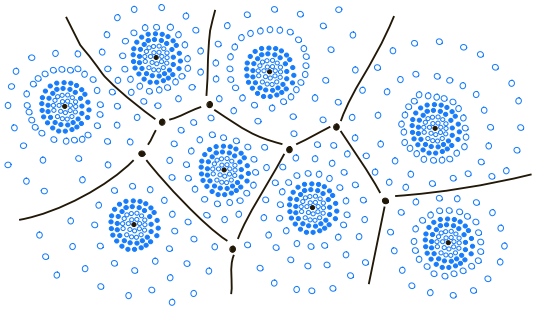
Chapter 2 Object-Oriented Design



Contents

2.1 Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented paradigm are called ***objects***. Each object is an ***instance*** of a ***class***. Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects. The class definition typically specifies the ***data fields***, also known as ***instance variables***, that an object contains, as well as the ***methods*** (operations) that an object can execute. This view of computing fulfill several goals and incorporates design principles, which we will discuss in this chapter.

2.1.1 Object-Oriented Design Goals

Software implementations should achieve ***robustness***, ***adaptability***, and ***reusability***. (See Figure 2.1.)

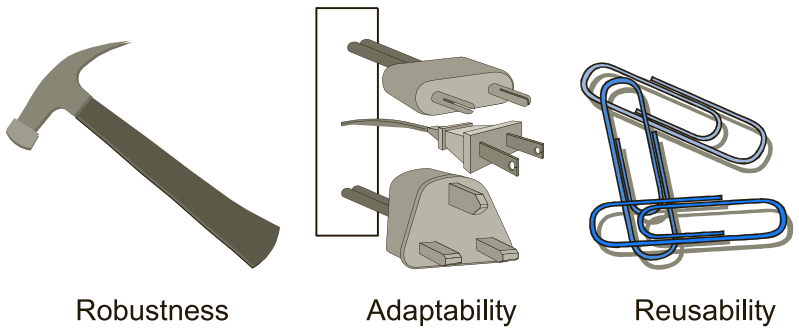


Figure 2.1: Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be ***robust***, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (perhaps representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error. More importantly, in ***life-critical applications***, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Java is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

2.1.2 Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following (see Figure 2.2):

- Abstraction
- Encapsulation
- Modularity

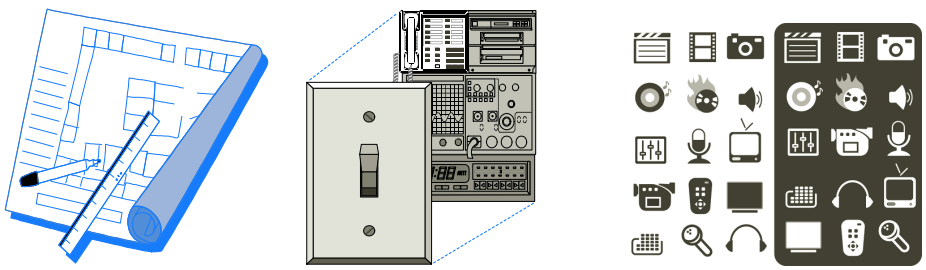


Figure 2.2: Principles of object-oriented design.

Abstraction

The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies ***what*** each operation does, but not ***how*** it does it. In Java, an ADT can be expressed by an ***interface***, which is simply a list of method declarations, where each method has an empty body. (We will say more about Java interfaces in Section 2.3.1.)

An ADT is realized by a concrete data structure, which is modeled in Java by a ***class***. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify ***how*** the operations are performed in the body of each method. A Java class is said to ***implement an interface*** if its methods include all the methods declared in the interface, thus providing a body for them. However, a class can have more methods than those of the interface.

Encapsulation

Another important principle of object-oriented design is ***encapsulation***; different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface. Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Modularity

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. ***Modularity*** refers to an organizing principle in which different components of a software system are divided into separate functional units. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system.

2.1.3 Design Patterns

Object-oriented design facilitates reusable, robust, and adaptable software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.

Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. Of special relevance to this book is the concept of a ***design pattern***, which describes a solution to a "typical" software design problem. A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of a name, which identifies the pattern; a context, which describes the scenarios for which this pattern can be applied; a template, which describes how the pattern is applied; and a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be consistently applied to implementations of data structures and algorithms. These design patterns fall into two groups—patterns for solving algorithm design problems and patterns for solving software engineering problems. Some of the algorithm design patterns we discuss include the following:

- Recursion (Chapter 5)
- Amortization (Sections 7.2.3, 11.4.4, and 14.7.3)
- Divide-and-conquer (Section 12.1.1)
- Prune-and-search, also known as decrease-and-conquer (Section 12.5.1)
- Brute force (Section 13.2.1)
- The greedy method (Sections 13.4.2, 14.6.2, and 14.7)
- Dynamic programming (Section 13.5)

Likewise, some of the software engineering design patterns we discuss include:

- Template method (Sections 2.3.3, 10.5.1, and 11.2.1)
- Composition (Sections 2.5.2, 2.6, and 9.2.1)
- Adapter (Section 6.1.3)
- Position (Sections 7.3, 8.1.2, and 14.7.3)
- Iterator (Section 7.4)
- Factory Method (Sections 8.3.1 and 11.2.1)
- Comparator (Sections 9.2.2, 10.3, and Chapter 12)
- Locator (Section 9.5.1)

Rather than explain each of these concepts here, however, we will introduce them throughout the text as noted above. For each pattern, be it for algorithm engineering or software engineering, we explain its general use and we illustrate it with at least one concrete example.

2.2 Inheritance

A natural way to organize various structural components of a software package is in a ***hierarchical*** fashion, with similar abstract definitions grouped together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. An example of such a hierarchy is shown in Figure 2.3. Using mathematical notations, the set of houses is a ***subset*** of the set of buildings, but a ***superset*** of the set of ranches. The correspondence between levels is often referred to as an ***"is a" relationship***, as a house is a building, and a ranch is a house.

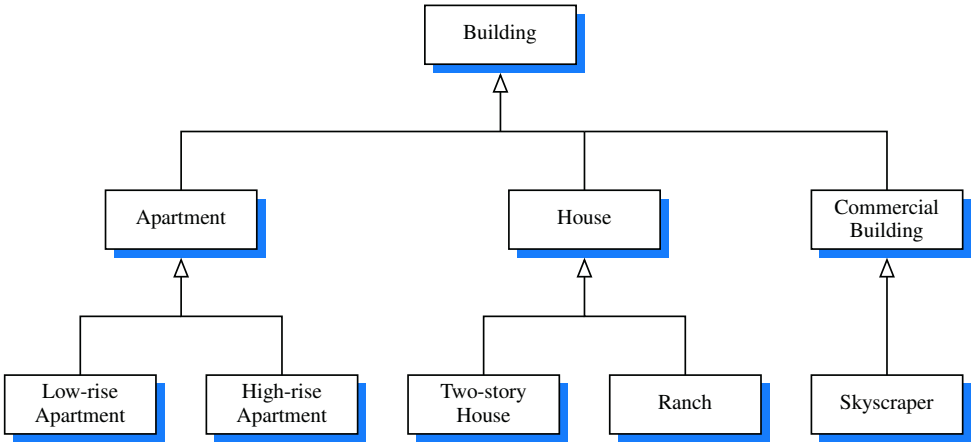


Figure 2.3: An example of an "is a" hierarchy involving architectural buildings.

A hierarchical design is useful in software development, as common functionality can be grouped at the most general level, thereby promoting reuse of code, while differentiated behaviors can be viewed as extensions of the general case. In object-oriented programming, the mechanism for a modular and hierarchical organization is a technique known as ***inheritance***. This allows a new class to be defined based upon an existing class as the starting point. In object-oriented terminology, the existing class is typically described as the ***base class***, ***parent class***, or ***superclass***, while the newly defined class is known as the ***subclass*** or ***child class***. We say that the subclass ***extends*** the superclass.

When inheritance is used, the subclass automatically inherits, as its starting point, all methods from the superclass (other than constructors). The subclass can differentiate itself from its superclass in two ways. It may ***augment*** the superclass by adding new fields and new methods. It may also ***specialize*** existing behaviors by providing a new implementation that ***overrides*** an existing method.

2.2.1 Extending the CreditCard Class

As an introduction to the use of inheritance, we revisit the CreditCard class of Section 1.7, designing a new subclass that, for lack of a better name, we name PredatoryCreditCard. The new class will differ from the original in two ways: (1) if an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged, and (2) there will be a mechanism for assessing a monthly interest charge on the outstanding balance, using an annual percentage rate (APR) specified as a constructor parameter.

Figure 2.4 provides a UML diagram that serves as an overview of our design for the new PredatoryCreditCard class as a subclass of the existing CreditCard class. The hollow arrow in that diagram indicates the use of inheritance, with the arrow oriented from the subclass to the superclass.

The PredatoryCreditCard class ***augments*** the original CreditCard class, adding a new instance variable named apr to store the annual percentage rate, and adding a new method named processMonth that will assess interest charges. The new class also ***specializes*** its superclass by ***overriding*** the original charge method in order to provide a new implementation that assess a $5 fee for an attempted overcharge.

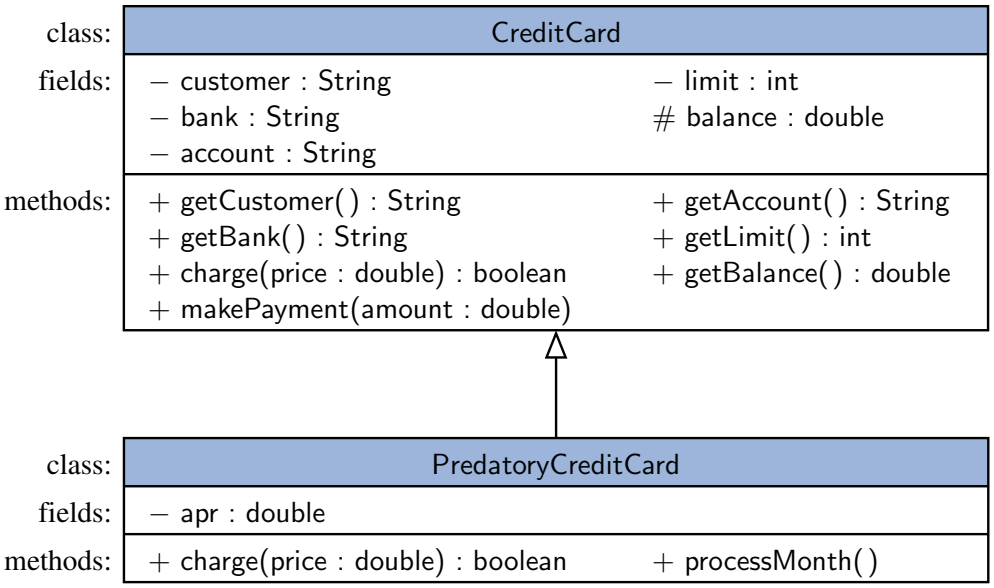


Figure 2.4: A UML diagram showing PredatoryCreditCard as a subclass of CreditCard. (See Figure 1.5 for the original CreditCard design.)

To demonstrate the mechanisms for inheritance in Java, Code Fragment 2.1 presents a complete implementation of the new PredatoryCreditCard class. We wish to draw attention to several aspects of the Java implementation.

We begin with the first line of the class definition, which indicates that the new class inherits from the existing CreditCard class by using Java's **extends** keyword followed by the name of its superclass. In Java, each class can extend exactly one other class. Because of this property, Java is said to allow only ***single inheritance*** among classes. We should also note that even if a class definition makes no explicit use of the **extends** clause, it automatically inherits from a class, java.lang.Object, which serves as the universal superclass in Java.

We next consider the declaration of the new apr instance variable, at line 3 of the code. Each instance of the PredatoryCreditCard class will store each of the variables inherited from the CreditCard definition (customer, bank, account, limit, and balance) in addition to the new apr variable. Yet we are only responsible for declaring the new instance variable within the subclass definition.

```
1  public class PredatoryCreditCard extends CreditCard {
2    // Additional instance variable
3    private double apr;                                  // annual percentage rate
4
5    // Constructor for this class
6    public PredatoryCreditCard(String cust, String bk, String acnt, int lim,
7                               double initialBal, double rate) {
8      super(cust, bk, acnt, lim, initialBal);            // initialize superclass attributes
9      apr = rate;
10   }
11
12   // A new method for assessing monthly interest charges
13   public void processMonth( ) {
14     if (balance > 0) {    // only charge interest on a positive balance
15       double monthlyFactor = Math.pow(1 + apr, 1.0/12); // compute monthly rate
16       balance *= monthlyFactor;                          // assess interest
17     }
18   }
19
20   // Overriding the charge method defined in the superclass
21   public boolean charge(double price) {
22     boolean isSuccess = super.charge(price);           // call inherited method
23     if (!isSuccess)
24       balance += 5;                                    // assess a $5 penalty
25     return isSuccess;
26   }
27 }
```

Code Fragment 2.1: A subclass of CreditCard that assesses interest and fees.

Constructors are never inherited in Java. Lines 6–10 of Code Fragment 2.1 define a constructor for the new class. When a PredatoryCreditCard instance is created, all of its fields must be properly initialized, including any inherited fields. For this reason, the first operation performed within the body of a constructor must be to invoke a constructor of the superclass, which is responsible for properly initializing the fields defined in the superclass.

In Java, a constructor of the superclass is invoked by using the keyword **super** with appropriate parameterization, as demonstrated at line 8 of our implementation:

```
super(cust, mk, acnt, lim, initialBal);
```

This use of the **super** keyword is very similar to use of the keyword **this** when invoking a different constructor within the same class (as described on page 15 of Section 1.2.2). If a constructor for a subclass does not make an explicit call to **super** or **this** as its first command, then an implicit call to **super**( ), the zero-parameter version of the superclass constructor, will be made. Returning our attention to the constructor for PredatoryCreditCard, after calling the superclass constructor with appropriate parameters, line 9 initializes the newly declared apr field. (That field was unknown to the superclass.)

The processMonth method is a new behavior, so there is no inherited version upon which to rely. In our model, this method should be invoked by the bank, once each month, to add new interest charges to the customer's balance. From a technical aspect, we note that this method accesses the value of the inherited balance field (at line 14), and potentially modifies that balance at line 16. This is permitted precisely because the balance attributed was declared with **protected** visibility in the original CreditCard class. (See Code Fragment 1.5.)

The most challenging aspect in implementing the processMonth method is making sure we have working knowledge of how an annual percentage rate translates to a monthly rate. We do not simply divide the annual rate by twelve to get a monthly rate (that would be too predatory, as it would result in a higher APR than advertised). The correct computation is to take the twelfth-root of $1 + \text{apr}$, and use that as a multiplicative factor. For example, if the APR is 0.0825 (representing 8.25%), we compute $\sqrt[12]{1.0825} \approx 1.006628$, and therefore charge 0.6628% interest per month. In this way, each \$100 of debt will amass \$8.25 of compounded interest in a year. Notice that we use the Math.pow method from Java's libraries.

Finally, we consider the new implementation of the charge method provided for the PredatoryCreditCard class (lines 21–27). This definition ***overrides*** the inherited method. Yet, our implementation of the new method relies on a call to the inherited method, with syntax **super**.charge(price) at line 22. The return value of that call designates whether the charge was successful. We examine that return value to decide whether to assess a fee, and in either case return that boolean to the caller, so that the new version of charge maintains a similar outward interface as the original.

2.2.2 Polymorphism and Dynamic Dispatch

The word ***polymorphism*** literally means "many forms." In the context of object-oriented design, it refers to the ability of a reference variable to take different forms. Consider, for example, the declaration of a variable having CreditCard as its type:

```
CreditCard card;
```

Because this is a reference variable, the statement declares the new variable, which does not yet refer to any card instance. While we have already seen that we can assign it to a newly constructed instance of the CreditCard class, Java also allows us to assign that variable to refer to an instance of the PredatoryCreditCard subclass. That is, we can do the following:

```
CreditCard card = new PredatoryCreditCard(...);    // parameters omitted
```

This is a demonstration of what is known as the ***Liskov Substitution Principle***, which states that a variable (or parameter) with a declared type can be assigned an instance from any direct or indirect subclass of that type. Informally, this is a manifestation of the "is a" relationship modeled by inheritance, as a predatory credit card is a credit card (but a credit card is not necessarily predatory).

We say that the variable, card, is ***polymorphic***; it may take one of many forms, depending on the specific class of the object to which it refers. Because card has been declared with type CreditCard, that variable may only be used to call methods that are declared as part of the CreditCard definition. So we can call card.makePayment(50) and card.charge(100), but a compilation error would be reported for the call card.processMonth( ) because a CreditCard is not guaranteed to have such a behavior. (That call could be made if the variable were originally declared to have PredatoryCreditCard as its type.)

An interesting (and important) issue is how Java handles a call such as card.charge(100) when the variable card has a declared type of CreditCard. Recall that the object referenced by card might be an instance of the CreditCard class or an instance of the PredatoryCreditCard class, and that there are distinct implementations of the charge method: CreditCard.charge and PredatoryCreditCard.charge. Java uses a process known as ***dynamic dispatch***, deciding at runtime to call the version of the method that is most specific to the *actual* type of the referenced object (not the declared type). So, if the object is a PredatoryCreditCard instance, it will execute the PredatoryCreditCard.charge method, even if the reference variable has a declared type of CreditCard.

Java also provides an **instanceof** operator that tests, at runtime, whether an instance satisfies as a particular type. For example, the evaluation of the boolean condition, (card **instanceof** PredatoryCreditCard), produces **true** if the object currently referenced by the variable card belongs to the PredatoryCreditCard class, or any further subclass of that class. (See Section 2.5.1 for further discusion.)

2.2.3 Inheritance Hierarchies

Although a subclass may not inherit from multiple superclasses in Java, a superclass may have many subclasses. In fact, it is quite common in Java to develop complex inheritance hierarchies to maximize the reusability of code.

As a second example of the use of inheritance, we develop a hierarchy of classes for iterating numeric progressions. A numeric progression is a sequence of numbers, where each number depends on one or more of the previous numbers. For example, an ***arithmetic progression*** determines the next number by adding a fixed constant to the previous value, and a ***geometric progression*** determines the next number by multiplying the previous value by a fixed constant. In general, a progression requires a first value, and a way of identifying a new value based on one or more previous values.

Our hierarchy stems from a general base class that we name Progression. This class produces the progression of whole numbers: 0, 1, 2, ... More importantly, this class has been designed so that it can easily be specialized by other progression types, producing a hierarchy given in Figure 2.5.

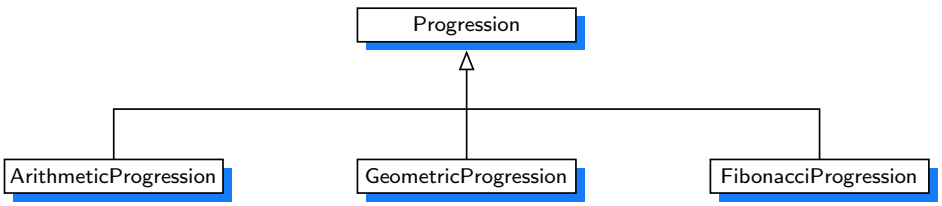


Figure 2.5: An overview of our hierarchy of progression classes.

Our implementation of the basic Progression class is provided in Code Fragment 2.2. This class has a single field, named current. It defines two constructors, one accepting an arbitrary starting value for the progression and the other using 0 as the default value. The remainder of the class includes three methods:

nextValue( ): A public method that returns the next value of the progression, implicitly advancing the value each time.

advance( ): A protected method that is responsible for advancing the value of current in the progression.

printProgression(n): A public utility that advances the progression n times while displaying each value.

Our decision to factor out the protected advance( ) method, which is called during the execution of nextValue( ), is to minimize the burden on subclasses, which are solely responsible for overriding the advance method to update the current field.

```
1  /** Generates a simple progression. By default: 0, 1, 2, ... */
2  public class Progression {
3
4    // instance variable
5    protected long current;
6
7    /** Constructs a progression starting at zero. */
8    public Progression( ) { this(0); }
9
10   /** Constructs a progression with given start value. */
11   public Progression(long start) { current = start; }
12
13   /** Returns the next value of the progression. */
14   public long nextValue( ) {
15     long answer = current;
16     advance( );      // this protected call is responsible for advancing the current value
17     return answer;
18   }
19
20   /** Advances the current value to the next value of the progression. */
21   protected void advance( ) {
22     current++;
23   }
24
25   /** Prints the next n values of the progression, separated by spaces. */
26   public void printProgression(int n) {
27     System.out.print(nextValue( ));           // print first value without leading space
28     for (int j=1; j < n; j++)
29       System.out.print(" " + nextValue( )); // print leading space before others
30     System.out.println( );                    // end the line
31   }
32 }
```

Code Fragment 2.2: General numeric progression class.

The body of the nextValue method temporarily records the current value of the progression, which will soon be returned, and then calls the protected advance method in order to update the value in preparation for a subsequent call.

The implementation of the advance method in our Progression class simply increments the current value. This method is the one that will be overridden by our specialized subclasses in order to alter the progression of numbers.

In the remainder of this section, we present three subclasses of the Progression class—ArithmeticProgression, GeometricProgression, and FibonacciProgression–which respectively produce arithmetic, geometric, and Fibonacci progressions.

An Arithmetic Progression Class

Our first example of a specialized progression is an arithmetic progression. While the default progression increases its value by one in each step, an arithmetic progression adds a fixed constant to one term of the progression to produce the next. For example, using an increment of 4 for an arithmetic progression that starts at 0 results in the sequence $0, 4, 8, 12, \ldots$.

Code Fragment 2.3 presents our implementation of an ArithmeticProgression class, which relies on Progression as its base class. We include three constructor forms, with the most general (at lines 12–15) accepting an increment value and a start value, such that ArithmeticProgression(4, 2) produces the sequence $2, 6, 10, 14, \ldots$. The body of that constructor invokes the superclass constructor, with syntax **super**(start), to initialize current to the given start value, and then it initializes the increment field introduced by this subclass.

For convenience, we offer two additional constructors, so that the default progression produces $0, 1, 2, 3, \ldots$, and a one-parameter constructor produces an arithmetic progression with a given increment value (but a default starting value of 0).

Finally (and most importantly), we override the protected advance method so that the given increment is added to each successive value of the progression.

```
1  public class ArithmeticProgression extends Progression {
2
3    protected long increment;
4
5    /** Constructs progression 0, 1, 2, ... */
6    public ArithmeticProgression( ) { this(1, 0); }      // start at 0 with increment of 1
7
8    /** Constructs progression 0, stepsize, 2*stepsize, ... */
9    public ArithmeticProgression(long stepsize) { this(stepsize, 0); }      // start at 0
10
11   /** Constructs arithmetic progression with arbitrary start and increment. */
12   public ArithmeticProgression(long stepsize, long start) {
13     super(start);
14     increment = stepsize;
15   }
16
17   /** Adds the arithmetic increment to the current value. */
18   protected void advance( ) {
19     current += increment;
20   }
21 }
```

Code Fragment 2.3: Class for arithmetic progressions, which inherits from the general progression class shown in Code Fragment 2.2.

A Geometric Progression Class

Our second example of a specialized progression is a geometric progression, in which each value is produced by multiplying the preceding value by a fixed constant, known as the ***base*** of the geometric progression. The starting point of a geometric progression is traditionally 1, rather than 0, because multiplying 0 by any factor results in 0. As an example, a geometric progression with base 2, starting at value 1, produces the sequence $1, 2, 4, 8, 16, \ldots$.

Code Fragment 2.4 presents our implementation of a GeometricProgression class. It is quite similar to the ArithmeticProgression class in terms of the programming techniques used. In particular, it introduces one new field (the base of the geometric progression), provides three forms of a constructor for convenience, and overrides the protected advance method so that the current value of the progression is multiplied by the base at each step.

In the case of a geometric progression, we have chosen to have the default (zero-parameter) constructor use a starting value of 1 and a base of 2 so that it produces the progression $1, 2, 4, 8, \ldots$. The one-parameter version of the constructor accepts an arbitrary base and uses 1 as the starting value, thus GeometricProgression(3) produces the sequence $1, 3, 9, 27, \ldots$. Finally, we offer a two-parameter version accepting both a base and start value, such that GeometricProgression(3,2) produces the sequence $2, 6, 18, 54, \ldots$.

```
1  public class GeometricProgression extends Progression {
2
3    protected long base;
4
5    /** Constructs progression 1, 2, 4, 8, 16, ... */
6    public GeometricProgression( ) { this(2, 1); }            // start at 1 with base of 2
7
8    /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
9    public GeometricProgression(long b) { this(b, 1); }                  // start at 1
10
11   /** Constructs geometric progression with arbitrary base and start. */
12   public GeometricProgression(long b, long start) {
13     super(start);
14     base = b;
15   }
16
17   /** Multiplies the current value by the geometric base. */
18   protected void advance( ) {
19     current *= base;                          // multiply current by the geometric base
20   }
21 }
```

Code Fragment 2.4: Class for geometric progressions.

A Fibonacci Progression Class

As our final example, we demonstrate how to use our progression framework to produce a ***Fibonacci progression***. Each value of a Fibonacci series is the sum of the two most recent values. To begin the series, the first two values are conventionally 0 and 1, leading to the Fibonacci series $0, 1, 1, 2, 3, 5, 8, \ldots$. More generally, such a series can be generated from any two starting values. For example, if we start with values 4 and 6, the series proceeds as $4, 6, 10, 16, 26, 42, \ldots$.

Code Fragment 2.5 presents an implementation of the FibonacciProgression class. This class is markedly different from those for the arithmetic and geometric progressions because we cannot determine the next value of a Fibonacci series solely from the current one. We must maintain knowledge of the two most recent values. Our FibonacciProgression class introduces a new member, named prev, to store the value that proceeded the current one (which is stored in the inherited current field).

However, the question arises as to how to initialize the previous value in the constructor, when provided with the desired first and second values as parameters. The first should be stored as current so that it is reported by the first call to nextValue( ). Within that method call, an assignment will set the new current value (which will be the second value reported) equal to the first value plus the "previous." By initializing the previous value to (second − first), the initial advancement will set the new current value to first + (second − first) = second, as desired.

```
1  public class FibonacciProgression extends Progression {
2
3    protected long prev;
4
5    /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
6    public FibonacciProgression( ) { this(0, 1); }
7
8    /** Constructs generalized Fibonacci, with give first and second values. */
9    public FibonacciProgression(long first, long second) {
10     super(first);
11     prev = second − first;        // fictitious value preceding the first
12   }
13
14   /** Replaces (prev,current) with (current, current+prev). */
15   protected void advance( ) {
16     long temp = prev;
17     prev = current;
18     current += temp;
19   }
20 }
```

Code Fragment 2.5: Class for the Fibonacci progression.

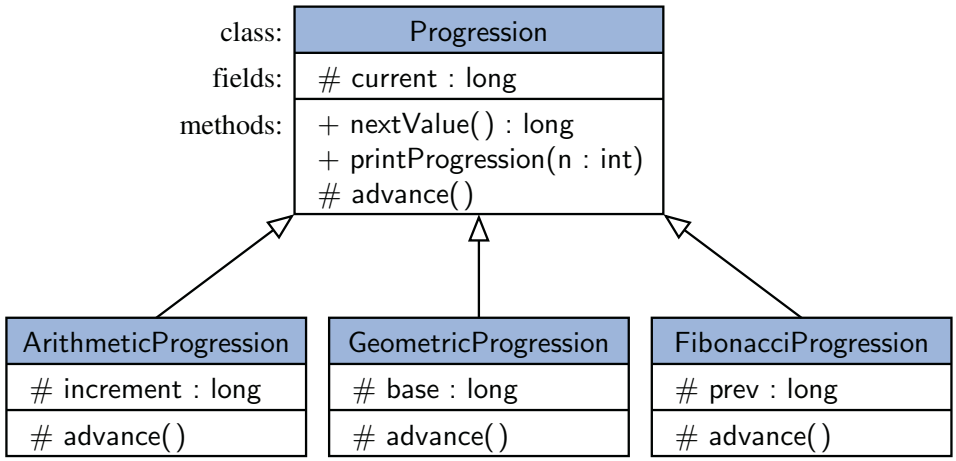


Figure 2.6: Detailed inheritance diagram for class Progression and its subclasses.

As a summary, Figure 2.6 presents a more detailed version of our inheritance design than was originally given in Figure 2.5. Notice that each of these classes introduces an additional field that allows it to properly implement the advance( ) method in an appropriate manner for its progression.

Testing Our Progression Hierarchy

To complete our example, we define a class TestProgression, shown in Code Fragment 2.6, which performs a simple test of each of the three classes. In this class, variable prog is polymorphic during the execution of the main method, since it references objects of class ArithmeticProgression, GeometricProgression, and FibonacciProgression in turn. When the main method of the TestProgression class is invoked by the Java runtime system, the output shown in Code Fragment 2.7 is produced.

The example presented in this section is admittedly simple, but it provides an illustration of an inheritance hierarchy in Java. As an interesting aside, we consider how quickly the numbers grow in the three progressions, and how long it would be before the long integers used for computations overflow. With the default increment of one, an arithmetic progression would not overflow for 2^{63} steps (that is approximately 10 billion billions). In contrast, a geometric progression with base $b = 3$ will overflow a long integer after 40 iterations, as $3^{40} > 2^{63}$. Likewise, the 94th Fibonacci number is greater than 2^{63}; hence, the Fibonacci progression will overflow a long integer after 94 iterations.

```
1  /** Test program for the progression hierarchy. */
2  public class TestProgression {
3    public static void main(String[ ] args) {
4      Progression prog;
5      // test ArithmeticProgression
6      System.out.print("Arithmetic progression with default increment: ");
7      prog = new ArithmeticProgression( );
8      prog.printProgression(10);
9      System.out.print("Arithmetic progression with increment 5: ");
10     prog = new ArithmeticProgression(5);
11     prog.printProgression(10);
12     System.out.print("Arithmetic progression with start 2: ");
13     prog = new ArithmeticProgression(5, 2);
14     prog.printProgression(10);
15     // test GeometricProgression
16     System.out.print("Geometric progression with default base: ");
17     prog = new GeometricProgression( );
18     prog.printProgression(10);
19     System.out.print("Geometric progression with base 3: ");
20     prog = new GeometricProgression(3);
21     prog.printProgression(10);
22     // test FibonacciProgression
23     System.out.print("Fibonacci progression with default start values: ");
24     prog = new FibonacciProgression( );
25     prog.printProgression(10);
26     System.out.print("Fibonacci progression with start values 4 and 6: ");
27     prog = new FibonacciProgression(4, 6);
28     prog.printProgression(8);
29   }
30 }
```

Code Fragment 2.6: Program for testing the progression classes.

```
Arithmetic progression with default increment: 0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5: 0 5 10 15 20 25 30 35 40 45
Arithmetic progression with start 2: 2 7 12 17 22 27 32 37 42 47
Geometric progression with default base: 1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3: 1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values: 0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6: 4 6 10 16 26 42 68 110
```

Code Fragment 2.7: Output of the TestProgression program of Code Fragment 2.6.

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.)

2.4 Exceptions

Exceptions are unexpected events that occur during the execution of a program. An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer. In Java, exceptions are objects that can be ***thrown*** by code that encounters an unexpected situation, or by the Java Virtual Machine, for example, if running out of memory. An exception may also be ***caught*** by a surrounding block of code that "handles" the problem in an appropriate fashion. If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console. In this section, we discuss common exception types in Java, as well as the syntax for throwing and catch exceptions within user-defined blocks of code.

2.4.1 Catching Exceptions

If an exception occurs and is not handled, then the Java runtime system will terminate the program after printing an appropriate message together with a trace of the runtime stack. The stack trace shows the series of nested method calls that were active at the time the exception occurred, as in the following example:

```
Exception in thread "main" java.lang.NullPointerException
  at java.util.ArrayList.toArray(ArrayList.java:358)
  at net.datastructures.HashChainMap.bucketGet(HashChainMap.java:35)
  at net.datastructures.AbstractHashMap.get(AbstractHashMap.java:62)
  at dsaj.design.Demonstration.main(Demonstration.java:12)
```

However, before a program is terminated, each method on the stack trace has an opportunity to ***catch*** the exception. Starting with the most deeply nested method in which the exception occurs, each method may either catch the exception, or allow it to pass through to the method that called it. For example, in the above stack trace, the ArrayList.java method had the first opportunity to catch the exception. Since it did not do so, the exception was passed upward to the HashChainMap.bucketGet method, which in turn ignored the exception, causing it to pass further upward to the AbstractHashMap.get method. The final opportunity to catch the exception was in the Demonstration.main method, but since it did not do so, the program terminated with the above diagnostic message.

The general methodology for handling exceptions is a ***try-catch*** construct in which a guarded fragment of code that might throw an exception is executed. If it throws an exception, then that exception is ***caught*** by having the flow of control jump to a predefined **catch** block that contains the code to analyze the exception and apply an appropriate resolution. If no exception occurs in the guarded code, all catch blocks are ignored.

A typical syntax for a ***try-catch statement*** in Java is as follows:

```
try {
    guardedBody
} catch (exceptionType_1 variable_1) {
    remedyBody_1
} catch (exceptionType_2 variable_2) {
    remedyBody_2
} ...
    ...
```

Each $exceptionType_i$ is the type of some exception, and each $variable_i$ is a valid Java variable name.

The Java runtime environment begins performing a try-catch statement such as this by executing the block of statements, *guardedBody*. If no exceptions are generated during this execution, the flow of control continues with the first statement beyond the last line of the entire try-catch statement.

If, on the other hand, the block, guardedBody, generates an exception at some point, the execution of that block immediate terminates and execution jumps to the **catch** block whose *exceptionType* most closely matches the exception thrown (if any). The *variable* for this catch statement references the exception object itself, which can be used in the block of the matching **catch** statement. Once execution of that **catch** block completes, control flow continues with the first statement beyond the entire try-catch construct.

If an exception occurs during the execution of the block, guardedBody, that does not match any of the exception types declared in the catch statements, that exception is rethrown in the surrounding context.

There are several possible reactions when an exception is caught. One possibility is to print out an error message and terminate the program. There are also some interesting cases in which the best way to handle an exception is to quietly catch and ignore it (this can be done by having an empty body as a **catch** block). Another legitimate way of handling exceptions is to create and throw another exception, possibly one that specifies the exceptional condition more precisely.

We note briefly that try-catch statements in Java support a few advanced techniques that we will not use in this book. There can be an optional **finally** clause with a body that will be executed whether or not an exception happens in the original guarded body; this can be useful, for example, to close a file before proceeding onward. Java SE 7 introduced a new syntax known as a "try with resource" that provides even more advanced cleanup techniques for resources such as open files that must be properly cleaned up. Also as of Java SE 7, each catch statement can designate multiple exception types that it handles; previously, a separate clause would be needed for each one, even if the same remedy were applied in each case.

```
1  public static void main(String[ ] args) {
2    int n = DEFAULT;
3    try {
4      n = Integer.parseInt(args[0]);
5      if (n <= 0) {
6        System.out.println("n must be positive. Using default.");
7        n = DEFAULT;
8      }
9    } catch (ArrayIndexOutOfBoundsException e) {
10     System.out.println("No argument specified for n. Using default.");
11   } catch (NumberFormatException e) {
12     System.out.println("Invalid integer argument. Using default.");
13   }
14 }
```

Code Fragment 2.13: A demonstration of catching an exception.

As a tangible example of a try-catch statement, we consider the simple application presented in Code Fragment 2.13. This main method attempts to interpret the first command-line argument as a positive integer. (Command-line arguments were introduced on page 16.)

The statement at risk of throwing an exception, at line 4, is the command n = Integer.parseInt(args[0]). That command may fail for one of two reasons. First, the attempt to access args[0] will fail if the user did not specify any arguments, and thus, the array args is empty. An ArrayIndexOutOfBoundsException will be thrown in that case (and caught by us at line 9). The second potential exception is when calling the Integer.parseInt method. That command succeeds so long as the parameter is a string that is a legitimate integer representation, such as "2013". Of course, since a command-line argument can be any string, the user might provide an invalid integer representation, in which case the parseInt method throws a NumberFormatException (caught by us at line 11).

A final condition we wish to enforce is that the integer specified by the user is positive. To test this property, we rely on a traditional conditional statement (lines 5–8). However, notice that we have placed that conditional statement within the primary body of the try-catch statement. That conditional statement will only be evaluated if the command at line 4 succeeded without exception; had an exception occurred at line 4, the primary try block is terminated, and control proceeds directly to the exception handling for the appropriate catch statement.

As an aside, if we had been willing to use the same error message for the two exceptional cases, we can use a single catch clause with the following syntax:

```
} catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
  System.out.println("Using default value for n.");
}
```

2.4.2 Throwing Exceptions

Exceptions originate when a piece of Java code finds some sort of problem during execution and ***throws*** an exception object. This is done by using the **throw** keyword followed by an instance of the exception type to be thrown. It is often convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a **throw** statement is typically written as follows:

```
throw new exceptionType(parameters);
```

where *exceptionType* is the type of the exception and the parameters are sent to that type's constructor; most exception types offer a version of a constructor that accepts an error message string as a parameter.

As an example, the following method takes an integer parameter, which it expects to be positive. If a negative integer is sent, an IllegalArgumentException is thrown.

```
public void ensurePositive(int n) {
  if (n < 0)
    throw new IllegalArgumentException("That's not positive!");
  // ...
}
```

The execution of a **throw** statement immediately terminates the body of a method.

The Throws Clause

When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method. It does not matter whether the exception is directly from a **throw** statement in that method body, or propagated upward from a secondary method call made from within the body.

The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual **throw** statement). For example, the parseInt method of the Integer class has the following formal signature:

```
public static int parseInt(String s) throws NumberFormatException;
```

The designation "**throws** NumberFormatException" warns users about the possibility of an exceptional case, so that they might be better prepared to handle an exception that may arise. If one of many exception types may possibly be thrown, all such types can be listed, separated with commas. Alternatively, it may be possible to list an appropriate superclass that encompasses all specific exceptions that may be thrown.

The use of a **throws** clause in a method signature does not take away the responsibility of properly documenting all possible exceptions through the use of the @throws tag within a javadoc comment (see Section 1.9.4). The type and reasons for any potential exceptions should always be properly declared in the documentation for a method.

In contrast, the use of the **throws** clause in a method signature is optional for many types of exceptions. For example, the documentation for the nextInt( ) method of the Scanner class makes clear that three different exception types may arise:

- An IllegalStateException, if the scanner has been closed
- A NoSuchElementException, if the scanner is active, but there is currently no token available for input
- An InputMismatchException, if the next available token does not represent an integer

However, no potential exceptions are formally declared within the method signature; they are only noted in the documentation.

To better understand the functional purpose of the **throws** declaration in a method signature, it is helpful to know more about the way Java organizes its hierarchy of exception types.

2.4.3 Java's Exception Hierarchy

Java defines a rich inheritance hierarchy of all objects that are deemed Throwable. We show a small portion of this hierarchy in Figure 2.7. The hierarchy is intentionally divided into two subclasses: Error and Exception. ***Errors*** are typically thrown only by the Java Virtual Machine and designate the most serious situations that are unlikely to be recoverable, such as when the virtual machine is asked to execute a corrupt class file, or when the system runs out of memory. In contrast, ***exceptions*** designate situations in which a running program might reasonably be able to recover, for example, when unable to open a data file.

Checked and Unchecked Exceptions

Java provides further refinement by declaring the RuntimeException class as an important subclass of Exception. All subtypes of RuntimeException in Java are officially treated as ***unchecked exceptions***, and any exception type that is not part of the RuntimeException is a ***checked exception***.

The intent of the design is that runtime exceptions occur entirely due to mistakes in programming logic, such as using a bad index with an array, or sending an inappropriate value as a parameter to a method. While such programming errors

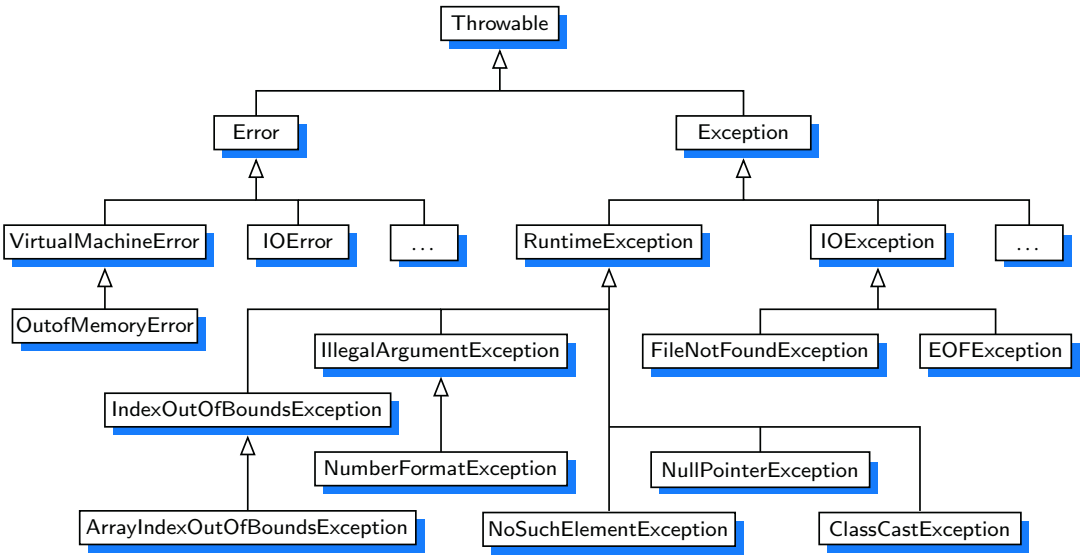


Figure 2.7: A small portion of Java's hierarchy of Throwable types.

will certainly occur as part of the software development process, they should presumably be resolved before software reaches production quality. Therefore, it is not in the interest of efficiency to explicitly check for each such mistake at runtime, and thus these are designated as "unchecked" exceptions.

In contrast, other exceptions occur because of conditions that cannot easily be detected until a program is executing, such as an unavailable file or a failed network connection. Those are typically designated as "checked" exceptions in Java (and thus, not a subtype of RuntimeException).

The designation between checked and unchecked exceptions plays a significant role in the syntax of the language. In particular, ***all checked exceptions that might propagate upward from a method must be explicitly declared in its signature.***

A consequence is that if one method calls a second method declaring checked exceptions, then the call to that second method must either be guarded within a try-catch statement, or else the calling method must itself declare the checked exceptions in its signature, since there is risk that such an exception might propagate upward from the calling method.

Defining New Exception Types

In this book, we will rely entirely on existing RuntimeException types to designate various requirements on the use of our data structures. However, some libraries define new classes of exceptions to describe more specific conditions. Specialized exceptions should inherit either from the Exception class (if checked), from the RuntimeException class (if unchecked), or from an existing Exception subtype that is more relevant.

2.5 Casting and Generics

In this section, we discuss casting among reference variables, as well as a technique, called generics, that allows us to define methods and classes that work with a variety of data types without the need for explicit casting.

2.5.1 Casting

We begin our discussion with methods for type conversions for objects.

Widening Conversions

A ***widening conversion*** occurs when a type T is converted into a "wider" type U. The following are common cases of widening conversions:

- T and U are class types and U is a superclass of T.
- T and U are interface types and U is a superinterface of T.
- T is a class that implements interface U.

Widening conversions are automatically performed to store the result of an expression into a variable, without the need for an explicit cast. Thus, we can directly assign the result of an expression of type T into a variable v of type U when the conversion from T to U is a widening conversion. When discussing polymorphism on page 68, we gave the following example of an implicit widening cast, assigning an instance of the narrower PredatoryCreditCard class to a variable of the wider CreditCard type:

```
CreditCard card = new PredatoryCreditCard(...);    // parameters omitted
```

The correctness of a widening conversion can be checked by the compiler and its validity does not require testing by the Java runtime environment during program execution.

Narrowing Conversions

A ***narrowing conversion*** occurs when a type T is converted into a "narrower" type S. The following are common cases of narrowing conversions:

- T and S are class types and S is a subclass of T.
- T and S are interface types and S is a subinterface of T.
- T is an interface implemented by class S.

In general, a narrowing conversion of reference types requires an explicit cast. Also, the correctness of a narrowing conversion may not be verifiable by the compiler. Thus, its validity should be tested by the Java runtime environment during program execution.

The example code fragment below shows how to use a cast to perform a narrowing conversion from type PredatoryCreditCard to type CreditCard.

```
CreditCard card = new PredatoryCreditCard(...);          // widening
PredatoryCreditCard pc = (PredatoryCreditCard) card;     // narrowing
```

Although variable card happens to reference an instance of a PredatoryCreditCard, the variable has declared type, CreditCard. Therefore, the assignment pc = card is a narrowing conversion and requires an explicit cast that will be evaluated at runtime (as not all cards are predatory).

Casting Exceptions

In Java, we can cast an object reference o of type T into a type S, provided the object o is referring to is actually of type S. If, on the other hand, object o is not also of type S, then attempting to cast o to type S will throw an exception called ClassCastException. We illustrate this rule in the following code fragment, using Java's Number abstract class, which is a superclass of both Integer and Double.

```
Number n;
Integer i;
n = new Integer(3);
i = (Integer) n;                 // This is legal
n = new Double(3.1415);
i = (Integer) n;                 // This is illegal
```

To avoid problems such as this and to avoid peppering our code with try-catch blocks every time we perform a cast, Java provides a way to make sure an object cast will be correct. Namely, it provides an operator, **instanceof**, that allows us to test whether an object variable is referring to an object that belongs to a particular type. The syntax for this operator is *objectReference* **instanceof** *referenceType*, where *objectReference* is an expression that evaluates to an object reference and *referenceType* is the name of some existing class, interface, or enum (Section 1.3). If *objectReference* is indeed an instance satisfying *referenceType*, then the operator returns **true**; otherwise, it returns **false**. Thus, we can avoid a ClassCastException from being thrown in the code fragment above by modifying it as follows:

```
Number n;
Integer i;
n = new Integer(3);
if (n instanceof Integer)
  i = (Integer) n;               // This is legal
n = new Double(3.1415);
if (n instanceof Integer)
  i = (Integer) n;               // This will not be attempted
```

Casting with Interfaces

Interfaces allow us to enforce that objects implement certain methods, but using interface variables with concrete objects sometimes requires casting. Suppose we declare a Person interface as shown in Code Fragment 2.14. Note that method equals of the Person interface takes one parameter of type Person. Thus, we can pass an object of any class implementing the Person interface to this method.

```
1  public interface Person {
2    public boolean equals(Person other);          // is this the same person?
3    public String getName( );                     // get this person's name
4    public int getAge( );                         // get this person's age
5  }
```

Code Fragment 2.14: Interface Person.

In Code Fragment 2.15, we show a class, Student, that implements Person. Because the parameter to equals is a Person, the implementation must not assume that it is necessarily of type Student. Instead, it first uses the **instanceof** operator at line 15, returning **false** if the argument is not a student (since it surely is not the student in question). Only after verifying that the parameter is a student, is it explicitly cast to a Student, at which point its id field can be accessed.

```
1  public class Student implements Person {
2    String id;
3    String name;
4    int age;
5    public Student(String i, String n, int a) {      // simple constructor
6      id = i;
7      name = n;
8      age = a;
9    }
10   protected int studyHours( ) { return age/2;}     // just a guess
11   public String getID( ) { return id;}             // ID of the student
12   public String getName( ) { return name; }        // from Person interface
13   public int getAge( ) { return age; }             // from Person interface
14   public boolean equals(Person other) {            // from Person interface
15     if (!(other instanceof Student)) return false; // cannot possibly be equal
16     Student s = (Student) other;                   // explicit cast now safe
17     return id.equals(s.id);                        // compare IDs
18   }
19   public String toString( ) {                      // for printing
20     return "Student(ID:" + id + ", Name:" + name + ", Age:" + age + ")";
21   }
22 }
```

Code Fragment 2.15: Class Student implementing interface Person.

2.5.2 Generics

Java includes support for writing ***generic*** classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts. The generics framework allows us to define a class in terms of a set of ***formal type parameters***, which can then be used as the declared type for variables, parameters, and return values within the class definition. Those formal type parameters are later specified when using the generic class as a type elsewhere in a program.

To better motivate the use of generics, we consider a simple case study. Often, we wish to treat a pair of related values as a single object, for example, so that the pair can be returned from a method. A solution is to define a new class whose instances store both values. This is our first example of an object-oriented design pattern known as the ***composition design pattern***. If we know, for example, that we want a pair to store a string and a floating-point number, perhaps to store a stock ticker label and a price, we could easily design a custom class for that purpose. However, for another purpose, we might want to store a pair that consists of a Book object and an integer that represents a quantity. The goal of generic programming is to be able to write a single class that can represent all such pairs.

The generics framework was not a part of the original Java language; it was added as part of Java SE 5. Prior to that, generic programming was implemented by relying heavily on Java's Object class, which is the universal supertype of all objects (including the wrapper types corresponding to primitives). In that "classic" style, a generic pair might be implemented as shown in Code Fragment 2.16.

```
1  public class ObjectPair {
2    Object first;
3    Object second;
4    public ObjectPair(Object a, Object b) {          // constructor
5      first = a;
6      second = b;
7    }
8    public Object getFirst( ) { return first; }
9    public Object getSecond( ) { return second;}
10 }
```

Code Fragment 2.16: Representing a generic pair of objects using a classic style.

An ObjectPair instance stores the two objects that are sent to the constructor, and provides individual accessors for each component of the pair. With this definition, a pair can be declared and instantiated with the following command:

```
ObjectPair bid = new ObjectPair("ORCL", 32.07);
```

This instantiation is legal because the parameters to the constructor undergo widening conversions. The first parameter, "ORCL", is a String, and thus also an Object.

The second parameter is a **double**, but it is automatically boxed into a Double, which then qualifies as an Object. (For the record, this is not quite the "classic" style, as automatic boxing was not introduced until Java SE 5.)

The drawback of the classic approach involves use of the accessors, both of which formally return an Object reference. Even if we know that the first object is a string in our application, we cannot legally make the following assignment:

```
String stock = bid.getFirst( );          // illegal; compile error
```

This represents a narrowing conversion from the declared return type of Object to the variable of type String. Instead, an explicit cast is required, as follows:

```
String stock = (String) bid.getFirst( ); // narrowing cast: Object to String
```

With the classic style for generics, code became rampant with such explicit casts.

Using Java's Generics Framework

With Java's generics framework, we can implement a pair class using formal type parameters to represent the two relevant types in our composition. An implementation using this framework is given in Code Fragment 2.17.

```
1  public class Pair<A,B> {
2    A first;
3    B second;
4    public Pair(A a, B b) {                    // constructor
5      first = a;
6      second = b;
7    }
8    public A getFirst( ) { return first; }
9    public B getSecond( ) { return second;}
10 }
```

Code Fragment 2.17: Representing a pair of objects with generic type parameters.

Angle brackets are used at line 1 to enclose the sequence of formal type parameters. Although any valid identifier can be used for a formal type parameter, single-letter uppercase names are conventionally used (in this example, A and B). We may then use these type parameters within the body of the class definition. For example, we declare instance variable, first, to have type A; we similarly use A as the declared type for the first constructor parameter and for the return type of method, getFirst.

When subsequently declaring a variable with such a parameterize type, we must explicitly specify ***actual type parameters*** that will take the place of the generic formal type parameters. For example, to declare a variable that is a pair holding a stock-ticker string and a price, we write the following:

```
Pair<String,Double> bid;
```

Effectively, we have stated that we wish to have String serve in place of type A, and Double serve in place of type B for the pair known as bid. The actual types for generic programming must be object types, which is why we use the wrapper class Double instead of the primitive type **double**. (Fortunately, the automatic boxing and unboxing will work in our favor.)

We can subsequently instantiate the generic class using the following syntax:

```
bid = new Pair<>("ORCL", 32.07);          // rely on type inference
```

After the **new** operator, we provide the name of the generic class, then an empty set of angle brackets (known as the "diamond"), and finally the parameters to the constructor. An instance of the generic class is created, with the actual types for the formal type parameters determined based upon the original declaration of the variable to which it is assigned (bid in this example). This process is known as ***type inference***, and was introduced to the generics framework in Java SE 7.

It is also possible to use a style that existed prior to Java SE 7, in which the generic type parameters are explicitly specified between angle brackets during instantiation. Using that style, our previous example would be implemented as:

```
bid = new Pair<String,Double>("ORCL", 32.07);    // give explicit types
```

However, it is important that one of the two above styles be used. If angle brackets are entirely omitted, as in the following example,

```
bid = new Pair("ORCL", 32.07);                   // classic style
```

this reverts to the classic style, with Object automatically used for all generic type parameters, and resulting in a compiler warning when assigning to a variable with more specific types.

Although the syntax for the declaration and instantiation of objects using the generics framework is slightly more cluttered than the classic style, the advantage is that there is no longer any need for explicit narrowing casts from Object to a more specific type. Continuing with our example, since bid was declared with actual type parameters <String,Double>, the return type of the getFirst( ) method is String, and the return type of the getSecond( ) method is Double. Unlike the classic style, we can make the following assignments without any explicit casting (although there is still an automatic unboxing of the Double):

```
String stock = bid.getFirst( );
double price = bid.getSecond( );
```

Generics and Arrays

There is an important caveat related to generic types and the use of arrays. Although Java allows the declaration of an array storing a parameterized type, it does not technically allow the instantiation of new arrays involving those types. Fortunately, it allows an array defined with a parameterized type to be initialized with a newly created, nonparametric array, which can then be cast to the parameterized type. Even so, this latter mechanism causes the Java compiler to issue a warning, because it is not 100% type-safe.

We will see this issue arise in two ways:

- Code outside a generic class may wish to declare an array storing instances of the generic class with actual type parameters.
- A generic class may wish to declare an array storing objects that belong to one of the formal parameter types.

As an example of the first use case, we continue with our stock market example and presume that we would like to keep an array of Pair<String,Double> objects. Such an array can be declared with a parameterized type, but it must be instantiated with an *unparameterized* type and then cast back to the parameterized type. We demonstrate this usage in the following:

```
Pair<String,Double>[ ] holdings;
holdings = new Pair<String,Double>[25];     // illegal; compile error
holdings = new Pair[25];     // correct, but warning about unchecked cast
holdings[0] = new Pair<>("ORCL", 32.07);   // valid element assignment
```

As an example of the second use case, assume that we want to create a generic Portfolio class that can store a fixed number of generic entries in an array. If the class uses <T> as a parameterized type, it can declare an array of type T[ ], but it cannot directly instantiate such an array. Instead, a common approach is to instantiate an array of type Object[ ], and then make a narrowing cast to type T[ ], as shown in the following:

```
public class Portfolio<T> {
  T[ ] data;
  public Portfolio(int capacity) {
    data = new T[capacity];                  // illegal; compiler error
    data = (T[ ]) new Object[capacity];     // legal, but compiler warning
  }
  public T get(int index) { return data[index]; }
  public void set(int index, T element) { data[index] = element; }
}
```

Generic Methods

The generics framework allows us to define generic versions of individual methods (as opposed to generic versions of entire classes). To do so, we include a generic formal type declaration among the method modifiers.

For example, we show below a nonparametric GenericDemo class with a parameterized static method that can reverse an array containing elements of any object type.

```
public class GenericDemo {
  public static <T> void reverse(T[ ] data) {
    int low = 0, high = data.length − 1;
    while (low < high) {                    // swap data[low] and data[high]
      T temp = data[low];
      data[low++] = data[high];             // post-increment of low
      data[high−−] = temp;                  // post-decrement of high
    }
  }
}
```

Note the use of the <T> modifier to declare the method to be generic, and the use of the type T within the method body, when declaring the local variable, temp.

The method can be called using the syntax, GenericDemo.reverse(books), with type inference determining the generic type, assuming books is an array of some object type. (This generic method cannot be applied to primitive arrays, because autoboxing does not apply to entire arrays.)

As an aside, we note that we could have implemented a reverse method equally well using a classic style, acting upon an Object[ ] array.

Bounded Generic Types

By default, when using a type name such as T in a generic class or method, a user can specify any object type as the actual type of the generic. A formal parameter type can be restricted by using the **extends** keyword followed by a class or interface. In that case, only a type that satisfies the stated condition is allowed to substitute for the parameter. The advantage of such a bounded type is that it becomes possible to call any methods that are guaranteed by the stated bound.

As an example, we might declare a generic ShoppingCart that could only be instantiated with a type that satisfied the Sellable interface (from Code Fragment 2.8 on page 77). Such a class would be declared beginning with the line:

```
public class ShoppingCart<T extends Sellable> {
```

Within that class definition, we would then be allowed to call methods such as description( ) and lowestPrice( ) on any instances of type T.

2.6 Nested Classes

Java allows a class definition to be ***nested*** inside the definition of another class. The main use for nesting classes is when defining a class that is strongly affiliated with another class. This can help increase encapsulation and reduce undesired name conflicts. Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure. We will use nested classes in many implementations within this book.

To demonstrate the mechanics of a nested class, we consider a new Transaction class to support logging of transactions associated with a credit card. That new class definition can be nested within the CreditCard class using a style as follows:

```
public class CreditCard {
  private static class Transaction { /* details omitted */ }

  // instance variable for a CreditCard
  Transaction[ ] history;          // keep log of all transactions for this card
}
```

The containing class is known as the ***outer class***. The ***nested class*** is formally a member of the outer class, and its fully qualified name is *OuterName.NestedName*. For example, with the above definition the nested class is CreditCard.Transaction, although we may refer to it simply as Transaction from within the CreditCard class.

Much like packages (see Section 1.8), the use of nested classes can help reduce name collisions, as it is perfectly acceptable to have another class named Transaction nested within some other class (or as a self-standing class).

A nested class has an independent set of modifiers from the outer class. Visibility modifiers (e.g., **public**, **private**) effect whether the nested class definition is accessible beyond the outer class definition. For example, a **private** nested class can be used by the outer class, but by no other classes.

A nested class can also be designated as either **static** or (by default) nonstatic, with significant consequences. A **static** nested class is most like a traditional class; its instances have no association with any specific instance of the outer class.

A nonstatic nested class is more commonly known as an ***inner class*** in Java. An instance of an inner class can only be created from within a nonstatic method of the outer class, and that inner instance becomes associated with the outer instance that creates it. Each instance of an inner class implicitly stores a reference to its associated outer instance, accessible from within the inner class methods using the syntax *OuterName*.**this** (as opposed to **this**, which refers to the inner instance). The inner instance also has private access to all members of its associated outer instance, and can rely on the formal type parameters of the outer class, if generic.

2.7 Exercises

Reinforcement

R-2.1 Give three examples of life-critical software applications.

R-2.2 Give an example of a software application in which adaptability can mean the difference between a prolonged lifetime of sales and bankruptcy.

R-2.3 Describe a component from a text-editor GUI and the methods that it encapsulates.

R-2.4 Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance has **private** visibility. Why is the following implementation of the PredatoryCreditCard.charge method flawed?

```
public boolean charge(double price) {
  boolean isSuccess = super.charge(price);
  if (!isSuccess)
    charge(5);              // the penalty
  return isSuccess;
}
```

R-2.5 Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance has **private** visibility. Why is the following implementation of the PredatoryCreditCard.charge method flawed?

```
public boolean charge(double price) {
  boolean isSuccess = super.charge(price);
  if (!isSuccess)
    super.charge(5);        // the penalty
  return isSuccess;
}
```

R-2.6 Give a short fragment of Java code that uses the progression classes from Section 2.2.3 to find the eighth value of a Fibonacci progression that starts with 2 and 2 as its first two values.

R-2.7 If we choose an increment of 128, how many calls to the nextValue method from the ArithmeticProgression class of Section 2.2.3 can we make before we cause a long-integer overflow?

R-2.8 Can two interfaces mutually extend each other? Why or why not?

R-2.9 What are some potential efficiency disadvantages of having very deep inheritance trees, that is, a large set of classes, A, B, C, and so on, such that B extends A, C extends B, D extends C, etc.?

R-2.10 What are some potential efficiency disadvantages of having very shallow inheritance trees, that is, a large set of classes, A, B, C, and so on, such that all of these classes extend a single class, Z?

R-2.11 Consider the following code fragment, taken from some package:

```
public class Maryland extends State {
  Maryland( ) { /* null constructor */ }
  public void printMe( ) { System.out.println("Read it."); }
  public static void main(String[ ] args) {
    Region east = new State( );
    State md = new Maryland( );
    Object obj = new Place( );
    Place usa = new Region( );
    md.printMe( );
    east.printMe( );
    ((Place) obj).printMe( );
    obj = md;
    ((Maryland) obj).printMe( );
    obj = usa;
    ((Place) obj).printMe( );
    usa = md;
    ((Place) usa).printMe( );
  }
}
class State extends Region {
  State( ) { /* null constructor */ }
  public void printMe( ) { System.out.println("Ship it."); }
}
class Region extends Place {
  Region( ) { /* null constructor */ }
  public void printMe( ) { System.out.println("Box it."); }
}
class Place extends Object {
  Place( ) { /* null constructor */ }
  public void printMe( ) { System.out.println("Buy it."); }
}
```

What is the output from calling the main( ) method of the Maryland class?

R-2.12 Draw a class inheritance diagram for the following set of classes:

- Class Goat extends Object and adds an instance variable tail and methods milk( ) and jump( ).
- Class Pig extends Object and adds an instance variable nose and methods eat(food) and wallow( ).
- Class Horse extends Object and adds instance variables height and color, and methods run( ) and jump( ).
- Class Racer extends Horse and adds a method race( ).
- Class Equestrian extends Horse and adds instance variable weight and isTrained, and methods trot( ) and isTrained( ).

R-2.13 Consider the inheritance of classes from Exercise R-2.12, and let d be an object variable of type Horse. If d refers to an actual object of type Equestrian, can it be cast to the class Racer? Why or why not?

R-2.14 Give an example of a Java code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints the following error message:
"`Don't try buffer overflow attacks in Java!`"

R-2.15 If the parameter to the makePayment method of the CreditCard class (see Code Fragment 1.5) were a negative number, that would have the effect of *raising* the balance on the account. Revise the implementation so that it throws an IllegalArgumentException if a negative amount is sent as a parameter.

Creativity

C-2.16 Suppose you are on the design team for a new e-book reader. What are the primary classes and methods that the Java software for your reader will need? You should include an inheritance diagram for this code, but you don't need to write any actual code. Your software architecture should at least include ways for customers to buy new books, view their list of purchased books, and read their purchased books.

C-2.17 Most modern Java compilers have optimizers that can detect simple cases when it is logically impossible for certain statements in a program to ever be executed. In such cases, the compiler warns the programmer about the useless code. Write a short Java method that contains code for which it is provably impossible for that code to ever be executed, yet the Java compiler does not detect this fact.

C-2.18 The PredatoryCreditCard class provides a processMonth( ) method that models the completion of a monthly cycle. Modify the class so that once a customer has made ten calls to charge during a month, each additional call to that method in the current month results in an additional $1 surcharge.

C-2.19 Modify the PredatoryCreditCard class so that a customer is assigned a minimum monthly payment, as a percentage of the balance, and so that a late fee is assessed if the customer does not subsequently pay that minimum amount before the next monthly cycle.

C-2.20 Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance has **private** visibility, but a new **protected** method is added, with signature setBalance(newBalance). Show how to properly implement the method PredatoryCreditCard.processMonth( ) in this setting.

C-2.21 Write a program that consists of three classes, A, B, and C, such that B extends A and that C extends B. Each class should define an instance variable named "x" (that is, each has its own variable named x). Describe a way for a method in C to access and set A's version of x to a given value, without changing B or C's version.

C-2.22 Explain why the Java dynamic dispatch algorithm, which looks for the method to invoke for a call obj.foo( ), will never get into an infinite loop.

C-2.23 Modify the advance method of the FibonacciProgression class so as to avoid use of any temporary variable.

C-2.24 Write a Java class that extends the Progression class so that each value in the progression is the absolute value of the difference between the previous two values. You should include a default constructor that starts with 2 and 200 as the first two values and a parametric constructor that starts with a specified pair of numbers as the first two values.

C-2.25 Redesign the Progression class to be abstract and generic, producing a sequence of values of generic type T, and supporting a single constructor that accepts an initial value. Make all corresponding modifications to the rest of the classes in our hierarchy so that they remain as nongeneric classes, while inheriting from the new generic Progression class.

C-2.26 Use a solution to Exercise C-2.25 to create a new progression class for which each value is the square root of the previous value, represented as a Double. You should include a default constructor that has 65,536 as the first value and a parametric constructor that starts with a specified number as the first value.

C-2.27 Use a solution to Exercise C-2.25 to reimplement the FibonacciProgression subclass to rely on the BigInteger class, in order to avoid overflows all together.

C-2.28 Write a set of Java classes that can simulate an Internet application in which one party, Alice, is periodically creating a set of packets that she wants to send to Bob. An Internet process is continually checking if Alice has any packets to send, and if so, it delivers them to Bob's computer; Bob is periodically checking if his computer has a packet from Alice, and if so, he reads and deletes it.

C-2.29 Write a Java program that inputs a polynomial in standard algebraic notation and outputs the first derivative of that polynomial.

Projects

P-2.30 Write a Java program that inputs a document and then outputs a bar-chart plot of the frequencies of each alphabet character that appears within that document.

P-2.31 Write a Java program to simulate an ecosystem containing two types of creatures, ***bears*** and ***fish***. The ecosystem consists of a river, which is modeled as a relatively large array. Each cell of the array should contain an Animal object, which can be a Bear object, a Fish object, or **null**. In each time step, based on a random process, each animal either attempts to move into an adjacent array cell or stay where it is. If two animals of the same type are about to collide in the same cell, then they stay where they are, but they create a new instance of that type of animal, which is placed in a random empty (i.e., previously **null**) cell in the array. If a bear and a fish collide, however, then the fish dies (i.e., it disappears). Use actual object creation, via the **new** operator, to model the creation of new objects, and provide a visualization of the array after each time step.

P-2.32 Write a simulator as in the previous project, but add a boolean gender field and a floating-point strength field to each Animal object. Now, if two animals of the same type try to collide, then they only create a new instance of that type of animal if they are of different genders. Otherwise, if two animals of the same type and gender try to collide, then only the one of larger strength survives.

P-2.33 Write a Java program that simulates a system that supports the functions of an e-book reader. You should include methods for users of your system to "buy" new books, view their list of purchased books, and read their purchased books. Your system should use actual books, which have expired copyrights and are available on the Internet, to populate your set of available books for users of your system to "purchase" and read.

P-2.34 Define a Polygon interface that has methods area( ) and perimeter( ). Then implement classes for Triangle, Quadrilateral, Pentagon, Hexagon, and Octagon, which implement this interface, with the obvious meanings for the area( ) and perimeter( ) methods. Also implement classes, IsoscelesTriangle, EquilateralTriangle, Rectangle, and Square, which have the appropriate inheritance relationships. Finally, write a simple user interface, which allows users to create polygons of the various types, input their geometric dimensions, and then output their area and perimeter. For extra effort, allow users to input polygons by specifying their vertex coordinates and be able to test if two such polygons are similar.

P-2.35 Write a Java program that inputs a list of words, separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

P-2.36 Write a Java program that can "make change." Your program should take two numbers as input, one that is a monetary amount charged and the other that is a monetary amount given. It should then return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins can be based on the monetary system of any current or former government. Try to design your program so that it returns the fewest number of bills and coins as possible.

Chapter Notes

For a broad overview of developments in computer science and engineering, we refer the reader to *The Computer Science and Engineering Handbook* [89]. For more information about the Therac-25 incident, please see the paper by Leveson and Turner [65].

The reader interested in studying object-oriented programming further is referred to the books by Booch [16], Budd [19], and Liskov and Guttag [67]. Liskov and Guttag also provide a nice discussion of abstract data types, as does the book chapter by Demurjian [28] in the *The Computer Science and Engineering Handbook* [89]. Design patterns are described in the book by Gamma *et al.* [37].