Module 2

Object-Oriented Programming

Objectives

Upon completion of this module, you should be able to:

- Define modeling concepts: *abstraction*, *encapsulation*, and *packages*
- Discuss why you can reuse Java technology application code
- Define *class*, *member*, *attribute*, *method*, *constructor*, and *package*
- Use the access modifiers `private` and `public` as appropriate for the guidelines of encapsulation
- Invoke a method on a particular object
- Use the Java technology API online documentation

This module is the first of three modules that describe the object-oriented (OO) paradigm and the object-oriented features of the Java programming language.

Relevance

Discussion – The following questions are relevant to the material presented in this module:

- What is your understanding of software analysis and design?

  _____

  _____

  _____

- What is your understanding of design and code reuse?

  _____

  _____

  _____

- What features does the Java programming language possess that make it an object-oriented language?

  _____

  _____

  _____

- Define the term *object-oriented.*

  _____

  _____

  _____

Software Engineering

Software engineering is a difficult and often unruly discipline. For the past half-century, computer scientists, software engineers, and system architects have sought to make creating software systems easier by providing reusable code. Figure 2-1 shows a brief history of software engineering.

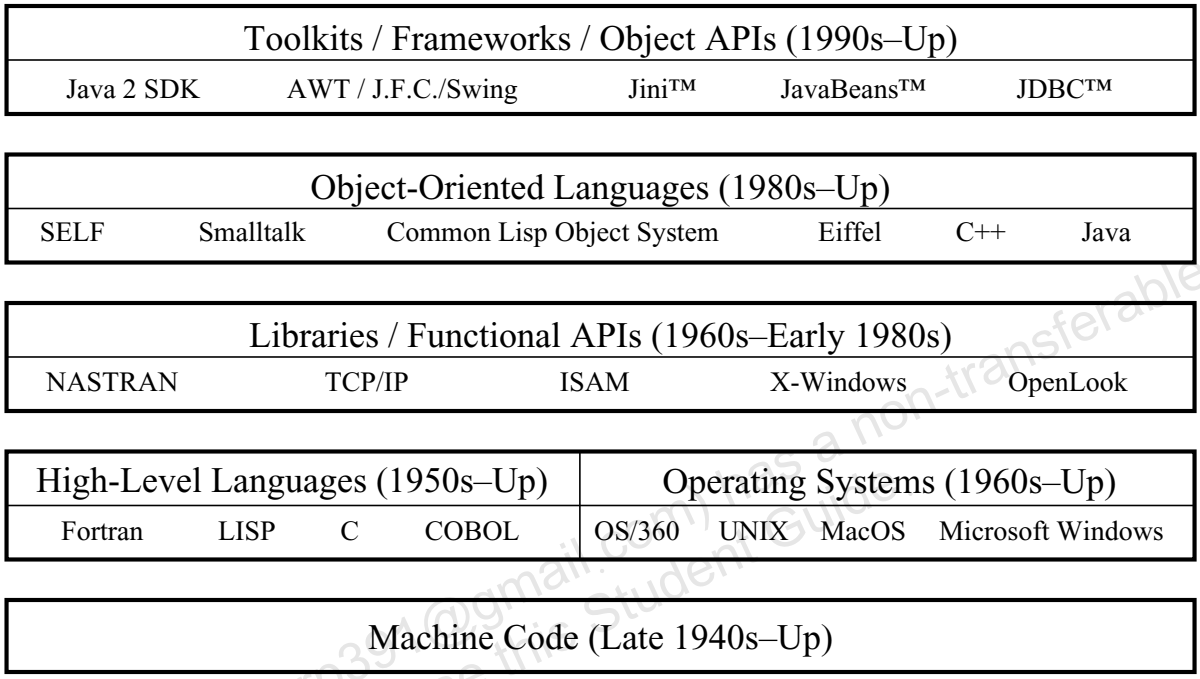

Figure 2-1 Brief History of Software Engineering

At first, they created computer languages to conceal the complexity of the machine language and added callable operating system procedures to handle common operations, such as opening, reading, and writing to files.

Other developers grouped collections of common functions and procedures into libraries for anything from calculating structural loads for engineering (NASTRAN), writing character and byte streams between computers on a network (TCP/IP), accessing data through an indexed sequential access method (ISAM), and creating windows, graphics, and other GUI widgets on a bit-mapped monitor (X-Windows and Open Look).

Many of these libraries manipulated data in the form of *open* record data-structures, such as the C language `struct`. The main problem with record structures is that the library designer cannot hide the implementation of the data used in the procedures. This makes it difficult to modify the implementation of the library without affecting the client code because that code is often tied to the particular details of the data structures.

By the late 1980s, object-oriented programming (OOP) became popular with C++. One of the greatest advantages of OOP was the ability to hide certain aspects of a library's implementation so that updates do not affect client code (assuming the interfaces do not change). The other important advantage is that procedures were associated with the data structure. The combination of data attributes and procedures (called methods) are known as a *class*.

Today's equivalent of function libraries are class libraries or toolkits. These libraries provide classes to perform many of the same operations as functional libraries but, with the use of subclassing, client programmers can easily extend these tools for their own applications. Frameworks provide APIs that different vendors can implement to allow you to choose the amount of flexibility and performance suitable to your applications.

Java technology is a platform that is continuously extended by new APIs and frameworks, such as Java Foundation Classes/Swing (J.F.C./Swing) and other J.F.C. technology, JavaBeans architecture (Java technology's component architecture), and the Java DataBase Connectivity API (JDBC API). The list of Java APIs is long and growing.

The Analysis and Design Phase

There are five primary workflows in a software development project: Requirement Capture, Analysis, Design, Implementation, and Test. They are all important, however, you must ensure that you schedule enough time for the analysis and design phases.

During the Analysis phase, you define *what* the system is supposed to accomplish. You do this by defining the set of actors (users, devices, and other systems that interact with the proposed system) and activities that the proposed system must accommodate. Also, the Analysis phase must identify the *domain objects* (both physical and conceptual) that the proposed system will manipulate and the behaviors and interactions among these objects. These behaviors implement the activities that the proposed system must support. The description of the activities should be detailed enough to create baseline criteria for the Test phase.

During the Design phase, you define *how* the system will achieve its goals. In this phase, you create a model of the actors, activities, objects, and behaviors for the proposed system. For this class, you use the Unified Modeling Language (UML) as your modeling tool.

Note – UML is a large and complex language. You use only a small portion of it. Appendix B is a reference to the UML elements that are used in this course. It also shows you how to implement Java technology code from a UML class diagram.

Analysis and Design Example

This module uses the example of a shipping company. You assume a simple set of requirements:

- The software must support a single shipping company.
- The shipping company maintains a fleet of vehicles that transport boxes.
- The weight of the boxes is the only important factor in loading a vehicle.
- The shipping company owns two types of vehicles: trucks and river barges.
- Boxes are weighed on scales that measure in kilograms; however, the algorithms for calculating vehicle engine power require the total vehicle load to be measured in newtons.

Note – A *newton* is a measure of force (or weight) that is equivalent to 9.8 times the mass of the object in kilograms.

- You use a GUI to keep track of adding boxes to vehicles.
- You must generate several reports from the fleet records.

From these requirements, you can create a high-level design:

- The following objects must be represented in the system: a company and two types of vehicles.
- A company is an aggregate of multiple vehicle objects.
- Other functional objects exist: several reports and GUI screens.

Abstraction

Software design has moved from low-level constructs, such as writing in machine code, toward much higher levels. There are two interrelated forces that guided this process: simplification and abstraction. Simplification was at work when early language designers built high-level language constructs, such as the `IF` statements and `FOR` loops, out of raw machine codes. Abstraction is the force that hides private implementation details behind public interfaces.

The concept of abstraction led to the use of subroutines (functions) in high-level languages and to the pairing of functions and data into objects. At higher levels, abstraction led to the development of frameworks and APIs.

Classes as Blueprints for Objects

Just as a draftsman can create a blueprint for a device that can be used many times to construct actual devices, a *class* is a software blueprint that you can use to *instantiate* (that is, create) many individual objects. A class defines the set of data elements (*attributes*) that define the objects, as well as the set of behaviors or functions (called *methods*) that manipulate the object or perform interactions between related objects. Together, attributes and methods are called *members*. For example, a vehicle object in a shipping application must keep track of its maximum and current load along with methods for adding a box (with a certain weight) to the vehicle.

The Java technology programming language supports three key features of OOP: encapsulation, inheritance, and polymorphism.

Note – Encapsulation is covered in ‘‘Encapsulation’’ on page 2-15. Inheritance and polymorphism are described in Module 6.

Declaring Java Technology Classes

The Java technology class declaration takes the following form:

```
<modifier>* class <class_name> {
  <attribute_declaration>*
  <constructor_declaration>*
  <method_declaration>*
}
```

The <*class_name*> can be any legal identifier, and it is the name of the class being declared. There are several possible <*modifier*> keywords, but for now, use only `public`. This declares that the class is accessible to the universe. The body of the class declares the set of data attributes, constructors, and methods associated with the class. Code 2-1 shows an example class declaration.

Code 2-1 Example Class Declaration

```
1   public class Vehicle {
2     private double maxLoad;
3     public void setMaxLoad(double value) {
4       maxLoad = value;
5     }
6   }
```

Declaring Attributes

The declaration of an object attribute takes the following form:

<modifier> <type> <name> [* **=** *<initial_value>]*;

Example:

```
1   public class Foo {
2     private int x;
3     private float y = 10000.0F;
4     private String name = "Bates Motel";
5   }
```

The *<name>* can be any legal identifier, and it is the name of the attribute being declared. There are several possible values for *<modifier>*, but for now, use either `public` or `private`. The `private` keyword declares that the attribute is accessible only to the methods within this class. The *<type>* of the attribute can be any primitive type (`int`, `float`, and so on) or any class.

Declaring Methods

To define methods, the Java programming language uses an approach that is similar to other languages, particularly C and C++. The declaration takes the following basic form:

```
<modifier>* <return_type> <name> ( <argument>* ) {
  <statement>*
}
```

The <*name*> can be any legal identifier, with some restrictions based on the names that are already in use.

The <*modifier*> segment is optional and can carry a number of different modifiers, including (but not limited to) `public`, `protected`, and `private`. The `public` access modifier indicates that the method can be called from other code. The `private` method indicates that a method can be called only by the other methods in the class. The `protected` method is described later in this course.

The <*return_type*> indicates the type of value returned by the method. If the method does not return a value, it should be declared `void`. Java technology is rigorous about returned values, and if the declaration states that the method returns an `int`, for example, then the method must return an `int` from all possible return paths (and can be invoked only in contexts that expect an `int` to be returned). Use the `return` statement within a method to pass back a value.

The <*argument*> list allows argument values to be passed into a method. Elements of the list are separated by commas, while each element consists of a type and an identifier.

Example

Code 2-2 shows two methods for the `Dog` class. The method `getWeight` returns the `weight` data attribute; it uses no parameters. A value is returned from a method using the `return` statement (Line 4). The method `setWeight` modifies the `weight` value with the parameter `newWeight`; it does not return any value. This method uses a conditional statement to restrict the client code from setting the dog's weight to a negative number or zero.

Code 2-2 Example Methods

```
1   public class Dog {
2     private int weight;
3     public int getWeight() {
4       return weight;
5     }
6     public void setWeight(int newWeight) {
7       if ( newWeight > 0 ) {
8         weight = newWeight;
9       }
10    }
11  }
```

Accessing Object Members

In the previous example, you saw the following line of code in the `TestDog.main` method:

```
d.setWeight(42);
```

This line of code tells the `d` object (actually a variable, *d*, holding a reference to an object of type `Dog`) to execute its `setWeight` method. This is called *dot notation*. The dot operator enables you to access non-private attribute and method members of a class.

Within the definition of a method, you do not need to use the dot notation for accessing local members. For example, the `setWeight` method of the `Dog` class does not use the dot notation to access the `weight` attribute.

Code 2-3 demonstrates the behavior of the `Dog` methods. When the `Dog` object is created the `weight` instance variable is initialized to `0`. Therefore, the `getWeight` method returns `0`. Line 6 of this code sets the weight to `42`; this is a valid argument and `setWeight` method sets the `weight` variable. However, setting the weight to `-42` (Line 9) is illegal and `setWeight` method does not alter the `weight` variable.

Code 2-3 Example Invocation of Methods

```
1   public class TestDog {
2     public static void main(String[] args) {
3       Dog d = new Dog();
4       System.out.println("Dog d's weight is "
5                          + d.getWeight());
6       d.setWeight(42);
7       System.out.println("Dog d's weight is "
8                          + d.getWeight());
9       d.setWeight(-42);
10      System.out.println("Dog d's weight is "
11                         + d.getWeight());
12    }
13  }
```

The output is:

```
Dog d's weight is 0
Dog d's weight is 42
Dog d's weight is 42
```

Information Hiding

Suppose that you have a `MyDate` class that includes the attributes: `day`, `month`, and `year`. Figure 2-2 shows a class diagram of a possible implementation of the `MyDate` class.

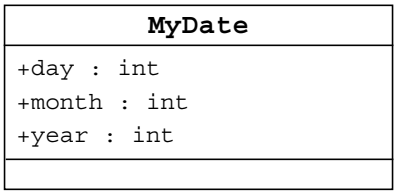

Figure 2-2 UML Class Diagram of the `MyDate` Class

A simple implementation allows direct access to these data attributes; for example:

```
public class MyDate {
  public int day;
  public int month;
  public int year;
}
```

Client code then accesses the attributes directly and makes mistakes; for example (`d` refers to a `MyDate` object):

```
d.day = 32;
// invalid day

d.month = 2; d.day = 30;
// plausible but wrong

d.day = d.day + 1;
// no check for wrap around
```

To solve the problem, hide the data attributes by making them private and supply retrieval access methods, `get`*`Xyz`*`()`, which are often called *getters*, and storage access methods, `set`*`Xyz`*`()`, which are often called *setters*. Figure 2-3 shows another UML diagram of the `MyDate` class that hides the instance variables behind getter and setter methods.

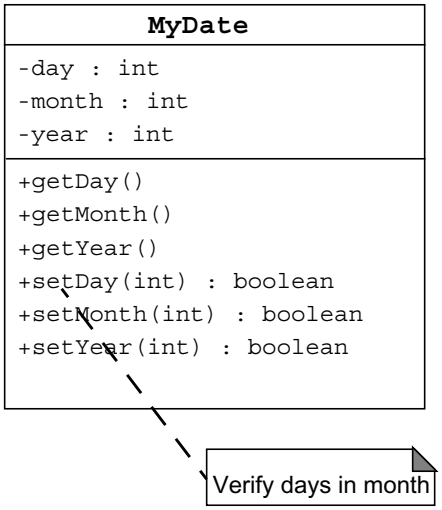

Figure 2-3 Hiding the Instance Variables of the `MyDate` Class

These methods allow the class to modify the internal data, but more importantly, to verify that the requested changes are valid. For example:

```
MyDate d = new MyDate();

d.setDay(32);
// invalid day, returns false

d.setMonth(2); d.setDay(30);
// plausible but wrong, setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around needs to occur
```

Encapsulation

Encapsulation is the methodology of hiding certain elements of the implementation of a class but providing a public interface for the client software. This is an extension of information hiding because the information in the data attributes is a significant element of a class's implementation.

For example, the programmer for the `MyDate` class might decide to reimplement the internal representation of a date as the number of days since the beginning of some epoch. This could make date comparisons and calculating date intervals easier. Because the programmer encapsulated the attributes behind a public interface, the programmer can make this change without affecting the client code. Figure 2-4 shows this variation on the `MyDate` class.

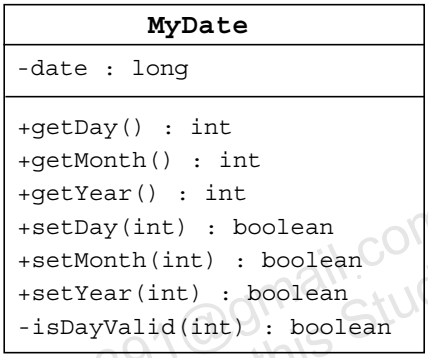

Figure 2-4 Encapsulation Provides Data Representation Flexibility

Declaring Constructors

A constructor is a set of instructions designed to initialize an instance. Parameters can be passed to the constructor in the same way as for a method. The basic declaration takes the following form:

```
[<modifier>] <class_name> ( <argument>* ) {
  <statement>*
}
```

The name of the constructor must always be the same as the class name. If present, the only valid modifiers (*<modifier>*) for constructors are `public`, `protected`, and `private`.

The *<argument>* list is the same as for method declarations.

Note – Constructors are not methods. They do not have return values and are not inherited.

For example:

```
1   public class Dog {
2     private int weight;
3
4     public Dog() {
5       weight = 42;
6     }
```

The `Dog` class has a single instance variable `weight`. The constructor (with no parameters) initializes `weight` to `42`.

Constructors can also be declared with parameters. This is discussed later in this course.

The Default Constructor

Every class has at least one constructor. If you do not write a constructor, the Java programming language provides one for you. This constructor takes no arguments and has an empty body.

The default constructor enables you to create object instances with `new Xyz()`; otherwise, you must provide a constructor for every class.

Note – If you add any constructor declaration to a class that previously had no explicit constructors, you lose the default constructor. From that point, unless the constructor you wrote takes no arguments, calls to `new Xyz()` cause compiler errors.

Source File Layout

A Java technology source file takes the following form:

```
[<package_declaration>]
<import_declaration>*
<class_declaration>+
```

Note – The plus (+) indicates one or more. To be meaningful, a source file must contain at least one class definition.

The order of these items is important. That is, any import statements must precede all class declarations and, if you use a package declaration, it must precede both the classes and imports.

The name of the source file must be the same as the name of the public class declaration in that file. A source file can include more than one class declaration, but *only one* class can be declared public. If a source file contains no public class declarations, then the name of the source file is not restricted. However, it is good practice to have one source file for every class declaration, and the name of the file is identical to the name of the class.

For example, the file `VehicleCapacityReport.java` should look like the following:

```
1   package shipping.reports;
2
3   import shipping.domain.*;
4   import java.util.List;
5   import java.io.*;
6
7   public class VehicleCapacityReport {
8      private List  vehicles;
9      public void generateReport(Writer output) {
10       // code to generate the report
11     }
12  }
```

Software Packages

Most software systems are large. It is common to group classes into packages to ease the management of the system. UML includes the concept of packages in its modeling language. Packages can contain classes as well as other packages that form a hierarchy of packages. An example package structure is illustrated in Figure 2-5.

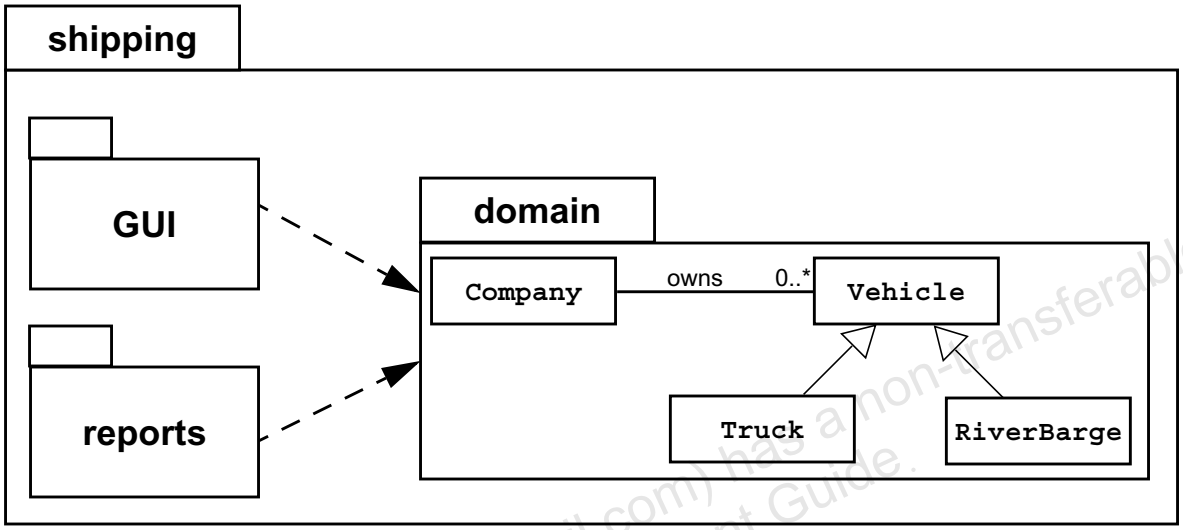

Figure 2-5 An Example UML Diagram of Java Packages

There are many ways to group classes into meaningful packages. There is no right or wrong way; but a common technique is to group classes into a package by semantic similarity.

For example, a shipping software system could contain a set of domain objects (such as the company and vehicles, boxes, destinations, and so on), a set of reports, and a set of GUI panels that are used to create the main data entry application. The `GUI` and `reports` subsystems depend on the `domain` package. UML packages can be useful for modeling subsystems or other groupings according to your need. All of these packages are contained in the top-level package called `shipping`.

The package Statement

The Java technology programming language provides the package statement as a way to group related classes. The package statement takes the following form:

```
package <top_pkg_name>[.<sub_pkg_name>]*;
```

You can indicate that classes in a source file belong to a particular package by using the package statement; for example:

```
1   package shipping.domain;
2
3   // Class Vehicle of the 'domain' sub-package within
4   // the 'shipping' application package.
5   public class Vehicle {
6        ...
7   }
```

The package declaration, if any, must be at the beginning of the source file. You can precede it with white space and comments, but nothing else. Only one package declaration is permitted, and it governs the entire source file. If a Java technology source file does not contain a package declaration, then the classes declared in that file belongs to the unnamed (default) package.

Package names are hierarchical and are separated by dots. It is usual for the elements of the package name to be entirely lower case. However, the class name usually starts with a capital letter, and you can capitalize the first letter of each additional word to distinguish words in the class name. These naming conventions and others are described in "Java Programming Language Coding Conventions" on page 3-21.

Note – If a package statement is not included in the file, then all classes declared in that file *belong* to the default package (that is, a package with no name).

The import Statement

The import statement takes the following form:

```
import <pkg_name>[.<sub_pkg_name>].<class_name>;
```

or

```
import <pkg_name>[.<sub_pkg_name>].*;
```

When you want to use packages, use the import statement to tell the compiler where to find the classes. In fact, the package name (for example, shipping.domain) forms part of the name of the classes within the package. You could refer to the Company class as shipping.domain.Company throughout, or you could use the import statement and just the class name Company.

Note – The import statements must precede all class declarations.

The following is a file fragment that uses the import statement.

```
1   package shipping.reports;
2
3   import shipping.domain.*;
4   import java.util.List;
5   import java.io.*;
6
7   public class VehicleCapacityReport {
8        private Company  companyForReport;
9     ...
10  }
```

When you use a package declaration, you do not need to import the same package or any element of that package. Remember that the import statement is used to make classes in other packages accessible to the current class.

The import statement specifies the class to which you want access. For example, if you want only the Writer class (from the java.io package) included in the current name space, then you would use:

```
import java.io.Writer;
```

If you want access to all classes within a package, use "*." For example, to access all classes in the java.io package, use:

```
import java.io.*;
```

Note – The import statement allows you to use the short class names in your source program, nothing more. An import statement does not cause the compiler to load anything additional into working memory. In this respect, the import statement is quite different from the #include statement in C or C++. The import statement, whether or not it uses the wild card (*), does not have any effect on the output class file nor does it have any effect on runtime performance. It is also very unlikely that any form of import statement will cause any difference in compilation performance.

Directory Layout and Packages

Packages are stored in a directory tree containing a branch that is the package name. For example, the `Company.class` file should exist in the directory structure shown in Figure 2-6.

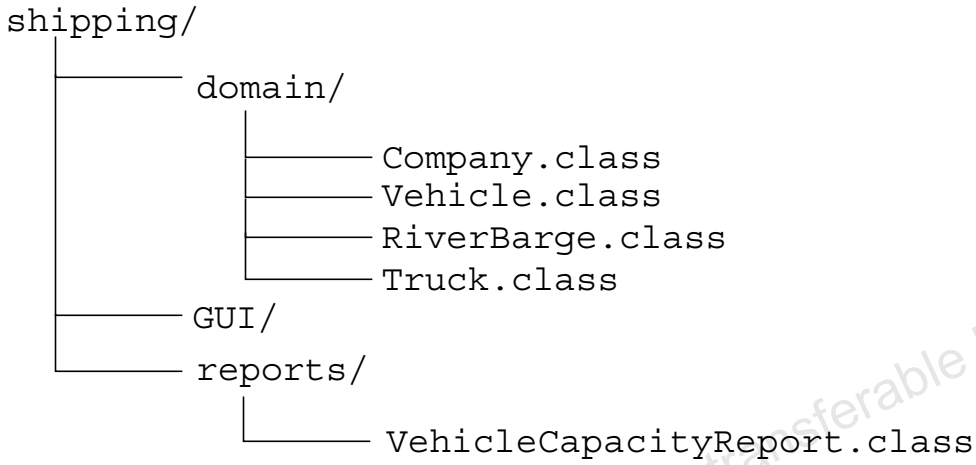

Figure 2-6 Example Class Package Directory Structure

Development

It is common to be working on several development projects at once. There are many ways to organize your development files. This section describes one such technique.

Figure 2-7 demonstrates an example development directory hierarchy for a development project. The important element of this hierarchy is that the source files of each project are separated from the compiled (`.class`) files.

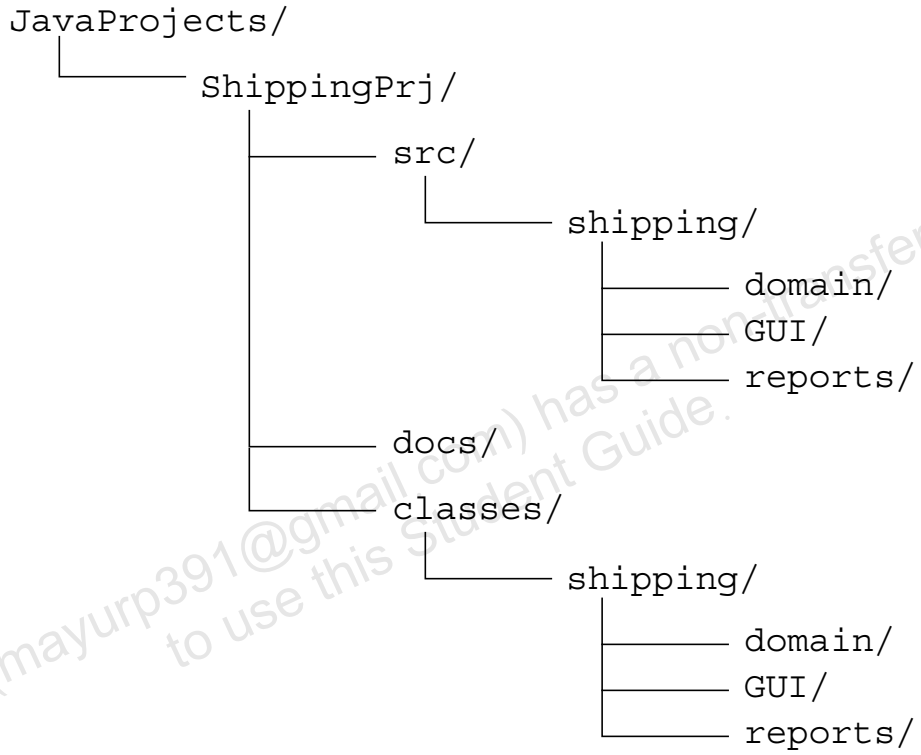

Figure 2-7 Example Project Development Directory Structure

Compiling Using the `-d` Option

Normally, the Java compiler places the class files in the same directory as the source files. You can reroute the class files to another directory using the `-d` option of the `javac` command. The simplest way to compile files within packages is to be working in the directory one level above the beginning of the package. (In this example, the `src` directory.)

To compile all the files within the `shipping.domain` package and have the compiled classes end up in their correct package directory under `ShippingPrj/class/`, type the following:

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

Deployment

You can deploy an application on a client machine without manipulating the user's `CLASSPATH` environment variable. Usually, this is best done by creating an *executable Java archive (JAR) file*. To create an executable JAR file, you must create a temporary file that indicates the class name that contains your `main` method, like this:

```
Main-Class: mypackage.MyClass
```

Next, build the JAR file as normal except that you add an additional option so that the contents of this temporary file are copied into the `META-INF/MANIFEST.MF` file. Do this using the "`m`" option, like this:

```
jar cmf tempfile MyProgram.jar
```

Finally, the program can be run simply by executing a command like this:

```
java -jar /path/to/file/MyProgram.jar
```

Note – On some platforms, simply double clicking the icon for an executable JAR file is sufficient to launch the program.

Library Deployment

Sometimes you need to deploy library code in a JAR file. In such situations it is possible to copy the JAR file into the `ext` subdirectory of the `lib` directory in the main directory of the JRE. Be careful when you do this, because if you deploy classes in this way they are usually granted full security privileges, and might cause runtime problems if any classes have naming conflicts with other classes in the core JDK or that have been installed in this way.

Terminology Recap

The following describes some of the terms introduced in this module:

- Class – A way to define new types of objects in the Java programming language. The class can be considered as a blueprint, which is a model of the object that you are describing.
- Object – An actual instance of a class. An object is what you get each time you instantiate a class using `new`. An object is also known as an *instance*.
- Attribute – A data element of an object. An attribute stores information for an object. An attribute is also known as a *data member*, an *instance variable*, or a *data field*.
- Method – A functional element of an object. A method is also known as a *function* or a *procedure*.
- Constructor – A *method-like* construct used to initialize (or build) a new object. Constructors have the same name as the class.
- Package – A grouping of classes, subpackages, or both.

Using the Java Technology API Documentation

A set of HTML files document the supplied API. The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. When you select a particular package hotlink, the classes that are members of that package are listed. Selecting a class hotlink from a package page presents a page of information about that class. Figure 2-8 shows one such class.

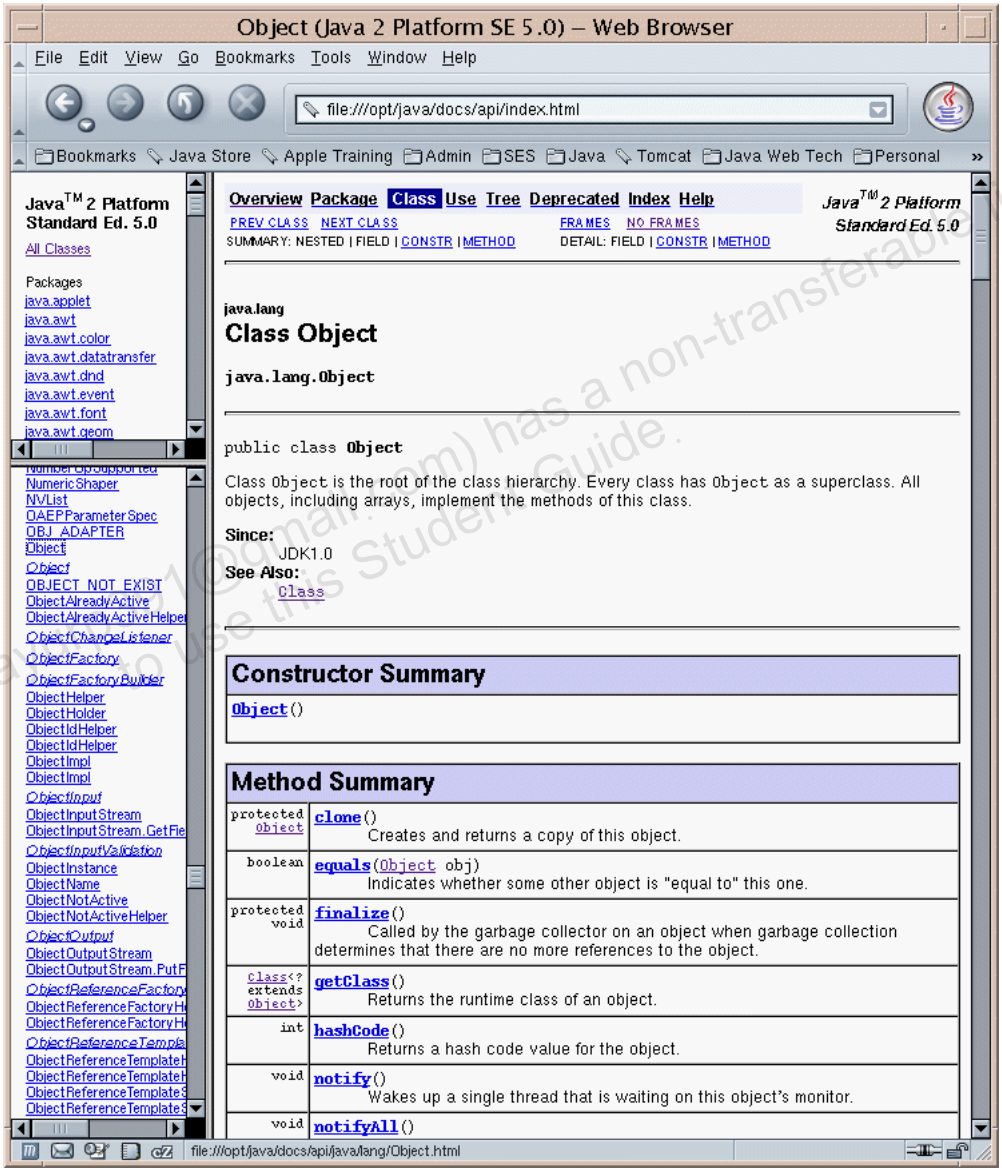

Figure 2-8 Java Technology API Documentation

The main sections of a class document include the following:

- The class hierarchy
- A description of the class and its general purpose
- A list of attributes
- A list of constructors
- A list of methods
- A detailed list of attributes with descriptions
- A detailed list of constructors with descriptions and formal parameter lists
- A detailed list of methods with descriptions and formal parameter lists