CHAPTER 3

Inheritance and Polymorphism

Say not you know another entirely,
till you have divided an inheritance with him.
—Johann Kaspar Lavatar, Aphorisms on Man

Chapter Goals

- Superclasses and subclasses
- Inheritance hierarchy
- Polymorphism
- Type compatibility
- Abstract classes
- Interfaces

INHERITANCE

Superclass and Subclass

Inheritance defines a relationship between objects that share characteristics. Specifically it is the mechanism whereby a new class, called a *subclass*, is created from an existing class, called a *superclass*, by absorbing its state and behavior and augmenting these with features unique to the new class. We say that the subclass *inherits* characteristics of its superclass.

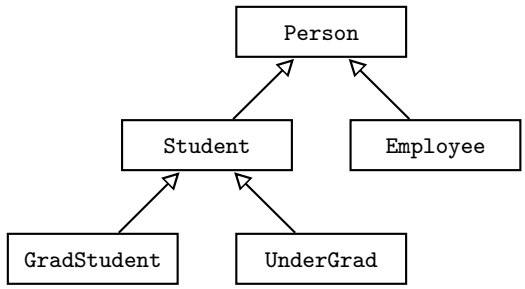
Don't get confused by the names: a subclass is bigger than a superclass—it contains more data and more methods!

Inheritance provides an effective mechanism for code reuse. Suppose the code for a superclass has been tested and debugged. Since a subclass object shares features of a superclass object, the only new code required is for the additional characteristics of the subclass.

Inheritance Hierarchy

A subclass can itself be a superclass for another subclass, leading to an *inheritance hierarchy* of classes.

For example, consider the relationship between these objects: `Person`, `Employee`, `Student`, `GradStudent`, and `UnderGrad`.



For any of these classes, an arrow points to its superclass. The arrow designates an inheritance relationship between classes, or, informally, an *is-a* relationship. Thus, an `Employee` *is-a* `Person`; a `Student` *is-a* `Person`; a `GradStudent` *is-a* `Student`; an `UnderGrad` *is-a* `Student`. Notice that the opposite is not necessarily true: A `Person` may not be a `Student`, nor is a `Student` necessarily an `UnderGrad`.

Note that the *is-a* relationship is transitive: If a `GradStudent` *is-a* `Student` and a `Student` *is-a* `Person`, then a `GradStudent` *is-a* `Person`.

Every subclass inherits the public or protected variables and methods of its superclass (see p. 135). Subclasses may have additional methods and instance variables that are not in the superclass. A subclass may redefine a method it inherits. For example, `GradStudent` and `UnderGrad` may use different algorithms for computing the course grade, and need to change a `computeGrade` method inherited from `Student`. This is called *method overriding*. If part of the original method implementation from the superclass is retained, we refer to the rewrite as *partial overriding* (see p. 135).

Implementing Subclasses

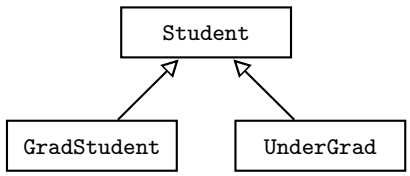
THE `extends` KEYWORD

The inheritance relationship between a subclass and a superclass is specified in the declaration of the subclass, using the keyword `extends`. The general format looks like this:

```
public class Superclass
{
    //private instance variables
    //other data members
    //constructors
    //public methods
    //private methods
}

public class Subclass extends Superclass
{
    //additional private instance variables
    //additional data members
    //constructors  (Not inherited!)
    //additional public methods
    //inherited public methods whose implementation is overridden
    //additional private methods
}
```

For example, consider the following inheritance hierarchy:



The implementation of the classes may look something like this (discussion follows the code):

```
public class Student
{
    //data members
    public final static int NUM_TESTS = 3;
    private String name;
    private int[] tests;
    private String grade;

    //constructor
    public Student()
    {
        name = "";
        tests = new int[NUM_TESTS];
        grade = "";
    }

    //constructor
    public Student(String studName, int[] studTests, String studGrade)
    {
        name = studName;
        tests = studTests;
        grade = studGrade;
    }

    public String getName()
    { return name; }

    public String getGrade()
    { return grade; }

    public void setGrade(String newGrade)
    { grade = newGrade; }

    public void computeGrade()
    {
        if (name.equals(""))
            grade = "No grade";
        else if (getTestAverage() >= 65)
            grade = "Pass";
        else
            grade = "Fail";
    }
```

```
    public double getTestAverage()
    {
        double total = 0;
        for (int score : tests)
            total += score;
        return total/NUM_TESTS;
    }
}

public class UnderGrad extends Student
{
    public UnderGrad()   //default constructor
    { super(); }

    //constructor
    public UnderGrad(String studName, int[] studTests, String studGrade)
    { super(studName, studTests, studGrade); }

    public void computeGrade()
    {
        if (getTestAverage() >= 70)
            setGrade("Pass");
        else
            setGrade("Fail");
    }
}

public class GradStudent extends Student
{
    private int gradID;

    public GradStudent()      //default constructor
    {
        super();
        gradID = 0;
    }

    //constructor
    public GradStudent(String studName, int[] studTests,
            String studGrade, int gradStudID)
    {
        super(studName, studTests, studGrade);
        gradID = gradStudID;
    }

    public int getID()
    { return gradID; }

    public void computeGrade()
    {
        //invokes computeGrade in Student superclass
        super.computeGrade();
        if (getTestAverage() >= 90)
            setGrade("Pass with distinction");
    }
}
```

INHERITING INSTANCE METHODS AND VARIABLES

The semantics of talking about inheritance is tricky. Subclasses do not inherit the private instance variables or private methods of their superclasses. However, objects of subclasses contain memory for those private instance variables, even though they can't directly access them. A subclass inherits all the public and protected data members of its parent.

In the `Student` example, the `UnderGrad` and `GradStudent` subclasses inherit all of the methods of the `Student` superclass. Notice, however, that the `Student` instance variables `name`, `tests`, and `grade` are private and are therefore not inherited or directly accessible to the methods in the `UnderGrad` and `GradStudent` subclasses. A subclass can, however, directly invoke the public accessor and mutator methods of the superclass. Thus, both `UnderGrad` and `GradStudent` use `getTestAverage`. Additionally, both `UnderGrad` and `GradStudent` use `setGrade` to access indirectly—and modify—`grade`.

If, instead of `private`, the access specifier for the instance variables in `Student` were `public` or `protected`, then the subclasses could directly access these variables. The keyword `protected` is not part of the AP Java subset.

Classes on the same level in a hierarchy diagram do not inherit anything from each other (for example, `UnderGrad` and `GradStudent`). All they have in common is the identical code they inherit from their superclass.

METHOD OVERRIDING AND THE `super` KEYWORD

Any public method in a superclass can be overridden in a subclass by defining a method with the same return type and signature (name and parameter types). For example, the `computeGrade` method in the `UnderGrad` subclass overrides the `computeGrade` method in the `Student` superclass.

Sometimes the code for overriding a method includes a call to the superclass method. This is called *partial overriding*. Typically this occurs when the subclass method wants to do what the superclass does, plus something extra. This is achieved by using the keyword `super` in the implementation. The `computeGrade` method in the `GradStudent` subclass partially overrides the matching method in the `Student` class. The statement

```
super.computeGrade();
```

signals that the `computeGrade` method in the superclass should be invoked here. The additional test

```
if (getTestAverage() >= 90)
    ...
```

allows a `GradStudent` to have a grade `Pass with distinction`. Note that this option is open to `GradStudents` only.

NOTE

Private methods cannot be overridden.

CONSTRUCTORS AND `super`

Constructors are never inherited! If no constructor is written for a subclass, the superclass default constructor with no parameters is generated. If the superclass does

Be sure to provide at least one constructor when you write a subclass. Constructors are never inherited from the superclass.

not have a default (zero-parameter) constructor, but only a constructor with parameters, a compiler error will occur. If there is a default constructor in the superclass, inherited data members will be initialized as for the superclass. Additional instance variables in the subclass will get a default initialization—`0` for primitive types and `null` for reference types.

A subclass constructor can be implemented with a call to the `super` method, which invokes the superclass constructor. For example, the default constructor in the `UnderGrad` class is identical to that of the `Student` class. This is implemented with the statement

```
super();
```

The second constructor in the `UnderGrad` class is called with parameters that match those in the constructor of the `Student` superclass.

```
public UnderGrad(String studName, int[] studTests, String studGrade)
{ super(studName, studTests, studGrade); }
```

For each constructor, the call to `super` has the effect of initializing the instance variables `name`, `tests`, and `grade` exactly as they are initialized in the `Student` class.

Contrast this with the constructors in `GradStudent`. In each case, the instance variables `name`, `tests`, and `grade` are initialized as for the `Student` class. Then the new instance variable, `gradID`, must be explicitly initialized.

```
public GradStudent()
{
    super();
    gradID = 0;
}

public GradStudent(String studName, int[] studTests,
            String studGrade, int gradStudID)
{
    super(studName, studTests, studGrade);
    gradID = gradStudID;
}
```

NOTE

1. If `super` is used in the implementation of a subclass constructor, it *must* be used in the first line of the constructor body.
2. If no constructor is provided in a subclass, the compiler provides the following default constructor:

```
public SubClass()
{
    super();      //calls default constructor of superclass
}
```

Rules for Subclasses

- A subclass can add new private instance variables.
- A subclass can add new public, private, or static methods.
- A subclass can override inherited methods.
- A subclass may not redefine a public method as private.
- A subclass may not override static methods of the superclass.
- A subclass should define its own constructors.
- A subclass cannot directly access the private members of its superclass. It must use accessor or mutator methods.

Declaring Subclass Objects

When a superclass object is declared in a client program, that reference can refer not only to an object of the superclass, but also to objects of any of its subclasses. Thus, each of the following is legal:

```
Student s = new Student();
Student g = new GradStudent();
Student u = new UnderGrad();
```

This works because a `GradStudent` *is-a* `Student`, and an `UnderGrad` *is-a* `Student`.

Note that since a `Student` is not necessarily a `GradStudent` nor an `UnderGrad`, the following declarations are *not* valid:

```
GradStudent g = new Student();
UnderGrad u = new Student();
```

Consider these valid declarations:

```
Student s = new Student("Brian Lorenzen", new int[] {90,94,99},
        "none");
Student u = new UnderGrad("Tim Broder", new int[] {90,90,100},
        "none");
Student g = new GradStudent("Kevin Cristella",
        new int[] {85,70,90}, "none", 1234);
```

Suppose you make the method call

```
s.setGrade("Pass");
```

The appropriate method in `Student` is found and the new grade assigned. The method calls

```
g.setGrade("Pass");
```

and

```
u.setGrade("Pass");
```

achieve the same effect on `g` and `u` since `GradStudent` and `UnderGrad` both inherit the `setGrade` method from `Student`. The following method calls, however, won't work:

```
int studentNum = s.getID();
int underGradNum = u.getID();
```

Neither `Student s` nor `UnderGrad u` inherit the `getID` method from the `GradStudent` class: A superclass does not inherit from a subclass.

Now consider the following valid method calls:

```
s.computeGrade();
g.computeGrade();
u.computeGrade();
```

Since `s`, `g`, and `u` have all been declared to be of type `Student`, will the appropriate method be executed in each case? That is the topic of the next section, *polymorphism*.

NOTE

The initializer list syntax used in constructing the array parameters—for example, `new int[] {90,90,100}`— will not be tested on the AP exam.

POLYMORPHISM

A method that has been overridden in at least one subclass is said to be *polymorphic*. An example is `computeGrade`, which is redefined for both `GradStudent` and `UnderGrad`.

Polymorphism is the mechanism of selecting the appropriate method for a particular object in a class hierarchy. The correct method is chosen because, in Java, method calls are always determined by the type of the *actual object*, not the type of the object reference. For example, even though `s`, `g`, and `u` are all declared as type `Student`, `s.computeGrade()`, `g.computeGrade()`, and `u.computeGrade()` will all perform the correct operations for their particular instances. In Java, the selection of the correct method occurs *during the run of the program*.

Dynamic Binding (Late Binding)

Making a run-time decision about which instance method to call is known as *dynamic binding* or *late binding*. Contrast this with selecting the correct method when methods are *overloaded* (see p. 99) rather than overridden. The compiler selects the correct overloaded method at compile time by comparing the methods' signatures. This is known as *static binding*, or *early binding*. In polymorphism, the actual method that will be called is not determined by the compiler. Think of it this way: The compiler determines *if* a method can be called (i.e., is it legal?), while the run-time environment determines *how* it will be called (i.e., which overridden form should be used?).

Example 1

```
Student s = null;
Student u = new UnderGrad("Tim Broder", new int[] {90,90,100},
        "none");
Student g = new GradStudent("Kevin Cristella",
        new int[] {85,70,90}, "none", 1234);
System.out.print("Enter student status: ");
System.out.println("Grad (G), Undergrad (U), Neither (N)");
String str = IO.readString();       //read user input
if (str.equals("G"))
    s = g;
else if (str.equals("U"))
    s = u;
else
    s = new Student();
s.computeGrade();
```

When this code fragment is run, the `computeGrade` method used will depend on the type of the actual object `s` refers to, which in turn depends on the user input.

Example 2

```
public class StudentTest
{
    public static void computeAllGrades(Student[] studentList)
    {
        for (Student s : studentList)
            if (s != null)
                s.computeGrade();
    }

    public static void main(String[] args)
    {
        Student[] stu = new Student[5];
        stu[0] = new Student("Brian Lorenzen",
                        new int[] {90,94,99}, "none");
        stu[1] = new UnderGrad("Tim Broder",
                        new int[] {90,90,100}, "none");
        stu[2] = new GradStudent("Kevin Cristella",
                        new int[] {85,70,90}, "none", 1234);
        computeAllGrades(stu);
    }
}
```

Polymorphism applies only to overridden methods in subclasses.

Here an array of five `Student` references is created, all of them initially null. Three of these references, `stu[0]`, `stu[1]`, and `stu[2]`, are then assigned to actual objects. The `computeAllGrades` method steps through the array invoking for each of the objects the appropriate `computeGrade` method, using dynamic binding in each case. The null test in `computeAllGrades` is necessary because some of the array references could be null.

Using `super` in a Subclass

A subclass can call a method in its superclass by using `super`. Suppose that the superclass method then calls another method that has been overridden in the subclass. By polymorphism, the method that is executed is the one in the subclass. The computer keeps track and executes any pending statements in either method.

Example

```
public class Dancer
{
    public void act()
    {
        System.out.print (" spin");
        doTrick();
    }

    public void doTrick()
    {
        System.out.print (" float");
    }
}
```

```
public class Acrobat extends Dancer
{
    public void act()
    {
        super.act();
        System.out.print (" flip");
    }

    public void doTrick()
    {
        System.out.print (" somersault");
    }
}
```

Suppose the following declaration appears in a class other than `Dancer` or `Acrobat`:

```
Dancer a = new Acrobat();
```

What is printed as a result of the call `a.act()`?

When `a.act()` is called, the `act` method of `Acrobat` is executed. This is an example of polymorphism. The first line, `super.act()`, goes to the `act` method of `Dancer`, the superclass. This prints `spin`, then calls `doTrick()`. Again, using polymorphism, the `doTrick` method in `Acrobat` is called, printing `somersault`. Now, completing the `act` method of `Acrobat`, `flip` is printed. So what all got printed?

```
spin somersault flip
```

NOTE

Even though there are no constructors in either the `Dancer` or `Acrobat` classes, the declaration

```
Dancer a = new Acrobat();
```

compiles without error. This is because `Dancer`, while not having an explicit superclass, has an implicit superclass, `Object`, and gets its default (no-argument) constructor slotted into its code. Similarly the `Acrobat` class gets this constructor slotted into its code.

The statement `Dancer a = new Acrobat();` will not compile, however, if the `Dancer` class has at least one constructor with parameters but no default constructor.

TYPE COMPATIBILITY

Downcasting

Consider the statements

```
Student s = new GradStudent();
GradStudent g = new GradStudent();
int x = s.getID();             //compile-time error
int y = g.getID();             //legal
```

Both `s` and `g` represent `GradStudent` objects, so why does `s.getID()` cause an error? The reason is that `s` is of type `Student`, and the `Student` class doesn't have a `getID` method. At compile time, only nonprivate methods of the `Student` class can appear

to the right of the dot operator when applied to `s`. Don't confuse this with polymorphism: `getID` is not a polymorphic method. It occurs in just the `GradStudent` class and can therefore be called only by a `GradStudent` object.

The error shown above can be fixed by casting `s` to the correct type:

```
int x = ((GradStudent) s).getID();
```

Since `s` (of type `Student`) is actually representing a `GradStudent` object, such a cast can be carried out. Casting a superclass to a subclass type is called a *downcast*.

NOTE

1. The outer parentheses are necessary:

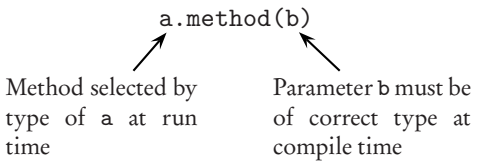
   ```
   int x = (GradStudent) s.getID();
   ```

   will still cause an error, despite the cast. This is because the dot operator has higher precedence than casting, so `s.getID()` is invoked before `s` is cast to `GradStudent`.
2. The statement

   ```
   int y = g.getID();
   ```

   compiles without problem because g is declared to be of type `GradStudent`, and this is the class that contains `getID`. No cast is required.

Type Rules for Polymorphic Method Calls



- For a declaration like

  ```
  Superclass a = new Subclass();
  ```

  the type of `a` at compile time is `Superclass`; at run time it is `Subclass`.
- At compile time, `method` must be found in the class of `a`, that is, in `Superclass`. (This is true whether the method is polymorphic or not.) If `method` cannot be found in the class of `a`, you need to do an explicit cast on `a` to its actual type.
- For a polymorphic method, at run time the actual type of `a` is determined—`Subclass` in this example—and `method` is selected from `Subclass`. This could be an inherited method if there is no overriding method.
- The type of parameter `b` is checked at compile time. You may need to do an explicit cast to the subclass type to make this correct.

The ClassCastException

The ClassCastException is a run-time exception thrown to signal an attempt to cast an object to a class of which it is not an instance.

```
Student u = new UnderGrad();
System.out.println((String) u);    //ClassCastException
                                  //u is not an instance of String
int x = ((GradStudent) u).getID(); //ClassCastException
                                  //u is not an instance of GradStudent
```

ABSTRACT CLASSES

Abstract Class

An *abstract class* is a superclass that represents an abstract concept, and therefore should not be instantiated. For example, a maze program could have several different maze components—paths, walls, entrances, and exits. All of these share certain features (e.g., location, and a way of displaying). They can therefore all be declared as subclasses of the abstract class MazeComponent. The program will create path objects, wall objects, and so on, but no instances of MazeComponent.

An abstract class may contain *abstract methods*. An abstract method has no implementation code, just a header. The rationale for an abstract method is that there is no good default code for the method. Every subclass will need to override this method, so why bother with a meaningless implementation in the superclass? The method appears in the abstract class as a placeholder. The implementation for the method occurs in the subclasses. If a class contains any abstract methods, it *must* be declared an abstract class.

The abstract Keyword

An abstract class is declared with the keyword abstract in the header:

```
public abstract class AbstractClass
{    ...
```

The keyword extends is used as before to declare a subclass:

```
public class SubClass extends AbstractClass
{    ...
```

If a subclass of an abstract class does not provide implementation code for all the abstract methods of its superclass, it too becomes an abstract class and must be declared as such to avoid a compile-time error:

```
public abstract class SubClass extends AbstractClass
{    ...
```

Here is an example of an abstract class, with two concrete (nonabstract) subclasses.

```
public abstract class Shape
{
    private String name;

    //constructor
    public Shape(String shapeName)
    { name = shapeName; }

    public String getName()
    { return name; }

    public abstract double area();
    public abstract double perimeter();

    public double semiPerimeter()
    { return perimeter() / 2; }
}

public class Circle extends Shape
{
    private double radius;

    //constructor
    public Circle(double circleRadius, String circleName)
    {
        super(circleName);
        radius = circleRadius;
    }

    public double perimeter()
    { return 2 * Math.PI * radius; }

    public double area()
    { return Math.PI * radius * radius; }
}

public class Square extends Shape
{
    private double side;

    //constructor
    public Square(double squareSide, String squareName)
    {
        super(squareName);
        side = squareSide;
    }

    public double perimeter()
    { return 4 * side; }

    public double area()
    { return side * side; }
}
```

NOTE

1. It is meaningless to define `perimeter` and `area` methods for `Shape`—thus, these are declared as abstract methods.
2. An abstract class can have both instance variables and concrete (nonabstract) methods. See, for example, `name`, `getName`, and `semiPerimeter` in the `Shape` class.
3. Abstract methods are declared with the keyword `abstract`. There is no method body. The header is terminated with a semicolon.
4. A concrete (non-abstract) subclass of an abstract superclass must provide implementation code for all abstract methods of the superclass. Therefore both the `Circle` and `Square` classes implement both the `perimeter` and `area` methods.
5. It is possible for an abstract class to have no abstract methods. (An abstract subclass of an abstract superclass inherits the abstract methods without explicitly declaring them.)
6. An abstract class may or may not have constructors.
7. No instances can be created for an abstract class:

```
Shape a = new Shape("blob");  //Illegal.
                    //Can't create instance of abstract class.
Shape c = new Circle(1.5, "small circle");  //legal
```

8. Polymorphism works with abstract classes as it does with concrete classes:

```
Shape circ = new Circle(10, "circle");
Shape sq = new Square(9.4, "square");
Shape s = null;
System.out.println("Which shape?");
String str = IO.readString();        //read user input
if (str.equals("circle"))
    s = circ;
else
    s = sq;
System.out.println("Area of " + s.getName() + " is "
        + s.area());
```

INTERFACES

Interface

An *interface* is a collection of related methods, either abstract (headers only) or default (implementation provided in the interface). Default methods are new in Java 8, and will not be tested on the AP exam. Non-default (i.e., abstract) methods will be tested on the exam and are discussed below.

Students may be required to design, create, or modify classes that implement interfaces with abstract methods.

The non-default methods are both public and abstract—no need to explicitly include these keywords. As such, they provide a framework of behavior for any class.

The classes that implement a given interface may represent objects that are vastly different. They all, however, have in common a capability or feature expressed in the methods of the interface. An interface called `FlyingObject`, for example, may have the methods `fly` and `isFlying`. Some classes that implement `FlyingObject` could be `Bird`,

Airplane, Missile, Butterfly, and Witch. A class called Turtle would be unlikely to implement FlyingObject because turtles don't fly.

An interface called Computable may have just three methods: add, subtract, and multiply. Classes that implement Computable could be Fraction, Matrix, LongInteger, and ComplexNumber. It would not be meaningful, however, for a TelevisionSet to implement Computable—what does it mean, for example, to multiply two TelevisionSet objects?

A class that implements an interface can define any number of methods. In particular, it contracts to provide implementations for *all* the non-default (i.e., abstract) methods declared in the interface. If it fails to implement any of the methods, the class must be declared abstract.

A nonabstract class that implements an interface must implement every abstract method of the interface.

Defining an Interface

An interface is declared with the interface keyword. For example,

```
public interface FlyingObject
{
    void fly();          //method that simulates flight of object
    boolean isFlying();  //true if object is in flight,
                         //false otherwise
}
```

The implements Keyword

Interfaces are implemented using the implements keyword. For example,

```
public class Bird implements FlyingObject
{
    ...
```

This declaration means that two of the methods in the Bird class must be fly and isFlying. Note that any subclass of Bird will automatically implement the interface FlyingObject, since fly and isFlying will be inherited by the subclass.

A class that extends a superclass can also *directly* implement an interface. For example,

```
public class Mosquito extends Insect implements FlyingObject
{
    ...
```

NOTE

1. The extends clause must precede the implements clause.
2. A class can have just one superclass, but it can implement any number of interfaces:

```
public class SubClass extends SuperClass
        implements Interface1, Interface2, ...
```

The Comparable Interface

Starting in 2015, this will not be tested on the AP exam. Students will, however, be required to use compareTo for comparison of strings (p. 178).

The standard java.lang package contains the Comparable interface, which provides a useful method for comparing objects.

Optional topic

(continued)

```
public interface Comparable
{
    int compareTo(Object obj);
}
```

Classes written for objects that need to be compared should implement `Comparable`.

Any class that implements `Comparable` must provide a `compareTo` method. This method compares the implicit object (`this`) with the parameter object (`obj`) and returns a negative integer, zero, or a positive integer depending on whether the implicit object is less than, equal to, or greater than the parameter. If the two objects being compared are not type compatible, a `ClassCastException` is thrown by the method.

Example

The abstract `Shape` class defined previously (p. 143) is modified to implement the `Comparable` interface:

```
public abstract class Shape implements Comparable
{
    private String name;

    //constructor
    public Shape(String shapeName)
    { name = shapeName; }

    public String getName()
    { return name; }

    public abstract double area();
    public abstract double perimeter();

    public double semiPerimeter()
    { return perimeter() / 2; }

    public int compareTo(Object obj)
    {
        final double EPSILON = 1.0e-15;   //slightly bigger than
                                          //machine precision
        Shape rhs = (Shape) obj;
        double diff = area() - rhs.area();
        if (Math.abs(diff) <= EPSILON * Math.abs(area()))
            return 0;  //area of this shape equals area of obj
        else if (diff < 0)
            return -1; //area of this shape less than area of obj
        else
            return 1;  //area of this shape greater than area of obj
    }
}
```

NOTE

1. The `Circle`, `Square`, and other subclasses of `Shape` will all automatically implement `Comparable` and inherit the `compareTo` method.
2. It is tempting to use a simpler test for equality of areas, namely

```
if (diff == 0)
    return 0;
```

(continued)

But recall that real numbers can have round-off errors in their storage (Box p. 65). This means that the simple test may return false even though the two areas are essentially equal. A more robust test is implemented in the code given, namely to test if the relative error in `diff` is small enough to be considered zero.

3. The `Object` class is a universal superclass (see p. 174). This means that the `compareTo` method can take as a parameter any object reference that implements `Comparable`.
4. One of the first steps of a `compareTo` method must cast the `Object` argument to the class type, in this case `Shape`. If this is not done, the compiler won't find the `area` method—remember, an `Object` is not necessarily a `Shape`.
5. The algorithm one chooses in `compareTo` should in general be consistent with the `equals` method (see p. 176): Whenever `object1.equals(object2)` returns `true`, `object1.compareTo(object2)` returns `0`.

Here is a program that finds the larger of two `Comparable` objects.

```
public class FindMaxTest
{
    /** Return the larger of two objects a and b. */
    public static Comparable max(Comparable a, Comparable b)
    {
        if (a.compareTo(b) > 0) //if a > b ...
            return a;
        else
            return b;
    }

    /** Test max on two Shape objects. */
    public static void main(String[] args)
    {
        Shape s1 = new Circle(3.0, "circle");
        Shape s2 = new Square(4.5, "square");
        System.out.println("Area of " + s1.getName() + " is " +
                s1.area());
        System.out.println("Area of " + s2.getName() + " is " +
                s2.area());
        Shape s3 = (Shape) max(s1, s2);
        System.out.println("The larger shape is the " +
                s3.getName());
    }
}
```

Here is the output:

```
Area of circle is 28.27
Area of square is 20.25
The larger shape is the circle
```

NOTE

1. The `max` method takes parameters of type `Comparable`. Since `s1` *is-a* `Comparable` object and `s2` *is-a* `Comparable` object, no casting is necessary in the method call.
2. The `max` method can be called with any two `Comparable` objects, for example, two `String` objects or two `Integer` objects (see Chapter 4).

(continued)

3. The objects must be type compatible (i.e., it must make sense to compare them). For example, in the program shown, if `s1` *is-a* `Shape` and `s2` *is-a* `String`, the `compareTo` method will throw a `ClassCastException` at the line

```
Shape rhs = (Shape) obj;
```

4. The cast is needed in the line

```
Shape s3 = (Shape) max(s1, s2);
```

since `max(s1, s2)` returns a `Comparable`.

5. A primitive type is not an object and therefore cannot be passed as `Comparable`. You can, however, use a wrapper class and in this way convert a primitive type to a `Comparable` (see p. 180).

ABSTRACT CLASS VS. INTERFACE

Consider writing a program that simulates a game of Battleships. The program may have a `Ship` class with subclasses `Submarine`, `Cruiser`, `Destroyer`, and so on. The various ships will be placed in a two-dimensional grid that represents a part of the ocean.

An abstract class `Ship` is a good design choice. There will not be any instances of `Ship` objects because the specific features of the subclasses must be known in order to place these ships in the grid. A `Grid` interface that manipulates objects in a two-dimensional setting suggests itself for the two-dimensional grid.

Notice that the abstract `Ship` class is specific to the Battleships application, whereas the `Grid` interface is not. You could use the `Grid` interface in any program that has a two-dimensional grid.

Interface vs. Abstract Class

- Use an abstract class for an object that is application-specific but incomplete without its subclasses.
- Consider using an interface when its methods are suitable for your program but could be equally applicable in a variety of programs.
- An interface typically doesn't provide implementations for any of its methods, whereas an abstract class does. (In Java 8, implementation of default methods is allowed in interfaces.)
- An interface cannot contain instance variables, whereas an abstract class can.
- It is not possible to create an instance of an interface object or an abstract class object.

Chapter Summary

You should be able to write your own subclasses, given any superclass, and also design, create, or modify a class that implements an interface.

Be sure you understand the use of the keyword `super`, both in writing constructors and calling methods of the superclass.

You should understand what polymorphism is: Recall that it only operates when methods have been overridden in at least one subclass. You should also be able to explain the difference between the following concepts:

- An abstract class and an interface.
- An overloaded method and an overridden method.
- Dynamic binding (late binding) and static binding (early binding).