Chapter 5

Mastering Advanced Language Features Part 2

Chapters 2 and 3 laid a foundation for learning the Java language, and Chapter 4 built onto this foundation by introducing some of Java's more advanced language features. Chapter 5 continues to cover advanced language features by focusing on those features related to assertions, annotations, generics, and enums.

Assertions

Writing source code is not an easy task. All too often, *bugs* (defects) are introduced into the code. When a bug is not discovered before compiling the source code, it makes it into runtime code, which will probably fail unexpectedly. At this point, the cause of failure can be very difficult to determine.

Developers often make assumptions about application correctness, and some developers think that specifying comments that state their beliefs about what they think is true at the comment locations is sufficient for determining correctness. However, comments are useless for preventing bugs because the compiler ignores them.

Many languages address this problem by providing an assertions language feature that lets the developer codify assumptions about application correctness. When the application runs, if an assertion fails, the application terminates with a message that helps the developer diagnose the failure's cause.

This section introduces you to Java's assertions language feature. After defining this term, showing you how to declare assertions, and providing examples, the section looks at using and avoiding assertions. Finally, you learn how to selectively enable and disable assertions via the Java SE 6u16 `javac` compiler tool's command-line arguments.

Declaring Assertions

An *assertion* is a statement that lets you express an assumption of program correctness via a Boolean expression. If this expression evaluates to true, execution continues with the next statement. Otherwise, an error that identifies the cause of failure is thrown.

There are two forms of the assertion statement, with each form beginning with reserved word `assert`:

```
assert expression1 ;
assert expression1 : expression2 ;
```

In both forms of this statement, *`expression1`* is the Boolean expression. In the second form, *`expression2`* is any expression that returns a value. It cannot be a call to a method whose return type is `void`.

When *`expression1`* evaluates to false, this statement instantiates the `AssertionError` class. The first statement form calls this class's noargument constructor, which does not associate a message identifying failure details with the `AssertionError` instance.

The second form calls an `AssertionError` constructor whose type matches the type of *`expression2`*'s value. This value is passed to the constructor and its string representation is used as the error's detail message.

When the error is thrown, the name of the source file and the number of the line from where the error was thrown are output to the console as part of the thrown error's stack trace. In many situations, this information is sufficient for identifying what led to the failure, and the first form of the assertion statement should be used.

Listing 5–1 demonstrates the first form of the assertion statement.

Listing 5–1. *Throwing an assertion error without a detail message*

```
public class AssertionDemo
{
   public static void main(String[] args)
   {
      int x = 1;
      assert x == 0;
   }
}
```

When assertions are enabled (I discuss this task later), running the previous application results in the following output:

```
Exception in thread "main" java.lang.AssertionError
        at AssertionDemo.main(AssertionDemo.java:6)
```

In other situations, more information is needed to help diagnose the cause of failure. For example, suppose *`expression1`* compares variables x and y, and throws an error when x's value exceeds y's value. Because this should never happen, you would probably use the second statement form to output these values so you could diagnose the problem.

Listing 5–2 demonstrates the second form of the assertion statement.

Listing 5–2. *Throwing an assertion error with a detail message*

```
public class AssertionDemo
{
   public static void main(String[] args)
   {
      int x = 1;
      assert x == 0: x;
   }
}
```

Once again, it is assumed that assertions are enabled. Running the previous application results in the following output:

```
Exception in thread "main" java.lang.AssertionError: 1
        at AssertionDemo.main(AssertionDemo.java:6)
```

The value in x is appended to the end of the first output line, which is somewhat cryptic. To make this output more meaningful, you might want to specify an expression that also includes the variable's name: `assert x == 0: "x = " + x;`, for example.

Using Assertions

There are many situations where assertions should be used. These situations organize into internal invariant, control-flow invariant, and design-by-contract categories. An *invariant* is something that does not change.

Internal Invariants

An *internal invariant* is expression-oriented behavior that is not expected to change. For example, Listing 5–3 introduces an internal invariant by way of chained if-else statements that output the state of water based on its temperature.

Listing 5–3. *Discovering that an internal invariant can vary*

```
public class IIDemo
{
   public static void main(String[] args)
   {
      double temperature = 50.0; // Celsius
      if (temperature < 0.0)
         System.out.println("water has solidified");
      else
      if (temperature >= 100.0)
         System.out.println("water is boiling into a gas");
      else
      {
         // temperature > 0.0 and temperature < 100.0
         assert(temperature > 0.0 && temperature < 100.0): temperature;
         System.out.println("water is remaining in its liquid state");
      }
   }
}
```

A developer might specify only a comment stating an assumption as to what expression causes the final `else` to be reached. Because the comment might not be enough to detect the lurking `< 0.0` expression bug, an assertion statement is necessary.

Another example of an internal invariant concerns a switch statement with no default case. The default case is avoided because the developer believes that all paths have been covered. However, this is not always true, as Listing 5–4 demonstrates.

Listing 5–4. *Another buggy internal invariant*

```
public class IIDemo
{
   final static int NORTH = 0;
   final static int SOUTH = 1;
   final static int EAST = 2;
   final static int WEST = 3;
   public static void main(String[] args)
   {
      int direction = (int)(Math.random()*5);
      switch (direction)
      {
         case NORTH: System.out.println("travelling north"); break;
         case SOUTH: System.out.println("travelling south"); break;
         case EAST : System.out.println("travelling east"); break;
         case WEST : System.out.println("travelling west"); break;
         default   : assert false;
      }
   }
}
```

Listing 5–4 assumes that the expression tested by switch will only evaluate to one of four integer constants. However, `(int)(Math.random()*5)` can also return 4, causing the default case to execute `assert false;`, which always throws `AssertionError`. (You might have to run this application a few times to see the assertion error, but first you need to learn how to enable assertions, which I discuss later in this chapter.)

> **TIP:** When assertions are disabled, `assert false;` does not execute and the bug goes undetected. To always detect this bug, replace `assert false;` with `throw new AssertionError(direction);`.

Control-Flow Invariants

A *control-flow invariant* is a flow of control that is not expected to change. For example, Listing 5–4 uses an assertion to test an assumption that switch's default case will not execute. Listing 5–5, which fixes Listing 5–4's bug, provides another example.

Listing 5–5. *A buggy control-flow invariant*

```
public class CFDemo
{
   final static int NORTH = 0;
   final static int SOUTH = 1;
```

```
   final static int EAST = 2;
   final static int WEST = 3;
   public static void main(String[] args)
   {
      int direction = (int)(Math.random()*4);
      switch (direction)
      {
         case NORTH: System.out.println("travelling north"); break;
         case SOUTH: System.out.println("travelling south"); break;
         case EAST : System.out.println("travelling east"); break;
         case WEST : System.out.println("travelling west");
         default   : assert false;
      }
   }
}
```

Because the original bug has been fixed, the default case should never be reached. However, the omission of a break statement that terminates `case WEST` causes execution to reach the default case. This control-flow invariant has been broken. (Again, you might have to run this application a few times to see the assertion error, but first you need to learn how to enable assertions, which I discuss later in this chapter.)

> **CAUTION:** Be careful when using an assertion statement to detect code that should never be executed. If the assertion statement cannot be reached according to the rules set forth in *The Java Language Specification, Third Edition,* by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha (Addison-Wesley, 2005; ISBN: 0321246780) (also available at `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`), the compiler will report an error. For example, `for(;;); assert false;` causes the compiler to report an error because the infinite for loop prevents the assertion statement from executing.

Design-by-Contract

Design-by-Contract is a way to design software based on preconditions, postconditions, and invariants (internal, control-flow, and class). Assertion statements support an informal design-by-contract style of development.

Preconditions

A *precondition* is something that must be true when a method is called. Assertion statements are often used to satisfy a helper method's preconditions by checking that its arguments are legal. Listing 5–6 provides an example.

Listing 5–6. *Verifying a precondition*

```
public class Lotto649
{
   public static void main(String[] args)
   {
      // Lotto 649 requires that six unique numbers be chosen.
```

```
      int[] selectedNumbers = new int[6];

      // Assign a unique random number from 1 to 49 (inclusive) to each slot
      // in the selectedNumbers array.
      for (int slot = 0; slot < selectedNumbers.length; slot++)
      {
           int num;

           // Obtain a random number from 1 to 49. That number becomes the
           // selected number if it has not previously been chosen.
           try_again:
           do
           {
               num = rnd(49)+1;
               for (int i = 0; i < slot; i++)
                    if (selectedNumbers[i] == num)
                        continue try_again;
               break;
           }
           while (true);

           // Assign selected number to appropriate slot.
           selectedNumbers[slot] = num;
      }

      // Sort all selected numbers into ascending order and then print these
      // numbers.
      sort(selectedNumbers);
      for (int i = 0; i < selectedNumbers.length; i++)
           System.out.print(selectedNumbers[i] + " ");
   }
   private static int rnd(int limit)
   {
      // This method returns a random number (actually, a pseudorandom number)
      // ranging from 0 through limit-1 (inclusive).
      assert limit > 1: "limit = " + limit;
      return (int)(Math.random()*limit);
   }
   private static void sort(int[] x)
   {
      // This method sorts the integers in the passed array into ascending
      // order.
      for (int pass = 0; pass < x.length-1; pass++)
         for (int i = x.length-1; i > pass; i--)
            if (x[i] < x[pass])
            {
               int temp = x[i];
               x[i] = x[pass];
               x[pass] = temp;
            }
   }
}
```

Listing 5–6's application simulates Lotto 6/49, one of Canada's national lottery games. The `rnd()` helper method returns a randomly chosen integer between 0 and `limit`-1. An assertion statement verifies the precondition that `limit`'s value must be 2 or higher.

NOTE: The `sort()` helper method *sorts* (orders) the `selectedNumbers` array's integers into ascending order by implementing an *algorithm* (a recipe for accomplishing some task) called *Bubble Sort*.

Bubble Sort works by making multiple passes over the array. During each pass, various comparisons and swaps ensure that the next smallest element value "bubbles" toward the top of the array, which would be the element at index 0.

Bubble Sort is not efficient, but is more than adequate for sorting a six-element array. Although I could have used one of the efficient `sort()` methods located in the `java.util` package's `Arrays` class (for example, `Arrays.sort(selectedNumbers);` accomplishes the same objective as Listing 5–6's `sort(selectedNumbers);` method call, but does so more efficiently), I chose to use Bubble Sort because I prefer to wait until Chapter 8 before getting into the `Arrays` class.

Postconditions

A *postcondition* is something that must be true after a method successfully completes. Assertion statements are often used to satisfy a helper method's postconditions by checking that its result is legal. Listing 5–7 provides an example.

Listing 5–7. *Verifying a postcondition in addition to preconditions*

```
public class MergeArrays
{
   public static void main(String[] args)
   {
      int[] x = { 1, 2, 3, 4, 5 };
      int[] y = { 1, 2, 7, 9 };
      int[] result = merge(x, y);
      for (int i = 0; i < result.length; i++)
         System.out.println(result[i]);
   }
   public static int[] merge(int[] a, int[] b)
   {
      if (a == null)
         throw new NullPointerException("a is null");
      if (b == null)
         throw new NullPointerException("b is null");
      int[] result = new int[a.length+b.length];
      // Precondition
      assert result.length == a.length+b.length: "length mismatch";
      for (int i = 0; i < a.length; i++)
         result[i] = a[i];
      for (int i = 0; i < b.length; i++)
         result[a.length+i-1] = b[i];
      // Postcondition
      assert containsAll(result, a, b): "value missing from array";
      return result;
   }
```

```
   private static boolean containsAll(int[] result, int[] a, int[] b)
   {
      for (int i = 0; i < a.length; i++)
         if (!contains(result, a[i]))
            return false;
      for (int i = 0; i < b.length; i++)
         if (!contains(result, b[i]))
            return false;
      return true;
   }
   private static boolean contains(int[] a, int val)
   {
      for (int i = 0; i < a.length; i++)
         if (a[i] == val)
            return true;
      return false;
   }
}
```

Listing 5–7 uses an assertion statement to verify the postcondition that all of the values in the two arrays being merged are present in the merged array. The postcondition is not satisfied, however, because this listing contains a bug.

Listing 5–7 also shows preconditions and postconditions being used together. The solitary precondition verifies that the merged array length equals the lengths of the arrays being merged prior to the merge logic.

Class Invariants

A *class invariant* is a kind of internal invariant that applies to every instance of a class at all times, except when an instance is transitioning from one consistent state to another.

For example, suppose instances of a class contain arrays whose values are sorted in ascending order. You might want to include an `isSorted()` method in the class that returns true if the array is still sorted, and verify that each constructor and method that modifies the array specifies `assert isSorted();` prior to exit, to satisfy the assumption that the array is still sorted when the constructor/method exists.

Avoiding Assertions

Although there are many situations where assertions should be used, there also are situations where they should be avoided. For example, you should not use assertions to check the arguments that are passed to public methods, for the following reasons:

- Checking a public method's arguments is part of the contract that exists between the method and its caller. If you use assertions to check these arguments, and if assertions are disabled, this contract is violated because the arguments will not be checked.

- Assertions also prevent appropriate exceptions from being thrown. For example, when an illegal argument is passed to a public method, it is common to throw `IllegalArgumentException` or `NullPointerException`. However, `AssertionError` is thrown instead.

You should also avoid using assertions to perform work required by the application to function correctly. This work is often performed as a side effect of the assertion's Boolean expression. When assertions are disabled, the work is not performed.

For example, suppose you have a list of `Employee` objects and a few null references that are also stored in this list, and you want to remove all of the null references. It would not be correct to remove these references via the following assertion statement:

```
assert employees.removeAll(null);
```

Although the assertion statement will not throw `AssertionError` because there is at least one null reference in the `employees` list, the application that depends upon this statement executing will fail when assertions are disabled.

Instead of depending on the former code to remove the null references, you would be better off using code similar to the following:

```
boolean allNullsRemoved = employees.removeAll(null);
assert allNullsRemoved;
```

This time, all null references are removed regardless of whether assertions are enabled or disabled, and you can still specify an assertion to verify that nulls were removed.

Enabling and Disabling Assertions

The compiler records assertions in the classfile. However, assertions are disabled at runtime because they can affect performance. An assertion might call a method that takes awhile to complete, and this would impact the running application's performance.

You must enable the classfile's assertions before you can test assumptions about the behaviors of your classes. Accomplish this task by specifying the `-enableassertions` or `-ea` command-line option when running the `java` application launcher tool.

The `-enableassertions` and `-ea` command-line options let you enable assertions at various granularities based upon one of the following arguments (except for the noargument scenario, you must use a colon to separate the option from its argument):

- *No argument*: Assertions are enabled in all classes except system classes.
- *`PackageName`*`...`: Assertions are enabled in the specified package and its subpackages by specifying the package name followed by `...`.
- `...`: Assertions are enabled in the unnamed package, which happens to be whatever directory is current.
- *`ClassName`*: Assertions are enabled in the named class by specifying the class name.

For example, you can enable all assertions except system assertions when running the `MergeArrays` application via `java -ea MergeArrays`. Also, you could enable any assertions in Chapter 4's `logging` package by specifying `java -ea:logging TestLogger`.

Assertions can be disabled, and also at various granularities, by specifying either of the `-disableassertions` or `-da` command-line options. These options take the same arguments as `-enableassertions` and `-ea`. For example, `java -ea -da:`*`loneclass mainclass`* enables all assertions except for those in *`loneclass`*. (*`loneclass`* and *`mainclass`* are placeholders for the actual classes that you specify.)

The previous options apply to all classloaders. Except when taking no arguments, they also apply to system classes. This exception simplifies the enabling of assertion statements in all classes except for system classes, which is often desirable.

To enable system assertions, specify either `-enablesystemassertions` or `-esa`; for example, `java -esa -ea:logging TestLogger`. Specify either `-disablesystemassertions` or `-dsa` to disable system assertions.

Annotations

While developing a Java application, you might want to *annotate*, or associate *metadata* (data that describes other data) with, various application elements. For example, you might want to identify methods that are not fully implemented so that you will not forget to implement them. Java's annotations language feature lets you accomplish this task.

This section introduces you to annotations. After defining this term and presenting three kinds of compiler-supported annotations as examples, the section shows you how to declare your own annotation types and use these types to annotate source code. Finally, you discover how to process your own annotations to accomplish useful tasks.

> **NOTE:** Java has always supported ad hoc annotation mechanisms. For example, the `java.lang.Cloneable` interface identifies classes whose instances can be shallowly cloned via `Object`'s `clone()` method, the `transient` reserved word marks fields that are to be ignored during serialization, and the `@deprecated` `javadoc` tag documents methods that are no longer supported. In contrast, the annotations feature is a standard for annotating code.

Discovering Annotations

An *annotation* is an instance of an annotation type and associates metadata with an application element. It is expressed in source code by prefixing the type name with the `@` symbol. For example, `@Readonly` is an annotation and `Readonly` is its type.

> **NOTE:** You can use annotations to associate metadata with constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface).
>
> Beginning with Java version 7, annotations can appear on any use of a type. For example, you might declare `Employee[@Readonly] emps;` to indicate that `emps` is an unmodifiable one-dimensional array of *mutable* (modifiable) `Employee` instances.

The compiler supports the `Override`, `Deprecated`, and `SuppressWarnings` annotation types. These types are located in the `java.lang` package.

`@Override` annotations are useful for expressing that a subclass method overrides a method in the superclass, and does not overload that method instead. Listing 5–8 reveals that this annotation prefixes the overriding method.

Listing 5–8. *Annotating an overriding method*

```
@Override
public void draw(int color)
{
   // drawing code
}
```

> **NOTE:** In Chapter 3, I presented `@Override` on the same line as the method header. In Listing 5–8, I present this annotation on a separate line above the method header.
>
> It is common practice to place annotations on separate lines above the application elements that they annotate, unless they are being used to annotate parameters. In that case, an annotation must appear on the same line as and in front of the parameter's name.

`@Deprecated` annotations are useful for indicating that the marked application element is *deprecated* (phased out) and should no longer be used. The compiler warns you when a deprecated application element is accessed by nondeprecated code.

In contrast, the `@deprecated` `javadoc` tag and associated text warns you against using the deprecated item, and tells you what to use instead. Listing 5–9 demonstrates that `@Deprecated` and `@deprecated` can be used together.

Listing 5–9. *Deprecating a method via `@Deprecated` and `@deprecated`*

```
/**
 * Allocates a <code>Date</code> object and initializes it so that
 * it represents midnight, local time, at the beginning of the day
 * specified by the <code>year</code>, <code>month</code>, and
 * <code>date</code> arguments.
 *
 * @param   year    the year minus 1900.
 * @param   month   the month between 0-11.
 * @param   date    the day of the month between 1-31.
 * @see     java.util.Calendar
 * @deprecated As of JDK version 1.1,
```

```
 * replaced by <code>Calendar.set(year + 1900, month, date)</code>
 * or <code>GregorianCalendar(year + 1900, month, date)</code>.
 */
@Deprecated
public Date(int year, int month, int date)
{
    this(year, month, date, 0, 0, 0);
}
```

Listing 5–9 excerpts one of the constructors in Java's `Date` class (located in the `java.util` package). This constructor has been deprecated in favor of using the `set()` method in the `Calendar` class (also located in the `java.util` package).

The compiler suppresses warnings if a compilation unit (typically a class or interface) refers to a deprecated class, method, or field. This feature lets you modify legacy APIs without generating deprecation warnings, and is demonstrated in Listing 5–10.

Listing 5–10. *Referencing a deprecated field from within the same class declaration*

```
public class Employee
{
   /**
    * Employee's name
    * @deprecated New version uses firstName and lastName fields.
    */
   @Deprecated
   String name;
   String firstName;
   String lastName;
   public static void main(String[] args)
   {
      Employee emp = new Employee();
      emp.name = "John Doe";
   }
}
```

Listing 5–10 declares an `Employee` class with a `name` field that has been deprecated. Although `Employee`'s `main()` method refers to `name`, the compiler will suppress a deprecation warning because the deprecation and reference occur in the same class.

Suppose you refactor Listing 5–10 by introducing a new `UseEmployee` class and moving `Employee`'s `main()` method to this class. Listing 5–11 presents the resulting class structure.

Listing 5–11. *Referencing a deprecated field from within another class declaration*

```
class Employee
{
   /**
    * Employee's name
    * @deprecated New version uses firstName and lastName fields.
    */
   @Deprecated
   String name;
   String firstName;
   String lastName;
}
```

```
public class UseEmployee
{
   public static void main(String[] args)
   {
      Employee emp = new Employee();
      emp.name = "John Doe";
   }
}
```

If you attempt to compile this source code via Java SE 6u16's `javac` compiler tool, you will discover the following messages:

```
Note: Employee.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

You will need to specify `-Xlint:deprecation` as one of `javac`'s command-line arguments to discover the deprecated item and the code that refers to this item:

```
Employee.java:17: warning: [deprecation] name in Employee has been deprecated
      emp.name = "John Doe";
          ^
1 warning
```

`@SuppressWarnings` annotations are useful for suppressing deprecation or unchecked warnings via a `"deprecation"` or an `"unchecked"` argument. (Unchecked warnings occur when mixing code that uses generics with pre-generics legacy code.)

For example, Listing 5–12 uses `@SuppressWarnings` with a `"deprecation"` argument to suppress the compiler's deprecation warnings when code within the `UseEmployee` class's `main()` method accesses the `Employee` class's `name` field.

Listing 5–12. *Suppressing the previous deprecation warning*

```
public class UseEmployee
{
   @SuppressWarnings("deprecation")
   public static void main(String[] args)
   {
      Employee emp = new Employee();
      emp.name = "John Doe";
   }
}
```

Declaring Annotation Types and Annotating Source Code

Before you can annotate source code, you need annotation types that can be instantiated. Java supplies many annotation types in addition to `Override`, `Deprecated`, and `SuppressWarnings`. Java also lets you declare your own types.

You declare an annotation type by specifying the `@` symbol, immediately followed by reserved word `interface`, followed by the type's name, followed by a body. For example, Listing 5–13 uses `@interface` to declare an annotation type named `Stub`.

Listing 5–13. *Declaring the `Stub` annotation type*

```
public @interface Stub
```

```
{
}
```

Instances of annotation types that supply no data apart from a name—their bodies are empty—are known as *marker annotations* because they mark application elements for some purpose. As Listing 5–14 reveals, `@Stub` is used to mark empty methods (stubs).

Listing 5–14. *Annotating a stubbed-out method*

```
public class Deck // Describes a deck of cards.
{
   @Stub
   public void shuffle()
   {
      // This method is empty and will presumably be filled in with appropriate
      // code at some later date.
   }
}
```

Listing 5–14's `Deck` class declares an empty `shuffle()` method. This fact is indicated by instantiating `Stub` and prefixing `shuffle()`'s method header with the resulting `@Stub` annotation.

> **NOTE:** Although marker interfaces (discussed in Chapter 3) appear to have been replaced by marker annotations, this is not the case, because marker interfaces have advantages over marker annotations. One advantage is that a marker interface specifies a type that is implemented by a marked class, which lets you catch problems at compile time. For example, if a class does not implement the `Cloneable` interface, its instances cannot be shallowly cloned via `Object`'s `clone()` method. If `Cloneable` had been implemented as a marker annotation, this problem would not be detected until runtime.

Although marker annotations are useful (`@Override` and `@Deprecated` are good examples), you will typically want to enhance an annotation type so that you can store metadata via its instances. You accomplish this task by adding elements to the type.

An *element* is a method header that appears in the annotation type's body. It cannot have parameters or a throws clause, and its return type must be a primitive type (such as `int`), `String`, `Class`, an enum, an annotation type, or an array of the preceding types. However, it can have a default value.

Listing 5–15 adds three elements to `Stub`.

Listing 5–15. *Adding three elements to the `Stub` annotation type*

```
public @interface Stub
{
   int id(); // A semicolon must terminate an element declaration.
   String dueDate();
   String developer() default "unassigned";
}
```

The id() element specifies a 32-bit integer that identifies the stub. The dueDate() element specifies a String-based date that identifies when the method stub is to be implemented. Finally, developer() specifies the String-based name of the developer responsible for coding the method stub.

Unlike id() and dueDate(), developer() is declared with a default value, "unassigned". When you instantiate Stub and do not assign a value to developer() in that instance, as is the case with Listing 5–16, this default value is assigned to developer().

Listing 5–16. *Initializing a Stub instance's elements*

```
public class Deck
{
   @Stub
   (
      id = 1,
      dueDate = "10/21/2010"
   )
   public void shuffle()
   {
   }
}
```

Listing 5–16 reveals one @Stub annotation that initializes its id() element to 1 and its dueDate() element to "10/21/2010". Each element name does not have a trailing (), and the comma-separated list of two element initializers appears between (and).

Suppose you decide to replace Stub's id(), dueDate(), and developer() elements with a single String value() element whose string specifies comma-separated ID, due date, and developer name values. Listing 5–17 shows you two ways to initialize value.

Listing 5–17. *Initializing each Stub instance's value() element*

```
public class Deck
{
   @Stub(value = "1,10/21/2010,unassigned")
   public void shuffle()
   {
   }
   @Stub("2,10/21/2010,unassigned")
   public Card[] deal(int ncards)
   {
      return null;
   }
}
```

This listing reveals special treatment for the value() element. When it is an annotation type's only element, you can omit value()'s name and = from the initializer. I used this fact to specify @SuppressWarnings("deprecation") in Listing 5–12.

Using Meta-Annotations in Annotation Type Declarations

Each of the `Override`, `Deprecated`, and `SuppressWarnings` annotation types is itself annotated with *meta-annotations* (annotations that annotate annotation types). For example, Listing 5–18 shows you that the `SuppressWarnings` annotation type is annotated with two meta-annotations.

Listing 5–18. *The annotated* `SuppressWarnings` *type declaration*

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

The `Target` annotation type, which is located in the `java.lang.annotation` package, identifies the kinds of application elements to which an annotation type applies. `@Target` indicates that `@SuppressWarnings` annotations can be used to annotate types, fields, methods, parameters, constructors, and local variables.

Each of `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, and `LOCAL_VARIABLE` is a member of the `ElementType` enum, which is also located in the `java.lang.annotation` package.

The { and } characters surrounding the comma-separated list of values assigned to `Target`'s `value()` element signify an array—`value()`'s return type is `String[]`. Although these braces are necessary (unless the array consists of one item), `value=` could be omitted when initializing `@Target` because `Target` declares only a `value()` element.

The `Retention` annotation type, which is located in the `java.lang.annotation` package, identifies the retention (also known as lifetime) of an annotation type's annotations. `@Retention` indicates that `@SuppressWarnings` annotations have a lifetime that is limited to source code—they do not exist after compilation.

`SOURCE` is one of the members of the `RetentionPolicy` enum (located in the `java.lang.annotation` package). The other members are `CLASS` and `RUNTIME`. These three members specify the following retention policies:

- `CLASS`: The compiler records annotations in the classfile, but the virtual machine does not retain them (to save memory space). This policy is the default.
- `RUNTIME`: The compiler records annotations in the classfile, and the virtual machine retains them so that they can be read via the Reflection API (discussed in Chapter 7) at runtime.
- `SOURCE`: The compiler discards annotations after using them.

There are two problems with the `Stub` annotation type shown in Listings 5–13 and 5–15. First, the lack of an `@Target` meta-annotation means that you can annotate any application element `@Stub`. However, this annotation only makes sense when applied to methods and constructors. Check out Listing 5–19.

Listing 5–19. *Annotating undesirable application elements*

```
@Stub("1,10/21/2010,unassigned")
public class Deck
{
   @Stub("2,10/21/2010,unassigned")
   private Card[] cardsRemaining;
   @Stub("3,10/21/2010,unassigned")
   public Deck()
   {
   }
   @Stub("4,10/21/2010,unassigned")
   public void shuffle()
   {
   }
   @Stub("5,10/21/2010,unassigned")
   public Card[] deal(@Stub("5,10/21/2010,unassigned") int ncards)
   {
      return null;
   }
}
```

Listing 5–19 uses `@Stub` to annotate the `Deck` class, the `cardsRemaining` field, and the `ncards` parameter in addition to annotating the constructor and the two methods. The first three application elements are inappropriate to annotate because they are not stubs.

You can fix this problem by prefixing the `Stub` annotation type declaration with `@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})` so that `Stub` only applies to methods and constructors. After doing this, the Java SE 6u16 `javac` compiler tool will output the following error messages when you attempt to compile Listing 5–19:

```
Deck.java:1: annotation type not applicable to this kind of declaration
@Stub("1,10/21/2010,unassigned")
^
Deck.java:4: annotation type not applicable to this kind of declaration
   @Stub("2,10/21/2010,unassigned")
   ^
Deck.java:15: annotation type not applicable to this kind of declaration
   public Card[] deal(@Stub("5,10/21/2010,unassigned") int ncards)
                      ^
3 errors
```

The second problem is that the default `CLASS` retention policy makes it impossible to process `@Stub` annotations at runtime. You can fix this problem by prefixing the `Stub` type declaration with `@Retention(RetentionPolicy.RUNTIME)`.

Listing 5–20 presents the `Stub` annotation type with the desired `@Target` and `@Retention` meta-annotations.

Listing 5–20. *A revamped `Stub` annotation type*

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stub
{
   String value();
}
```

NOTE: Java also provides `Documented` and `Inherited` meta-annotation types in the `java.lang.annotation` package. Instances of `@Documented`-annotated annotation types are to be documented by `javadoc` and similar tools.

Instances of `@Inherited`-annotated annotation types are automatically inherited. According to `Inherited`'s Java documentation, if "the user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class's superclass will automatically be queried for the annotation type. This process will be repeated until an annotation for this type is found, or the top of the class hierarchy (`Object`) is reached. If no superclass has an annotation for this type, then the query will indicate that the class in question has no such annotation."

Processing Annotations

It is not enough to declare an annotation type and use that type to annotate source code. Unless you do something specific with those annotations, they remain dormant. One way to accomplish something specific is to write an application that processes the annotations. Listing 5–21's `StubFinder` application does just that.

Listing 5–21. *The `StubFinder` application*

```
import java.lang.reflect.*;

public class StubFinder
{
   public static void main(String[] args) throws Exception
   {
      if (args.length != 1)
      {
         System.err.println("usage: java StubFinder classfile");
         return;
      }
      Method[] methods = Class.forName(args[0]).getMethods();
      for (int i = 0; i < methods.length; i++)
         if (methods[i].isAnnotationPresent(Stub.class))
         {
            Stub stub = methods[i].getAnnotation(Stub.class);
            String[] components = stub.value().split(",");
            System.out.println("Stub ID = " + components[0]);
            System.out.println("Stub Date = " + components[1]);
            System.out.println("Stub Developer = " + components[2]);
            System.out.println();
         }
   }
}
```

`StubFinder` loads a classfile whose name is specified as a command-line argument, and outputs the metadata associated with each `@Stub` annotation that precedes each `public` method header. These annotations are instances of Listing 5–20's `Stub` annotation type.

StubFinder next uses a special class named Class and its forName() class method to load a classfile. Class also provides a getMethods() method that returns an array of Method objects describing the loaded class's public methods.

For each loop iteration, a Method object's isAnnotationPresent() method is called to determine if the method is annotated with the annotation described by the Stub class (referred to as Stub.class).

If isAnnotationPresent() returns true, Method's getAnnotation() method is called to return the annotation Stub instance. This instance's value() method is called to retrieve the string stored in the annotation.

Next, String's split() method is called to split the string's comma-separated list of ID, date, and developer values into an array of String objects. Each object is then output along with descriptive text.

Class's forName() method is capable of throwing various exceptions that must be handled or explicitly declared as part of a method's header. For simplicity, I chose to append a throws Exception clause to the main() method's header.

> **CAUTION:** There are two problems with throws Exception. First, it is better to handle the exception and present a suitable error message than to "pass the buck" by throwing it out of main(). Second, Exception is generic—it hides the names of the kinds of exceptions that are thrown. However, it is convenient to specify throws Exception in a throwaway utility.

Do not be concerned if you do not understand Class, forName(), getMethods(), Method, isAnnotationPresent(), .class, getAnnotation(), and split(). You will learn about these items in Chapter 7.

After compiling StubFinder (javac StubFinder.java), Stub (javac Stub.java), and Listing 5–17's Deck class (javac Deck.java), run StubFinder with Deck as its single command-line argument (java StubFinder Deck). You will observe the following output:

```
Stub ID = 2
Stub Date = 10/21/2010
Stub Developer = unassigned

Stub ID = 1
Stub Date = 10/21/2010
Stub Developer = unassigned
```

If you expected the output to reflect the order of appearance of @Stub annotations in Deck.java, you are probably surprised by the output's unsorted order. This lack of order is caused by getMethods(). According to this method's Java documentation, "the elements in the array returned are not sorted and are not in any particular order."

> **NOTE:** Java version 5 introduced an `apt` tool for processing annotations. This tool's functionality has been integrated into the compiler beginning with Java version 6—`apt` is being phased out. My "Processing Annotations in Java SE 6" article (`http://javajeff.mb.ca/cgi-bin/mp.cgi?a=/java/javase/articles/paijse6`) provides a tutorial on using the Java version 6 compiler to process annotations.

Generics

Java version 5 introduced *generics*, language features for declaring and using type-agnostic classes and interfaces. When working with Java's collections framework (which I introduce in Chapter 8), these features help you avoid `java.lang.ClassCastException`s.

> **NOTE:** Although the main use for generics is the collections framework, Java's class library also contains *generified* (retrofitted to make use of generics) classes that have nothing to do with this framework: `java.lang.Class`, `java.lang.ThreadLocal`, and `java.lang.ref.WeakReference` are three examples.

This section introduces you to generics. You first learn how generics promote type safety in the context of the collections classes, and then you explore generics in the contexts of generic types and generic methods.

Collections and the Need for Type Safety

Java's collections framework makes it possible to store objects in various kinds of containers (known as collections) and later retrieve those objects. For example, you can store objects in a list, a set, or a map. You can then retrieve a single object, or iterate over the collection and retrieve all objects.

Before Java version 5 overhauled the collections framework to take advantage of generics, there was no way to prevent a collection from containing objects of mixed types. The compiler did not check an object's type to see if it was suitable before it was added to a collection, and this lack of static type checking led to `ClassCastException`s.

Listing 5–22 demonstrates how easy it is to generate a `ClassCastException`.

Listing 5–22. *Lack of type safety leading to a `ClassCastException` at runtime*

```
public static void main(String[] args)
{
   List employees = new ArrayList();
   employees.add(new Employee("John Doe"));
   employees.add(new Employee("Jane Doe"));
   employees.add("Doe Doe");
   Iterator iter = employees.iterator();
   while (iter.hasNext())
```

```
   {
      Employee emp = (Employee) iter.next();
      System.out.println(emp.getName());
   }
}
```

After instantiating `ArrayList`, `main()` uses this list collection object's reference to add a pair of `Employee` objects to the list. It then adds a `String` object, which violates the implied contract that `ArrayList` is supposed to store only `Employee` objects.

Moving on, `main()` obtains an `Iterator` for iterating over the list of `Employees`. As long as `Iterator`'s `hasNext()` method returns true, its `next()` method is called to return an object stored in the `ArrayList`.

The `Object` that `next()` returns must be downcast to `Employee` so that the `Employee` object's `getName()` method can be called to return the employee's name. The string that this method returns is then output to the standard output device via `System.out.println()`.

The `(Employee)` cast checks the type of each object returned by `next()` to make sure that it is an `Employee`. Although this is true of the first two objects, it is not true of the third object. Attempting to cast `"Doe Doe"` to `Employee` results in a `ClassCastException`.

The `ClassCastException` occurs because of an assumption that a list is *homogenous*. In other words, a list stores only objects of a single type or a family of related types. In reality, the list is *heterogeneous* in that it can store any `Object`.

Listing 5–23's generics-based homogenous list avoids `ClassCastException`.

Listing 5–23. *Lack of type safety leading to a compiler error*

```
public static void main(String[] args)
{
   List<Employee> employees = new ArrayList<Employee>();
   employees.add(new Employee("John Doe"));
   employees.add(new Employee("Jane Doe"));
   employees.add("Doe Doe");
   Iterator<Employee> iter = employees.iterator();
   while (iter.hasNext())
   {
      Employee emp = iter.next();
      System.out.println(emp.getName());
   }
}
```

Three of Listing 5–23's bolded code fragments illustrate the central feature of generics, which is the *parameterized type* (a class or interface name followed by an angle bracket–delimited type list identifying what kinds of objects are legal in that context).

For example, `List<Employee>` indicates only `Employee` objects can be stored in the `List`. The `<Employee>` designation must be repeated with `ArrayList`, which is the collection implementation that stores the `Employees`.

> **NOTE:** Java version 7 introduces the diamond operator (<>) to save you from having to repeat a parameterized type. For example, you could use this operator to specify `List<Employee> employees = new ArrayList<>();`.

Also, `Iterator<Employee>` indicates that `iterator()` returns an `Iterator` whose `next()` method returns only `Employee` objects. It is not necessary to cast `iter.next()`'s returned value to `Employee` because the compiler inserts the cast on your behalf.

If you attempt to compile this listing, the compiler will report an error when it encounters `employees.add("Doe Doe");`. The error message will tell you that the compiler cannot find an `add(java.lang.String)` method in the `java.util.List<Employee>` interface.

Unlike in the pre-generics `List` interface, which declares an `add(Object)` method, the generified `List` interface's `add()` method parameter reflects the interface's parameterized type name. For example, `List<Employee>` implies `add(Employee)`.

Listing 5–22 revealed that the unsafe code causing the `ClassCastException` (`employees.add("Doe Doe");`) and the code that triggers the exception (`(Employee) iter.next()`) are quite close. However, they are often farther apart in larger applications.

Rather than having to deal with angry clients while hunting down the unsafe code that ultimately led to the `ClassCastException`, you can rely on the compiler saving you this frustration and effort by reporting an error when it detects this code during compilation. Detecting type safety violations at compile time is the benefit of using generics.

Generic Types

A *generic type* is a class or interface that introduces a family of parameterized types by declaring a *formal type parameter list* (a comma-separated list of *type parameter* names between angle brackets). This syntax is expressed as follows:

```
class identifier<formal_type_parameter_list> {}
interface identifier<formal_type_parameter_list> {}
```

For example, `List<E>` is a generic type, where `List` is an interface and type parameter `E` identifies the list's element type. Similarly, `Map<K, V>` is a generic type, where `Map` is an interface and type parameters `K` and `V` identify the map's key and value types.

> **NOTE:** When declaring a generic type, it is conventional to specify single uppercase letters as type parameter names. Furthermore, these names should be meaningful. For example, `E` indicates element, `T` indicates type, `K` indicates key, and `V` indicates value. If possible, you should avoid choosing a type parameter name that is meaningless where it is used. For example, `List<E>` means list of elements, but what does `List<S>` mean?

Parameterized types instantiate generic types. Each parameterized type replaces the generic type's type parameters with type names. For example, `List<Employee>` (`List` of

Employee) and List<String> (List of String) are examples of parameterized types based on List<E>. Similarly, Map<String, Employee> is an example of a parameterized type based on Map<K, V>.

The type name that replaces a type parameter is known as an *actual type argument*. Generics supports five kinds of actual type arguments:

- *Concrete type*: The name of a class or interface is passed to the type parameter. For example, List<Employee> employees; specifies that the list elements are Employee instances.
- *Concrete parameterized type*: The name of a parameterized type is passed to the type parameter. For example, List<List<String>> nameLists; specifies that the list elements are lists of strings.
- *Array type*: An array is passed to the type parameter. For example, List<String[]> countries; specifies that the list elements are arrays of Strings, possibly city names.
- *Type parameter*: A type parameter is passed to the type parameter. For example, given class declaration class X<E> { List<E> queue; }, X's type parameter E is passed to List's type parameter E.
- *Wildcard*: The ? is passed to the type parameter. For example, List<?> list; specifies that the list elements are unknown. You will learn about this type parameter later in the chapter, in "The Need for Wildcards" section.

A generic type also identifies a *raw type*, which is a generic type without its type parameters. For example, List<Employee>'s raw type is List. Raw types are nongeneric and can hold any Object.

> **NOTE:** Java allows raw types to be intermixed with generic types to support the vast amount of legacy code that was written prior to the arrival of generics. However, the compiler outputs a warning message whenever it encounters a raw type in source code.

Declaring and Using Your Own Generic Types

It is not difficult to declare your own generic types. In addition to specifying a formal type parameter list, your generic type specifies its type parameter(s) throughout its implementation. For example, Listing 5–24 declares a Queue<E> generic type.

Listing 5–24. *Declaring and using a Queue<E> generic type*

```
public class Queue<E>
{
   private E[] elements;
   private int head, tail;
   @SuppressWarnings("unchecked")
   public Queue(int size)
```

```
   {
      elements = (E[]) new Object[size];
      head = 0;
      tail = 0;
   }
   public void insert(E element)
   {
      if (isFull()) // insert() should throw an exception when full. I did
         return;    // not implement insert() to do so for brevity.
      elements[tail] = element;
      tail = (tail+1)%elements.length;
   }
   public E remove()
   {
      if (isEmpty())
         return null;
      E element = elements[head];
      head = (head+1)%elements.length;
      return element;
   }
   public boolean isEmpty()
   {
      return head == tail;
   }
   public boolean isFull()
   {
      return (tail+1)%elements.length == head;
   }
   public static void main(String[] args)
   {
      Queue<String> queue = new Queue<String>(5);
      System.out.println(queue.isEmpty());
      queue.insert("A");
      queue.insert("B");
      queue.insert("C");
      queue.insert("D");
      queue.insert("E");
      System.out.println(queue.isFull());
      System.out.println(queue.remove());
      queue.insert("F");
      while (!queue.isEmpty())
         System.out.println(queue.remove());
      System.out.println(queue.isEmpty());
      System.out.println(queue.isFull());
   }
}
```

`Queue` implements a *queue*, a data structure that stores elements in first-in, first-out order. An element is inserted at the *tail* and removed at the *head*. The queue is empty when the head equals the tail, and full when the tail is one less than the head.

Notice that `Queue<E>`'s `E` type parameter appears throughout the source code. For example, `E` appears in the `elements` array declaration to denote the array's element type. `E` is also specified as the type of `insert()`'s parameter and as `remove()`'s return type.

E also appears in elements = (E[]) new Object[size];. (I will explain later why I specified this expression instead of specifying the more compact elements = new E[size]; expression.)

The E[] cast results in the compiler warning about this cast being unchecked. The compiler is concerned that downcasting from Object[] to E[] might result in a violation of type safety because any kind of object can be stored in Object[].

The compiler's concern is not justified in this example. There is no way that a non-E object can appear in the E[] array. Because the warning is meaningless in this context, it is suppressed by prefixing the constructor with @SuppressWarnings("unchecked").

> **CAUTION:** Be careful when suppressing an unchecked warning. You must first prove that a ClassCastException cannot occur, and then you can suppress the warning.

When you run this application, it generates the following output:

```
true
true
A
B
C
D
F
true
false
```

Type Parameter Bounds

List<E>'s E type parameter and Map<K, V>'s K and V type parameters are examples of unbounded type parameters. You can pass any actual type argument to an unbounded type parameter.

It is sometimes necessary to restrict the kinds of actual type arguments that can be passed to a type parameter. For example, you might want to declare a class whose instances can only store instances of classes that subclass an abstract Shape class.

To restrict actual type arguments, you can specify an *upper bound*, a type that serves as an upper limit on the types that can be chosen as actual type arguments. The upper bound is specified via reserved word extends followed by a type name.

For example, ShapesList<E extends Shape> identifies Shape as an upper bound. You can specify ShapesList<Circle>, ShapesList<Rectangle>, and even ShapesList<Shape>, but not ShapesList<String> because String is not a subclass of Shape.

You can assign more than one upper bound to a type parameter, where the first bound is a class or interface, and where each additional upper bound is an interface, by using the ampersand character (&) to separate bound names. Consider Listing 5–25.

Listing 5–25. *Assigning multiple upper bounds to a type parameter*

```
abstract class Shape
{
}
class Circle extends Shape implements Comparable<Circle>
{
   private double x, y, radius;
   Circle(double x, double y, double radius)
   {
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
   @Override
   public int compareTo(Circle circle)
   {
      if (radius < circle.radius)
         return -1;
      else
      if (radius > circle.radius)
         return 1;
      else
         return 0;
   }
   @Override
   public String toString()
   {
      return "(" + x + ", " + y + ", " + radius + ")";
   }
}
class SortedShapesList<S extends Shape & Comparable<S>>
{
   @SuppressWarnings("unchecked")
   private S[] shapes = (S[]) new Shape[2];
   private int index = 0;
   void add(S shape)
   {
      shapes[index++] = shape;
      if (index < 2)
         return;
      System.out.println("Before sort: " + this);
      sort();
      System.out.println("After sort: " + this);
   }
   private void sort()
   {
      if (index == 1)
         return;
      if (shapes[0].compareTo(shapes[1]) > 0)
      {
         S shape = (S) shapes[0];
         shapes[0] = shapes[1];
         shapes[1] = shape;
      }
   }
   @Override
   public String toString()
```

```
   {
      return shapes[0].toString() + " " + shapes[1].toString();
   }
}
public class SortedShapesListDemo
{
   public static void main(String[] args)
   {
      SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();
      ssl.add(new Circle(100, 200, 300));
      ssl.add(new Circle(10, 20, 30));
   }
}
```

Listing 5–25's `Circle` class extends `Shape` and implements the `java.lang.Comparable` interface, which is used to specify the *natural ordering* of `Circle` objects. The interface's `compareTo()` method implements this ordering by returning a value to reflect the order:

- A negative value is returned if the current object should precede the object passed to `compareTo()` in some fashion.
- A zero value is returned if the current and argument objects are the same.
- A positive value is returned if the current object should succeed the argument object.

`Circle`'s overriding `compareTo()` method compares two `Circle` objects based on their radii. This method orders a `Circle` instance with the smaller radius before a `Circle` instance with a larger radius.

The `SortedShapesList` class specifies `<S extends Shape & Comparable<S>>` as its parameter list. The actual type argument passed to the `S` parameter must subclass `Shape`, and it must also implement the `Comparable` interface.

`Circle` satisfies both criteria: it subclasses `Shape` and implements `Comparable`. As a result, the compiler does not report an error when it encounters the `main()` method's `SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();` statement.

An upper bound offers extra static type checking that guarantees that a parameterized type adheres to its bounds. This assurance means that the upper bound's methods can be called safely. For example, `sort()` can call `Comparable`'s `compareTo()` method.

If you run this application, you will discover the following output, which shows that the two `Circle` objects are sorted in ascending order of radius:

```
Before sort: (100.0, 200.0, 300.0) (10.0, 20.0, 30.0)
After sort: (10.0, 20.0, 30.0) (100.0, 200.0, 300.0)
```

You can also restrict actual type arguments by specifying a *lower bound*, a type that serves as a lower limit on the types that can be chosen as actual type arguments. The lower bound is specified via the wildcard, reserved word `super`, and a type name.

Because lower bounds are used exclusively with the wildcard type argument, I will have more to say about lower bounds when I discuss the need for wildcards.

Type Parameter Scope

A type parameter's *scope* (visibility) is its generic type, which includes the formal type parameter list of which the type parameter is a member. For example, the scope of `S` in `SortedShapesList<S extends Shape & Comparable<S>>` is all of `SortedShapesList` and the formal type parameter list.

> **NOTE:** A type parameter bound that includes the type parameter is known as a *recursive type bound*. For example, `Comparable<S>` in `<S extends Shape & Comparable<S>>` is a recursive type bound. Recursive type bounds are rare and typically show up in conjunction with the `Comparable` interface, for specifying a type's natural ordering.

It is possible to mask a type parameter by declaring a same-named type parameter in a nested type's formal type parameter list. For example, Listing 5–26 masks an enclosing class's `T` type parameter.

Listing 5–26. *Masking a type variable*

```
class EnclosingClass<T>
{
   static class EnclosedClass<T extends Comparable<T>>
   {
   }
}
```

`EnclosingClass`'s `T` type parameter is masked by `EnclosedClass`'s `T` type parameter, which specifies an upper bound where only those types that implement the `Comparable` interface can be passed to `EnclosedClass`.

If masking is undesirable, it is best to choose a different name for the type parameter. For example, you might specify `EnclosedClass<U extends Comparable<U>>`. Although `U` is not as meaningful a name as `T`, this situation justifies this choice.

The Need for Wildcards

In Chapter 3, you learned that a subtype is a kind of supertype. For example, `Circle` is a kind of `Shape` and `String` is a kind of `Object`. This polymorphic behavior also applies to related parameterized types with the same type parameters (`List<Object>` is a kind of `Collection<Object>`, for example).

However, this polymorphic behavior does not apply to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, `List<String>` is not a kind of `List<Object>`. Listing 5–27 reveals why parameterized types differing only in type parameters are not polymorphic.

Listing 5–27. *Proving that parameterized types differing only in type parameters are not polymorphic*

```
public static void main(String[] args)
{
   List<String> ls = new ArrayList<String>();
   List<Object> lo = ls;
```

```
   lo.add(new Employee());
   String s = ls.get(0);
}
```

If Listing 5–27 compiled, a `ClassCastException` would be thrown at runtime. After instantiating a `List` of `String` and upcasting its reference to a `List` of `Object`, `main()` adds a new `Employee` object to the `List` of `Object`. The `Employee` object is then returned via `get()` and the `List` of `String` reference variable. The `ClassCastException` is thrown because of the implicit cast to `String`—an `Employee` is not a `String`.

> **NOTE:** Although you cannot upcast `List<String>` to `List<Object>`, you can upcast `List<String>` to the raw type `List` in order to interoperate with legacy code.

This example can be generalized into the following rule: for a given subtype *x* of type *y*, and given *G* as a raw type declaration, *G*<*x*> is not a subtype of *G*<*y*>. Listing 5–28 shows you how easy it is to run afoul of this rule.

Listing 5–28. *Attempting to output a list of string*

```
public static void main(String[] args)
{
   List<String> ls = new ArrayList<String>();
   ls.add("first");
   ls.add("second");
   ls.add("third");
   outputList(ls);
}
static void outputList(List<Object> list)
{
   for (int i = 0; i < list.size(); i++)
      System.out.println(list.get(i));
}
```

Listing 5–28 will not compile because it assumes that `List` of `String` is also a `List` of `Object`, which you have just learned is not true because it violates type safety. The `outputList()` method can only output a list of objects, which makes it useless.

The wildcard type argument (?) provides a typesafe workaround to this problem because it accepts any actual type argument. Simply change `List<Object>` list to `List<?> list` and Listing 5–28 will compile.

However, you cannot add elements to a `List<?>` because doing so would violate type safety. For example, Listing 5–29 presents a `copyList()` method that attempts to copy one `List<?>` to another `List<?>`.

Listing 5–29. *Attempting to copy one `List<?>` to another `List<?>`*

```
public static void main(String[] args)
{
   List<String> ls1 = new ArrayList<String>();
   ls1.add("first");
   ls1.add("second");
   ls1.add("third");
   List<String> ls2 = new ArrayList<String>();
```

```
   copyList(ls1, ls2);
}
static void copyList(List<?> list1, List<?> list2)
{
   for (int i = 0; i < list1.size(); i++)
      list2.add(list1.get(i));
}
```

Listing 5–29 will not compile. Instead, the compiler reports the following error message when it encounters `list2.add(list1.get(i));`:

```
x.java:13: cannot find symbol
symbol  : method add(java.lang.Object)
location: interface java.util.List<capture#469 of ?>
      list2.add(list1.get(i));
            ^
1 error
```

The error message reflects that although `list1`'s elements (which can be of any type) can be assigned to `Object`, `list2`'s element type is unknown. If this type is anything other than `Object` (such as `String`), type safety is violated. One solution to this problem is to specify `static void copyList(List<String> list1, List<String> list2)`.

A second solution to this problem requires that `copyList()`'s first type parameter be specified as `? extends String` and its second parameter be specified as `? super String`. Listing 5–30 reveals the same `copyList()` method with these upper and lower bounds.

Listing 5–30. *Using bounded wildcards to successfully copy one list to another*

```
public static void main(String[] args)
{
   List<String> ls1 = new ArrayList<String>();
   ls1.add("first");
   ls1.add("second");
   ls1.add("third");
   List<String> ls2 = new ArrayList<String>();
   copyList(ls1, ls2);
}
static void copyList(List<? extends String> list1, List<? super String> list2)
{
   for (int i = 0; i < list1.size(); i++)
      list2.add(list1.get(i));
}
```

Listing 5–30's `copyList()` method reveals that the `list1` parameter accepts `String` or any subclass. Because `String` is declared `final`, no subclasses can be passed as actual type arguments. In contrast, the `list2` parameter accepts `String` and its `Object` superclass—a subclass is a kind of superclass.

Although Listing 5–30 compiles and runs, its version of the `copyList()` header is little better than specifying `static void copyList(List<String> list1, List<String> list2)`. You can only copy a `List` of `String` to a `List` of `String` or a `List` of `Object`. You will shortly discover that generic methods offer a much better solution.

Reification and Erasure

Reification is the process or result of treating the abstract as if it was concrete. For example, `0xa000000` is an abstract hexadecimal integer literal that is treated as if it was the concrete 32-bit memory address that it represents.

Java arrays are reified. Because they are aware of their element types, they can enforce these types at runtime. Attempting to store an invalid element in an array (see Listing 5–31) results in a `java.lang.ArrayStoreException`.

Listing 5–31. *How an `ArrayStoreException` occurs*

```
class Point
{
   int x, y;
}
class ColoredPoint extends Point
{
   int color;
}
public class ReificationDemo
{
   public static void main(String[] args)
   {
      ColoredPoint[] cptArray = new ColoredPoint[1];
      Point[] ptArray = cptArray;
      ptArray[0] = new Point();
   }
}
```

Listing 5–31 demonstrates that arrays are *covariant* in that an array of supertype references is a supertype of an array of subtype references. A subtype array can be assigned to a supertype variable; for example, `Point[] ptArray = cptArray;`.

Covariance can be dangerous. For example, an `ArrayStoreException` is thrown when `ptArray[0] = new Point();` tries to store a `Point` object in the `ColoredPoint` array via the `ptArray` intermediary. The exception occurs because a `Point` is not a `ColoredPoint`.

In contrast to arrays, a generic type's type parameters are not reified. In other words, they are not available at runtime. Instead, these type parameters are discarded following compilation. This throwing away of type parameters is known as *erasure*.

Erasure also involves replacing uses of other type variables by the upper bound of the type variable (such as `Object`) and inserting casts to the appropriate type when the resulting code is not type correct.

Although erasure makes it possible for generic types to interoperate with raw types, it does have consequences. For example, you cannot create an array of a generic type: `new List<E>[10]`, `new E[size]`, and `new Map<String, String>[100]` are illegal.

The compiler reports a "generic array creation" error when it encounters an attempt to create an array of a generic type. It does so because the attempt is not typesafe. If allowed, compiler-inserted casts could trigger `ClassCastExceptions`—see Listing 5–32.

Listing 5–32. *Why creating an array of a generic type is not a good idea*

```
List<Employee>[] empListArray = new List<Employee>[1];
List<String> strList = new ArrayList<String>(); strList.add("string");
Object[] objArray = empListArray;
objArray[0] = strList;
Employee e = empListArray[0].get(0);
```

Let us assume that Listing 5–32 is legal. The first line creates a one-element array where this element stores a `List` of `Employee`. The second line creates a `List` of `String` and stores a single `String` in this list.

The third line assigns `empListArray` to `objArray`. This assignment is legal because arrays are covariant. The fourth line stores the `List` of `String` into `objArray[0]`, which works because of erasure. An `ArrayStoreException` does not occur because `List<String>`'s runtime type is `List` and `List<Employee>[]`'s runtime type is `List[]`.

However, there is a problem. A `List<String>` instance has been stored in an array that can only hold `List<Employee>` instances. When the compiler-inserted cast operator attempts to cast `empListArray[0].get(0)`'s return value (`"string"`) to `Employee`, the cast operator throws a `ClassCastException` object.

Another consequence of erasure is that the `instanceof` operator cannot be used with parameterized types apart from unbounded wildcard types. For example, `List<String> ls = null; if (ls instanceof LinkedList<String>) {}` is illegal. Instead you must change the `instanceof` expression to either of the following expressions:

```
ls instanceof LinkedList<?> // legal to use with unbounded wildcard type
ls instanceof LinkedList    // legal to use with raw type (the preferred use)
```

Generic Methods

A *generic method* is a `static` or non-`static` method with a type-generalized implementation. A formal type parameter list precedes the method's return type, uses the same syntax, and has the same meaning as the generic type's formal type parameter list. This syntax is expressed as follows:

```
<formal_type_parameter_list> return_type identifier(parameter_list)
{
}
```

The collections framework provides many examples of generic methods. For example, the `Collections` class provides a `public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)` method for returning the minimum element in the given collection according to the natural ordering of its elements.

Listing 5–33 converts the aforementioned `copyList()` method into a generic method so that it can copy a list of an arbitrary type to another list of the same type.

Listing 5–33. *Declaring and using a `copyList()` generic method*

```
class Circle
{
   private double x, y, radius;
```

```
   Circle(double x, double y, double radius)
   {
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
   @Override
   public String toString()
   {
      return "(" + x + ", " + y + ", " + radius + ")";
   }
}
public class CopyList
{
   public static void main(String[] args)
   {
      List<String> ls = new ArrayList<String>();
      ls.add("A");
      ls.add("B");
      ls.add("C");
      outputList(ls);
      List<String> lsCopy = new ArrayList<String>();
      copyList(ls, lsCopy);
      outputList(lsCopy);
      List<Circle> lc = new ArrayList<Circle>();
      lc.add(new Circle(10.0, 20.0, 30.0));
      lc.add(new Circle (5.0, 4.0, 16.0));
      outputList(lc);
      List<Circle> lcCopy = new ArrayList<Circle>();
      copyList(lc, lcCopy);
      outputList(lcCopy);
   }
   static <T> void copyList(List<T> c1, List<T> c2)
   {
      for (int i = 0; i < c1.size(); i++)
         c2.add(c1.get(i));
   }
   static void outputList(List<?> l)
   {
      for (int i = 0; i < l.size(); i++)
         System.out.println (l.get(i));
      System.out.println();
   }
}
```

The generic method's type parameters are inferred from the context in which the method was invoked. For example, the compiler determines that `copyList(ls, lsCopy);` copies a `List` of `String` to another `List` of `String`. Similarly, it determines that `copyList(lc, lcCopy);` copies a `List` of `Circle` to another `List` of `Circle`.

When you run this application, it generates the following output:

```
A
B
C

A
B
C

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)
```

Enums

An *enumerated type* is a type that specifies a named sequence of related constants as its legal values. The months in a calendar, the coins in a currency, and the days of the week are examples of enumerated types.

Java developers have traditionally used sets of named integer constants to represent enumerated types. Because this form of representation has proven to be problematic, Java version 5 introduced the enum alternative.

This section introduces you to enums. After discussing the problems with traditional enumerated types, the section presents the enum alternative. It then introduces you to the `Enum` class, from which enums originate.

The Trouble with Traditional Enumerated Types

Listing 5–34 declares a `Coin` enumerated type whose set of constants identifies different kinds of coins in a currency.

Listing 5–34. *An enumerated type identifying coins*

```
public class Coin
{
   public final static int PENNY = 0;
   public final static int NICKEL = 1;
   public final static int DIME = 2;
   public final static int QUARTER = 3;
}
```

Listing 5–35 declares a `Weekday` enumerated type whose set of constants identifies the days of the week.

Listing 5–35. *An enumerated type identifying weekdays*

```
public class Weekday
{
   public final static int SUNDAY = 0;
   public final static int MONDAY = 1;
```

```
   public final static int TUESDAY = 2;
   public final static int WEDNESDAY = 3;
   public final static int THURSDAY = 4;
   public final static int FRIDAY = 5;
   public final static int SATURDAY = 6;
}
```

Listing 5–34's and 5–35's approach to representing an enumerated type is problematic, where the biggest problem is the lack of compile-time type safety. For example, you can pass a coin to a method that requires a weekday and the compiler will not complain.

You can also compare coins to weekdays, as in `Coin.NICKEL == Weekday.MONDAY`, and specify even more meaningless expressions, such as `Coin.DIME+Weekday.FRIDAY-1/Coin.QUARTER`. The compiler does not complain because it only sees `int`s.

Applications that depend upon enumerated types are brittle. Because the type's constants are compiled into an application's classfiles, changing a constant's `int` value requires you to recompile dependent applications or risk them behaving erratically.

Another problem with enumerated types is that `int` constants cannot be translated into meaningful string descriptions. For example, what does `4` mean when debugging a faulty application? Being able to see `THURSDAY` instead of `4` would be more helpful.

> **NOTE:** You could circumvent the previous problem by using `String` constants. For example, you might specify `public final static String THURSDAY = "THURSDAY";`. Although the constant value is more meaningful, `String`-based constants can impact performance because you cannot use `==` to efficiently compare just any old strings (as you will discover in Chapter 7). Other problems related to `String`-based constants include hard-coding the constant's value (`"THURSDAY"`) instead of the constant's name (`THURSDAY`) into source code, which makes it very difficult to change the constant's value at a later time; and misspelling a hard-coded constant (`"THURZDAY"`), which compiles correctly but is problematic at runtime.

The Enum Alternative

Java version 5 introduced enums as a better alternative to traditional enumerated types. An *enum* is an enumerated type that is expressed via reserved word `enum`. Listing 5–36 uses `enum` to declare Listing 5–34's and 5–35's enumerated types.

Listing 5–36. *Improved enumerated types for coins and weekdays*

```
public enum Coin { PENNY, NICKEL, DIME, QUARTER }
public enum Weekday { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY }
```

Despite their similarity to the `int`-based enumerated types found in C++ and other languages, Listing 5–36's enums are classes. Each constant is a `public static final` field that represents an instance of its enum class.

Because constants are final, and because you cannot call an enum's constructors to create more constants, you can use == to compare constants efficiently and (unlike string constant comparisons) safely. For example, you can specify `c == Coin.NICKEL`.

Enums promote compile-time type safety by preventing you from comparing constants in different enums. For example, the compiler will report an error when it encounters `Coin.PENNY == Weekday.SUNDAY`.

The compiler also frowns upon passing a constant of the wrong enum kind to a method. For example, you cannot pass `Weekday.FRIDAY` to a method whose parameter type is `Coin`.

Applications depending upon enums are not brittle because the enum's constants are not compiled into an application's classfiles. Also, the enum provides a `toString()` method for returning a more useful description of a constant's value.

Because enums are so useful, Java version 5 enhanced the switch statement to support them. Listing 5–37 demonstrates this statement switching on one of the constants in Listing 5–36's `Coin` enum.

Listing 5–37. *Using the switch statement with an enum*

```
public class EnhancedSwitch
{
   private enum Coin { PENNY, NICKEL, DIME, QUARTER }
   public static void main(String[] args)
   {
      Coin coin = Coin.NICKEL;
      switch (coin)
      {
         case PENNY  : System.out.println("1 cent"); break;
         case NICKEL : System.out.println("5 cents"); break;
         case DIME   : System.out.println("10 cents"); break;
         case QUARTER: System.out.println("25 cents");
         default     : assert false;
      }
   }
}
```

Listing 5–37 demonstrates switching on an enum's constants. This enhanced statement only allows you to specify the name of a constant as a case label. If you prefix the name with the enum, as in `case Coin.DIME`, the compiler reports an error.

Enhancing an Enum

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. For example, Listing 5–38 adds a field, a constructor, and two methods to `Coin` to associate a denomination value with a `Coin` constant (such as 1 for penny and 5 for nickel) and convert pennies to the denomination.

Listing 5–38. *Enhancing the* `Coin` *enum*

```
public enum Coin
{
   PENNY(1),
   NICKEL(5),
   DIME(10),
   QUARTER(25);

   private final int denomValue;
   Coin(int denomValue)
   {
      this.denomValue = denomValue;
   }
   public int denomValue()
   {
      return denomValue;
   }
   public int toDenomination(int numPennies)
   {
      return numPennies/denomValue;
   }
}
```

Listing 5–38's constructor accepts a denomination value, which it assigns to a `private` blank final field named `denomValue`—all fields should be declared `final` because constants are immutable. Notice that this value is passed to each constant during its creation (`PENNY(1)`, for example).

> **CAUTION:** When the comma-separated list of constants is followed by anything other than an enum's closing brace, you must terminate the list with a semicolon or the compiler will report an error.

Furthermore, this listing's `denomValue()` method returns `denomValue`, and its `toDenomination()` method returns the number of coins of that denomination that are contained within the number of pennies passed to this method as its argument. For example, 3 nickels are contained in 16 pennies.

Listing 5–39 shows you how to use the enhanced `Coin` enum.

Listing 5–39. *Exercising the enhanced* `Coin` *enum*

```
public static void main(String[] args)
{
   if (args.length == 1)
   {
      int numPennies = Integer.parseInt(args[0]);
      System.out.println(numPennies + " pennies is equivalent to:");
      int numQuarters = Coin.QUARTER.toDenomination(numPennies);
      System.out.println(numQuarters + " " + Coin.QUARTER.toString() +
                         (numQuarters != 1 ? "s," : ","));
      numPennies -= numQuarters*Coin.QUARTER.denomValue();
      int numDimes = Coin.DIME.toDenomination(numPennies);
      System.out.println(numDimes + " " + Coin.DIME.toString() +
```

```
                        (numDimes != 1 ? "s, " : ","));
      numPennies -= numDimes*Coin.DIME.denomValue();
      int numNickels = Coin.NICKEL.toDenomination(numPennies);
      System.out.println(numNickels + " " + Coin.NICKEL.toString() +
                        (numNickels != 1 ? "s, " : ", and"));
      numPennies -= numNickels*Coin.NICKEL.denomValue();
      System.out.println(numPennies + " " + Coin.PENNY.toString() +
                        (numPennies != 1 ? "s" : ""));
   }
   System.out.println();
   System.out.println("Denomination values:");
   for (int i = 0; i < Coin.values().length; i++)
      System.out.println(Coin.values()[i].denomValue());
}
```

Listing 5–39 describes an application that converts its solitary "pennies" command-line argument to an equivalent amount expressed in quarters, dimes, nickels, and pennies. In addition to calling a `Coin` constant's `denomValue()` and `toDenomValue()` methods, the application calls `toString()` to output a string representation of the coin.

Another called enum method is `values()`. This method returns an array of all `Coin` constants that are declared in the `Coin` enum (`value()`'s return type, in this example, is `Coin[]`). This array is useful when you need to iterate over these constants. For example, Listing 5–39 calls this method to output each coin's denomination.

When you run this application with `119` as its command-line argument, it generates the following output:

```
119 pennies is equivalent to:
4 QUARTERs,
1 DIME,
1 NICKEL, and
4 PENNYs

Denomination values:
1
5
10
25
```

The output shows that `toString()` returns a constant's name. It is sometimes useful to override this method to return a more meaningful value. For example, a method that extracts *tokens* (named character sequences) from a string might use a `Token` enum to list token names and, via an overriding `toString()` method, values—see Listing 5–40.

Listing 5–40. *Overriding `toString()` to return a `Token` constant's value*

```
public enum Token
{
   IDENTIFIER("ID"),
   INTEGER("INT"),
   LPAREN("("),
   RPAREN(")"),
   COMMA(",");

   private final String tokValue;
   Token(String tokValue)
```

```
   {
      this.tokValue = tokValue;
   }
   @Override
   public String toString()
   {
      return tokValue;
   }
   public static void main(String[] args)
   {
      System.out.println("Token values:");
      for (int i = 0; i < Token.values().length; i++)
         System.out.println(Token.values()[i].name() + " = " +
                            Token.values()[i]);
   }
}
```

This application calls `values()` to return the array of `Token` constants. For each constant, it calls the constant's `name()` method to return the constant's name, and implicitly calls `toString()` to return the constant's value. If you were to run this application, you would observe the following output:

```
Token values:
IDENTIFIER = ID
INTEGER = INT
LPAREN = (
RPAREN = )
COMMA = ,
```

Another way to enhance an enum is to assign a different behavior to each constant. You can accomplish this task by introducing an abstract method into the enum and overriding this method in an anonymous subclass of the constant. Listing 5–41's `TempConversion` enum demonstrates this technique.

Listing 5–41. *Using anonymous subclasses to vary the behaviors of enum constants*

```
public enum TempConversion
{
   C2F("Celsius to Fahrenheit")
   {
      @Override
      public double convert(double value)
      {
         return value*9.0/5.0+32.0;
      }
   },
   F2C("Fahrenheit to Celsius")
   {
      @Override
      public double convert(double value)
      {
         return (value-32.0)*5.0/9.0;
      }
   };

   TempConversion(String desc)
   {
```

```
      this.desc = desc;
   }
   private String desc;
   @Override
   public String toString()
   {
      return desc;
   }
   public abstract double convert(double value);
   public static void main(String[] args)
   {
      System.out.println(C2F + " for 100.0 degrees = " +
                         C2F.convert(100.0));
      System.out.println(F2C + " for 98.6 degrees = " +
                         F2C.convert(98.6));
   }
}
```

When you run this application, it generates the following output:

```
Celsius to Fahrenheit for 100.0 degrees = 212.0
Fahrenheit to Celsius for 98.6 degrees = 37.0
```

The Enum Class

The compiler regards `enum` as syntactic sugar. When it encounters an enum type declaration (`enum Coin {}`), it generates a class whose name (`Coin`) is specified by the declaration, and which also subclasses the abstract `Enum` class (in the `java.lang` package), the common base class of all Java language–based enumeration types.

If you examine `Enum`'s Java documentation, you will discover that it overrides `Object`'s `clone()`, `equals()`, `finalize()`, `hashCode()`, and `toString()` methods:

- `clone()` is overridden to prevent constants from being cloned so that there is never more than one copy of a constant; otherwise, constants could not be compared via `==`.
- `equals()` is overridden to compare constants via their references—constants with the same identities (`==`) must have the same contents (`equals()`), and different identities imply different contents.
- `finalize()` is overridden to ensure that constants cannot be finalized.
- `hashCode()` is overridden because `equals()` is overridden.
- `toString()` is overridden to return the constant's name.

Except for `toString()`, all of the overridden methods are declared `final` so that they cannot be overridden in a subclass.

`Enum` also provides its own methods. These methods include the `final` `compareTo()`, (`Enum` implements `Comparable`), `getDeclaringClass()`, `name()`, and `ordinal()` methods:

- `compareTo()` compares the current constant with the constant passed as an argument to see which constant precedes the other constant in the enum, and returns a value indicating their order. This method makes it possible to sort an array of unsorted constants.
- `getDeclaringClass()` returns the `Class` object corresponding to the current constant's enum. For example, the `Class` object for `Coin` is returned when calling `Coin.PENNY.getDeclaringClass()` for Listing 5–36's `Coin` enum. Also, `TempConversion` is returned when calling `TempConversion.C2F.getDeclaringClass()` for Listing 5–41's `TempConversion` enum. The `compareTo()` method uses `Class`'s `getClass()` method and `Enum`'s `getDeclaringClass()` method to ensure that only constants belonging to the same enum are compared. Otherwise, a `ClassCastException` is thrown. (I will discuss `Class` in Chapter 7.)
- `name()` returns the constant's name. Unless overridden to return something more descriptive, `toString()` also returns the constant's name.
- `ordinal()` returns a zero-based *ordinal*, an integer that identifies the position of the constant within the enum type. `compareTo()` compares ordinals.

`Enum` also provides the static `valueOf(Class<T>enumType, String name)` method for returning the enum constant from the specified enum with the specified name:

- `enumType` identifies the `Class` object of the enum from which to return a constant.
- `name` identifies the name of the constant to return.

For example, `Coin penny = Enum.valueOf(Coin.class, "PENNY");` assigns the `Coin` constant whose name is `PENNY` to `penny`.

You will not discover a `values()` method in `Enum`'s Java documentation because the compiler *synthesizes* (manufactures) this method while generating the class.

Extending the Enum Class

`Enum`'s generic type is `Enum<E extends Enum<E>>`. Although the formal type parameter list looks ghastly, it is not that hard to understand. But first, take a look at Listing 5–42.

Listing 5–42. *The `Coin` class as it appears from the perspective of its classfile*

```
public final class Coin extends Enum<Coin>
{
   public static final Coin PENNY = new Coin("PENNY", 0);
   public static final Coin NICKEL = new Coin("NICKEL", 1);
   public static final Coin DIME = new Coin("DIME", 2);
   public static final Coin QUARTER = new Coin("QUARTER", 3);
   private static final Coin[] $VALUES = { PENNY, NICKEL, DIME, QUARTER };
   public static Coin[] values()
```

```
   {
      return Coin.$VALUES.clone();
   }
   public static Coin valueOf(String name)
   {
      return Enum.valueOf(Coin.class, "Coin");
   }
   private Coin(String name, int ordinal)
   {
      super(name, ordinal);
   }
}
```

Behind the scenes, the compiler converts Listing 5–36's `Coin` enum declaration into a class declaration that is similar to Listing 5–42.

The following rules show you how to interpret `Enum<E extends Enum<E>>` in the context of `Coin extends Enum<Coin>`:

- Any subclass of `Enum` must supply an actual type argument to `Enum`. For example, `Coin`'s header specifies `Enum<Coin>`.
- The actual type argument must be a subclass of `Enum`. For example, `Coin` is a subclass of `Enum`.
- A subclass of `Enum` (such as `Coin`) must follow the idiom that it supplies its own name (`Coin`) as an actual type argument.

The third rule allows `Enum` to declare methods—`compareTo()`, `getDeclaringClass()`, and `valueOf()`—whose parameter and/or return types are specified in terms of the subclass (`Coin`), and not in terms of `Enum`.

The rationale for doing this is to avoid having to specify casts. For example, you do not need to cast `valueOf()`'s return value to `Coin` in `Coin penny = Enum.valueOf(Coin.class, "PENNY");`.

> **NOTE:** You cannot compile Listing 5–42 because the compiler will not compile any class that extends `Enum`. It will also complain about `super(name, ordinal);`.

EXERCISES

The following exercises are designed to test your understanding of assertions, annotations, generics, and enums:

1. What is an assertion?
2. When would you use assertions?
3. True or false: Specifying the `-ea` command-line option with no argument enables all assertions, including system assertions.
4. What is an annotation?

5. What kinds of application elements can be annotated?
6. Identify the three compiler-supported annotation types.
7. How do you declare an annotation type?
8. What is a marker annotation?
9. What is an element?
10. How do you assign a default value to an element?
11. What is a meta-annotation?
12. Identify Java's four meta-annotation types.
13. Define generics.
14. Why would you use generics?
15. What is the difference between a generic type and a parameterized type?
16. Which one of the nonstatic member class, local class, and anonymous class inner class categories cannot be generic?
17. Identify the five kinds of actual type arguments.
18. True or false: You cannot specify a primitive type name (such as `double` or `int`) as an actual type argument.
19. What is a raw type?
20. When does the compiler report an unchecked warning message and why?
21. How do you suppress an unchecked warning message?
22. True or false: `List<E>`'s `E` type parameter is unbounded.
23. How do you specify a single upper bound?
24. True or false: `MyList<E super Circle>` specifies that the `E` type parameter has a lower bound of `Circle`.
25. What is a recursive type bound?
26. Why are wildcard type arguments necessary?
27. What is reification?
28. True or false: Type parameters are reified.
29. What is erasure?
30. What is a generic method?
31. In Listing 5–43, which overloaded method does the `methodCaller()` generic method call?

Listing 5–43. ***Which*** *`someOverloadedMethod()`* ***is called?***

```
import java.util.Date;

public class CallOverloadedNGMethodFromGMethod
{
```

```
   public static void someOverloadedMethod(Object o)
   {
      System.out.println("call to someOverloadedMethod(Object o)");
   }
   public static void someOverloadedMethod(Date d)
   {
      System.out.println("call to someOverloadedMethod(Date d)");
   }
   public static <T> void methodCaller(T t)
   {
      someOverloadedMethod(t);
   }
   public static void main(String[] args)
   {
      methodCaller(new Date());
   }
}
```

32. What is an enumerated type?

33. Identify three problems that can arise when you use enumerated types whose constants are `int`-based.

34. What is an enum?

35. How do you use the switch statement with an enum?

36. In what ways can you enhance an enum?

37. What is the purpose of the abstract `Enum` class?

38. What is the difference between `Enum`'s `name()` and `toString()` methods?

39. True or false: `Enum`'s generic type is `Enum<E extends Enum<E>>`.

40. Declare a `ToDo` marker annotation type that annotates only type elements, and that also uses the default retention policy.

41. Rewrite the `StubFinder` application to work with Listing 5–15's `Stub` annotation type (with appropriate `@Target` and `@Retention` annotations) and Listing 5–16's `Deck` class.

42. Implement a stack in a manner that is similar to Listing 5–24's `Queue` class. `Stack` must be generic, it must declare `push()`, `pop()`, and `isEmpty()` methods (it could also declare an `isFull()` method but that method is not necessary in this exercise), `push()` must throw a `StackFullException` instance when the stack is full, and `pop()` must throw a `StackEmptyException` instance when the stack is empty. (You must create your own package-private `StackFullException` and `StackEmptyException` helper classes because they are not provided for you in Java's class library.) Declare a similar `main()` method, and insert two assertions into this method that validate your assumptions about the stack being empty immediately after being created and immediately after popping the last element.

NOTE: A *stack* is a data structure that stores elements in a last-in, first-out order. Elements are added to the stack via an operation known as *push*. They are removed from the stack via an operation known as *pop*. The last element pushed onto the stack is the first element popped off of the stack.

43. Declare a `Compass` enum with `NORTH`, `SOUTH`, `EAST`, and `WEST` members. Declare a `UseCompass` class whose `main()` method randomly selects one of these constants and then switches on that constant. Each of the switch statement's cases should output a message such as `heading north`.

Summary

An assertion is a statement that lets you express an assumption of application correctness via a Boolean expression. If this expression evaluates to true, execution continues with the next statement. Otherwise, an error that identifies the cause of failure is thrown.

There are many situations where assertions should be used. These situations organize into internal invariant, control-flow invariant, and design-by-contract categories. An invariant is something that does not change.

Although there are many situations where assertions should be used, there also are situations where they should be avoided. For example, you should not use assertions to check the arguments that are passed to public methods.

The compiler records assertions in the classfile. However, assertions are disabled at runtime because they can affect performance. You must enable the classfile's assertions before you can test assumptions about the behaviors of your classes.

Annotations are instances of annotation types and associate metadata with application elements. They are expressed in source code by prefixing their type names with `@` symbols. For example, `@Readonly` is an annotation and `Readonly` is its type.

Java supplies a wide variety of annotation types, including the compiler-oriented `Override`, `Deprecated`, and `SuppressWarnings` types. However, you can also declare your own annotation types by using the `@interface` syntax.

Annotation types can be annotated with meta-annotations that identify the application elements they can target (such as constructors, methods, or fields), their retention policies, and other characteristics.

Annotations whose types are assigned a runtime retention policy via `@Retention` annotations can be processed at runtime using custom applications or Java's `apt` tool, whose functionality has been integrated into the compiler starting with Java version 6.

Java version 5 introduced generics, language features for declaring and using type-agnostic classes and interfaces. When working with Java's collections framework, these features help you avoid `ClassCastException`s.

A generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list. A generic method is a `static` or non-`static` method with a type-generalized implementation.

An enumerated type is a type that specifies a named sequence of related constants as its legal values. Java developers have traditionally used sets of named integer constants to represent enumerated types.

Because sets of named integer constants have proven to be problematic, Java version 5 introduced the enum alternative. An enum is an enumerated type that is expressed via reserved word `enum`.

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. Also, you can override `toString()` to provide a more useful description of a constant's value, and subclass constants to assign different behaviors.

The compiler regards `enum` as syntactic sugar for a class that subclasses `Enum`. This abstract class overrides various `Object` methods to provide default behaviors (usually for safety reasons), and provides additional methods for various purposes.

This chapter largely completes our tour of the Java language. However, there are a few more advanced language features to explore. You will encounter one of these minor features in Chapter 6, which begins a multichapter exploration of various types that are located in Java SE's/Android's standard class library.