CHAPTER 2

Java, Object-Oriented Analysis and Design, and UML

IN THIS CHAPTER

As mentioned in Chapter 1, to be successful in today's ever-changing business climate, software development must follow an approach that is different from the big-bang approach. The big-bang approach, or waterfall model, offers little risk aversion or support for modification of requirements during development. The waterfall model forces the project team to accept insurmountable risks and create software that usually doesn't approximate the original vision of the project sponsors.

This chapter looks at Java as an enterprise solution for constructing and implementing industrial-strength applications that will better approximate what the sponsors intended. Java is a language that not only supports object-oriented concepts, but also formally acknowledges many constructs not formally found in other object languages, such as the interface. This chapter explores Java's object strengths.

The UML is object-oriented, and its diagrams lend themselves to being implemented in software that is object-oriented. This chapter examines how UML, coupled with a sound software process model, such as the Unified Process, can produce applications that not only meet the project sponsor's goals, but also are adaptive to the ever-changing needs of the business.

GOALS

- To review Java's object capabilities.
- To explore Java and its relationship to UML.
- To review how UML diagrams are mapped to Java.

Java as an Industrial-Strength Development Language

Numerous tomes chronicle the emergence of Java onto the technology landscape. Suffice it to say, things have not been quite the same since James Gosling (the visionary behind Java's birth at Sun Microsystems) created Sun's first Java applet running in a Mosaic-clone Web browser.

Java has grown immensely since that time and gone through many upgrades and enhancements, including sizeable replacements of major components within Java (the Swing graphics library), along with the advent of enterprise-level Java commitment in the form of Enterprise JavaBeans (EJB). This book focuses on the most recent release of the Java Development Kit, JDK 1.3—more affectionately called Java 2.0. In addition, both JavaBeans and Enterprise JavaBeans will be used extensively to implement most of the Java components, and bean-managed and container-managed persistence using the EJB 2.0 specification will be used with commercial application servers.

Java as a career path has also turned out to be a smart decision. Studies have revealed that a majority of job postings in the U.S. market include Java experience as a requirement over other programming languages. In fact, a recent study by the Forrester research firm reported that 79 percent of all Fortune 1000 companies were deploying enterprise Java applications. Forrester also predicted that that figure will be 100 percent by the end of the year 2003.

Java and Object-Oriented Programming

Many seasoned Java developers will scoff at the fact that this section even exists in this book. It is here for two very important reasons. The first is that I continually run across Java applications built with a procedural mind-set. The fact that you know Java doesn't mean that you

have the ability to transform that knowledge into well-designed object-oriented systems. As both an instructor and consultant, I see many data-processing shops send COBOL and/or Visual Basic developers to a three-day class on UML and a five-day class on Java and expect miracles. Case in point: I was recently asked to review a Java application to assess its design architecture and found that it had only two classes—`SystemController` and `ScreenController`—which contained over 70,000 lines of Java code.

The second reason for the emphasis on how the language maps to object-oriented principles is that people like language comparisons and how they stack up to their counterparts. To appease those that live and die by language comparisons, let's put Java under the scrutiny of what constitutes an object-oriented language.

No definitive definition of what makes a language object-oriented is globally accepted. However, a common set of criteria I personally find useful is that the language must support the following:

- Classes
- Complex types (Java reference types)
- Message passing
- Encapsulation
- Inheritance
- Polymorphism

These are discussed in the next subsections.

Java and Classes

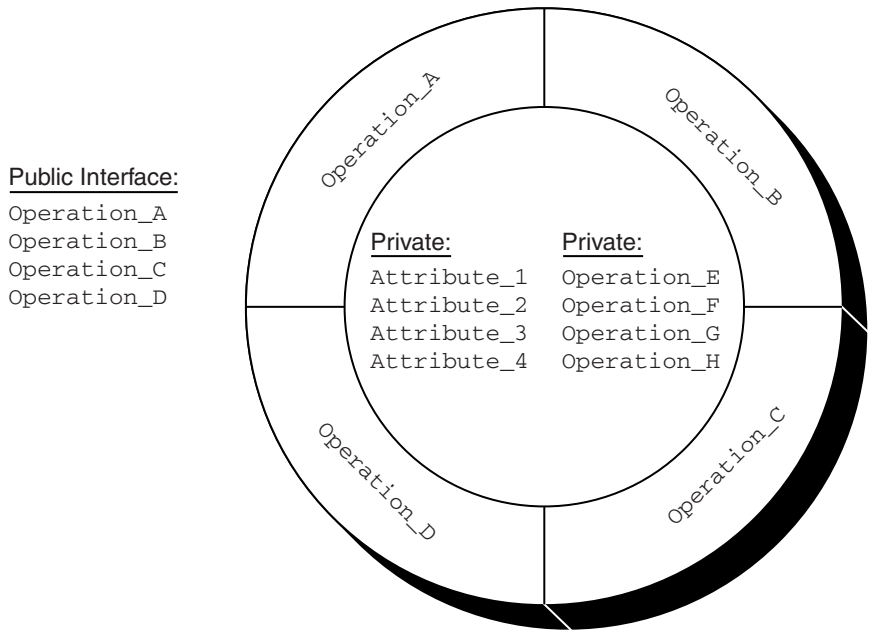
Java allows classes to be defined. There are no stray functions floating around in Java. A **class** is a static template that contains the defined structure (attributes) and behavior (operations) of a real-world entity in the application domain. At runtime, the class is **instantiated**, or brought to life, as an object born in the image of that class. In my seminars, when several folks new to the object world are in attendance, I often use the analogy of a cookie cutter. The cookie cutter is merely the template used to stamp out what will become individually decorated and unique cookies. The cookie cutter is the class; the unique blue, green, and yellow gingerbread man is the object (which I trust supports a bite operation).

Java exposes the class to potential outside users through its public interface. A **public interface** consists of the signatures of the public operations supported by the class. A **signature** is the operation name and its input parameter types (the return type, if any, is not part of the operation's signature).

Good programming practice encourages developers to declare all attributes as private and allow access to them only via operations. As with most other languages, however, this is not enforced in Java. Figure 2-1 outlines the concept of a class and its interface.

The figure uses a common eggshell metaphor to describe the concept of the class's interface, as well as encapsulation. The internal details of the class are hidden from the outside via a well-defined interface. In this case, only four operations are exposed in the classes interface (`Operation_A`, `B`, `C`, and `D`). The other attributes and operations are protected from the outside world. Actually, to the outside world, it's as if they don't even exist.

FIGURE 2-1 *Public interface of a class*



Suppose you want to create an `Order` class in Java that has three attributes—`orderNumber`, `orderDate`, and `orderTotal`—and two operations—`calcTotalValue()` and `getInfo()`. The class definition could look like this:

```
/**
 * Listing 1
 * This is the Order class for the Java/UML book
 */
package com.jacksonreed;
import java.util.*;

public class Order
{
  private Date orderDate;
  private long orderNumber;
  private long orderTotal;

  public Order()
  {
  }

  public boolean getInfo()
  {
    return true;
  }

  public long calcTotalValue()
  {
    return 0;
  }

  public Date getOrderDate()
  {
    return orderDate;
  }

  public void setOrderDate(Date aOrderDate)
  {
    orderDate = aOrderDate;
  }

  public long getOrderNumber()
  {
    return orderNumber;
  }
```

```
  public void setOrderNumber(long aOrderNumber)
  {
    orderNumber = aOrderNumber;
  }

  public long getOrderTotal()
  {
    return orderTotal;
  }

  public void setOrderTotal(long aOrderTotal)
  {
    orderTotal = aOrderTotal;
  }

  public static void main(String[] args)
  {
    Order order = new Order();
    System.out.println("instantiated Order");
    System.out.println(order.getClass().getName());
    System.out.println(order.calcTotalValue());

    try {
      Thread.currentThread().sleep(5*1000);
      } catch(InterruptedException e) {}
  }
}
```

A few things are notable about the first bit of Java code presented in this book. Notice that each of the three attributes has a get and a set operation to allow for the retrieval and setting of the `Order` object's properties. Although doing so is not required, it is common practice to provide these accessor-type operations for all attributes defined in a class. In addition, if the `Order` class ever wanted to be a JavaBean, it would have to have "getters and setters" defined in this way.

Some of the method code in the `main()` operation does a few things of note. Of interest is that a `try` block exists at the end of the operation that puts the current thread to sleep for a bit. This is to allow the console display to freeze so that you can see the results.

If you type in this class and then compile it and execute it in your favorite development tool or from the command prompt with

```
javac order.java //* to compile it
java order //* to run it
```

you should get results that look like this:

```
instantiated Order
com.jacksonreed.Order
0
```

Note: Going forward, I promise you will see no code samples with class, operation, or attribute names of `foo`, `bar`, or `foobar`.

More on Java and Classes

A class can also have what are called *class-level operations* and *attributes.* Java supports these with the `static` keyword. This keyword would go right after the visibility (public, private, protected) component of the operation or attribute. Static operations and attributes are needed to invoke either a service of the class before any real instances of that class are instantiated or a service that doesn't directly apply to any of the instances. The classic example of a static operation is the Java constructor. The constructor is what is called when an object is created with the `New` keyword. Perhaps a more business-focused example is an operation that retrieves a list of `Customer` instances based on particular search criteria.

A class-level attribute can be used to store information that all instances of that class may access. This attribute might be, for example, a count of the number of objects currently instantiated or a property about `Customer` that all instances might need to reference.

Java and Complex Types (Java Reference Types)

A **complex type**, which in Java is called a reference type, allows variables typed as something other than primitive types (e.g., `int` and `boolean`) to be declared. In Java, these are called reference types. In object-oriented systems, variables that are "of" a particular class, such as `Order`, `Customer`, or `Invoice`, must be defined. Taken a step further, `Order` could consist of other class instances, such as `OrderHeader` and `OrderLine`.

In Java, you can define different variables that are references to runtime objects of a particular class type:

```
Public Order myOrder;
Public Customer myCustomer;
Public Invoice myInvoice;
```

Such variables can then be used to store actual object instances and subsequently to serve as recipients of messages sent by other objects. In the previous code fragment, the variable `myOrder` is an instance of `Order`. After the `myOrder` object is created, a message can be sent to it and `myOrder` will respond, provided that the operation is supported by `myOrder`'s interface.

Java and Message Passing

Central to any object-oriented language is the ability to pass messages between objects. In later chapters you will see that work is done in a system only by objects that collaborate (by sending messages) to accomplish a goal (which is specified in a use-case) of the system.

Java doesn't allow stray functions floating around that are not attached to a class. In fact, Java demands this. Unfortunately, as my previous story suggested, just saying that a language requires everything to be packaged in classes doesn't mean that the class design will be robust, let alone correct.

Java supports message passing, which is central to the use of Java's object-oriented features. The format closely resembles the syntax of other languages, such as C++ and Visual Basic. In the following code fragment, assume that a variable called `myCustomer`, of type `Customer`, is defined and that an operation called `calcTotalValue()` is defined for `Customer`. Then the `calcTotalValue()` message being sent to the `myCustomer` object in Java would look like this:

```
myCustomer.calcTotalValue();
```

Many developers feel that, in any other structured language, this is just a fancy way of calling a procedure. Calling a procedure and sending a message are similar in that, once invoked, both a procedure and a message implement a set of well-defined steps. However, a message differs in two ways:

1. There is a designated receiver, the object. Procedures have no designated receiver.
2. The interpretation of the message—that is, the how-to code (called the *method*) used to respond to the message—can vary with different receivers. This point will become more important later in the chapter, when polymorphism is reviewed.

The concepts presented in this book rely heavily on classes and the messaging that takes place between their instances, or objects.

Java and Encapsulation

Recall that a class exposes itself to the outside world via its public interface and that this should be done through exposure to operations only, and not attributes. Java supports encapsulation via its ability to declare both attributes and operations as public, private, or protected. In UML this is called *visibility.*

Using the code from the previous `Order` example, suppose you want to set the value of the `orderDate` attribute. In this case, you should do so with an operation. An operation that gets or sets values is usually called a *getter* or a *setter,* respectively, and collectively such operations are called accessors. The local copy of the order date, `orderDate`, is declared private. (Actually, all attributes of a class should be declared private or protected, so that they are accessible only via operations exposed as public to the outside world.)

Encapsulation provides some powerful capabilities. To the outside world, the design can hide how it derives its attribute values. If the `orderTotal` attribute is stored in the `Order` object, the corresponding get operation defined previously looks like this:

```
public long getOrderTotal()
  {
    return orderTotal;
  }
```

This snippet of code would be invoked if the following code were executed by an interested client:

```
private long localTotal;
private Order localOrder;
localOrder = New Order();
localTotal = localOrder.getOrderTotal()
```

However, suppose the attribute `orderTotal` isn't kept as a local value of the `Order` class, but rather is derived via another mechanism (perhaps messaging to its `OrderLine` objects). If `Order` contains `OrderLine` objects (declared as a `Vector` or `ArrayList` of `OrderLine` objects called `myOrderLines`) and `OrderLine` knows how to obtain its line totals via the message `getOrderLineTotal()`, then the corresponding get operation for `orderTotal` within `Order` will look like this:

```
public long getOrderTotal()
  {
    long totalAmount=0;

    for (int i=0; i < myOrderLines.length; i++)
    {
      totalAmount = totalAmount +
              myOrderLines[i].getOrderLineTotal();
    }
    return totalAmount;
  }
```

This code cycles through the `myOrderLines` collection, which contains all the `Orderline` objects related to the `Order` object, sending the `getOrderLineTotal()` message to each of `Order`'s `OrderLine` objects. The `getOrderTotal()` operation will be invoked if the following code is executed by an interested client:

```
long localTotal;
Order myOrder;
myOrder = new Order();
localTotal = localOrder.getOrderTotal()
```

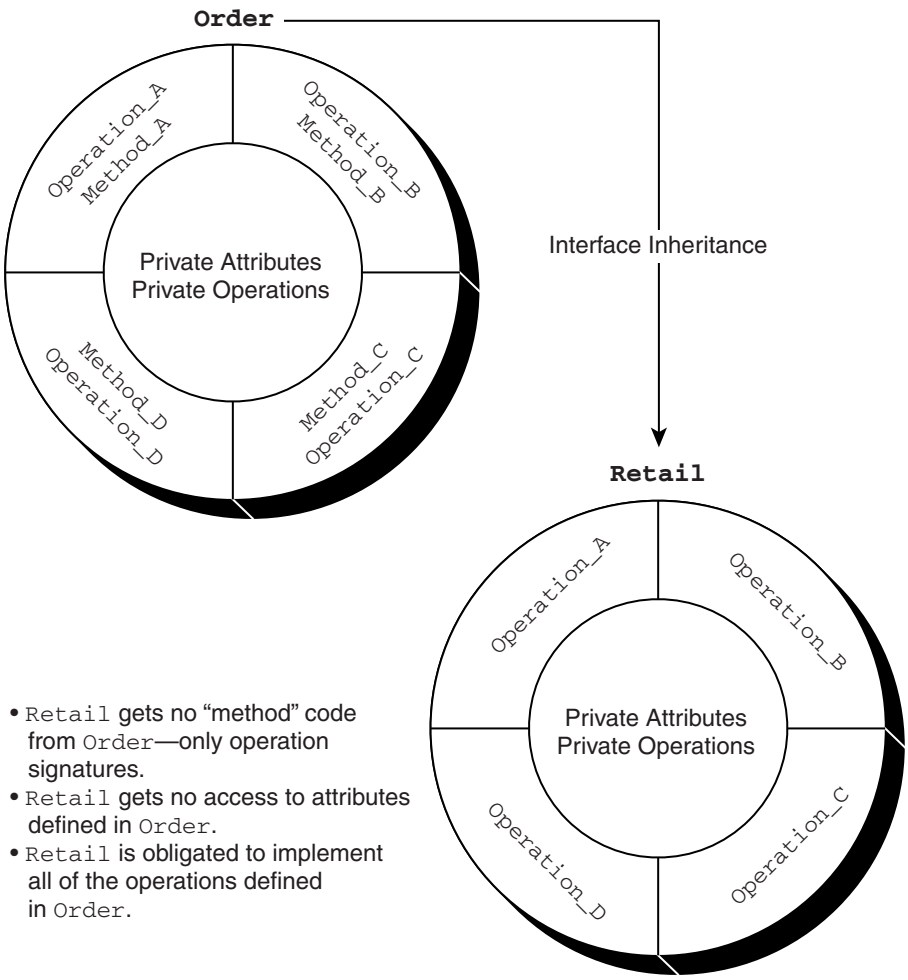
Notice that the "client" code didn't change. To the outside world, the class still has an `orderTotal` attribute. However, you have hidden, or *encapsulated,* just how the value was obtained. This encapsulation allows the class's interface to remain the same (hey, I have an `orderTotal` that you can ask me about), while the class retains the flexibility to change its implementation in the future (sorry, how we do business has changed and now we must derive `orderTotal` like this). This kind of resiliency is one of the compelling business reasons to use an object-oriented programming language in general.

Java and Inheritance

The inclusion of inheritance is often the most cited reason for granting a language object-oriented status. There are two kinds of inheritance: *interface* and *implementation.* As we shall see, Java is one of the few languages that makes a clear distinction between the two.

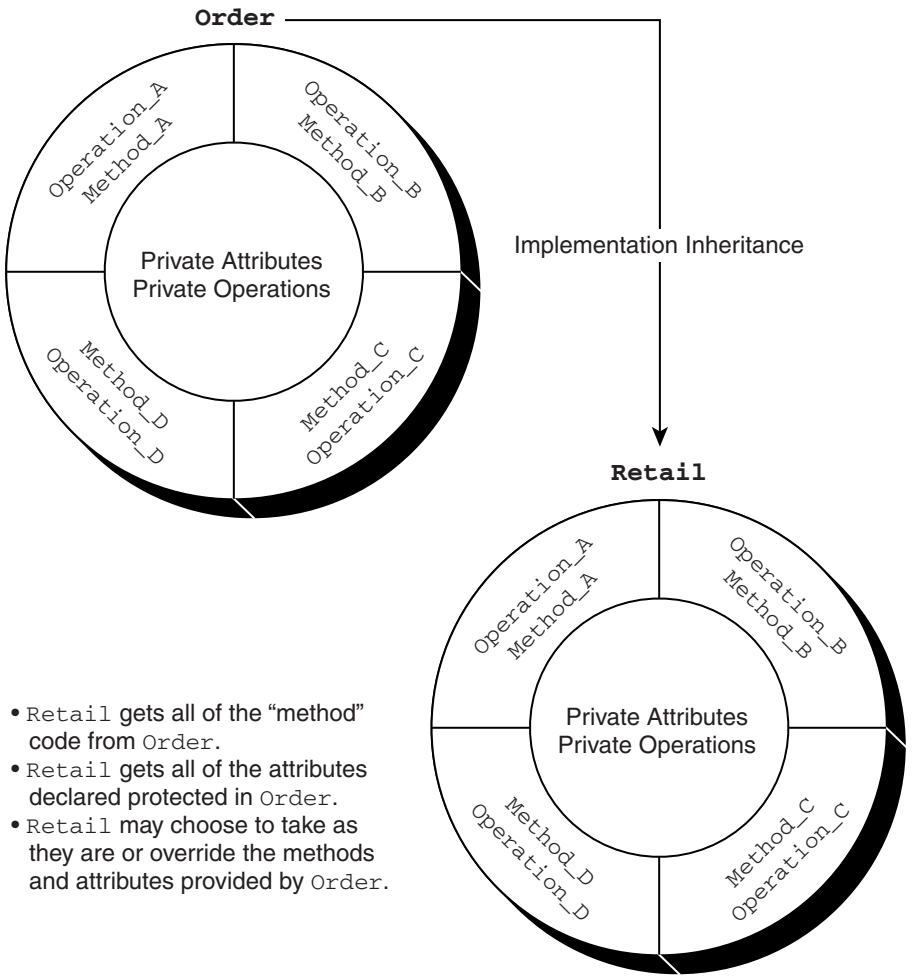
Interface inheritance (Figure 2-2) declares that a class that is inheriting an interface will be responsible for implementing all of the method code of each operation defined in that interface. Only the signatures of the interface are inherited; there is no method or how-to code.

FIGURE 2-2 *Interface inheritance*



Implementation inheritance (Figure 2-3) declares that a class that is inheriting an interface may, at its option, use the method code implementation already established for the interface. Alternatively, it may choose to implement its own version of the interface. In addition, the class inheriting the interface may extend that interface by adding its own operations and attributes.

FIGURE 2-3 *Implementation inheritance*



Each type of inheritance should be scrutinized and used in the appropriate setting. Interface inheritance is best used under the following conditions:

- The base class presents a generic facility, such as a table lookup, or a derivation of system-specific information, such as operating-system semantics or unique algorithms.

- The number of operations is small.
- The base class has few, if any, attributes.
- Classes realizing or implementing the interface are diverse, with little or no common code.

Implementation inheritance is best used under the following conditions:

- The class in question is a domain class that is of primary interest to the application (i.e., not a utility or controller class).
- The implementation is complex, with a large number of operations.
- Many attributes and operations are common across specialized implementations of the base class.

Some practitioners contend that implementation inheritance leads to a symptom called the *fragile base class* problem. Chiefly, this term refers to the fact that over time, what were once common code and attributes in the superclass may not stay common as the business evolves. The result is that many, if not all, of the subclasses, override the behavior of the superclass. Worse yet, the subclasses may find themselves overriding the superclass, doing their own work, and then invoking the same operation again on the superclass. These practitioners espouse the idea of using only interface inheritance. Particularly with the advent of Java and its raising of the interface to a first-class type, the concept and usage of interface-based programming have gained tremendous momentum.

As this book evolves, keeping in mind the pointers mentioned here when deciding between the two types of inheritance will be helpful. Examples of both constructs will be presented in the theme project that extends throughout this book.

Implementation Inheritance

Java supports implementation inheritance with the `extends` keyword. A class wanting to take advantage of implementation inheritance simply adds an `extendsClassName` statement to its class definition. To continue the previous example, suppose you have two different types of orders, both warranting their own subclasses: `Commercial` and `Retail`. You would still have an `Order` class (which isn't instantiated

directly and which is called abstract). The previous fragment showed the code for the `Order` class. Following is the code for the `Commercial` class.

```
package com.jacksonreed;
public class Commercial extends Order
{
  public Commercial()
  {
  }

  /* Unique Commercial code goes here */
}
```

Implementation inheritance allows the `Commercial` class to utilize all attributes and operations defined in `Order`. This will be done automatically by the Java Virtual Machine (JVM) in conjunction with the language environment. In addition, implementation inheritance has the ability to override and/or extend any of `Order`'s behavior. `Commercial` may also add completely new behavior if it so chooses.

Interface Inheritance

Java supports interface inheritance with the `implements` keyword. A class wanting to realize a given interface (actually being responsible for the method code) simply adds an `implements InterfaceName` statement. However, unlike extension of one class by another class, implementation of an interface by a class requires that the interface be specifically defined as an interface beforehand.

Looking again at the previous example with `Order`, let's assume that this system will contain many classes—some built in this release, and some built in future releases—that need the ability to price themselves. Remember from earlier in this chapter that one of the indicators of using interface inheritance is the situation in which there is little or no common code but the functional intent of the classes is the same. This pricing functionality includes three services: the abilities to calculate tax, to calculate an extended price, and to calculate a total price. Let's call the operations for these services `calcExtendedPrice()`, `calcTax()`, and `calcTotalPrice()`, respectively, and assign them to a Java interface called `IPrice`. Sometimes interface names are prefixed with the letter *I* to distinguish them from other classes:

```
package com.jacksonreed;

interface IPrice
{
  long calcExtendedPrice();
  long calcTax();
  long calcTotalPrice();
}
```

Notice that the interface contains only operation signatures; it has no implementation code. It is up to other classes to implement the actual behavior of the operations. For the `Order` class to implement, or realize, the `IPrice` interface, it must include the `implements` keyword followed by the interface name:

```
public class Order implements IPrice
{
}
```

If you try to implement an interface without providing implementations for all of its operations, your class will not compile. Even if you don't want to implement any method code for some of the operations, you still must have the operations defined in your class.

One very powerful aspect of interface inheritance is that a class can implement many interfaces at the same time. For example, `Order` could implement the `IPrice` interface and perhaps a search interface called `ISearch`. However, a Java class may extend from only one other class.

Java and Polymorphism

Polymorphism is one of those $50 words that dazzles the uninformed and sounds really impressive. In fact, polymorphism is one of the most powerful features of any object-oriented language.

Roget's II: The New Thesaurus cross-references the term *polymorphism* to the main entry of *variety.* That will do for starters. Variety is the key to polymorphism. The Latin root for *polymorphism* means simply "many forms." Polymorphism applies to operations in the object-oriented context. So by combining these two thoughts, you could say that operations are *polymorphic* if they are identical (not just in name but also in signatures) but offer variety in their implementations.

Polymorphism is the ability of two different classes each to have an operation that has the same signature, while having two very different

forms of method code for the operation. Note that to take advantage of polymorphism, either an interface inheritance or an implementation inheritance relationship must be involved.

In languages such as COBOL and FORTRAN, defining a routine to have the same name as another routine will cause a compile error. In object-oriented languages such as Java and C++, several classes might have an operation with the same signature. Such duplication is in fact encouraged because of the power and flexibility it brings to the design.

As mentioned previously, the `implements` and `extends` keywords let the application take advantage of polymorphism. As we shall see, the sample project presented later in this book is an order system for a company called Remulak Productions. Remulak sells musical equipment, as well as other types of products. There will be a `Product` class, as well as `Guitar`, `SheetMusic`, and `Supplies` classes.

Suppose, then, that differences exist in the fundamental algorithms used to determine the best time to reorder each type of product (called the economic order quantity, or EOQ). I don't want to let too much out of the bag at this point, but there will be an implementation inheritance relationship created with `Product` as the ancestor class (or superclass) and the other three classes as its descendants (or subclasses). The scenario that follows uses implementation inheritance with a polymorphic example. Note that interface inheritance would yield the same benefits and be implemented in the same fashion.

To facilitate extensibility and be able to add new products in the future in a sort of plug-and-play fashion, we can make `calcEOQ()` polymorphic. To do this in Java, `Product` would define `calcEOQ()` as abstract, thereby informing any inheriting subclass that it must provide the implementation. A key concept behind polymorphism is this: *A class implementing an interface or inheriting from an ancestor class can be treated as an instance of that ancestor class. In the case of a Java interface, the interface itself is a valid type.*

For example, assume that a collection of `Product` objects is defined as a property of the `Inventory` class. `Inventory` will support an operation, `getAverageEOQ()`, that needs to calculate the average economic order quantity for all products the company sells. To do this requires that we iterate over the collection of `Product` objects called `myProducts` to get each object's unique economic order quantity individually, with the goal of getting an average:

```
public long getAverageEOQ()
  {
    long totalAmount=0;

    for (int i=0; i < myProducts.length; i++)
    {
      totalAmount = totalAmount + myProducts[i].calcEOQ();
    }
    return totalAmount / myProducts.length;
  }
```

But wait! First of all, how can `Inventory` have a collection of `Product` objects when the `Product` class is abstract (no instances were ever created on their own)? Remember the maxim from earlier: Any class implementing an interface or extending from an ancestor class can be treated as an instance of that interface or extended class. A `Guitar` "is a" `Product`, `SheetMusic` "is a" `Product`, and `Supplies` "is a" `Product`. So anywhere you reference `Guitar`, `SheetMusic`, or `Supplies`, you can substitute `Product`.

Resident in the array `myProducts` within the `Inventory` class are individual concrete `Guitar`, `SheetMusic`, and `Supplies` objects. Java figures out dynamically which object should get its own unique `calcEOQ()` message. The beauty of this construct is that later, if you add a new type of `Product`—say, `Organ`—it will be totally transparent to the `Inventory` class. That class will still have a collection of `Product` types, but it will have four different ones instead of three, each of which will have its own unique implementation of the `calcEOQ()` operation.

This is polymorphism at its best. At runtime, the class related to the object in question will be identified and the correct "variety" of the operation will be invoked. Polymorphism provides powerful extensibility features to the application by letting future unknown classes implement a predictable and well-conceived interface without affecting how other classes deal with that interface.

Why UML and Java

When modeling elements, our goal is to sketch the application's framework with a keen eye toward using sound object-oriented principles. For this reason, UML, as an object-oriented notation, is a nice fit for any

project using Java as its implementation language. Java was built from the ground up with the necessary "object plumbing" to benefit from the design elements of UML models. More importantly, when UML is combined with a sound software process such as the Unified Process, the chances for the project's success increase dramatically.

James Rumbaugh once said, "You can't expect a method to tell you everything to do. Writing software is a creative process, like painting, writing, or architectural design. There are principles of painting, for example, that give guidelines on composition, color selection, and perspective, but they won't make you a Picasso." You will see what he means when later in the book the UML elements are presented in a workable context during the development of an application using Java. At that time, artifacts will be chosen that add the most value to the problem. We will still need a sound process to be successful, however—and a little luck wouldn't hurt, either.

All of the UML artifacts used in this book will cumulatively lead to better-built Java applications. However, some of the UML deliverables will have a much closer counterpart to the actual Java code produced. For example, use-cases are technology neutral. Actually, use-cases would benefit any project, regardless of the software implementation technology employed, because they capture the application's essential requirements. All subsequent UML deliverables will derive from the foundations built in the use-cases.

For core business and commercial applications, three UML diagrams most heavily affect the Java deliverable: use-case, class, and sequence (or collaboration). Now, I run the risk already of having you think the other diagrams are never used; they are, depending on a project's characteristics and requirements. Yes, the project may also benefit, on the basis of its unique characteristics, from other diagrams, such as state and activity diagrams. In my experience, however, the previously mentioned three diagrams, along with their supporting documentation, are the pivotal models that will be most heavily used. Table 2-1 maps the UML diagrams to Java.

Class Diagram

The king of UML diagrams is the class diagram. This diagram is used to generate Java code with a visual modeling tool (in this book, Rational Software's Rose). In addition, everything learned from all of the

TABLE 2-1 *Mapping UML Diagrams to Java*

UML Diagram	Specific Element	Java Counterpart
Package	Instance of	Java packages
Use-case	Instance of	User interface artifacts (downplayed early on) in the form of pathways that will eventually become sequence diagrams
Class	Operations	Operations/methods
	Attributes	Member variables and related accessor operations
	Associations	Member variables and related accessor operations
Sequence	Instance of	Operation in a controller class to coordinate flow
	Message target	Operation in the target class
Collaboration	Instance of	Operation in a controller class to coordinate flow
	Message target	Operation in the target class
State	Actions/activities	Operations in the class being lifecycled
	Events	Operations in the class being lifecycled or in another collaborating class
	State variables	Attributes in the class being lifecycled
Activity	Action states	Method code to implement a complex operation or to coordinate the messaging of a use-case pathway
Component	Components	Typically one *.java* and/or one *.class* file
Deployment	Nodes	Physical, deployable install sets destined for client and/or server hosting

other diagrams will in one way or another influence this diagram. For example, the key class diagram components are represented in Java as follows:

- **Classes:** The classes identified will end up as automatically generated *.java* class files.
- **Attributes:** The attributes identified in the class will be generated as private (optionally public or protected) member variables in the class module. At the option of the designer, the generation process will also automatically generate the necessary accessor operations (i.e., get and set).

- **Interface:** Through the messaging patterns uncovered in the sequence diagrams, the interface of the class—that is, its public operations—will begin to take shape as operations are added to the class.
- **Operations:** Every operation defined for a class will end up as a public, private, or protected operation within the class. The operations initially will lack the complete signature specification (operation name only), but eventually they will contain fully specified signatures.
- **Associations:** The associations identified between classes will end up as attributes of the classes to enable messaging patterns as detailed by sequence diagrams.
- **Finalized classes:** Finalized classes can often be used to generate first-cut database schemas (assuming a relational database as the persistence store) in the form of Data Definition Language (DDL).

The UML class diagram and its Java counterpart, the class *.java* file, are the core of what drives the application's implementation.

Sequence Diagram

The tasks required to satisfy an application's goals are specified as *pathways* through a use-case. For a banking environment, one use-case might be Handle Deposits. A pathway through this use-case, one of many, might be deposits processed at the teller window. In most cases, each major pathway will have a sequence diagram created for it. Each, although logically stated in the use-case, will eventually end up as a dynamic collaboration between runtime objects, all sending messages to one another.

For example, when the `Customer` object wants each of its `Order` objects to perform the operation `calcTotalValue()`, it sends a message. Each message requires the receiver object (the `Order`) to have an operation defined to honor the request. Operations all end up in a class somewhere. These classes eventually are used to generate code in the Java environment.

The project team uses the sequence diagram to "walk through" the application. Once the project team has become comfortable with UML, and the accompanying Unified Process, it will no longer need to walk through code. Once the sequence diagram has passed inspection, the method-level coding can be implemented.

Eventually the sequence diagram walk-throughs will be the primary confirmation of whether a use-case pathway is correct. Most visual modeling tools, at present, do not generate Java code from the message patterns outlined in the sequence diagram (Together Control Center from TogetherSoft will reverse engineer sequence diagrams from Java code). However, I contend that this wouldn't be difficult for all visual modeling tools, and the next version of these products likely will support this ability. Having it would certainly differentiate competitors.

Component Diagram

The fully developed classes are assigned to components in the visual modeling tool's component diagrams. Many will fit a variety of possible physical manifestations:

- Graphical forms (applets and/or applications)
- Business-level rule components
- Transaction or persistence components

These component choices will be reflected in *.java* files.

Deployment Diagram

Components defined in the visual modeling tool are deployed on nodes specified in the deployment diagrams. These diagrams depict the physical machines that will house the components specified as components previously. These deployment strategies may be Web-based solutions, multitier solutions, or standalone Java applications.

Visual Modeling Tool Support

The UML and Java fit together well. The value that UML adds is enhanced by the use of a visual modeling tool that supports both forward and reverse engineering (creating code from models and creating models from code). The use of a visual modeling tool also aids traceability and cross-checking of the model components.

A project that combines an effective software process model and a robust modeling language such as UML nevertheless will be hindered if it lacks a visual modeling tool. Without such a tool, the project will produce paper models that won't be updated or, worse, that will be lost

in the shuffle of day-to-day project activity. Many excellent products are available, and they continue to evolve. Pick a tool and make it part of your process. Personally, I wouldn't be caught dead without one.

Checkpoint

Where We've Been

- Once a set-top language destined to control the toasters of the world, Java has become the language darling in the software industry and is quickly eclipsing many other long-standing languages that have been around for years.
- Java has grown in acceptance for many reasons, including its support of a write-once, run-anywhere strategy. In addition, the vast middleware marketplace that affords multitier solutions has embraced Java as its prime source of enablement.
- Java cleanly implemented the notion of interface and implementation inheritance, allowing for a more natural and easy-to-understand use of the constructs.
- Java is greatly influenced by the work done in three UML diagrams: use-case, class, and sequence (or collaboration).

Where We're Going Next

In the next chapter we:

- Explore the project plan for the Unified Process model.
- Review the importance of creating a vision for a project, and look at deliverables from that effort.
- Get acquainted with the book's continuing project, Remulak Productions.
- Produce an event list as a precursor to use-case analysis.