INTRODUCTION TO JAVA — PL/SQL DEVELOPERS TAKE HEART!

Peter Koletzke, Quovera

> *Out beyond ideas of wrongdoing and rightdoing there is a field.*
> *I will meet you there.*
> *When the soul lies down in that grass*
> *the world is too full to talk about.*
> *Ideas, language, even the phrase "each other"*
> *doesn't make any sense.*
>
> —Jelaluddin Rumi (1207–1273), *A Great Wagon*,
> (translated by Coleman Barks)

If you are a currently a PL/SQL developer or live or work close to such a person, you may have become aware of Oracle's current focus on Java. You know that Oracle has stated repeatedly that PL/SQL is here for the long term, but you also watch as Oracle implements more and more Java features in the database, application server, and development tools. All of this may have led you to the conclusion that it is now time to learn Java.

As a PL/SQL enthusiast, your first view of Java may be a bit discouraging because its object-oriented core makes it look very different. Also, you may be trying to sort from all the marketing hype basic concepts about Java's strengths and weaknesses and where it fits in the industry.

Demystifying the unknown usually helps. Therefore, an objective of this paper is to explain the basic concepts of and terms used in Java. When learning a new language, it is usually only necessary to learn the following:

- **The syntax for program control** such as iteration and conditional statements
- **How programs code is organized**, compiled, distributed, and run
- **The built in datatypes** offered by the language and how to declare and use variables

When you are learning Java, it is still necessary to understand these things. However, if you have not come in contact with object orientation (OO) concepts while developing code, when learning Java you also need to learn how OO works because Java is an OO language. This paper starts with a discussion of why you would use Java and how object orientation works. Often language explanations are a bit dry, so this paper then takes a slightly non-traditional route to explaining Java language elements. Assuming that you know a language such as PL/SQL as well as programming logic and can recognize parallels and syntax variations in another language, this paper shows an annotated example of the code and documents each line. This section acts as a guided tour of most key Java elements. The paper then discusses other elements necessary to understand Java.

Note: This white paper was adapted from the material in the *Oracle JDeveloper 10g Handbook*; McGraw-Hill/Osborne, 2004; by Dr. Avrom Roy-Faderman, Peter Koletzke, and Dr. Paul Dorsey.

Note

Naturally, it is not possible to explain an entire language in a short white paper such as this. To complete your understanding of Java, you will need further study, through either self-directed or formal training. A good resource for more information is the no-cost Java Tutorial available at Sun Microsystem's website, java.sun.com.

WHY JAVA?

Java is a relatively new, object-oriented language (officially launched in 1995) that provides many ways to deploy the code. Object orientation offers benefits in analysis and design because business concepts are more easily matched with objects than with standard relational structures. These concepts map easily to programming elements in an object-oriented language such as Java.

If you are in the process of evaluating the Java language for use in a production environment, you need to consider both its benefits and drawbacks as well as what you will need to make the transition.

BENEFITS

The IT industry is proceeding at a breakneck speed into Java technologies (primarily Java 2 Platform, Enterprise Edition) because of the perceived benefits. It is useful to examine some of the main strengths that Java offers.

FLEXIBILITY

Java is implemented as a rich set of core libraries that you can easily extend because the language is object oriented. Distributing these extensions is a normal and supported part of working with Java.

Java supports light-client applications (through technologies such as JavaServer Pages), which only require a browser on the client side. Running the client in a browser virtually eliminates runtime installation and maintenance concerns, which were a stumbling point with client/server application environments such as Forms (before it could be web deployed).

Java also supports deployment as a standalone application with a Java Virtual Machine (JVM) runtime on the client. It solves the problem of supporting different screen resolutions with layout managers that are part of the core libraries.

The Java language is a core component of standards such as Java Platform, Enterprise Edition (Java EE, formerly known as "J2EE"). It is used as the language in which basic libraries, such as those from which JavaServer Pages (JSP) code is built, are written. In addition, you can embed Java code snippets inside JSP files to perform actions specific to the web page.

A popular Java EE design pattern, Model-View-Controller (MVC) defines layers of application code that can be swapped out when the needs of the enterprise change. With MVC, the view (presentation) layer can be implemented as a Java application running on the client with Swing components or as HTML elements presented in a browser. Although these views are different, they can share the same model (data definition and access) and controller (behavior and operation) layers. This kind of flexibility is a benefit.

Current strategies for deployment of Java code emphasize multi-tier architectures that provide one or more application servers in addition to client and database server tiers. Although this feature is not unique to Java environments, it is one of the main design features of current Java web architectures. The application server approach offers flexibility and better scalability as the enterprise grows. For example, to add support for more clients, it is only necessary to add application servers and software that distribute the load among servers. The client and database tiers are unaffected by this scaling. A multi-tier approach also offers a central location to support business logic that is common to many applications.

Note

Another characteristic that makes Java attractive is its relative ease of use. For example, the Java runtime automatically handles memory management and garbage collection. Also, Java supports multiple threads so that you can write a program in Java that runs in multiple simultaneous threads of execution.

WIDE SUPPORT FROM VENDORS

A compelling reason to use Java is that it is supported by many vendors. Instead of one main vendor, as with other technologies (for example, Microsoft .NET Framework), hundreds of companies produce and support Java products. Oracle is one of these companies. Oracle has a large stake in the Java world and continues to offer its customers guidance and robust product features for Java application development and deployment. Due to this wide support from vendors, the choice of Java as the language is not strongly tied to a single vendor who may not be viable or strong in the future.

WIDE SUPPORT FROM USERS

Another source of wide support is the user community. Java has a well-established user base that is not necessarily tied to a particular company. The Java community is reminiscent of the early days of Unix, when users made their work available to other users on a not-for-profit basis. The concept of *open source* (www.opensource.org) includes free access to the source code, no-cost licenses, and the ability for others to extend the product. For example, the Linux operating system started and continues to be enhanced through open-source channels.

Although the Java language is not an open-source venture, there are many Java products, such as the Apache web server, that are open-source products. Sample Java code is readily available from many sources on the Internet. In addition, many freeware (with no-cost licenses) or shareware (try before you buy) class libraries are available to Java developers.

PLATFORM INDEPENDENCE

Java source code and runtime library and application files are not specific to a particular operating system. Therefore, you can create and compile Java class (runtime) files in a Windows environment and deploy the same files in a Unix environment without any changes. This aspect of Java, sometimes referred to as *portability*, is important to enterprises that find themselves outgrowing a particular operating environment but that need to support previously created systems in a new environment.

DRAWBACKS

Many of Java's drawbacks are derived from the same features as its benefits and result from the newness of the language.

RAPIDLY CHANGING ENVIRONMENT

The Java environment is less mature than traditional environments that access a relational database. This immaturity has two main effects: frequent updates that add significant new features, and shifts in technologies that occur more rapidly than in traditional environments. For example, when Java was first released, the main deployment environment was within a Java Virtual Machine (JVM) running on the client machine. As that environment matured, there were features added and features *deprecated* (supported, but specially marked as being removed or replaced in future releases).

In addition to updates in the language, additional technologies were added to the mix. Associated specifications such as Java Database Connectivity (JDBC), portlets and portals, and wireless Java guided how Java was used. Different environments were also developed. For example, in addition to the environment of Java running on the client, there are now many variations on web-deploying a system developed in Java. In fact, the Java web-deployment landscape is so complex that as part of the Java EE specifications, Sun Microsystems has created *blueprints* (called BluePrints*),* which are descriptions of proven techniques and best practices for deploying applications. Java EE also includes descriptions of proven, lower-level coding techniques called *design patterns* that are used as additional guidelines for development.

MULTI-VENDOR SUPPORT

Although multi-vendor support was listed as one of Java's strengths, it can also be thought of as a drawback. You may need to merge Java technologies from different vendors, and each vendor is responsible only for their part. Oracle offers a complete solution for development (JDeveloper) and deployment (Oracle Application Server), but you may find yourself in a multi-vendor situation if Oracle products were not selected or were extended with components from other vendors. In addition, Oracle is not responsible for the base Java language. In that respect, a Java environment will always be multi-vendor because Sun Microsystems, which is responsible for the language, does not offer a complete solution including a database.

SIGNIFICANT LANGUAGE SKILLS REQUIRED

Java developers need to think in an object-oriented way as well as to understand all aspects of the language and how the code pieces tie together in a production application. Java coding is largely a 3GL effort at this point. The IDEs assist by generating starting code and providing frameworks (such as Oracle's Application Development Framework), but developers also need to have solid programming skills to effectively create production-level Java programs. Java is a popular language now with colleges and universities, so many new graduates are well trained in Java.

In addition, for web-deployed Java, developers need to have skills in other languages and technologies such as HTML, XML, JavaScript, and JSP tags. These skills are easily obtained, but are essentially prerequisites to effective work in the Java web environment.

If developers are to be completely effective, they must also have solid knowledge of database languages. Tools such as JDeveloper's Application Development Framework Business Components (ADF BC) and other technologies (such as Oracle Application Server TopLink) hide the SQL statements from the developers. However, developers must be aware of how the SQL statements are produced by the tools so that the statements can be as efficient as possible.

In addition, although database-stored code (packages, procedures, functions, and triggers) can be coded in Java, developers may still need to interface with existing code built in PL/SQL and, therefore, will need to understand some PL/SQL.

TRANSITIONING TO JAVA

Working in a Java environment is very different from working with traditional database development environments such as Oracle Forms Developer or Visual Basic (VB). If you are not already using Java but the benefits of Java have convinced you of the need to make the transition, you will need to plan that transition carefully.

It will take time to learn the nuances of a Java environment. If you are committed to creating an organization-wide Java environment, building a traditional client/server application using a local Java client (Java running on the desktop) may still make sense if your application is used in a small group or departmental situation. Coding and deploying the application locally on the desktop in this way postpones the complexities of working with web deployments and can give you a feel for how Java works.

If your development team has skills in other languages, Java will require retraining and ramp-up time. Building client/server applications directly in Java can be a good first step. This method leverages the improved flexibility of Java and its ability to build sophisticated applications. It also makes the transition of your business to the Web easier because Java is a primary language of the Web. The smaller the application, the easier it will be to concentrate on the language and not the application. A prototype or internal administrative application that will not see extensive use might be a good candidate for this first effort.

Another variation on this transition advice is to develop a small web application in Java. This can be the next step after building a client/server application, or it can be the first step. Web applications add the complexity of application and web servers, and this will give you a taste of this extra layer of software.

MAKING THE LEAP

The transition to Java may not need to be (and probably should not be) a big bang where you move all new development to Java and start converting existing applications to Java. Although you want to minimize the number of tools and environments that you support, it is likely that you will have to support existing applications in the environments in which they were written. With all current development tools trying to improve their web-enabled capabilities, it becomes increasingly difficult to make a compelling argument for abandoning these technologies. For example, following a long evolution, Oracle Forms Developer running over the Web is now a stable and viable environment. Especially since you can extend Forms with Java plugins, there may be no reason to convert legacy Forms applications to Java.

Therefore, the best approach to transitioning into the Java environment is to leave core application development in whatever legacy environment you are comfortable with and to build a few systems of limited scope in Java using JDeveloper. Once you have some experience in building and deploying applications, you can make an informed decision about whether your organization is ready to make the transition to an entirely Java-based environment. There may still be good reasons to stay with a legacy environment for core applications and only to use a Java environment for e-commerce and other web-based applications.

As you become more comfortable with working in Java and have more Java projects under way, you can think about migrating current applications. However, some applications may never need to make the transition.

Note

As with many shops that support legacy COBOL-based programs, it is likely that you will have to support your current development environment for large enterprise-wide applications for some time.

OBJECT-ORIENTATION CONCEPTS

If you have experience in the C++ language, you will notice keyword and syntax similarities between Java and C++. However, there are enough differences between Java and C++ that it is worth reviewing the basics of the language even if you understand the concepts behind C++. Java used C++ concepts as a springboard but was designed as an object-oriented language in its first incarnation (unlike C++). Understanding Java requires a comfort level with the concepts of object orientation (OO). If you have any experience with another object-oriented language, such as Smalltalk or C#, you may have already grasped the OO concepts you need to work with Java

The fundamental building block of an object-oriented language like Java is a structure called a *class*. The class acts as a pattern or blueprint from which *objects* are built. In object-speak, "an object is an instance of a class." An object is built from a class and has an identity (or name). This means that, primarily, the class is actually not used as a programmatic element except to create other elements that you will manipulate (assign values to and query values from). For example, an object called "someBox" can be instantiated from a class called "Box." In this example, the someBox object is created from the pattern defined by the Box class.

The concept of a class and its object loosely parallels the concept of datatyping (and record variable definition) in programming languages such as PL/SQL. In PL/SQL, a datatype is a pattern upon which variables are built. If the datatype concept were expressed in object-oriented terms, a variable would be the object that instantiates the datatype (acting as a class). In fact, as you will see later in this paper, Java objects are thought of as being typed from classes. In the Box example, you can say that "someBox is of type Box." In common Java parlance, you might also refer to "someBox" as "the Box" or "the Box object."

The parallel between a PL/SQL variable and a Java object is loose because, in addition to creating and initializing the Java object (as you do a variable in PL/SQL), you need code to "create" the Java object using code (you normally use the new operator to accomplish this creation). The creation operation runs constructor code to assign it some default data as well as to run code specific to the class. This concept will be discussed more a bit later in this paper.

A class contains both data (values in variables, also called *attributes* or *fields*) and behavior or application code logic (in methods). This makes it different from anything in the world of relational databases. The closest concept to the class data and behavior characteristics is a relational table with a dedicated package of procedures that are used for SQL operations such as INSERT, UPDATE, DELETE, and SELECT. In this case, the combination of relational table and procedural package contains data (in the table) and behavior (procedures and functions in the package assigned to the table). Another example in the Oracle relational database paradigm is a database view with a procedural package run by INSTEAD OF triggers. The database view (the data) combines with the package and INSTEAD OF trigger (the behavior) to form a rough equivalent to an object-oriented class.

The difference between this example from the relational/procedural world and the object-oriented paradigm is that there is only a conceptual link or loose coupling between the table and the code package. The table and package exist as separate objects and can be used separately. (Although you could link the table and PL/SQL package using table triggers, this mechanism is not required by or native behavior of the language.) In object orientation, the class is inherently both data and behavior; the link is tight and perfectly integrated. The class is used as a pattern to create objects that contain data and pointers to the code in the class. Figure 1 depicts the coupling difference between data and application code for the relational/procedural and object-oriented paradigms.

HANDLING AND STORING DATA

The ways in which data is handled in an object-oriented language such as Java and in a relational database system are also fundamentally different. Data is not inherently *persistent* (permanently stored) in Java. Therefore, data is available only for the time in which the Java program is running. There are ways to store data in between program sessions; the method included as part of the base language is *object serialization*. Object serialization includes the ability to write object values to, and read object values from, a persistent stream (such as a file).

Object serialization is the built-in Java way to handle persistence. However, many programmers of Java and other languages have become accustomed to using a fully featured relational database management system (RDBMS) to handle data persistence. An RDBMS provides solid facilities for fast and safe storage, retrieval, backup, and recovery of mission-critical data. However, the RDBMS, by definition, is built around the concept of storing data in relational tables. This concept does not correspond to the way in which the Java language handles data in objects. Figure 2 shows a conceptual mapping that you can make between relational and object-oriented data storage.

This diagram shows how a row in a table roughly corresponds to the data in an object. You can describe a table in object-oriented terms as a collection of records representing instances of related data that are defined by the structure of the table. The problem is the difference in the way that data appears in the two paradigms. A table contains rows that are accessible using a relational database language such as SQL, which addresses requests for sets of data to the table. Objects contain data and you address requests for that data directly to the object. You cannot use SQL to access data in an OO environment because the source structure of the data is different; data is distributed across many objects. The standard solution to this mapping problem is to create an array object (often called a *row set*) of values that represents multiple rows in a database table. The row set is a single object with methods for retrieving individual rows and column values.

Therefore, there are basic differences between the relational and the object paradigms in the areas of persistence, data structures and access code, and conceptual foundations. Using Java code to access a relational database is a common requirement, and there are many solutions to making an effective map between relational tables and objects. For example, architectures such as JDeveloper's ADF Business Components or Oracle Application Server TopLink hide the complexity of the relational-object mapping and provide programmers with object-oriented, Java-friendly structures that can be easily incorporated into application programs. Other strategies, such as JDBC, also ease the burden of accessing a relational database from an object-oriented language.

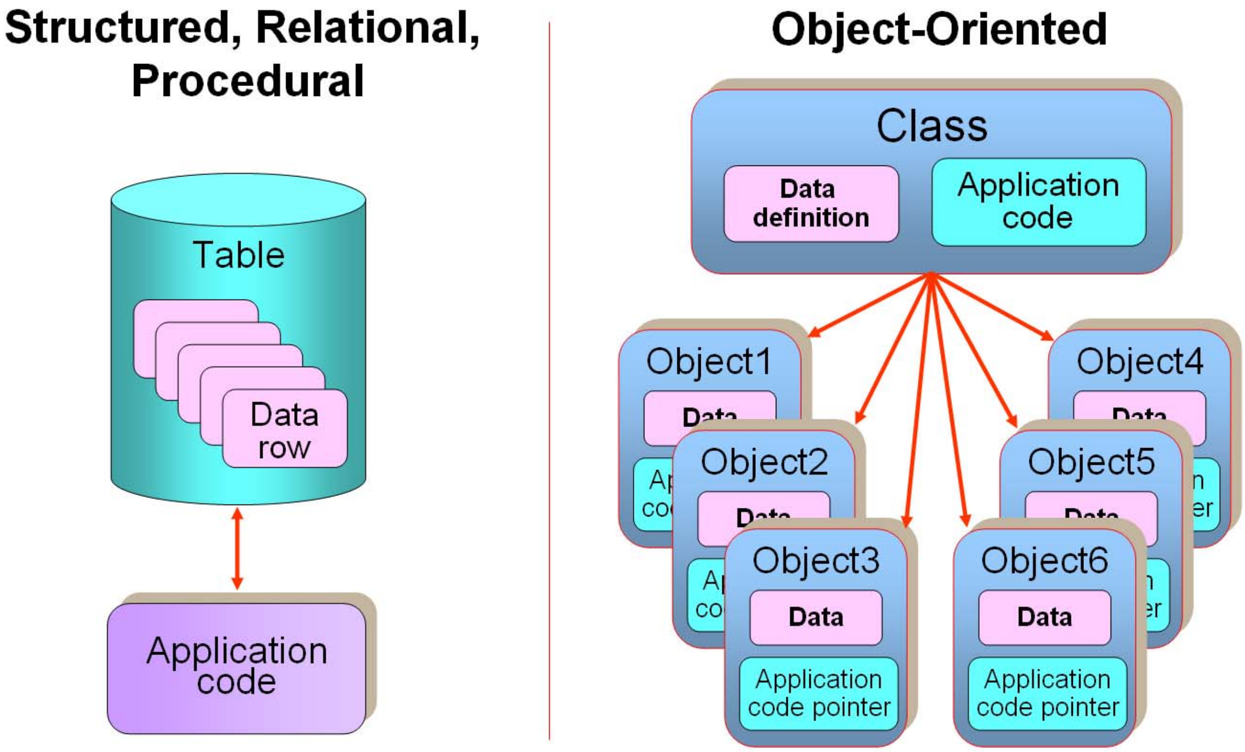


Figure 1: Relational/procedural and object-oriented paradigms

INHERITANCE

One of the key characteristics of object orientation is the ability of a class to automatically take on the attributes and methods of another class. This is the concept of *inheritance,* where one class is a parent of another. The parent, also called the *superclass* (base class or generalization), contains elements (data and behavior) that are available in the *subclass* (child class or specialization). The lines of inheritance can be deep, with one class acting as the grandparent or great-grandparent of another. To base a class on a parent class, you *extend* the parent class. The child class can then supplement, modify, or disable the attributes and behavior of the parent class. This kind of inheritance is shown in an annotated code example later in this paper.

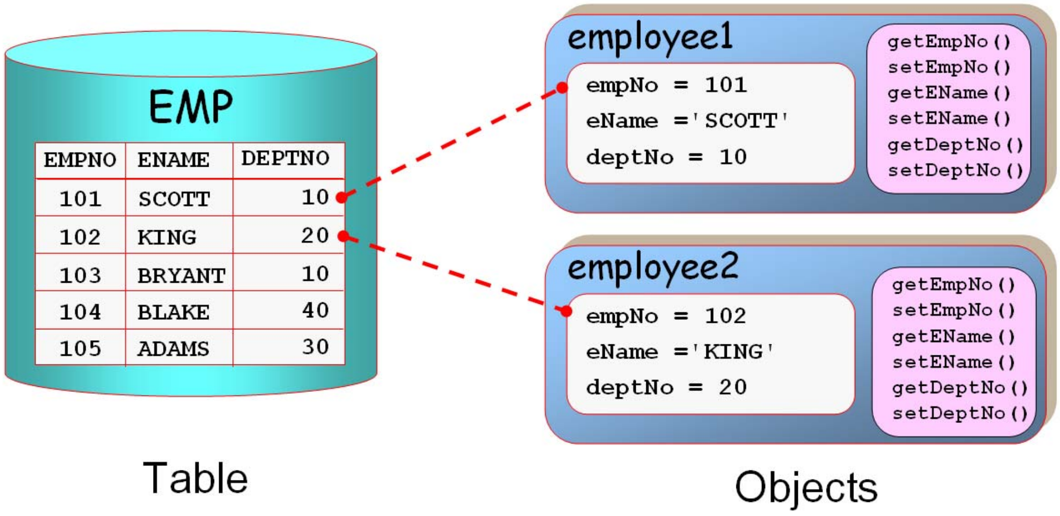
OTHER PRIMARY OBJECT-ORIENTED CONCEPTS

Two other major concepts are inherent to object orientation—*polymorphism* and *encapsulation*. Although they are key to object orientation and it is necessary to understand them when you are developing production systems using Java, you do not need to understand them fully to begin working with Java. The following provides a brief definition of these concepts:

- **Polymorphism** The ability of a class to modify or override inherited attributes or behavior. This is a key feature of the Java programming language allowing the developer to create template classes as well as extensions to these classes, which may (but are not required to) inherit attributes and behavior from the generalization (master) class.
- **Encapsulation** Only the important characteristics of an object are revealed; the internals are hidden. Encapsulation is accomplished in Java by means of access modifiers (explained later in this paper).

JAVA LANGUAGE OVERVIEW

High-level, theoretical discussions of object orientation often glaze the eyes of the audience. The theory makes more sense when it is demonstrated using some code examples. The following section shows a code sample and explains its contents to demonstrate object orientation and Java language concepts. Even if you do not have extensive Java experience but have been exposed to other programming languages, you will be able to identify some of the language elements as well as the structure of a typical source code file. The example also demonstrates, in the context of the Java language, some of the key object-oriented concepts just explained.



EMPNO	ENAME	DEPTNO
101	SCOTT	10
102	KING	20
103	BRYANT	10
104	BLAKE	40
105	ADAMS	30

Figure 2: Mapping between relational and object data storage

ANNOTATED JAVA CODE EXAMPLE

All Java code is contained in class files. A class file is made up of a number of standard elements. The following is a representative class file that contains standard elements. The line numbers in the code listing serve as reference points and would not appear in the actual code file.

```
01: package shapes;
02: import java.lang.*;
03:
04: /*
05:    This class defines a shape with three dimensions
06: */
07: public class Box extends Shape {
08:   int height;    // override the Shape height
09:   int depth;     // unique to Box
10:
11:   public Box() {
12:     height = 4;
13:     width = 3;    // width is inherited from Shape
14:     depth = 2;
15:   }
16:
17:  public int getDepth() {
18:    return depth;
19:  }
20:
21:  public void setDepth(int newDepth) {
22:    depth = newDepth;
23:  }
24:
25:   // super.getWidth is the same as getWidth() here
26:   public int getVolume() {
27:     return height * super.getWidth() * getDepth();
28:   }
29: }
```

> **Note**
>
> All code in the Java language is case-sensitive. Therefore, a class called “BOX” is different from a class called “Box.” By convention, class names are mixed case, with each word in the name initial-capped. For example, a class that defines salary history would be called “SalaryHistory.”

Package Declaration

Line 01 identifies the location of this file. *Packages* are collections of class files in subdirectories in a file system. A package normally represents a directory in the file system. Packages of files can be archived into a .zip or .jar (*Java Archive* or *JAR*) file, and the Java runtime can search in this archive (sometimes called a *library*) for a specific class file. If a package is archived, the archive instead of the file system then contains the directory structure.

The *CLASSPATH* operating system environment variable contains a list of these archive files separated by colons (for Unix) or semicolons (for Windows), for example:

```
C:\JDev10g\jdk\jre\lib\rt.jar;C:\JDev10g\jdk\jre\lib\jce.jar;C:\JDev10g\
jdk\jre\lib\charsets.jar;C:\JDev10g\jdk\jre\classes;C:\JDev10g\jdev\lib\
jdev-rt.jar
```

(The path is a single variable value entered on one line.).

Whether a particular file is inside a file system directory or a directory inside an archive file is irrelevant, but the CLASSPATH must include the archive file name if the file required is in an archived directory. The CLASSPATH must include the name of the file system directory if the file required is in a file system directory.

Line 01 ends with a semicolon, as do all Java statements.

> **Note**
>
> You can view the contents of JAR files by using any file decompression utility.

Import

Line 02 defines an *import* (called an “include” in other languages such as C++), specifying the external or library classes required for the code to compile and execute. Imports are identified by class name as well as by the package in which they reside. Many import statements may appear if more than one package structure or class is required. The import statement can reference a single class file or an entire package as in line 02. The directory is listed using fully qualified dot syntax (package.subpackage.subpackage.*), with “*” indicating that all classes in that package will be available. Java libraries are often grouped by function into the same directory (package) so that associated functions can be called more easily. This example is provided for discussion purposes. The java.lang classes are automatically available without an import statement.

Comment

Lines 04–06 show a multi-line *comment* (using the beginning and ending symbols “/*” and “*/”). Line 08 shows a single-line comment (using the beginning symbol “//”) at the end of the code line. As shown in line 25, this style of comment can be used on a line by itself. Java also offers special multi-line comments that start with “/**” and end with “*/” and are used to generate a type of documentation called “Javadoc.” The javadoc program extracts these special comments and presents them in a formatted HTML file.

Class Declaration

Line 07 is the class declaration. It includes the keyword `public`, indicating that this class is available to all other classes. The keyword `public` is called an *access modifier* (or *access specifier*). Other choices for access modifiers are `private` (where access to the class member is limited to other members of the same class) and `protected` (where you cannot access the class from outside the package unless the calling class is a subclass). If you do not use a modifier keyword, this indicates the *default* modifier, where members are available only to code in the same package.

CODE BLOCK

The end of line 07 contains an opening curly bracket "{" indicating the start of a *block of code*. Between this and the matched closing curly bracket "}" are code elements and other blocks of code. The code blocks do not need to contain any code (unlike blocks in PL/SQL). Blocks of Java code may be nested. Blocks define the scope of variables and code structures such as `if` and `while`, as discussed later.

USING BRACKETS FOR CODE BLOCKS

It is good coding practice to always use curly brackets to contain code in code structures (such as `if` and `for`). Technically, you do not need curly brackets if there is only one statement to execute as in the following example:

```
if (width == 10)
  depth = 20;
```

However, it is easy to make the mistake of adding a line of code under an `if` statement that has no curly brackets and assume that it will execute conditionally. Since only the first line of code under the `if` statement is part of the conditional logic, the next statement would always be executed if no curly brackets contain the statements, for example:

```
if (width == 10)
   depth = 20;
   height = 30; // this will always execute
```

Always using curly brackets, even for single-line blocks, will prevent this type of error as in the following example:

```
if (width == 10)
{
   depth = 20;
}
```

Note

No recognized standard exists for whether to place the starting bracket for a block at the end of the line above or on a new line. This paper shows examples of both techniques but for consistency and readability, you will want to make a particular starting bracket style a standard for your application code.

SUBCLASSING (EXTENDS)

This code defines a class called `Box` that is built (subclassed) from a class called `Shape`. The `extends` keyword declares that `Shape` is the parent of `Box` and defines the inheritance for `Box`. The `Shape` class might be defined as follows:

```
package shapes;

public class Shape  {
  int height;
  int width;

  public Shape() {
    height = 1;
    width = 1;
  }

  public int getHeight() {
    return height;
  }

  public void setHeight(int newHeight) {
    height = newHeight;
  }
  // getWidth() and setWidth() methods go here
}
```

Note

A Java source code file can contain only one public class. The file name is the name of the public class in that file (with a .java extension).

VARIABLE DECLARATION

Lines 08–09 (of the `Box` class) declare two variables (as `int` types). These constitute the data (attribute or field) for the class that was mentioned in the discussion of object-oriented concepts. Since the variables of the parent class are available in the child class, the `Shape` variables `height` and `width` are available to the `Box` class. In addition, the `Box` class declares its own `height` variable. This variable is available to objects created from the `Box` class. The parent's `height` variable is also available using the symbol `super.height` (the `height` variable of the `Shape` superclass). This code also declares a variable that is not in the parent: `depth`. Variable names are formatted with initial caps for all words except the first.

Note

The modifier "super" can be also be used in the child class to access methods from the parent class. For example, the `Shape` class contains a `setHeight()` method, which is available to the `Box` class (subclass) as `super.setHeight()`. This modifier can also be used for constants and variables.

Technically, *variables* in Java are declared using primitive datatypes such as `int`, `float`, or `boolean`. Other data resides inside objects that are instantiations of classes such as `String` or `StringBuffer`. Datatypes are discussed in more detail later in this paper.

CODE UNIT—METHOD AND CONSTRUCTOR

Lines 11–28 define methods and constructors, which are the main containers for functional code in the class.

METHOD

The standard unit of code in Java is called a *method.* The first line of the method is called the *method signature* or *signature* because it contains the unique characteristics of this code unit such as the arguments, access modifier, and return type.

You can use the same name for more than one method in the class if the methods with that name all have different argument lists (for example, `showWidth(int width)` and `showWidth(String widthUnit)`). Methods with the same name but different arguments are referred to as *overloaded methods.*

Methods implement the object-oriented concept of behavior in a class. Java does not distinguish between code units such as functions that return a value and procedures that do not return a value. It has only one unit of code—the method. Methods must declare a return type. They can return a primitive or class—as with functions in other languages—or they can return *void* (nothing)—as with procedures in other languages. Methods can only return one thing, but that thing could be an array or an object, which could be made up of many values.

Method names are, by convention, mixed case, with each word except for the first one initial-capped.

CONSTRUCTOR

Lines 11–15 define a unit of code called "Box," which has the same name as the class. This unit is called a *constructor* and is not a method. Constructors have a signature similar to methods, but do not return anything (not even `void`). If a return type is declared, the signature identifies a method not a constructor. Constructors are used to create an object that is based on the class (using the keyword `new`); in this case, the constructor just sets values for the variables in the object that is created from the class. The constructor code is contained within a block of code delimited by curly brackets. If you do not define a constructor, you can still create an object from the class because there is an implicit constructor that is available to calling programs.

Constructors must have the same name as the class. Since class names usually begin with an initial-capped word, the constructor also begins with an initial-capped word.

ACCESSOR (GETTER AND SETTER)

Lines 17–28 define getters and setters (also called *accessors* or *accessor methods*). A *getter* is a method that is usually named with a "get" prefix and variable name suffix. It returns the value of the variable for which it is named. Although this is a standard and expected method that you would write for each property or variable that you want to expose, it is not required for variables that you want to hide. Getters may include security features to restrict access to the values and may not be as simple as this example, which just returns the value of a single variable. In this example, the `getDepth()` method returns the value of the variable, and the `getVolume()` method calculates the volume based on the three dimensions of the box. As with the variable prefix "super." before, this method references the superclass methods getWidth() and getDepth().

Note

For readability, getter methods for boolean variables are usually prefixed with "is" instead of "get." For example, the boolean variable `hasWheels` would be accessed using a getter `isHasWheels()`.

A *setter* is a method that is named with a "set" prefix and is normally used to change the value of the property for which it is named. As with the getter, it is normal and expected that you would write a setter for variables you want to expose to the caller; you would omit the setter if you did not want the variable changed. Setters can include validation logic (for example, not allowing the height to be set to a negative number), but they usually also assign a new value to the variable as in this example.

The object-oriented concept of encapsulation is implemented here by accessor methods that read and write to private variables in the class. Other code outside the class cannot see or manipulate the private variables in this class directly, but must go through the getters and setters to retrieve and change data values, respectively. The accessor methods can have specific logic that protects the variable values, for example, a setter called `setHeight()` could enforce the rule that the value for height may not be less than 0. If the variable were not private, the caller would be responsible for knowing and complying with the rule.

In this example, the `Box` class contains no setters and getters for the `height` and `width` variables. These would be defined in the parent class, `Shape`, and are available to `Box`.

Line 29 is the closing bracket for the class definition.

Note

Java is a case-sensitive language. If you are transitioning from a non-case-sensitive language, it is useful to keep reminding yourself of this fact.

ANNOTATED USE OF THE BOX EXAMPLE CLASS

Code that uses a class such as `Box` demonstrates some other principles of the Java language. Consider the following example usage:

```
01: package shapes;
02:
03: public class TestBox  {
04:
05:   public static void main(String[] args) {
06:     // this creates an object called someBox
07:     Box someBox = new Box();
08:     someBox.setDepth(3);
09:     // getHeight() and setHeight() are from Shape
10:     // height shows the height variable from Box
11:     System.out.println (
12:       "The height of Shape is " + someBox.getHeight() +
13:       " and of someBox is " + someBox.height);
14:
15:     // getDepth and getVolume are from Box
16:     System.out.println (
17:       "The depth of someBox is " + someBox.getDepth() +
18:       " and the volume of someBox is " + someBox.getVolume());
19:    }
20: }
```

The output for the main method will be the following:

```
The height of Shape is 1 and of someBox is 4
The depth of someBox is 3 and the volume of someBox is 36
```

Lines 01–03 state the package and declare the class `TestBox`.

Main() Method

Lines 05–19 define a method called `main()`. This is a specially named method that executes automatically when the JVM runs the class from the command line (for example: `java.exe client.TestBox`). The `main()` method can contain any accessible code; in this case, it shows messages in the Java console (that displays in the JDeveloper Log window).

The keyword `static` indicates that `main()` is a *class method* that can be run without declaring an instance of the class. Normally, you need to create an object and then call the method by prefixing it with the object name. With static methods, you can still run the method (in this example, `main()`) without having to create an object from the class.

The `main()` method signature also includes an argument within the parentheses that follow the method name. This argument is typed as a String and is named “args.” The common parlance for expressing object type uses the datatype name, for example, “args is a String” or “args is [an instance] of the String type (or class).” (The words in square brackets are optional.)

The square brackets [ ] after args indicate that it is an array. *Arrays* are collections of similar objects. The String array args is used to pass any command-line arguments available when the class is executed. You can also use the expression “`String args[]`” to represent an array of strings.

Object Creation

Line 07 creates an object called someBox based on the `Box` class. The object is made from the `Box` class and, therefore, has the same variables (such as `someBox.width`) and methods (such as `someBox.getVolume()`) as the class. Line 07 accomplishes two tasks: it declares an object of type `Box` and creates the object by calling the constructor `Box()`. This line could also be expanded into the following two lines to separate the tasks:

```
Box someBox;
someBox = new Box();
```

Assign Values

Line 08 sets the value of the depth variable using the setDepth() method. This overwrites the value set in the constructor.

Console Output

Lines 11–13 output a message (using the `System.out.println()` method) that will be displayed in the Java console window. If you run a Java program from the command line, the Java console window will be the command-line window. If you run a Java program in JDeveloper, the message will appear in the Log window. You can concatenate literal strings (in quotes) and variables with the “+” operator regardless of type.

Variable and Accessor Usage

Line 12 references `getHeight()`, which is a method from the `Shape` class—the parent of `Box`, the class from which someBox is built. Since this method displays the value of the `height` variable in `Shape`, the value will be 1 (the default for that class). This demonstrates that you can call a method of a parent class from an object created from the subclass.

Line 13 references the `height` variable of `someBox` (which was built from the `Box` class). In this case, the `height` variable will be displayed as the default from the `Box` class (“4”). This output could be a bit confusing, and this system would probably not be used outside of demonstration purposes because the height of the parent class is set differently from the height of the subclass.

Lines 16–18 display the results of `someBox` method calls. Both `getDepth()` and `getVolume()` are declared in the `Box` class and will be output as 3 and 36, respectively.

Lines 19 and 20 close the method and class.

Note

When naming Java elements, you may use a combination of uppercase and lowercase letters, numbers, the underscore, and the dollar sign. However, you may not begin names with a number. There is no limit to the number of characters that you can use in a name.

OTHER JAVA LANGUAGE CONCEPTS

There are some other Java language concepts that were not demonstrated in the examples but that are useful to review.

THE CODE DEVELOPMENT AND DEPLOYMENT PROCESS

The typical Java development process, if you are not using an IDE such as JDeveloper, follows:

1. Write a source code file with a text editor, and name it using the name of the class that the file represents and a .java extension, for example, TestBox.java.
2. Compile the source code using the javac.exe executable (included in the Java SDK). If there are no syntax errors, the compiler creates a file with the same name and a class extension, for example, TestBox.class. This binary compiled file (called *bytecode*) is interpreted by the Java runtime engine when the program is executed. Java is compiled in this way, but it is considered an interpreted language. Java programs require a runtime interpreter—the JVM, a component of but often used synonymously with *Java runtime environment* or *JRE*.
3. Test the class file using the java.exe executable, the JVM (also included in the Java SDK). If the Java code is a Java application, the command line is as simple as the following example:
   ```
   java client.TestBox
   ```
4. Repeat steps 1–3 until the program performs as required.
5. Package the program file with the library files that it uses (libraries or classes declared in the import statements at the beginning of the program), and install the package on a client machine that has a Java runtime environment installed (containing the Java runtime engine—java.exe—and the base Java libraries such as java.lang.*).

Although an IDE such as JDeveloper automates many of these steps, the tasks are the same. Also, different types of Java programs have different requirements for the compile and runtime steps, but the concepts are the same. For example, working with JSP files requires the development of a .jsp file that is translated into a .java file and is compiled automatically into a .class file by a special Java runtime engine.

NAMING CONVENTIONS

Since Java is a case-sensitive language, its keywords must always be entered in the case they are designed. All keywords for the Java syntax are lower case. User-defined symbols such as class, member, and variable names can be used in any case, just so their usage is consistent within the code. For example, if you define a method "getWidth()," it must be used with the same case designation in subsequent code.

Even though Java does not enforce a particular case for user-defined symbols, there are generally recognized that Java programmers used. These standards can be summarized as follows:

- **Package and library** (archive file) names are all lower case.
- **Class names** are mixed case, with each word in the name initial capped. For example, a class that defines salary history would be called "SalaryHistory."
- **Method, variable (object), and exception names** are mixed case in the same way but the first character is lower case, for example, "usefulBox."
- **Constant (final variable) names** are all uppercase with underscores between words.

Generally names do not start with an underscore "_" because that syntax is often used for internal purposes.

CONTROL STATEMENTS

The idea of control statements is familiar to anyone who has written program code. Learning the Java control structures is usually just a matter of learning a different syntax (unless the other language is C++, which provided some of the syntax used in Java). This section reviews only the basic structures, since most structures are similar to those in other programming languages. You can refer to a standard Java language text to understand the variations and usage requirements for these control statements.

SEQUENCE

One of the main concepts of control statements is sequence, and Java code is executed in the order in which it appears in the file. The method that is executed first varies with the style of Java program; for example, a Java application executes the `main()` method first; a Java applet executes the `init()` method first; JavaServer Pages applications execute a `service()` method first. The commands within these methods are executed in the order in which they appear in the code file. As in other languages, calls to other methods execute the method and return to the line of code after the method call. The keyword `return` jumps out of the current method and returns control to the statement in the calling unit after that method was called (or to the command line if the command line was the caller).

CONDITIONAL BRANCHING

Java uses the statements `if-else` and `switch` to branch the code based upon a condition as follows:

```
class ShowQuarter {

  public static void main (String args[]) {
    int taxMonth = 10;
    String taxQuarter;

    if (taxMonth == 1 || taxMonth == 2 || taxMonth == 3) {
        taxQuarter = "1st Quarter";
    } else if (taxMonth == 4 || taxMonth == 5 || taxMonth == 6) {
        taxQuarter = "2nd Quarter";
    // more conditions would appear here
    } else {
        taxQuarter = "Not Valid";
    }
    System.out.println("Your current Tax Quarter is: " + taxQuarter );
  }
}
```

This is a branching statement that uses multiple `if` statements. The "`||`" symbol is a logical OR operator ("&&" is a logical AND). Logical conditions are enclosed in parentheses. The "= =" symbol is the equality comparison operator. Each condition is followed by a single statement or block of code. As mentioned before, it is a good idea to always define a block of code enclosed in curly brackets under the `if` statement.

The `switch` statement is an alternative to multiple `if` statements that test the same value. The following example could be used instead of the `if-then` example:

```
class ShowQuarter2 {
  public static void main (String args[]) {
    int taxMonth = 10;
    String taxQuarter;
    // The break statement jumps out of the conditional testing
    switch (taxMonth) {
      case 1:    case 2:    case 3:
        taxQuarter = "1st Quarter";
        break;
      case 4:    case 5:    case 6:
        taxQuarter = "2nd Quarter";
        break;
      // more conditions would appear here
      default:
        taxQuarter = "Not Valid";
    }    // end of the switch
    System.out.println("Your current Tax Quarter is: " + taxQuarter);
  }     // end of the main() method
}       // end of the class
```

ITERATION OR LOOPING

There are three loop statements: `for`, `while`, and `do-while`. The `for` loop controls loop iteration by incrementing and testing the value of a variable as shown in the following example:

```
class TestLoops
{
  public static void main (String args[]) {
    for (int i = 1; i <= 10; i++) {
       System.out.println("Loop 1 count is " + i);
    }
  }
}
```

The `while` and `do-while` loops test a condition at the start or end of the loop, respectively. Refer to a Java language reference, such as the Java Tutorial at java.sun.com, for more examples of loop structures.

EXCEPTION HANDLING

Exceptions can occur in the Java runtime environment when undefined conditions are encountered. To catch exceptions, you enclose the code in a block defined by the keywords `try`, `catch`, and (optionally) `finally` as in the following example:

```
public class TestException {
  public static void main(String[] args) {
    int numerator = 5, denominator = 0;
    int ratio;
    try {
      ratio = numerator / denominator ;
      System.out.println("The ratio is " + ratio);
    }
    catch (Exception e)  {
      // This shows an error message on the console
      e.printStackTrace();
    }
    finally {
      System.out.println("The end.");
    }
  }
}
```

If a `finally` block appears, it will be executed regardless of whether an exception is thrown. You may also raise an exception by using the keyword `throw` anywhere in the code.

Note

The preceding example shows how you can declare more than one variable (numerator and denominator in this example) of the same type on the same line.

VARIABLE SCOPE

Variables can be declared and objects can be created anywhere in a class and are available within the block in which they are declared. For example, the following code shows a variable, `currentSalary`, that is available throughout the `main()` method. Another variable, `currentCommission`, is available only within the `if` block in which it is declared. The last print statement will cause a compilation error because the variable is out of scope for that statement.

```
class TestScope {
  public static void main (String[] args) {
    int currentSalary = 0;
    if (currentSalary < 0) {
      int currentCommission = 10;
      System.out.println("No salary but the commission is " + currentCommission);
    }
    else {
      System.out.println("Salary but no commission.");
    }
    // This will cause a compilation error.
    System.out.println(currentCommission);
  }
}
```

Note

Although Java does not use the concept of a variable declaration section (such as the DECLARE section of PL/SQL), Java variables have the scope of their enclosing curly brackets. It is good programming practice to put all variable declarations at the beginning of their scope. For example, if you are declaring method-wide variables, place their declarations at the beginning of the method. If you are declaring variables with a class scope, place their declarations under the class declaration statement. Positioning the variable declarations in this way makes the code easier to read.

In addition to the scope within a block, variable scope is affected by where and how the variable is declared in the class file. The following example demonstrates these usages:

```
class ShowSalary {
  static int previousSalary = 0;
  int commission = 10;

  public static void main (String[] args) {
    int currentSalary = 100;
    if (currentSalary == 0) {
      System.out.println("There is only a commission.");
    }
    else {
      System.out.println("Current salary is " + currentSalary);
    }
    System.out.println("{Previous salary is " + previousSalary);

    // The following would cause a compile error.
    // System.out.println(commission);
  }
}
```

This example demonstrates three usages for variables—instance variables (`commission`), class variables (`previousSalary`), and local variables (`currentSalary`). Both instance variables and class variables are categorized as *member variables* because they are members of a class (not within a method or constructor). Member variables are available to any method within the class. Methods are also considered members of a class because the class is the container for the method.

INSTANCE VARIABLES

These variables are created outside of any method. In this example, the variable `commission` is an instance variable. It does not use the keyword `static` in the declaration and is not available to class methods (that are declared with the keyword `static`). Therefore, the variable `commission` is not available to the `main()` method in this example. Instance variables are available to objects created from the class. For example, you could create an object (instance) from this sample class using "`ShowSalary calcSalary = new ShowSalary();`", and the variable `calcSalary.commission` would be available. Each object receives its own copy of the class variable. Therefore, if you instantiate objects `salary1` and `salary2` from the `ShowSalary` class, `salary1.commission` and `salary2.commission` could contain different values.

CLASS VARIABLES

As with instance variables, class variables, such as `previousSalary` in this example, are declared outside of any method. The difference with class variables is that their declaration includes the `static` keyword and they are available to class methods that are also declared with the keyword `static`. The variable can be used without creating an instance of the class using the syntax "Classname.VariableName" (for example, ShowSalary.previousSalary). There is only one copy of the class variable regardless of the number of objects that have been created from the class. Therefore, if you create `salary1` and `salary2` from the `ShowSalary` class, the same variable `previousSalary` will be available from both objects (as `salary1.previousSalary` and `salary2.previousSalary`). If `salary2` changes the value of this variable, `salary1` will see that new value because there is only one variable. The following is an example that demonstrates this principle:

```
class TestShowSalary
{
  public static void main(String[] args)
  {
    ShowSalary salary1 = new ShowSalary();
    ShowSalary salary2 = new ShowSalary();
    //
    System.out.println("From salary1: " + salary1.previousSalary);
    salary2.previousSalary = 300;
    System.out.println("After salary2 changed it: " + salary1.previousSalary);
  }
}
```

The output from this program follows:

```
From salary1: 0
After salary2 changed it: 300
```

LOCAL VARIABLES

This variable usage is declared inside a method. In the sample `ShowSalary` class, `currentSalary` is a local variable because it is declared inside a method (`main()`). The variable is available only within the scope of that method.

CONSTANTS AND "FINAL"

A variable can be marked as `final`, which means that its value cannot change. Since you cannot change the value, you must assign a value when you declare the variable. This is similar to the idea of a constant in other languages. The following is an example of a final "variable." Final variable names use all uppercase characters by convention.

```
final int FEET_IN_MILE = 5280;
```

You can also mark methods as `final`, which means that you cannot override the method in a subclass. Thus, if class A has a `final` method b( ), and if class C extends A, then class C cannot override the inherited method b( ) in class C. For example:

```
final int getCommission() {
}
```

Classes may be marked with `final` to indicate that they cannot be subclassed. That is, no class may extend that class. For example:

```
class final CalcSalary {
}
```

Caution

The keyword `final` stops inheritance (subclassing) of final classes, and the overriding of final methods, but does not stop the overriding of a final variable (constant).

PRIMITIVE DATATYPES

Variable types fall into two categories: primitive and reference. *Primitive datatypes* can hold only a single value and cannot be passed by reference or pointers. Primitives are not based on classes and therefore have no methods. The primitive datatypes include `boolean` (for true and false values), several number types differentiated by the magnitude and precision of data they can represent (`byte`, `short`, `int`, `long`, `float`, `double`), and `char`.

A `char` is a single-byte number between 0 and 65,536 that is used to represent a single character in the Unicode international character set. A `char` datatype can be assigned in a number of ways as follows:

```
// decimal equivalent of the letter 'a'
char charDecimal = 97;
// using an actual character inside single quotes
char charChar = 'a';
// octal equivalent of the letter 'a'
char charOctal = '\141';
```

```
// Hex value for the letter 'a'
char charHex = 0x0061;
// Unicode (hex) value for the letter 'a'
char charUnicode = '\u0061';
```

> **Note**
>
> Assigning values to a byte requires surrounding the value in single quotes(‘). This is the only time that single quotes are used in Java. Double quotes (") are used to define a character string in code.

REFERENCE DATATYPES

Reference datatypes represent a memory location for a value or set of values. Since Java does not support pointers or memory addresses, you use the variable name to represent the reference. You can type an object using these reference datatypes, and the object instantiated in this way will have available to it the members in the class or referenced element (methods and variables). Reference datatypes may be arrays, interfaces, or classes.

ARRAYS

Collections are programmatic groups of objects or primitives. There are various types of collections available in Java, such as arrays, sets, dynamic arrays, linked lists, trees, hash tables, and key-value pairs (maps). Java provides a type of collection appropriately called Collection. This section discusses arrays. You will find information about the other categories of collections in Java language references.

Arrays in Java are collections of objects or primitives of similar type and may have one or more dimensions. Arrays are the only type of collection that can store primitive types. Elements within an array are accessed by indexes, which start at zero ([0]). To create an array, you declare it, allocate memory (size), and initialize the elements. These operations can be performed in two basic steps as shown here:

```
String animals[];
animals = new String[10];
```

The first line of code creates the array variable by adding a pair of square brackets to the variable name. The second line sets the size of the array (in this case, 10), which allocates memory, creates the object (animals), and initializes the elements. Arrays must be declared with a fixed number of members. This code could be condensed into the following line:

```
String animals[] = new String[10];
```

The next step is to store values in the array. In this example, the index numbers run from 0 to 9, and you store a value using that number as follows:

```
animals[3] = "Cat";
```

In Java, you can create arrays of arrays, more commonly known as *multi-dimensional arrays*. Since each array can be independently created, you can even create irregular combinations where array sizes vary within a given dimension. The more complex the array, the harder it is to keep track of, so moderation is advised. The following is a shorthand method for creating and assigning a two-dimensional array that stores pet owner names and the pet types:

```
class PetNames {
  public static main (String args[]) {
    String petFriends[ ][ ] = {
      {"George", "Snake", "Alligator" },
      {"Denise", "Butterfly"},
      {"Christine", "Tiger"},
      {"Robert", "Parrot", "Dove", "Dog", "Cat"}
    };
  }
}
```

INTERFACES

An interface is somewhat like a PL/SQL package specification because it lists method signatures and constants without any method code body. Classes that *implement* (or inherit) from the interface must include all methods in the interface. Interfaces are useful for providing a common type for a number of classes. For example, if you have a method that needs to return a type that will be manipulated by three different classes (that execute slightly differently), you can use an interface as the return type.

Each of the three classes would implement the interface and, therefore, the classes could be used in the same way by the method.

You can base a class on one or more interfaces, and this also provides a form of multi-parent inheritance. For example, if you had interfaces called SalaryHistory and CommissionHistory, you could define a class as follows:

```
public class HistoryAmounts extends CalcSalary implements SalaryHistory,
  CommissionHistory {
}
```

The `HistoryAmounts` class is a subclass of the `CalcSalary` class and will implement (provide method code declared in) the SalaryHistory and CommissionHistory interfaces. If you did not want to provide the code for the methods, you could declare `HistoryAmounts` as abstract (for example, `abstract class HistoryAmounts`). An *abstract class* cannot be instantiated but can be subclassed.

CLASSES

You can use any class to "type" an object (with the exception of abstract classes and classes with private constructors). The object becomes an instantiation of the class and has available to it the methods and member variables defined by the class. Therefore, classes can be used to create objects with the data and behavior characteristics defined in the class.

The Java language includes *wrapper* classes, such as `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Number`, that implement the corresponding primitive datatypes and are commonly used as types for variables. These classes include methods that act upon the objects, such as a method that converts a `Long` to an `int`. For example, using a Long object called longVar, the `int` value is `longVar.intValue()`. The next section describes wrapper classes further.

WRAPPER CLASSES

A *wrapper class* is a Java class that is created to control access to another Java class. A wrapper class can simplify the wrapped class' API, provide additional validation to the wrapped class, or change the access level of the wrapped class' methods. A wrapper class contains the wrapped class as an object member. For example, consider the following class:

```
package sample;
public class WrappedClass {
  public WrappedClass() {
  }
  int sumNumbers(int number1, int number2, boolean reallySum) {
    int result = 0;
    if (reallySum){
      result = number1 + number2;
    }
    return result;
  }
}
```

Note that the sumNumbers() method is not public so many classes will not be able to access it. In addition, sumNumbers() accepts three arguments. This class could be wrapped by the following wrapper class:

```
package sample;
public class WrapperClass {
  private final WrappedClass contents;
  public WrapperClass(WrappedClass newContents) {
    contents = newContents;
  }
  public int sumNumbers(int number1, int number2) {
    return contents.sumNumbers(number1, number2, true);
  }
}
```

Individual instances of WrappedClass are passed to the constructor and are stored in the `contents` field. The class exposes the API of WrappedClass, changing the number of arguments to `sumNumbers` and exposing it publicly. WrapperClass is said to wrap WrappedClass. In addition, instances of WrapperClass can be said to wrap the instances of WrappedClass that they contain.

CHARACTER STRING CLASSES

Two commonly used classes are `String` and `StringBuffer`.

String Class

A String object can be declared and assigned a set of characters as follows:

```
String stringVar = "This is a Java test string";
```

Objects built from `String` can take advantage of the methods in the `String` class. The methods provide functions to create strings from literals, chars, char arrays, and other string reference objects. The following Java Strings store the value "Java" by assigning a value to one String object and concatenating that object to another string using the `concat()` method that is part of the `String` class.

```
String startingLetters = "Ja";
String newString = startingLetters.concat("va");
```

Tip

To view the Javadoc for a basic Java class such as `String`, type "String" into the Code Editor (or find the class name "String" in the file), place the cursor in the word, and select Quick Javadoc from the right-click menu. A window will pop up and display the documentation heading for that class. If you need to look at the entire Javadoc topic for the class including methods and other details, select Go to Javadoc. A window will appear with the full Javadoc topic containing methods and constants available to objects built from the class.

You can compare, concatenate, change the case of, find the length of, extract characters from, search, and modify strings. Strings in Java are considered *immutable,* that is, they cannot be changed. Whenever you alter a string through a string operation, the result is a new String object that contains the modifications. The old String object is no longer accessible because the object name points to the new object just created. You can take advantage of the overloading of the concatenation operator "+" to assign string values from number literals as in the following example:

```
// This assigns "The age is 235" to age.
String age = ("The age is " + 2 + 35);
// This assigns "The age is 37" to age.
String age = "The age is " + (2 + 35);
```

Note

In Java, the method `substring(int startIndex, int endIndex)` returns a portion of a string from the startIndex to the (endIndex –1). As with arrays, the index numbers start with zero. The following example will assign "This is a Java" to the newString variable:

```
String baseString = "This is a Java string";
String newString = baseString.substring(0, 15);
```

StringBuffer Class

`StringBuffer` is a sister class to `String` and represents character sequences that are *mutable,* that is, they can change size and/or be modified. What this means to the developer is that methods such as `append()` and `insert()` are available to modify a `StringBuffer` variable without creating a new object. Thus, the `StringBuffer` class is best if the character sequences being stored may need to be changed. The `String` class is good if the character sequence will not need to be changed.

The following shows an example usage of the `append()` method available to `StringBuffer`:

```
class StringAppend {
  public static void main (String args[]) {
    StringBuffer stringBuff = new StringBuffer("A string");
    stringBuff = stringBuff.append(" is added");
    System.out.println(stringBuff.toString());
  }
}
```

You could also append to a String variable using the String concat() method but, due to the immutable nature of String objects, that method would create a new String variable with the same name as the old variable. There is overhead and a bit of memory required by additional objects, so `StringBuffer` is better for concatenation.

DATATYPE MATCHING

Java is a semi-strongly typed language—every variable has a type, and every type is strictly defined. Type matching is strictly enforced in cases such as the following:

- The arguments passed to a method must match the argument types in the method's signature.
- Both sides of an assignment expression must contain the same datatype.
- Both sides of a Boolean comparison, such as an equality condition, must use matching datatypes.

There are few automatic conversions of one variable type to another. In practice, Java is not as restrictive as you might think, since most built-in methods are heavily overloaded (defined for different types of arguments). For example, you can combine strings, numbers, and dates using a concatenation operator (+) without formal variable type conversion, because the concatenation operator (which is, technically speaking, a base-language method) is overloaded.

In addition to overloading, an exact match is not always required, as shown in the following example:

```
public class TestCast {
  public static void main (String args[]) {
    byte smallNumber = 10;
    int largeNumber;
    largeNumber = smallNumber * 5;
    System.out.println("largeNumber is " + largeNumber);
    // smallNumber = largeNumber;
    smallNumber = (byte) largeNumber;
    System.out.println("smallNumber is " + smallNumber);
  }
}
```

The assignment starting with `largeNumber` assigns the `byte` variable `smallNumber` (times five) to the `int` variable `largeNumber`. In this case, there is a datatype mismatch (`byte` times `int`), but the code will compile without a problem because you are storing a smaller type (`byte`) in a larger type (`int`).

Rounding errors can occur from misuse of datatypes. The following shows an example of one of these errors:

```
int numA = 2;
int numB = 3;
System.out.println(numB/numA);
```

Although the division of these two variables results in “1.5,” the print statement shows “1” because the output of the operator is the same type as the variables: `int`.

CASTING VARIABLES

In the preceding example class (`TestCast`), the statement that is commented out will generate a compilation error because it tries to store a larger-capacity datatype (`int`) in a smaller-capacity datatype (`byte`), even though the actual value of 50 is within the range of the `byte` datatype.

You can *cast* (explicitly convert) one type to another by preceding the variable name with the datatype in parentheses. The statement after the commented lines in this example corrects the typing error by casting `largeNumber` as a `byte` so that it can be stored in the `smallNumber` `byte` variable. The disadvantage of casting is that the compiler will not catch any type mismatch as it will for explicit, non-cast types. Another disadvantage with older JDKs is performance—the cast takes time; the more recent JDKs minimize or eliminate this overhead.

CASTING OBJECTS

You can also cast objects to classes and interfaces so that you can take advantage of the methods defined for the classes and interfaces. Casting allows you to match objects of different, but related, types. For example, the `Integer` class is a subclass of the `Number` class. The following code creates an object called `numWidth` as a Number cast from an Integer object. The cast is required because the `Number` class is abstract and you cannot instantiate it. The code then creates an object called `width` and assigns it the value of `numWidth`. Since `numWidth` is a Number object, which is less restrictive (or wider), this code needs to cast it to `Integer` match the new object.

```
Number numWidth = (Number) new Integer(10);
Integer width = (Integer) numWidth;
```

If the example were reversed so that the Integer was created first and the Number second, casting would not be required. Consider the following example:

```
Integer width2 = new Integer(10);
Number numWidth2 = width2;
```

Explicit casting of the Integer (`width2`) into the Number (`numWidth2`) is not required because Number is less restrictive (or wider). Casting to interfaces works in the same way.

CASTING LITERALS

Floating-point literals (such as the value 34.5) default to the `double` datatype. If you want to assign a datatype of `float` to the literal, you must add an "F" suffix (for example, 34.5F). Alternatively, you may cast the literal using an expression such as `(float) 34.5`. Some examples for assigning datatypes to literals follow. ("L" is used for a `long` datatype, and "F" is used for a `float` datatype. It does not matter whether the suffix letters are upper- or lowercase.)

```
long population = 1234567890123456789L;
int age = 38;
float price = 460.95F;
float price = (float) 460.95;
double area, length = 3.15, width = 4.2;
area = length * width;
```

Non-floating literals (such as 38 in the example) will be assigned an `int` datatype. This can make an expression such as the following fail at compile time:

```
smallNumber = 5 + smallNumber;
```

The right side of the expression (`5 + smallNumber`) is assigned an `int` type because "5" is an `int` and `smallNumber` is implicitly cast up to match it. The right side does not match the left side because `smallNumber` is a `byte`. Explicit casting will solve the problem if you apply the cast to the entire side of the expression as follows:

```
smallNumber = (byte) (5 + smallNumber);
```

CONCLUSION

To combat confusion and fear about Java, PL/SQL enthusiasts only need a bit of knowledge about object orientation and basic Java concepts and elements. This paper has provided an introduction to those topics. Further study and some real-world experience will make you as expert in Java as you are in SQL and PL/SQL.

ABOUT THE AUTHOR

Peter Koletzke is a technical director and principal instructor for the Enterprise e-Commerce Solutions practice at Quovera, in Mountain View, California, and has 25 years of industry experience. Peter has presented at various Oracle users group conferences more than 220 times and has won awards such as Pinnacle Publishing's Technical Achievement, Oracle Development Tools Users Group (ODTUG) Editor's Choice, ECO/SEOUC Oracle Designer Award, ODTUG Volunteer of the Year, and NYOUG Editor's Choice. He is an Oracle Certified Master, Oracle ACE Director, and coauthor of the Oracle Press Books: *Oracle JDeveloper 10*g *for Forms & PL/SQL Developers* (with Duncan Mills); *Oracle JDeveloper 10*g *Handbook* and *Oracle9*i *JDeveloper Handbook* (with Dr. Paul Dorsey and Avrom Roy-Faderman); *Oracle JDeveloper 3 Handbook*, *Oracle Developer Advanced Forms and Reports*, *Oracle Designer Handbook, 2nd Edition*, and *Oracle Designer/2000 Handbook* (all with Dr. Paul Dorsey).