```
      System.out.println(csStudent);

      // Display the number of remaining hours.
      System.out.println("Hours remaining: " +
                         csStudent.getRemainingHours());
   }
}
```

Program Output

```
Name: Jennifer Haynes
ID Number: 167W98337
Year Admitted: 2004
Major: Computer Science
Math Hours Taken: 12
Computer Science Hours Taken: 20
General Ed Hours Taken: 40
Hours remaining: 48
```

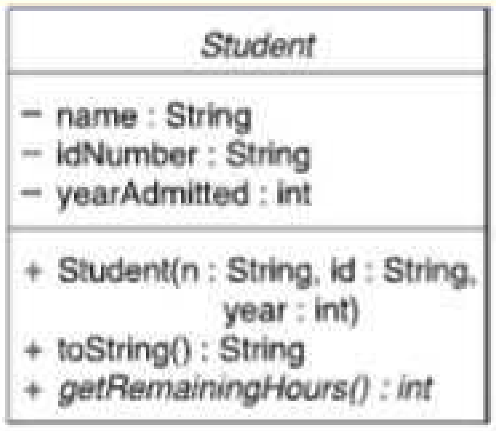
Remember the following points about abstract methods and classes:

- Abstract methods and abstract classes are defined with the `abstract` key word.
- Abstract methods have no body, and their header must end with a semicolon.
- An abstract method must be overridden in a subclass.
- When a class contains an abstract method, it cannot be instantiated. It must serve as a superclass.
- An abstract class cannot be instantiated. It must serve as a superclass.

Abstract Classes in UML

Abstract classes are drawn like regular classes in UML, except the name of the class and the names of abstract methods are shown in italics. For example, Figure 10-19 shows a UML diagram for the `Student` class.

Figure 10-19 UML diagram for the `Student` class



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.19 What is the purpose of an abstract method?

10.20 If a subclass extends a superclass with an abstract method, what must you do in the subclass?

10.21 What is the purpose of an abstract class?

10.22 If a class is defined as abstract, what can you not do with the class?

10.9 Interfaces

CONCEPT: An interface specifies behavior for a class.

In the previous section you learned that an abstract class cannot be instantiated, but is intended to serve as a superclass. You also learned that an abstract method has no body and must be overridden in a subclass. An interface is similar to an abstract class that has all abstract methods. It cannot be instantiated, and all of the methods listed in an interface must be written elsewhere. The purpose of an interface is to specify behavior for a class.

An interface looks similar to a class, except the key word `interface` is used instead of the key word `class`, and the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons. Here is the general format of an interface definition:

```
public interface InterfaceName
{
   (Method headers . . .)
}
```

For example, Code Listing 10-29 shows an interface named `Relatable`, which is intended to be used with the `GradedActivity` class presented earlier. This interface has three method headers: `equals`, `isGreater`, and `isLess`. Notice that each method accepts a `GradedActivity` object as its argument. Also notice that no access specifier is used with the method headers, because all methods specified by an interface are public.

Code Listing 10-29 **(Relatable.java)**

```
/**
   Relatable interface
*/

public interface Relatable
{
   boolean equals(GradedActivity g);
   boolean isGreater(GradedActivity g);
   boolean isLess(GradedActivity g);
}
```

In order for a class to use an interface, it must implement the interface. This is accomplished with the `implements` key word. For example, suppose we have a class named `FinalExam3` that inherits from the `GradedActivity` class and implements the `Relatable` interface. The first line of its definition would look like the following:

```
public class FinalExam3 extends GradedActivity
                           implements Relatable
```

When a class implements an interface, it is agreeing to provide all of the methods that are specified by the interface. It is often said that an interface is like a "contract," and when a class implements an interface it must adhere to the contract.

A class that implements an interface must provide all of the methods that are listed in the interface, with the exact signatures specified and with the same return type. So, in the example previously shown, the `FinalExam3` class must provide an `equals` method, an `isGreater` method, and an `isLess` method, all of which accept a `GradedActivity` object as an argument and return a `boolean` value.

You might have guessed that the `Relatable` interface is named "Relatable" because it specifies methods that presumably, make relational comparisons with `GradedActivity` objects. The intent is to make any class that implements this interface "relatable" with `GradedActivity` objects by ensuring that it has an `equals`, an `isGreater`, and an `isLess` method that perform relational comparisons. But, the interface only specifies the headers for these methods, not what the methods should do. Although the programmer of a class that implements the `Relatable` interface can choose what those methods should do, he or she should provide methods that comply with this intent.

Code Listing 10-30 shows the complete code for the `FinalExam3` class, which implements the `Relatable` interface. The `equals`, `isGreater`, and `isLess` methods compare the calling object with the object passed as an argument. The program in Code Listing 10-31 demonstrates the class.

Code Listing 10-30 (`FinalExam3.java`)

```
/**
   This class determines the grade for a final exam.
*/

public class FinalExam3 extends GradedActivity
                        implements Relatable
{
   private int numQuestions;    // Number of questions
   private double pointsEach;   // Points for each question
   private int numMissed;       // Questions missed

   /**
      The constructor sets the number of questions on the
      exam and the number of questions missed.
      @param questions The number of questions.
      @param missed The number of questions missed.
   */

   public FinalExam3(int questions, int missed)
   {
      double numericScore;       // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the inherited setScore method to
      // set the numeric score.
      setScore(numericScore);
   }

   /**
      The getPointsEach method returns the number of
      points each question is worth.
      @return The value in the pointsEach field.
   */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
      The getNumMissed method returns the number of
      questions missed.
      @return The value in the numMissed field.
   */

   public int getNumMissed()
   {
      return numMissed;
   }

   /**
      The equals method compares the calling object
      to the argument object for equality.
      @return true if the calling
      object's score is equal to the argument's
      score.
   */

   public boolean equals(GradedActivity g)
   {
      boolean status;

      if (this.getScore() == g.getScore())
         status = true;
      else
         status = false;

      return status;
   }

   /**
      The isGreater method determines whether the calling
      object is greater than the argument object.
      @return true if the calling object's score is
      greater than the argument object's score.
   */

   public boolean isGreater(GradedActivity g)
   {
      boolean status;

      if (this.getScore() > g.getScore())
         status = true;
      else
         status = false;

      return status;
   }

   /**
      The isLess method determines whether the calling
      object is less than the argument object.
      @return true if the calling object's score is
      less than the argument object's score.
   */

   public boolean isLess(GradedActivity g)
   {
      boolean status;

      if (this.getScore() < g.getScore())
         status = true;
      else
         status = false;

      return status;
   }
}
```

Code Listing 10-31 `(InterfaceDemo.java)`

```
/**
   This program demonstrates the FinalExam3 class, which
   implements the Relatable interface.
*/

public class InterfaceDemo
{
   public static void main(String[] args)
   {
      // Exam #1 had 100 questions and the student
      // missed 20 questions.
      FinalExam3 exam1 = new FinalExam3(100, 20);

      // Exam #2 had 100 questions and the student
      // missed 30 questions.
      FinalExam3 exam2 = new FinalExam3(100, 30);

      // Display the exam scores.
      System.out.println("Exam 1: " +
                         exam1.getScore());
      System.out.println("Exam 2: " +
                         exam2.getScore());

      // Compare the exam scores.
      if (exam1.equals(exam2))
         System.out.println("The exam scores " +
                            "are equal.");

      if (exam1.isGreater(exam2))
         System.out.println("The Exam 1 score " +
                            "is the highest.");

      if (exam1.isLess(exam2))
         System.out.println("The Exam 1 score " +
                            "is the lowest.");
   }
}
```

Program Output

```
Exam 1: 80.0
Exam 2: 70.0
The Exam 1 score is the highest.
```

Fields in Interfaces

An interface can contain field declarations, but all fields in an interface are treated as `final` and `static`. Because they automatically become `final`, you must provide an initialization value. For example, look at the following interface definition:

```
public interface Doable
{
   int FIELD1 = 1;
   int FIELD2 = 2;
   (Method headers . . .)
}
```

In this interface, `FIELD1` and `FIELD2` are `final static int` variables. Any class that implements this interface has access to these variables.

Implementing Multiple Interfaces

You might be wondering why we need both abstract classes and interfaces, since they are so similar to each other. The reason is that a class can extend only one superclass, but Java allows a class to implement multiple interfaces. When a class implements multiple interfaces, it must provide the methods specified by all of them.

To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the `implements` key word. Here is the first line of an example of a class that implements multiple interfaces:

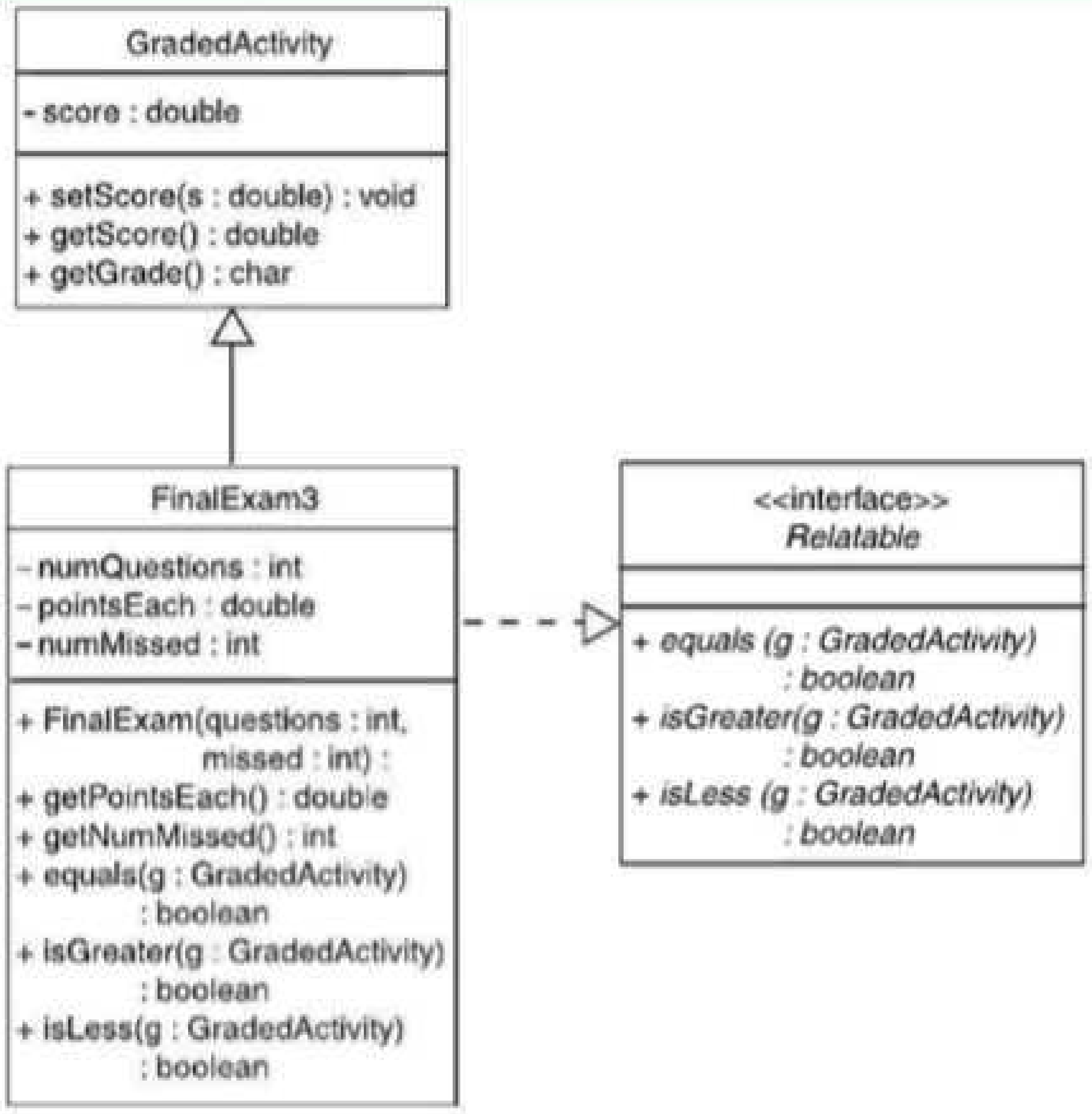
```
public class MyClass implements Interface1,
                                Interface2,
                                Interface3
```

This class implements three interfaces: `Interface1`, `Interface2`, and `Interface3`.

Interfaces in UML

In a UML diagram, an interface is drawn like a class, except the interface name and the method names are italicized, and the <<interface>> tag is shown above the interface name. The relationship between a class and an interface is known as a realization relationship (the class realizes the interfaces). You show a realization relationship in a UML diagram by connecting a class and an interface with a dashed line that has an open arrowhead at one end. The arrowhead points to the interface. This depicts the realization relationship. Figure 10-20 is a UML diagram showing the relationships among the `GradedActivity` class, the `FinalExam3` class, and the `Relatable` interface.

Figure 10-20 Realization relationship in a UML diagram


GradedActivity
- score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
FinalExam3
- numQuestions : int
- pointsEach : double
- numMissed : int
+ FinalExam(questions : int, missed : int) :
+ getPointsEach() : double
+ getNumMissed() : int
+ equals(g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess(g : GradedActivity) : boolean
<<interface>>
Relatable
+ equals (g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess (g : GradedActivity) : boolean

Polymorphism and Interfaces

Just as you can create reference variables of a class type, Java allows you to create reference variables of an interface type. An interface reference variable can reference any object that implements that interface, regardless of its class type. This is another example of polymorphism. For example, look at the `RetailItem` interface in Code Listing 10-32.

Code Listing 10-32 (`RetailItem.java`)

```
/**
   RetailItem interface
*/

public interface RetailItem
{
   public double getRetailPrice();
}
```

This interface specifies only one method: `getRetailPrice`. Both the `CompactDisc` and `DvdMovie` classes, shown in Code Listings 10-33 and 10-34, implement this interface.

Code Listing 10-33 (`CompactDisc.java`)

```java
/**
   Compact Disc class
*/

public class CompactDisc implements RetailItem
{
   private String title;          // The CD's title
   private String artist;         // The CD's artist
   private double retailPrice;    // The CD's retail price

   /**
      Constructor
      @param cdTitle The CD title.
      @param cdArtist The name of the artist.
      @param cdPrice The CD's price.
   */

   public CompactDisc(String cdTitle, String cdArtist,
                      double cdPrice)
   {
      title = cdTitle;
      artist = cdArtist;
      retailPrice = cdPrice;
   }

   /**
      getTitle method
      @return The CD's title.
   */

   public String getTitle()
   {
      return title;
   }

   /**
      getArtist method
      @return The name of the artist.
   */

   public String getArtist()
   {
         return artist;
      }

      /**
         getRetailPrice method (Required by the RetailItem
         interface)
         @return The retail price of the CD.
      */

      public double getRetailPrice()
      {
         return retailPrice;
      }
   }
```

Code Listing 10-34 (`DvdMovie.java`)

```
/**
   DvdMovie class
*/

public class DvdMovie implements RetailItem
{
   private String title;          // The DVD's title
   private int runningTime;       // Running time in minutes
   private double retailPrice;    // The DVD's retail price

   /**
      Constructor
      @param dvdTitle The DVD title.
      @param runTime The running time in minutes.
      @param dvdPrice The DVD's price.
   */

   public DvdMovie(String dvdTitle, int runTime,
                                 double dvdPrice)
   {
      title = dvdTitle;
      runningTime = runTime;
      retailPrice = dvdPrice;
   }

   /**
      getTitle method
      @return The DVD's title.
   */

   public String getTitle()
   {
      return title;
   }

   /**
      getRunningTime method
      @return The running time in minutes.
   */

   public int getRunningTime()
   {
      return runningTime;
   }

   /**
      getRetailPrice method (Required by the RetailItem
      interface)
      @return The retail price of the DVD.
   */

   public double getRetailPrice()
   {
      return retailPrice;
   }
}
```

Because they implement the `RetailItem` interface, objects of these classes may be referenced by a `RetailItem` reference variable. The following code demonstrates:

```
RetailItem item1 = new CompactDisc("Songs From the Heart",
                                   "Billy Nelson",
                                   18.95);
RetailItem item2 = new DvdMovie("Planet X",
                                102,
                                22.95);
```

In this code, two `RetailItem` reference variables, `item1` and `item2`, are declared. The `item1` variable references a `CompactDisc` object and the `item2` variable references a `DvdMovie` object. This is possible because both the `CompactDisc` and `DvdMovie` classes implement the `RetailItem` interface. When a class implements an interface, an inheritance relationship known as interface inheritance is established. Because of this inheritance relationship, a `CompactDisc` object is a `RetailItem`, and likewise, a `DvdMovie` object is a `RetailItem`. Therefore, we can create `RetailItem` reference variables and have them reference `CompactDisc` and `DvdMovie` objects.

The program in Code Listing 10-35 demonstrates how an interface reference variable can be used as a method parameter.

Code Listing 10-35 (PolymorphicInterfaceDemo.java)

```
/**
   This program demonstrates that an interface type may
   be used to create a polymorphic reference.
*/

public class PolymorphicInterfaceDemo
{
   public static void main(String[] args)
   {
      // Create a CompactDisc object.
      CompactDisc cd =
              new CompactDisc("Greatest Hits",
                              "Joe Looney Band",
                               18.95);
      // Create a DvdMovie object.
      DvdMovie movie =
              new DvdMovie("Wheels of Fury",
                             137, 12.95);

      // Display the CD's title.
      System.out.println("Item #1: " +
                         cd.getTitle());

      // Display the CD's price.
      showPrice(cd);

      // Display the DVD's title.
      System.out.println("Item #2: " +
                         movie.getTitle());

      // Display the DVD's price.
      showPrice(movie);
   }

   /**
      The showPrice method displays the price
      of a RetailItem object.
      @param item A reference to a RetailItem object.
   */

   private static void showPrice(RetailItem item)
   {
      System.out.printf("Price: $%,.2f\n", item.getRetailPrice());
   }
}
```

Program Output

```
Item #1: Greatest Hits
Price: $18.95
Item #2: Wheels of Fury
Price: $12.95
```

There are some limitations to using interface reference variables. As previously mentioned, you cannot create an instance of an interface. In addition, when an interface variable references an object, you can use the interface variable to call only the methods that are specified in the interface. For example, look at the following code:

```
// Reference a CompactDisc object with a RetailItem variable.
RetailItem item = new CompactDisc("Greatest Hits",
                                  "Joe Looney Band",
                                  18.95);

// Call the getRetailPrice method . . .
System.out.println(item.getRetailPrice()); // OK, this works.
// Attempt to call the getTitle method . . .
System.out.println(item.getTitle()); // ERROR! Will not compile!
```

The last line of code will not compile because the `RetailItem` interface specifies only one method: `getRetailPrice`. So, we cannot use a `RetailItem` reference variable to call any other method.

TIP: It is possible to cast an interface reference variable to the type of the object it references, and then call methods that are members of that type. The syntax is somewhat awkward, however. The statement that causes the compiler error in the example code could be rewritten as:

```
System.out.println(((CompactDisc)item).getTitle());
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

10.23 What is the purpose of an interface?

10.24 How is an interface similar to an abstract class?

10.25 How is an interface different from an abstract class, or any class?

10.26 If an interface has fields, how are they treated?

10.27 Write the first line of a class named `Customer`, which implements an interface named `Relatable`.

10.28 Write the first line of a class named `Employee`, which implements interfaces named `Payable` and `Listable`.

10.10 Common Errors to Avoid

The following list describes several errors that are commonly committed when learning this chapter's topics:

- **Attempting to access a private superclass member directly from a subclass.** Private superclass members cannot be directly accessed by a method in a subclass. The subclass must call a public or protected superclass method in order to access the superclass's private members.
- **Forgetting to call a superclass constructor explicitly when the superclass does not have a default constructor or a programmer-defined no-arg constructor.** When a superclass does not have a default constructor or a programmer-defined no-arg constructor, the subclass's constructor must explicitly call one of the constructors that the superclass does have.
- **Allowing the superclass's no-arg constructor to be implicitly called when you intend to call another superclass constructor.** If a subclass's constructor does not explicitly call a superclass constructor, Java automatically calls the superclass's no-arg constructor.
- **Forgetting to precede a call to an overridden superclass method with `super`.** When a subclass method calls an overridden superclass method, it must precede the method call with the key word `super` and a dot (`.`). Failing to do so results in the subclass's version of the method being called.
- **Forgetting a class member's access specifier.** When you do not give a class member an access specifier, it is granted package access by default. This means that any method in the same package may access the member.
- **Writing a body for an abstract method.** An abstract method cannot have a body. It must be overridden in a subclass.
- **Forgetting to terminate an abstract method's header with a semicolon.** An abstract method header does not have a body, and it must be terminated with a semicolon.
- **Failing to override an abstract method.** An abstract method must be overridden in a subclass.
- **Overloading an abstract method instead of overriding it.** Overloading is not the same as overriding. When a superclass has an abstract method, the subclass must have a method with the same signature as the abstract method.
- **Trying to instantiate an abstract class.** You cannot create an instance of an abstract class.
- **Implementing an interface but forgetting to provide all of the methods specified by the interface.** When a class implements an interface, all of the methods specified by the interface must be provided in the class.
- **Writing a method specified by an interface but failing to use the exact signature and return type.** When a class implements an interface, the class must have methods with the same signature and return type as the methods specified in the interface.

Review Questions and Exercises

Multiple Choice and True/False

1. In an inheritance relationship, this is the general class.
   a. subclass
   b. superclass
   c. slave class
   d. child class
2. In an inheritance relationship, this is the specialized class.
   a. superclass
   b. master class
   c. subclass
   d. parent class
3. This key word indicates that a class inherits from another class.
   a. `derived`
   b. `specialized`
   c. `based`
   d. `extends`
4. A subclass does not have access to these superclass members.
   a. public
   b. private
   c. protected
   d. all of these
5. This key word refers to an object's superclass.
   a. `super`
   b. `base`
   c. `superclass`
   d. `this`
6. In a subclass constructor, a call to the superclass constructor must __________.
   a. appear as the very first statement
   b. appear as the very last statement
   c. appear between the constructor's header and the opening brace
   d. not appear
7. The following is an explicit call to the superclass's default constructor.
   a. `default();`
   b. `class();`
   c. `super();`
   d. `base();`
8. A method in a subclass that has the same signature as a method in the superclass is an example of __________.
   a. overloading
   b. overriding

c. composition
d. an error

9. A method in a subclass having the same name as a method in the superclass but a different signature is an example of __________.
   a. overloading
   b. overriding
   c. composition
   d. an error
10. These superclass members are accessible to subclasses and classes in the same package.
    a. private
    b. public
    c. protected
    d. all of these
11. All classes directly or indirectly inherit from this class.
    a. `Object`
    b. `Super`
    c. `Root`
    d. `Java`
12. With this type of binding, the Java Virtual Machine determines at runtime which method to call, depending on the type of the object that a variable references.
    a. static
    b. early
    c. flexible
    d. dynamic
13. This operator can be used to determine whether a reference variable references an object of a particular class.
    a. `isclass`
    b. `typeof`
    c. `instanceof`
    d. `isinstance`
14. When a class implements an interface, it must __________.
    a. overload all of the methods listed in the interface
    b. provide all of the methods that are listed in the interface, with the exact signatures and return types specified
    c. not have a constructor
    d. be an abstract class
15. Fields in an interface are __________.
    a. `final`
    b. `static`
    c. both `final` and `static`
    d. not allowed

16. Abstract methods must be _________.
    a. overridden
    b. overloaded
    c. deleted and replaced with real methods
    d. declared as private
17. Abstract classes cannot _________.
    a. be used as superclasses
    b. have abstract methods
    c. be instantiated
    d. have fields
18. **True or False:** Constructors are not inherited.
19. **True or False:** In a subclass, a call to the superclass constructor can only be written in the subclass constructor.
20. **True or False:** If a subclass constructor does not explicitly call a superclass constructor, Java will not call any of the superclass's constructors.
21. **True or False:** An object of a superclass can access members declared in a subclass.
22. **True or False:** The superclass constructor always executes before the subclass constructor.
23. **True or False:** When a method is declared with the `final` modifier, it must be overridden in a subclass.
24. **True or False:** A superclass has a member with package access. A class that is outside the superclass's package but inherits from the superclass can access the member.
25. **True or False:** A superclass reference variable can reference an object of a subclass that extends the superclass.
26. **True or False:** A subclass reference variable can reference an object of the superclass.
27. **True or False:** When a class contains an abstract method, the class cannot be instantiated.
28. **True or False:** A class may only implement one interface.
29. **True or False:** By default all members of an interface are public.

Find the Error

Find the error in each of the following code segments:

1.
```
// Superclass
public class Vehicle
{
   (Member declarations . . .)
}
// Subclass
```

```
   public class Car expands Vehicle
   {
      (Member declarations . . .)
   }
```

2.
```
   // Superclass
   public class Vehicle
   {
      private double cost;
      (Other methods . . .)
   }
   // Subclass
   public class Car extends Vehicle
   {
      public Car(double c)
      {
          cost = c;
      }
   }
```

3.
```
   // Superclass
   public class Vehicle
   {
      private double cost;
      public Vehicle(double c)
      {
          cost = c;
      }
      (Other methods . . .)
   }
   // Subclass
   public class Car extends Vehicle
   {
      private int passengers;
      public Car(int p)
      {
          passengers = c;
      }
      (Other methods . . .)
   }
```

4.
```
   // Superclass
   public class Vehicle
   {
      public abstract double getMilesPerGallon();
      (Other methods . . .)
   }
```

```
// Subclass
public class Car extends Vehicle
{
   private int mpg;
   public int getMilesPerGallon();
   {
      return mpg;
   }
   (Other methods . . .)
}
```

Algorithm Workbench

1. Write the first line of the definition for a `Poodle` class. The class should extend the `Dog` class.
2. Look at the following code, which is the first line of a class definition:

   ```
   public class Tiger extends Felis
   ```

   In what order will the class constructors execute?
3. Write the declaration for class `B`. The class's members should be as follows:
   - `m`, an integer. This variable should not be accessible to code outside the class or to any class that extends class `B`.
   - `n`, an integer. This variable should be accessible only to classes that extend class `B` or are in the same package as class `B`.
   - `setM`, `getM`, `setN`, and `getN`. These are the mutator and accessor methods for the member variables `m` and `n`. These methods should be accessible to code outside the class.
   - `calc`. This is a public abstract method.

   Next, write the declaration for class `D`, which extends class `B`. The class's members should be as follows:
   - `q`, a `double`. This variable should not be accessible to code outside the class.
   - `r`, a `double`. This variable should be accessible to any class that extends class `D` or is in the same package.
   - `setQ`, `getQ`, `setR`, and `getR`. These are the mutator and accessor methods for the member variables `q` and `r`. These methods should be accessible to code outside the class.
   - `calc`, a public method that overrides the superclass's abstract `calc` method. This method should return the value of `q` times `r`.
4. Write the statement that calls a superclass constructor and passes the arguments `x`, `y`, and `z`.
5. A superclass has the following method:

   ```
   public void setValue(int v)
   {
       value = v;
   }
   ```

Write a statement that may appear in a subclass that calls this method, passing 10 as an argument.

6. A superclass has the following abstract method:

```
public abstract int getValue();
```

Write an example of a `getValue` method that can appear in a subclass.

7. Write the first line of the definition for a `Stereo` class. The class should extend the `SoundSystem` class, and it should implement the `CDplayable`, `TunerPlayable`, and `CassettePlayable` interfaces.

8. Write an interface named `Nameable` that specifies the following methods:

```
public void setName(String n)
public String getName()
```

Short Answer

1. What is an "is-a" relationship?
2. A program uses two classes: `Animal` and `Dog`. Which class is the superclass and which is the subclass?
3. What is the superclass and what is the subclass in the following line?

```
public class Pet extends Dog
```

4. What is the difference between a protected class member and a private class member?
5. Can a subclass ever directly access the private members of its superclass?
6. Which constructor is called first, that of the subclass or the superclass?
7. What is the difference between overriding a superclass method and overloading a superclass method?
8. Reference variables can be polymorphic. What does this mean?
9. When does dynamic binding take place?
10. What is an abstract method?
11. What is an abstract class?
12. What are the differences between an abstract class and an interface?

Programming Challenges

MyProgrammingLab™ *Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.*

1. `Employee` and `ProductionWorker` Classes

VideoNote
The `Employee` and `ProductionWorker` Classes Problem

Design a class named `Employee`. The class should keep the following information in fields:

- Employee name
- Employee number in the format XXX–L, where each X is a digit within the range 0–9 and the L is a letter within the range A–M.
- Hire date

Write one or more constructors and the appropriate accessor and mutator methods for the class.

Next, write a class named `ProductionWorker` that extends the `Employee` class. The `ProductionWorker` class should have fields to hold the following information:

- Shift (an integer)
- Hourly pay rate (a `double`)

The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

2. `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that extends the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a field that holds the annual salary and a field that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. `TeamLeader` Class

In a particular factory, a team leader is an hourly paid production worker that leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that extends the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have fields for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

4. `Essay` Class

Design an `Essay` class that extends the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives for an essay. The student's essay score can be up to 100 and is determined in the following manner:

Grammar: 30 points
Spelling: 20 points
Correct length: 20 points
Content: 30 points

Demonstrate the class in a simple program.

5. Course Grades

In a course, a teacher gives the following tests and assignments:

- A **lab activity** that is observed by the teacher and assigned a numeric score.
- A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
- An **essay** that is assigned a numeric score.
- A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have a `GradedActivity` array named `grades` as a field. The array should have four elements, one for each of the assignments previously described. The class should have the following methods:

`setLab`:	This method should accept a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` field should reference this object.
`setPassFailExam`:	This method should accept a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` field should reference this object.
`setEssay`:	This method should accept an `Essay` object as its argument. (See Programming Challenge 4 for the `Essay` class. If you have not completed Programming Challenge 4, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` field should reference this object.
`setFinalExam`:	This method should accept a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` field should reference this object.
`toString`:	This method should return a string that contains the numeric scores and grades for each element in the `grades` array.

Demonstrate the class in a program.

6. `Analyzable` Interface

Modify the `CourseGrades` class you created in Programming Challenge 5 so it implements the following interface:

```
public interface Analyzable
{
   double getAverage();
   GradedActivity getHighest();
   GradedActivity getLowest();
}
```

The `getAverage` method should return the average of the numeric scores stored in the `grades` array. The `getHighest` method should return a reference to the element of the grades array that has the highest numeric score. The `getLowest` method should return a reference to the element of the grades array that has the lowest numeric score. Demonstrate the new methods in a complete program.

7. `Person` and `Customer` Classes

Design a class named `Person` with fields for holding a person's name, address, and telephone number. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields.

Next, design a class named `Customer`, which extends the `Person` class. The `Customer` class should have a field for a customer number and a `boolean` field indicating whether the customer wishes to be on a mailing list. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate an object of the `Customer` class in a simple program.

8. `PreferredCustomer` Class

A retail store has a preferred customer plan where customers can earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store as follows:

- When a preferred customer spends $500, he or she gets a 5 percent discount on all future purchases.
- When a preferred customer spends $1,000, he or she gets a 6 percent discount on all future purchases.
- When a preferred customer spends $1,500, he or she gets a 7 percent discount on all future purchases.
- When a preferred customer spends $2,000 or more, he or she gets a 10 percent discount on all future purchases.

Design a class named `PreferredCustomer`, which extends the `Customer` class you created in Programming Challenge 7. The `PreferredCustomer` class should have fields for the amount of the customer's purchases and the customer's discount level. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate the class in a simple program.

9. `BankAccount` and `SavingsAccount` Classes

Design an abstract class named `BankAccount` to hold the following data for a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following methods:

Constructor:	The constructor should accept arguments for the balance and annual interest rate.

`deposit:`	A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits.
`withdraw:`	A method that accepts an argument for the amount of the withdrawal. The method should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.
`calcInterest:`	A method that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas: *Monthly Interest Rate* = (*Annual Interest Rate* / 12) *Monthly Interest* = *Balance* * *Monthly Interest Rate* *Balance* = *Balance* + *Monthly Interest*
`monthlyProcess:`	A method that subtracts the monthly service charges from the balance, calls the `calcInterest` method, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a `SavingsAccount` class that extends the `BankAccount` class. The `SavingsAccount` class should have a status field to represent an active or inactive account. If the balance of a savings account falls below $25, it becomes inactive. (The `status` field could be a `boolean` variable.) No more withdrawals may be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following methods:

`withdraw:`	A method that determines whether the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the superclass version of the method.
`deposit:`	A method that determines whether the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. A deposit is then made by calling the superclass version of the method.
`monthlyProcess:`	Before the superclass method is called, this method checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the superclass field that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.)

10. `Ship`, `CruiseShip`, and `CargoShip` Classes

Design a `Ship` class that the following members:

- A field for the name of the ship (a string).
- A field for the year that the ship was built (a string).

- A constructor and appropriate accessors and mutators.
- A `toString` method that displays the ship's name and the year it was built.

Design a `CruiseShip` class that extends the `Ship` class. The `CruiseShip` class should have the following members:

- A field for the maximum number of passengers (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following members:

- A field for the cargo capacity in tonnage (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has a `Ship` array. Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements. The program should then step through the array, calling each object's `toString` method. (See Code Listing 10-25 as an example.)

CHAPTER 11

Exceptions and Advanced File I/O

TOPICS

11.1 Handling Exceptions

CONCEPT: **An exception is an object that is generated as the result of an error or an unexpected event. To prevent exceptions from crashing your program, you must write code that detects and handles them.**

VideoNote Handling Exceptions

There are many error conditions that can occur while a Java application is running that will cause it to halt execution. By now you have probably experienced this many times. For example, look at the program in Code Listing 11-1. This program attempts to read beyond the bounds of an array.

Code Listing 11-1 (BadArray.java)

```
/**
   This program causes an error and crashes.
*/

public class BadArray
{
   public static void main(String[] args)
   {
      // Create an array with 3 elements.
      int[] numbers = { 1, 2, 3 };

      // Attempt to read beyond the bounds
      // of the array.
      for (int i = 0; i <= 3; i++)
         System.out.println(numbers[i]);
   }
}
```

Program Output

```
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at BadArray.main(BadArray.java:15)
```

The `numbers` array in this program has only three elements, with the subscripts 0 though 2. The program crashes when it tries to read the element at `numbers[3]`, and displays an error message similar to that shown at the end of the program output. This message indicates that an exception occurred, and it gives some information about it. An *exception* is an object that is generated in memory as the result of an error or an unexpected event. When an exception is generated, it is said to have been "thrown." Unless an exception is detected by the application and dealt with, it causes the application to halt.

To detect that an exception has been thrown and prevent it from halting your application, Java allows you to create exception handlers. An *exception handler* is a section of code that gracefully responds to exceptions when they are thrown. The process of intercepting and responding to exceptions is called *exception handling*. If your code does not handle an exception when it is thrown, the *default exception handler* deals with it, as shown in Code Listing 11-1. The default exception handler prints an error message and crashes the program.

The error that caused the exception to be thrown in Code Listing 11-1 is easy to avoid. If the loop were written properly, it would not have tried to read outside the bounds of the array. Some errors, however, are caused by conditions that are outside the application and cannot be avoided. For example, suppose an application creates a file on the disk and the user deletes it. Later the application attempts to open the file to read from it, and because it does not exist, an error occurs. As a result, an exception is thrown.

Exception Classes

As previously mentioned, an exception is an object. Exception objects are created from classes in the Java API. The API has an extensive hierarchy of exception classes. A small part of the hierarchy is shown in Figure 11-1.

As you can see, all of the classes in the hierarchy inherit from the `Throwable` class. Just below the `Throwable` class are the classes `Error` and `Exception`. Classes that inherit from

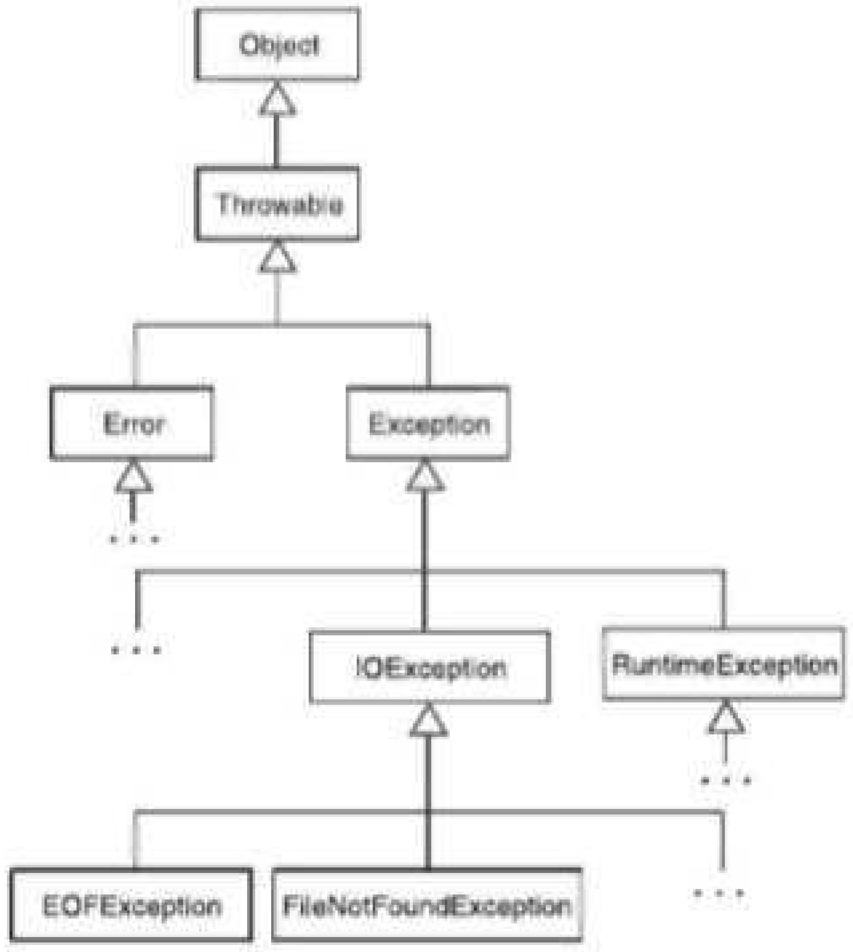
`Error` are for exceptions that are thrown when a critical error occurs, such as an internal error in the Java Virtual Machine or running out of memory. Your applications should not try to handle these errors because they are the result of a serious condition.

All of the exceptions that you will handle are instances of classes that inherit from `Exception`. Figure 11-1 shows two of these classes: `IOException` and `RuntimeException`. These classes also serve as superclasses. `IOException` serves as a superclass for exceptions that are related to input and output operations. `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript.

The chart in Figure 11-1 shows two of the classes that inherit from the `IOException` class: `EOFException` and `FileNotFoundException`. These are examples of classes that exception objects are created from. An `EOFException` object is thrown when an application attempts to read beyond the end of a file, and a `FileNotFoundException` object is thrown when an application tries to open a file that does not exist.

NOTE: The exception classes are in packages in the Java API. For example, `FileNotFoundException` is in the `java.io` package. When you handle an exception that is not in the `java.lang` package, you will need the appropriate `import` statement.

Figure 11-1 Part of the exception class hierarchy



Handling an Exception

To handle an exception, you use a `try` statement. We will look at several variations of the `try` statement, beginning with the following general format:

```
try
{
   (try block statements . . .)
}
catch (ExceptionType parameterName)
{
   (catch block statements . . .)
}
```

First the key word `try` appears. Next, a block of code appears inside braces, which are required. This block of code is known as a *try block*. A *try block* is one or more statements that are executed and can potentially throw an exception. You can think of the code in the try block as being "protected" because the application will not halt if the try block throws an exception.

After the try block, a `catch` clause appears. A `catch` clause begins with the key word `catch`, followed by the code *`(ExceptionType parameterName)`*. This is a parameter variable declaration, where *`ExceptionType`* is the name of an exception class and *`parameterName`* is a variable name. If code in the try block throws an exception of the *`ExceptionType`* class, then the parameter variable will reference the exception object. In addition, the code that immediately follows the `catch` clause is executed. The code that immediately follows the `catch` clause is known as a *`catch block`*. Once again, the braces are required.

Let's look at an example of code that uses a `try` statement. The statement inside the following try block attempts to open the file *MyFile.txt*. If the file does not exist, the `Scanner` object throws an exception of the `FileNotFoundException` class. This code is designed to handle that exception if it is thrown.

```
try
{
   File file = new File("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

Let's look closer. First, the code in the try block is executed. If this code throws an exception, the Java Virtual Machine searches for a `catch` clause that can deal with the exception. In order for a `catch` clause to be able to deal with an exception, its parameter must be of a type that is compatible with the exception's type. Here is this code's `catch` clause:

```
catch (FileNotFoundException e)
```

This `catch` clause declares a reference variable named `e` as its parameter. The `e` variable can reference an object of the `FileNotFoundException` class. So, this `catch` clause can deal with an exception of the `FileNotFoundException` class. If the code in the try block throws an exception of the `FileNotFoundException` class, the `e` variable will reference the exception object and the code in the catch block will execute. In this case, the message "File not found." will be printed. After the catch block is executed, the program will resume with the code that appears after the entire `try/catch` construct.

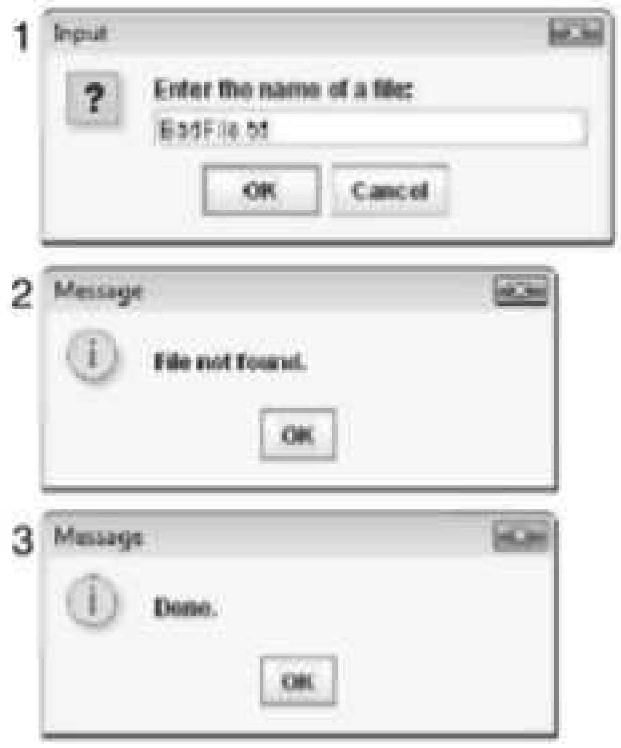
NOTE: The Java API documentation lists all of the exceptions that can be thrown from each method.

Code Listing 11-2 shows a program that asks the user to enter a file name, then attempts to open the file. If the file does not exist, an error message is printed. Figures 11-2 and 11-3 show examples of interaction with the program.

Code Listing 11-2 `(OpenFile.java)`

```java
import java.io.*;    // For File class and FileNotFoundException
import java.util.Scanner;          // For the Scanner class
import javax.swing.JOptionPane;    // For the JOptionPane class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class OpenFile
{
   public static void main(String[] args)
   {
      File file;                   // For file input
      Scanner inputFile;           // For file input
      String fileName;             // To hold a file name

      // Get a file name from the user.
      fileName = JOptionPane.showInputDialog("Enter " +
                           "the name of a file:");

      // Attempt to open the file.
      try
      {
         file = new File(fileName);
         inputFile = new Scanner(file);
         JOptionPane.showMessageDialog(null,
                           "The file was found.");
      }
      catch (FileNotFoundException e)
      {
         JOptionPane.showMessageDialog(null,
                           "File not found.");
      }

      JOptionPane.showMessageDialog(null, "Done.");
      System.exit(0);
   }
}
```

Figure 11-2 Interaction with the `OpenFile.java` program (assume that *BadFile.txt* does not exist)

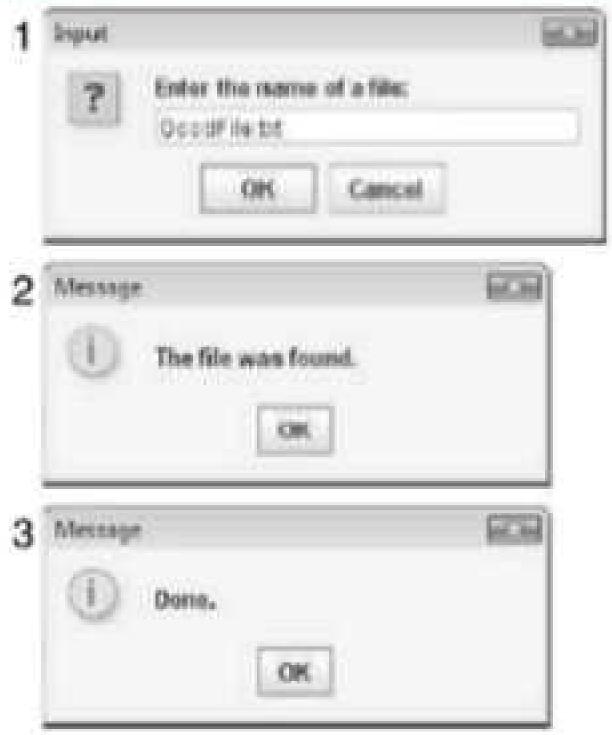


Enter the name of a file:
BadFile.txt

File not found.

Done.

Figure 11-3 Interaction with the `OpenFile.java` program (assume that *GoodFile.txt* does exist)



Enter the name of a file:
GoodFile.txt

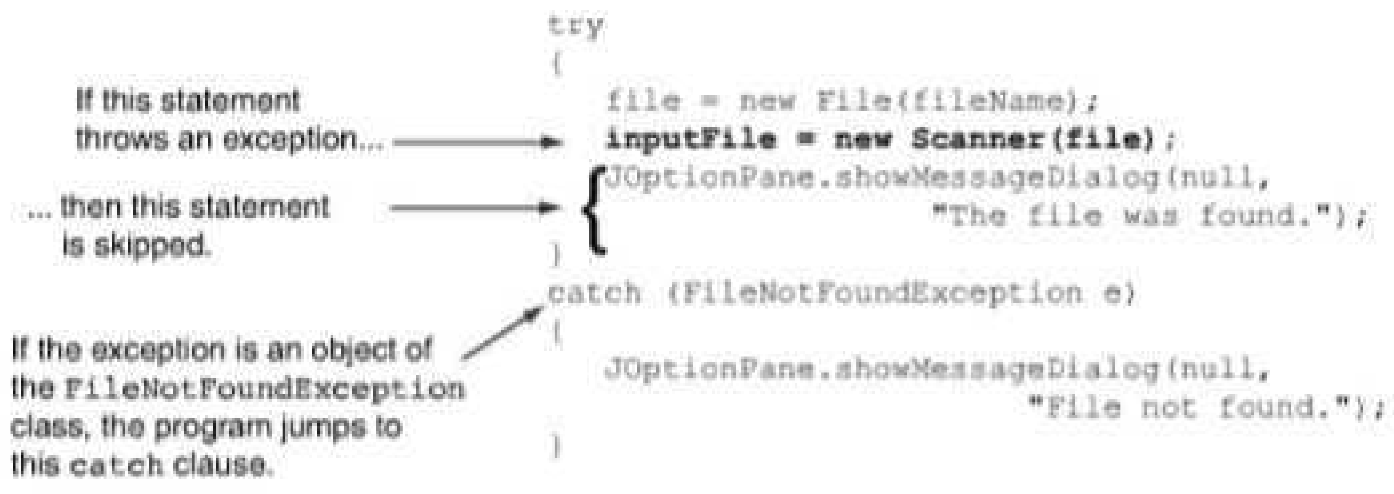
The file was found.

Done.

Look at the example run of the program in Figure 11-2. The user entered *BadFile.txt* as the file name. In line 25, the first statement inside the try block, a `File` object is created and this name is passed to the `File` constructor. In line 26 a reference to the `File` object is passed to the `Scanner` constructor. Because *BadFile.txt* does not exist, an exception of the `FileNotFoundException` class is thrown by the `Scanner` class constructor. When the exception is thrown, the program immediately exits the try block, skipping the remaining statement in the block (lines 27 through 28). The program jumps to the `catch` clause in line 30, which has a `FileNotFoundException` parameter, and executes the catch block that follows it. Figure 11-4 illustrates this sequence of events.

Notice that after the catch block executes, the program resumes at the statement that immediately follows the `try/catch` construct. This statement, which is in line 36, displays the message "Done."

Figure 11-4 Sequence of events with an exception

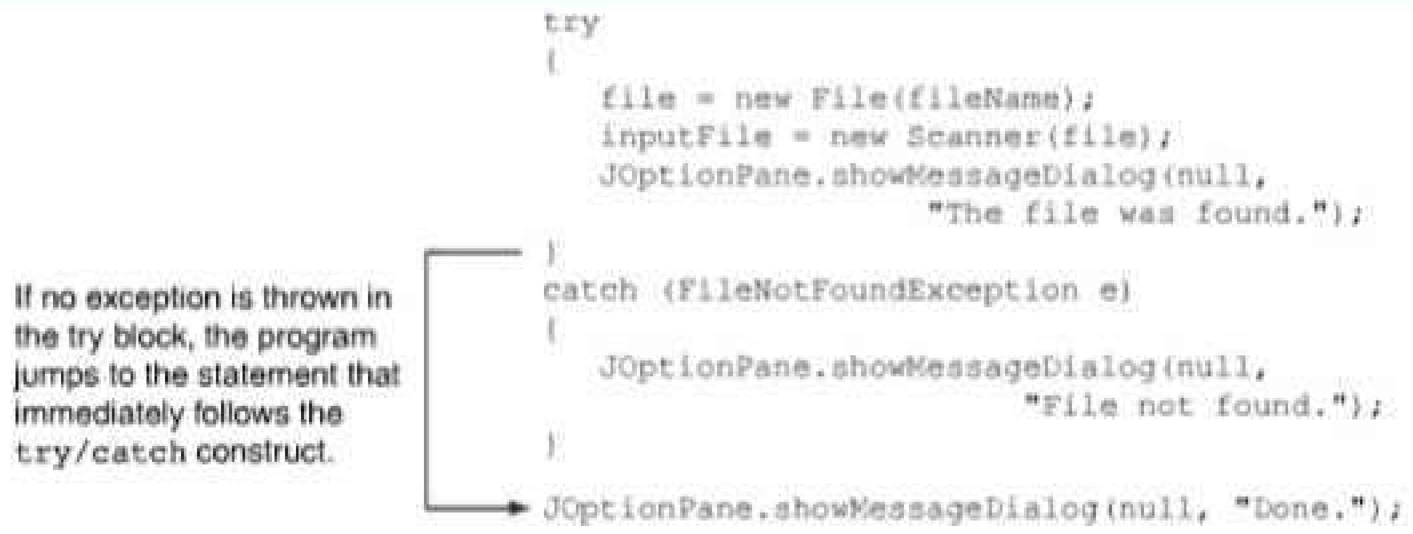


```
try
{
   file = new File(fileName);
   inputFile = new Scanner(file);
   JOptionPane.showMessageDialog(null,
                        "The file was found.");
}
catch (FileNotFoundException e)
{
   JOptionPane.showMessageDialog(null,
                             "File not found.");
}
```

If this statement throws an exception...
... then this statement is skipped.
If the exception is an object of the FileNotFoundException class, the program jumps to this catch clause.

Now look at the example run of the program in Figure 11-3. In this case, the user entered *GoodFile.txt*, which is the name of a file that exists. No exception was thrown in the try block, so the program skips the `catch` clause and its catch block and jumps directly to the statement in line 36, which follows the `try/catch` construct. This statement displays the message "Done." Figure 11-5 illustrates this sequence of events.

Figure 11-5 Sequence of events with no exception



```
try
{
   file = new File(fileName);
   inputFile = new Scanner(file);
   JOptionPane.showMessageDialog(null,
                        "The file was found.");
}
catch (FileNotFoundException e)
{
   JOptionPane.showMessageDialog(null,
                             "File not found.");
}
JOptionPane.showMessageDialog(null, "Done.");
```

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

Retrieving the Default Error Message

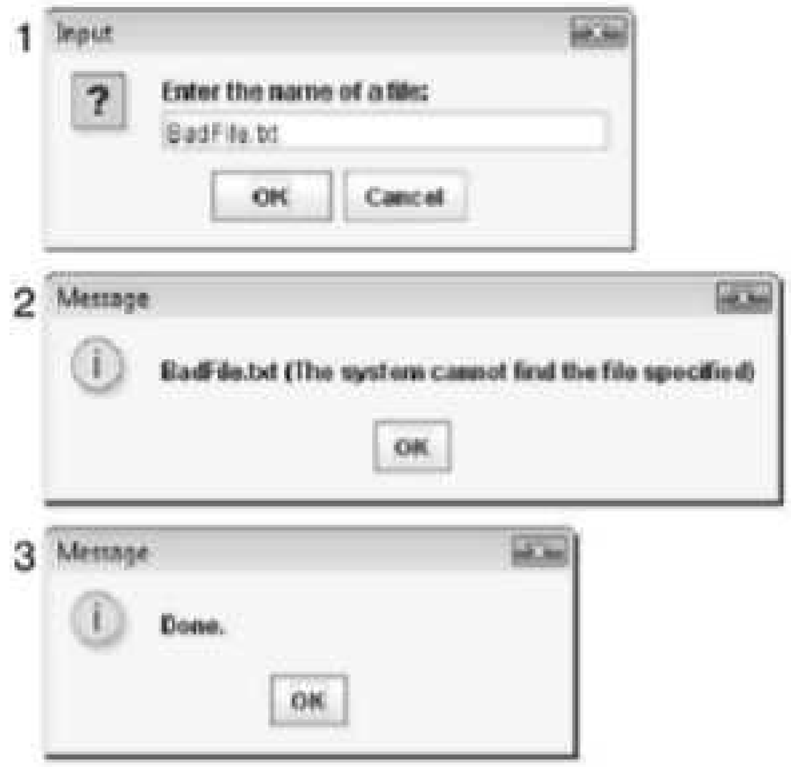
Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception. This is the same message that is displayed when the exception is not handled and the application halts. The program in Code Listing 11-3 demonstrates the `getMessage` method. This is a modified version of the program in Code Listing 11-2. Figure 11-6 shows the program running. In the figure, the user entered the name of a file that does not exist.

Code Listing 11-3 `(ExceptionMessage.java)`

```
import java.io.*;                 // For file I/O classes
import java.util.Scanner;         // For the Scanner class
import javax.swing.JOptionPane;   // For the JOptionPane class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage
{
   public static void main(String[] args)
   {
      File file;                  // For file input
      Scanner inputFile;          // For file input
      String fileName;            // To hold a file name

      // Get a file name from the user.
      fileName = JOptionPane.showInputDialog("Enter " +
                                  "the name of a file:");

      // Attempt to open the file.
      try
      {
         file = new File(fileName);
         inputFile = new Scanner(file);
         JOptionPane.showMessageDialog(null,
                             "The file was found.");
      }
      catch (FileNotFoundException e)
      {
         JOptionPane.showMessageDialog(null, e.getMessage());
      }

      JOptionPane.showMessageDialog(null, "Done.");
      System.exit(0);
   }
}
```

Code Listing 11-4 shows another example. This program forces the `parseInt` method of the `Integer` wrapper class to throw an exception.

Figure 11-6 Interaction with the `ExceptionMessage.java` program (assume that *BadFile.txt* does not exist)



Enter the name of a file:
BadFile.txt

BadFile.txt (The system cannot find the file specified)

Done.

Code Listing 11-4 `(ParseIntError.java)`

```
/**
   This program demonstrates how the Integer.parseInt
   method throws an exception.
*/

public class ParseIntError
{
   public static void main(String[] args)
   {
       String str = "abcde";
       int number;

       try
       {
          number = Integer.parseInt(str);
       }
       catch (NumberFormatException e)
       {
          System.out.println("Conversion error: " +
                             e.getMessage());
       }
   }
}
```

Program Output

```
Conversion error: For input string: "abcde"
```

The numeric wrapper classes' "parse" methods all throw an exception of the `NumberFormatException` type if the string being converted does not contain a convertible numeric value.

Polymorphic References to Exceptions

Recall from Chapter 10 that a reference variable of a superclass type can reference subclass objects. This is called polymorphism. When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause. For example, all of the exceptions that we have dealt with inherit from the `Exception` class. So, a `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that inherits from the `Exception` class. For example, the `try` statement in Code Listing 11-4 could be written as follows:

```
try
{
   number = Integer.parseInt(str);
}
catch (Exception e)
{
   System.out.println("Conversion error: " +
                      e.getMessage());
}
```

Although the `Integer` class's `parseInt` method throws a `NumberFormatException` object, this code still works because the `NumberFormatException` class inherits from the `Exception` class.

Using Multiple `catch` Clauses to Handle Multiple Exceptions

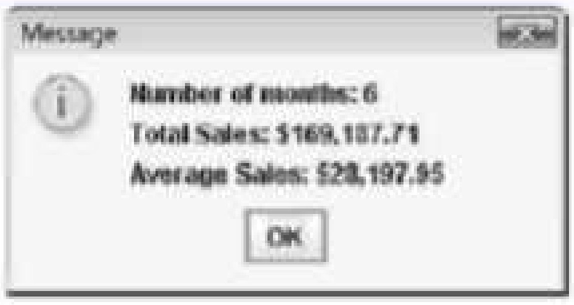
The programs we have studied so far test only for a single type of exception. In many cases, however, the code in the try block will be capable of throwing more than one type of exception. In such a case, you need to write a `catch` clause for each type of exception that could potentially be thrown.

For example, the program in Code Listing 11-5 reads the contents of a file named *SalesData.txt*. Each line in the file contains the sales amount for one month, and the file has several lines. Here are the contents of the file:

```
24987.62
26978.97
32589.45
31978.47
22781.76
29871.44
```

The program in Code Listing 11-5 reads each number from the file and adds it to an accumulator variable. The try block contains code that can throw different types of exceptions. For example, the `Scanner` class's constructor can throw a `FileNotFoundException` if the file is not found, and the `Scanner` class's `nextDouble` method can throw an `InputMismatchException` (which is in the `java.util` package) if it reads a non-numeric value from the file. To handle these exceptions, the `try` statement has two `catch` clauses. Figure 11-7 shows the dialog box displayed by the program when no errors occur. This dialog box is displayed by the statement in lines 51 through 56. Figure 11-8 shows the dialog box displayed by the statement in lines 62 through 64 when the file cannot be found.

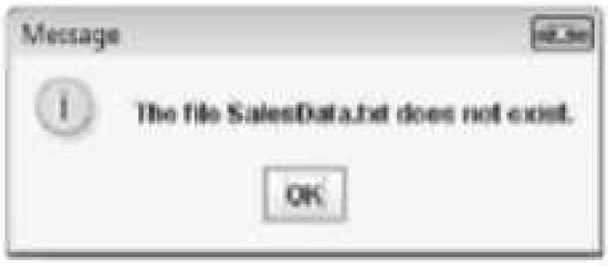
Figure 11-7 Dialog box displayed by the `SalesReport.java` program when no error occurs



Message

Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95

Figure 11-8 Dialog box displayed by the `SalesReport.java` program when the file cannot be found



Message

The file SalesData.txt does not exist.

Code Listing 11-5 **(SalesReport.java)**

```
 1 import java.io.*;     // For File class and FileNotFoundException
 2 import java.util.*;  // For Scanner and InputMismatchException
 3 import java.text.DecimalFormat;    // For the DecimalFormat class
 4 import javax.swing.JOptionPane;    // For the JOptionPane class
 5
 6 /**
 7    This program demonstrates how multiple exceptions can
 8    be caught with one try statement.
 9 */
10
11 public class SalesReport
12 {
13    public static void main(String[] args)
14    {
15       String filename = "SalesData.txt"; // File name
16       int months = 0;                    // Month counter
17       double oneMonth;                   // One month's sales
18       double totalSales = 0.0;           // Total sales
19       double averageSales;               // Average sales
20
21       // Create a DecimalFormat object.
```

```
      DecimalFormat dollar =
               new DecimalFormat("#,##0.00");

      try
      {
         // Open the file.
         File file = new File(filename);
         Scanner inputFile = new Scanner(file);

         // Process the contents of the file.
         while (inputFile.hasNext())
         {
            // Get a month's sales amount.
            oneMonth = inputFile.nextDouble();

            // Accumulate the amount.
            totalSales += oneMonth;

            // Increment the month counter
            months++;
         }

         // Close the file.
         inputFile.close();

         // Calculate the average.
         averageSales = totalSales / months;

         // Display the results.
         JOptionPane.showMessageDialog(null,
                           "Number of months: " + months +
                           "\nTotal Sales: $" +
                           dollar.format(totalSales) +
                           "\nAverage Sales: $" +
                           dollar.format(averageSales));
      }
      catch(FileNotFoundException e)
      {
         // Thrown by the Scanner constructor when
         // the file is not found.
         JOptionPane.showMessageDialog(null,
                              "The file " + filename +
                              " does not exist.");
      }
      catch(InputMismatchException e)
      {
         // Thrown by the Scanner class's nextDouble
         // method when a non-numeric value is found.
         JOptionPane.showMessageDialog(null,
                     "Non-numeric data found " +
                     "in the file.");
      }

      System.exit(0);
   }
}
```

When an exception is thrown by code in the try block, the JVM begins searching the `try` statement for a `catch` clause that can handle it. It searches the `catch` clauses from top to bottom and passes control of the program to the first `catch` clause with a parameter that is compatible with the exception.

Using Exception Handlers to Recover from Errors

The program in Code Listing 11-5 demonstrates how a `try` statement can have several `catch` clauses in order to handle different types of exceptions. However, the program does not use the exception handlers to recover from any of the errors. Regardless of whether the file is not found or a non-numeric item is encountered in the file, this program still halts. The program in Code Listing 11-6 is a better example of effective exception handling. It attempts to recover from as many of the exceptions as possible.

Code Listing 11-6 (`SalesReport2.java`)

```
import java.io.*;           // For File class and FileNotFoundException
import java.util.*;         // For Scanner and InputMismatchException
import java.text.DecimalFormat;  // For the DecimalFormat class
import javax.swing.JOptionPane;  // For the JOptionPane class

/**
   This program demonstrates how exception handlers can
   be used to recover from errors.
*/

public class SalesReport2
{
   public static void main(String[] args)
   {
      String filename = "SalesData.txt";       // File name
      int months = 0;                          // Month counter
      double oneMonth;                         // One month's sales
      double totalSales = 0.0;                 // Total sales
      double averageSales;                     // Average sales

      // Create a DecimalFormat object.
      DecimalFormat dollar =
                  new DecimalFormat("#,##0.00");

      // Attempt to open the file by calling the
      // openfile method.
      Scanner inputFile = openFile(filename);

      // If the openFile method returned null, then
      // the file was not found. Get a new file name.
      while (inputFile == null)
      {
         filename = JOptionPane.showInputDialog(
                    "ERROR: " + filename +
                    " does not exist.\n" +
                    "Enter another file name: ");
         inputFile = openFile(filename);
      }

      // Process the contents of the file.
      while (inputFile.hasNext())
      {
         try
         {
            // Get a month's sales amount.
            oneMonth = inputFile.nextDouble();

            // Accumulate the amount.
            totalSales += oneMonth;

            // Increment the month counter.
            months++;
         }
         catch(InputMismatchException e)
         {
            // Display an error message.
            JOptionPane.showMessageDialog(null,
                    "Non-numeric data found in the file.\n" +
                    "The invalid record will be skipped.");

            // Skip past the invalid data.
            inputFile.nextLine();
         }
      }

      // Close the file.
      inputFile.close();

      // Calculate the average.
      averageSales = totalSales / months;

```

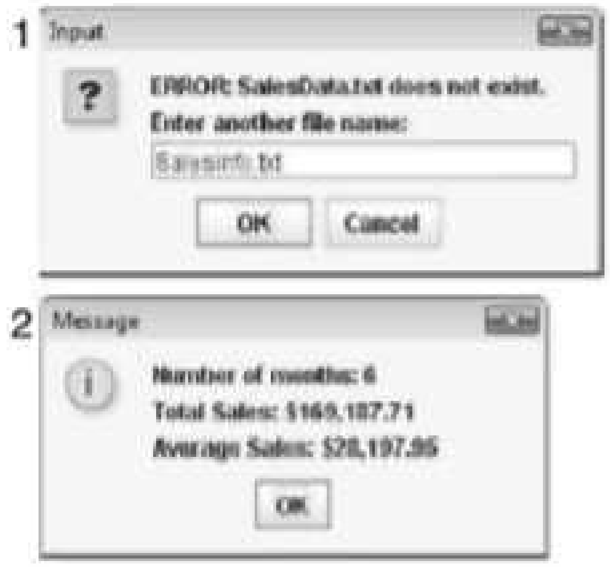
```
 72       // Display the results.
 73       JOptionPane.showMessageDialog(null,
 74              "Number of months: " + months +
 75              "\nTotal Sales: $" +
 76              dollar.format(totalSales) +
 77              "\nAverage Sales: $" +
 78              dollar.format(averageSales));
 79       System.exit(0);
 80    }
 81
 82    /**
 83       The openFile method opens the specified file and
 84       returns a reference to a Scanner object.
 85       @param filename The name of the file to open.
 86       @return A Scanner reference, if the file exists
 87              Otherwise, null is returned.
 88    */
 89
 90    public static Scanner openFile(String filename)
 91    {
 92       Scanner scan;
 93
 94       // Attempt to open the file.
 95       try
 96       {
 97          File file = new File(filename);
 98          scan = new Scanner(file);
 99       }
100       catch(FileNotFoundException e)
101       {
102          scan = null;
103       }
104
105       return scan;
106    }
107 }
```

Let's look at how this program recovers from a `FileNotFoundException`. The `openFile` method, in lines 90 through 106, accepts a file name as its argument. The method creates a `File` object (passing the file name to the constructor) and a `Scanner` object. If the `Scanner` class constructor throws a `FileNotFoundException`, the method returns `null`. Otherwise, it returns a reference to the `Scanner` object. In the `main` method, a loop is used in lines 31 through 38 to ask the user for a different file name in the event that the `openFile` method returns `null`.

Now let's look at how the program recovers from unexpectedly encountering a non-numeric item in the file. The statement in line 46, which calls the `Scanner` class's `nextDouble` method, is wrapped in a `try` statement that catches the `InputMismatchException`. If this exception is thrown by the `nextDouble` method, the catch block in lines 54 through 63 displays a message indicating that a non-numeric item was encountered and that the invalid record will be skipped. The invalid data is then read from the file with the `nextLine` method in line 62. Because the statement `months++` in line 52 is in the try block, it will not be executed when the exception occurs, so the number of months will still be correct. The loop continues processing with the next line in the file.

Let's look at some examples of how the program recovers from these errors. Suppose we rename *SalesData.txt* file as *SalesInfo.txt*. Figure 11-9 shows an example running of the program.

Figure 11-9 Interaction with the `SalesReport2.java` program


1
```
Input
ERROR: SalesData.txt does not exist.
Enter another file name:
SalesInfo.txt
```

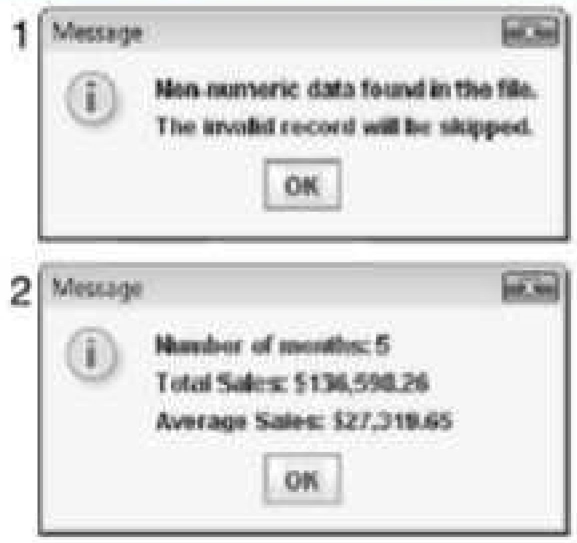
2
```
Message
Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95
```

Now, suppose we change the name of the file back to *SalesData.txt* and edit its contents as follows:

```
24987.62
26978.97
abc
31978.47
22781.76
29871.44
```

Notice that the third item is no longer a number. Figure 11-10 shows an example running of the program.

Figure 11-10 Dialog boxes displayed by the `SalesReport2.java` program


1
```
Message
Non-numeric data found in the file.
The invalid record will be skipped.
```

2
```
Message
Number of months: 5
Total Sales: $136,598.26
Average Sales: $27,319.65
```

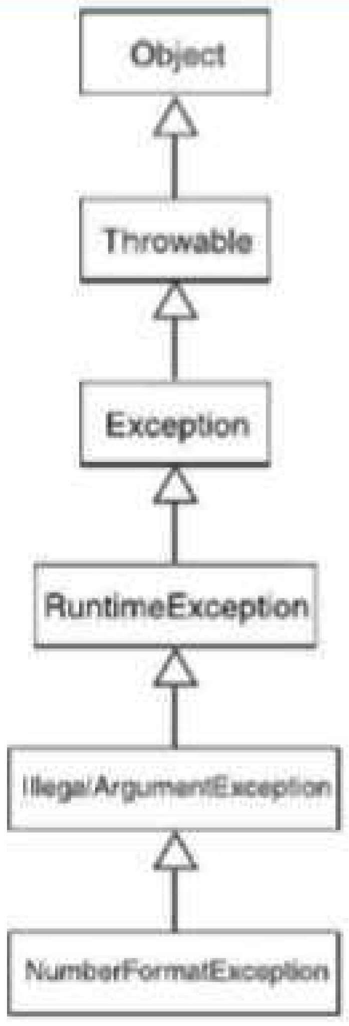
Handle Each Exception Only Once in a `try` Statement

Not including polymorphic references, a `try` statement may have only one `catch` clause for each specific type of exception. For example, the following `try` statement will cause the compiler to issue an error message because it handles a `NumberFormatException` object with two `catch` clauses:

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println("Bad number format.");
}
// ERROR!!! NumberFormatException has already been caught!
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

Sometimes you can cause this error by using polymorphic references. For example, look at Figure 11-11, which shows an inheritance hierarchy for the `NumberFormatException` class.

Figure 11-11 Inheritance hierarchy for the `NumberFormatException` class



As you can see from the figure, the `NumberFormatException` class inherits from the `IllegalArgumentException` class. Now look at the following code:

```
try
{
   number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
// This will also cause an error.
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
```

The compiler issues an error message regarding the second `catch` clause, reporting that `NumberFormatException` has already been caught. This is because the first `catch` clause, which catches `IllegalArgumentException` objects, will polymorphically catch `NumberFormatException` objects.

If you are handling multiple exceptions in the same `try` statement and some of the exceptions are related to each other through inheritance, then you should handle the more specialized exception classes before the more general exception classes. We can rewrite the previous code as follows, with no errors:

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number format.");
}
```

The `finally` Clause

The `try` statement may have an optional `finally` clause, which must appear after all of the `catch` clauses. Here is the general format of a `try` statement with a `finally` clause:

```
try
{
    (try block statements . . .)
}
```

```
catch (ExceptionType ParameterName)
{
    (catch block statements . . .)
}
finally
{
    (finally block statements . . .)
}
```

The *finally block* is one or more statements that are always executed after the try block has executed and after any catch blocks have executed if an exception was thrown. The statements in the finally block execute whether an exception occurs or not. For example, the following code opens a file of `doubles` and reads its contents. The outer `try` statement opens the file and has a `catch` clause that catches the `FileNotFoundException`. The inner `try` statement reads values from the file and has a `catch` clause that catches the `InputMismatchException`. The finally block closes the file regardless of whether an `InputMismatchException` occurs.

```
try
{
   // Open the file.
   File file = new File(filename);
   Scanner inputFile = new Scanner(file);

   try
   {
      // Read and display the file's contents.
      while (inputFile.hasNext())
      {
         System.out.println(inputFile.nextDouble());
      }
   }
   catch (InputMismatchException e)
   {
      System.out.println("Invalid data found.");
   }
   finally
   {
      // Close the file.
      inputFile.close();
   }
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

The Stack Trace

Quite often, a method will call another method, which will call yet another method. For example, method A calls method B, which calls method C. The *call stack* is an internal list of all the methods that are currently executing.

When an exception is thrown by a method that is executing under several layers of method calls, it is sometimes helpful to know which methods were responsible for the method being called. A *stack trace* is a list of all the methods in the call stack. It indicates the method that was executing when an exception occurred and all of the methods that were called in order to execute that method. For example, look at the program in Code Listing 11-7. It has three methods: `main`, `myMethod`, and `produceError`. The `main` method calls `myMethod`, which calls `produceError`. The `produceError` method causes an exception by passing an invalid position number to the `String` class's `charAt` method. The exception is not handled by the program, but is dealt with by the default exception handler.

Code Listing 11-7 (`StackTrace.java`)

```
/**
   This program demonstrates the stack trace that is
   produced when an exception is thrown.
*/

public class StackTrace
{
   public static void main(String[] args)
   {
      System.out.println("Calling myMethod...");
      myMethod();
      System.out.println("Method main is done.");
   }

   /**
      MyMethod
   */

   public static void myMethod()
   {
      System.out.println("Calling produceError...");
      produceError();
      System.out.println("myMethod is done.");
   }

   /**
      produceError
   */

30     public static void produceError()
31     {
32        String str = "abc";
33
34        // The following statement will cause an error.
35        System.out.println(str.charAt(3));
36        System.out.println("produceError is done.");
37     }
38  }
```

Program Output

```
Calling myMethod...
Calling produceError...
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
      String index out of range: 3
        at java.lang.String.charAt(Unknown Source)
        at StackTrace.produceError(StackTrace.java:35)
        at StackTrace.myMethod(StackTrace.java:22)
        at StackTrace.main(StackTrace.java:11)
```

When the exception occurs, the error message shows a stack trace listing the methods that were called in order to produce the exception. The first method that is listed in the stack trace, `charAt`, is the method that is responsible for the exception. The next method, `produceError`, is the method that called `charAt`. The next method, `myMethod`, is the method that called `produceError`. The last method, `main`, is the method that called `myMethod`. The stack trace shows the chain of methods that were called when the exception was thrown.

NOTE: All exception objects have a `printStackTrace` method, inherited from the `Throwable` class, which can be used to print a stack trace.

Handling Multiple Exceptions with One `catch` Clause (Java 7)

In versions of Java prior to Java 7, each `catch` clause can handle only one type of exception. Beginning with Java 7, however, a `catch` clause can handle more than one type of exception. This can reduce a lot of duplicated code in a `try` statement that needs to catch multiple exceptions, but perform the same operation for each one. For example, suppose we have the following `try` statement in a program:

```
try
{
   (try block statements . . .)
}
catch(NumberFormatException ex)
{
   respondToError();
}
```

```
catch(IOException ex)
{
   respondToError();
}
```

This `try` statement has two `catch` clauses: one that handles a `NumberFormatException`, and another that handles an `IOException`. Notice that both catch blocks do the same thing: they call a method named `respondToError`. Because both catch blocks perform the same operation, the `catch` clauses can be combined into a single `catch` clause that handles both types of exception, as shown here:

```
try
{
   (try block statements . . .)
}
catch(NumberFormatException | IOException ex)
{
   respondToError();
}
```

Notice in the `catch` clause that the exception types are separated by a | symbol, which is the same symbol as that used for the logical OR operator. You can think of this as meaning that the clause will catch a `NumberFormatException` or an `IOException`. The following code shows a `catch` clause that handles three types of exceptions:

```
try
{
   (try block statements . . .)
}
catch(NumberFormatException | IOException | InputMismatchException ex)
{
   respondToError();
}
```

In this code, the `catch` clause will handle a `NumberFormatException` or an `IOException` or an `InputMismatchException`.

The ability to catch multiple types of exceptions with a single `catch` clause is known as *multi-catch*, and was introduced in Java 7. Code Listing 11-8 shows a complete program that uses multi-catch. The `catch` clause in line 34 can handle a `FileNotFoundException` or an `InputMismatchException`.

Code Listing 11-8 (`MultiCatch.java`)

```
1 import java.io.*;      // For File class and FileNotFoundException
2 import java.util.*;    // For Scanner and InputMismatchException
3
4 /**
5    This program demonstrates how multiple exceptions can
6    be caught with a single catch clause.
7 */
8
```

```
public class MultiCatch
{
   public static void main(String[] args)
   {
      int number;    // To hold a number from the file

      try
      {
         // Open the file.
         File file = new File("Numbers.txt");
         Scanner inputFile = new Scanner(file);

         // Process the contents of the file.
         while (inputFile.hasNext())
         {
            // Get a number from the file.
            number = inputFile.nextInt();

            // Display the number.
            System.out.println(number);
         }

         // Close the file.
         inputFile.close();
      }
      catch(FileNotFoundException | InputMismatchException ex)
      {
         // Display an error message.
         System.out.println("Error processing the file.");
      }
   }
}
```

NOTE: If you are using a version of Java prior to Java 7, you cannot use multi-catch.

When an Exception Is Not Caught

When an exception is thrown, it cannot be ignored. It must be handled by the program, or by the default exception handler. When the code in a method throws an exception, the normal execution of that method stops and the JVM searches for a compatible exception handler inside the method. If there is no code inside the method to handle the exception, then control of the program is passed to the previous method in the call stack (that is, the method that called the offending method). If that method cannot handle the exception, then control is passed again, up the call stack, to the previous method. This continues until control reaches the `main` method. If the `main` method does not handle the exception, then the program is halted and the default exception handler handles the exception.

This was the case for the program in Code Listing 11-7. Because the `produceError` method did not handle the exception, control was passed back to `myMethod`. It didn't handle the exception either, so control was passed back to `main`. Because `main` didn't handle the exception, the program halted and the default exception handler displayed the error messages.

Checked and Unchecked Exceptions

In Java, there are two categories of exceptions: unchecked and checked. *Unchecked exceptions* are those that inherit from the `Error` class or the `RuntimeException` class. Recall that the exceptions that inherit from `Error` are thrown when a critical error occurs, such as running out of memory. You should not handle these exceptions because the conditions that cause them can rarely be dealt with in the program. Also recall that `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript. It is best not to handle these exceptions either, because they can be avoided with properly written code. So, you should not handle unchecked exceptions.

All of the remaining exceptions (that is, those that do *not* inherit from `Error` or `RuntimeException`) are *checked exceptions*. These are the exceptions that you should handle in your program. If the code in a method can potentially throw a checked exception, then that method must meet one of the following requirements:

- It must handle the exception, or
- It must have a `throws` clause listed in the method header.

The `throws` clause informs the compiler of the exceptions that could get thrown from a method. For example, look at the following method:

```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);

   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
      System.out.println(inputFile.nextLine());
   }

   // Close the file.
   inputFile.close();
}
```

The code in this method is capable of throwing a `FileNotFoundException`, which is a checked exception. Because the method does not handle this exception, it must have a `throws` clause in its header or it will not compile.

The key word `throws` is written at the end of the method header, followed by a list of the types of exceptions that the method can throw. Here is the revised method header:

```
public void displayFile(String name) throws FileNotFoundException
```

The `throws` clause tells the compiler that this method can throw a `FileNotFoundException`. (If there is more than one type of exception, you separate them with commas.)

Now you know why you wrote a throws clause on methods that perform file operations in the previous chapters. We did not handle any of the checked exceptions that might occur, so we had to inform the compiler that our methods might pass them up the call stack.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

11.1 Briefly describe what an exception is.

11.2 What does it mean to "throw" an exception?

11.3 If an exception is thrown and the program does not handle it, what happens?

11.4 Other than the Object class, what is the superclass for all exceptions?

11.5 What is the difference between exceptions that inherit from the Error class and exceptions that inherit from the Exception class?

11.6 What is the difference between a try block and a catch block?

11.7 After the catch block has handled the exception, where does program execution resume?

11.8 How do you retrieve an error message from an exception?

11.9 If multiple exceptions can be thrown by code in a try block, how does the JVM know which catch clause it should pass the control of the program to?

11.10 When does the code in a finally block execute?

11.11 What is the call stack? What is a stack trace?

11.12 A program's main method calls method A, which calls method B. None of these methods performs any exception handling. The code in method B throws an exception. Describe what happens.

11.13 What are the differences between a checked and an unchecked exception?

11.14 When are you required to have a throws clause in a method header?

11.2 Throwing Exceptions

CONCEPT: You can write code that throws one of the standard Java exceptions, or an instance of a custom exception class that you have designed.

You can use the throw statement to throw an exception manually. The general format of the throw statement is as follows:

```
throw new ExceptionType(MessageString);
```

The throw statement causes an exception object to be created and thrown. In this general format, *ExceptionType* is an exception class name and *MessageString* is an optional String argument passed to the exception object's constructor. The *MessageString* argument contains a custom error message that can be retrieved from the exception object's getMessage method. If you do not pass a message to the constructor, the exception will have a null message. Here is an example of a throw statement:

```
throw new Exception("Out of fuel");
```

This statement creates an object of the `Exception` class and passes the string "Out of fuel" to the object's constructor. The object is then thrown, which causes the exception-handling process to begin.

NOTE: Don't confuse the `throw` statement with the `throws` clause. The `throw` statement causes an exception to be thrown. The `throws` clause informs the compiler that a method throws one or more exceptions.

Recall the `DateComponent` class from Chapter 9. Its constructor accepts a string containing a date in the form MONTH/DAY/YEAR. It uses a `StringTokenizer` object to extract the month, day, and year from the string and stores these values in the `month`, `day`, and `year` fields. Suppose we want to prevent a `null` reference from being passed as an argument into the constructor. One way to accomplish this is to have the constructor throw an exception when such an argument is passed. Here is the modified code for the `DateComponent` constructor:

```
public DateComponent(String dateStr)
{
   // Ensure that dateStr is not null.
   if (dateStr == null)
   {
      throw new IllegalArgumentException(
                   "null reference passed to " +
                   "DateComponent constructor");
   }

   // Create a StringTokenizer object.
   StringTokenizer strTokenizer =
       new StringTokenizer(dateStr, "/");

   // Extract the tokens.
   month = strTokenizer.nextToken();
   day = strTokenizer.nextToken();
   year = strTokenizer.nextToken();
}
```

This constructor throws an `IllegalArgumentException` if the `dateStr` parameter is `null`. The message "null reference passed to `DateComponent` constructor" is passed to the exception object's constructor. When we catch this exception, we can retrieve the message by calling the object's `getMessage` method. The `IllegalArgumentException` class was chosen for this error condition because it seems like the most appropriate exception to throw in response to an illegal argument being passed to the constructor. (Note that `IllegalArgumentException` inherits from `RuntimeException`, which inherits from `Exception`.)

NOTE: Because the `IllegalArgumentException` class inherits from the `RuntimeException` class, it is unchecked. If we had chosen a checked exception class, we would have to put a `throws` clause in the constructor's header.

The program in Code Listing 11-9 demonstrates how the modified constructor works.

Code Listing 11-9 **(DateComponentExceptionDemo.java)**

```java
/**
   This program demonstrates how the DateComponent
   class constructor throws an exception.
*/

public class DateComponentExceptionDemo
{
   public static void main(String[] args)
   {
      // Create a null String reference.
      String str = null;

      // Attempt to pass the null reference to
      // the DateComponent constructor.
      try
      {
         DateComponent dc = new DateComponent(str);
      }
      catch (IllegalArgumentException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Program Output

```
null reference passed to DateComponent constructor
```

Creating Your Own Exception Classes

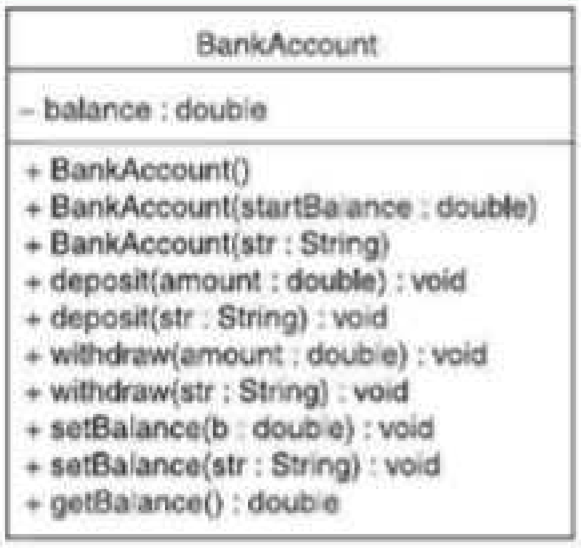
To meet the needs of a specific class or application, you can create your own exception classes by extending the `Exception` class or one of its subclasses.

Let's look at an example that uses programmer-defined exceptions. Recall the `BankAccount` class from Chapter 6. This class holds the data for a bank account. A UML diagram for the class is shown in Figure 11-12.

There are a number of errors that could cause a `BankAccount` object to perform its duties incorrectly. Here are some specific examples:

- A negative starting balance is passed to the constructor.
- A negative number is passed to the `deposit` method.
- A negative number is passed to the `withdraw` method.
- The amount passed to the `withdraw` method exceeds the account's balance.

Figure 11-12 UML diagram for the `BankAccount` class


BankAccount
- balance : double
+ BankAccount()
+ BankAccount(startBalance : double)
+ BankAccount(str : String)
+ deposit(amount : double) : void
+ deposit(str : String) : void
+ withdraw(amount : double) : void
+ withdraw(str : String) : void
+ setBalance(b : double) : void
+ setBalance(str : String) : void
+ getBalance() : double

We can create our own exceptions that represent each of these error conditions. Then we can rewrite the class so it throws one of our custom exceptions when any of these errors occur. Let's start by creating an exception class for a negative starting balance. Code Listing 11-10 shows an exception class named `NegativeStartingBalance`.

Code Listing 11-10 **(NegativeStartingBalance.java)**

```
/**
   NegativeStartingBalance exceptions are thrown by the
   BankAccount class when a negative starting balance is
   passed to the constructor.
*/

public class NegativeStartingBalance
                      extends Exception
{
   /**
      This constructor uses a generic
      error message.
   */

   public NegativeStartingBalance()
   {
      super("Error: Negative starting balance");
   }

   /**
      This constructor specifies the bad starting
      balance in the error message.
      @param The bad starting balance.
   */

   public NegativeStartingBalance(double amount)
   {
      super("Error: Negative starting balance: " +
            amount);
   }
}
```

Notice that this class extends the `Exception` class. It has two constructors. The no-arg constructor passes the string `"Error: Negative starting balance"` to the superclass constructor. This is the error message that is retrievable from an object's `getMessage` method. The second constructor accepts the starting balance as a `double` argument. This amount is used to pass a more detailed error message containing the starting balance amount to the superclass constructor.

The following code shows one of the `BankAccount` constructors rewritten to throw a `NegativeStartingBalance` exception when a negative value is passed as the starting balance.

```
public BankAccount(double startBalance)
                 throws NegativeStartingBalance
{
   if (startBalance < 0)
      throw new NegativeStartingBalance(startBalance);

   balance = startBalance;
}
```

Note that `NegativeStartingBalance` extends the `Exception` class. This means that it is a checked exception class. Because of this, the constructor header must have a `throws` clause listing the exception type.

You will find the modified *BankAccount.java* file in this chapter's source code, available on the book's companion Web site at www.pearsonhighered.com/gaddis. The program in Code Listing 11-11 demonstrates the new constructor by forcing it to throw the `NegativeStartingBalance` exception.

Code Listing 11-11 (`AccountTest.java`)

```
/**
   This program demonstrates how the BankAccount
   class constructor throws custom exceptions.
*/

public class AccountTest
{
   public static void main(String [] args)
   {
      // Force a NegativeStartingBalance exception.
      try
      {
         BankAccount account =
                        new BankAccount(-100.0);
      }
      catch(NegativeStartingBalance e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Program Output

```
Error: Negative starting balance: -100.0
```

Using the `@exception` Tag in Documentation Comments

When writing the documentation comments for a method, you can document the exceptions thrown by the method by using an `@exception` tag. When the `javadoc` utility sees an `@exception` tag inside a method's documentation comments, it knows that the name of an exception appears next, followed by a description of the events that cause the exception. The general format of an `@exception` tag comment is as follows:

```
@exception ExceptionName Description
```

ExceptionName is the name of an exception and *Description* is a description of the circumstances that cause the exception. Remember the following points about `@exception` tag comments:

- The `@exception` tag in a method's documentation comment must appear after the general description of the method.
- The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol), or at the beginning of another tag.

When a method's documentation comments contain an `@exception` tag, the `javadoc` utility will create a Throws section in the method's documentation. This is where the descriptions of the exceptions thrown by the method will be listed. As an example, here are the documentation comments for the `BankAccount` class's constructor presented earlier:

```
/**
   This constructor sets the starting balance
   to the value passed as an argument.
   @param startBalance The starting balance.
   @exception NegativeStartingBalance When
              startBalance is negative.
*/
```