## 2.3 Interfaces and Abstract Classes

In order for two objects to interact, they must "know" about the various messages that each will accept, that is, the methods each object supports. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the ***application programming interface*** (API), or simply ***interface***, that their objects present to other objects. In the ***ADT-based*** approach (see Section 2.1.2) to data structures followed in this book, an interface defining an ADT is specified as a type definition and a collection of methods for this type, with the arguments for each method being of specified types. This specification is, in turn, enforced by the compiler or runtime system, which requires that the types of parameters that are actually passed to methods rigidly conform with the type specified in the interface. This requirement is known as ***strong typing***. Having to define interfaces and then having those definitions enforced by strong typing admittedly places a burden on the programmer, but this burden is offset by the rewards it provides, for it enforces the encapsulation principle and often catches programming errors that would otherwise go unnoticed.

### 2.3.1 Interfaces in Java

The main structural element in Java that enforces an API is an ***interface***. An interface is a collection of method declarations with no data and no bodies. That is, the methods of an interface are always empty; they are simply method signatures. Interfaces do not have constructors and they cannot be directly instantiated.

When a class implements an interface, it must implement all of the methods declared in the interface. In this way, interfaces enforce requirements that an implementing class has methods with certain specified signatures.

Suppose, for example, that we want to create an inventory of antiques we own, categorized as objects of various types and with various properties. We might, for instance, wish to identify some of our objects as sellable, in which case they could implement the Sellable interface shown in Code Fragment 2.8.

We can then define a concrete class, Photograph, shown in Code Fragment 2.9, that implements the Sellable interface, indicating that we would be willing to sell any of our Photograph objects. This class defines an object that implements each of the methods of the Sellable interface, as required. In addition, it adds a method, isColor, which is specialized for Photograph objects.

Another kind of object in our collection might be something we could transport. For such objects, we define the interface shown in Code Fragment 2.10.

```
1  /** Interface for objects that can be sold. */
2  public interface Sellable {
3
4    /** Returns a description of the object. */
5    public String description( );
6
7    /** Returns the list price in cents. */
8    public int listPrice( );
9
10   /** Returns the lowest price in cents we will accept. */
11   public int lowestPrice( );
12 }
```

**Code Fragment 2.8:** Interface Sellable.

```
1  /** Class for photographs that can be sold. */
2  public class Photograph implements Sellable {
3    private String descript;                          // description of this photo
4    private int price;                                // the price we are setting
5    private boolean color;                            // true if photo is in color
6
7    public Photograph(String desc, int p, boolean c) {    // constructor
8      descript = desc;
9      price = p;
10     color = c;
11   }
12
13   public String description( ) { return descript; }
14   public int listPrice( ) { return price; }
15   public int lowestPrice( ) { return price/2; }
16   public boolean isColor( ) { return color; }
17 }
```

**Code Fragment 2.9:** Class Photograph implementing the Sellable interface.

```
1  /** Interface for objects that can be transported. */
2  public interface Transportable {
3    /** Returns the weight in grams. */
4    public int weight( );
5    /** Returns whether the object is hazardous. */
6    public boolean isHazardous( );
7  }
```

**Code Fragment 2.10:** Interface Transportable.

We could then define the class BoxedItem, shown in Code Fragment 2.11, for miscellaneous antiques that we can sell, pack, and ship. Thus, the class BoxedItem implements the methods of the Sellable interface and the Transportable interface, while also adding specialized methods to set an insured value for a boxed shipment and to set the dimensions of a box for shipment.

```
1  /** Class for objects that can be sold, packed, and shipped. */
2  public class BoxedItem implements Sellable, Transportable {
3    private String descript;          // description of this item
4    private int price;                // list price in cents
5    private int weight;               // weight in grams
6    private boolean haz;              // true if object is hazardous
7    private int height=0;             // box height in centimeters
8    private int width=0;              // box width in centimeters
9    private int depth=0;              // box depth in centimeters
10   /** Constructor */
11   public BoxedItem(String desc, int p, int w, boolean h) {
12     descript = desc;
13     price = p;
14     weight = w;
15     haz = h;
16   }
17   public String description( ) { return descript; }
18   public int listPrice( ) { return price; }
19   public int lowestPrice( ) { return price/2;      }
20   public int weight( ) { return weight; }
21   public boolean isHazardous( ) { return haz; }
22   public int insuredValue( ) { return price*2; }
23   public void setBox(int h, int w, int d) {
24     height = h;
25     width = w;
26     depth = d;
27   }
28 }
```

**Code Fragment 2.11:** Class BoxedItem.

The class BoxedItem shows another feature of classes and interfaces in Java, as well—that a class can implement multiple interfaces (even though it may only extend one other class). This allows us a great deal of flexibility when defining classes that should conform to multiple APIs.

### 2.3.2 Multiple Inheritance for Interfaces

The ability of extending from more than one type is known as ***multiple inheritance***. In Java, multiple inheritance is allowed for interfaces but not for classes. The reason for this rule is that interfaces do not define fields or method bodies, yet classes typically do. Thus, if Java were to allow multiple inheritance for classes, there could be a confusion if a class tried to extend from two classes that contained fields with the same name or methods with the same signatures. Since there is no such confusion for interfaces, and there are times when multiple inheritance of interfaces is useful, Java allows interfaces to use multiple inheritance.

One use for multiple inheritance of interfaces is to approximate a multiple inheritance technique called the ***mixin***. Unlike Java, some object-oriented languages, such as Smalltalk and C++, allow multiple inheritance of concrete classes, not just interfaces. In such languages, it is common to define classes, called ***mixin*** classes, that are never intended to be created as stand-alone objects, but are instead meant to provide additional functionality to existing classes. Such inheritance is not allowed in Java, however, so programmers must approximate it with interfaces. In particular, we can use multiple inheritance of interfaces as a mechanism for "mixing" the methods from two or more unrelated interfaces to define an interface that combines their functionality, possibly adding more methods of its own. Returning to our example of the antique objects, we could define an interface for insurable items as follows:

```
public interface Insurable extends Sellable, Transportable {
  /** Returns insured value in cents */
  public int insuredValue( );
}
```

This interface combines the methods of the Transportable interface with the methods of the Sellable interface, and adds an extra method, insuredValue. Such an interface could allow us to define the BoxedItem alternately as follows:

```
public class BoxedItem2 implements Insurable {

  // ... same code as class BoxedItem
}
```

In this case, note that the method insuredValue is not optional, whereas it was optional in the declaration of BoxedItem given previously.

Java interfaces that approximate the mixin include java.lang.Cloneable, which adds a copy feature to a class; java.lang.Comparable, which adds a comparability feature to a class (imposing a natural order on its instances); and java.util.Observer, which adds an update feature to a class that wishes to be notified when certain "observable" objects change state.

### 2.3.3 Abstract Classes

In Java, an ***abstract class*** serves a role somewhat between that of a traditional class and that of an interface. Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies; such methods are known as ***abstract methods***. However, unlike an interface, an abstract class may define one or more fields and any number of methods with implementation (so-called ***concrete methods***). An abstract class may also extend another class and be extended by further subclasses.

As is the case with interfaces, an abstract class may not be instantiated, that is, no object can be created directly from an abstract class. In a sense, it remains an incomplete class. A subclass of an abstract class must provide an implementation for the abstract methods of its superclass, or else remain abstract. To distinguish from abstract classes, we will refer to nonabstract classes as ***concrete classes***.

In comparing the use of interfaces and abstract classes, it is clear that abstract classes are more powerful, as they can provide some concrete functionality. However, the use of abstract classes in Java is limited to ***single inheritance***, so a class may have at most one superclass, whether concrete or abstract (see Section 2.3.2).

We will take great advantage of abstract classes in our study of data structures, as they support greater reusability of code (one of our object-oriented design goals from Section 2.1.1). The commonality between a family of classes can be placed within an abstract class, which serves as a superclass to multiple concrete classes. In this way, the concrete subclasses need only implement the additional functionality that differentiates themselves from each other.

As a tangible example, we reconsider the progression hierarchy introduced in Section 2.2.3. Although we did not formally declare the Progression base class as abstract in that presentation, it would have been a reasonable design to have done so. We did not intend for users to directly create instances of the Progression class; in fact, the sequence that it produces is simply a special case of an arithmetic progression with increment one. The primary purpose of the Progression class is to provide common functionality to all three subclasses: the declaration and initialization of the current field, and the concrete implementations of the nextValue and printProgression methods.

The most important aspect in specializing that class was in overriding the protected advance method. Although we gave a simple implementation of that method within the Progression class to increment the current value, none of our three subclasses rely on that behavior. On the next page, we demonstrate the mechanics of abstract classes in Java by redesigning the progression base class into an AbstractProgression base class. In that design, we leave the advance method as truly abstract, leaving the burden of an implementation to the various subclasses.

### Mechanics of Abstract Classes in Java

In Code Fragment 2.12, we give a Java implementation of a new abstract base class for our progression hierarchy. We name the new class AbstractProgression rather than Progression, only to differentiate it in our discussion. The definitions are almost identical; there are only two key differences that we highlight. The first is the use of the **abstract** modifier on line 1, when declaring the class. (See Section 1.2.2 for a discussion of class modifiers.)

As with our original class, the new class declares the current field and provides constructors that initialize it. Although our abstract class cannot be instantiated, the constructors can be invoked within the subclass constructors using the **super** keyword. (We do just that, within all three of our progression subclasses.)

The new class has the same concrete implementations of methods nextValue and printProgression as did our original. However, we explicitly define the advance method with the **abstract** modifier at line 19, and without any method body.

Even though we have not implemented the advance method as part of the AbstractProgression class, it is legal to call it from within the body of nextValue. This is an example of an object-oriented design pattern known as the ***template method pattern***, in which an abstract base class provides a concrete behavior that relies upon calls to other abstract behaviors. Once a subclass provides definitions for the missing abstract behaviors, the inherited concrete behavior is well defined.

```
1  public abstract class AbstractProgression {
2    protected long current;
3    public AbstractProgression( ) { this(0); }
4    public AbstractProgression(long start) { current = start; }
5
6    public long nextValue( ) {                    // this is a concrete method
7      long answer = current;
8      advance( );     // this protected call is responsible for advancing the current value
9      return answer;
10   }
11
12   public void printProgression(int n) {         // this is a concrete method
13     System.out.print(nextValue( ));             // print first value without leading space
14     for (int j=1; j < n; j++)
15       System.out.print(" " + nextValue( ));     // print leading space before others
16     System.out.println( );                      // end the line
17   }
18
19   protected abstract void advance( );           // notice the lack of a method body
20 }
```

**Code Fragment 2.12:** An abstract version of the progression base class, originally given in Code Fragment 2.2. (We omit documentation for brevity.)