Inheritance and Polymorphism

4

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

- → **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.

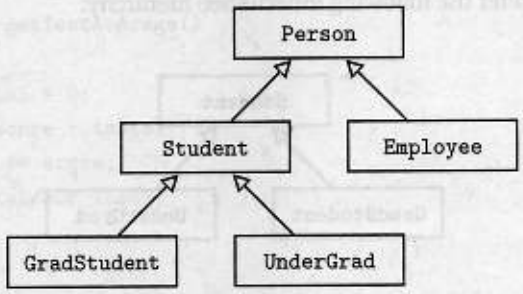
Don't get confused by the names: a subclass is bigger than a superclass—it contains more data and more methods!

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. 146). 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. 147).

Implementing Subclasses

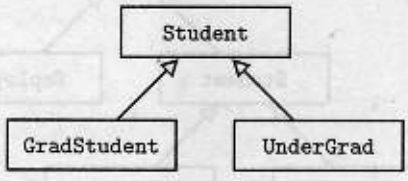
THE `extends` KEYWORD

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

A subclass inherits all the public and protected methods of its superclass. It does not, however, inherit the private instance variables or private methods of its parent class, and therefore does

not have direct access to them. To access private instance variables, a subclass must use the accessor or mutator methods that it has inherited.

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 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.

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

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
}
```

3. If the superclass has at least one constructor with parameters, the code in Note 2 above will cause a compile-time error if the superclass does not also contain a default (no parameter) constructor.

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. 110) 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 = ...;       //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();
   ```

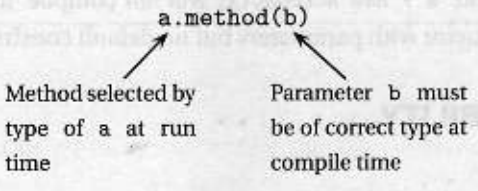
   compiles without problem because `g` is declared to be of type `GradStudent`, and this is the class that contains `getID`. No cast is required.
3. Class casts will not explicitly be tested on the AP exam. You should, however, understand why the following statement will cause a compile-time error:

   ```
   int x = s.getID(); //No getID method in Student class
   ```

   And the following statement will compile without error:

   ```
   int y = g.getID(); //getID method is in GradStudent class
   ```

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. It must pass the *is-a* test for the type in the method declaration. You may need to do an explicit cast to a subclass type to make this correct.

The `ClassCastException`

Optional topic

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
```

NOTE

The `ClassCastException` is not tested on the AP exam. But it's important to understand that you'll get an error if you try to cast an `Apple` to an `Orange`.

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 = ...;        //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 nonabstract class that implements an interface must implement every abstract method of the interface.

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.

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, ...
```

3. No instances can be created for an interface:

```
FlyingObject robin = new FlyingObject(); //Illegal
                   //Can't create instance of interface.
FlyingObject robin = new Bird(< Bird parameters >); //legal
```

The `Comparable` Interface

This is not tested on the AP exam. Students will, however, be required to use `compareTo` for comparison of strings (p. 192).

Optional topic

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

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

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

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

(continued)

The abstract `Shape` class defined previously (p. 155) 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;
   ```

   But recall that real numbers can have round-off errors in their storage (Box p. 75). 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. 188). This means that the `compareTo` method can take as a parameter any object reference that implements `Comparable`.

(continued)

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. 190): 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 5).
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. 194).

(continued)

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).

MULTIPLE-CHOICE QUESTIONS ON INHERITANCE AND POLYMORPHISM

Questions 1–9 refer to the `BankAccount`, `SavingsAccount`, and `CheckingAccount` classes defined below:

```
public class BankAccount
{
    private double balance;

    public BankAccount()
    { balance = 0; }

    public BankAccount(double acctBalance)
    { balance = acctBalance; }

    public void deposit(double amount)
    { balance += amount; }

    public void withdraw(double amount)
    { balance -= amount; }

    public double getBalance()
    { return balance; }
}

public class SavingsAccount extends BankAccount
{
    private double interestRate;

    public SavingsAccount()
    { /* implementation not shown */ }

    public SavingsAccount(double acctBalance, double rate)
    { /* implementation not shown */ }

    public void addInterest()    //Add interest to balance
    { /* implementation not shown */ }
}

public class CheckingAccount extends BankAccount
{
    private static final double FEE = 2.0;
    private static final double MIN_BALANCE = 50.0;

    public CheckingAccount(double acctBalance)
    { /* implementation not shown */ }

    /** FEE of $2 deducted if withdrawal leaves balance less
     *  than MIN_BALANCE. Allows for negative balance. */
    public void withdraw(double amount)
    { /* implementation not shown */ }
}
```

1. Of the methods shown, how many different nonconstructor methods can be invoked by a `SavingsAccount` object?

   (A) 1
   (B) 2
   (C) 3
   (D) 4
   (E) 5

2. Which of the following correctly implements the default constructor of the `SavingsAccount` class?

   I
   ```
   interestRate = 0;
   super();
   ```

   II
   ```
   super();
   interestRate = 0;
   ```

   III
   ```
   super();
   ```

   (A) II only
   (B) I and II only
   (C) II and III only
   (D) III only
   (E) I, II, and III

3. Which is a correct implementation of the constructor with parameters in the `SavingsAccount` class?

   (A)
   ```
   balance = acctBalance;
   interestRate = rate;
   ```

   (B)
   ```
   getBalance() =  acctBalance;
   interestRate = rate;
   ```

   (C)
   ```
   super();
   interestRate = rate;
   ```

   (D)
   ```
   super(acctBalance);
   interestRate = rate;
   ```

   (E)
   ```
   super(acctBalance, rate);
   ```

4. Which is a correct implementation of the `CheckingAccount` constructor?

   I
   ```
   super(acctBalance);
   ```

   II
   ```
   super();
   deposit(acctBalance);
   ```

   III
   ```
   deposit(acctBalance);
   ```

   (A) I only
   (B) II only
   (C) III only
   (D) II and III only
   (E) I, II, and III

5. Which is correct implementation code for the `withdraw` method in the `CheckingAccount` class?

(A)
```
super.withdraw(amount);
if (balance < MIN_BALANCE)
    super.withdraw(FEE);
```

(B)
```
withdraw(amount);
if (balance < MIN_BALANCE)
    withdraw(FEE);
```

(C)
```
super.withdraw(amount);
if (getBalance() < MIN_BALANCE)
    super.withdraw(FEE);
```

(D)
```
withdraw(amount);
if (getBalance() < MIN_BALANCE)
    withdraw(FEE);
```

(E)
```
balance -= amount;
if (balance < MIN_BALANCE)
    balance -= FEE;
```

6. Redefining the `withdraw` method in the `CheckingAccount` class is an example of
   (A) method overloading.
   (B) method overriding.
   (C) downcasting.
   (D) dynamic binding (late binding).
   (E) static binding (early binding).

Use the following for Questions 7 and 8.

A program to test the `BankAccount`, `SavingsAccount`, and `CheckingAccount` classes has these declarations:

```
BankAccount b = new BankAccount(1400);
BankAccount s = new SavingsAccount(1000, 0.04);
BankAccount c = new CheckingAccount(500);
```

7. Which method call will cause an error?
   (A) `b.deposit(200);`
   (B) `s.withdraw(500);`
   (C) `c.withdraw(500);`
   (D) `s.deposit(10000);`
   (E) `s.addInterest();`

8. In order to test polymorphism, which method must be used in the program?
   (A) Either a `SavingsAccount` constructor or a `CheckingAccount` constructor
   (B) `addInterest`
   (C) `deposit`
   (D) `withdraw`
   (E) `getBalance`

9. A new method is added to the BankAccount class.

```
/** Transfer amount from this BankAccount to another BankAccount.
 *  Precondition:  balance > amount
 *  @param another a different BankAccount object
 *  @param amount the amount to be transferred
 */
public void transfer(BankAccount another, double amount)
{
    withdraw(amount);
    another.deposit(amount);
}
```

A program has these declarations:

```
BankAccount b = new BankAccount(650);
SavingsAccount timsSavings = new SavingsAccount(1500, 0.03);
CheckingAccount daynasChecking = new CheckingAccount(2000);
```

Which of the following will transfer money from one account to another without error?

I `b.transfer(timsSavings, 50);`

II `timsSavings.transfer(daynasChecking, 30);`

III `daynasChecking.transfer(b, 55);`

(A) I only
(B) II only
(C) III only
(D) I, II, and III
(E) None

10. Consider these class declarations:

```
public class Person
{
   ...
}

public class Teacher extends Person
{
   ...
}
```

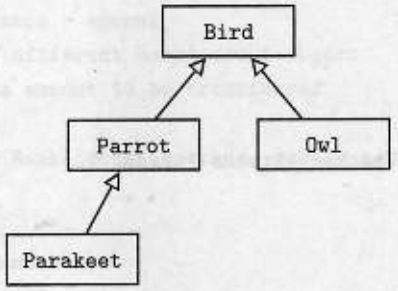
Which is a true statement?

I `Teacher` inherits the constructors of `Person`.
II `Teacher` can add new methods and private instance variables.
III `Teacher` can override existing private methods of `Person`.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

11. Which statement about abstract classes and interfaces is *false*?
(A) An interface cannot implement any non-default instance methods, whereas an abstract class can.
(B) A class can implement many interfaces but can have only one superclass.
(C) An unlimited number of unrelated classes can implement the same interface.
(D) It is not possible to construct either an abstract class object or an interface object.
(E) All of the methods in both an abstract class and an interface are public.

12. Consider the following hierarchy of classes:



A program is written to print data about various birds:

```
public class BirdStuff
{
    public static void printName(Bird b)
    { /* implementation not shown */ }

    public static void printBirdCall(Parrot p)
    { /* implementation not shown */ }

    //several more Bird methods

    public static void main(String[] args)
    {
        Bird bird1 = new Bird();
        Bird bird2 = new Parrot();
        Parrot parrot1 = new Parrot();
        Parrot parrot2 = new Parakeet();
        /* more code */
    }
}
```

Assuming that none of the given classes is abstract and all have default constructors, which of the following segments of `/* more code */` will cause an error?

(A) `printBirdCall(bird2);`

(B) `printName(parrot2);`

(C) `printName(bird2);`

(D) `printBirdCall(parrot2);`

(E) `printBirdCall(parrot1);`

Refer to the classes below for Questions 13 and 14.

```
public class ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* implementation of method1 */ }

    public void method2()
    { /* implementation of method2 */ }
}

public class ClassB extends ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* different implementation from method1 in ClassA*/ }

    public void method3()
    { /* implementation of method3 */ }
}
```

13. The `method1` method in `ClassB` is an example of
    (A) method overloading.
    (B) method overriding.
    (C) polymorphism.
    (D) information hiding.
    (E) procedural abstraction.

14. Consider the following declarations in a client class.

```
ClassA ob1 = new ClassA();
ClassA ob2 = new ClassB();
ClassB ob3 = new ClassB();
```

Which of the following method calls will cause an error?

I `ob1.method3();`

II `ob2.method3();`

III `ob3.method2();`

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

Use the declarations below for Questions 15–17.

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

   //constructor
   public Solid(String solidName)
   { name = solidName; }

   public String getName()
   { return name; }

   public abstract double volume();
}

public class Sphere extends Solid
{
   private double radius;

   //constructor
   public Sphere(String sphereName, double sphereRadius)
   {
      super(sphereName);
      radius = sphereRadius;
   }

   public double volume()
   { return (4.0/3.0) * Math.PI * radius * radius * radius; }
}

public class RectangularPrism extends Solid
{
   private double length;
   private double width;
   private double height;

   //constructor
   public RectangularPrism(String prismName, double l, double w,
         double h)
   {
      super(prismName);
      length = l;
      width = w;
      height = h;
   }

   public double volume()
   { return length * width * height; }
}
```

15. A program that tests these classes has the following declarations and assignments:

```
Solid s1, s2, s3, s4;
s1 = new Solid("blob");
s2 = new Sphere("sphere", 3.8);
s3 = new RectangularPrism("box", 2, 4, 6.5);
s4 = null;
```

How many of the above lines of code are incorrect?

(A) 0
(B) 1
(C) 2
(D) 3
(E) 4

16. Which is *false*?

(A) If a program has several objects declared as type `Solid`, the decision about which `volume` method to call will be resolved at run time.
(B) If the `Solid` class were modified to provide a default implementation for the `volume` method, it would no longer need to be an abstract class.
(C) If the `Sphere` and `RectangularPrism` classes failed to provide an implementation for the `volume` method, they would need to be declared as abstract classes.
(D) The fact that there is no reasonable default implementation for the `volume` method in the `Solid` class suggests that it should be an abstract method.
(E) Since `Solid` is abstract and its subclasses are nonabstract, polymorphism no longer applies when these classes are used in a program.

17. Here is a program that prints the volume of a solid:

```
public class SolidMain
{
    /** Output volume of Solid s. */
    public static void printVolume(Solid s)
    {
        System.out.println("Volume = " + s.volume() +
                " cubic units");
    }

    public static void main(String[] args)
    {
        Solid sol;
        Solid sph = new Sphere("sphere", 4);
        Solid rec = new RectangularPrism("box", 3, 6, 9);
        int flipCoin = (int) (Math.random() * 2);   //0 or 1
        if (flipCoin == 0)
            sol = sph;
        else
            sol = rec;
        printVolume(sol);
    }
}
```

Which is a true statement about this program?

(A) It will output the volume of the sphere or box, as intended.

(B) It will output the volume of the default `Solid s`, which is neither a sphere nor a box.

(C) A run-time error will occur because it is not specified whether `s` is a sphere or a box.

(D) A compile-time error will occur because there is no implementation code for `volume` in the `Solid` class.

(E) A run-time error will occur because of parameter type mismatch in the method call `printVolume(sol)`.

18. Consider the `Computable` interface below for performing simple calculator operations:

```
public interface Computable
{
    /** Return this Object + y. */
    Object add(Object y);

    /** Return this Object - y. */
    Object subtract(Object y);

    /** Return this Object * y. */
    Object multiply(Object y);
}
```

Which of the following is the *least* suitable class for implementing `Computable`?

```
(A) LargeInteger     //integers with 100 digits or more
(B) Fraction         //implemented with numerator and
                     //denominator of type int
(C) IrrationalNumber //nonrepeating, nonterminating decimal
(D) Length           //implemented with different units, such
                     //as inches, centimeters, etc.
(E) BankAccount      //implemented with balance
```

Refer to the `Player` interface shown below for Questions 19–22.

```
public interface Player
{
    /** Return an integer that represents a move in a game. */
    int getMove();

    /** Display the status of the game for this Player after
     *  implementing the next move. */
    void updateDisplay();
}
```

19. `HumanPlayer` is a class that implements the `Player` interface. Another class, `SmartPlayer`, is a subclass of `HumanPlayer`. Which statement is *false*?
    (A) `SmartPlayer` automatically implements the `Player` interface.
    (B) `HumanPlayer` must contain implementations of both the `updateDisplay` and `getMove` methods, or be declared as abstract.
    (C) It is not possible to declare a reference of type `Player`.
    (D) The `SmartPlayer` class can override the methods `updateDisplay` and `getMove` of the `HumanPlayer` class.
    (E) A method in a client program can have `Player` as a parameter type.

20. A programmer plans to write programs that simulate various games. In each case he will have several classes, each representing a different kind of competitor in the game, such as `ExpertPlayer`, `ComputerPlayer`, `RecklessPlayer`, `CheatingPlayer`, `Beginner`, `IntermediatePlayer`, and so on. It may or may not be suitable for these classes to implement the `Player` interface, depending on the particular game being simulated. In the games described below, which is the *least* suitable for having the competitor classes implement the given `Player` interface?

    (A) High-Low Guessing Game: The computer thinks of a number and the competitor who guesses it with the least number of guesses wins. After each guess, the computer tells whether its number is higher or lower than the guess.

    (B) Chips: Start with a pile of chips. Each player in turn removes some number of chips. The winner is the one who removes the final chip. The first player may remove any number of chips, but not all of them. Each subsequent player must remove at least one chip and at most twice the number removed by the preceding player.

    (C) Chess: Played on a square board of 64 squares of alternating colors. There are just two players, called White and Black, the colors of their respective pieces. The players each have a set of pieces on the board that can move according to a set of rules. The players alternate moves, where a move consists of moving any one piece to another square. If that square is occupied by an opponent's piece, the piece is captured and removed from the board.

    (D) Tic-Tac-Toe: Two players alternate placing "X" or "O" on a 3×3 grid. The first player to get three in a row, where a row can be vertical, horizontal, or diagonal, wins.

    (E) Battleships: There are two players, each with a 10×10 grid hidden from his opponent. Various "ships" are placed on the grid. A move consists of calling out a grid location, trying to "hit" an opponent's ship. Players alternate moves. The first player to sink his opponent's fleet wins.

Consider these declarations for Questions 21 and 22:

```
public class HumanPlayer implements Player
{
    private String name;

    //Constructors not shown ...

    //Code to implement getMove and updateDisplay not shown ...

    public String getName()
    { /* implementation not shown */ }
}

public class ExpertPlayer extends HumanPlayer
{
    private int rating;

    //Constructors not shown ...

    public int compareTo(ExpertPlayer expert)
    { /* implementation not shown */ }
}
```

21. Which code segment in a client program will cause an error?

I
```
Player p1 = new HumanPlayer();
Player p2 = new ExpertPlayer();
int x1 = p1.getMove();
int x2 = p2.getMove();
```

II
```
int x;
Player c1 = new ExpertPlayer(/* correct parameter list */);
Player c2 = new ExpertPlayer(/* correct parameter list */);
if (c1.compareTo(c2) < 0)
    x = c1.getMove();
else
    x = c2.getMove();
```

III
```
int x;
HumanPlayer h1 = new HumanPlayer(/* correct parameter list */);
HumanPlayer h2 = new HumanPlayer(/* correct parameter list */);
if (h1.compareTo(h2) < 0)
    x = h1.getMove();
else
    x = h2.getMove();
```

(A) II only
(B) III only
(C) II and III only
(D) I, II, and III
(E) None

22. Which of the following is correct implementation code for the `compareTo` method in the `ExpertPlayer` class?

I
```
if (rating == expert.rating)
    return 0;
else if (rating < expert.rating)
    return -1;
else
    return 1;
```

II
```
return rating - expert.rating;
```

III
```
if (getName().equals(expert.getName()))
    return 0;
else if (getName().compareTo(expert.getName()) < 0)
    return -1;
else
    return 1;
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

23. Consider the following class definitions:

```
public class Animal
{
    private String type;

    public Animal(String theType)
    {
        type = theType;
    }

    public String getType()
    {
        return type;
    }
}

public class Dog extends Animal
{
    public Dog(String theType)
    {
        super(theType);
    }
}
```

Refer to the following declarations and method in a client program:

```
Animal d1 = new Animal("poodle");
Animal d2 = new Dog("shnauzer");
Dog d3 = new Dog("yorkie");

public static void display(Animal a)
{
    System.out.println("This dog is a " + a.getType();)
}
```

Which of the following method calls will compile without error?

I `display(d1);`

II `display(d2);`

III `display(d3);`

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

24. Which of the following classes is the least suitable candidate for containing a `compareTo` method?

(A)
```
public class Point
{
    private double x;
    private double y;

    //various methods follow
        ...
}
```

(B)
```
public class Name
{
    private String firstName;
    private String lastName;

    //various methods follow
        ...
}
```

(C)
```
public class Car
{
    private int modelNumber;
    private int year;
    private double price;

    //various methods follow
        ...
}
```

(D)
```
public class Student
{
    private String name;
    private double gpa;

    //various methods follow
        ...
}
```

(E)
```
public class Employee
{
    private String name;
    private int hireDate;
    private double salary;

    //various methods follow
        ...
}
```

25. A programmer has the task of maintaining a database of students of a large university. There are two types of students, undergraduates and graduate students. About a third of the graduate students are doctoral candidates.

All of the students have the same personal information stored, like name, address, and phone number, and also student information like courses taken and grades. Each student's GPA is computed, but differently for undergraduates and graduates. The doctoral candidates have information about their dissertations and faculty advisors.

The programmer will write a Java program to handle all the student information. Which of the following is the best design, in terms of programmer efficiency and code reusability? Note: `{ ... }` denotes class code.

(A)
```
public interface Student { ...}
public class Undergraduate implements Student { ... }
public class Graduate implements Student { ... }
public class DocStudent extends Graduate { ... }
```

(B)
```
public abstract class Student { ...}
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }
public class DocStudent extends Graduate { ... }
```

(C)
```
public class Student { ...}
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }
public class DocStudent extends Graduate { ... }
```

(D)
```
public abstract class Student { ...}
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }
public class DocStudent extends Student { ... }
```

(E)
```
public interface PersonalInformation { ... }
public class Student implements PersonalInformation { ...}
public class Undergraduate extends Student { ... }
public abstract class Graduate extends Student { ... }
public class DocStudent extends Graduate { ... }
```

26. Consider the `Orderable` interface and the partial implementation of the `Temperature` class defined below:

```
public interface Orderable
{
    /** Returns -1, 0, or 1 depending on whether the implicit
     *  object is less than, equal to, or greater than other.
     */
    int compareTo (Object other);
}

public class Temperature implements Orderable
{
    private String scale;
    private double degrees;

    //default constructor
    public Temperature ()
    { /* implementation not shown */ }

    //constructor
    public Temperature(String tempScale, double tempDegrees)
    { /* implementation not shown */ }

    public int compareTo(Object obj)
    { /* implementation not shown */ }

    public String toString()
    { /* implementation not shown */ }

    //Other methods are not shown.
}
```

Here is a program that finds the lowest of three temperatures:

```
public class TemperatureMain
{
    /** Find smaller of objects a and b. */
    public static Orderable min(Orderable a, Orderable b)
    {
        if (a.compareTo(b) < 0)
            return a;
        else
            return b;
    }

    /** Find smallest of objects a, b, and c. */
    public static Orderable minThree(Orderable a,
            Orderable b, Orderable c)
    {
        return min(min(a, b), c);
    }

    public static void main(String[] args)
    {
        /* code to test minThree method */
    }
}
```

Which are correct replacements for `/* code to test minThree method */`?

I
```
Temperature t1 = new Temperature("C", 85);
Temperature t2 = new Temperature("F", 45);
Temperature t3 = new Temperature("F", 120);
System.out.println("The lowest temperature is " +
        minThree(t1, t2, t3));
```

II
```
Orderable c1 = new Temperature("C", 85);
Orderable c2 = new Temperature("F", 45);
Orderable c3 = new Temperature("F", 120);
System.out.println("The lowest temperature is " +
        minThree(c1, c2, c3));
```

III
```
Orderable c1 = new Orderable("C", 85);
Orderable c2 = new Orderable("F", 45);
Orderable c3 = new Orderable("F", 120);
System.out.println("The lowest temperature is " +
        minThree(c1, c2, c3));
```

(A) II only
(B) I and II only
(C) II and III only
(D) I and III only
(E) I, II, and III

27. A certain interface provided by a Java package contains just a single method:

```
public interface SomeName
{
    int method1(Object o);
}
```

A programmer adds some functionality to this interface by adding another abstract method to it, `method2`:

```
public interface SomeName
{
    int method1(Object ob1);
    void method2(Object ob2);
}
```

As a result of this addition, which of the following is true?

(A) A compile-time will occur if `ob1` and `ob2` are not compatible.

(B) All classes that implement the original `SomeName` interface will need to be rewritten because they no longer implement `SomeName`.

(C) A class that implements the original `SomeName` interface will need to modify its declaration as follows:

```
public class ClassName implements SomeName extends method2
{    ...
```

(D) `SomeName` will need to be changed to an abstract class and provide implementation code for `method2`, so that the original and upgraded versions of `SomeName` are compatible.

(E) Any new class that implements the upgraded version of `SomeName` will not compile.

ANSWER KEY

1. **D**
2. **C**
3. **D**
4. **E**
5. **C**
6. **B**
7. **E**
8. **D**
9. **D**
10. **B**
11. **E**
12. **A**
13. **B**
14. **D**
15. **B**
16. **E**
17. **A**
18. **E**
19. **C**
20. **C**
21. **C**
22. **E**
23. **E**
24. **A**
25. **B**
26. **B**
27. **B**

ANSWERS EXPLAINED

1. **(D)** The methods are `deposit`, `withdraw`, and `getBalance`, all inherited from the `BankAccount` class, plus `addInterest`, which was defined just for the class `SavingsAccount`.

2. **(C)** Implementation I fails because `super()` *must* be the first line of the implementation whenever it is used in a constructor. Implementation III may appear to be incorrect because it doesn't initialize `interestRate`. Since `interestRate`, however, is a primitive type—`double`—the compiler will provide a default initialization of `0`, which was required.

3. **(D)** First, the statement `super(acctBalance)` initializes the inherited private variable `balance` as for the `BankAccount` superclass. Then the statement `interestRate = rate` initializes `interestRate`, which belongs uniquely to the `SavingsAccount` class. Choice E fails because `interestRate` does not belong to the `BankAccount` class and therefore cannot be initialized by a `super` method. Choice A is wrong because the `SavingsAccount` class cannot directly access the private instance variables of its superclass. Choice B assigns a value to an accessor method, which is meaningless. Choice C is incorrect because `super()` invokes the *default* constructor of the superclass. This will cause `balance` of the `SavingsAccount` object to be initialized to `0`, rather than `acctBalance`, the parameter value.

4. **(E)** The constructor must initialize the inherited instance variable `balance` to the value of the `acctBalance` parameter. All three segments achieve this. Implementation I does it by invoking `super(acctBalance)`, the constructor in the superclass. Implementation II first initializes `balance` to `0` by invoking the *default* constructor of the superclass. Then it calls the inherited `deposit` method of the superclass to add `acctBalance` to the account. Implementation III works because `super()` is automatically called as the first line of the constructor code if there is no explicit call to `super`.

5. **(C)** First the `withdraw` method of the `BankAccount` superclass is used to withdraw `amount`. A prefix of `super` must be used to invoke this method, which eliminates choices B and D. Then the balance must be tested using the accessor method `getBalance`, which is inherited. You can't test `balance` directly since it is private to the `BankAccount` class. This eliminates choices A and E, and provides another reason for eliminating choice B.

6. **(B)** When a superclass method is redefined in a subclass, the process is called *method overriding*. Which method to call is determined at run time. This is called *dynamic binding* (p. 150). *Method overloading* is two or more methods with different signatures in the same class (p. 110). The compiler recognizes at compile time which method to call. This is *early binding*. The process of *downcasting* is unrelated to these principles (p. 153).

7. **(E)** The `addInterest` method is defined only in the `SavingsAccount` class. It therefore cannot be invoked by a `BankAccount` object. The error can be fixed by casting `s` to the correct type:

   ```
   ((SavingsAccount) s).addInterest();
   ```

   The other method calls do not cause a problem because `withdraw` and `deposit` are both methods of the `BankAccount` class.

8. **(D)** The `withdraw` method is the only method that has one implementation in the superclass and a *different* implementation in a subclass. Polymorphism is the mechanism of selecting the correct method from the different possibilities in the class hierarchy. Notice that the `deposit` method, for example, is available to objects of all three bank account classes, but it's the *same* code in all three cases. So polymorphism isn't tested.

9. **(D)** It is OK to use `timsSavings` and `daynasChecking` as parameters since each of these *is-a* `BankAccount` object. It is also OK for `timsSavings` and `daynasChecking` to call the `transfer` method (statements II and III), since they inherit this method from the `BankAccount` superclass.

10. **(B)** Statement I is false: A subclass must specify its own constructors. Otherwise the default constructor of the superclass will automatically be invoked. Note that statement III is false: Private instance methods cannot be overridden.

11. **(E)** All of the methods in an interface are by default public (the `public` keyword isn't needed). An abstract class can have both private and public methods. Note that choice A would be false if it simply stated "An interface cannot implement any methods, whereas an abstract class can." Java 8 allows an interface to implement default methods.

12. **(A)** There is a quick test you can do to find the answer to this question: Test the *is-a* relationship—namely, the parameter for `printName` *is-a* `Bird`? and the parameter for `printBirdCall` *is-a* `Parrot`? Note that to get the type of the actual parameter, you must look at its left-hand-side declaration. Choice A fails the test: `bird2` *is-a* `Parrot`? The variable `bird2` is declared a `Bird`, which is not necessarily a `Parrot`. Each other choice passes the test: Choice B: `parrot2` *is-a* `Bird`. Choice C: `bird2` *is-a* `Bird`. Choice D: `parrot2` *is-a* `Parrot`. Choice E: `parrot1` *is-a* `Parrot`.

13. **(B)** Method overriding occurs whenever a method in a superclass is redefined in a subclass. Method overloading is a method in the same class that has the same name but different parameter types. Polymorphism is when the correct overridden method is called for a particular subclass object during run time. Information hiding is the use of `private` to restrict access. Procedural abstraction is the use of helper methods.

14. **(D)** Both method calls I and II will cause errors:
    I: An object of a superclass does not have access to a new method of its subclass.
    II: `ob2` is declared to be of type `ClassA`, so a compile-time error will occur with a message indicating that there is no `method2` in `ClassA`. Casting `ob2` to `ClassB` would correct the problem. (Note: Class casting is no longer included in the AP Java subset.)

Method call III is correct because a subclass inherits all the public methods of its superclass.

15. **(B)** The only incorrect line is `s1 = new Solid("blob")`: You can't create an instance of an abstract class. Abstract class references can, however, refer to objects of concrete (nonabstract) subclasses. Thus, the assignments for `s2` and `s3` are OK. Note that an abstract class reference can also be null, so the final assignment, though redundant, is correct.

16. **(E)** The point of having an abstract method is to postpone until run time the decision about which subclass version to call. This is what polymorphism is—calling the appropriate method at run time based on the type of the object.

17. **(A)** This is an example of polymorphism: The correct `volume` method is selected at run time. The parameter expected for `printVolume` is a `Solid` reference, which is what it gets in `main()`. The reference `sol` will refer either to a `Sphere` or a `RectangularPrism` object depending on the outcome of the coin flip. Since a `Sphere` is a `Solid` and a `RectangularPrism` is a `Solid`, there will be no type mismatch when these are the actual parameters in the `printVolume` method. (Note: The `Math.random` method is discussed in Chapter 5.)

18. **(E)** Each of choices A through D represents `Computable` objects: It makes sense to add, subtract, or multiply two large integers, two fractions, two irrational numbers, and two lengths. (One can multiply lengths to get an area, for example.) While it may make sense under certain circumstances to add or subtract two bank accounts, it does not make sense to multiply them!

19. **(C)** You can *declare a reference* of type `Player`. What you cannot do is *construct an object* of type `Player`. The following declarations are therefore legal:

```
SmartPlayer s = new SmartPlayer();
Player p1 = s;
Player p2 = new HumanPlayer();
```

20. **(C)** Remember, to implement the `Player` interface a class must provide implementations for `getMove` and `updateDisplay`. The `updateDisplay` method is suitable for all five games described. The `getMove` method returns a single integer, which works well for the High-Low game of choice A and the Chips game of choice B. In Tic-Tac-Toe (choice D) and Battleships (choice E) a move consists of giving a grid location. This can be provided by a single integer if the grid locations are numbered in a unique way. It's not ideal, but certainly doable. In the Chess game, however, it's neither easy nor intuitive to describe a move with a single integer. The player needs to specify both the grid location he is moving the piece to *and* which piece he is moving. The `getMove` method would need to be altered in a way that changes its return type. This makes the `Player` interface unsuitable.

21. **(C)** Segments II and III have errors in the `compareTo` calls. References `c1` and `c2` are of type `Player`, which doesn't have a `compareTo` method, and references `h1` and `h2` are of type `HumanPlayer`, which also doesn't have a `compareTo` method. Note that Segment II can be fixed by downcasting `c1` and `c2` to `ExpertPlayer`:

```
if (((ExpertPlayer) c1).compareTo((ExpertPlayer) c2) < 0)
```

A cast won't work in Segment III, because you can't cast a `HumanPlayer` to an `ExpertPlayer`. In Segments I, II, and III, the `getMove` calls are all correct, because `p1`,

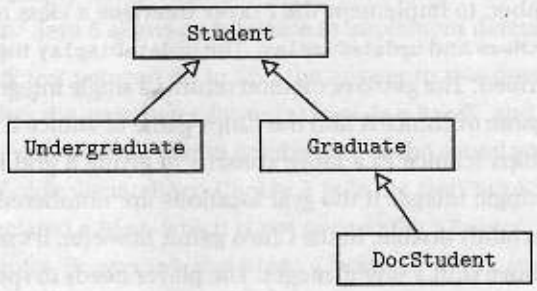
p2, c1, and c2 are all of type Player which has a getMove method; and h1 and h2 are of type HumanPlayer which implements Player and therefore has a getMove method.

22. **(E)** All implementations are correct. This is *not* a question about whether it is better to compare ExpertPlayers based on their ratings or their names! One might need an alphabetized list of players, or one might need a list according to ranking. In practice, the program specification will instruct the programmer which to use. Note that segment II is correct because compareTo doesn't need to return 1 or −1. Any positive or negative integer is OK. Note also that in segments I and II it is OK to use expert.rating, since expert is of type ExpertPlayer, the current class being written. Normally, a parameter of some class type cannot access the private instance variables of another class.

23. **(E)** All compile without error. For the method call display(arg), the compiler checks that the parameter arg *is-a* Animal, the type in the method's signature. Each of the objects d1, d2, and d3 passes the *is-a* test.

24. **(A)** While it is certainly possible to write a compareTo method for a Point class, there's no good intuitive way to compare points. Two points (x_1, y_1) and (x_2, y_2) are equal if and only if $x_1 = x_2$ and $y_1 = y_2$. But if points P_1 and P_2 are not equal, what will determine if $P_1 < P_2$ or $P_1 > P_2$? You could try using the distance from the origin. Define $P_1 > P_2$ if and only if $OP_1 > OP_2$, and $P_1 < P_2$ if and only if $OP_1 < OP_2$, where O is $(0,0)$. This definition means that points (a, b) and (b, a) are equal, which violates the definition of equals! The problem is that there is no way to map the two-dimensional set of points to a one-dimensional distance function and still be consistent with the definition of equals. The objects in each of the other classes can be compared without a problem. In choice B, two Name objects can be ordered alphabetically. In choice C, two Car objects can be ordered by year or by price. In choice D, two Student objects can be ordered by name or GPA. In choice E, two Employee objects can be ordered by name or seniority (date of hire).

25. **(B)** Here is the hierarchy of classes:



Eliminate choice D which fails to make DocStudent a subclass of Graduate. This is a poor design choice since a DocStudent *is-a* Graduate. Making Student an abstract class is desirable since the methods that are common to all students can go in there with implementations provided. The method to calculate the GPA, which differs among student types, will be declared in Student as an abstract method. Then unique implementations will be provided in both the Graduate and Undergraduate classes. Choice A is a poor design because making Student an interface means that all of its methods will need to be implemented in both the Undergraduate and Graduate classes. Many of these methods will have the same implementations. As far as possible, you want to arrange for classes to inherit common methods and to avoid repeated code. Choice C is slightly inferior

to choice B because you are told that all students are either graduates or undergraduates. Having the `Student` class abstract guarantees that you won't create an instance of a `Student` (who is neither a graduate nor an undergraduate). Choice E has a major design flaw: making `Graduate` an abstract class means that you can't create any instances of `Graduate` objects. Disaster! If the keyword `abstract` is removed from choice E, it becomes a fine design, as good as that in choice B. Once `Student` has implemented all the common `PersonalInformation` methods, these are inherited by each of the subclasses.

26. **(B)** Segment III is wrong because you can't construct an interface object. Segments I and II both work because the `minThree` method is expecting three parameters, each of which is an `Orderable`. Since `Temperature` implements `Orderable`, each of the `Temperature` objects is an `Orderable` and can be used as a parameter in this method. Note that the program assumes that the `compareTo` method is able to compare `Temperature` objects with different scales. This is an internal detail that would be dealt with in the `compareTo` method, and hidden from the client. When a class implements `Orderable` there is an assumption that the `compareTo` method will be implemented in a reasonable way.

27. **(B)** Classes that implement an interface must provide implementation code for all nondefault (i.e., abstract) methods in the interface. Adding `method2` to the `SomeName` interface means that all of those classes need to be rewritten with implementation code for `method2`. (This is not good—it violates the principle of code reusability, and programmers relying on the interface will not be happy when old code fails.) Choices A, C, and D are all meaningless garbage. Choice E *may* be true if there is some other error in the new class. Otherwise, as long as the new class provides implementation code for both `method1` and `method2`, the class will compile.