

CHAPTER 11



Washington Canoe Club

EXTENDING CLASS BEHAVIOR THROUGH INHERITANCE

LEARNING OBJECTIVES

- *STATE THE PURPOSE OF INHERITANCE*
- *STATE THE PURPOSE OF A BASE CLASS*
- *STATE THE PURPOSE OF A DERIVED CLASS*
- *STATE THE PURPOSE OF AN ABSTRACT METHOD*
- *STATE THE PURPOSE OF AN ABSTRACT BASE CLASS*
- *STATE THE PURPOSE OF AN INTERFACE*
- *DEMONSTRATE YOUR ABILITY TO WRITE POLYMORPHIC JAVA CODE*
- *STATE THE PURPOSE OF A DEFAULT CONSTRUCTOR*
- *DESCRIBE THE BEHAVIOR OF CONSTRUCTOR CHAINING*
- *DEMONSTRATE YOUR ABILITY TO CREATE A UML DIAGRAM THAT ILLUSTRATES INHERITANCE RELATIONSHIPS BETWEEN JAVA CLASSES*
- *STATE THE PURPOSE OF THE FINAL KEYWORD*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE ABSTRACT METHODS, ABSTRACT CLASSES, AND INTERFACES IN YOUR JAVA PROGRAMS*
- *DEMONSTRATE YOUR ABILITY TO UTILIZE FINAL METHODS AND CLASSES IN YOUR JAVA PROGRAMS.*

INTRODUCTION

Inheritance is a powerful object-oriented programming feature provided by the Java programming language. The behavior provided by or specified by one class (*base class*) can be adopted or extended by another class (*derived or subclass*). Up to this point you have been using inheritance in every Java program you have written although mostly this has been done for you by the Java compiler and platform behind the scenes. For example, every user-defined class you create automatically inherits from the `java.lang.Object` class. (See chapter 8, figure 8-3 or chapter 9, figure 9-2)

In this chapter you will learn how to create new derived classes from existing classes and interfaces. There are three ways of doing this: 1) by *extending* the functionality of an existing class, 2) by *implementing* one or more interfaces, or 3) you can combine these two methods to create derived classes that both extend the functionality of a base class and implement the operations declared in one or more interfaces.

Along the way I will show you how to create and use abstract methods to create *abstract classes*. You will learn how to create and utilize *interfaces* in your program designs as well as how to employ the *final* keyword to inhibit the inheritance mechanism. You will also learn how to use a UML class diagram to show inheritance hierarchies.

By the time you complete this chapter you will fully understand how to create an object-oriented Java program that exhibits *dynamic polymorphic behavior*. Most importantly, however, you will understand why dynamic polymorphic behavior is the desired end-state of an object-oriented program.

This chapter also builds on the material presented in chapter 10 - Compositional Design. The primary chapter example demonstrates the design possibilities you can achieve when you combine inheritance with compositional design.

THREE PURPOSES OF INHERITANCE

Inheritance serves three essential purposes. The first purpose of inheritance is to serve as an object-oriented design mechanism that enables you to think and reason about the structure of your programs in terms of *generalized* and *specialized* class behavior. A base class implements, or specifies, generalized behavior common to all of its subclasses. Subclasses derived from this base class capitalize on the behavior it provides. Additionally, subclasses may specify, or implement, specialized behavior if required in the context of the design.

When you think in terms of inheritance you think in terms of class hierarchies where the base classes that implement generalized behavior appear at or near the top of the hierarchy and the derived classes that implement specialized behavior appear toward the bottom. Figure 11-1 gives a classic example of an inheritance hierarchy showing generalized/specialized behavior.

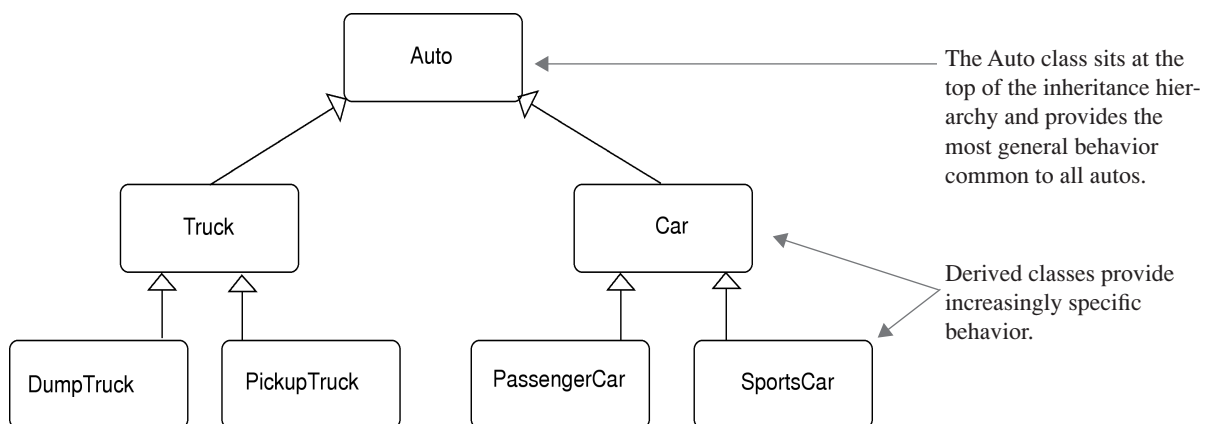


Figure 11-1: Inheritance Hierarchy Illustrating Generalized & Specialized Behavior

Referring to figure 11-1 — the Auto class sits at the top of the inheritance hierarchy and provides generalized behavior for all of its derived classes. The Truck and Car classes derive from Auto. They may provide specialized behavior found only in trucks and cars. The DumpTruck and PickupTruck classes derive from Truck, which means that Truck is also serving as a base class. DumpTruck and PickupTruck inherit Truck’s generalized behavior and implement the specialized behavior required of these classes. The same holds true for the PassengerCar and SportsCar classes. Their *direct base class* is Car, whose direct base class is Auto. For more real-world examples of inheritance hierarchies simply consult the Java API documentation or refer to chapter 5.

The second purpose of inheritance is to provide a way to gain a measure of code reuse within your programs. If you can implement generalized behavior in a base class that’s common to all of its subclasses then you don’t have to re-write the code in each subclass. If, in one of your programming projects, you create an inheritance hierarchy and find you are repeating a lot of the same code in each of the subclasses, then it’s time for you to refactor your design and migrate the common code into a base class higher up in your inheritance hierarchy.

The third purpose of inheritance is to enable you to incrementally develop code. It is rare for a programmer, or more often, a team of programmers, to sit down and in one heroic effort completely code the entire inheritance hierarchy for a particular application. It’s more likely the case that the inheritance hierarchy, and its accompanying classes, grows over time. (*Take the Java API as a prime example.*)

IMPLEMENTING THE “IS A” RELATIONSHIP

A class that belongs to an inheritance hierarchy participates in what is called an “*is a*” relationship. Referring again to figure 11-1, a Truck is an Auto and a Car is an Auto. Likewise, a DumpTruck is a Truck, and since a Truck is an Auto a DumpTruck is an Auto as well. In other words, class hierarchies are transitive in nature when navigating from specialized classes to more generalized classes. They are not transitive in the opposite direction however. For instance, an Auto is not a Truck or a Car, etc.

A thorough understanding of the “*is a*” relationships that exist within an inheritance hierarchy will pay huge dividends when you want to substitute a derived class object in code that specifies one of its base classes. (*This is a critical skill in Java programming because it’s done extensively in the API.*)

THE RELATIONSHIP BETWEEN THE TERMS TYPE, INTERFACE, AND CLASS

Before moving on it will help you to understand the relationship between the terms *type*, *class* and *interface*. Java is a strongly-typed programming language. This means that when you write a Java program and wish to call a method on a particular object, Java must know, in advance, the type of object to which you refer. In this context the term *type* refers to *that set of operations or methods a particular object supports*. Every object you use in your Java programs has an associated type. If, by mistake, you try and call a non-supported method on an object, you will be alerted to your mistake by a compiler error when you try and compile your program.

MEANING OF THE TERM INTERFACE

An interface is a Java construct that introduces a new data type and its set of authorized operations in the form of method declarations. A method declaration specifies the method signature but omits the method body. No body — no behavior. Interfaces are discussed in more detail later in the chapter.

MEANING OF THE TERM CLASS

A class is a Java construct that introduces and defines a new data type. Like the interface, the class specifies a set of legal operations that can be called on an object of its type. However, the class can go one step further and provide definitions (*i.e., behavior*) for some or all of its methods. When a class provides definitions for all of its methods it is a *concrete class*, meaning that objects of that class type can be created with the new operator. (*i.e., they can be instantiated*) If a class definition omits the body, and therefore the behavior, of one or more of its methods, then that class must be declared to be an *abstract class*. Abstract class objects cannot be created with the new operator. I will discuss abstract classes in greater detail later in the chapter.

Quick Review

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about your program structure in terms of generalized and specialized class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it provides a means to incrementally develop your programs over time.

Classes that belong to an inheritance hierarchy participate in an “is a” relationship between themselves and their chain of base classes. This “is a” relationship is transitive in the direction of specialized to generalized classes but not vice versa.

The Java class and interface constructs are each used to create new, user-defined data types. The interface construct is used to specify a set of authorized type methods and omits method behavior; the class construct is used to specify a set of authorized type methods and their behavior. A class construct, like an interface, can omit the bodies of one or more of its methods, however, such methods must be declared to be abstract. A class that declares one or more of its methods to be abstract must itself be declared to be an abstract class. Abstract class objects cannot be created with the new operator.

EXPRESSING GENERALIZATION & SPECIALIZATION IN THE UML

Generalization & specialization relationships can be expressed in a UML class diagram by drawing a solid line with a hollow-tipped arrow from the derived class to the base class as figure 11-2 illustrates.

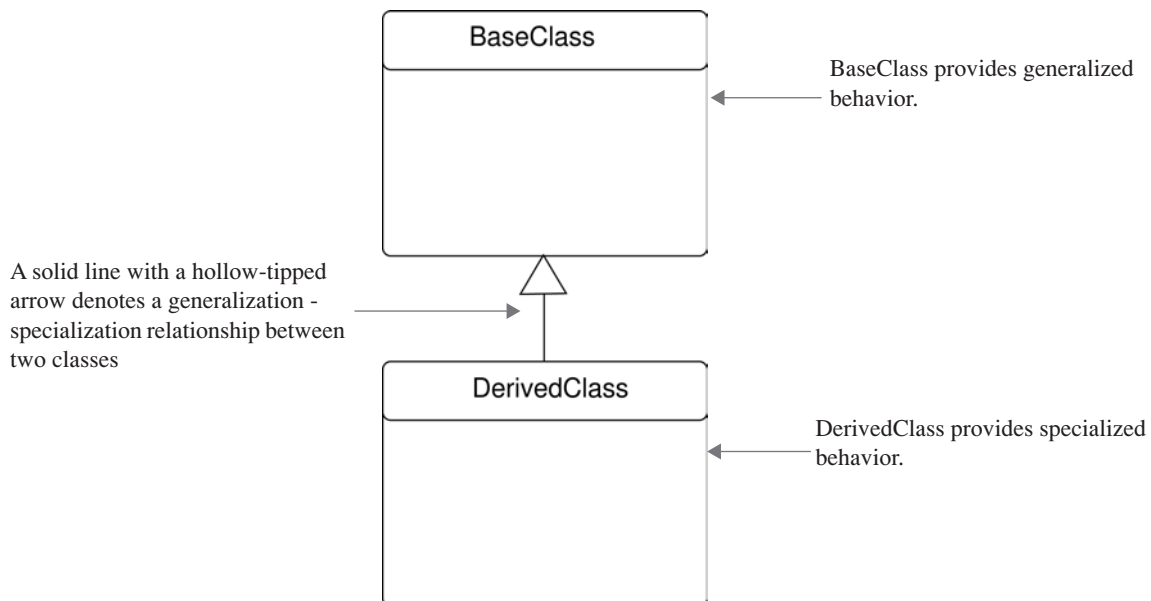


Figure 11-2: UML Class Diagram Showing DerivedClass Inheriting From BaseClass

Referring to figure 11-2 — the BaseClass class acts as the direct base class to DerivedClass. Behavior provided by BaseClass is inherited by DerivedClass. Let’s take a look at an example program that implements these two classes.

A SIMPLE INHERITANCE EXAMPLE

The simple inheritance example program presented in this section expands on the UML diagram shown in figure 11-2. The behavior implemented by BaseClass is kept intentionally simple so you can concentrate on the topic of inheritance. You'll be introduced to more complex programs soon enough.

THE UML DIAGRAM

A more complete UML diagram showing the fields and methods of BaseClass and DerivedClass is presented in figure 11-3

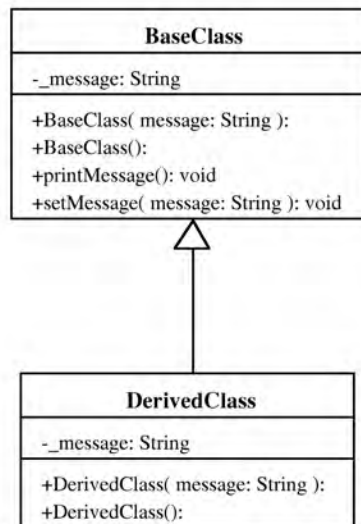


Figure 11-3: UML Diagram of BaseClass & DerivedClass Showing Fields and Methods

Referring to figure 11-3 — BaseClass contains one private field named `_message` which is of type `String`. BaseClass has four public methods: two constructors, `printMessage()`, and `setMessage()`. One of the constructors is a default constructor. A default constructor is simply a constructor that takes no arguments. The second constructor has one parameter of type `String` named `message`. Based on this information, objects of type `BaseClass` can be created in two ways. Once an object of type `BaseClass` is created the `printMessage()` or the `setMessage()` methods can be called on that object.

DerivedClass has its own private `_message` field and two constructors that are similar to the constructors found in BaseClass. Let's now take a look at the source code for each class.

BASECLASS SOURCE CODE

```

1      public class BaseClass {
2          private String _message = null;
3
4          public BaseClass(String message) {
5              _message = "Message from BaseClass: " + message;
6          }
7
8          public BaseClass() {
9              this("BaseClass message!");
10         }
11
12         public void printMessage() {
  
```

11.1 BaseClass.java

```

13         System.out.println(_message);
14     }
15
16     public void setMessage(String message) {
17         _message = "Message from BaseClass: " + message;
18     }
19 }

```

Referring to example 11.1 — BaseClass is fairly simple. Its first constructor begins on line 4 and declares one String parameter named `message`. The `_message` field is set by concatenating the `message` parameter and the String literal “*Message from BaseClass:*”. The default constructor begins on line 8. It calls the first constructor with the String literal “*BaseClass message!*”. The `printMessage()` method begins on line 12. It simply prints the `_message` field to the console. The `setMessage()` method begins on line 16. Its job is to change the value of the `_message` field.

Since all methods have a body, and are therefore defined, the BaseClass is also a concrete class. This means that objects of type BaseClass can be created with the new operator.

DERIVEDCLASS SOURCE CODE

11.2 *DerivedClass.java*

```

1     public class DerivedClass extends BaseClass {
2         private String _message = null;
3
4         public DerivedClass(String message) {
5             super(message);
6             _message = message;
7         }
8
9         public DerivedClass() {
10            this("DerivedClass message!");
11        }
12    }

```

Referring to example 11.2 — DerivedClass inherits the functionality of BaseClass by extending BaseClass using the *extends* keyword on line 1. DerivedClass itself provides two constructors and a private field named `_message`. The first constructor begins on line 4. It declares a String parameter named `message`. The first thing this constructor does, however, on line 5, is call the String parameter version of the BaseClass constructor using the `super()` method with the `message` parameter as an argument. If the `super()` method is called in a derived class constructor it must be the first line of code in the constructor body as is done here. The next thing the DerivedClass constructor does is set the value of its `_message` field to the value of the `message` parameter. (*Remember, these are references to String objects.*)

DerivedClass’s default constructor begins on line 9. It calls its version of the String parameter constructor using the String literal “*DerivedClass message!*” as an argument, which, as you learned above, then calls the BaseClass constructor via the `super()` method.

Let’s now take a look at how these two classes can be used in a program.

DRIVERAPPLICATION PROGRAM

11.3 *DriverApplication.java*

```

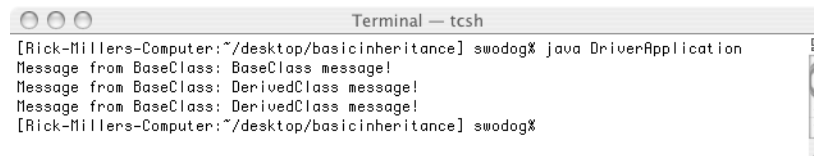
1     public class DriverApplication {
2         public static void main(String[] args) {
3             BaseClass bcr1 = new BaseClass();
4             BaseClass bcr2 = new DerivedClass();
5             DerivedClass dcr1 = new DerivedClass();
6
7             bcr1.printMessage();
8             bcr2.printMessage();
9             dcr1.printMessage();
10        }
11    }

```

The DriverApplication class is used to test the functionality of BaseClass and DerivedClass. The important thing to note in this example is what type of object is being declared and created on lines 3 through 5. Starting on line 3 a BaseClass reference named `bcr1` is declared and initialized to point to a BaseClass object. On line 4 another BaseClass reference named `bcr2` is declared and initialized to point to a DerivedClass object. On line 5 a DerivedClass reference named `dcr1` is declared and initialized to point to a DerivedClass object. Note that a reference to a base class

object can point to a derived class object. Also note that for this example only the default constructors are being used to create each object. This will result in the default message text being assigned to each object's `_message` field.

Continuing with example 11.3 — on lines 7 through 9 the `printMessage()` method is called on each reference. It's time now to compile and run the code. Figure 11-4 gives the results of running example 11.3.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/basicinheritance] swodog% java DriverApplication
Message from BaseClass: BaseClass message!
Message from BaseClass: DerivedClass message!
Message from BaseClass: DerivedClass message!
[Rick-Millers-Computer:~/desktop/basicinheritance] swodog%
```

Figure 11-4: Results of Running Example 11.3

As you will notice from studying figure 11-4 there are three lines of program output that correspond to the three `printMessage()` method calls on each reference `bcr1`, `bcr2`, and `dcr1`. Creating a `BaseClass` reference and initializing it to a `BaseClass` object results in the `BaseClass` version (*the only version at this point*) of the `printMessage()` method being called which prints the default `BaseClass` text message.

Creating a `BaseClass` reference and initializing it to point to a `DerivedClass` object has slightly different behavior, namely, the value of the resulting text printed to the console reflects the fact that a `DerivedClass` object was created, which resulted in the `BaseClass` `_message` field being set to the `DerivedClass` default value. Note that `DerivedClass` does not have a `printMessage()` method therefore it is the `BaseClass` version of `printMessage()` that is called. This behavior is inherited by `DerivedClass`.

Finally, creating a `DerivedClass` reference and initializing it to point to a `DerivedClass` object appears to have the same effect as the previous `BaseClass` reference -> `DerivedClass` object combination. This is the case in this simple example because `DerivedClass` simply inherits `BaseClass`'s default behavior and, except for its own constructors, leaves it unchanged.

Quick Review

A base class implements default behavior in the form of methods that can be inherited by derived classes. There are three reference -> object combinations: 1) if the base class is a concrete class, meaning it is not abstract, then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

ANOTHER INHERITANCE EXAMPLE: PERSON - STUDENT

Let's now take a look at a more real-world example of inheritance. This example will utilize the `Person` class presented in chapter 9 as a base class. The derived class will be called `Student`. Let's take a look at the UML diagram for this inheritance hierarchy.

THE PERSON - STUDENT UML CLASS DIAGRAM

Figure 11-5 gives the UML class diagram for the `Student` class inheritance hierarchy. Notice the behavior provided by the `Person` class in the form of its public interface methods. The `Student` class extends the functionality of `Person` and provides a small bit of specialized functionality of its own in the form of the `getStudentInfo()` method.

Since the `Student` class participates in an is-a relationship with class `Person`, a `Student` object can be used where a `Person` object is called for in your source code. However, now you must be keenly aware of the specialized behavior provided by the `Student` class as you will soon see when you examine and run the driver application program for this example.

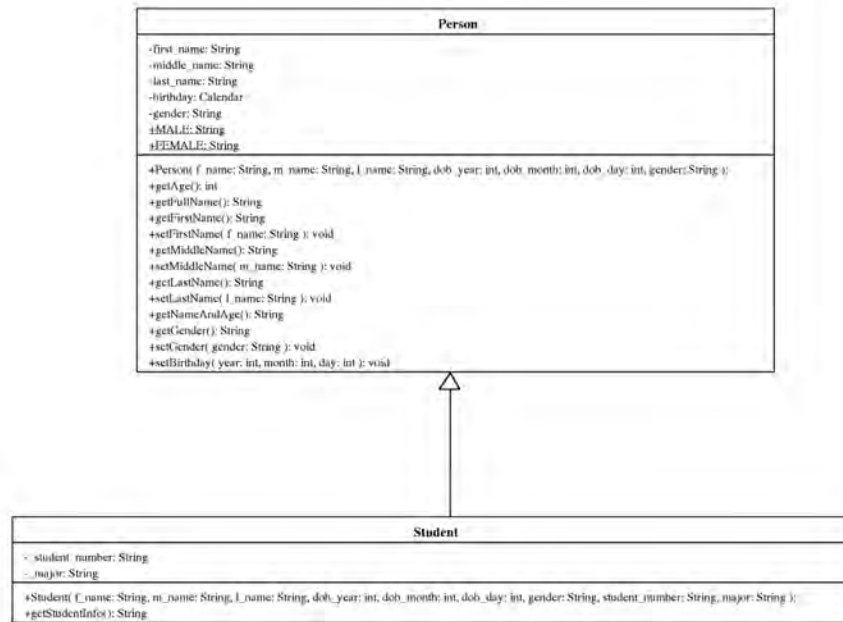


Figure 11-5: UML Diagram Showing Student Class Inheritance Hierarchy

PERSON - STUDENT SOURCE CODE

```

1      import java.util.*;
2
3      public class Person {
4          private String first_name = null;
5          private String middle_name = null;
6          private String last_name = null;
7          private Calendar birthday = null;
8          private String gender = null;
9
10         public static final String MALE = "Male";
11         public static final String FEMALE = "Female";
12
13         public Person(String f_name, String m_name, String l_name, int dob_year,
14             int dob_month, int dob_day, String gender){
15             first_name = f_name;
16             middle_name = m_name;
17             last_name = l_name;
18             this.gender = gender;
19
20             birthday = Calendar.getInstance();
21             birthday.set(dob_year, dob_month, dob_day);
22         }
23
24         public int getAge(){
25             Calendar today = Calendar.getInstance();
26             int now = today.get(Calendar.YEAR);
27             int then = birthday.get(Calendar.YEAR);
28             return (now - then);
29         }
30
31         public String getFullName(){ return (first_name + " " + middle_name + " " + last_name); }
32
33         public String getFirstName(){ return first_name; }
34         public void setFirstName(String f_name) { first_name = f_name; }
35
36         public String getMiddleName(){ return middle_name; }
37         public void setMiddleName(String m_name){ middle_name = m_name; }
38
39         public String getLastName(){ return last_name; }
  
```

11.4 Person.java


```

40     public void setLastName(String l_name){ last_name = l_name; }
41
42     public String getNameAndAge(){ return (getFullName() + " " + getAge()); }
43
44     public String getGender(){ return gender; }
45     public void setGender(String gender){ this.gender = gender; }
46
47     public void setBirthday(int year, int month, int day){ birthday.set(year, month, day); }
48
49 } //end Person class

```

The Person class code is unchanged from chapter 9.

11.5 Student.java

```

1     public class Student extends Person {
2         private String _student_number = null;
3         private String _major = null;
4
5         public Student(String f_name, String m_name, String l_name, int dob_year,
6             int dob_month, int dob_day, String gender, String student_number, String major){
7             super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
8             _student_number = student_number;
9             _major = major;
10        }
11
12        public String getStudentInfo(){
13            return getNameAndAge() + " " + _student_number + " " + _major;
14        }
15    } // end Student class

```

The Student class extends Person and implements specialized behavior in the form of the `getStudentInfo()` method. The Student class has one constructor that takes several String and integer parameters. With the exception of the last two parameters, `student_number` and `major`, the parameters are those required by the Person class and in fact they are used on line 7 as arguments to the `super()` method call. The parameters `student_number` and `major` are used to set a Student object's `_student_number` and `_major` fields respectively on lines 8 and 9.

The `getStudentInfo()` method begins on line 12. It returns a String that is a concatenation of the String returned by the Person object's `getNameAndAge()` method and the `_student_number` and `_major` fields. Take note specifically of the `getNameAndAge()` method call on line 13. You will frequently call public or protected base class methods from derived classes in this manner.

This is all the specialized functionality required of the Student class for this example. The majority of its functionality is provided by the Person class. Let's now take a look at these two classes in action. Example 11.6 gives the test driver program.

11.6 PersonStudentTestApp.java

```

1     public class PersonStudentTestApp {
2         public static void main(String[] args){
3
4             Person p1 = new Person("Steven", "Jay","Jones", 1946, 8, 30, Person.MALE);
5             Person p2 = new Student("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE,
6                 "000002", "Business");
7             Student s1 = new Student("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE,
8                 "000003", "Math");
9
10            System.out.println(p1.getNameAndAge());
11            System.out.println(p2.getNameAndAge());
12            // System.out.println(p2.getStudentInfo()); // error - p2 is a Person reference
13            System.out.println(s1.getNameAndAge());
14            System.out.println(s1.getStudentInfo());
15        }
16    }

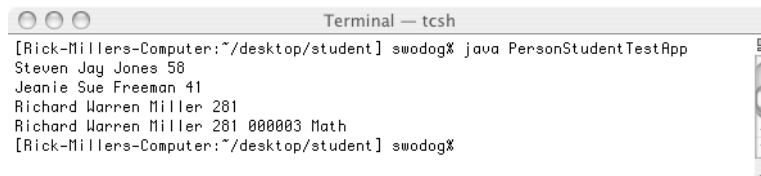
```

This program is similar in structure to example 11-3 in that it declares three references and shows you the effects of calling methods on those references. A Person reference named `p1` is declared on line 4 and initialized to point to a Person object. On line 5, another Person reference named `p2` is declared and initialized to point to a Student object. On line 7 a Student reference is declared and initialized to point to a Student object.

The method calls begin on line 10 with a call to the `getNameAndAge()` method on a Person object via the `p1` reference. This will work fine. On line 11 the `getNameAndAge()` method is called on a Student object via the `p2` reference, which, as you know, is a Person-type reference.

Line 12 is commented out. This line, if you try to compile it in its current form, will cause a compiler error because an attempt is being made to call the `getStudentInfo()` method, which is specialized behavior of the `Student` class, on a `Student` object, via a `Person`-type reference. Now, repeat the previous sentence to yourself several times until you fully understand its meaning. Good! Now, you may ask, and rightly so at this point, “But wait...why can’t you call the `getStudentInfo()` method on a `Student` object?” The answer is — you can, but `p2` is a `Person`-type reference, which means that the compiler is checking to ensure you only call methods defined by the `Person` class via that reference. Remember the “*Java is a strongly-typed language...*” spiel I delivered earlier in this chapter? I will show you how to use casting to resolve this issue after I show you how this program runs.

Continuing with example 11.6 — on line 13 the `getNameAndAge()` method is called on a `Student` object via a `Student` reference. This is perfectly fine since a `Student` is a `Person`. On line 14 the `getStudentInfo()` method is called on a `Student` object via a `Student` reference. This also performs as expected. Figure 11-6 gives the results of running example 11.6.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/student] swodog% java PersonStudentTestApp
Steven Jay Jones 58
Jeanie Sue Freeman 41
Richard Warren Miller 281
Richard Warren Miller 281 000003 Math
[Rick-Millers-Computer:~/desktop/student] swodog%
```

Figure 11-6: Results of Running Example 11.6

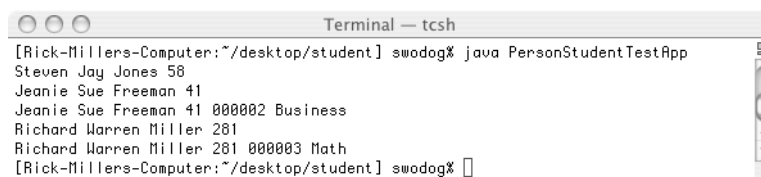
CASTING

OK, now let’s take a look at a modified version of example 11.6 that takes care of the problem encountered on line 12. Example 11.7 gives the modified version of `PersonStudentTestApp.java`.

11.7 *PersonStudentTestApp.java (Mod 1)*

```
1     public class PersonStudentTestApp {
2         public static void main(String[] args){
3
4             Person p1 = new Person("Steven", "Jay","Jones", 1946, 8, 30, Person.MALE);
5             Person p2 = new Student("Jeanie", "Sue", "Freeman", 1963, 10, 10, Person.FEMALE,
6                 "000002", "Business");
7             Student s1 = new Student("Richard", "Warren", "Miller", 1723, 2, 29, Person.MALE,
8                 "000003", "Math");
9
10            System.out.println(p1.getNameAndAge());
11            System.out.println(p2.getNameAndAge());
12            System.out.println(((Student)p2).getStudentInfo()); // casting resolves the issue
13            System.out.println(s1.getNameAndAge());
14            System.out.println(s1.getStudentInfo());
15        }
16    }
```

Notice on line 12 that the compiler has been instructed to treat the `p2` reference as though it were a `Student` reference. This form of explicit type coercion is called *casting*. Casting only works if the object really is of the type you are casting it to. In other words, you would be in big trouble if you tried to cast a `Person` to a `Car` since a `Person` is not a `Car`. Figure 11-7 shows the results of running this program.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/student] swodog% java PersonStudentTestApp
Steven Jay Jones 58
Jeanie Sue Freeman 41
Jeanie Sue Freeman 41 000002 Business
Richard Warren Miller 281
Richard Warren Miller 281 000003 Math
[Rick-Millers-Computer:~/desktop/student] swodog%
```

Figure 11-7: Results of Running Example 11.7

USE CASTING SPARINGLY

Casting is a helpful feature but too much casting usually means your design is not optimal from an object-oriented point of view. You will see more situations in this book where casting is required, but mostly, I will try and show you how to design programs that minimize the need to cast.

Quick Review

The Person class provided the default behavior for the Student class. The Student class inherited Person's default behavior and implemented some specialized behavior of its own. Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting so long as the type of the reference supports the method you are trying to call. You can force, or coerce, the compiler to treat a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances but, as a rule, casting should be used sparingly. Also, casting only works if the object really is of the type you are casting it to.

OVERRIDING BASE CLASS METHODS

So far you have only seen examples of inheritance where the derived class fully accepted the behavior provided by its base class. This section will show you how to override base class behavior in the derived class by overriding base class methods.

To override a base class method in a derived class you will need to re-define the method with the exact signature in the derived class. The overriding derived class method must also return the same type as the overridden base class method. (*The Java compiler will complain if you try otherwise!*) Let's take a look at a simple example. Figure 11-8 gives a UML class diagram for the BaseClass and DerivedClass classes.

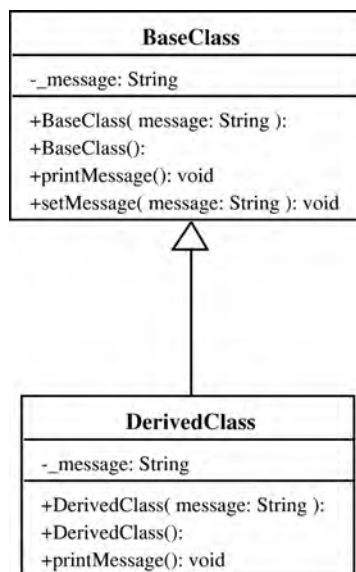


Figure 11-8: UML Class Diagram For BaseClass & DerivedClass

Referring to figure 11-8 — notice that DerivedClass now has a public method named printMessage(). BaseClass remains unchanged. Example 11.8 gives the source code for the modified version of DerivedClass.

11.8 DerivedClass.java (mod 1)

```

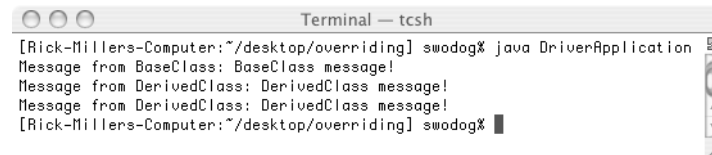
1     public class DerivedClass extends BaseClass {
2         private String _message = null;
3
4         public DerivedClass(String message) {
5             super(message);
6             _message = message;
7         }
  
```

```

8
9     public DerivedClass(){
10         this("DerivedClass message!");
11     }
12
13     public void printMessage(){
14         System.out.println("Message from DerivedClass: " + _message);
15     }
16 }

```

The only change to `DerivedClass` is the addition of the `printMessage()` method starting on line 13. `DerivedClass`'s version of `printMessage()` overrides the `BaseClass` version. How does this affect the behavior of these two classes? A good way to explore this issue is to recompile and run the `DriverApplication` given in example 11.3. Figure 11-9 shows the results of running the program using a modified version of `DerivedClass`.



```

Terminal - tcsh
[Rick-Millers-Computer:~/desktop/overriding] swodog% java DriverApplication
Message from BaseClass: BaseClass message!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: DerivedClass message!
[Rick-Millers-Computer:~/desktop/overriding] swodog% █

```

Figure 11-9: Results of Running Example 11.3 with Modified Version of `DerivedClass`

Referring to figure 11-9 — compare these results with those of figure 11-4. The first message is the same, which is as it should be. The `bcr1` reference points to a `BaseClass` object. The second message is different though. Why is this so? The `bcr2` reference is pointing to a `DerivedClass` object. When the `printMessage()` method is called on the `DerivedClass` object via the `BaseClass` reference the overriding `printMessage()` method provided in `DerivedClass` is called. This is an example of polymorphic behavior. A base class reference, `bcr2`, points to a derived class object. Via the base class reference you call a method provided by the base class interface but is overridden in the derived class and, voila, you have polymorphic behavior. Pretty cool, huh?

Quick Review

Derived classes can override base class behavior by providing overriding methods. An overriding method is a method in a derived class that has the same method signature as the base class method it is intending to override. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

ABSTRACT METHODS & ABSTRACT BASE CLASSES

An abstract method is one that appears in the body of a class declaration but omits the method body. A class that declares one or more abstract methods must be declared to be an abstract class. If you create an abstract method and forget to declare the class as being abstract the Java compiler will inform you of your mistake.

Now, you could simply declare a class to be abstract even though it provides implementations for all of its methods. This would prevent you from creating objects of the abstract class directly with the `new` operator. This may or may not be the intention of your application design goals. However, abstract classes of this nature are not the norm.

THE PRIMARY PURPOSE OF AN ABSTRACT BASE CLASS

The primary purpose of an abstract base class is to provide a set of one or more public interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy. The key phrase is “*expected to be found in some derived class further down the inheritance hierarchy.*” This means that as a designer you would employ an abstract class in your application architecture when you want a base class to specify rather than implement behavior and you expect derived classes to actually implement the behavior specified by the base class interface.

OK, why would you want to do this? Why create a class that does nothing but specify a set of interface methods? Good questions! The short answer is that abstract classes will comprise the upper tier(s) of your inheritance hierarchy. The upper tier(s) of an inheritance hierarchy is where you expect to find specifications for general behavior found in derived classes which comprise the lower tier(s) of an inheritance hierarchy. The derived classes, at some point, must provide implementations for those abstract methods specified in their base classes. Designing application architectures in this fashion — abstractions at the top and concrete implementations at the bottom — enables the architecture to be *extended* to accommodate new functionality rather than modified. This design technique injects a good dose of stability into your application architecture. This and other advanced object-oriented design techniques is discussed in more detail in chapter 24.

EXPRESSING ABSTRACT BASE CLASSES IN UML

Figure 11-10 shows a UML diagram that contains an abstract base class named `AbstractClass`.

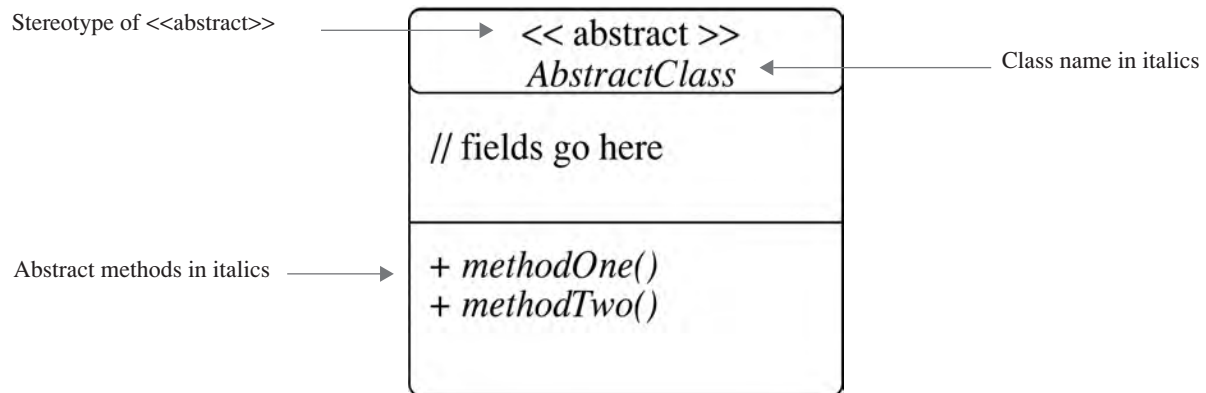


Figure 11-10: Expressing an Abstract Class in the UML

Referring to figure 11-10 — the stereotype `<<abstract>>` is optional but if you draw your UML diagrams by hand it's hard to write in italics, so, it will come in handy. Abstract classes can have fields and concrete methods like normal classes, but abstract methods are shown in italics.

Let's now have a look at a short abstract class inheritance example.

ABSTRACT CLASS EXAMPLE

Figure 11-11 gives the UML class diagram for our example:

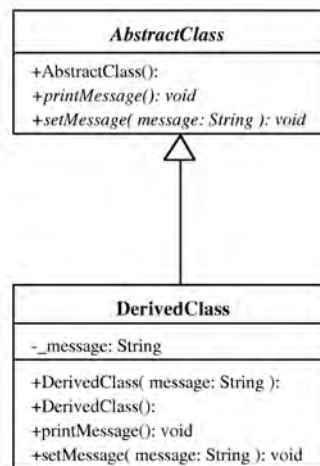


Figure 11-11: UML Class Diagram Showing the `AbstractClass` & `DerivedClass` Inheritance Hierarchy

Referring to figure 11-11 — `AbstractClass` has three methods and no fields. The first method is its default constructor and it is not abstract. (*Constructors cannot be abstract!*) The next two methods, `printMessage()` and `setMessage()` are shown in italics and are therefore abstract methods.

`DerivedClass` inherits from `AbstractClass`. Since `AbstractClass`'s methods are abstract, and have no implementation, `DerivedClass` must provide an implementation for them. `DerivedClass`'s methods are in plain font indicating they have implementations.

Now, if for some reason, you as a designer decided to create a class that inherited from `DerivedClass`, and you defer the implementation of one or more of `DerivedClass`'s methods to that class, then `DerivedClass` would itself have to be declared to be an abstract class. I just wanted to mention this because in most situations you will have more than a two-tiered inheritance hierarchy as I have used here in this simple example.

Let's now take a look at the code for these two classes. Example 11.9 gives the code for `AbstractClass`.

11.9 *AbstractClass.java*

```

1      public abstract class AbstractClass {
2
3          public AbstractClass(){
4              System.out.println("AbstractClass object created!");
5          }
6
7          public abstract void printMessage();
8
9          public abstract void setMessage(String message);
10
11     }
```

The important points to note here is that on line 1 the keyword `abstract` is used to indicate that this is an abstract class definition. The keyword is also used on lines 7 and 9 in the method declarations for `printMessage()` and `setMessage()`. Also note how both the `printMessage()` and `setMessage()` methods are terminated with a semicolon and have no body. (*i.e., no curly braces!*) Example 11.10 gives the code for `DerivedClass`.

11.10 *DerivedClass.java*

```

1      public class DerivedClass extends AbstractClass {
2          private String _message = null;
3
4          public DerivedClass(String message){
5              _message = "Message from DerivedClass: " + message;
6              System.out.println("DerivedClass object created!");
7          }
8
9          public DerivedClass(){
10             this("DerivedClass message!");
11         }
12
13         public void printMessage(){
14             System.out.println(_message);
15         }
16
17         public void setMessage(String message){
18             _message = "Message from DerivedClass: " + message;
19         }
20     }
```

`DerivedClass` extends `AbstractClass` with the `extends` keyword on line 1. Since this version of `AbstractClass` has no `_message` field there is no need for `DerivedClass` to call a base class constructor with the *super* keyword. `DerivedClass` provides an implementation for each of `AbstractClass`'s abstract methods. Let's take a look now at the test driver program that will exercise these two classes.

11.11 *DriverApplication.java*

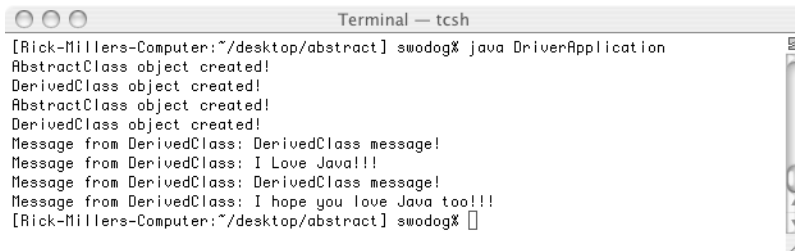
```

1      public class DriverApplication {
2          public static void main(String[] args){
3              AbstractClass ac1 = new DerivedClass();
4              DerivedClass dc1 = new DerivedClass();
5              ac1.printMessage();
6              ac1.setMessage("I Love Java!!!");
7              ac1.printMessage();
8              dc1.printMessage();
9              dc1.setMessage("I hope you love Java too!!!");
10             dc1.printMessage();
11         }
12     }
```

Remember, you cannot directly instantiate an abstract class object. On line 3 a reference to an `AbstractClass` type object named `ac1` is declared and initialized to point to a `DerivedClass` object. On line 4 a reference to a `DerivedClass` type object named `dc1` is declared and initialized to point to a `DerivedClass` object.

Lines 6 through 8 exercises reference `ac1`. The `printMessage()` method is called on line 6 followed by a call to the `setMessage()` method with the String literal “I Love Java!!!” as an argument. The next time the `printMessage()` method is called the new message is printed to the console.

The same series of method calls is performed on the `dc1` reference on lines 10 through 12. Figure 11-12 shows the results of running example 11.11.



```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/abstract] swodog% java DriverApplication
AbstractClass object created!
DerivedClass object created!
AbstractClass object created!
DerivedClass object created!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: I Love Java!!!
Message from DerivedClass: DerivedClass message!
Message from DerivedClass: I hope you love Java too!!!
[Rick-Millers-Computer:~/desktop/abstract] swodog% []
```

Figure 11-12: Results of Running Example 11.11

Quick Review

An abstract method is a method that omits its body and has no implementation behavior. A class that declares one or more abstract methods must be declared to be abstract.

The primary purpose of an abstract class is to provide a specification of a set of one or more class interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy.

Designers employ abstract classes to provide a measure of application architectural stability.

INTERFACES

An interface is a Java construct that functions like an implicit abstract class. In Java, a derived class can extend the behavior of only one class, but it can implement as many interfaces as it requires.

THE PURPOSE OF INTERFACES

The purpose of an interface is to provide a specification for a set of public interface methods. An interface declaration introduces a new data type, just as a class declaration and definition does.

AUTHORIZED INTERFACE MEMBERS

Java interfaces can only contain four types of members. These include:

- Constant Fields — All interface fields are implicitly public, static, and final
- Abstract Methods — All interface methods are implicitly public and abstract
- Nested Class Declarations — These are implicitly public and static
- Nested Interface Declarations — These have the same characteristics as interfaces!

THE DIFFERENCES BETWEEN AN INTERFACE AND AN ABSTRACT CLASS

Table 11-1 summarizes the differences between abstract classes and interfaces.

Abstract Class	Interface
Must be declared to be abstract with the abstract keyword.	Is implicitly abstract and the use of the abstract keyword is discouraged in an interface declaration.
Can contain abstract as well as concrete methods. The concrete methods can be public, private, or protected.	Can only contain abstract methods. All methods in an interface are implicitly public and abstract. Redundantly declaring interface methods to be public and abstract is permitted but discouraged.
Can contain public, private, and protected variable data fields and constants.	Can only contain public constant fields. All fields in an interface are implicitly public, static, and final. Redundantly declaring an interface field to be public, static, and final is allowed but discouraged.
Can contain nested class and interface declarations.	Can contain nested class and interface declarations. These inner class and interface declarations are implicitly public and static.
Can extend one class but implement many interfaces.	Can extend many interfaces.

Table 11-1: Differences Between Abstract Classes And Interfaces

EXPRESSING INTERFACES IN UML

Interfaces are expressed in the UML in two ways as is shown in figure 11-13.

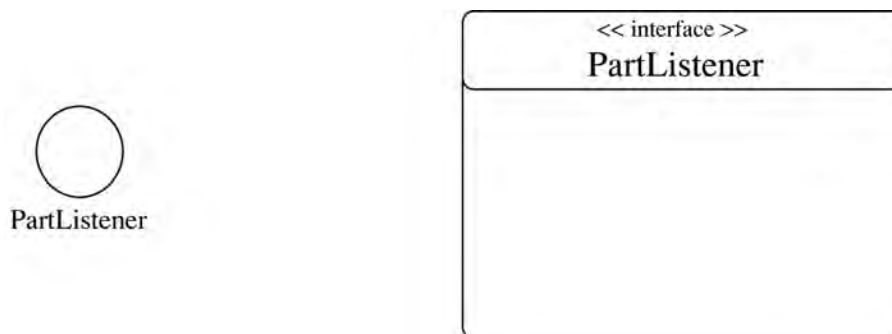


Figure 11-13: Two Types of UML Interface Diagrams

One way to show an interface in the UML is with a simple circle with the name of the interface close by. The second way involves the use of an ordinary class diagram that includes the stereotype << interface >>. Each of these diagrams, the circle and the class diagram, can be used to represent the use of interfaces in an inheritance hierarchy as is discussed in the following section.

EXPRESSING REALIZATION IN A UML CLASS DIAGRAM

When a class implements an interface it is said to be *realizing* the interface. Interface realization is expressed in the UML in two distinct forms: 1) the *simple form* where the circle is used to represent the interface and is combined with an association line to create a *lollipop diagram*, or 2) the *expanded form* where an ordinary class diagram is used to represent the interface. Figure 11-14 illustrates the use of the lollipop diagram to convey the simple form of realization and figure 11-15 shows an example of the expanded form of realization.

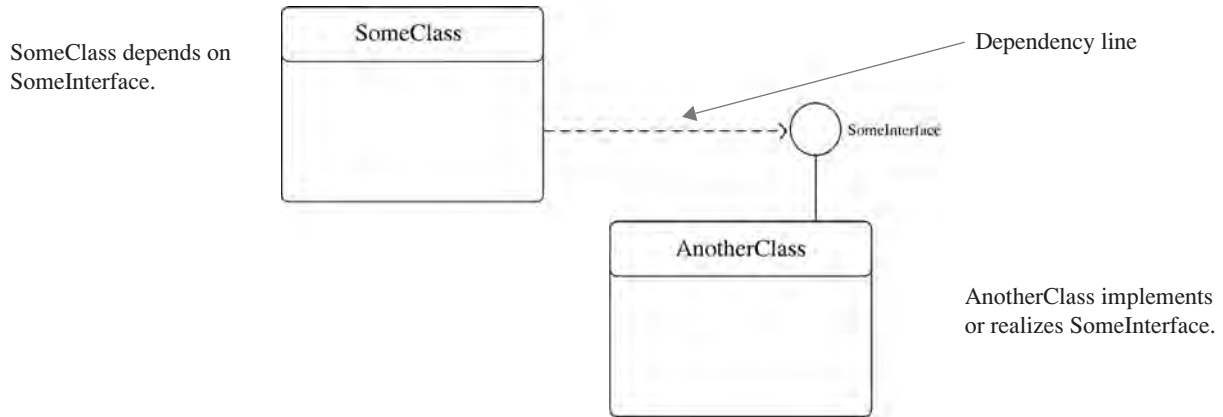


Figure 11-14: UML Diagram Showing the Simple Form of Realization

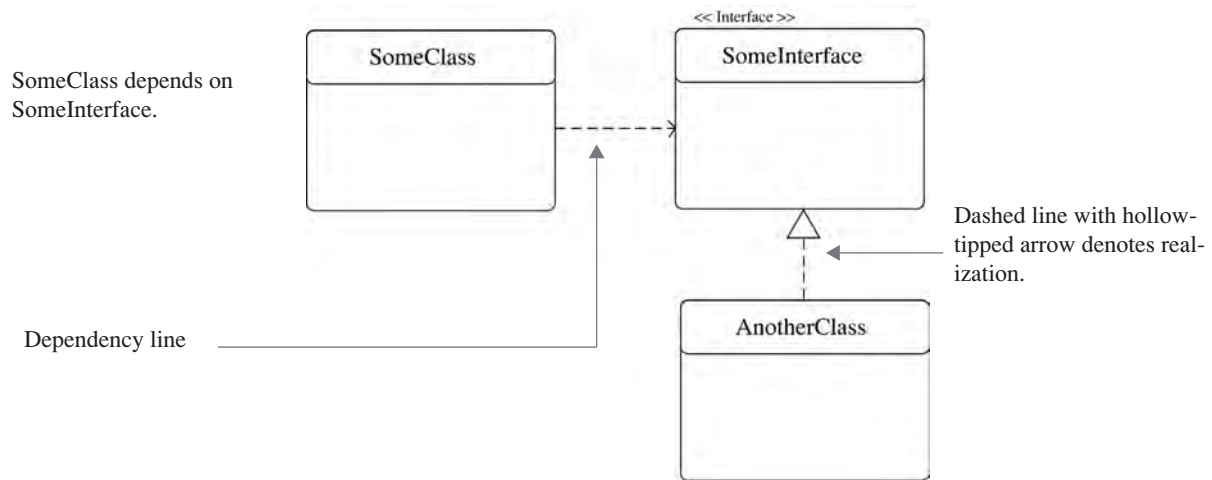


Figure 11-15: UML Diagram Showing the Expanded Form of Realization

AN INTERFACE EXAMPLE

Let's turn our attention to a simple example of an interface in action. Figure 11-16 gives the UML diagram of the interface named `MessagePrinter` and a class named `MessagePrinterClass` that implements the `MessagePrinter` interface. The source code for these two classes is given in examples 11.12 and 11.13.

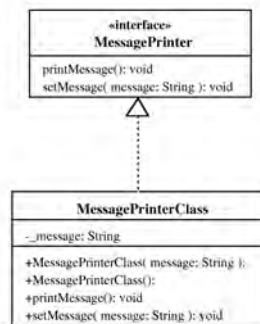


Figure 11-16: UML Diagram Showing the MessagePrinterClass Realizing the MessagePrinter Interface

11.12 MessagePrinter.java

```

1 interface MessagePrinter {
2     void printMessage();
3     void setMessage(String message);
4 }

```

11.13 MessagePrinterClass.java

```

1 public class MessagePrinterClass implements MessagePrinter {
2     private String _message = null;
3
4     public MessagePrinterClass(String message){
5         _message = message;
6     }
7
8     public MessagePrinterClass(){
9         this("Default message is boring!");
10    }
11
12    public void printMessage(){
13        System.out.println(_message);
14    }
15
16    public void setMessage(String message){
17        _message = message;
18    }
19 }

```

11.14 DriverApplication.java

```

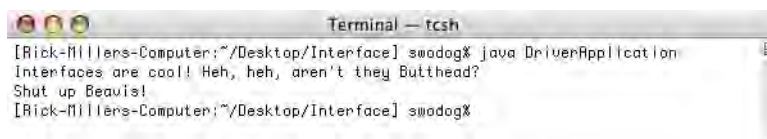
1 public class DriverApplication {
2     public static void main(String[] args){
3         MessagePrinter mp1 = new MessagePrinterClass("Interfaces are cool!" +
4             " Heh, heh, aren't they Butthead?");
5         MessagePrinterClass mpc1 = new MessagePrinterClass("Shut up Beavis!");
6
7         mp1.printMessage();
8         mpc1.printMessage();
9     }
10 }

```

As you can see from example 11.12 the MessagePrinter interface is short and simple. All it does is declare the two interface methods printMessage() and setMessage(). The implementation of these interface methods is left to any class that implements the MessagePrinter interface as the MessagePrinterClass does in example 11.13.

Example 11.14 gives the test driver program for this example. As you can see on line 3 you can declare an interface type reference. I called this one mp1. Although you cannot instantiate an interface directly with the new operator you can initialize an interface type reference to point to an object of any concrete class that implements the interface. The only methods you can call on the object via the interface type reference are those methods specified by the interface. You could of course cast to a different type if required but you must strive to minimize the need to cast in this manner.

Figure 11-17 gives the results of running example 11.14.



```

Terminal - tcsh
[Rick-Millers-Computer:~/Desktop/Interface] swadog% java DriverApplication
Interfaces are cool! Heh, heh, aren't they Butthead?
Shut up Beavis!
[Rick-Millers-Computer:~/Desktop/Interface] swadog%

```

Figure 11-17: Results of Running Example 11.14

Quick Review

The purpose of an interface is to specify a set of public interface methods. Interfaces can have four types of members: 1) constants, 2) abstract methods, 3) nested classes, and 4) nested interfaces. Classes can inherit from or extend only one other class, but they can implement as many interfaces as are required. Interfaces can extend as many other interfaces as necessary.

CONTROLLING HORIZONTAL & VERTICAL ACCESS

The term *horizontal access* is used to describe the level of access an object of one type has to the members (*i.e.*, *fields and methods*) of another type. This topic was discussed in detail in chapter 9. The term *vertical access* refers to the level of access a derived class has to its base class members. In both cases access is controlled by the three Java access modifiers *public*, *protected*, and *private*. There is a fourth level of access, known as *package*, that is inferred by the omission of an access modifier. In other words, if you declare a class member, be it a field or method, and don't specify it as being either public, protected, or private, then it is, by default, package. (*The exception to this rule is when you are declaring an interface in which case the members are public by default.*)

The behavior of protected and package accessibility is dictated by what package the classes in question belong. Figure 11-18 gives a diagram showing five classes named ClassOne, ClassTwo, ClassThree, ClassFour, and ClassFive. ClassOne, ClassTwo, and ClassThree belong to Package A, and ClassFour and ClassFive belong to Package B.

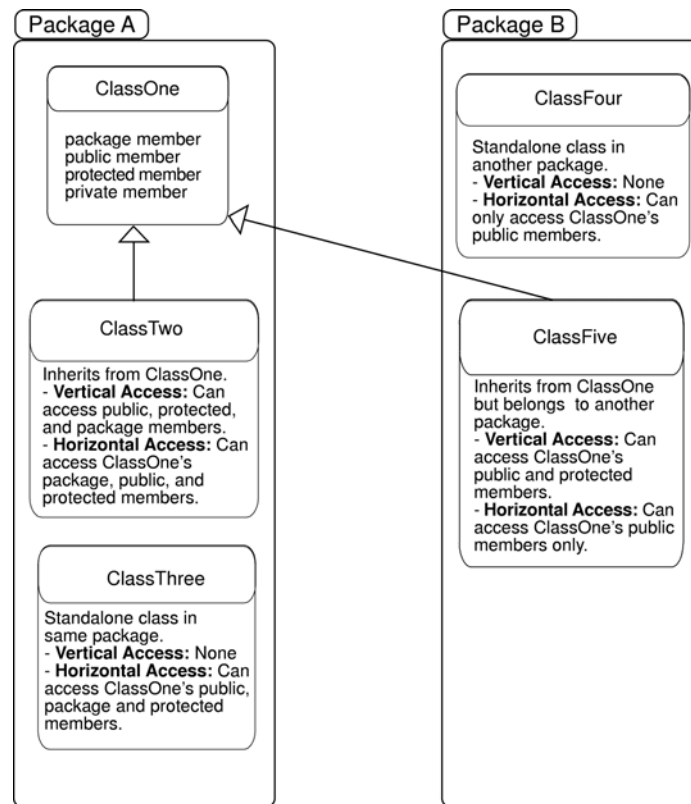


Figure 11-18: Horizontal And Vertical Access In Multi-Package Environment

Referring to figure 11-18 — access behavior is discussed relative to ClassOne from both the horizontal and vertical access perspective. ClassOne contains one package member, one public member, one protected member, and one private member.

ClassTwo inherits from ClassOne and belongs to the same package. From a vertical perspective, as a subclass of ClassOne it has access to ClassOne's public, protected, and package members. From a horizontal perspective it can also access ClassOne's public, protected, and package members.

ClassThree is a standalone class in the same package as ClassOne. Since it is not a subclass of ClassOne it has no vertical access. Horizontally it can access ClassOne's public, protected, and package members.

ClassFour is a standalone class in a different package than ClassOne. It too is not a subclass of ClassOne and therefore has no vertical access. Horizontally it can only access ClassOne's public members.

ClassFive is a subclass of ClassOne but belongs to another package. It has vertical access to ClassOne's public and protected members. It has horizontal access to ClassOne's public members.

The following examples give the code for each class shown in figure 11-18 with offending lines commented out.

11.15 ClassOne.java

```

1     package packageA;
2
3     public class ClassOne {
4         void methodOne(){}
5         public void methodTwo(){}
6         protected void methodThree(){}
7         private void methodFour(){}
8     }

```

11.16 ClassTwo.java

```

1     package packageA;
2
3     public class ClassTwo extends packageA.ClassOne {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             c1.methodOne();
8             c1.methodTwo();
9             c1.methodThree();
10            // c1.MethodFour(); // Error - no horizontal access to private member
11            methodOne();
12            methodTwo();
13            methodThree();
14            // methodFour(); // Error - no vertical access to private member
15        }
16    }

```

11.17 ClassThree.java

```

1     package packageA;
2
3     public class ClassThree {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             c1.methodOne();
8             c1.methodTwo();
9             c1.methodThree();
10            // c1.methodFour(); // Error - no horizontal access to private member
11        }
12    }
13

```

11.18 ClassFour.java

```

1     package packageB;
2
3     public class ClassFour {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             // c1.methodOne(); // Error - no horizontal access to package member in different package
8             c1.methodTwo();
9             // c1.methodThree(); // Error - no horizontal access to protected member
10            // c1.methodFour(); // Error - no horizontal access to private member
11        }
12    }

```

11.19 ClassFive.java

```

1     package packageB;
2
3     public class ClassFive extends packageA.ClassOne {
4         packageA.ClassOne c1 = new packageA.ClassOne();
5
6         void f(){
7             // c1.methodOne(); // Error - no horizontal access to outside package member
8             c1.methodTwo();
9             // c1.methodThree(); // Error - no horizontal access to protected member outside package
10            // c1.MethodFour(); // Error - no horizontal access to private member
11            // methodOne(); // Error - no vertical access outside package
12            methodTwo();
13            methodThree();
14            // methodFour(); // Error - no vertical access - private method
15        }
16    }

```

To test each class simply compile each one. Experiment by removing the comments from an offending line and then compiling and studying the resulting error message(s).

Quick Review

Horizontal and vertical access is controlled via the access specifiers `public`, `protected`, and `private`. A class member without an explicit access modifier declared has package accessibility by default. Essentially, classes belonging to the same package can access each other's `public`, `protected`, and package members horizontally. If a subclass belongs to the same package as its base class it has vertical access to its `public`, `protected`, and package members.

Classes belonging to different packages have horizontal access to each other's `public` members. A subclass in one package whose base class belongs to another package has horizontal access to the base class's `public` methods only but vertical access to both its `public` and `protected` members.

FINAL CLASSES & METHODS

Sometimes you want to prevent classes from being extended or individual methods of a particular class from being overridden. The keyword `final` is used for these purposes. When used to declare a class it prevents that class from being extended. When used to declare a method it prevents the method from being overridden in a derived class.

You cannot use the keyword `final` in combination with the keyword `abstract` for obvious reasons.

Quick Review

Use the `final` keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

POLYMORPHIC BEHAVIOR

A good definition of polymorphism is “*The ability to operate on and manipulate different derived objects in a uniform way...*” (Sadr) Add to this the following amplification: “*Without polymorphism, the developer ends up writing code consisting of large case or switch statements. This is in fact the litmus test for polymorphism. The existence of a switch statement that selects an action based upon the type of an object is often a warning sign that the developer has failed to apply polymorphic behavior effectively.*” (Booch)

Polymorphic behavior is easy to understand. In a nutshell it is simply the act of using the set of public interface methods defined for a particular class (*or interface*) to interact with that class's (*or interface's*) derived classes. When you write code you need some level of apriori knowledge about the type of objects your code will manipulate. In essence, you have to set the bar to some level, meaning that at some point in your code you need to make an assumption about the type of objects with which you are dealing and the behavior they manifest. An object's type, as you know, is important because it specifies the set of operations (*methods*) that are valid for objects of that type (*and subtypes*).

Code that's written to take advantage of polymorphic behavior is generally cleaner, easier to read, easier to maintain, and easier to extend. If you find yourself casting a lot you are not writing polymorphic code. If you use the `instanceof` operator frequently to determine object types then you are not writing polymorphic code. Polymorphic behavior is the essence of object-oriented programming.

Quick Review

Polymorphic behavior is achieved in a program by targeting a set of operations (*methods*) specified by a base class or interface and manipulating their derived class objects via those methods. This uniform treatment of derived

class objects results in cleaner code that's easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

INHERITANCE EXAMPLE: EMPLOYEE

This section offers an inheritance example that extends, once again, the functionality of the Person class given in chapter 9. Figure 11-19 gives the UML diagram.

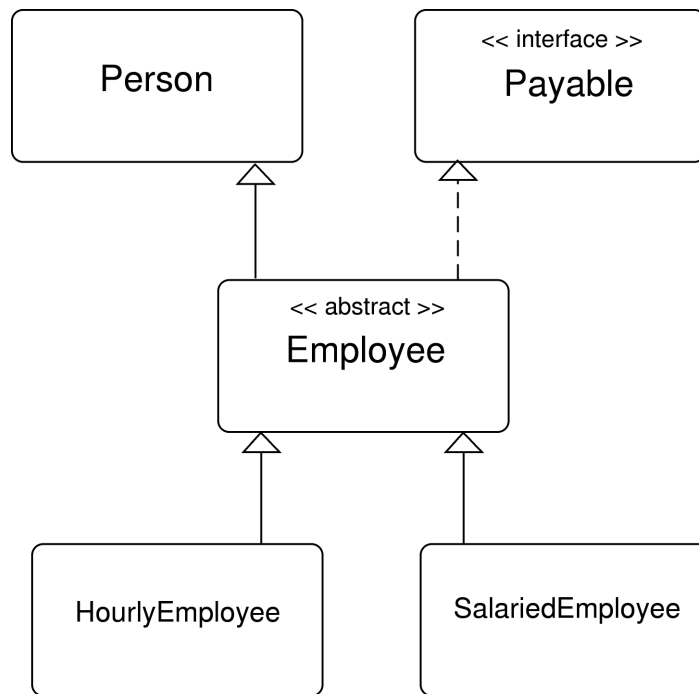


Figure 11-19: Employee Class Inheritance Hierarchy

Referring to figure 11-19 — The Employee class extends Person and implements the Payable interface. However, in this example, as you will see below, the implementation of the method specified in the Payable interface, `pay()`, is deferred by the Employee class to its derived classes. Because the `pay()` method is not actually implemented in the Employee class the Employee class must be declared to be abstract.

HourlyEmployee and SalariedEmployee extend the functionality of Employee. Each of these classes will implement the `pay()` method in their own special way.

From a polymorphic point of view you could write a program that uses these classes in several ways, it just depends on which set of interface methods you want to target. For instance, you could write a program that contains an array of Person references. Each of these Person references could then be initialized to point to an HourlyEmployee object or a SalariedEmployee object. In either case the only methods you can call, without casting, on these objects via a Person reference are those public methods specified by the Person class.

Another approach would be to declare an array of Payable references. Then again you could initialize each Payable reference to point to either an HourlyEmployee object or a SalariedEmployee object. Now the only method you can call on each of these objects, without casting, is the `pay()` method.

A third approach would be to declare an array of Employee references and initialize each reference to point to either an HourlyEmployee object or a SalariedEmployee object. In this scenario you could then call any method specified by Person, Payable, and Employee. This is the approach taken in the EmployeeTestApp program listed below.

The code for each of these classes (*except Person which was shown earlier in the chapter*) is given in examples 11.20 through 11.24.

11.20 Payable.java

```

1 interface Payable {
2     double pay();
3 }

```

11.21 Employee.java

```

1 public abstract class Employee extends Person implements Payable {
2     private String _employee_number = null;
3
4     public Employee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
5                     int dob_day, String gender, String employee_number){
6         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender);
7         _employee_number = employee_number;
8     }
9
10    public String getEmployeeNumber(){
11        return _employee_number;
12    }
13
14    public String getEmployeeInfo(){
15        return getEmployeeNumber() + " " + getFirstName() + " " + getLastName();
16    }
17 }

```

11.22 HourlyEmployee.java

```

1 public class HourlyEmployee extends Employee {
2     private double _hours_worked;
3     private double _hourly_wage_rate;
4
5     public HourlyEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
6                           int dob_day, String gender, String employee_number,
7                           double hourly_wage_rate, double hours_worked){
8         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
9         _hourly_wage_rate = hourly_wage_rate;
10        _hours_worked = hours_worked;
11    }
12
13    public double pay(){
14        return _hourly_wage_rate * _hours_worked;
15    }
16 }

```

11.23 SalariedEmployee.java

```

1 public class SalariedEmployee extends Employee {
2     private double _annual_salary;
3
4     public SalariedEmployee(String f_name, String m_name, String l_name, int dob_year, int dob_month,
5                             int dob_day, String gender, String employee_number, double annual_salary){
6         super(f_name, m_name, l_name, dob_year, dob_month, dob_day, gender, employee_number);
7         _annual_salary = annual_salary;
8     }
9
10    public double pay(){
11        return _annual_salary/24; // 24 pay periods
12    }
13 }

```

11.24 EmployeeTestApp.java

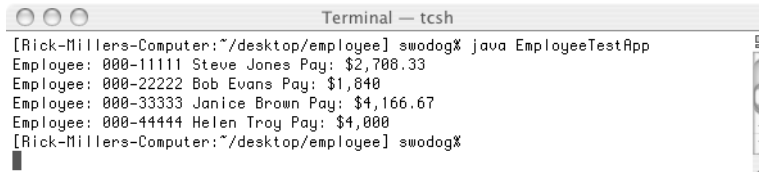
```

1 import java.text.*;
2
3 public class EmployeeTestApp {
4     public static void main(String[] args){
5         Employee[] employees = new Employee[4];
6         NumberFormat number_format = NumberFormat.getInstance();
7         number_format.setMaximumFractionDigits(2);
8
9         employees[0] = new SalariedEmployee("Steve", "J", "Jones", 1983, 3, 4, "M",
10                                           "000-11111", 65000.00);
11        employees[1] = new HourlyEmployee("Bob", "E", "Evans", 1992, 1, 2, "M",
12                                          "000-22222", 23.00, 80.00);
13        employees[2] = new SalariedEmployee("Janice", "A", "Brown", 1983, 3, 4, "F",
14                                          "000-33333", 100000.00);
15        employees[3] = new HourlyEmployee("Helen", "O", "Troy", 1946, 4, 8, "F",
16                                          "000-44444", 50.00, 80.00);
17
18        for(int i = 0; i<employees.length; i++){
19            System.out.println("Employee: " + employees[i].getEmployeeInfo() +
20                              " Pay: " + "$" + number_format.format(employees[i].pay()));
21        }
22    }
23 }

```

Referring to example 11.24 — the `EmployeeTestApp` program declares an array of `Employee` references on line 5 named `employees`. On lines 9 through 16 it initializes each `Employee` reference to point to either an `HourlyEmployee` or `SalariedEmployee` object.

In the `for` statement on line 18 each `Employee` object is manipulated polymorphically via the interface specified by the `Employee` class. (*Which includes the interfaces inherited from `Person` and `Payable`.*) The results of running this program are shown in figure 11-20.



```
Terminal — tcsh
[Rick-Millers-Computer: ~/desktop/employee] swodog% java EmployeeTestApp
Employee: 000-11111 Steve Jones Pay: $2,788.33
Employee: 000-22222 Bob Evans Pay: $1,840
Employee: 000-33333 Janice Brown Pay: $4,166.67
Employee: 000-44444 Helen Troy Pay: $4,800
[Rick-Millers-Computer: ~/desktop/employee] swodog%
```

Figure 11-20: Results of Running Example 11.24

INHERITANCE EXAMPLE: AIRCRAFT ENGINE SIMULATION

As the television chef Emeril Lagasse would say, “Bam! — kick it up a notch!”. This example expands on the aircraft engine simulation originally presented in chapter 10. Here the concepts of inheritance are fused with compositional design to yield a truly powerful combination.

In this example you will also be introduced to the concept of a listener. A listener is a component that responds to events. Listeners are used extensively in the AWT and Swing packages to respond to different GUI events. (*You will be formally introduced to GUI programming in chapter 12.*) This example also utilizes the `Vector` and `Hashtable` collection classes from the `java.util` package and the `synchronized` keyword which is used in several methods in the `Part` class to ensure atomic access to certain objects by client methods. The collection classes are discussed in detail in chapter 17 and the `synchronized` keyword is discussed in detail in chapter 16.

AIRCRAFT ENGINE SIMULATION UML DIAGRAM

Let’s start by discussing the UML diagram for the engine simulation shown in figure 11-21. As you can see from the class diagram this version of the aircraft engine simulation contains decidedly more classes than its chapter 10 cousin. The primary class in the inheritance hierarchy is the `Part` class. Essentially, every class is a `Part` with the exception of the `PartStatus` and `PartEvent` classes. `Part` implicitly inherits from the `java.lang.Object` class but this relationship is not shown on the diagram. The concept of a `Part` encompasses individual as well as composite parts. The `Part` class provides the general functionality for both single as well as composite part objects.

Two other Java API classes that are shown in the class diagram include `EventObject` and `EventListener`. The `PartEvent` class inherits from `EventObject` and the `PartListener` interface inherits from the `EventListener` interface.

The interfaces `IPump`, `ISensor`, and `IEngine` specify the methods that must be implemented by parts of these types. The classes `Pump`, `Sensor`, and `Engine` inherit the basic functionality provided by the `Part` class and implement the appropriate interface.

The `FuelPump` and `OxygenSensor` classes provide concrete implementations for the `Pump` and `Sensor` classes respectively. The `SimpleEngine` class extends `Engine` and implements an interface named `PartListener`. Note here that although `SimpleEngine` implements the `PartListener` interface, other parts could implement the `PartListener` interface as required.

It should also be noted that this program is incomplete at this stage. I have only provided implementations for three concrete classes: `FuelPump`, `OxygenSensor`, and `SimpleEngine`. You will have the opportunity to add parts and functionality to this program in one of the Suggested Projects found at the end of this chapter.

SIMULATION OPERATIONAL DESCRIPTION

The Part class provides the general functionality for all Part objects. A Part object can be a single, atomic part or it can be an aggregate of several Part objects. The Parts comprising a composite Part object are stored in a Part's `_sub_parts` Hashtable. Part objects are added to the `_sub_parts` Hashtable by calling the `addSubPart()` method.

A particular subclass of Part may be a PartListener if it implements the PartListener interface. The PartListener interface specifies three methods that correspond to state changes on Sensors, Pumps, and the working status of a part. A Part that is also a PartListener will need to register itself with the Part of interest. Take for example the SimpleEngine class. Referring to the code listing given in example 11.35 — on lines 9 and 10 in the constructor, the SimpleEngine class creates two subparts and adds them to the `_sub_parts` Hashtable by calling the `addSubPart()` method. On lines 11 and 12 it then gets a reference to each of its subparts by calling the `getSubPart()` method and registering itself as a PartListener for each of its subparts.

To see how the PartListener mechanism works let's step through a pump speed change. In the `start()` method of the SimpleEngine class the speed of its FuelPump subpart is set to 100. (*see line 31 of example 11.35*) The `setSpeed()` method is called on a Pump interface but, because the FuelPump class overrides Pump's `setSpeed()` method, it is the FuelPump's version of `setSpeed()` that gets called. Now, go to the FuelPump class listed in example 11.33 and find the `setSpeed()` method. If the speed desired falls within the valid range of a FuelPump object then the Pump version of `setSpeed()` is called via the *super* object. Go now to the Pump class listed in example 11.30 and find its version of the `setSpeed()` method. It's sole job is to call the `firePumpSpeedChangeEvent()` method with the indicated speed as an argument. The `firePumpSpeedChangeEvent()` method is located in the Part class so go now to example 11.25 and find it.

The `firePumpSpeedChangeEvent()` does several things. First, it checks to see if its `_part_listeners` Vector is empty. If it is then it doesn't have any interested PartListeners. If it has one or more interested PartListeners it clones its `_part_listeners` Vector object in a synchronized block to prevent sporadic behavior, creates a new PartEvent object, converts the `listeners_copy` vector to an Enumeration, and then steps through each element of the Enumeration calling the `pumpSpeedChangePerformed()` method on each interested PartListener object. This is when the `pumpSpeedChangePerformed()` method is called on the SimpleEngine object resulting in the console output you see when you run the program.

TAKE A DEEP BREATH AND RELAX!

At this point, since you have yet to be formally introduced to them, some of the concepts regarding the use of Vectors, Hashtables, and Enumerations will seem cryptic. The PartListener firing mechanism is also difficult at first to understand. If you don't fully grasp the complete workings of this complex example right now don't worry. My intent is to challenge you to dive into the code and trace the execution. That's how you learn how to program!

COMPILING THE AIRCRAFT ENGINE SIMULATION CODE

The easiest way to compile all the aircraft engine simulation code is to put the source files in one directory and issue the following command:

```
javac -source 1.4 *.java
```

You need to add the `-source 1.4` option because of the use of the `assert` keyword in the Part class.

COMPLETE AIRCRAFT ENGINE SIMULATION CODE LISTING

11.25 Part.java

```

1      import java.util.*;
2
3      public abstract class Part {
4
5          /*****
6              private attributes
7          *****/
8          private PartStatus _status = null;
9          private String _part_name = null;

```

```

10     private Vector _part_listeners = null;
11     private Hashtable _sub_parts = null;
12
13
14     /*****
15     constructor
16     *****/
17     public Part(PartStatus status, String part_name){
18         _status = status;
19         _part_name = part_name;
20         System.out.println("Part object created!");
21     }
22
23     /*****
24     default constructor
25     *****/
26     public Part(String part_name){
27         _status = PartStatus.WORKING;
28         _part_name = part_name;
29         System.out.println("Part object created!");
30     }
31
32     /*****
33     public interface methods
34     *****/
35
36     public PartStatus getPartStatus(){ return _status; }
37
38     public String getPartName(){ return _part_name; }
39
40     public void setPartStatus(PartStatus status){
41         _status = status;
42         firePartStatusChangeEvent();
43     }
44
45     public boolean isWorking(){
46         if(_sub_parts != null){ // there are subparts
47             for(Enumeration e = getSubPartsEnumeration(); e.hasMoreElements();){
48                 if(((Part) e.nextElement()).isWorking()) {
49                     continue; //great!
50                 }else{
51                     setPartStatus(PartStatus.NOT_WORKING);
52                     break;
53                 }
54             }
55         }
56         System.out.println(getPartName() + " " + " isWorking status = " + _status.isWorking());
57         return _status.isWorking();
58     }
59
60     public void addPartListener(PartListener pl){
61         if(_part_listeners == null){
62             _part_listeners = new Vector();
63             _part_listeners.add(pl);
64         }else {
65             _part_listeners.add(pl);
66         }
67     }
68
69     public void removePartListener(PartListener pl){
70         if(_part_listeners == null){
71             ; //do nothing
72         } else {
73             if(_part_listeners.isEmpty()){
74                 ; //do nothing
75             } else {
76                 _part_listeners.removeElement(pl);
77             }
78         }
79     }
80
81     public void enable(){
82         setPartStatus(PartStatus.WORKING);
83     }
84
85     public void disable(){
86         setPartStatus(PartStatus.NOT_WORKING);
87     }
88
89
90     public void fireSensorReadingChangeEvent(double reading){

```

```

91     if(_part_listeners.isEmpty()){
92         ; //there's nothing to do!
93     } else {
94         Vector listeners_copy = null;
95         synchronized(this){
96             listeners_copy = (Vector)_part_listeners.clone();
97         }
98         PartEvent pe = new PartEvent(this, _part_name, reading);
99         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
100             ((PartListener) (e.nextElement())).sensorReadingChangePerformed(pe);
101         }
102     }
103 }
104
105 public void firePumpSpeedChangeEvent(double speed){
106     if(_part_listeners.isEmpty()){
107         ; //there's nothing to do!
108     } else {
109         Vector listeners_copy = null;
110         synchronized(this){
111             listeners_copy = (Vector)_part_listeners.clone();
112         }
113         PartEvent pe = new PartEvent(this, _part_name, speed);
114         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
115             ((PartListener) (e.nextElement())).pumpSpeedChangePerformed(pe);
116         }
117     }
118 }
119
120 public void firePartStatusChangeEvent(){
121     if(_part_listeners.isEmpty()){
122         ; //there's nothing to do!
123     } else {
124         Vector listeners_copy = null;
125         synchronized(this){
126             listeners_copy = (Vector)_part_listeners.clone();
127         }
128         PartEvent pe = new PartEvent(this, _part_name);
129         for(Enumeration e = listeners_copy.elements(); e.hasMoreElements();){
130             ((PartListener) (e.nextElement())).partStatusChangePerformed(pe);
131         }
132     }
133 }
134
135 public void addSubPart(String name, Part part){
136     if(_sub_parts == null){
137         _sub_parts = new Hashtable();
138         _sub_parts.put(name, part);
139     }else{
140         _sub_parts.put(name, part);
141     }
142 }
143
144 public Part getSubPart(String name){
145     assert (_sub_parts != null); //can't call if no subparts exist!
146     return (Part)_sub_parts.get(name);
147 }
148
149 public Enumeration getSubPartsEnumeration(){
150     assert (_sub_parts != null );
151     return _sub_parts.elements();
152 }
153 }

```

11.26 IPump.java

```

1 public interface IPump {
2     public void setSpeed(int speed);
3     public void incrementSpeed(int i);
4     public int getSpeed();
5 }

```

11.27 ISensor.java

```

1 public interface ISensor {
2     public float getReading();
3     public void setReading(float reading);
4     public void incrementReading(float reading);
5 }

```

11.28 IEngine.java

```

1     public interface IEngine {
2         public void start();
3         public void stop();
4         public void incrementThrust();
5         public void decrementThrust();
6     }

```

11.29 PartListener.java

```

1     import java.util.*;
2
3     public interface PartListener extends EventListener {
4         public void sensorReadingChangePerformed(PartEvent pe);
5         public void pumpSpeedChangePerformed(PartEvent pe);
6         public void partStatusChangePerformed(PartEvent pe);
7     }

```

11.30 Pump.java

```

1     public abstract class Pump extends Part implements IPump {
2
3         public Pump(PartStatus status, String name){
4             super(status, name);
5             System.out.println("Pump object created!");
6         }
7
8         public Pump(String name){
9             super(name);
10            System.out.println("Pump object created!");
11        }
12
13        public void setSpeed(int speed){
14            firePumpSpeedChangeEvent(speed);
15        }
16
17        public void incrementSpeed(int increment){
18            firePumpSpeedChangeEvent(getSpeed());
19        }
20
21        public int getSpeed() { return 0; }
22
23    }

```

11.31 Sensor.java

```

1     public abstract class Sensor extends Part implements ISensor {
2
3         public Sensor(String name){
4             super(name);
5             System.out.println("Sensor object created!");
6         }
7
8         public Sensor(PartStatus status, String name){
9             super(status, name);
10            System.out.println("Sensor object created!");
11        }
12
13
14        /*****
15         ISensor Interface Methods
16         *****/
17
18        public float getReading(){
19            return 0;
20        }
21
22        public void setReading(float reading){
23            fireSensorReadingChangeEvent(reading);
24        }
25
26        public void incrementReading(float increment) {
27            fireSensorReadingChangeEvent(getReading());
28        }
29    }

```

11.32 Engine.java

```

1      public abstract class Engine extends Part implements IEngine {
2
3          public Engine(PartStatus status, String name){
4              super(status, name);
5              System.out.println("Engine object created!");
6          }
7
8          public Engine(String name){
9              super(name);
10             System.out.println("Engine object created!");
11         }
12
13         /*****
14             IEngine Interface Method Stubs
15
16             Default behavior will do nothing. They must be
17             overridden in a derived class.
18             *****/
19         public void start(){ }
20         public void stop(){ }
21         public void incrementThrust(){ }
22         public void decrementThrust(){ }
23     }
24

```

11.33 FuelPump.java

```

1      public class FuelPump extends Pump{
2
3          private static final int MIN_SPEED = 0;
4          private static final int MAX_SPEED = 1000;
5
6          private int _speed = 0;
7
8          public FuelPump(PartStatus status, String name){
9              super(status, name);
10             System.out.println("Fuel Pump " + name + " created!");
11         }
12
13         public FuelPump(String name){
14             super(name);
15             System.out.println("Fuel Pump " + name + " created!");
16         }
17
18         public void setSpeed(int speed){
19             if((speed >= MIN_SPEED) && (speed <= MAX_SPEED)){
20                 _speed = speed;
21                 super.setSpeed(speed);
22             }else{
23                 System.out.println("Pump speed cannot exceed specified range:" +
24                     MIN_SPEED + " - " + MAX_SPEED);
25             }
26         }
27
28         public void incrementSpeed(int i){
29             if(((_speed + i) >= MIN_SPEED) && ((_speed + i) <= MAX_SPEED)){
30                 _speed += i;
31                 super.incrementSpeed(i);
32             } else {
33                 System.out.println("Pump speed cannot exceed specified range: " + MIN_SPEED + " - " + MAX_SPEED);
34             }
35         }
36
37         public int getSpeed() { return _speed; }
38     }
39

```

11.34 OxygenSensor.java

```

1      public class OxygenSensor extends Sensor {
2          private static final int MIN_READING = 0;
3          private static final int MAX_READING = 1000;
4          private float _reading = 0;
5
6          public OxygenSensor(String name){
7              super(name);
8              System.out.println("Oxygen Sensor " + name + " object created!");
9              _reading = 0;

```

```

10     }
11
12     public OxygenSensor(PartStatus status, String name){
13         super(status, name);
14         System.out.println("Oxygen Sensor " + name + " object created!");
15         _reading = 0;
16     }
17
18
19     public float getReading(){ return _reading; }
20
21     public void setReading(float reading){
22         if( (reading >= MIN_READING) && (reading <= MAX_READING)){
23             _reading = reading;
24             super.setReading(reading); //call Sensor method to fire sensor reading change event
25         } else {
26             System.out.println("Warning: Attempt to set Oxygen Sensor outside valid range: " +
27                 MIN_READING + " - " + MAX_READING);
28         }
29     }
30 }
31
32     public void incrementReading(float increment){
33         if(((_reading + increment) >= MIN_READING) && ((_reading + increment) <= MAX_READING)){
34             _reading += increment;
35             super.setReading(_reading); //call Sensor method to fire sensor reading change event
36         } else {
37             System.out.println("Warning: Attempt to set Oxygen Sensor outside valid range: " +
38                 MIN_READING + " - " + MAX_READING);
39         }
40     }
41 } // end OxygenSensor class
42

```

11.35 SimpleEngine.java

```

1     public class SimpleEngine extends Engine implements PartListener {
2
3
4         private String[] part_names = {"FuelPump 1", "Oxygen Sensor 1"};
5
6         public SimpleEngine(PartStatus status, String name){
7             super(status, name);
8
9             addSubPart(part_names[0], new FuelPump(part_names[0]));
10            addSubPart(part_names[1], new OxygenSensor(part_names[1]));
11            getSubPart(part_names[0]).addPartListener(this);
12            getSubPart(part_names[1]).addPartListener(this);
13            System.out.println("SimpleEngine object created!");
14        }
15
16        public SimpleEngine(String name){
17            super(name);
18
19            addSubPart(part_names[0], new FuelPump(part_names[0]));
20            addSubPart(part_names[1], new OxygenSensor(part_names[1]));
21            getSubPart(part_names[0]).addPartListener(this);
22            getSubPart(part_names[1]).addPartListener(this);
23            System.out.println("SimpleEngine object created!");
24        }
25
26        /*****
27         * IEngine Interface Method Implementations
28         *****/
29        public void start(){
30            if(isWorking()){
31                ((Pump) getSubPart(part_names[0])).setSpeed(100);
32            }
33        }
34
35
36        public void stop(){
37            ((Pump) getSubPart(part_names[0])).setSpeed(0);
38        }
39
40        public void incrementThrust(){
41            ((Pump) getSubPart(part_names[0])).incrementSpeed(25);
42        }
43
44        public void decrementThrust(){
45

```

```

46         ((Pump) getSubPart(part_names[0])).incrementSpeed(-25);
47     }
48
49
50     /*****
51     PartListener Interface Methods
52     *****/
53
54     public void sensorReadingChangePerformed(PartEvent pe){
55         System.out.println("SimpleEngine: sensorReadingChangePerformed() method called!");
56         System.out.println(pe.getPartName() + " reading changed to " + pe.getValue());
57     }
58
59
60     public void pumpSpeedChangePerformed(PartEvent pe){
61         System.out.println("SimpleEngine: pumpSpeedChangePerformed() method called!");
62         System.out.println(pe.getPartName() + " speed changed to " + pe.getValue());
63     }
64
65     public void partStatusChangePerformed(PartEvent pe){
66         System.out.println("SimpleEngine: partStatusChangePerformed() method called!");
67         System.out.println(pe.getPartName() + pe.statusToString());
68     }
69
70 }

```

11.36 PartEvent.java

```

1     import java.util.*;
2
3     public class PartEvent extends EventObject {
4         private String _part_name = null;
5         private double _value = 0.0;
6
7         public PartEvent(Object source, String part_name, double value){
8             super(source);
9             _part_name = part_name;
10            _value = value;
11        }
12
13        public PartEvent(Object source, String part_name){
14            super(source);
15            _part_name = part_name;
16        }
17
18        public String getPartName(){ return _part_name; }
19
20        public double getValue(){ return _value; }
21
22        public String toString(){ return super.toString() + " " + getPartName();}
23
24        public String statusToString(){
25            return ((Part) source).isWorking() ? " is working properly!" : " has malfunctioned!";
26        }
27
28    }

```

11.37 PartStatus.java

```

1     public final class PartStatus {
2         public static final PartStatus NOT_WORKING = new PartStatus(false);
3         public static final PartStatus WORKING = new PartStatus(true);
4
5         private boolean _is_working = false;
6
7         private PartStatus(boolean is_working){
8             _is_working = is_working;
9         }
10
11        public boolean isWorking(){ return _is_working; }
12    }

```

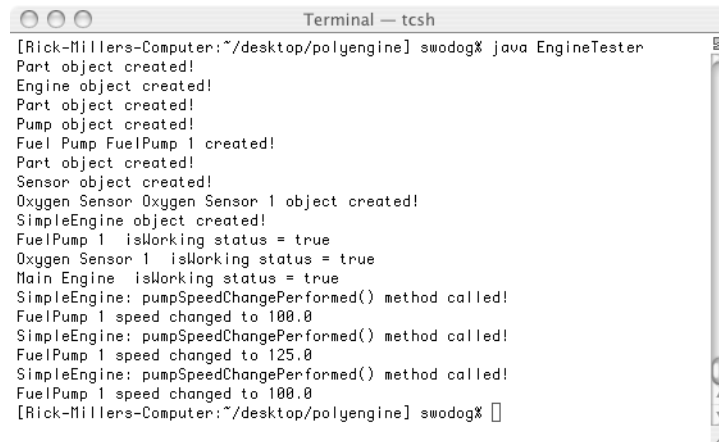
11.38 EngineTester.java

```

1     public class EngineTester {
2         public static void main(String[] args){
3             IEngine engine = new SimpleEngine("Main Engine");
4
5             engine.start();
6             engine.incrementThrust();
7             engine.decrementThrust();
8         }
9     }

```


Figure 11-22 shows the results of running example 11.38.



```

Terminal — tcsh
[Rick-Millers-Computer:~/desktop/polyengine] swodog% java EngineTester
Part object created!
Engine object created!
Part object created!
Pump object created!
Fuel Pump FuelPump 1 created!
Part object created!
Sensor object created!
Oxygen Sensor Oxygen Sensor 1 object created!
SimpleEngine object created!
FuelPump 1 isWorking status = true
Oxygen Sensor 1 isWorking status = true
Main Engine isWorking status = true
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 100.0
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 125.0
SimpleEngine: pumpSpeedChangePerformed() method called!
FuelPump 1 speed changed to 100.0
[Rick-Millers-Computer:~/desktop/polyengine] swodog%

```

Figure 11-22: Results of Running Example 11.38

TERMS & DEFINITIONS

Term	Definition
class	A Java construct that introduces a new data type. A class definition can have package, public, protected, or private members. Class methods can implement behavior.
type	The specification of a set of valid operations over a set of objects. You can only perform an operation on an object if the operation is supported by the object's type.
interface	A Java construct that introduces a new data type. Interface methods only specify behavior — they have no implementation.
base class	A class whose behavior is inherited (<i>extended</i>) by a derived class.
derived class	A class that inherits (<i>extends</i>) the behavior of an existing class. A derived class can extend only one class but can implement as many interfaces as necessary.
abstract method	A class method that omits its body and therefore has no implementation. Interface methods are abstract by default.
abstract class	A class that declares one or more abstract methods. Also, a class that implements an interface but defers implementing the interface's method(s) to a derived class further down the inheritance hierarchy.

Table 11-2: Chapter 11 Terms and Definitions

SUMMARY

Inheritance serves three essential purposes: 1) it is an object-oriented design mechanism that enables you to think and reason about your program structure in terms of generalized and specialized class behavior, 2) it provides you with a measure of code reuse within your program by locating common class behavior in base classes, and 3) it serves as a means to incrementally develop programs over time.

Classes that belong to an inheritance hierarchy participate in an “is a” relationship between themselves and their chain of base classes. This “is a” relationship is transitive in the direction of specialized to generalized classes but not vice versa.

The Java class and interface constructs are each used to create new, user-defined data types. The interface construct is used to specify a set of authorized type methods and omits method behavior; the class construct is used to specify a set of authorized type methods and their behavior. A class construct, like an interface, can omit the bodies of one or more of its methods, however, such methods must be declared to be abstract. A class that declares one or more of its methods to be abstract must itself be declared to be an abstract class. Abstract class objects cannot be created with the new operator.

A base class implements default behavior in the form of methods that can be inherited by derived classes. There are three reference -> object combinations: 1) if the base class is a concrete class, meaning it is not abstract, then a base class reference can point to a base class object, 2) a base class reference can point to a derived class object, and 3) a derived class reference can point to a derived class object.

Reference variables have an associated type. Method calls to an object pointed to by a reference will succeed without casting so long as the type of the reference supports the method you are trying to call. You can force, or coerce, the compiler to treat a reference to an object of one type as if it were a reference to an object of another. This is extremely helpful in some circumstances but, as a rule, casting should be used sparingly. Also, casting only works if the object really is of the type you are casting it to.

Derived classes can override base class behavior by providing overriding methods. An overriding method is a method in a derived class that has the same method signature as the base class method it is intending to override. Overriding methods can be called polymorphically via a base class reference that points to a derived class object.

An abstract method is a method that omits its body and has no implementation behavior. A class that declares one or more abstract methods must be declared to be abstract.

The primary purpose of an abstract class is to provide a specification of a set of one or more class interface methods whose implementations are expected to be found in some derived class further down the inheritance hierarchy.

Designers employ abstract classes to provide a measure of application architectural stability.

The purpose of an interface is to specify a set of public interface methods. Interfaces can have four types of members: 1) constants, 2) abstract methods, 3) nested classes, and 4) nested interfaces. Classes can inherit from or extend only one other class, but they can implement as many interfaces as are required. Interfaces can extend as many other interfaces as necessary.

Horizontal and vertical access is controlled via the access specifiers public, protected, and private. A class member without an explicit access modifier declared has package accessibility by default. Essentially, classes belonging to the same package can access each other’s public, protected, and package members horizontally. If a subclass belongs to the same package as its base class it has vertical access to its public, protected, and package members.

Classes belonging to different packages have horizontal access to each other’s public members. A subclass in one package whose base class belongs to another package has horizontal access to the base class’s public methods only but vertical access to both its public and protected members.

Use the *final* keyword to stop the inheritance mechanism or prevent base class methods from being overridden in derived classes.

Polymorphic behavior is achieved in a program by targeting a set of operations (*methods*) specified by a base class or interface and manipulating their derived class objects via those methods. This uniform treatment of derived class objects results in cleaner code that’s easier to extend and maintain. Polymorphic behavior is the essence of object-oriented programming.

Skill-Building Exercises

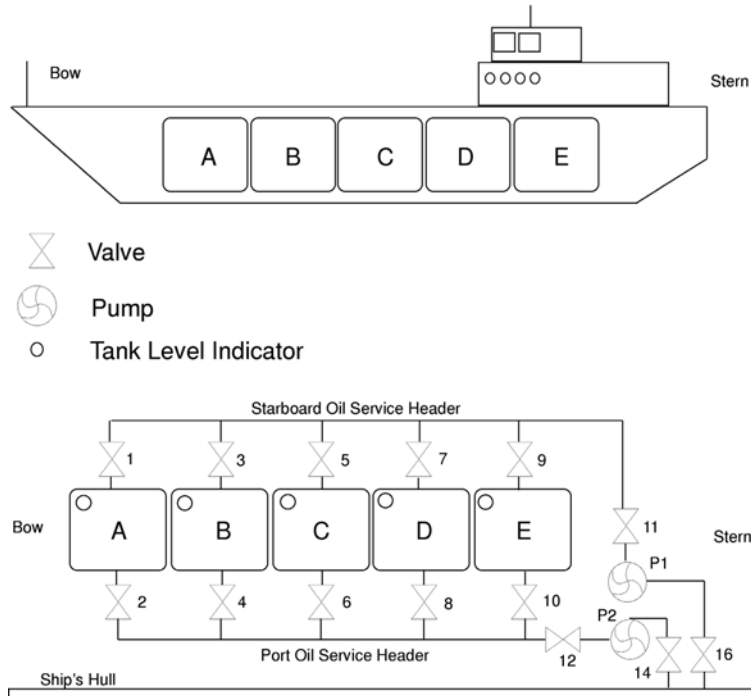
1. **Question:** How many classes can a derived class extend? How many interfaces can a derived class implement?
2. **Question:** How many interfaces can a derived interface extend?

3. **API Familiarization:** Look up the Vector, Hashtable, and Enumeration classes in the Java API documentation and study their usage. This will help you to understand their usage in the aircraft engine simulation code presented in this chapter.
4. **Simple Inheritance:** Write a small program to test the effects of inheritance. Create a class named ClassA that implements the following methods: a(), b(), and c(). Each method should print a short text message to the screen. Create a default constructor for ClassA that prints a message to the screen announcing the creation of a ClassA object. Next, create a class named ClassB that extends ClassA. Give ClassB a default constructor that announces the creation of a ClassB object. In a test driver program create three references. Two of the references should be of type ClassA and the third should be of type ClassB. Initialize the first reference to point to a ClassA object, initialize the second reference to point to a ClassB object, and initialize the third reference to point to a ClassB object as well. Via each of the references call the methods a(), b(), and c(). Run the test driver program and note the results.
5. **Overriding Methods:** Reusing some of the code you created in the previous exercise create another class named ClassC that extends ClassA and provides overriding methods for each of ClassA's methods a(), b(), and c(). Have each of the methods defined in ClassC print short messages to the screen. In the test driver program declare three references, the first two of type ClassA and the third of type ClassC. Initialize the first reference to point to an object of type ClassA, the second to point to an object of type ClassC, and the third to point to an object of ClassC as well. Via each of the references call the methods a(), b(), and c(). Run the test driver program and note the results.
6. **Abstract Classes:** Create an abstract class named AbstractClassA and give it a default constructor and three abstract methods named a(), b(), and c(). Create another class named ClassB that extends ClassA. Provide overriding methods for each of the abstract methods declared in ClassA. Each overriding method should print a short text message to the screen. Create a test driver program that declares two references. The first reference should be of type ClassA, the second reference should be of type ClassB. Initialize the first reference to point to an object of type ClassB, and the second reference to point to an object of ClassB as well. Via each reference call the methods a(), b(), and c(). Run the program and note the results.
7. **Interfaces:** Convert the abstract class you created in the previous exercise to an interface. What changes did you have to make to the code? Compile your interface and test driver program code, re-run the program, and note the results.
8. **Mental Exercise:** Consider the scenario then answer the questions that follow: Given an abstract base class named ClassOne with the following abstract public interface methods a(), b(), and c(). Given a class named ClassTwo that derives from ClassOne, provides implementations for each of ClassOne's abstract methods, and defines one additional method named d(). Now, you have two references. One is of type ClassOne, the other of type ClassTwo. What methods can be called via the ClassOne reference without casting? Likewise, what methods can be called via the ClassTwo reference without casting?

SUGGESTED PROJECTS

1. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine constructor call. Refer to the code supplied in examples 11.25 through 11.38.
2. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine start() method.
3. **Draw Sequence Diagram:** Draw a UML sequence diagram of the SimpleEngine incrementThrust() method.

- Extend Functionality:** Extend the functionality of the Employee example given in this chapter. Create a subclass named PartTimeEmployee that extends HourlyEmployee. Limit the number of hours a PartTimeEmployee can have to 30 hours per pay period.
- Extend Functionality:** Extend the functionality of the aircraft engine simulation program given in this chapter. Create several more types of pumps and sensors. For instance, you might create an OilPump or an OilTemperature-Sensor class. What other types of parts might you want to model in the simulation? Create different types of engines that consist of different types of parts. Use your imagination!
- Oil Tanker Pumping System:** Design and create an oil tanker pumping system simulation. Assume your tanker ship has five oil cargo compartments as shown in the diagram below.



Each compartment can be filled and drained from either the port or starboard service header. The oil pumping system consists of 14 valves numbered 1 through 16. Even-numbered valves are located on the port side of the ship and odd-numbered valves are located on the starboard side of the ship. (*Valve numbers 13 and 15 are not used.*)

The system also consists of two pumps that can be run in two speeds, slow or fast speed, and in two directions, drain and fill. When a pump is running in the drain direction it is taking a suction from the tank side, when running in the fill direction it is taking a suction from the hull side. Assume a pumping capacity of 1000 gallons per minute in fast mode.

Each tank contains one tank level indicator that is a type of sensor. The indicators sense a continuous tank level from 0 (empty) to 100,000 gallons.

Your program should let you drain and fill the oil compartments by opening and closing valves and starting and setting pump speeds. For instance, to fill tank A quickly you could open valves 1, 2, 11, 12, 14 & 16, and start pumps P1 and P2 in the fill direction in the fast mode.

SELF-TEST QUESTIONS

1. What are the three essential purposes of inheritance?

2. A class that belongs to an inheritance hierarchy participates in what type of relationship with its base class?
3. Describe the relationship between the terms interface, class, and type.
4. How do you express generalization and specialization in a UML class diagram? You may draw a picture to answer the question.
5. Describe how to override a base class method in a derived class.
6. Why would it be desirable to override a base class method in a derived class?
7. What's the difference between an ordinary method and an abstract method?
8. How many abstract methods must a class have before it must be declared to be an abstract class?
9. List several differences between classes and interfaces.
10. How do you express an abstract class in a UML class diagram?
11. Hi, I'm a class that declares a set of interface methods but fails to provide an implementation for the those methods. What type of class am I?
12. List the four authorized members of an interface.
13. What two ways can you express realization in a UML class diagram. You may use pictures to answer the question.
14. How do you call a base class constructor from a derived class constructor?
15. How do you call a base class method, other than a constructor, from a derived class method?
16. Describe the effects using the access modifiers public, package, private, and the default access package, has on classes belonging to the same and different packages. State the effects from the both the horizontal and vertical member access perspectives.
17. What can you do to prevent or stop a class from being inherited?
18. What can you do to prevent a method from being overridden in a derived class?
19. State, in your own words, a good definition for the term polymorphism.

REFERENCES

Grady Booch, et. al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-57168-4

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA, 2003. ISBN: 1-932504-02-8

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Second Edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN: 0-8053-5340-2

Babak Sadr. *Unified Objects: Object-Oriented Programming Using C++*. The IEEE Computer Society, Los Alamitos, CA. ISBN: 0-8186-7733-3

Antero Taivalsaari. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, September 1996, pp. 438 - 479.

Clyde Ruby and Gary T. Levens. *Safely Creating Correct Subclasses without Seeing Superclass Code*. In OOP-SLA '00 Conference Proceedings.

Derek Rayside and Gerard T. Campbell. *An Aristotelian Understanding of Object-Oriented Programming*. OOP-SLA '00 Conference Proceedings.

NOTES
