Chapter 4

Mastering Advanced Language Features Part 1

Chapters 2 and 3 laid a foundation for learning the Java language. Chapter 4 builds onto this foundation by introducing you to some of Java's more advanced language features, specifically those features related to nested types, packages, static imports, and exceptions. Additional advanced language features are covered in Chapter 5.

Nested Types

Classes that are declared outside of any class are known as *top-level classes*. Java also supports *nested classes*, which are classes declared as members of other classes or scopes. Nested classes help you implement top-level class architecture.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as *inner classes*.

This section introduces you to static member classes and inner classes. For each kind of nested class, I provide you with a brief introduction, an abstract example, and a more practical example. The section then briefly examines the topic of nesting interfaces within classes.

Static Member Classes

A *static member class* is a `static` member of an enclosing class. Although enclosed, it does not have an enclosing instance of that class, and cannot access the enclosing class's instance fields and call its instance methods. However, it can access or call `static` members of the enclosing class, even those members that are declared `private`.

Listing 4–1 presents a static member class declaration.

Listing 4–1. *Declaring a static member class*

```
public class EnclosingClass
{
   private static int i;
   private static void m1()
   {
      System.out.println(i); // Output: 1
   }
   public static void m2()
   {
      EnclosedClass.accessEnclosingClass();
   }
   public static class EnclosedClass
   {
      public static void accessEnclosingClass()
      {
         i = 1;
         m1();
      }
      public void accessEnclosingClass2()
      {
         m2();
      }
   }
}
```

Listing 4–1 declares a top-level class named `EnclosingClass` with class field `i`, class methods `m1()` and `m2()`, and static member class `EnclosedClass`. Also, `EnclosedClass` declares class method `accessEnclosingClass()` and instance method `accessEnclosingClass2()`.

Because `accessEnclosingClass()` is declared `static`, `m2()` must prefix this method's name with `EnclosedClass` and the member access operator to call this method. Also, `EnclosingClass` must be part of the prefix when calling this method from beyond this class. For example, `EnclosingClass.EnclosedClass.accessEnclosingClass();`.

Because `accessEnclosingClass2()` is nonstatic, it must be called from an instance of `EnclosedClass`. For example, when calling this method from beyond `EnclosingClass`, you might specify `EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass(); ec.accessEnclosingClass2();`.

Static member classes have their uses. For example, Listing 4–2's `Double` and `Float` static member classes provide different implementations of their enclosing `Rectangle` class. The `Float` version occupies less memory because of its 32-bit `float` fields, and the `Double` version provides greater accuracy because of its 64–bit `double` fields.

Listing 4–2. *Using static member classes to declare multiple implementations of their enclosing class*

```
public abstract class Rectangle
{
   public abstract double getX();
   public abstract double getY();
   public abstract double getWidth();
   public abstract double getHeight();
   public static class Double extends Rectangle
```

```
   {
      private double x, y, width, height;
      public Double(double x, double y, double width, double height)
      {
         this.x = x;
         this.y = y;
         this.width = width;
         this.height = height;
      }
      public double getX() { return x; }
      public double getY() { return y; }
      public double getWidth() { return width; }
      public double getHeight() { return height; }
   }
   public static class Float extends Rectangle
   {
      private float x, y, width, height;
      public Float(float x, float y, float width, float height)
      {
         this.x = x;
         this.y = y;
         this.width = width;
         this.height = height;
      }
      public double getX() { return x; }
      public double getY() { return y; }
      public double getWidth() { return width; }
      public double getHeight() { return height; }
   }
   // Prevent subclassing. Use the type-specific Double and Float
   // implementation subclass classes to instantiate.
   private Rectangle() {}
   public boolean contains(double x, double y)
   {
      return (x >= getX() && x < getX()+getWidth()) &&
             (y >= getY() && y < getY()+getHeight());
   }
}
```

Listing 4–2's `Rectangle` class demonstrates nested subclasses. Each of the `Double` and `Float` static member classes subclass the abstract `Rectangle` class, providing private floating-point or double precision floating-point fields, and overriding `Rectangle`'s abstract methods to return these fields' values as `double`s.

`Rectangle` is abstract because it makes no sense to instantiate this class. Because it also makes no sense to directly extend `Rectangle` with new implementations (the `Double` and `Float` nested subclasses should be sufficient), its default constructor is declared `private`. Instead, you must instantiate `Rectangle.Float` (to save memory) or `Rectangle.Double` (when accuracy is required). Check out Listing 4–3.

Listing 4–3. *Creating and using different* `Rectangle` *implementations*

```
public static void main(String[] args)
{
   Rectangle r = new Rectangle.Double(10.0, 10.0, 20.0, 30.0);
   System.out.println("x = " + r.getX());
   System.out.println("y = " + r.getY());
```

```
      System.out.println("width = " + r.getWidth());
      System.out.println("height = " + r.getHeight());
      System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
      System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
      System.out.println();
      r = new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f);
      System.out.println("x = " + r.getX());
      System.out.println("y = " + r.getY());
      System.out.println("width = " + r.getWidth());
      System.out.println("height = " + r.getHeight());
      System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
      System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
   }
```

This method generates the following output:

```
x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false

x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false
```

Java's class library contains many static member classes. For example, the `Character` class (in the `java.lang` package) encloses a static member class named `Subset` whose instances represent subsets of the Unicode character set. `AbstractMap.SimpleEntry`, `ObjectInputStream.GetField`, and `KeyStore.PrivateKeyEntry` are other examples.

> **NOTE:** When you compile an enclosing class that contains a static member class, the compiler creates a classfile for the static member class whose name consists of its enclosing class's name, a dollar-sign character, and the static member class's name. For example, compile Listing 4–1 and you will discover `EnclosingClass$EnclosedClass.class` in addition to `EnclosingClass.class`. This format also applies to nonstatic member classes.

Nonstatic Member Classes

A *nonstatic member class* is a non-`static` member of an enclosing class. Each instance of the nonstatic member class implicitly associates with an instance of the enclosing class. The nonstatic member class's instance methods can call instance methods in the enclosing class and access the enclosing class instance's nonstatic fields.

Listing 4–4 presents a nonstatic member class declaration.

Listing 4–4. *Declaring a nonstatic member class*

```
public class EnclosingClass
{
   private int i;
   private void m1()
   {
      System.out.println(i); // Output: 1
   }
   public class EnclosedClass
   {
      public void accessEnclosingClass()
      {
         i = 1;
         m1();
      }
   }
}
```

Listing 4–4 declares a top-level class named `EnclosingClass` with instance field `i`, instance method `m1()`, and nonstatic member class `EnclosedClass`. Furthermore, `EnclosedClass` declares instance method `accessEnclosingClass()`.

Because `accessEnclosingClass()` is nonstatic, `EnclosedClass` must be instantiated before this method can be called. This instantiation must take place via an instance of `EnclosingClass`. Listing 4–5 accomplishes these tasks.

Listing 4–5. *Calling a nonstatic member class's instance method*

```
public class NSMCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass ec = new EnclosingClass();
      ec.new EnclosedClass().accessEnclosingClass();
   }
}
```

Listing 4–5's `main()` method first instantiates `EnclosingClass` and saves its reference in local variable `ec`. Then, `main()` uses this reference as a prefix to the `new` operator, to instantiate `EnclosedClass`, whose reference is then used to call `accessEnclosingClass()`.

> **NOTE:** Prefixing `new` with a reference to the enclosing class is rare. Instead, you will typically call an enclosed class's constructor from within a constructor or an instance method of its enclosing class.

Suppose you need to maintain a to-do list of items, where each item consists of a name and a description. After some thought, you create Listing 4–6's `ToDo` class to implement these items.

Listing 4–6. *Implementing to-do items as name-description pairs*

```
public class ToDo
{
   private String name;
```

```
   private String desc;
   public ToDo(String name, String desc)
   {
      this.name = name;
      this.desc = desc;
   }
   public String getName()
   {
      return name;
   }
   public String getDesc()
   {
      return desc;
   }
   public String toString()
   {
      return "Name = " + getName() + ", Desc = " + getDesc();
   }
}
```

You next create a `ToDoList` class to store `ToDo` instances. `ToDoList` uses its `ToDoArray` nonstatic member class to store `ToDo` instances in a growable array—you do not know how many instances will be stored, and Java arrays have fixed lengths. See Listing 4–7.

Listing 4–7. *Storing a maximum of two `ToDo` instances in a `ToDoArray` instance*

```
public class ToDoList
{
   private ToDoArray toDoArray;
   private int index = 0;
   public ToDoList()
   {
      toDoArray = new ToDoArray(2);
   }
   public boolean hasMoreElements()
   {
      return index < toDoArray.size();
   }
   public ToDo nextElement()
   {
      return toDoArray.get(index++);
   }
   public void add(ToDo item)
   {
      toDoArray.add(item);
   }
   private class ToDoArray
   {
      private ToDo[] toDoArray;
      private int index = 0;
      ToDoArray(int initSize)
      {
         toDoArray = new ToDo[initSize];
      }
      void add(ToDo item)
      {
         if (index >= toDoArray.length)
         {
```

```
            ToDo[] temp = new ToDo[toDoArray.length*2];
            for (int i = 0; i < toDoArray.length; i++)
               temp[i] = toDoArray[i];
            toDoArray = temp;
         }
         toDoArray[index++] = item;
      }
      ToDo get(int i)
      {
         return toDoArray[i];
      }
      int size()
      {
         return index;
      }
   }
}
```

In addition to providing an `add()` method to store `ToDo` instances in the `ToDoArray` instance, `ToDoList` provides `hasMoreElements()` and `nextElement()` methods to iterate over and return the stored instances. Listing 4–8 demonstrates these methods.

Listing 4–8. *Creating a list of `ToDo` instances and iterating over this list*

```
public static void main(String[] args)
{
   ToDoList toDoList = new ToDoList();
   toDoList.add(new ToDo("#1", "Do laundry."));
   toDoList.add(new ToDo("#2", "Buy groceries."));
   toDoList.add(new ToDo("#3", "Vacuum apartment."));
   toDoList.add(new ToDo("#4", "Write report."));
   toDoList.add(new ToDo("#5", "Wash car."));
   while (toDoList.hasMoreElements())
      System.out.println(toDoList.nextElement());
}
```

This method generates the following output:

```
Name = #1, Desc = Do laundry.
Name = #2, Desc = Buy groceries.
Name = #3, Desc = Vacuum apartment.
Name = #4, Desc = Write report.
Name = #5, Desc = Wash car.
```

Java's class library presents many examples of nonstatic member classes. For example, the `java.util` package's `HashMap` class declares private `HashIterator`, `ValueIterator`, `KeyIterator`, and `EntryIterator` classes for iterating over a hashmap's values, keys, and entries. (I will discuss `HashMap` in Chapter 8.)

> **NOTE:** Code within an enclosed class can obtain a reference to its enclosing class instance by qualifying reserved word `this` with the enclosing class's name and the member access operator. For example, if code within `accessEnclosingClass()` needed to obtain a reference to its `EnclosingClass` instance, it would specify `EnclosingClass.this`.

Anonymous Classes

An *anonymous class* is a class without a name. Furthermore, it is not a member of its enclosing class. Instead, an anonymous class is simultaneously declared (as an anonymous extension of a class or as an anonymous implementation of an interface) and instantiated any place where it is legal to specify an expression.

Listing 4–9 demonstrates an anonymous class declaration and instantiation.

Listing 4–9. *Declaring and instantiating an anonymous class that extends a class*

```
abstract class Speaker
{
   abstract void speak();
}
public class ACDemo
{
   public static void main(final String[] args)
   {
      new Speaker()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";
         void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```

Listing 4–9 introduces an abstract class named `Speaker` and a concrete class named `ACDemo`. The latter class's `main()` method declares an anonymous class that extends `Speaker` and overrides its `speak()` method. When this method is called, it outputs `main()`'s first command-line argument or a default message if there are no arguments.

An anonymous class does not have a constructor (because the anonymous class does not have a name). However, its classfile does contain a hidden method that performs instance initialization. This method calls the superclass's noargument constructor (prior to any other initialization), which is the reason for specifying `Speaker()` after `new`.

Anonymous class instances should be able to access the surrounding scope's local variables and parameters. However, an instance might outlive the method in which it was conceived (as a result of storing the instance's reference in a field), and try to access local variables and parameters that no longer exist after the method returns.

Because Java cannot allow this illegal access, which would most likely crash the virtual machine, it lets an anonymous class instance only access local variables and parameters that are declared `final`. Upon encountering a final local variable/parameter name in an anonymous class instance, the compiler does one of two things:

- If the variable's type is primitive (`int` or `double`, for example), the compiler replaces its name with the variable's read-only value.
- If the variable's type is reference (`String`, for example), the compiler introduces, into the classfile, a *synthetic variable* (a manufactured variable) and code that stores the local variable's/parameter's reference in the synthetic variable.

Listing 4–10 demonstrates an alternative anonymous class declaration and instantiation.

Listing 4–10. *Declaring and instantiating an anonymous class that implements an interface*

```
interface Speakable
{
   void speak();
}
public class ACDemo
{
   public static void main(final String[] args)
   {
      new Speakable()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";
         public void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```

Listing 4–10 is very similar to Listing 4–9. However, instead of subclassing a `Speaker` class, this listing's anonymous class implements an interface named `Speakable`. Apart from the hidden method calling `Object()` (interfaces have no constructors), Listing 4–10 behaves like Listing 4–9.

Although an anonymous class does not have a constructor, you can provide an instance initializer to handle complex initialization. For example, `new Office() {{addEmployee(new Employee("John Doe"));}};` instantiates an anonymous subclass of `Office` and adds one `Employee` object to this instance by calling `Office`'s `addEmployee()` method.

You will often find yourself creating and instantiating anonymous classes for their convenience. For example, suppose you need to return a list of all filenames having the `.java` suffix. Listing 4–11 shows you how an anonymous class simplifies using the `java.io` package's `File` and `FilenameFilter` classes to achieve this objective.

Listing 4–11. *Using an anonymous class instance to return a list of files with `.java` extensions*

```
String[] list = new File(directory).list(new FilenameFilter()
                {
                   public boolean accept(File f, String s)
                   {
                      return s.endsWith(".java");
                   }
                });
```

NOTE: An instance of an anonymous class is similar to a *closure*, which is a first-class *function* (a method not declared in a class) with free variables that are bound in the *lexical environment* (surrounding scope). A *first-class function* is a function that can be passed as an argument to or returned from a method. A *free variable* is a variable referred to in a function that is not a local variable or a parameter. Think of this variable as a placeholder.

Despite their similarity, there are two key differences between these language features. First, anonymous classes are more syntactically verbose than closures. Second, an anonymous class instance does not really close over its surrounding scope, because Java cannot allow the anonymous class instance to access non-`final` local variables and parameters.

Java version 7 will introduce closures, although the exact syntax and implementation are unknown at the time of writing. However, Baptiste Wicht revealed Oracle's first attempt at implementing closures via his May 29, 2010 blog post "Java 7: Oracle pushes a first version of closures" (`http://www.baptiste-wicht.com/2010/05/oracle-pushes-a-first-version-of-closures/`).

Local Classes

A *local class* is a class that is declared anywhere that a local variable is declared. Furthermore, it has the same scope as a local variable. Unlike an anonymous class, a local class has a name and can be reused. Like anonymous classes, local classes only have enclosing instances when used in nonstatic contexts.

A local class instance can access the surrounding scope's local variables and parameters. However, the local variables and parameters that are accessed must be declared `final`. For example, Listing 4–12's local class declaration accesses a final parameter and a final local variable.

Listing 4–12. *Declaring a local class*

```
public class EnclosingClass
{
   public void m(final int x)
   {
      final int y = x*2;
      class LocalClass
      {
         int a = x;
         int b = y;
      }
      LocalClass lc = new LocalClass();
      System.out.println(lc.a);
      System.out.println(lc.b);
   }
}
```

Listing 4–12 declares `EnclosingClass` with its instance method `m()` declaring a local class named `LocalClass`. This local class declares a pair of instance fields (`a` and `b`) that are initialized to the values of final parameter `x` and final local variable `y` when `LocalClass` is instantiated: `new EnclosingClass().m(10);`, for example.

Local classes help improve code clarity because they can be moved closer to where they are needed. For example, Listing 4–13 declares an `Iterator` interface and a `ToDoList` class whose `iterator()` method returns an instance of its local `Iter` class as an `Iterator` instance (because `Iter` implements `Iterator`).

Listing 4–13. *The `Iterator` interface and the `ToDoList` class*

```
public interface Iterator
{
   boolean hasMoreElements();
   Object nextElement();
}
public class ToDoList
{
   private ToDo[] toDoList;
   private int index = 0;
   public ToDoList(int size)
   {
      toDoList = new ToDo[size];
   }
   public Iterator iterator()
   {
      class Iter implements Iterator
      {
         int index = 0;
         public boolean hasMoreElements()
         {
            return index < toDoList.length;
         }
         public Object nextElement()
         {
            return toDoList[index++];
         }
      }
      return new Iter();
   }
   public void add(ToDo item)
   {
      toDoList[index++] = item;
   }
}
```

Because each of `Iterator` and `ToDoList` is declared `public`, these types need to be stored in separate source files.

Listing 4–14's `main()` method demonstrates this revised `ToDoList` class, Listing 4–6's `ToDo` class, and `Iterator`.

Listing 4–14. *Creating a list of* `ToDo` *instances and iterating over this list*

```
public static void main(String[] args)
{
   ToDoList toDoList = new ToDoList(5);
   toDoList.add(new ToDo("#1", "Do laundry."));
   toDoList.add(new ToDo("#2", "Buy groceries."));
   toDoList.add(new ToDo("#3", "Vacuum apartment."));
   toDoList.add(new ToDo("#4", "Write report."));
   toDoList.add(new ToDo("#5", "Wash car."));
   Iterator iter = toDoList.iterator();
   while (iter.hasMoreElements())
      System.out.println(iter.nextElement());
}
```

The `Iterator` instance that is returned from `iterator()` returns `ToDo` items in the same order as when they were added to the list. Although you can only use the returned `Iterator` once, you can call `iterator()` whenever you need a new `Iterator`. This capability is a big improvement over the one-shot iterator presented in Listing 4–7.

Interfaces Within Classes

Interfaces can be nested within classes. Once declared, an interface is considered to be static, even if it is not declared `static`. For example, Listing 4–15 declares an enclosing class named `X` along with two nested static interfaces named `A` and `B`.

Listing 4–15. *Declaring a pair of interfaces within a class*

```
class X
{
   interface A
   {
   }
   static interface B
   {
   }
}
```

As with nested classes, nested interfaces help to implement top-level class architecture by being implemented by nested classes. Collectively, these types are nested because they cannot (as in Listing 4–13's `Iter` local class) or need not appear at the same level as a top-level class and pollute its package namespace.

> **NOTE:** The previous chapter's introduction to interfaces showed you how to declare constants and method headers in the body of an interface. You can also declare interfaces and classes in an interface's body. Because there does not appear to be a good reason to do this, it is probably best to avoid nesting interfaces and/or classes within interfaces.

Packages

Hierarchical structures organize items in terms of hierarchical relationships that exist between those items. For example, a filesystem might contain a `taxes` directory with multiple year subdirectories, where each subdirectory contains tax information pertinent to that year. Also, an enclosing class might contain multiple nested classes that only make sense in the context of the enclosing class.

Hierarchical structures also help to avoid name conflicts. For example, two files cannot have the same name in a nonhierarchical filesystem (which consists of a single directory). In contrast, a hierarchical filesystem lets same-named files exist in different directories. Similarly, two enclosing classes can contain same-named nested classes. Name conflicts do not exist because items are partitioned into different *namespaces*.

Java also supports the partitioning of top-level types into multiple namespaces, to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

This section introduces you to packages. After defining this term and explaining why package names must be unique, the section presents the package and import statements. It next explains how the virtual machine searches for packages and types, and then presents an example that shows you how to work with packages. This section closes by showing you how to encapsulate a package of classfiles into JAR files.

What Are Packages?

A *package* is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages. Only types that are declared `public` can be accessed from outside the package. Furthermore, the constants, constructors, methods, and nested types that describe a class's interface must be declared `public` to be accessible from beyond the package.

Every package has a name, which must be a nonreserved identifier. The member access operator separates a package name from a subpackage name, and separates a package or subpackage name from a type name. For example, the two member access operators in `graphics.shapes.Circle` separate package name `graphics` from the `shapes` subpackage name, and separate subpackage name `shapes` from the `Circle` type name.

> **NOTE:** Each of Java SE's standard class library and Android's class library organizes its many classes and other top-level types into multiple packages. Many of these packages are subpackages of the standard `java` package. Examples include `java.io` (types related to input/output operations), `java.lang` (language-oriented types), `java.lang.reflect` (reflection-oriented language types), `java.net` (network-oriented types), and `java.util` (utility types).

Package Names Must Be Unique

Suppose you have two different `graphics.shapes` packages, and suppose that each `shapes` subpackage contains a `Circle` class with a different interface. When the compiler encounters `System.out.println(new Circle(10.0, 20.0, 30.0).area());` in the source code, it needs to verify that the `area()` method exists.

The compiler will search all accessible packages until it finds a `graphics.shapes` package that contains a `Circle` class. If the found package contains the appropriate `Circle` class with an `area()` method, everything is fine. Otherwise, if the `Circle` class does not have an `area()` method, the compiler will report an error.

This scenario illustrates the importance of choosing unique package names. Specifically, the top-level package name must be unique. The convention in choosing this name is to take your Internet domain name and reverse it. For example, I would choose `ca.mb.javajeff` as my top-level package name because `javajeff.mb.ca` is my domain name. I would then specify `ca.mb.javajeff.graphics.shapes.Circle` to access `Circle`.

> **NOTE:** Reversed Internet domain names are not always valid package names. One or more of its component names might start with a digit (`6.com`), contain a hyphen (`-`) or other illegal character (`aq-x.com`), or be one of Java's reserved words (`int.com`). Convention dictates that you prefix the digit with an underscore (`com._6`), replace the illegal character with an underscore (`com.aq_x`), and suffix the reserved word with an underscore (`com.int_`).

The Package Statement

The package statement identifies the package in which a source file's types are located. This statement consists of reserved word `package`, followed by a member access operator–separated list of package and subpackage names, followed by a semicolon.

For example, `package graphics;` specifies that the source file's types locate in a package named `graphics`, and `package graphics.shapes;` specifies that the source file's types locate in the `graphics` package's `shapes` subpackage.

By convention, a package name is expressed in lowercase. If the name consists of multiple words, each word except for the first word is capitalized.

Only one package statement can appear in a source file. When it is present, nothing apart from comments must precede this statement.

> **CAUTION:** Specifying multiple package statements in a source file or placing anything apart from comments above a package statement causes the compiler to report an error.

Java implementations map package and subpackage names to same-named directories. For example, an implementation would map `graphics` to a directory named

graphics, and would map graphics.shapes to a shapes subdirectory of graphics. The Java compiler stores the classfiles that implement the package's types in the corresponding directory.

> **NOTE:** If a source file does not contain a package statement, the source file's types are said to belong to the *unnamed package*. This package corresponds to the current directory.

The Import Statement

Imagine having to repeatedly specify ca.mb.javajeff.graphics.shapes.Circle or some other lengthy package-qualified type name for each occurrence of that type in source code. Java provides an alternative that lets you avoid having to specify package details. This alternative is the import statement.

The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation. This statement consists of reserved word import, followed by a member access operator–separated list of package and subpackage names, followed by a type name or * (asterisk), followed by a semicolon.

The * symbol is a wildcard that represents all unqualified type names. It tells the compiler to look for such names in the import statement's specified package, unless the type name is found in a previously searched package.

For example, import ca.mb.javajeff.graphics.shapes.Circle; tells the compiler that an unqualified Circle class exists in the ca.mb.javajeff.graphics.shapes package. Similarly, import ca.mb.javajeff.graphics.shapes.*; tells the compiler to look in this package if it encounters a Rectangle class, a Triangle class, or even an Employee class (if Employee has not already been found).

> **TIP:** You should avoid using the * wildcard so that other developers can easily see which types are used in source code.

Because Java is case sensitive, package and subpackage names specified in an import statement must be expressed in the same case as that used in the package statement.

When import statements are present in source code, only a package statement and comments can precede them.

> **CAUTION:** Placing anything other than a package statement, import/static import statements, and comments above an import statement causes the compiler to report an error.

You can run into name conflicts when using the wildcard version of the import statement because any unqualified type name matches the wildcard. For example, you have graphics.shapes and geometry packages that each contain a Circle class, the source

code begins with `import geometry.*;` and `import graphics.shape.*;` statements, and it also contains an unqualified occurrence of `Circle`. Because the compiler does not know if `Circle` refers to `geometry`'s `Circle` class or `graphics.shape`'s `Circle` class, it reports an error. You can fix this problem by qualifying `Circle` with the correct package name.

> **NOTE:** The compiler automatically imports the `String` class and other types from the `java.lang` package, which is why it is not necessary to qualify `String` with `java.lang`.

Searching for Packages and Types

Newcomers to Java who first start to work with packages often become frustrated by "no class definition found" and other errors. This frustration can be partly avoided by understanding how the virtual machine searches for packages and types.

This section explains how the search process works. To understand this process, you need to realize that the compiler is a special Java application that runs under the control of the virtual machine. Furthermore, there are two different forms of search.

Compile-Time Search

When the compiler encounters a type expression (such as a method call) in source code, it must locate that type's declaration to verify that the expression is legal (a method exists in the type's class whose parameter types match the types of the arguments passed in the method call, for example).

The compiler first searches the Java platform packages (which contain class library types). It then searches extension packages (for extension types). If the `-sourcepath` command-line option was specified when starting the virtual machine (via `javac`), the compiler searches the indicated path's source files.

> **NOTE:** Java platform packages are stored in `rt.jar` and a few other important JAR files. Extension packages are stored in a special extensions directory named `ext`.

Otherwise, it searches the user classpath (in left-to-right order) for the first user classfile or source file containing the type. If no user classpath is present, the current directory is searched. If no package matches or the type still cannot be found, the compiler reports an error. Otherwise, the compiler records the package information in the classfile.

> **NOTE:** The user classpath is specified via the `-classpath` option used to start the virtual machine or, if not present, the `CLASSPATH` environment variable.

Runtime Search

When the compiler or any other Java application runs, the virtual machine will encounter types and must load their associated classfiles via special code known as a *classloader*. It will use the previously stored package information that is associated with the encountered type in a search for that type's classfile.

The virtual machine searches the Java platform packages, followed by extension packages, followed by the user classpath (in left-to-right order) for the first classfile that contains the type. If no user classpath is present, the current directory is searched. If no package matches or the type cannot be found, a "no class definition found" error is reported. Otherwise, the classfile is loaded into memory.

> **NOTE:** Whether you use the `-classpath` option or the `CLASSPATH` environment variable to specify a user classpath, there is a specific format that must be followed. Under Windows, this format is expressed as `path1;path2;...`, where `path1`, `path2`, and so on are the locations of package directories. Under Unix and Linux, this format changes to `path1:path2:...`.

Playing with Packages

Suppose your application needs to log messages to the console, to a file, or to another destination (perhaps to an application running on another computer). Furthermore, suppose the application needs to perform some combination of these tasks.

To demonstrate packages, this section presents a simple and reusable logging library. This library consists of an interface named `Logger`, an abstract class named `LoggerFactory`, and a pair of package-private classes named `Console` and `File`.

> **NOTE:** The logging library is an example of the *Abstract Factory design pattern*, which is presented on page 87 of *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612).

Listing 4–16 presents the `Logger` interface, which describes objects that log messages.

Listing 4–16. *Describing objects that log messages via the* `Logger` *interface*

```
package logging;

public interface Logger
{
   boolean connect();
   boolean disconnect();
   boolean log(String msg);
}
```

Each of the `connect()`, `disconnect()`, and `log()` methods returns true upon success, and false upon failure. (Later in this chapter, you will discover a better technique for dealing with failure.)

Listing 4–17 presents the `LoggerFactory` abstract class.

Listing 4–17. *Obtaining a logger for logging messages to a specific destination*

```
package logging;

public abstract class LoggerFactory
{
   public final static int CONSOLE = 0;
   public final static int FILE = 1;

   public static Logger newLogger(int dstType, String...dstName)
   {
      switch (dstType)
      {
         case CONSOLE: return new Console(dstName.length == 0 ? null
                                                              : dstName[0]);
         case FILE   : return new File(dstName.length == 0 ? null
                                                           : dstName[0]);
         default     : return null;
      }
   }
}
```

`newLogger()` returns a `Logger` for logging messages to an appropriate destination. It uses the variable arguments feature to optionally accept an extra `String` argument for those destination types that require the argument. For example, `FILE` requires a filename.

Listing 4–18 presents the package-private `Console` class.

Listing 4–18. *Logging messages to the console*

```
package logging;

class Console implements Logger
{
   private String dstName;
   Console(String dstName)
   {
      this.dstName = dstName;
   }
   public boolean connect()
   {
      return true;
   }
   public boolean disconnect()
   {
      return true;
   }
   public boolean log(String msg)
   {
      System.out.println(msg);
      return true;
   }
}
```

Console's package-private constructor saves its argument, which most likely will be null because there is no need for a String argument. Perhaps a future version of Console will use this argument to identify one of multiple console windows.

Listing 4–19 presents the package-private File class.

Listing 4–19. *Logging messages to a file (eventually)*

```
package logging;

class File implements Logger
{
   private String dstName;
   File(String dstName)
   {
      this.dstName = dstName;
   }
   public boolean connect()
   {
      if (dstName == null)
          return false;
      System.out.println("opening file " + dstName);
      return true;
   }
   public boolean disconnect()
   {
      if (dstName == null)
          return false;
      System.out.println("closing file " + dstName);
      return true;
   }
   public boolean log(String msg)
   {
      if (dstName == null)
          return false;
      System.out.println("writing " + msg + " to file " + dstName);
      return true;
   }
}
```

Unlike Console, File requires a nonnull argument. Each method first verifies that this argument is not null. If the argument is null, the method returns false to signify failure. (In Chapter 10, I refactor File to incorporate appropriate file-writing code.)

The logging library allows us to introduce portable logging code into an application. Apart from a call to newLogger(), this code will remain the same regardless of the logging destination. Listing 4–20 presents an application that tests this library.

Listing 4–20. *Testing the logging library*

```
import logging.*;

public class TestLogger
{
   public static void main(String[] args)
   {
      Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
      if (logger.connect())
```

```
      {
         logger.log("test message #1");
         logger.disconnect();
      }
      else
         System.out.println("cannot connect to console-based logger");

      logger = LoggerFactory.newLogger(LoggerFactory.FILE, "x.txt");
      if (logger.connect())
      {
         logger.log("test message #2");
         logger.disconnect();
      }
      else
         System.out.println("cannot connect to file-based logger");

      logger = LoggerFactory.newLogger(LoggerFactory.FILE);
      if (logger.connect())
      {
         logger.log("test message #3");
         logger.disconnect();
      }
      else
         System.out.println("cannot connect to file-based logger");
   }
}
```

Follow the steps (which assume that the JDK has been installed) to create the `logging` package and `TestLogger` application, and to run this application:

1. Create a new directory and make this directory current.
2. Create a `logging` directory in the current directory.
3. Copy Listing 4–16 to a file named `Logger.java` in the `logging` directory.
4. Copy Listing 4–17 to a file named `LoggerFactory.java` in the `logging` directory.
5. Copy Listing 4–18 to a file named `Console.java` in the `logging` directory.
6. Copy Listing 4–19 to a file named `File.java` in the `logging` directory.
7. Copy Listing 4–20 to a file named `TestLogger.java` in the current directory.
8. Execute `javac TestLogger.java`, which also compiles `logger`'s source files.
9. Execute `java TestLogger`.

After completing the previous step, you should observe the following output from the `TestLogger` application:

```
test message #1
opening file x.txt
writing test message #2 to file x.txt
closing file x.txt
cannot connect to file-based logger
```

What happens when `logging` is moved to another location? For example, move `logging` to the root directory and run `TestLogger`. You will now observe an error message about the virtual machine not finding the `logging` package and its `LoggerFactory` classfile.

You can solve this problem by specifying `-classpath` when running the `java` tool, or by adding the location of the `logging` package to the `CLASSPATH` environment variable. For example, I chose to use `-classpath` in the following Windows-specific command line:

```
java -classpath \;. TestLogger
```

The backslash represents the root directory in Windows. (I could have specified a forward slash as an alternative.) Also, the period represents the current directory. If it is missing, the virtual machine complains about not finding the `TestLogger` classfile.

> **TIP:** If you discover an error message where the virtual machine reports that it cannot find an application classfile, try appending a period character to the classpath. Doing so will probably fix the problem.

Packages and JAR Files

Chapter 1 introduced you to the Java SDK's `jar` tool, which is used to archive classfiles in JAR files, and is also used to extract a JAR file's classfiles. It probably comes as no surprise that you can store packages in JAR files, which greatly simplify the distribution of your package-based class libraries.

> **NOTE:** Java version 7 will introduce *modules* as a replacement to JAR files. Modules address JAR file problems such as a lack of versioning support; no reliable way to express, resolve, and enforce one JAR file's dependency on another JAR file; and having to specify a JAR file as part of the classpath. Because the JAR file's location might change during deployment, developers are forced to correct all references to the JAR file. This new feature will probably include a new reserved word named `module`.

To show you how easy it is to store a package in a JAR file, we will create a `logger.jar` file that contains the `logging` package's four classfiles (`Logger.class`, `LoggerFactory.class`, `Console.class`, and `File.class`). Complete the following steps to accomplish this task:

1. Make sure that the current directory contains the previously created `logging` directory with its four classfiles.
2. Execute `jar cf logger.jar logging\*.class`. You could alternatively execute `jar cf logger.jar logging/*.class`.

You should now find a `logger.jar` file in the current directory. To prove to yourself that this file contains the four classfiles, execute `jar tf logger.jar`.

You can run `TestLogger.class` by adding `logger.jar` to the classpath. For example, you can run `TestLogger` under Windows via `java -classpath logger.jar;. TestLogger`.

Static Imports

An interface should only be used to declare a type. However, some developers violate this principle by using interfaces to only export constants. Such interfaces are known as *constant interfaces*, and Listing 4–21 presents an example.

Listing 4–21. *Declaring a constant interface*

```
public interface Directions
{
   int NORTH = 0;
   int SOUTH = 1;
   int EAST = 2;
   int WEST = 3;
}
```

Developers who resort to constant interfaces do so to avoid having to prefix a constant's name with the name of its class (as in `Math.PI`, where `PI` is a constant in the `java.lang.Math` class). They do this by implementing the interface—see Listing 4–22.

Listing 4–22. *Implementing a constant interface*

```
public class TrafficFlow implements Directions
{
   public static void main(String[] args)
   {
      showDirection((int)(Math.random()*4));
   }
   private static void showDirection(int dir)
   {
      switch (dir)
      {
         case NORTH: System.out.println("Moving north"); break;
         case SOUTH: System.out.println("Moving south"); break;
         case EAST : System.out.println("Moving east"); break;
         case WEST : System.out.println("Moving west");
      }
   }
}
```

Listing 4–22's `TrafficFlow` class implements `Directions` for the sole purpose of not having to specify `Directions.NORTH`, `Directions.SOUTH`, `Directions.EAST`, and `Directions.WEST`.

This is an appalling misuse of an interface. These constants are nothing more than an implementation detail that should not be allowed to leak into the class's exported *interface*, because they might confuse the class's users (what is the purpose of these constants?). Also, they represent a future commitment: even when the class no longer uses these constants, the interface must remain to ensure binary compatibility.

Java version 5 introduced an alternative that satisfies the desire for constant interfaces while avoiding their problems. This static imports feature lets you import a class's `static` members so that you do not have to qualify them with their class names. It is implemented via a small modification to the import statement, as follows:

```
import static packagespec . classname . ( staticmembername | * );
```

The static import statement specifies `static` after `import`. It then specifies a member access operator–separated list of package and subpackage names, which is followed by the member access operator and a class's name. Once again, the member access operator is specified, followed by a single static member name or the asterisk wildcard.

> **CAUTION:** Placing anything apart from a package statement, import/static import statements, and comments above a static import statement causes the compiler to report an error.

You specify a single static member name to import only that name:

```
import static java.lang.Math.PI;  // Import the PI static field only.
import static java.lang.Math.cos; // Import the cos() static method only.
```

In contrast, you specify the wildcard to import all static member names:

```
import static java.lang.Math.*;   // Import all static members from Math.
```

You can now refer to the static member(s) without having to specify the class name:

```
System.out.println(cos(PI));
```

Using multiple static import statements can result in name conflicts, which causes the compiler to report errors. For example, suppose your `geom` package contains a `Circle` class with a static member named `PI`. Now suppose you specify `import static java.lang.Math.*;` and `import static geom.Circle.*;` at the top of your source file. Finally, suppose you specify `System.out.println(PI);` somewhere in that file's code. The compiler reports an error because it does not know if `PI` belongs to `Math` or `Circle`.

Exceptions

In an ideal world, nothing bad ever happens when an application runs. For example, a file always exists when the application needs to open the file, the application is always able to connect to a remote computer, and the virtual machine never runs out of memory when the application needs to instantiate objects.

In contrast, real-world applications occasionally attempt to open files that do not exist, attempt to connect to remote computers that are unable to communicate with them, and require more memory than the virtual machine can provide. Your goal is to write code that properly responds to these and other exceptional situations (exceptions).

This section introduces you to exceptions. After defining this term, the section looks at representing exceptions in source code. It then examines the topics of throwing and

handling exceptions, and concludes by discussing how to perform cleanup tasks before a method returns, whether or not an exception has been thrown.

What Are Exceptions?

An *exception* is a divergence from an application's normal behavior. For example, the application attempts to open a nonexistent file for reading. The normal behavior is to successfully open the file and begin reading its contents. However, the file cannot be read if the file does not exist.

This example illustrates an exception that cannot be prevented. However, a workaround is possible. For example, the application can detect that the file does not exist and take an alternate course of action, which might include telling the user about the problem. Unpreventable exceptions where workarounds are possible must not be ignored.

Exceptions can occur because of poorly written code. For example, an application might contain code that accesses each element in an array. Because of careless oversight, the array-access code might attempt to access a nonexistent array element, which leads to an exception. This kind of exception is preventable by writing correct code.

Finally, an exception might occur that cannot be prevented, and for which there is no workaround. For example, the virtual machine might run out of memory, or perhaps it cannot find a classfile. This kind of exception, known as an *error*, is so serious that it is impossible (or at least inadvisable) to work around; the application must terminate, presenting a message to the user that states why it is terminating.

Representing Exceptions in Source Code

An exception can be represented via error codes or objects. This section discusses each kind of representation and explains why objects are superior. It then introduces you to Java's exception and error class hierarchy, emphasizing the difference between checked and runtime exceptions. It closes by discussing custom exception classes.

Error Codes Versus Objects

One way to represent exceptions in source code is to use error codes. For example, a method might return true on success and false when an exception occurs. Alternatively, a method might return 0 on success and a nonzero integer value that identifies a specific kind of exception.

Developers traditionally designed methods to return error codes; I demonstrated this tradition in each of the three methods in Listing 4–16's `Logger` interface. Each method returns true on success, or returns false to represent an exception (unable to connect to the logger, for example).

Although a method's return value must be examined to see if it represents an exception, error codes are all too easy to ignore. For example, a lazy developer might ignore the

return code from `Logger`'s `connect()` method and attempt to call `log()`. Ignoring error codes is one reason why a new approach to dealing with exceptions has been invented.

This new approach is based on objects. When an exception occurs, an object representing the exception is created by the code that was running when the exception occurred. Details describing the exception's surrounding context are stored in the object. These details are later examined to work around the exception.

The object is then *thrown*, or handed off to the virtual machine to search for a *handler*, code that can handle the exception. (If the exception is an error, the application should not provide a handler.) When a handler is located, its code is executed to provide a workaround. Otherwise, the virtual machine terminates the application.

Apart from being too easy to ignore, an error code's Boolean or integer value is less meaningful than an object name. For example, `fileNotFound` is self-evident, but what does `false` mean? Also, an object can contain information about what led to the exception. These details can be helpful to a suitable workaround.

The Throwable Class Hierarchy

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in `java.lang.Throwable`, the ultimate superclass for all *throwables* (exception and error objects—exceptions and errors, for short—that can be thrown). Table 4–1 identifies and describes most of `Throwable`'s constructors and methods.

Table 4–1. *`Throwable`'s Constructors and Methods*

Method	Description
`Throwable()`	Create a throwable with a null detail message and cause.
`Throwable(String message)`	Create a throwable with the specified detail message and a null cause.
`Throwable(String message, Throwable cause)`	Create a throwable with the specified detail message and cause.
`Throwable(Throwable cause)`	Create a throwable whose detail message is the string representation of a nonnull cause, or null.
`Throwable getCause()`	Return the cause of this throwable. If there is no cause, null is returned.
`String getMessage()`	Return this throwable's detail message, which might be null.
`StackTraceElement[] getStackTrace()`	Provide programmatic access to the stack trace information printed by `printStackTrace()` as an array of stack trace elements, each representing one stack frame.

Method	Description
`Throwable initCause(Throwable cause)`	Initialize the cause of this throwable to the specified value.
`void printStackTrace()`	Print this throwable and its backtrace of stack frames to the standard error stream.

It is not uncommon for a class's public methods to call helper methods that throw various exceptions. A public method will probably not document exceptions thrown from a helper method because they are implementation details that often should not be visible to the public method's caller.

However, because this exception might be helpful in diagnosing the problem, the public method can wrap the lower-level exception in a higher-level exception that is documented in the public method's contract interface. The wrapped exception is known as a *cause* because its existence causes the higher-level exception to be thrown.

When an exception is thrown, it leaves behind a stack of unfinished method calls. Each stack entry is represented by an instance of the `java.lang.StackTraceElement` class. This class's methods provide access to information about a stack entry. For example, `public String getMethodName()` returns the name of an unfinished method.

Moving down the throwable hierarchy, you encounter the `java.lang.Exception` and `java.lang.Error` classes, which respectively represent exceptions and errors. Each class offers four constructors that pass their arguments to their `Throwable` counterparts, but provides no methods apart from those that are inherited from `Throwable`.

`Exception` is itself subclassed by `java.lang.CloneNotSupportedException` (discussed in Chapter 3), `java.lang.IOException` (discussed in Chapter 10), and other classes. Similarly, `Error` is itself subclassed by `java.lang.AssertionError` (discussed in Chapter 5), `java.lang.OutOfMemoryError`, and other classes.

> **CAUTION:** Never instantiate `Throwable`, `Exception`, or `Error`. The resulting objects are meaningless because they are too generic.

Checked Exceptions Versus Runtime Exceptions

A *checked exception* is an exception that represents a problem with the possibility of recovery, and for which the developer must provide a workaround. The developer checks (examines) the code to ensure that the exception is handled in the method where it is thrown, or is explicitly identified as being handled elsewhere.

`Exception` and all subclasses except for `RuntimeException` (and its subclasses) describe checked exceptions. For example, the aforementioned `CloneNotSupportedException` and `IOException` classes describe checked exceptions. (`CloneNotSupportedException` should not be checked because there is no runtime workaround for this kind of exception.)

A *runtime exception* is an exception that represents a coding mistake. This kind of exception is also known as an *unchecked exception* because it does not need to be handled or explicitly identified—the mistake must be fixed. Because these exceptions can occur in many places, it would be burdensome to be forced to handle them.

`RuntimeException` and its subclasses describe unchecked exceptions. For example, `java.lang.ArithmeticException` describes arithmetic problems such as integer division by zero. Another example is `java.lang.ArrayIndexOutOfBoundsException`. (In hindsight, `RuntimeException` should have been named `UncheckedException` because all exceptions occur at runtime.)

> **NOTE:** Many developers are not happy with checked exceptions because of the work involved in having to handle them. This problem is made worse by libraries providing methods that throw checked exceptions when they should throw unchecked exceptions. As a result, many modern languages support only unchecked exceptions.

Custom Exception Classes

You can declare your own exception classes. Before doing so, ask yourself if an existing exception class in Java's class library meets your needs. If you find a suitable class, you should reuse it. (Why reinvent the wheel?) Other developers will already be familiar with the existing class, and this knowledge will make your code easier to learn.

If no existing class meets your needs, think about whether to subclass `Exception` or `RuntimeException`. In other words, will your exception class be checked or unchecked? As a rule of thumb, your class should subclass `RuntimeException` if you think that it will describe a coding mistake.

> **TIP:** When you name your class, follow the convention of providing an `Exception` suffix. This suffix clarifies that your class describes an exception.

Suppose you are creating a `Media` class whose static methods perform various media-oriented utility tasks. For example, one method converts files in non-MP3 media formats to MP3 format. This method will be passed source file and destination file arguments, and will convert the source file to the format implied by the destination file's extension.

Before performing the conversion, the method needs to verify that the source file's format agrees with the format implied by its file extension. If there is no agreement, an exception must be thrown. Furthermore, this exception must store the expected and existing media formats so that a handler can identify them in a message to the user.

Because Java's class library does not provide a suitable exception class, you decide to introduce a class named `InvalidMediaFormatException`. Detecting an invalid media format is not the result of a coding mistake, and so you also decide to extend `Exception` to indicate that the exception is checked. Listing 4–23 presents this class's declaration.

Listing 4–23. *Declaring a custom exception class*

```
public class InvalidMediaFormatException extends Exception
{
   private String expectedFormat;
   private String existingFormat;
   public InvalidMediaFormatException(String expectedFormat,
                                      String existingFormat)
   {
      super("Expected format: " + expectedFormat + ", Existing format: " +
            existingFormat);
      this.expectedFormat = expectedFormat;
      this.existingFormat = existingFormat;
   }
   public String getExpectedFormat()
   {
      return expectedFormat;
   }
   public String getExistingFormat()
   {
      return existingFormat;
   }
}
```

`InvalidMediaFormatException` provides a constructor that calls `Exception`'s `public Exception(String message)` constructor with a detail message that includes the expected and existing formats. It is wise to capture such details in the detail message because the problem that led to the exception might be hard to reproduce.

`InvalidMediaFormatException` also provides `getExpectedFormat()` and `getExistingFormat()` methods that return these formats. Perhaps a handler will present this information in a message to the user. Unlike the detail message, this message might be *localized*, expressed in the user's language (French, German, English, and so on).

Throwing Exceptions

Now that you have created an `InvalidMediaFormatException` class, you can declare the `Media` class and begin to code its `convert()` method. The initial version of this method validates its arguments, and then verifies that the source file's media format agrees with the format implied by its file extension. Check out Listing 4–24.

Listing 4–24. *Throwing exceptions from the `convert()` method*

```
public static void convert(String srcName, String dstName)
   throws InvalidMediaFormatException
{
   if (srcName == null)
      throw new NullPointerException(srcName + " is null");
   if (dstName == null)
      throw new NullPointerException(dstName + " is null");
   // Code to access source file and verify that its format matches the
   // format implied by its file extension.
   //
   // Assume that the source file's extension is RM (for Real Media) and
   // that the file's internal signature suggests that its format is
```

```
   // Microsoft WAVE.
   String expectedFormat = "RM";
   String existingFormat = "WAVE";
   throw new InvalidMediaFormatException(expectedFormat, existingFormat);
}
```

Listing 4–24 demonstrates a throws clause, which consists of reserved word `throws` followed by a comma-separated list of checked exception class names, and which is appended to a method header. This clause identifies all checked exceptions that are thrown out of the method, and which must be handled by some other method.

Listing 4–24 also demonstrates the throw statement, which consists of reserved word `throw` followed by an instance of `Throwable` or a subclass. (You typically instantiate an `Exception` subclass.) This statement throws the instance to the virtual machine, which then searches for a suitable handler to handle the exception.

The first use of the throw statement is to throw a `java.lang.NullPointerException` instance when a null reference is passed as the source or destination filename. This unchecked exception is commonly thrown to indicate that a contract has been violated via a passed null reference. For example, you cannot pass null filenames to `convert()`.

The second use of the throw statement is to throw an `InvalidMediaFormatException` instance when the expected media format does not match the existing format. In the contrived example, the exception is thrown because the expected format is RM and the existing format is WAVE.

Unlike `InvalidMediaFormatException`, `NullPointerException` is not listed in `convert()`'s throws clause because `NullPointerException` instances are unchecked. They can occur so frequently that it is too big a burden to force the developer to properly handle these exceptions. Instead, the developer should write code that minimizes their occurrences.

`NullPointerException` is one kind of exception that is thrown when an argument proves to be invalid. The `java.lang.IllegalArgumentException` class generalizes the illegal argument scenario to include other kinds of illegal arguments. For example, Listing 4–25 throws an `IllegalArgumentException` instance when a numeric argument is negative.

Listing 4–25. *Throwing an `IllegalArgumentException` instance when `x` is negative (you can't calculate a negative number's square root)*

```
public static double sqrt(double x)
{
   if (x < 0)
      throw new IllegalArgumentException(x + " is negative");
   // Calculate the square root of x.
}
```

There are a few additional items to keep in mind when working with throws clauses and throw statements:

- You can append a throws clause to a constructor and throw an exception from the constructor when something goes wrong while the constructor is executing. The resulting object will not be created.

- When an exception is thrown out of an application's `main()` method, the virtual machine terminates the application and calls the exception's `printStackTrace()` method to print, to the console, the sequence of nested method calls that was awaiting completion when the exception was thrown.
- If a superclass method declares a throws clause, the overriding subclass method does not have to declare a throws clause. However, if it does declare a throws clause, the clause must not include the names of exception classes that are not also included in the superclass method's throws clause.
- A checked exception class name does not need to appear in a throws clause when the name of its superclass appears.
- The compiler reports an error when a method throws a checked exception and does not also handle the exception or list the exception in its throws clause.
- Do not include the names of unchecked exception classes in a throws clause. These names are not required because such exceptions should never occur. Furthermore, they only clutter source code, and possibly confuse someone who is trying to understand that code.
- You can declare a checked exception class name in a method's throws clause without throwing an instance of this class from the method. Perhaps the method has yet to be fully coded.

Handling Exceptions

A method indicates its intention to handle one or more exceptions by specifying a try statement and one or more appropriate catch clauses. The try statement consists of reserved word `try` followed by a brace-delimited body. You place code that throws exceptions into this body.

A catch clause consists of reserved word `catch`, followed by a round bracket–delimited single-parameter list that specifies an exception class name, followed by a brace-delimited body. You place code that handles exceptions whose types match the type of the catch clause's parameter list's exception class parameter in this body.

A catch clause is specified immediately after a try statement's body. When an exception is thrown, the virtual machine searches for a handler by first examining the catch clause to see if its parameter type matches or is the superclass type of the exception that has been thrown.

If the catch clause is found, its code executes and the exception is handled. Otherwise, the virtual machine proceeds up the method-call stack, looking for the first method whose try statement contains an appropriate catch clause. This process continues unless a catch clause is found or execution leaves the `main()` method.

Listing 4–26 illustrates try and catch.

Listing 4–26. *Handling a thrown exception*

```
public static void main(String[] args)
{
   if (args.length != 2)
   {
      System.err.println("usage: java Converter srcfile dstfile");
      return;
   }
   try
   {
      Media.convert(args[0], args[1]);
   }
   catch (InvalidMediaFormatException imfe)
   {
      System.out.println("Unable to convert " + args[0] + " to " + args[1]);
      System.out.println("Expecting " + args[0] + " to conform to " +
                         imfe.getExpectedFormat() + " format.");
      System.out.println("However, " + args[0] + " conformed to " +
                         imfe.getExistingFormat() + " format.");
   }
}
```

`Media`'s `convert()` method is placed in a try statement's body because this method is capable of throwing an instance of the checked `InvalidMediaFormatException` class—checked exceptions must be handled or be declared to be thrown via a throws clause that is appended to the method.

A catch clause immediately follows try's body. This clause presents a parameter list whose single parameter matches the type of the `InvalidMediaFormatException` object thrown from `convert()`. When the object is thrown, the virtual machine will transfer execution to the statements within this clause.

> **TIP:** You might want to name your catch clause parameters using the abbreviated style shown in Listing 4–26. Not only does this convention result in more meaningful exception-oriented parameter names (`imfe` indicates that an `InvalidMediaFormatException` has been thrown), it will probably reduce compiler errors.
>
> It is common practice to name a catch clause's parameter `e`, for convenience. (Why type a long name?) However, the compiler will report an error when a previously declared local variable or parameter also uses `e` as its name—multiple same-named local variables and parameters cannot exist in the same scope.

The catch clause's statements are designed to provide a descriptive error message to the user. A more sophisticated application would localize these names so that the user could read the message in the user's language. The developer-oriented detail message is not output because it is not necessary in this trivial application.

> **NOTE:** A developer-oriented detail message is typically not localized. Instead, it is expressed in the developer's language. Users should never see detail messages.

You can specify multiple catch clauses after try's body. For example, a later version of `convert()` will also throw `java.io.FileNotFoundException` when it cannot open the source file or create the destination file, and `IOException` when it cannot read from the source file or write to the destination file. All of these exceptions must be handled.

Listing 4–27 illustrates multiple catch clauses.

Listing 4–27. *Handling more than one thrown exception*

```
try
{
   Media.convert(args[0], args[1]);
}
catch (InvalidMediaFormatException imfe)
{
   System.out.println("Unable to convert " + args[0] + " to " + args[1]);
   System.out.println("Expecting " + args[0] + " to conform to " +
                      imfe.getExpectedFormat() + " format.");
   System.out.println("However, " + args[0] + " conformed to " +
                      imfe.getExistingFormat() + " format.");
}
catch (FileNotFoundException fnfe)
{
}
catch (IOException ioe)
{
}
```

Listing 4–27 assumes that `convert()` also throws `IOException` and `FileNotFoundException`. Although this assumption suggests that both classes need to be listed in `convert()`'s throws clause, only `IOException` needs to be listed because it is the superclass of `FileNotFoundException`.

> **CAUTION:** The compiler reports an error when you specify two or more catch clauses with the same parameter type after a try body. Example: `try {} catch (IOException ioe1) {} catch (IOException ioe2) {}`. You must merge these catch clauses into one clause.

Catch clauses often can be specified in any order. However, the compiler restricts this order when one catch clause's parameter is a supertype of another catch clause's parameter. The subtype parameter catch clause must precede the supertype parameter catch clause; otherwise, the subtype parameter catch clause will never be called.

For example, the `FileNotFoundException` catch clause must precede the `IOException` catch clause. If the compiler allowed the `IOException` catch clause to be specified first, the `FileNotFoundException` catch clause would never execute because a `FileNotFoundException` instance is also an instance of its `IOException` superclass.

NOTE: Java version 7 introduces a catch clause improvement known as *multicatch*, which lets you place common exception-handling code in a single catch clause. For example, `catch (InvalidMediaFormatException | UnsupportedMediaFormatException ex) { /* common code */ }` handles `InvalidMediaFormatException` and a similar `UnsupportedMediaFormatException` in one place.

Multicatch is not always necessary. For example, you do not need to specify `catch (FileNotFoundException | IOException exc) { /* suitable common code */ }` to handle `FileNotFoundException` and `IOException` because `catch (IOException ioe)` accomplishes the same task, by catching `FileNotFoundException` as well as `IOException`.

The empty `FileNotFoundException` and `IOException` catch clauses illustrate the often-seen problem of leaving catch clauses empty because they are inconvenient to code. Unless you have a good reason, do not create an empty catch clause. It swallows exceptions and you do not know that the exceptions were thrown.

CAUTION: Do not code empty catch clauses. Because they swallow exceptions, you will probably find it more difficult to debug a faulty application.

While discussing the `Throwable` class, I discussed wrapping lower-level exceptions in higher-level exceptions. This activity will typically take place in a catch clause, and is illustrated in Listing 4–28.

Listing 4–28. *Throwing a new exception that contains a wrapped exception*

```
catch (IOException ioe)
{
   throw new ReportCreationException(ioe);
}
```

This example assumes that a helper method has just thrown a generic `IOException` as the result of trying to create a report. The public method's contract states that `ReportCreationException` is thrown in this case. To satisfy the contract, the latter exception is thrown. To satisfy the developer who is responsible for debugging a faulty application, the `IOException` instance is wrapped inside the `ReportCreationException` instance that is thrown to the public method's caller.

Sometimes, a catch clause might not be able to fully handle an exception. Perhaps it needs access to information provided by some ancestor method in the method-call stack. However, the catch clause might be able to partly handle the exception. In this case, it should partly handle the exception, and then rethrow the exception so that a handler in the ancestor method can finish handling the exception. This scenario is demonstrated in Listing 4–29.

Listing 4–29. *Rethrowing an exception*

```
catch (FileNotFoundException fnfe)
{
   // Provide code to partially handle the exception here.
   throw fnfe; // Rethrow the exception here.
}
```

NOTE: Java version 7 introduces a catch clause improvement known as *final rethrow*, which lets you declare a catch clause parameter `final` in order to throw only those checked exception types that were thrown in the try body, are a subtype of the catch parameter type, and are not caught in preceding catch clauses. For example, suppose you declare the following method:

```
void method() throws Exc1, Exc2 // Exc1 and Exc2 extend Exception
{
   try
   {
      /* Code that can throw Exc1,Exc2 */
   }
   catch (Exception exc)
   {
      logger.log(exc);
      throw exc; // Attempt to throw caught exception as an Exception
   }
}
```

The compiler would report an error when asked to compile this method because you are trying to rethrow an exception that is first upcasted to `Exception`, but `Exception` is not listed in `method()`'s throws clause. However, if you change the catch clause header to `catch (final Exception exc)`, the compiler will not report an error because you are rethrowing `Exc1` or `Exc2` exceptions without the upcasting.

Performing Cleanup

In some situations, you might want to prevent an exception from being thrown out of a method before the method's cleanup code is executed. For example, you might want to close a file that was opened, but could not be written, possibly because of insufficient disk space. Java provides the finally clause for this situation.

The finally clause consists of reserved word `finally` followed by a body, which provides the cleanup code. A finally clause follows either a catch clause or a try body. In the former case, the exception is handled (and possibly rethrown) before finally executes. In the latter case, finally executes before the exception is thrown and handled.

Listing 4–30 demonstrates the finally clause in the context of a file-copying application's `main()` method.

Listing 4–30. *Cleaning up after handling a thrown exception*

```
public static void main(String[] args)
{
   if (args.length != 2)
   {
      System.err.println("usage: java Copy srcfile dstfile");
      return;
   }
   FileInputStream fis = null;
   try
   {
      fis = new FileInputStream(args[0]);
      FileOutputStream fos = null;
      try
      {
         fos = new FileOutputStream(args[1]);
         int b; // I chose b instead of byte because byte is a reserved word.
         while ((b = fis.read()) != -1)
            fos.write(b);
      }
      catch (FileNotFoundException fnfe)
      {
         String msg = args[1] + " could not be created, possibly because " +
                      "it might be a directory";
         System.err.println(msg);
      }
      catch (IOException ioe)
      {
         String msg = args[0] + " could not be read, or " + args[1] +
                      " could not be written";
         System.err.println(msg);
      }
      finally
      {
         if (fos != null)
            try
            {
               fos.close();
            }
            catch (IOException ioe)
            {
               System.err.println("unable to close " + args[1]);
            }
      }
   }
   catch (FileNotFoundException fnfe)
   {
      String msg = args[0] + " could not be found or might be a directory";
      System.err.println(msg);
   }
   finally
   {
      if (fis != null)
         try
```

```
            {
               fis.close();
            }
            catch (IOException ioe)
            {
               System.err.println("unable to close " + args[0]);
            }
      }
}
```

> **NOTE:** Do not be concerned if you find this listing's file-oriented code difficult to grasp; I will formally introduce I/O and the listing's file-oriented types in Chapter 10. I'm presenting this code here because file copying provides a perfect example of the finally clause.

Listing 4–30 presents an application that copies bytes from a source file to a destination file via a nested pair of try bodies. The outer try body uses a `FileInputStream` object to open the source file for reading; the inner try body uses a `FileOutputStream` object to create the destination file for writing, and also contains the file-copying code.

If the `fis = new FileInputStream(args[0]);` expression throws `FileNotFoundException`, execution flows into the outer try statement's `catch (FileNotFoundException fnfe)` clause, which outputs a suitable message to the user. Execution then enters the outer try statement's finally clause.

The outer try statement's finally clause closes an open source file. However, when `FileNotFoundException` is thrown, the source file is not open—no reference was assigned to `fis`. The finally clause uses `if (fis != null)` to detect this situation, and does not attempt to close the file.

If `fis = new FileInputStream(args[0]);` succeeds, execution flows into the inner try statement, whose body executes `fos = new FileOutputStream(args[1]);`. If this expression throws `FileNotFoundException`, execution moves into the inner try's `catch (FileNotFoundException fnfe)` clause, which outputs a suitable message to the user.

This time, execution continues with the inner try statement's finally clause. Because the destination file was not created, no attempt is made to close this file. In contrast, the open source file must be closed, and this is accomplished when execution moves from the inner finally clause to the outer finally clause.

`FileInputStream`'s and `FileOutputStream`'s `close()` methods throw `IOException` when a file is not open. Because `IOException` is checked, these exceptions must be handled; otherwise, it would be necessary to append a `throws IOException` clause to the `main()` method header.

You can specify a try statement with only a finally clause. You would do so when you are not prepared to handle an exception in the enclosing method (or enclosing try statement, if present), but need to perform cleanup before the thrown exception causes execution to leave the method. Listing 4–31 provides a demonstration.

Listing 4–31. *Cleaning up before handling a thrown exception*

```
public static void main(String[] args)
{
   if (args.length != 2)
   {
      System.err.println("usage: java Copy srcfile dstfile");
      return;
   }
   try
   {
      copy(args[0], args[1]);
   }
   catch (FileNotFoundException fnfe)
   {
      String msg = args[0] + " could not be found or might be a directory," +
                   " or " + args[1] + " could not be created, " +
                   "possibly because " + args[1] + " is a directory";
      System.err.println(msg);
   }
   catch (IOException ioe)
   {
      String msg = args[0] + " could not be read, or " + args[1] +
                   " could not be written";
      System.err.println(msg);
   }
}
static void copy(String srcFile, String dstFile) throws IOException
{
   FileInputStream fis = new FileInputStream(srcFile);
   try
   {
      FileOutputStream fos = new FileOutputStream(dstFile);
      try
      {
         int b;
         while ((b = fis.read()) != -1)
            fos.write(b);
      }
      finally
      {
         try
         {
            fos.close();
         }
         catch (IOException ioe)
         {
            System.err.println("unable to close " + dstFile);
         }
      }
   }
   finally
   {
      try
      {
         fis.close();
      }
```

```
      catch (IOException ioe)
      {
         System.err.println("unable to close " + srcFile);
      }
   }
}
```

Listing 4–31 provides an alternative to Listing 4–30 that attempts to be more readable. It accomplishes this task by introducing a `copy()` method that uses a nested pair of try-finally constructs to perform the file-copy operation, and also close each open file whether an exception is or is not thrown.

If the `FileInputStream fis = new FileInputStream(srcFile);` expression results in a thrown `FileNotFoundException`, execution leaves `copy()` without entering the outer try statement. This statement is only entered after the `FileInputStream` object has been created, indicating that the source file was opened.

If the `FileOutputStream fos = new FileOutputStream(dstFile);` expression results in a thrown `FileNotFoundException`, execution leaves `copy()` without entering the inner try statement. However, execution leaves `copy()` only after entering the finally clause that is mated with the outer try statement. This clause closes the open source file.

If the `read()` or `write()` method in the inner try statement's body throws an `IOException` object, the finally clause associated with the inner try statement is executed. This clause closes the open destination file. Execution then flows into the outer finally clause, which closes the open source file, and continues on out of `copy()`.

> **CAUTION:** If the body of a try statement throws an exception, and if the finally clause results in another exception being thrown, this new exception replaces the previous exception, which is lost.

Despite Listing 4–31 being somewhat more readable than Listing 4–30, there is still too much boilerplate thanks to each finally clause requiring a try statement to close a file. This boilerplate is necessary; its removal results in a new `IOException` possibly being thrown from the catch clause, which would mask a previously thrown `IOException`.

> **NOTE:** Java version 7 introduces *automatic resource management* to eliminate the boilerplate associated with closing files and other resources. Furthermore, this feature can eliminate bugs that arise from masking thrown exceptions with other exceptions.
>
> Automatic resource management consists of a new `Disposable` interface that resource classes (such as `FileInputStream`) implement, and syntactic sugar that associates a semicolon-separated list of resource class instantiations with `try`.
>
> For example, automatic resource management can turn Listing 4–31's `copy()` method into the following shorter method:

```
static void copy(String srcFile, String dstFile) throws IOException
{
   try (FileInputStream fis = new FileInputStream(srcFile);
        FileOutputStream fos = new FileOutputStream(dstFile))
   {
      int b;
      while ((b = fis.read()) != -1)
         fos.write(b);
   }
}
```

EXERCISES

The following exercises are designed to test your understanding of nested types, packages, static imports, and exceptions:

1. What is a nested class?
2. Identify the four kinds of nested classes.
3. Which nested classes are also known as inner classes?
4. True or false: A static member class has an enclosing instance.
5. How do you instantiate a nonstatic member class from beyond its enclosing class?
6. When is it necessary to declare local variables and parameters `final`?
7. True or false: An interface can be declared within a class or within another interface.
8. What is a package?
9. How do you ensure that package names are unique?
10. What is a package statement?
11. True or false: You can specify multiple package statements in a source file.
12. What is an import statement?
13. How do you indicate that you want to import multiple types via a single import statement?
14. During a runtime search, what happens when the virtual machine cannot find a classfile?
15. How do you specify the user classpath to the virtual machine?
16. What is a constant interface?
17. Why are constant interfaces used?
18. Why are constant interfaces bad?

19. What is a static import statement?
20. How do you specify a static import statement?
21. What is an exception?
22. In what ways are objects superior to error codes for representing exceptions?
23. What is a throwable?
24. What does the `getCause()` method return?
25. What is the difference between `Exception` and `Error`?
26. What is a checked exception?
27. What is a runtime exception?
28. Under what circumstance would you introduce your own exception class?
29. True or false: You use a throw statement to identify exceptions that are thrown from a method by appending this statement to a method's header.
30. What is the purpose of a try statement, and what is the purpose of a catch clause?
31. What is the purpose of a finally clause?
32. A 2D graphics package supports two-dimensional drawing and transformations (rotation, scaling, translation, and so on). These transformations require a 3-by-3 matrix (a table). Declare a `G2D` class that encloses a private `Matrix` nonstatic member class. Instantiate `Matrix` within `G2D`'s noargument constructor, and initialize the `Matrix` instance to the *identity matrix* (a matrix where all entries are 0 except for those on the upper-left to lower-right diagonal, which are 1).
33. Extend the `logging` package to support a null device in which messages are thrown away.
34. Modify the `logging` package so that `Logger`'s `connect()` method throws `CannotConnectException` when it cannot connect to its logging destination, and the other two methods each throw `NotConnectedException` when `connect()` was not called or when it threw `CannotConnectException`.
35. Modify `TestLogger` to respond appropriately to thrown `CannotConnectException` and `NotConnectedException` objects.

Summary

Classes that are declared outside of any class are known as top-level classes. Java also supports nested classes, which are classes declared as members of other classes or scopes.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as inner classes.

Java supports the partitioning of top-level types into multiple namespaces, to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

The package statement identifies the package in which a source file's types are located. The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation.

An exception is a divergence from an application's normal behavior. Although it can be represented by an error code or object, Java uses objects because error codes are meaningless and cannot contain information about what led to the exception.

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in `Throwable`. Moving down the throwable hierarchy, you encounter the `Exception` and `Error` classes, which represent nonerror exceptions and errors.

`Exception` and its subclasses, except for `RuntimeException` (and its subclasses), describe checked exceptions. They are checked because you must check the code to ensure that an exception is handled where thrown or identified as being handled elsewhere.

`RuntimeException` and its subclasses describe unchecked exceptions. You do not have to handle these exceptions because they represent coding mistakes (fix the mistakes). Although the names of their classes can appear in throws clauses, doing so adds clutter.

The throw statement throws an exception to the virtual machine, which searches for an appropriate handler. If the exception is checked, its name must appear in the method's throws clause, unless the name of the exception's superclass is listed in this clause.

A method handles one or more exceptions by specifying a try statement and appropriate catch clauses. A finally clause can be included to execute cleanup code whether an exception is thrown or not, and before a thrown exception leaves the method.

Now that you have mastered the advanced language features related to nested types, packages, static imports, and exceptions, you can leverage this knowledge in Chapter 5, where you explore features related to assertions, annotations, generics, and enums.