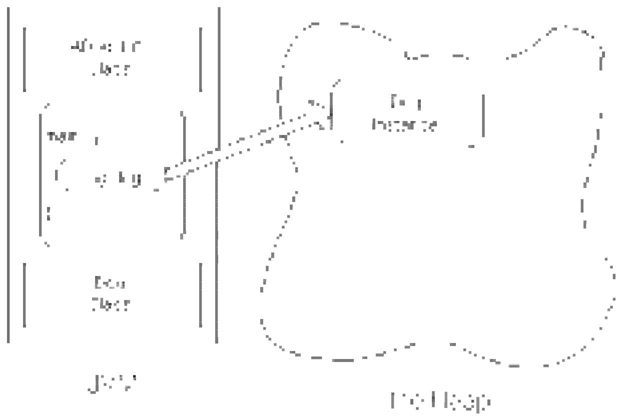
The output from this program confirms the left-to-right evaluation:

```
D:\>java EvalDemo
One, Two, Three.
```

Anytime you have a class design with `void` methods, that is, the methods otherwise do not need to return a particular value, you can use this strategy to support method chaining.

Garbage Collection

One of the really powerful features of Java is its memory-management strategy. Java allocates objects on the heap dynamically as requested by the `new` operator:



Other languages put the burden on the programmer to free these objects when they're no longer needed with an operator such as `delete` or a library function such as `free()`. Java does not provide this functionality for the programmer because the Java runtime environment automatically reclaims the memory for objects that are no longer associated with a reference variable. This memory reclamation process is called *garbage collection*.

Garbage collection involves (1) keeping a count of all references to each object and (2) periodically reclaiming memory for all objects with a reference count of zero. Garbage collection is an old computing technology; it has been used with several computing languages and with environments such as text editors.

Consider the following method:

```
...
  void aMethod() {
    Dog dog = new Dog();
    // do something with the instance dog
    ...
  }
...
```

`dog` is a local variable within `aMethod()`, allocated automatically upon method invocation. The `new` operation creates an instance of `Dog`; its memory address is stored in `dog`; and, its reference count is incremented to `1`. When this method finishes/returns, the reference variable `dog` goes out of scope and the reference count is decremented to `0`. At this point, the `Dog` instance is subject to reclamation by the garbage collector.

Next, consider the following code segment:

```
...
while (true)
```

```
  Dog dog = new Dog();
...
```

Each iteration of the `while` loop (which continues forever) allocates a new instance of `Dog`, storing the reference in the variable `dog` and replacing the reference to the `Dog` instance allocated in the previous iteration. At this point, the previously allocated instance is subject to reclamation by the garbage collector.

The garbage collector automatically runs periodically. You can manually invoke the garbage collector at any time with `System.gc()`.

Runtime Environments and Class Path Settings

The Java interpreter (runtime environment) dynamically loads classes upon the first reference to the class. It searched for classes based on the directories listed in the environment variable `CLASSPATH`. If you use an IDE, it may automatically handle `CLASSPATH` internally, or write a classpath setting to the appropriate system file during installation.

If you do *not* use an IDE, for example, if you're using the Java Development Kit (JDK) from Sun, you may have to set a classpath before running the Java compiler and interpreter, `javac` and `java`, respectively. Also, note that in most situations the installation procedure will automatically update the `PATH` environment variable, but if you're unable to run `javac` or `java`, you should be aware that this setting could be wrong.

`PATH` environment variable settings vary across operating systems and vendors. In a Windows environment, the following setting augments the old/existing `PATH` setting (`%PATH%`) with `c:\java\bin`:

```
set PATH=%PATH%;c:\java\bin
```

For this example, when attempting to run a Java IDE, or the JDK's compiler or interpreter, Windows includes the directory `c:\java\bin` in the search for the executable program. Of course, this setting (`c:\java\bin`) will vary from one Java environment to the next. Note that the path separator character is "`;`" for Windows environments and "`:`" for UNIX environments.

If you find it necessary to set the `CLASSPATH` environment variable, for example, if you're using the JDK from Sun, it should include all directories on your computer system where you have Java class files that you want the Java compiler and interpreter to locate. As you add new class-file directories, you will typically augment this classpath setting. In a Windows environment, the following statement sets `CLASSPATH` to include three components/sites:

```
set CLASSPATH=c:\java\lib\classes.zip;c:\myjava\classes;.
```

Note that this setting includes a zipped class file archive `classes.zip` in the `lib` directory of a particular Java environment's distribution directory, represented generically here as `c:\java\`. That is, most Java environments can read class files stored in archive files of type `.zip` and `.jar`, as well as unarchived class files in any specified directory. During installation, many Java environments "remember" the location of their class files; thus, setting the Java environment's class file location is not necessary.

In this example, `CLASSPATH`'s semicolon-separated entries also includes `c:\myjava\classes\`, a personal/user collection of class files, and "`.`", which represents the current directory. The latter setting is convenient for working with Java files in an arbitrary directory that's not listed in the classpath setting.

Windows 9x and NT users can set classpaths manually with a text editor in the file `autoexec.bat`, plus Windows NT users can set a classpath via the control panel's System dialog. UNIX users can set a classpath manually in the appropriate shell script's configuration file. Please refer to the appropriate system reference books and documentation that describe how to set an environment variable.

If you're using the JDK from Sun, you can (1) install one of several freeware, or cheapware, tools that automate the process of presenting a text editor window for writing Java programs and then invoking `javac` or `java` from a graphical IDE button or (2) invoke these programs directly from a command window.

It's impractical to attempt a demonstration of the many IDEs, however, compiling and running a Java application with the JDK is quite straightforward. The following commands demonstrate the appropriate commands:

```
D:\>javac SimpleProgram.java
D:\>java SimpleProgram
This is a simple program.
```

Your Java environment will almost certainly vary in several ways from what we've described here.

Java Applets

Java is a powerful and elegant programming language. Ironically, however, many people think of Java only in terms of its use for developing applets. In reality, Java is becoming the language of choice for a broad range of other development areas. Nevertheless, applets play an role important in many intranet environments because they provide an (elegant) way of implementing web-based user interfaces to enterprise-wide computing services.

An applet is an instance of a user-defined class that specializes (inherits from) `Applet` (`java.applet.Applet`). Class inheritance is beyond the scope of this section, but, for now, to *specialize* a class is to extend its capabilities. `Applet` is a placeholder class with an empty `paint()` method. Thus, to develop a minimal applet that displays in a portion of a web browser window, you implement a `paint()` method that renders graphical output.

Applets employ Java's Abstract Windowing Toolkit (AWT) for the `Graphics` class, which provides drawing primitives, as well as for GUI components such as `Button` and `TextField`. With these components it's straightforward to design graphical forms-entry utilities that corporate-wide users access from a web browser.

Although applet programmers often develop task-specific implementations of several methods such as `init()`, `start()`, `stop()` that control the applet lifecycle in the browser window, a minimal example with `init()` and `paint()` is sufficient here. `DogApplet.java` implements a simple applet that renders a graphical barking message:

```
import java.awt.*;
import java.applet.Applet;
public class DogApplet extends Applet {
  public void init() {
    setBackground(Color.pink);
  }
  public void paint(Graphics g) {
    g.drawString("Woof!", 10, 20);
  }
}
```

`init()` set the background to an uncommon color to ensure that its allocated browser window area is visible. Java-enabled web browsers execute `init()` only once, and prior to other methods. `paint()` uses the `Graphics` instance, passed as an argument by the browser environment, to draw a string at coordinates (10, 20) relative to the applet's window area.

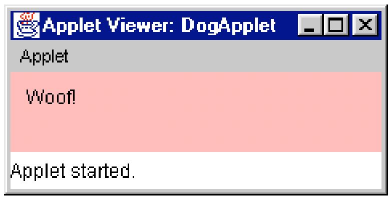
To specify an applet in a web page, you must provide an HTML applet tag that specifies the class file (`code="`*`class-file`*`"`) and its relative location (`codebase="`*`location`*`"`), as well as a width and height request for the applet's window area relative to other components in the web page. For example, this document includes the following applet tag:

```
<applet code="DogApplet" codebase="classes"
  width=100 height=50>
</applet>
```

In processing this tag the web browser:

- Loads the `DogApplet` class file
- Allocates its area in the window
- Instantiates `DogApplet`
- Executes prescribed methods such as `init()`

`DogApplet` appears as follows:



Woof!

Applet started.

Applets are addressed elsewhere in detail. One reason for mentioning them in this introduction is to point out that applet development is not trivial, and in many cases, is not the best solution for simple animations.

It's true that applets can be used for iterating over a series of GIF images to present a simple animation. Recently, however, with the availability of several editors for animated GIF images, the latter is often more appropriate for *simple animations*. With GIF editors you can easily control common animation characteristics, whereas with applets you must program this functionality. Of course, the applet technology provides a more powerful range of programming facilities for handling complex animations.

Further Reading and References

Complete information on the Java language, including all syntax-related issues such as operator precedence, literals, and so on is available in *The Java Language Specification*, which is included in the following list. The following list of references includes several of the more popular Java programming books as well as an academic-oriented, introduction to programming with Java (*Java Gently*), and lastly, the online tutorial link at the Sun website:

- *The Java Language Specification*, Gosling, J., Joy, B., and Steele, G. Addison-Wesley, 1996, ISBN 0-201-63451-1.
- *Java in a Nutshell*, Flanagan, D. O'Reilly & Associates, 1997, ISBN 1-56592-262-X.
- *Core Java 1.1, Volume I - Fundamentals*, Horstmann, C.S. and Cornell, G. Prentice Hall, 1998, ISBN 0-13-766965-8.
- *Core Java 1.1, Volume II - Advanced Features*, Horstmann, C.S. and Cornell, G. Prentice Hall, 1998, ISBN 0-13-766965-8.
- *Java Gently*, Judy Bishop. Peachpit Press, 1998, ISBN 0-201-34297-9.
- The JavaSoft website (http://java.sun.com/docs/books/tutorial/index.html).

[*MML: 1.03*]
[*Version: $ /JavaIntro/JavaIntro.mml#7 $*]

Appendix: A COBOL Programmer's View of Java

This section is meant as a learning aid to show the relationships between COBOL and some of Java's syntax and object-oriented semantics for those COBOL programmers without C, Pascal, or C++ experience. The assumption is that you have been through or are going through an object-oriented programming tutorial or course. The following topics are covered:

- Program organization
- Variable declarations
- Data aggregates (records/objects)
- Arrays
- Scope Overriding and `this`
- Assignment
- Displaying results
- Calling subprograms/methods
- Symbol visibility

For your convenience, a quick reference guide is provided at the end.

Program organization

A COBOL program is divided up into four divisions:

1. `IDENTIFICATION DIVISION`. Defines the name of the program
2. `ENVIRONMENT DIVISION`. Defines computer-specific environment details
3. `DATA DIVISION`. Defines variables, input/output formats, constants, and work areas (storage space)
4. `PROCEDURE DIVISION`. A group of procedures (paragraphs) that do the work of the program

COBOL programs explicitly separate the data and procedures that operate on the data. Procedures are called *methods* in Java.

Most COBOL statements are divided up into "area A" (four characters wide starting from left edge) and "area B" (the rest of the line). All this means is that the division, section, paragraph, and 01 level data start in A and the executable statements start in B. Java has no such formatting restrictions that impart meaning and ignores white-space for the most part. Also, Java is case-sensitive so that `dog` and `Dog` are totally different names.

Java programs are organized by units that correspond to entities in the real world: *objects* that encapsulate both the data and the methods to manipulate that data. Objects with the same characteristics and behavior are described by classes (storage templates) and correspond roughly to COBOL groups. A Java program is then primarily just a collection of class definitions such as:

```
class Vehicle {
  ...
}

class Car extends Vehicle {
  ...
}

class Truck extends Vehicle {
  ...
}
```

Program execution begins in the main() method of any of the classes in your program; you specify which class when you launch the program.

Variable declarations

Variables in COBOL are either elementary or group items and correspond loosely to *primitive* variables (such as integers, characters, and real numbers) and objects in Java. COBOL is not strongly typed like Java, however. In COBOL, you define a "picture" of what can be stored in the variable whereas Java requires a rigid type to be associated with that data storage element. A "`PIC`" can be alphabetic (A), alphanumeric (X), or numeric (9). The commonly-used Java primitives are `boolean`, `char`, `String`, `int`, and `float`.

COBOL uses level numbers on all data items:

- 01 is reserved for group names, which begin in area A
- elementary items begin in area B and use user-defined level values

You initialize items with a `VALUE` clause such as:

```
01  Result         PIC 99 VALUE ZEROS.
```

In Java, there are no level numbers. You must explicitly label something as a primitive or a class (similar to a "group") definition. You initialize variables with an assignment operator:

```
int result = 0;
```

Data aggregates (records/objects)

Here is a simple group called `TheDate`:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TheDate.
    02 CurrentYear PIC 9(4).
    02 CurrentMonth PIC 99.
    02 CurrentDay PIC 99.
```

You can approximate this group of 3 variables in Java with a class definition:

```
class TheDate  {
  String currentYear;
  String currentMonth;
  String currentDay;
}
```

The differences are primarily that the COBOL group defines actual storage space and the exact memory layout ("picture") whereas the Java class definition describes a template for storage rather than the storage itself. A running Java program is a collection of objects. Objects with the same characteristics and behavior are described by the same class. Objects of the same type (class) are therefore considered *instances* of that class. To actually allocate storage for an object, use the `new` operator:

```
// make d refer to a TheDate type object.
TheDate d = new TheDate();
```

A better way to look at the difference between a class and an object is, perhaps, the difference between a `FILE` and a `RECORD`. A record is a collection of fields and a file is a collection of records. There is only one record definition, but there may be many records within a file with that same structure. Similarly, for any class definition, there may be many class instances, objects. The type (class) defines the structure of the object and the object stores the content.

On the other hand, COBOL reads files one record at a time--there is exactly one record from a file in memory at once (the "record buffer"). In Java, the instances of a class all exist in memory at once. If you want 3000 `TheDate` objects, you have to have enough memory to hold 3000 of these objects.

The following file descriptor entry describes a student file with a record structure specified in the `StudentRecord` group.

```
DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentRecord.
   02  StudentId       PIC 9(7).
   02  StudentName.
       03 Surname      PIC X(8).
       03 Initials     PIC XX.
```

You can imagine that the records in the file are like "freeze-dried" Java objects; that is, they are on the disk instead of in memory.

Sometimes you have a COBOL file containing records of different record types. Because you cannot always establish the record type just by examining its contents, you normally put a special data item at the start of each record indicating the record or transation type.

```
FD TransactionLogFile.
01 StudentRecord.
   02  RecordType       PIC X.
   ...
```

Java has a similar notion for objects. You can ask every object for its type. The type information refers to the object's class definition, which includes the list of methods, data fields and so on.

Arrays

COBOL has tables that hold contiguous sequences of elements in memory just like a file holds contiguous sequences of records. Each element in uniquely identified by an integer index ("subscript" in COBOL terminology) within the table. Instead of have 5 name variables called name1, name2, ..., name5 you would define a table with 5 elements:

```
    02 NAMES OCCURS 5 TIMES PIC X(15).
```

The third name variable would correspond to the third element in the table. You access the elements of a table by specifying the table name followed by the index in parentheses:

```
MOVE 'John' TO names(3).
```

Table elements are typically accessed in `PERFORM` "loop" statements:

```
PERFORM VARYING Idx FROM 1 BY 1
        UNTIL Idx GREATER THAN 5
  DISPLAY names(Idx)
END-PERFORM
```

Java has arrays that correspond to COBOL tables, however, Java arrays are themselves objects and are indexed from 0 instead of 1. You must use the `new` operator to create space for an array object. Here is the Java code that declares an array called `names` containing 5 strings:

```
// elements names[0]..names[4]
String[] names = new String[5];
```

To set array elements, use the name followed by square brackets instead of parentheses:

```
names[3] = "John";
```

Java loops are also often used to access arrays. The following loop prints out the elements within the `names` array.

```
for (int i=0; i<=4; i=i+1) {        // i=0..4
  System.out.println(names[i]);
}
```

Scope overrides and `this`

The notion of the `this` variable in Java, referring to the target of a message send, has a few analogs in the COBOL world.

In COBOL, the record buffer associated with a file is sort of a "cursor" or "current record" buffer that changes as you read through the file. If you walk through a list of Java objects, sending them each a message, the `this` variable will refer to each of the objects in turn just like the moving file cursor.

The ambiguity resolution or *scope override* ability of the `this` variable also has an analog in COBOL. When you have two items in two different tables or files with the same name, you must use the `OF` clause to indicate which one you are referring to such as `StudentName OF StudentFile`.

Executable Statements

Program execution/termination

In COBOL, execution begins at the first procedure in the `PROCEDURE DIVISION` and terminates with the statement:

```
STOP RUN.
```

In Java, program execution begins in the `main()` method of the class you tell Java to start with. For example, here is a class with a `main()` method that prints "Hello":

```
public class Simple {
  public static void main(String[] args) {
    System.out.println("Hello");
  }
}
```

From the command-line, you would execute this program by saying:

```
java Simple
```

Assignment

In strongly-typed languages like Java, assignments are only allowed between compatible types. Assignment in Java corresponds to the `MOVE` statement in COBOL; in other words, `MOVE` is really a misnomer--`COPY` would have been more accurate. Assignment copies from source to destination(s).

You can assign a variable to another variable or a literal to a variable as in :

```
MOVE AVERAGE-VALUE TO SUM.
MOVE 'John' to NAME.
```

In Java, you would say:

```
sum = averageValue;
name = "John";
```

The assignment statement in Java is very simple compared to COBOL; there is no truncation or filling of space ala COBOL. The types of the left and right hand sides must be compatible or they are not allowed.

Displaying results

To display results to the terminal in COBOL, you use the `DISPLAY` command:

```
DISPLAY "AVERAGE: ", AVERAGE.
```

Java has a similar statement:

```
System.out.println("AVERAGE: "+average);
```

To display more than just a string, you build up a comma-separated list of elements in COBOL:

```
DISPLAY "AVERAGE: ", AVERAGE WITH NO ADVANCING.
```

In Java, you build up a string, which is then displayed. The plus operator in Java concatenates two strings:

```
System.out.print("AVERAGE: "+average);
```

Calling subprograms/methods

COBOL has two mechanisms that correspond to method calls in Java: `PERFORM`ing paragraphs and calling subprograms.

COBOL procedures are like Java methods with no return values nor arguments. In the following fragment, execution flows from `Begin` to procedure `Blort` to display Hello.

```
PROCEDURE DIVISION.
Begin.
    PERFORM Blort
    STOP RUN.

Blort.
    DISPLAY "Hello".
```

In Java, you would have the following:

```
class Whatever {
  void begin() {
    blort();
  }
  void blort() {
    System.out.println("Hello");
  }
}
```

In Java, you can think of `{...}` statement groups as COBOL unnamed paragraphs. Java for-loops are like `PERFORM` in-lines.

You cannot pass parameters to a procedure. You need to use subprograms in COBOL for that. In Java, you can pass parameters to a method or not depending on your needs.

A method call corresponds to `CALL` of program in COBOL. You can have a single return value in both Java and COBOL, but you can pass in items that the subprogram modifies. COBOL subprogram calls pass parameters with the `USING` clause and the return value is found in `RETURN-CODE`:

```
    CALL 'DISPLAY-COUNT' USING COUNT.
* use RETURN-CODE if you want
    IF RETURN-CODE
    ...
    END-IF
```

In Java, you use the following syntax:

```
displayCount(count);
```

Primitive types in Java such as `int` can only be passed by value (`CONTENT` in COBOL) and all objects are passed by `REFERENCE`.

The definition of a program with parameters in COBOL requires that you define the storage in the `LINKAGE SECTION` and then identify them on the `PROCEDURE DIVISION` header:

```
IDENTIFICATION DIVISION.
PROGRAM-ID DISPLAY-COUNT IS INITIAL.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AVERAGE                     PIC 9(8) VALUE 0.
LINKAGE SECTION.
01  COUNT                       PIC 9(8).
PROCEDURE DIVISION USING COUNT.
Begin.
       DISPLAY "COUNT: ", COUNT
       EXIT PROGRAM.
```

The variables you define in the `WORKING-STORAGE SECTION` are initialized each time you call the subprogram if you use the `IS INITIAL` on the `PROGRAM-ID` statement. This corresponds closely with Java. Here is the Java method equivalent to the above `DisplayCount` program:

```
void displayCount(int count) {
  // set average to zero upon each entry
  int average = 0;
  System.out.println("COUNT: "+count);
}
```

Please see the quick reference at the end of this document for information on other executable statements.

Symbol Visibility

In COBOL, contained subprograms are not visible to other contained subprograms by default. You must use the `IS COMMON PROGRAM` clause to make a subprogram visible to other subprograms:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
PROCEDURE DIVISION.
    ...
    EXIT PROGRAM.

IDENTIFICATION DIVISION.
PROGRAM-ID. A.
* CAN CALL B
    CALL B.
END-PROGRAM A.

IDENTIFICATION DIVISION.
PROGRAM-ID. B IS COMMON PROGRAM.
* CANNOT CALL A
    CALL A
END-PROGRAM B.
END-PROGRAM MAIN.
```

Java has data hiding, but it is used to restrict access to methods within a class from methods outside of the class. Class members labeled as `public` are visible to any method in any other class. Class members labeled as `private` are **not** visible to any method in any other class.

Quick Reference

Statements	
COBOL	**Java**
`MOVE b TO a`	`a=b;`
`MOVE a TO` *`arrayName`*`(`*`index`*`)`	*`arrayName`*`[`*`index`*`]=a;`
`ADD b TO a`	`a = a+b; or a+=b;`
`ADD b TO a GIVING c`	`c = a+b;`
`ADD 1 TO a`	`a++; or a=a+1;`

`IF` *`conditional`*
  `statements1`
`ELSE`
  `statements2`
`END-IF`
	`if (` *`conditional`* `) {`
	  `statements1`
	`}`
	`else {`
	  `statements2`
	`}`

`EVALUATE` *`conditional`*	`switch (`*`conditional`*`) {`

WHEN *value1*
  statements1
WHEN *value2*
  statements2
WHEN OTHER
  statements3
END-EVALUATE
	case *value1*:
	  statements1
	  break;
	case *value2*:
	  statements2
	  break;
	default:
	  statements3
	}

PERFORM TEST BEFORE
  UNTIL *conditional*
statements
END-PERFORM
	while (*conditional*) {
	  statements
	}

PERFORM TEST AFTER
  UNTIL *conditional*
statements
END-PERFORM
	do {
	  statements
	} while (*conditional*);

PERFORM VARYING *a*
  FROM *start* BY
  inc UNTIL *conditional*
statements
END-PERFORM
	for (*a=start*;
	  conditional;
	  *a+=inc*) {
	 statements
	}

PERFORM *routine*.	*routine*();
CALL '*DUMMY-FUNCTION*' USING *arguments*.	*dummyFunction*(*arguments*);

Relational Operators

COBOL	Java
a EQUAL TO b	a==b for primitive types a.equals(b) for objects
a NOT EQUAL b	a!=b for primitive types !(a.equals(b)) for objects
a GREATER THAN b	a>b for primitive types
a GREATER THAN OR EQUAL b	a>=b for primitive types
a LESS THAN b	a<b for primitive types
a LESS THAN OR EQUAL b	a<=b for primitive types

Boolean Operators	
COBOL	**Java**
`a AND b`	`a&&b` for boolean expressions `a`,`b`
`a OR b`	`a||b` for boolean expressions `a`,`b`

Reference

jGuru.com.