Chapter 16 Exceptions, Assertions, and Localization

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Exception Handling and Assertions**
  - Use the try-with-resources construct
  - Create and use custom exception classes
  - Test invariants by using assertions
- **Localization**
  - Use the Locale class
  - Use resource bundles
  - Format messages, dates, and numbers with Java

This chapter is about creating applications that adapt to change. What happens if a user enters invalid data on a web page, or our connection to a database goes down in the middle of a sale? How do we ensure rules about our data are enforced? Finally, how do we build applications that can support multiple languages or geographic regions?

In this chapter, we will discuss these problems and solutions to them using exceptions, assertions, and localization. One way to make sure your applications respond to change is to build in support early on. For example, supporting localization doesn't mean you actually need to support

multiple languages right away. It just means your application can be more easily adapted in the future. By the end of this chapter, we hope we've provided structure for designing applications that better adapt to change.

Reviewing Exceptions

An *exception* is Java's way of saying, “I give up. I don't know what to do right now. You deal with it.” When you write a method, you can either deal with the exception or make it the calling code's problem. In this section, we cover the fundamentals of exceptions.

If you've recently studied for the 1Z0-815 exam, then you can probably skip this section and go straight to “Creating Custom Exceptions.” This section is meant only for review.

Handling Exceptions

A *try statement* is used to handle exceptions. It consists of a `try` clause, zero or more *catch clauses* to handle the exceptions that are thrown, and an optional *finally clause*, which runs regardless of whether an exception is thrown. Figure 16.1 shows the syntax of a `try` statement.

A traditional `try` statement must have at least one of the following: a `catch` block or a `finally` block. It can have more than one `catch` block, including multi-`catch` blocks, but at most one `finally` block.

Swallowing an exception is when you handle it with an empty `catch` block. When presenting a topic, we often do this to keep things simple. Please, *never do this in practice!* Oftentimes, it is added by developers who do not want to handle or declare an exception properly and can lead to bugs in production code.

```
try {

    // Protected code

} catch (IOException e) {
                        ← Exception identifier
    // Handler for IOException

} catch (ArithmeticException | IllegalArgumentException e) {
                                                          ← Exception identifier
    // Multi-catch handler
                        ← Catch either of these exceptions.

} finally {

    // Always runs after try/catch blocks are finished

}
```

FIGURE 16.1 The syntax of a `try` statement

You can also create a *try-with-resources* statement to handle exceptions. A try-with-resources statement looks a lot like a `try` statement, except that it includes a list of resources inside a set of parentheses, `()`. These resources are automatically closed in the reverse order that they are declared at the conclusion of the `try` clause. The syntax of the try-with-resources statement is presented in Figure 16.2.

Like a regular `try` statement, a try-with-resources statement can include optional `catch` and `finally` blocks. Unlike a `try` statement, though, neither is required. We'll cover try-with-resources statements in more detail in this chapter.

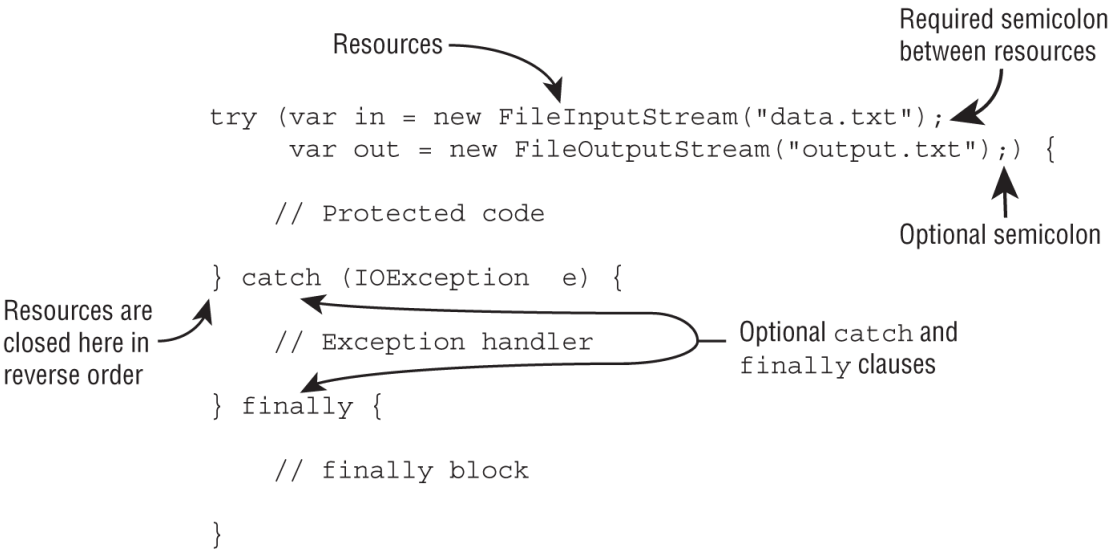

FIGURE 16.2 The syntax of a try-with-resources statement

Did you notice we used `var` for the resource type? While `var` is not required, it is convenient when working with streams, database objects, and especially generics, whose declarations can be lengthy.

While presenting try-with-resources statements, we include a number of examples that use I/O stream classes that we'll be covering later in this book. For this chapter, you can assume these resources are declared correctly. For example, you can assume the previous code snippet correctly creates `FileInputStream` and `FileOutputStream` objects. If you see a try-with-resources statement with an I/O stream on the exam, though, it could be testing either topic.

Distinguishing between throw and throws

By now, you should know the difference between `throw` and `throws`. The `throw` keyword means an exception is actually being thrown, while the `throws` keyword indicates that the method merely has the potential to throw that exception. The following example uses both:

```
10: public String getDataFromDatabase() throws SQLException {
11:    throw new UnsupportedOperationException();
12: }
```

Line 10 declares that the method might or might not throw a `SQLException`. Since this is a checked exception, the caller needs to handle or declare it. Line 11 actually does throw an `UnsupportedOperationException`. Since this is a runtime exception, it does not need to be declared on line 10.

Examining Exception Categories

In Java, all exceptions inherit from `Throwable`, although in practice, the only ones you should be handling or declaring extend from the `Exception` class. Figure 16.3 reviews the hierarchy of the top-level exception classes.

To begin with, a *checked exception* must be handled or declared by the application code where it is thrown. The *handle or declare rule* dictates that a checked exception must be either caught in a `catch` block or thrown to the caller by including it in the method declaration.

The `ZooMaintenance` class shows an example of a method that handles an exception, and one that declares an exception.

```
public class ZooMaintenance {
   public void open() {
      try {
         throw new Exception();
      } catch (Exception e) {
         // Handles exception
      }
   }

   public void close() throws Exception {  // Declares exceptions
      throw new Exception();
```

```
    }
}
```

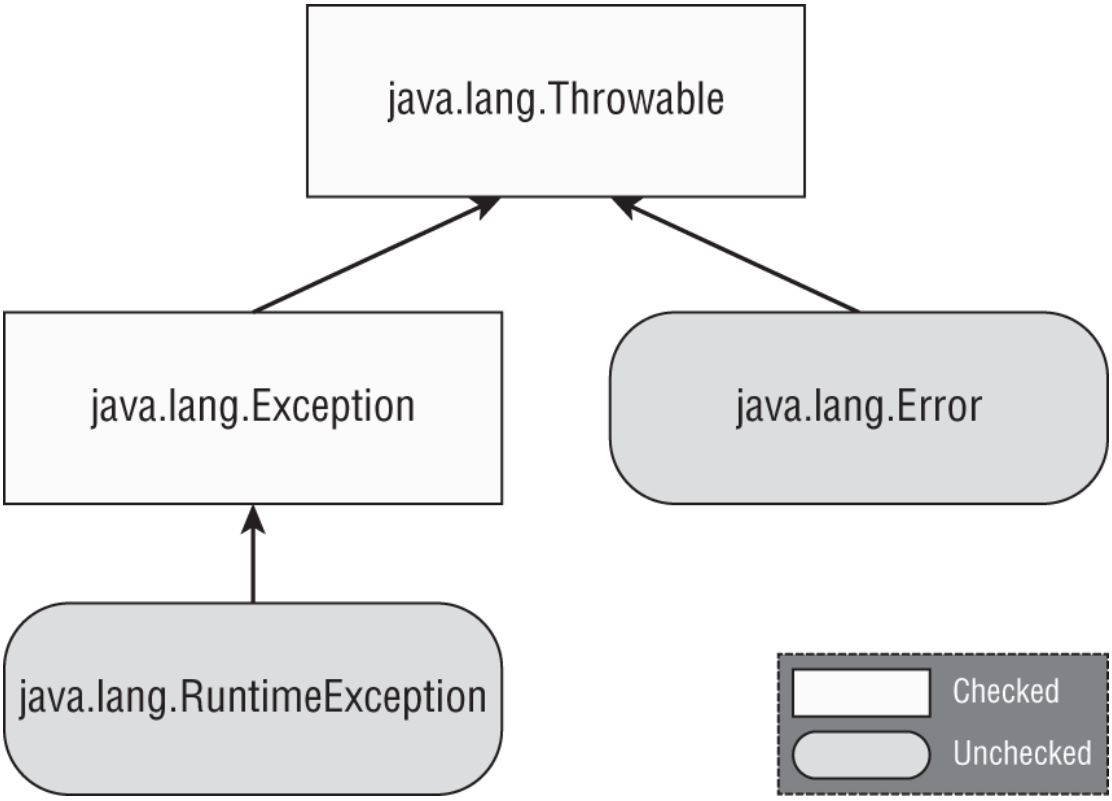

FIGURE 16.3 Categories of exceptions

In Java, all exceptions that inherit `Exception` but not `RuntimeException` are considered checked exceptions.

On the other hand, an *unchecked exception* does not need to be handled or declared. Unchecked exceptions are often referred to as runtime exceptions, although in Java unchecked exceptions include any class that inherits `RuntimeException` or `Error`. An `Error` is fatal, and it is considered a poor practice to catch it.

For the exam, you need to memorize some common exception classes. You also need to know whether they are checked or unchecked exceptions. Table 16.1 lists the unchecked exceptions that inherit `RuntimeException` that you should be familiar with for the exam.

TABLE 16.1 Unchecked exceptions

`ArithmeticException`	`ArrayIndexOutOfBoundsException`
`ArrayStoreException`	`ClassCastException`
`IllegalArgumentException`	`IllegalStateException`
`MissingResourceException`	`NullPointerException`
`NumberFormatException`	`UnsupportedOperationException`

Table 16.2 presents the checked exceptions you should also be familiar with.

TABLE 16.2 Checked exceptions

`FileNotFoundException`	`IOException`
`NotSerializableException`	`ParseException`
`SQLException`	

Inheriting Exception Classes

When evaluating `catch` blocks, the inheritance of the exception types can be important. For the exam, you should know that `NumberFormatException` inherits from `IllegalArgumentException`. You should also know that `FileNotFoundException` and `NotSerializableException` both inherit from `IOException`.

This comes up often in multi-`catch` expressions. For example, why does the following not compile?

```
try {
   throw new IOException();
} catch (IOException | FileNotFoundException e) {} // DOES NOT COMPILE
```

Since `FileNotFoundException` is a subclass of `IOException`, listing both in a multi-`catch` expression is redundant, resulting in a compilation error.

Ordering of exceptions in consecutive `catch` blocks matters too. Do you understand why the following does not compile?

```
try {
   throw new IOException();
} catch (IOException e) {
} catch (FileNotFoundException e) {} // DOES NOT COMPILE
```

For the exam, remember that trying to catch a more specific exception (after already catching a broader exception) results in unreachable code and a compiler error.

If you're a bit rusty on exceptions, then you may want to review Chapter 10, “Exceptions”. The rest of this book assumes you know the basics of exception handling.

Creating Custom Exceptions

Java provides many exception classes out of the box. Sometimes, you want to write a method with a more specialized type of exception. You can create your own exception class to do this.

Declaring Exception Classes

When creating your own exception, you need to decide whether it should be a checked or unchecked exception. While you can extend any exception class, it is most common to extend `Exception` (for checked) or `RuntimeException` (for unchecked).

Creating your own exception class is really easy. Can you figure out whether the exceptions are checked or unchecked in this example?

```
1: class CannotSwimException extends Exception {}
2: class DangerInTheWater extends RuntimeException {}
3: class SharkInTheWaterException extends DangerInTheWater {}
4: class Dolphin {
5:    public void swim() throws CannotSwimException {
6:       // logic here
7:    }
8: }
```

On line 1, we have a checked exception because it extends directly from `Exception` . Not being able to swim is pretty bad when we are trying to swim, so we want to force callers to deal with this situation. Line 2 declares an unchecked exception because it extends directly from `RuntimeException` . On line 3, we have another unchecked exception because it extends indirectly from `RuntimeException` . It is pretty unlikely that there will be a shark in the water. We might even be swimming in a pool where the odds of a shark are 0 percent! We don't want to force the caller to deal with everything that might remotely happen, so we leave this as an unchecked exception.

The method on lines 5–7 declares that it might throw the checked `CannotSwimException` . The method implementation could be written to actually throw it or not. The method implementation could also be written to throw a `SharkInTheWaterException` , an `ArrayIndexOutOfBoundsException` , or any other runtime exception.

Adding Custom Constructors

These one-liner exception declarations are pretty useful, especially on the exam where they need to communicate quickly whether an exception is checked or unchecked. Let's see how to pass more information in your exception.

The following example shows the three most common constructors defined by the `Exception` class:

```
public class CannotSwimException extends Exception {
   public CannotSwimException() {
      super();  // Optional, compiler will insert automatically
   }
   public CannotSwimException(Exception e) {
      super(e);
   }
   public CannotSwimException(String message) {
      super(message);
   }
}
```

The first constructor is the default constructor with no parameters. The second constructor shows how to wrap another exception inside yours. The third constructor shows how to pass a custom error message.

Remember from Chapter 8, "Class Design," that the default no-argument constructor is provided automatically if you don't write any constructors of your own.

In these examples, our constructors and parent constructors took the same parameters, but this is certainly not required. For example, the following constructor takes an `Exception` and calls the parent constructor that takes a `String`:

```
    public CannotSwimException(Exception e) {
       super("Cannot swim because: " + e.toString());
    }
```

Using a different constructor allows you to provide more information about what went wrong. For example, let's say we have a `main()` method with the following line:

```
15: public static void main(String[] unused) throws Exception {
16:    throw new CannotSwimException();
17: }
```

The output for this method is as follows:

```
Exception in thread "main" CannotSwimException
   at CannotSwimException.main(CannotSwimException.java:16)
```

The JVM gives us just the exception and its location. Useful, but we could get more. Now, let's change the `main()` method to include some text, as shown here:

```
15: public static void main(String[] unused) throws Exception {
16:    throw new CannotSwimException("broken fin");
17: }
```

The output of this new `main()` method is as follows:

```
Exception in thread "main" CannotSwimException: broken fin
   at CannotSwimException.main(CannotSwimException.java:16)
```

This time we see the message text in the result. You might want to provide more information about the exception depending on the problem.

We can even pass another exception, if there is an underlying cause for the exception. Take a look at this version of our `main()` method:

```
15: public static void main(String[] unused) throws Exception {
16:    throw new CannotSwimException(
17:       new FileNotFoundException("Cannot find shark file"));
18: }
```

This would yield the longest output so far:

```
Exception in thread "main" CannotSwimException:
   java.io.FileNotFoundException: Cannot find shark file
   at CannotSwimException.main(CannotSwimException.java:16)
Caused by: java.io.FileNotFoundException: Cannot find shark file
   … 1 more
```

Printing Stack Traces

The error messages that we've been showing are called *stack traces*. A stack trace shows the exception along with the method calls it took to get there. The JVM automatically prints a stack trace when an exception is thrown that is not handled by the program.

You can also print the stack trace on your own. The advantage is that you can read or log information from the stack trace and then continue to handle or even rethrow it.

```
try {
   throw new CannotSwimException();
} catch (CannotSwimException e) {
   e.printStackTrace();
}
```

Automating Resource Management

As previously described, a try-with-resources statement ensures that any resources declared in the `try` clause are automatically closed at the conclusion of the `try` block. This feature is also known as *automatic resource management*, because Java automatically takes care of closing the resources for you.

For the exam, a *resource* is typically a file or a database that requires some kind of stream or connection to read or write data. In Chapter 19, "I/O," Chapter 20, "NIO.2," and Chapter 21, "JDBC," you'll create numerous resources that will need to be closed when you are finished with them. In Chapter 22, "Security," we'll discuss how failure to close resources can lead to resource leaks that could make your program more vulnerable to attack.

RESOURCE MANAGEMENT VS. GARBAGE COLLECTION

Java has great built-in support for garbage collection. When you are finished with an object, it will automatically (over time) reclaim the memory associated with it.

The same is not true for resource management without a try-with-resources statement. If an object connected to a resource is not closed, then the connection could remain open. In fact, it may interfere with Java's ability to garbage collect the object.

To eliminate this problem, it is recommended that you close resources in the same block of code that opens them. By using a try-with-resources statement to open all your resources, this happens automatically.

Constructing Try-With-Resources Statements

What types of resources can be used with a try-with-resources statement? The first rule you should know is: *try-with-resources statements require resources that implement the `AutoCloseable` interface*. For example, the

following does not compile as `String` does not implement the `AutoCloseable` interface:

```
try (String reptile = "lizard") {
}
```

Inheriting `AutoCloseable` requires implementing a compatible `close()` method.

```
interface AutoCloseable {
   public void close() throws Exception;
}
```

From your studies of method overriding, this means that the implemented version of `close()` can choose to throw `Exception` or a subclass, or not throw any exceptions at all.

In Chapter 19 and Chapter 20, you will encounter resources that implement `Closeable`, rather than `AutoCloseable`. Since `Closeable` extends `AutoCloseable`, they are both supported in try-with-resources statements. The only difference between the two is that `Closeable`'s `close()` method declares `IOException`, while `AutoCloseable`'s `close()` method declares `Exception`.

Let's define our own custom resource class for use in a try-with-resources statement.

```
public class MyFileReader implements AutoCloseable {
   private String tag;
   public MyFileReader(String tag) { this.tag = tag;}
```

```
    @Override public void close() {
       System.out.println("Closed: "+tag);
    }
}
```

The following code snippet makes use of our custom reader class:

```
try (var bookReader = new MyFileReader("monkey")) {
   System.out.println("Try Block");
} finally {
   System.out.println("Finally Block");
}
```

The code prints the following at runtime:

```
Try Block
Closed: monkey
Finally Block
```

As you can see, the resources are closed at the end of the `try` statement, before any `catch` or `finally` blocks are executed. Behind the scenes, the JVM calls the `close()` method inside a hidden `finally` block, which we can refer to as the *implicit* `finally` block. The `finally` block that the programmer declared can be referred to as the *explicit* `finally` block.

In a try-with-resources statement, you need to remember that the resource will be closed at the completion of the `try` block, before any declared `catch` or `finally` blocks execute.

The second rule you should be familiar with is: *a try-with-resources statement can include multiple resources, which are closed in the reverse order*

in which they are declared. Resources are terminated by a semicolon (`;`), with the last one being optional.

Consider the following code snippet:

```
try (var bookReader = new MyFileReader("1");
     var movieReader = new MyFileReader("2");
     var tvReader = new MyFileReader("3");) {
   System.out.println("Try Block");
} finally {
   System.out.println("Finally Block");
}
```

When executed, this code prints the following:

```
Try Block
Closed: 3
Closed: 2
Closed: 1
Finally Block
```

Real World Scenario

WHY YOU SHOULD BE USING TRY-WITH-RESOURCES STATEMENTS

If you have been working with files and databases and have never used try-with-resources before, you've definitely been missing out. For example, consider this code sample, which does not use automatic resource management:

```
11: public void copyData(Path path1, Path path2) throws Exception {
12:    BufferedReader in = null;
13:    BufferedWriter out = null;
14:    try {
15:       in = Files.newBufferedReader(path1);
16:       out = Files.newBufferedWriter(path2);
17:       out.write(in.readLine());
18:    } finally {
19:       if (out != null) {
20:          out.close();
21:       }
22:       if (in != null) {
23:          in.close();
24:       }
25:    }
26: }
```

Switching to the try-with-resources syntax, we can replace it with the following, much shorter implementation:

```
11: public void copyData(Path path1, Path path2) throws Exception {
12:    try (var in = Files.newBufferedReader(path1);
13:         var out = Files.newBufferedWriter(path2)) {
14:       out.write(in.readLine());
15:    }
16: }
```

Excluding the method declaration, that's 14 lines of code to do something that can be done in 4 lines of code. In fact, the first version even

contains a bug! If `out.close()` throws an exception on line 20, the `in` resource will never be closed. The `close()` statements would each need to be wrapped in a `try` / `catch` block to ensure the resources were properly closed.

The final rule you should know is: *resources declared within a try-with-resources statement are in scope only within the `try` block.*

This is another way to remember that the resources are closed before any `catch` or `finally` blocks are executed, as the resources are no longer available. Do you see why lines 6 and 8 don't compile in this example?

```
3: try (Scanner s = new Scanner(System.in)) {
4:    s.nextLine();
5: } catch(Exception e) {
6:    s.nextInt(); // DOES NOT COMPILE
7: } finally {
8:    s.nextInt(); // DOES NOT COMPILE
9: }
```

The problem is that `Scanner` has gone out of scope at the end of the `try` clause. Lines 6 and 8 do not have access to it. This is actually a nice feature. You can't accidentally use an object that has been closed.

Resources do not need to be declared inside a try-with-resources statement, though, as we will see in the next section.

Learning the New Effectively Final Feature

Starting with Java 9, it is possible to use resources declared prior to the try-with-resources statement, provided they are marked `final` or effectively final. The syntax is just to use the resource name in place of the resource declaration, separated by a semicolon (`;`).

```
11: public void relax() {
12:    final var bookReader = new MyFileReader("4");
13:    MyFileReader movieReader = new MyFileReader("5");
14:    try (bookReader;
15:         var tvReader = new MyFileReader("6");
16:         movieReader) {
17:       System.out.println("Try Block");
18:    } finally {
19:       System.out.println("Finally Block");
20:    }
21: }
```

Let's take this one line at a time. Line 12 declares a `final` variable `bookReader`, while line 13 declares an effectively final variable `movieReader`. Both of these resources can be used in a try-with-resources statement. We know `movieReader` is effectively final because it is a local variable that is assigned a value only once. Remember, the test for effectively final is that if we insert the `final` keyword when the variable is declared, the code still compiles.

Lines 14 and 16 use the new syntax to declare resources in a try-with-resources statement, using just the variable name and separating the resources with a semicolon (`;`). Line 15 uses the normal syntax for declaring a new resource within the `try` clause.

On execution, the code prints the following:

```
Try Block
Closed: 5
Closed: 6
Closed: 4
Finally Block
```

If you come across a question on the exam that uses a try-with-resources statement with a variable not declared in the `try` clause, make sure it is effectively final. For example, the following does not compile:

```
31: var writer = Files.newBufferedWriter(path);
32: try(writer) {  // DOES NOT COMPILE
33:    writer.append("Welcome to the zoo!");
34: }
35: writer = null;
```

The `writer` variable is reassigned on line 35, resulting in the compiler not considering it effectively final. Since it is not an effectively final variable, it cannot be used in a try-with-resources statement on line 32.

The other place the exam might try to trick you is accessing a resource after it has been closed. Consider the following:

```
41: var writer = Files.newBufferedWriter(path);
42: writer.append("This write is permitted but a really bad idea!");
43: try(writer) {
44:    writer.append("Welcome to the zoo!");
45: }
46: writer.append("This write will fail!");  // IOException
```

This code compiles but throws an exception on line 46 with the message `Stream closed`. While it was possible to write to the resource before the try-with-resources statement, it is not afterward.

TAKE CARE WHEN USING RESOURCES DECLARED BEFORE TRY-WITH-RESOURCES STATEMENTS

On line 42 of the previous code sample, we used `writer` before the try-with-resources statement. While this is allowed, it's a really bad idea. What happens if line 42 throws an exception? In this case, the resource declared on line 41 will *never* be closed! What about the following code snippet?

```
51: var reader = Files.newBufferedReader(path1);
52: var writer = Files.newBufferedWriter(path2);  // Don't do this!
53: try (reader; writer) {}
```

It has the same problem. If line 52 throws an exception, such as the file cannot be found, then the resource declared on line 51 will never be closed. We recommend you use this new syntax sparingly or with only one resource at a time. For example, if line 52 was removed, then the resource created on line 51 wouldn't have an opportunity to throw an exception before entering the automatic resource management block.

Understanding Suppressed Exceptions

What happens if the `close()` method throws an exception? Let's try an illustrative example:

```
public class TurkeyCage implements AutoCloseable {
   public void close() {
      System.out.println("Close gate");
   }
   public static void main(String[] args) {
      try (var t = new TurkeyCage()) {
         System.out.println("Put turkeys in");
      }
   }
}
```

If the `TurkeyCage` doesn't close, the turkeys could all escape. Clearly, we need to handle such a condition. We already know that the resources are closed before any programmer-coded `catch` blocks are run. This means we can catch the exception thrown by `close()` if we want. Alternatively, we can allow the caller to deal with it.

Let's expand our example with a new `JammedTurkeyCage` implementation, shown here:

```
1:  public class JammedTurkeyCage implements AutoCloseable {
2:     public void close() throws IllegalStateException {
3:        throw new IllegalStateException("Cage door does not close");
4:     }
5:     public static void main(String[] args) {
6:        try (JammedTurkeyCage t = new JammedTurkeyCage()) {
7:           System.out.println("Put turkeys in");
8:        } catch (IllegalStateException e) {
9:           System.out.println("Caught: " + e.getMessage());
10:       }
11:    }
12: }
```

The `close()` method is automatically called by try-with-resources. It throws an exception, which is caught by our `catch` block and prints the following:

```
Caught: Cage door does not close
```

This seems reasonable enough. What happens if the `try` block also throws an exception? When multiple exceptions are thrown, all but the first are called *suppressed exceptions*. The idea is that Java treats the first exception as the primary one and tacks on any that come up while automatically closing.

What do you think the following implementation of our `main()` method outputs?

```
5:     public static void main(String[] args) {
6:        try (JammedTurkeyCage t = new JammedTurkeyCage()) {
7:           throw new IllegalStateException("Turkeys ran off");
8:        } catch (IllegalStateException e) {
9:           System.out.println("Caught: " + e.getMessage());
10:          for (Throwable t: e.getSuppressed())
11:             System.out.println("Suppressed: "+t.getMessage());
12:       }
13:    }
```

Line 7 throws the primary exception. At this point, the `try` clause ends, and Java automatically calls the `close()` method. Line 3 of `JammedTurkeyCage` throws an `IllegalStateException`, which is added as a suppressed exception. Then line 8 catches the primary exception. Line 9 prints the message for the primary exception. Lines 10–11 iterate through any suppressed exceptions and print them. The program prints the following:

```
Caught: Turkeys ran off
Suppressed: Cage door does not close
```

Keep in mind that the `catch` block looks for matches on the primary exception. What do you think this code prints?

```
5:     public static void main(String[] args) {
6:        try (JammedTurkeyCage t = new JammedTurkeyCage()) {
7:           throw new RuntimeException("Turkeys ran off");
8:        } catch (IllegalStateException e) {
9:           System.out.println("caught: " + e.getMessage());
10:       }
11:    }
```

Line 7 again throws the primary exception. Java calls the `close()` method and adds a suppressed exception. Line 8 would catch the `IllegalStateException`. However, we don't have one of those. The primary exception is a `RuntimeException`. Since this does not match the

`catch` clause, the exception is thrown to the caller. Eventually the `main()` method would output something like the following:

```
Exception in thread "main" java.lang.RuntimeException: Turkeys ran off
   at JammedTurkeyCage.main(JammedTurkeyCage.java:7)
   Suppressed: java.lang.IllegalStateException:
         Cage door does not close
      at JammedTurkeyCage.close(JammedTurkeyCage.java:3)
      at JammedTurkeyCage.main(JammedTurkeyCage.java:8)
```

Java remembers the suppressed exceptions that go with a primary exception even if we don't handle them in the code.

If more than two resources throw an exception, the first one to be thrown becomes the primary exception, with the rest being grouped as suppressed exceptions. And since resources are closed in reverse order in which they are declared, the primary exception would be on the last declared resource that throws an exception.

Keep in mind that suppressed exceptions apply only to exceptions thrown in the `try` clause. The following example does not throw a suppressed exception:

```
5:     public static void main(String[] args) {
6:        try (JammedTurkeyCage t = new JammedTurkeyCage()) {
7:           throw new IllegalStateException("Turkeys ran off");
8:        } finally {
9:           throw new RuntimeException("and we couldn't find them");
10:       }
11:    }
```

Line 7 throws an exception. Then Java tries to close the resource and adds a suppressed exception to it. Now we have a problem. The `finally` block runs after all this. Since line 9 also throws an exception, the previous exception from line 7 is lost, with the code printing the following:

```
Exception in thread "main" java.lang.RuntimeException:
   and we couldn't find them
   at JammedTurkeyCage.main(JammedTurkeyCage.java:9)
```

This has always been and continues to be bad programming practice. We don't want to lose exceptions! Although out of scope for the exam, the reason for this has to do with backward compatibility. Automatic resource management was added in Java 7, and this behavior existed before this feature was added.

Declaring Assertions

An *assertion* is a `boolean` expression that you place at a point in your code where you expect something to be `true` . An *assert statement* contains this statement along with an optional message.

An assertion allows for detecting defects in the code. You can turn on assertions for testing and debugging while leaving them off when your program is in production.

Why assert something when you know it is true? It is true only when everything is working properly. If the program has a defect, it might not actually be true. Detecting this earlier in the process lets you know something is wrong.

In the following sections, we cover the syntax for using an assertion, how to turn assertions on/off, and some common uses of assertions.

Real World Scenario

ASSERTIONS VS. UNIT TESTS

Most developers are more familiar with unit test frameworks, such as JUnit, than with assertions. While there are some similarities, assertions are commonly used to verify the internal state of a program, while unit tests are most frequently used to verify behavior.

Additionally, unit test frameworks tend to be fully featured with lots of options and tools available. While you need to know assertions for the exam, you are far better off writing unit tests when programming professionally.

Validating Data with the assert Statement

The syntax for an `assert` statement has two forms, shown in Figure 16.4.

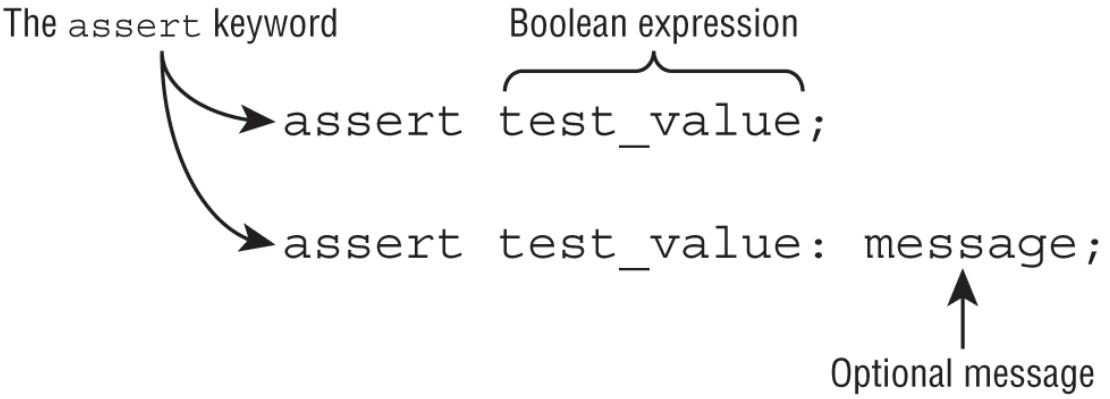

FIGURE 16.4 The syntax of `assert` statements

When assertions are enabled and the `boolean` expression evaluates to `false`, then an `AssertionError` will be thrown at runtime. Since programs aren't supposed to catch an `Error`, this means that assertion failures are fatal and end the program!

Since Java 1.4, `assert` is a keyword. That means it can't be used as an identifier at compile time, even if assertions will be disabled at runtime. Keep an eye out for questions on the exam that use it as anything other than a statement.

Assertions may include optional parentheses and a message. For example, each of the following is valid:

```
assert 1 == age;
assert(2 == height);
assert 100.0 == length : "Problem with length";
assert ("Cecelia".equals(name)): "Failed to verify user data";
```

When provided, the error message will be sent to the `AssertionError` constructor. It is commonly a `String`, although it can be any value.

RECOGNIZING ASSERTION SYNTAX ERRORS

While the preceding examples demonstrate the appropriate syntax, the exam may try to trick you with invalid syntax. See whether you can determine why the following do not compile:

```
assert(1);
assert x -> true;
assert 1==2 ? "Accept" : "Error";
assert.test(5> age);
```

The first three statements do not compile because they expect a `boolean` value. The last statement does not compile because the syntax is invalid.

The three possible outcomes of an `assert` statement are as follows:

- If assertions are disabled, Java skips the assertion and goes on in the code.
- If assertions are enabled and the `boolean` expression is `true`, then our assertion has been validated and nothing happens. The program continues to execute in its normal manner.
- If assertions are enabled and the `boolean` expression is `false`, then our assertion is invalid and an `AssertionError` is thrown.

Presuming assertions are enabled, an assertion is a shorter way of writing the following:

```
if (!
boolean_expression) throw new AssertionError(
error_message);
```

Let's try an example. Consider the following:

```
1: public class Party {
2:    public static void main(String[] args) {
3:       int numGuests = -5;
4:       assert numGuests> 0;
5:       System.out.println(numGuests);
6:    }
7: }
```

We can enable assertions by executing it using the single-file source-code command, as shown here:

```
java –ea Party.java
```

Uh-oh, we made a typo in our `Party` class. We intended for there to be five guests and not negative five guests. The assertion on line 4 detects

this problem. Java throws the `AssertionError` at this point. Line 5 never runs since an error was thrown.

The program ends with a stack trace similar to this:

```
Exception in thread "main" java.lang.AssertionError
   at asserts.Assertions.main(Assertions.java:4)
```

If we run the same program using the command line `java Party`, we get a different result. The program prints `-5`. Now, in this example, it is pretty obvious what the problem is since the program is only seven lines. In a more complicated program, knowing the state of affairs is more useful.

Enabling Assertions

By default, `assert` statements are ignored by the JVM at runtime. To enable assertions, use the `-enableassertions` flag on the command line.

```
java -enableassertions Rectangle
```

You can also use the shortcut `-ea` flag.

```
java -ea Rectangle
```

Using the `-enableassertions` or `-ea` flag without any arguments enables assertions in all classes (except system classes). You can also enable assertions for a specific class or package. For example, the following command enables assertions only for classes in the `com.demos` package and any subpackages:

```
java -ea:com.demos… my.programs.Main
```

The ellipsis (…) means any class in the specified package or subpackages. You can also enable assertions for a specific class.

```
java -ea:com.demos.TestColors my.programs.Main
```

Enabling assertions is an important aspect of using them, because if assertions are not enabled, `assert` statements are ignored at run-time. Keep an eye out for questions that contain an `assert` statement where assertions are not enabled.

Disabling Assertions

Sometimes you want to enable assertions for the entire application but disable it for select packages or classes. Java offers the `-disableassertions` or `-da` flag for just such an occasion. The following command enables assertions for the `com.demos` package but disables assertions for the `TestColors` class:

```
java -ea:com.demos… -da:com.demos.TestColors my.programs.Main
```

For the exam, make sure you understand how to use the `-ea` and `-da` flags in conjunction with each other.

By default, all assertions are disabled. Then, those items marked with `-ea` are enabled. Finally, all of the remaining items marked with `-da` are disabled.

Applying Assertions

Table 16.3 list some of the common uses of assertions. You won't be asked to identify the type of assertion on the exam. This is just to give you some ideas of how they can be used.

TABLE 16.3 Assertion applications

Usage	Description
Internal invariants	Assert that a value is within a certain constraint, such as `assert x < 0`.
Class invariants	Assert the validity of an object's state. Class invariants are typically `private` methods within the class that return a `boolean`.
Control flow invariants	Assert that a line of code you assume is unreachable is never reached.
Pre-conditions	Assert that certain conditions are met before a method is invoked.
Post-conditions	Assert that certain conditions are met after a method executes successfully.

Writing Assertions Correctly

One of the most important rules you should remember from this section is: *assertions should never alter outcomes*. This is especially true because assertions can, should, and probably will be turned off in a production environment.

For example, the following assertion is not a good design because it alters the value of a variable:

```
int x = 10;
assert ++x> 10; // Not a good design!
```

When assertions are turned on, `x` is incremented to `11`; but when assertions are turned off, the value of `x` is `10`. This is not a good use of assertions because the outcome of the code will be different depending on whether assertions are turned on.

Assertions are used for debugging purposes, allowing you to verify that something that you think is true during the coding phase is actually true at runtime.

Working with Dates and Times

The older Java 8 certification exams required you to know a lot about the Date and Time API. This included knowing many of the various date/time classes and their various methods, how to specify amounts of time with the `Period` and `Duration` classes, and even how to resolve values across time zones with daylight savings.

For the Java 11 exam, none of those topics is in scope, although strangely enough, you still need to know how to format dates. This just means if you see a date/time on the exam, you aren't being tested about it. Before we learn how to format dates, let's learn what they are and how to create them.

Creating Dates and Times

In the real world, we usually talk about dates and times as relative to our current location. For example, “I'll call you at 11 a.m. on Friday morning.” You probably use date and time independently, like this: “The package will arrive by Friday” or “We're going to the movies at 7:15 p.m.” Last but

not least, you might also use a more absolute time when planning a meeting across a time zone, such as "Everyone in Seattle and Philadelphia will join the call at 10:45 EST."

Understanding Date and Time Types

Java includes numerous classes to model the examples in the previous paragraph. These types are listed in Table 16.4.

TABLE 16.4 Date and time types

Class	Description	Example
`java.time.LocalDate`	Date with day, month, year	Birth date
`java.time.LocalTime`	Time of day	Midnight
`java.time.LocalDateTime`	Day and time with no time zone	10 a.m. next Monday
`java.time.ZonedDateTime`	Date and time with a specific time zone	9 a.m. EST on 2/20/2021

Each of these types contains a `static` method called `now()` that allows you to get the current value.

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

Your output is going to depend on the date/time when you run it and where you live, although it should resemble the following:

```
2020-10-14
12:45:20.854
2020-10-14T12:45:20.854
2020-10-14T12:45:20.854-04:00[America/New_York]
```

The first line contains only a date and no time. The second line contains only a time and no date. The time displays hours, minutes, seconds, and fractional seconds. The third line contains both a date and a time. Java uses `T` to separate the date and time when converting `LocalDateTime` to a `String`. Finally, the fourth line adds the time zone offset and time zone. New York is four time zones away from Greenwich mean time (GMT) for half of the year due to daylight savings time.

Using the of() Methods

We can create some date and time values using the `of()` methods in each class.

```
LocalDate date1 = LocalDate.of(2020, Month.OCTOBER, 20);
LocalDate date2 = LocalDate.of(2020, 10, 20);
```

Both pass in the year, month, and date. Although it is good to use the `Month` constants (to make the code easier to read), you can pass the `int` number of the month directly.

While programmers often count from zero, working with dates is one of the few times where it is expected for you to count from 1, just like in the real world.

When creating a time, you can choose how detailed you want to be. You can specify just the hour and minute, or you can include the number of

seconds. You can even include nanoseconds if you want to be very precise (a nanosecond is a billionth of a second).

```
LocalTime time1 = LocalTime.of(6, 15);           // hour and minute
LocalTime time2 = LocalTime.of(6, 15, 30);       // + seconds
LocalTime time3 = LocalTime.of(6, 15, 30, 200);  // + nanoseconds
```

These three times are all different but within a minute of each other. You can combine dates and times in multiple ways.

```
var dt1 = LocalDateTime.of(2020, Month.OCTOBER, 20, 6, 15, 30);

LocalDate date = LocalDate.of(2020, Month.OCTOBER, 20);
LocalTime time = LocalTime.of(6, 15);
var dt2 = LocalDateTime.of(date, time);
```

The `dt1` example shows how you can specify all of the information about the `LocalDateTime` right in the same line. The `dt2` example shows how you can create `LocalDate` and `LocalTime` objects separately and then combine them to create a `LocalDateTime` object.

Real World Scenario

THE FACTORY PATTERN

Did you notice that we did not use a constructor in any of these examples? Rather than use a constructor, creation of these objects is delegated to a `static` factory method. The *factory pattern*, or factory method pattern, is a creational design pattern in which a factory class is used to provide instances of an object, rather than instantiating them directly. Oftentimes, factory methods return instances that are subtypes of the interface or class you are expecting.

We will be using the factory pattern throughout this book (and even later in this chapter!). In some cases, using a constructor may even be prohibited. For example, you cannot call `new LocalDate()` since all of the constructors in this class are `private`.

Formatting Dates and Times

The date and time classes support many methods to get data out of them.

```
LocalDate date = LocalDate.of(2020, Month.OCTOBER, 20);
System.out.println(date.getDayOfWeek());  // TUESDAY
System.out.println(date.getMonth());      // OCTOBER
System.out.println(date.getYear());       // 2020
System.out.println(date.getDayOfYear());  // 294
```

Java provides a class called `DateTimeFormatter` to display standard formats.

```
LocalDate date = LocalDate.of(2020, Month.OCTOBER, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dt = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
```

```
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dt.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

The code snippet prints the following:

```
2020-10-20
11:12:34
2020-10-20T11:12:34
```

The `DateTimeFormatter` will throw an exception if it encounters an incompatible type. For example, each of the following will produce an exception at runtime since it attempts to format a date with a time value, and vice versa:

```
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_DATE));
```

If you don't want to use one of the predefined formats, `DateTimeFormatter` supports a custom format using a date format `String`.

```
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");
System.out.println(dt.format(f));  // October 20, 2020 at 11:12
```

Let's break this down a bit. Java assigns each letter or symbol a specific date/time part. For example, `M` is used for month, while `y` is used for year. And case matters! Using `m` instead of `M` means it will return the minute of the hour, not the month of the year.

What about the number of symbols? The number often dictates the format of the date/time part. Using `M` by itself outputs the minimum number of characters for a month, such as `1` for January, while using `MM` always outputs two digits, such as `01`. Furthermore, using `MMM` prints the three-letter abbreviation, such as `Jul` for July, while `MMMM` prints the full month name.

THE *DATE* AND *SIMPLEDATEFORMAT* CLASSES

When Java introduced the Date and Time API in Java 8, many developers switched to the new classes, such as `DateTimeFormatter`. The exam may include questions with the older date/time classes. For example, the previous code snippet could be written using the `java.util.Date` and `java.text.SimpleDateFormat` classes.

```
DateFormat s = new SimpleDateFormat("MMMM dd, yyyy 'at' hh:mm");
System.out.println(s.format(new Date()));  // October 20, 2020 at 06:15
```

As we said earlier, if you see dates or times on the exam, regardless of whether they are using the old or new APIs, you are not being tested on them. You only need to know how to format them. For the exam, the rules for defining a custom `DateTimeFormatter` and `SimpleDateFormat` symbols are the same.

Learning the Standard Date/Time Symbols

For the exam, you should be familiar enough with the various symbols that you can look at a date/time `String` and have a good idea of what the output will be. Table 16.5 includes the symbols you should be familiar with for the exam.

TABLE 16.5 Common date/time symbols

Symbol	Meaning	Examples
`y`	Year	`20`, `2020`
`M`	Month	`1`, `01`, `Jan`, `January`
`d`	Day	`5`, `05`
`h`	Hour	`9`, `09`
`m`	Minute	`45`
`s`	Second	`52`
`a`	a.m./p.m.	`AM`, `PM`
`z`	Time Zone Name	`Eastern Standard Time`, `EST`
`Z`	Time Zone Offset	`-0400`

Let's try some examples. What do you think the following prints?

```
var dt = LocalDateTime.of(2020, Month.OCTOBER, 20, 6, 15, 30);

var formatter1 = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");
System.out.println(dt.format(formatter1));

var formatter2 = DateTimeFormatter.ofPattern("MM_yyyy_-_dd");
System.out.println(dt.format(formatter2));

var formatter3 = DateTimeFormatter.ofPattern("h:mm z");
System.out.println(dt.format(formatter3));
```

The output is as follows:

```
10/20/2020 06:15:30
10_2020_-_20
Exception in thread "main" java.time.DateTimeException:
   Unable to extract ZoneId from temporal 2020-10-20T06:15:30
```

The first example prints the date, with the month before the day, followed by the time. The second example prints the date in a weird format with extra characters that are just displayed as part of the output.

The third example throws an exception at runtime because the underlying `LocalDateTime` does not have a time zone specified. If `ZonedDateTime` was used instead, then the code would have completed successfully and printed something like `06:15 EDT`, depending on the time zone.

As you saw in the previous example, you need to make sure the format `String` is compatible with the underlying date/time type. Table 16.6 shows which symbols you can use with each of the date/time objects.

TABLE 16.6 Supported date/time symbols

Symbol	LocalDate	LocalTime	LocalDateTime	ZonedDateTime
y	√		√	√
M	√		√	√
d	√		√	√
h		√	√	√
m		√	√	√
s		√	√	√
a		√	√	√
z				√
Z				√

Make sure you know which symbols are compatible with which date/time types. For example, trying to format a month for a `LocalTime` or an hour for a `LocalDate` will result in a runtime exception.

Selecting a *format()* Method

The date/time classes contain a `format()` method that will take a formatter, while the formatter classes contain a `format()` method that will take a date/time value. The result is that either of the following is acceptable:

```
var dateTime = LocalDateTime.of(2020, Month.OCTOBER, 20, 6, 15, 30);
var formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");
```

```
System.out.println(dateTime.format(formatter)); // 10/20/2020 06:15:30
System.out.println(formatter.format(dateTime)); // 10/20/2020 06:15:30
```

These statements print the same value at runtime. Which syntax you use is up to you.

Adding Custom Text Values

What if you want your format to include some custom text values? If you just type it as part of the format `String`, the formatter will interpret each character as a date/time symbol. In the best case, it will display weird data based on extra symbols you enter. In the worst case, it will throw an exception because the characters contain invalid symbols. Neither is desirable!

One way to address this would be to break the formatter up into multiple smaller formatters and then concatenate the results.

```
var dt = LocalDateTime.of(2020, Month.OCTOBER, 20, 6, 15, 30);

var f1 = DateTimeFormatter.ofPattern("MMMM dd, yyyy ");
var f2 = DateTimeFormatter.ofPattern(" hh:mm");
System.out.println(dt.format(f1) + "at" + dt.format(f2));
```

This prints `October 20, 2020 at 06:15` at runtime.

While this works, it could become difficult if there are a lot of text values and date symbols intermixed. Luckily, Java includes a much simpler solution. You can *escape* the text by surrounding it with a pair of single quotes (`'`). Escaping text instructs the formatter to ignore the values inside the single quotes and just insert them as part of the final value. We saw this earlier with the `'at'` inserted into the formatter.

```
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");
System.out.println(dt.format(f));  // October 20, 2020 at 06:15
```

But what if you need to display a single quote in the output too? Welcome to the fun of escaping characters! Java supports this by putting two single quotes next to each other.

We conclude our discussion of date formatting with some various examples of formats and their output that rely on text values, shown here:

```
var g1 = DateTimeFormatter.ofPattern("MMMM dd', Party''s at' hh:mm");
System.out.println(dt.format(g1)); // October 20, Party's at 06:15

var g2 = DateTimeFormatter.ofPattern("'System format, hh:mm: 'hh:mm");
System.out.println(dt.format(g2)); // System format, hh:mm: 06:15

var g3 = DateTimeFormatter.ofPattern("'NEW! 'yyyy', yay!'");
System.out.println(dt.format(g3)); // NEW! 2020, yay!
```

Without escaping the text values with single quotes, an exception will be thrown at runtime if the text cannot be interpreted as a date/time symbol.

```
DateTimeFormatter.ofPattern("The time is hh:mm");  // Exception thrown
```

This line throws an exception since `T` is an unknown symbol. The exam might also present you with an incomplete escape sequence.

```
DateTimeFormatter.ofPattern("'Time is: hh:mm: ");  // Exception thrown
```

Failure to terminate an escape sequence will trigger an exception at runtime.

Supporting Internationalization and Localization

Many applications need to work in different countries and with different languages. For example, consider the sentence “The zoo is holding a special event on 4/1/15 to look at animal behaviors.” When is the event? In

the United States, it is on April 1. However, a British reader would interpret this as January 4. A British reader might also wonder why we didn't write “behaviours.” If we are making a website or program that will be used in multiple countries, we want to use the correct language and formatting.

Internationalization is the process of designing your program so it can be adapted. This involves placing strings in a properties file and ensuring the proper data formatters are used. *Localization* means actually supporting multiple locales or geographic regions. You can think of a locale as being like a language and country pairing. Localization includes translating strings to different languages. It also includes outputting dates and numbers in the correct format for that locale.

Initially, your program does not need to support multiple locales. The key is to future-proof your application by using these techniques. This way, when your product becomes successful, you can add support for new languages or regions without rewriting everything.

In this section, we will look at how to define a locale and use it to format dates, numbers, and strings.

Picking a Locale

While Oracle defines a locale as “a specific geographical, political, or cultural region,” you'll only see languages and countries on the exam. Oracle certainly isn't going to delve into political regions that are not countries. That's too controversial for an exam!

The `Locale` class is in the `java.util` package. The first useful `Locale` to find is the user's current locale. Try running the following code on your

computer:

```
Locale locale = Locale.getDefault();
System.out.println(locale);
```

When we run it, it prints `en_US`. It might be different for you. This default output tells us that our computers are using English and are sitting in the United States.

Notice the format. First comes the lowercase language code. The language is always required. Then comes an underscore followed by the uppercase country code. The country is optional. Figure 16.5 shows the two formats for `Locale` objects that you are expected to remember.

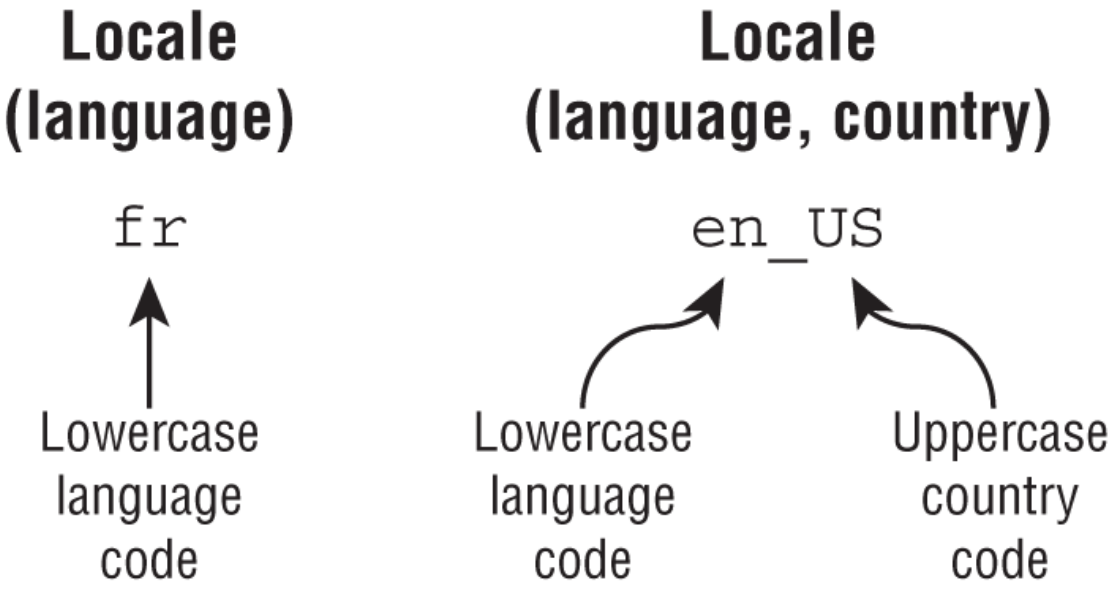

FIGURE 16.5 `Locale` formats

As practice, make sure that you understand why each of these `Locale` identifiers is invalid:

```
US    // Cannot have country without language
enUS  // Missing underscore
US_en // The country and language are reversed
EN    // Language must be lowercase
```

The corrected versions are `en` and `en_US`.

You do not need to memorize language or country codes. The exam will let you know about any that are being used. You do need to recognize valid and invalid formats. Pay attention to uppercase/lowercase and the underscore. For example, if you see a locale expressed as `es_CO`, then you should know that the language is `es` and the country is `CO`, even if you didn't know they represent Spanish and Colombia, respectively.

As a developer, you often need to write code that selects a locale other than the default one. There are three common ways of doing this. The first is to use the built-in constants in the `Locale` class, available for some common locales.

```
System.out.println(Locale.GERMAN);  // de
System.out.println(Locale.GERMANY); // de_DE
```

The first example selects the German language, which is spoken in many countries, including Austria (`de_AT`) and Liechtenstein (`de_LI`). The second example selects both German the language and Germany the country. While these examples may look similar, they are not the same. Only one includes a country code.

The second way of selecting a `Locale` is to use the constructors to create a new object. You can pass just a language, or both a language and country:

```
System.out.println(new Locale("fr"));       // fr
System.out.println(new Locale("hi", "IN")); // hi_IN
```

The first is the language French, and the second is Hindi in India. Again, you don't need to memorize the codes. There is another constructor that lets you be even more specific about the locale. Luckily, providing a variant value is not on the exam.

Java will let you create a `Locale` with an invalid language or country, such as `xx_XX`. However, it will not match the `Locale` that you want to use, and your program will not behave as expected.

There's a third way to create a `Locale` that is more flexible. The builder design pattern lets you set all of the properties that you care about and then build it at the end. This means that you can specify the properties in any order. The following two `Locale` values both represent `en_US`:

```
Locale l1 = new Locale.Builder()
   .setLanguage("en")
   .setRegion("US")
   .build();

Locale l2 = new Locale.Builder()
   .setRegion("US")
   .setLanguage("en")
   .build();
```

Real World Scenario

THE BUILDER PATTERN

Another design pattern commonly used in Java APIs is the builder pattern. The *builder pattern* is a creational pattern in which the task of setting the properties to create an object and the actual creation of the object are distinct steps.

In Java, it is often implemented with an instance of a `static` nested class. Since the builder and the target class tend to be tightly coupled, it makes sense for them to be defined within the same class.

Once all of the properties to create the object are specified, a `build()` method is then called that returns an instance of the desired object. It is commonly used to construct immutable objects with a lot of parameters since an immutable object is created only at the end of a method chain.

When testing a program, you might need to use a `Locale` other than the default of your computer.

```
System.out.println(Locale.getDefault()); // en_US
Locale locale = new Locale("fr");
Locale.setDefault(locale);               // change the default
System.out.println(Locale.getDefault()); // fr
```

Try it, and don't worry—the `Locale` changes for only that one Java program. It does not change any settings on your computer. It does not even change future executions of the same program.

The exam may use `setDefault()` because it can't make assumptions about where you are located. In practice, we rarely write code to change a user's default locale.

Localizing Numbers

It might surprise you that formatting or parsing currency and number values can change depending on your locale. For example, in the United States, the dollar sign is prepended before the value along with a decimal point for values less than one dollar, such as `$2.15`. In Germany, though, the euro symbol is appended to the value along with a comma for values less than one euro, such as `2,15 €`.

Luckily, the `java.text` package includes classes to save the day. The following sections cover how to format numbers, currency, and dates based on the locale.

The first step to formatting or parsing data is the same: obtain an instance of a `NumberFormat`. Table 16.7 shows the available factory methods.

TABLE 16.7 Factory methods to get a `NumberFormat`

Description	Using default `Locale` and a specified `Locale`
A general-purpose formatter	`NumberFormat.getInstance()` `NumberFormat.getInstance(locale)`
Same as `getInstance`	`NumberFormat.getNumberInstance()` `NumberFormat.getNumberInstance(locale)`
For formatting monetary amounts	`NumberFormat.getCurrencyInstance()` `NumberFormat.getCurrencyInstance(locale)`
For formatting percentages	`NumberFormat.getPercentInstance()` `NumberFormat.getPercentInstance(locale)`
Rounds decimal values before displaying	`NumberFormat.getIntegerInstance()` `NumberFormat.getIntegerInstance(locale)`

Once you have the `NumberFormat` instance, you can call `format()` to turn a number into a `String`, or you can use `parse()` to turn a `String` into a number.

The format classes are not thread-safe. Do not store them in instance variables or `static` variables. You'll learn more about thread safety in Chapter 18, "Concurrency."

Formatting Numbers

When we format data, we convert it from a structured object or primitive value into a `String`. The `NumberFormat.format()` method formats the given number based on the locale associated with the `NumberFormat` object.

Let's go back to our zoo for a minute. For marketing literature, we want to share the average monthly number of visitors to the San Diego Zoo. The following shows printing out the same number in three different locales:

```
int attendeesPerYear = 3_200_000;
int attendeesPerMonth = attendeesPerYear / 12;

var us = NumberFormat.getInstance(Locale.US);
System.out.println(us.format(attendeesPerMonth));

var gr = NumberFormat.getInstance(Locale.GERMANY);
System.out.println(gr.format(attendeesPerMonth));

var ca = NumberFormat.getInstance(Locale.CANADA_FRENCH);
System.out.println(ca.format(attendeesPerMonth));
```

The output looks like this:

```
266,666
266.666
266 666
```

This shows how our U.S., German, and French Canadian guests can all see the same information in the number format they are accustomed to using. In practice, we would just call `NumberFormat.getInstance()` and rely on the user's default locale to format the output.

Formatting currency works the same way.

```
double price = 48;
var myLocale = NumberFormat.getCurrencyInstance();
```

```
System.out.println(myLocale.format(price));
```

When run with the default locale of `en_US` for the United States, it outputs `$48.00`. On the other hand, when run with the default locale of `en_GB` for Great Britain, it outputs `£48.00`.

In the real world, use `int` or `BigDecimal` for money and not `double`. Doing math on amounts with `double` is dangerous because the values are stored as floating-point numbers. Your boss won't appreciate it if you lose pennies or fractions of pennies during transactions!

Parsing Numbers

When we parse data, we convert it from a `String` to a structured object or primitive value. The `NumberFormat.parse()` method accomplishes this and takes the locale into consideration.

For example, if the locale is the English/United States (`en_US`) and the number contains commas, the commas are treated as formatting symbols. If the locale relates to a country or language that uses commas as a decimal separator, the comma is treated as a decimal point.

The `parse()` method, found in various types, declares a checked exception `ParseException` that must be handled or declared in the method in which they are called.

Let's look at an example. The following code parses a discounted ticket price with different locales. The `parse()` method actually throws a checked `ParseException`, so make sure to handle or declare it in your own code.

```
String s = "40.45";

var en = NumberFormat.getInstance(Locale.US);
System.out.println(en.parse(s));  // 40.45

var fr = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(fr.parse(s));  // 40
```

In the United States, a dot (`.`) is part of a number, and the number is parsed how you might expect. France does not use a decimal point to separate numbers. Java parses it as a formatting character, and it stops looking at the rest of the number. The lesson is to make sure that you parse using the right locale!

The `parse()` method is also used for parsing currency. For example, we can read in the zoo's monthly income from ticket sales.

```
String income = "$92,807.99";
var cf = NumberFormat.getCurrencyInstance();
double value = (Double) cf.parse(income);
System.out.println(value); // 92807.99
```

The currency string `"$92,807.99"` contains a dollar sign and a comma. The `parse` method strips out the characters and converts the value to a number. The return value of `parse` is a `Number` object. `Number` is the parent class of all the `java.lang` wrapper classes, so the return value can be cast to its appropriate data type. The `Number` is cast to a `Double` and then automatically unboxed into a `double`.

Writing a Custom Number Formatter

Like you saw earlier when working with dates, you can also create your own number format strings using the `DecimalFormat` class, which extends `NumberFormat`. When creating a `DecimalFormat` object, you use a constructor rather than a factory method. You pass the pattern that you would like to use. The patterns can get complex, but you need to know only about two formatting characters, shown in Table 16.8.

TABLE 16.8 `DecimalFormat` symbols

Symbol	Meaning	Examples
`#`	Omit the position if no digit exists for it.	`$2.2`
`0`	Put a `0` in the position if no digit exists for it.	`$002.20`

These examples should help illuminate how these symbols work:

```
12: double d = 1234567.467;
13: NumberFormat f1 = new DecimalFormat("###,###,###.0");
14: System.out.println(f1.format(d));  // 1,234,567.5
15:
16: NumberFormat f2 = new DecimalFormat("000,000,000.00000");
17: System.out.println(f2.format(d));  // 001,234,567.46700
18:
19: NumberFormat f3 = new DecimalFormat("$#,###,###.##");
20: System.out.println(f3.format(d));  // $1,234,567.47
```

Line 14 displays the digits in the number, rounding to the nearest 10th after the decimal. The extra positions to the left are left off because we used `#`. Line 17 adds leading and trailing zeros to make the output the desired length. Line 20 shows prefixing a nonformatting character (`$` sign) along with rounding because fewer digits are printed than available.

Localizing Dates

Like numbers, date formats can vary by locale. Table 16.9 shows methods used to retrieve an instance of a `DateTimeFormatter` using the default locale.

TABLE 16.9 Factory methods to get a `DateTimeFormatter`

Description	Using default `Locale`
For formatting dates	`DateTimeFormatter.ofLocalizedDate(dateStyle)`
For formatting times	`DateTimeFormatter.ofLocalizedTime(timeStyle)`
For formatting dates and times	`DateTimeFormatter.ofLocalizedDateTime(dateStyle, timeStyle)` `DateTimeFormatter.ofLocalizedDateTime(dateTimeStyle)`

Each method in the table takes a `FormatStyle` parameter, with possible values `SHORT`, `MEDIUM`, `LONG`, and `FULL`. For the exam, you are not required to know the format of each of these styles.

What if you need a formatter for a specific locale? Easy enough—just append `withLocale(locale)` to the method call.

Let's put it all together. Take a look at the following code snippet, which relies on a `static` import for the `java.time.format.FormatStyle.SHORT` value:

```
public static void print(DateTimeFormatter dtf,
      LocalDateTime dateTime, Locale locale) {
   System.out.println(dtf.format(dateTime) + ", "
```

```
        + dtf.withLocale(locale).format(dateTime));
    }
    public static void main(String[] args) {
       Locale.setDefault(new Locale("en", "US"));
       var italy = new Locale("it", "IT");
       var dt = LocalDateTime.of(2020, Month.OCTOBER, 20, 15, 12, 34);

       // 10/20/20, 20/10/20
       print(DateTimeFormatter.ofLocalizedDate(SHORT),dt,italy);

       // 3:12 PM, 15:12
       print(DateTimeFormatter.ofLocalizedTime(SHORT),dt,italy);

       // 10/20/20, 3:12 PM, 20/10/20, 15:12
       print(DateTimeFormatter.ofLocalizedDateTime(SHORT,SHORT),dt,italy);
    }
```

First, we establish `en_US` as the default locale, with `it_IT` as the requested locale. We then output each value using the two locales. As you can see, applying a locale has a big impact on the built-in date and time formatters.

Specifying a Locale Category

When you call `Locale.setDefault()` with a locale, several display and formatting options are internally selected. If you require finer-grained control of the default locale, Java actually subdivides the underlying formatting options into distinct categories, with the `Locale.Category` enum.

The `Locale.Category` enum is a nested element in `Locale`, which supports distinct locales for displaying and formatting data. For the exam, you should be familiar with the two enum values in Table 16.10.

TABLE 16.10 `Locale.Category` values

Value	Description
`DISPLAY`	Category used for displaying data about the locale
`FORMAT`	Category used for formatting dates, numbers, or currencies

When you call `Locale.setDefault()` with a locale, both the `DISPLAY` and `FORMAT` are set together. Let's take a look at an example:

```
10: public static void printCurrency(Locale locale, double money) {
11:    System.out.println(
12:       NumberFormat.getCurrencyInstance().format(money)
13:       + ", " + locale.getDisplayLanguage());
14: }
15: public static void main(String[] args) {
16:    var spain = new Locale("es", "ES");
17:    var money = 1.23;
18:
19:    // Print with default locale
20:    Locale.setDefault(new Locale("en", "US"));
21:    printCurrency(spain, money);  // $1.23, Spanish
22:
23:    // Print with default locale and selected locale display
24:    Locale.setDefault(Category.DISPLAY, spain);
25:    printCurrency(spain, money);  // $1.23, español
26:
27:    // Print with default locale and selected locale format
28:    Locale.setDefault(Category.FORMAT, spain);
29:    printCurrency(spain, money);  // 1,23 €, español
30: }
```

The code prints the same data three times. First, it prints the language of the `spain` and `money` variables using the locale `en_US`. Then, it prints it using the `DISPLAY` category of `es_ES`, while the `FORMAT` category re-

mains `en_US`. Finally, it prints the data using both categories set to `es_ES`.

For the exam, you do not need to memorize the various display and formatting options for each category. You just need to know that you can set parts of the locale independently. You should also know that calling `Locale.setDefault(us)` after the previous code snippet will change both locale categories to `en_US`.

Loading Properties with Resource Bundles

Up until now, we've kept all of the text strings displayed to our users as part of the program inside the classes that use them. Localization requires externalizing them to elsewhere.

A *resource bundle* contains the locale-specific objects to be used by a program. It is like a map with keys and values. The resource bundle is commonly stored in a properties file. A *properties file* is a text file in a specific format with key/value pairs.

For the exam, you only need to know about resource bundles that are created from properties files. That said, you can also create a resource bundle from a class by extending `ResourceBundle`. One advantage of this approach is that it allows you to specify values using a method or in formats other than `String`, such as other numeric primitives, objects, or lists.

Our zoo program has been successful. We are now getting requests to use it at three more zoos! We already have support for U.S.-based zoos. We now need to add Zoo de La Palmyre in France, the Greater Vancouver Zoo

in English-speaking Canada, and Zoo de Granby in French-speaking Canada.

We immediately realize that we are going to need to internationalize our program. Resource bundles will be quite helpful. They will let us easily translate our application to multiple locales or even support multiple locales at once. It will also be easy to add more locales later if we get zoos in even more countries interested. We thought about which locales we need to support, and we came up with four.

```
Locale us             = new Locale("en", "US");
Locale france         = new Locale("fr", "FR");
Locale englishCanada = new Locale("en", "CA");
Locale frenchCanada   = new Locale("fr", "CA");
```

In the next sections, we will create a resource bundle using properties files. A *properties file* is a text file that contains a list of key/value pairs. It is conceptually similar to a `Map<String,String>`, with each line representing a different key/value. The key and value are separated by an equal sign (`=`) or colon (`:`). To keep things simple, we use an equal sign throughout this chapter. We will also look at how Java determines which resource bundle to use.

Creating a Resource Bundle

We're going to update our application to support the four locales listed previously. Luckily, Java doesn't require us to create four different resource bundles. If we don't have a country-specific resource bundle, Java will use a language-specific one. It's a bit more involved than this, but let's start with a simple example.

For now, we need English and French properties files for our `Zoo` resource bundle. First, create two properties files.

```
Zoo_en.properties
hello=Hello
```

```
open=The zoo is open

Zoo_fr.properties
hello=Bonjour
open=Le zoo est ouvert
```

The filenames match the name of our resource bundle, `Zoo`. They are then followed by an underscore (`_`), target locale, and `.properties` file extension. We can write our very first program that uses a resource bundle to print this information.

```
10: public static void printWelcomeMessage(Locale locale) {
11:    var rb = ResourceBundle.getBundle("Zoo", locale);
12:    System.out.println(rb.getString("hello")
13:       + ", " + rb.getString("open"));
14: }
15: public static void main(String[] args) {
16:    var us = new Locale("en", "US");
17:    var france = new Locale("fr", "FR");
18:    printWelcomeMessage(us);     // Hello, The zoo is open
19:    printWelcomeMessage(france); // Bonjour, Le zoo est ouvert
20: }
```

Lines 16–17 create the two locales that we want to test, but the method on lines 10–14 does the actual work. Line 11 calls a factory method on `ResourceBundle` to get the right resource bundle. Lines 12 and 13 retrieve the right string from the resource bundle and print the results.

Remember we said you'd see the factory pattern again in this chapter? It will be used a lot in this book, so it helps to be familiar with it.

Since a resource bundle contains key/value pairs, you can even loop through them to list all of the pairs. The `ResourceBundle` class provides a `keySet()` method to get a set of all keys.

```
var us = new Locale("en", "US");
ResourceBundle rb = ResourceBundle.getBundle("Zoo", us);
```

```
rb.keySet().stream()
   .map(k -> k + ": " + rb.getString(k))
   .forEach(System.out::println);
```

This example goes through all of the keys. It maps each key to a `String` with both the key and the value before printing everything.

```
hello: Hello
open: The zoo is open
```

Real World Scenario

LOADING RESOURCE BUNDLE FILES AT RUNTIME

For the exam, you don't need to know where the properties files for the resource bundles are stored. If the exam provides a properties file, it is safe to assume it exists and is loaded at runtime.

In your own applications, though, the resource bundles can be stored in a variety of places. While they can be stored inside the JAR that uses them, it not recommended. This approach forces you to rebuild the application JAR any time some text changes. One of the benefits of using resource bundles is to decouple the application code from the locale-specific text data.

Another approach is to have all of the properties files in a separate properties JAR or folder and load them in the classpath at runtime. In this manner, a new language can be added without changing the application JAR.

Picking a Resource Bundle

There are two methods for obtaining a resource bundle that you should be familiar with for the exam.

```
ResourceBundle.getBundle("name");
ResourceBundle.getBundle("name", locale);
```

The first one uses the default locale. You are likely to use this one in programs that you write. Either the exam tells you what to assume as the default locale or it uses the second approach.

Java handles the logic of picking the best available resource bundle for a given key. It tries to find the most specific value. Table 16.11 shows what Java goes through when asked for resource bundle `Zoo` with the locale `new Locale("fr", "FR")` when the default locale is U.S. English.

TABLE 16.11 Picking a resource bundle for French/France with default locale English/US

Step	Looks for file	Reason
1	`Zoo_fr_FR.properties`	The requested locale
2	`Zoo_fr.properties`	The language we requested with no country
3	`Zoo_en_US.properties`	The default locale
4	`Zoo_en.properties`	The default locale's language with no country
5	`Zoo.properties`	No locale at all—the default bundle
6	If still not found, throw `MissingResourceException`.	No locale or default bundle available

As another way of remembering the order of Table 16.11, learn these steps:

1. Look for the resource bundle for the requested locale, followed by the one for the default locale.
2. For each locale, check language/country, followed by just the language.
3. Use the default resource bundle if no matching locale can be found.

As we mentioned earlier, Java supports resource bundles from Java classes and properties alike. When Java is searching for a matching resource bundle, it will first check for a resource bundle file with the matching class name. For the exam, you just need to know how to work with properties files.

Let's see if you understand Table 16.11. What is the maximum number of files that Java would need to consider to find the appropriate resource bundle with the following code?

```
Locale.setDefault(new Locale("hi"));
ResourceBundle rb = ResourceBundle.getBundle("Zoo", new Locale("en"));
```

The answer is three. They are listed here:

1. `Zoo_en.properties`
2. `Zoo_hi.properties`
3. `Zoo.properties`

The requested locale is `en`, so we start with that. Since the `en` locale does not contain a country, we move on to the default locale, `hi`. Again, there's no country, so we end with the default bundle.

Selecting Resource Bundle Values

Got all that? Good—because there is a twist. The steps that we've discussed so far are for finding the matching resource bundle to use as a base. Java isn't required to get all of the keys from the same resource bundle. It can get them from any parent of the matching resource bundle. A parent resource bundle in the hierarchy just removes components of the name until it gets to the top. Table 16.12 shows how to do this.

TABLE 16.12 Selecting resource bundle properties

Matching resource bundle	Properties files keys can come from

`Zoo_fr_FR`	`Zoo_fr_FR.properties`
	`Zoo_fr.properties`
	`Zoo.properties`

Once a resource bundle has been selected, only properties along a single hierarchy will be used. Contrast this behavior with Table 16.11, in which the default `en_US` resource bundle is used if no other resource bundles are available.

What does this mean exactly? Assume the requested locale is `fr_FR` and the default is `en_US` . The JVM will provide data from an `en_US` *only if there is no matching* `fr_FR` *or* `fr` *resource bundles*. If it finds a `fr_FR` or `fr` resource bundle, then only those bundles, along with the default bundle, will be used.

Let's put all of this together and print some information about our zoos. We have a number of properties files this time.

`Zoo.properties`

```
name=Vancouver Zoo
```

`Zoo_en.properties`

```
hello=Hello
open=is open

Zoo_en_US.properties
name=The Zoo

Zoo_en_CA.properties
visitors=Canada visitors
```

Suppose that we have a visitor from Quebec (which has a default locale of French Canada) who has asked the program to provide information in English. What do you think this outputs?

```
11: Locale.setDefault(new Locale("en", "US"));
12: Locale locale = new Locale("en", "CA");
13: ResourceBundle rb = ResourceBundle.getBundle("Zoo", locale);
14: System.out.print(rb.getString("hello"));
15: System.out.print(". ");
16: System.out.print(rb.getString("name"));
17: System.out.print(" ");
18: System.out.print(rb.getString("open"));
19: System.out.print(" ");
20: System.out.print(rb.getString("visitors"));
```

The program prints the following:

```
Hello. Vancouver Zoo is open Canada visitors
```

The default locale is `en_US`, and the requested locale is `en_CA`. First, Java goes through the available resource bundles to find a match. It finds one right away with `Zoo_en_CA.properties`. This means the default locale of `en_US` is irrelevant.

Line 14 doesn't find a match for the key `hello` in `Zoo_en_CA.properties`, so it goes up the hierarchy to `Zoo_en.properties`. Line 16 doesn't find a match for `name` in either of the first two properties files, so it has to go all the way to the top of the hi-

erarchy to `Zoo.properties`. Line 18 has the same experience as line 14, using `Zoo_en.properties`. Finally, line 20 has an easier job of it and finds a matching key in `Zoo_en_CA.properties`.

In this example, only three properties files were used: `Zoo_en_CA.properties`, `Zoo_en.properties`, and `Zoo.properties`. Even when the property wasn't found in `en_CA` or `en` resource bundles, the program preferred using `Zoo.properties` (the default resource bundle) rather than `Zoo_en_US.properties` (the default locale).

What if a property is not found in any resource bundle? Then, an exception is thrown. For example, attempting to call `rb.getString("close")` in the previous program results in a `MissingResourceException` at runtime.

Formatting Messages

Often, we just want to output the text data from a resource bundle, but sometimes you want to format that data with parameters. In real programs, it is common to substitute variables in the middle of a resource bundle `String`. The convention is to use a number inside braces such as `{0}`, `{1}`, etc. The number indicates the order in which the parameters will be passed. Although resource bundles don't support this directly, the `MessageFormat` class does.

For example, suppose that we had this property defined:

```
helloByName=Hello, {0} and {1}
```

In Java, we can read in the value normally. After that, we can run it through the `MessageFormat` class to substitute the parameters. The second parameter to `format()` is a vararg, allowing you to specify any number of input values.

Given a resource bundle `rb`:

```
String format = rb.getString("helloByName");
System.out.print(MessageFormat.format(format, "Tammy", "Henry"));
```

that would then print the following:

```
Hello, Tammy and Henry
```

Using the *Properties* Class

When working with the `ResourceBundle` class, you may also come across the `Properties` class. It functions like the `HashMap` class that you learned about in Chapter 14, "Generics and Collections," except that it uses `String` values for the keys and values. Let's create one and set some values.

```
import java.util.Properties;
public class ZooOptions {
   public static void main(String[] args) {
      var props = new Properties();
      props.setProperty("name", "Our zoo");
      props.setProperty("open", "10am");
   }
}
```

The `Properties` class is commonly used in handling values that may not exist.

```
System.out.println(props.getProperty("camel"));          // null
System.out.println(props.getProperty("camel", "Bob"));  // Bob
```

If a key were passed that actually existed, both statements would have printed it. This is commonly referred to as providing a default, or backup value, for a missing key.

The `Properties` class also includes a `get()` method, but only `getProperty()` allows for a default value. For example, the following call is invalid since `get()` takes only a single parameter:

```
props.get("open");                                  // 10am

props.get("open", "The zoo will be open soon");  // DOES NOT COMPILE
```

USING THE PROPERTY METHODS

A `Properties` object isn't just similar to a `Map`; it actually inherits `Map<Object,Object>`. Despite this, you should use the `getProperty()` and `setProperty()` methods when working with a `Properties` object, rather than the `get()`/`put()` methods. Besides supporting default values, it also ensures you don't add data to the `Properties` object that cannot be read.

```
var props = new Properties();
props.put("tigerAge", "4");
props.put("lionAge", 5);
System.out.println(props.getProperty("tigerAge"));  // 4
System.out.println(props.getProperty("lionAge"));   // null
```

Since a `Properties` object works only with `String` values, trying to read a numeric value returns `null`. Don't worry, you don't have to know this behavior for the exam. The point is to avoid using the `get`/`put()` methods when working with `Properties` objects.

Summary

This chapter covered a wide variety of topics centered around building applications that respond well to change. We started our discussion with exception handling. Exceptions can be divided into two categories: checked and unchecked. In Java, checked exceptions inherit `Exception`

but not `RuntimeException` and must be handled or declared. Unchecked exceptions inherit `RuntimeException` or `Error` and do not need to be handled or declared. It is considered a poor practice to catch an `Error`.

You can create your own checked or unchecked exceptions by extending `Exception` or `RuntimeException`, respectively. You can also define custom constructors and messages for your exceptions, which will show up in stack traces.

Automatic resource management can be enabled by using a try-with-resources statement to ensure the resources are properly closed. Resources are closed at the conclusion of the `try` block, in the reverse order in which they are declared. A suppressed exception occurs when more than one exception is thrown, often as part of a `finally` block or try-with-resources `close()` operation. The first exception to be encountered will be the primary exception, with the additional exceptions being suppressed. New in Java 9 is the ability to use existing resources in a try-with-resources statement.

An assertion is a `boolean` expression placed at a particular point in your code where you think something should always be true. A failed assertion throws an `AssertionError`. Assertions should not change the state of any variables. You saw how to use the `-ea` and `-enableassertions` flags to turn on assertions and how the `-disableassertions` and `-da` flags can selectively disable assertions for particular classes or packages.

You can create a `Locale` class with a required lowercase language code and optional uppercase country code. For example, `en` and `en_US` are locales for English and U.S. English, respectively. For the exam, you do not need to know how to create dates, but you do need to know how to format them, along with numbers, using a locale. You also need to know how to create custom date and number formatters.

A `ResourceBundle` allows specifying key/value pairs in a properties file. Java goes through candidate resource bundles from the most specific to the most general to find a match. If no matches are found for the re-

quested locale, Java switches to the default locale and then finally the default resource bundle. Once a matching resource bundle is found, Java looks only in the hierarchy of that resource bundle to select values.

By applying the principles you learned about in this chapter to your own projects, you can build applications that last longer, with built-in support for whatever unexpected events may arise.

Exam Essentials

Be able to create custom exception classes. A new checked exception class can be created by extending `Exception`, while an unchecked exception class can be created by extending `RuntimeException`. You can create numerous constructors that call matching parent constructors, with similar arguments. This provides greater control over the exception handling and messages reported in stack traces.

Perform automatic resource management with try-with-resources statements. A try-with-resources statement supports classes that inherit the `AutoCloseable` interface. It automatically closes resources in the reverse order in which they are declared. A try-with-resources statement, as well as a `try` statement with a `finally` block, may generate multiple exceptions. The first becomes the primary exception, and the rest are suppressed exceptions.

Apply try-with-resources to existing resources. A try-with-resources statement can use resources declared before the start of the statement, provided they are `final` or effectively final. They are closed following the execution of the try-with-resources body.

Know how to write assert statements and enable assertions. Assertions are implemented with the `assert` keyword, a `boolean` condition, and an optional message. Assertions are disabled by default. Watch for a question that uses assertions but does not enable them, or a question that tests your knowledge of how assertions are enabled or selectively disabled from the command line.

Identify valid locale strings. Know that the language code is lowercase and mandatory, while the country code is uppercase and optional. Be able to select a locale using a built-in constant, constructor, or builder class.

Format dates, numbers, and messages. Be able to format dates, numbers, and messages into various `String` formats. Also, know how to define a custom date or number formatter using symbols, including how to escape literal values. For messages, you should also be familiar with using the `MessageFormat` and `Properties` classes.

Determine which resource bundle Java will use to look up a key. Be able to create resource bundles for a set of locales using properties files. Know the search order that Java uses to select a resource bundle and how the default locale and default resource bundle are considered. Once a resource bundle is found, recognize the hierarchy used to select values.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which of the following classes contain at least one compiler error? (Choose all that apply.)

```
 class Danger extends RuntimeException {
   public Danger(String message) {
      super();
   }
   public Danger(int value) {
      super((String) null);
   }
}
class Catastrophe extends Exception {
   public Catastrophe(Throwable c) throws RuntimeException {
      super(new Exception());
      c.printStackTrace();
   }
```

```
}
class Emergency extends Danger {
   public Emergency() {}
   public Emergency(String message) {
      super(message);
   }
}
```

A. `Danger`
B. `Catastrophe`
C. `Emergency`
D. All of these classes compile correctly.
E. The answer cannot be determined from the information given.

2. Which of the following are common types to localize? (Choose all that apply.)
   A. Dates
   B. Lambda expressions
   C. Class names
   D. Currency
   E. Numbers
   F. Variable names
3. What is the output of the following code?

```
import java.io.*;
public class EntertainmentCenter {
   static class TV implements AutoCloseable {
      public void close() {
         System.out.print("D");
   } }
   static class MediaStreamer implements Closeable {
      public void close() {
         System.out.print("W");
   } }
   public static void main(String[] args) {
      var w = new MediaStreamer();
      try {
         TV d = new TV(); w;
         }
```

```
            {
               System.out.print("T");
            } catch (Exception e) {
               System.out.print("E");
            } finally {
               System.out.print("F");
            }
         }
      }
```

A. `TWF`
B. `TWDF`
C. `TWDEF`
D. `TWF` followed by an exception.
E. `TWDF` followed by an exception.
F. `TWEF` followed by an exception.
G. The code does not compile.

4. Which statement about the following class is correct?

```
1:  class Problem extends Exception {
2:     public Problem() {}
3:  }
4:  class YesProblem extends Problem {}
5:  public class MyDatabase {
6:     public static void connectToDatabase() throw Problem {
7:        throws new YesProblem();
8:     }
9:     public static void main(String[] c) throw Exception {
10:       connectToDatabase();
11:    }
12: }
```

A. The code compiles and prints a stack trace for `YesProblem` at runtime.
B. The code compiles and prints a stack trace for `Problem` at runtime.
C. The code does not compile because `Problem` defines a constructor.

D. The code does not compile because `YesProblem` does not define a constructor.

E. The code does not compile but would if `Problem` and `YesProblem` were switched on lines 6 and 7.

F. None of the above

5. What is the output of the following code?

```
LocalDate date = LocalDate.parse("2020-04-30",
   DateTimeFormatter.ISO_LOCAL_DATE_TIME);
System.out.println(date.getYear() + " "
   + date.getMonth() + " "+ date.getDayOfMonth());
```

A. `2020 APRIL 2`

B. `2020 APRIL 30`

C. `2020 MAY 2`

D. The code does not compile.

E. A runtime exception is thrown.

6. Assume that all of the files mentioned in the answer choices exist and define the same keys. Which one will be used to find the key in line 8?

```
6: Locale.setDefault(new Locale("en", "US"));
7: var b = ResourceBundle.getBundle("Dolphins");
8: System.out.println(b.getString("name"));
```

A. `Dolphins.properties`

B. `Dolphins_US.properties`

C. `Dolphins_en.properties`

D. `Whales.properties`

E. `Whales_en_US.properties`

F. The code does not compile.

7. For what value of `pattern` will the following print `<005.21> <008.49> <1,234.0>`?

```
String pattern = "__________";
var message = DoubleStream.of(5.21, 8.49, 1234)
```

```
            .mapToObj(v -> new DecimalFormat(pattern).format(v))
            .collect(Collectors.joining("> <"));
        System.out.println("<"+message+">");
```

A. `##.#`
B. `0,000.0#`
C. `#,###.0`
D. `#,###,000.0#`
E. The code does not compile regardless of what is placed in the blank.
F. None of the above

8. Which of the following prints `OhNo` with the assertion failure when the number `magic` is positive? (Choose all that apply.)
   A. `assert magic < 0: "OhNo";`
   B. `assert magic < 0, "OhNo";`
   C. `assert magic < 0 ("OhNo");`
   D. `assert(magic < 0): "OhNo";`
   E. `assert(magic < 0, "OhNo");`
9. Which of the following exceptions must be handled or declared in the method in which they are thrown? (Choose all that apply.)

```
class Apple extends RuntimeException{}
class Orange extends Exception{}
class Banana extends Error{}
class Pear extends Apple{}
class Tomato extends Orange{}
class Peach extends Banana{}
```

   A. `Apple`
   B. `Orange`
   C. `Banana`
   D. `Pear`
   E. `Tomato`
   F. `Peach`
10. Which of the following changes when made independently would make this code compile? (Choose all that apply.)

```
1:  import java.io.*;
2:  public class StuckTurkeyCage implements AutoCloseable {
3:     public void close() throws IOException {
4:        throw new FileNotFoundException("Cage not closed");
5:     }
6:     public static void main(String[] args) {
7:        try (StuckTurkeyCage t = new StuckTurkeyCage()) {
8:           System.out.println("put turkeys in");
9:        }
10:    } }
```

A. Remove `throws IOException` from the declaration on line 3.

B. Add `throws Exception` to the declaration on line 6.

C. Change line 9 to `} catch (Exception e) {}`.

D. Change line 9 to `} finally {}`.

E. The code compiles as is.

F. None of the above

11. What is the result of running `java EnterPark bird.java sing` with the following code?

```
public class EnterPark extends Exception {
   public EnterPark(String message) {
      super();
   }
   private static void checkInput(String[] v) {
      if (v.length <= 3)
         assert(false) : "Invalid input";
   }
   public static void main(String… args) {
      checkInput(args);
      System.out.println(args[0] + args[1] + args[2]);
   }
}
```

A. `birdsing`

B. The `assert` statement throws an `AssertionError`.

C. The code throws an `ArrayIndexOutOfBoundsException`.

D. The code compiles and runs successfully, but there is no output.

E. The code does not compile.

12. Which of the following are true statements about exception handling in Java? (Choose all that apply.)

    A. A traditional `try` statement without a `catch` block requires a `finally` block.

    B. A traditional `try` statement without a `finally` block requires a `catch` block.

    C. A traditional `try` statement with only one statement can omit the `{}`.

    D. A try-with-resources statement without a `catch` block requires a `finally` block.

    E. A try-with-resources statement without a `finally` block requires a `catch` block.

    F. A try-with-resources statement with only one statement can omit the `{}`.

13. Which of the following, when inserted independently in the blank, use locale parameters that are properly formatted? (Choose all that apply.)

```
import java.util.Locale;
public class ReadMap implements AutoCloseable {
   private Locale locale;
   private boolean closed = false;
   void check() {
      assert !closed;
   }
   @Override public void close() {
      check();
      System.out.println("Folding map");
      locale = null;
      closed = true;
   }
   public void open() {
      check();
      this.locale = _____________;
   }
   public void use() {
      // Implementation omitted
```

```
        }
    }
```

A. `new Locale("xM");`
B. `new Locale("MQ", "ks");`
C. `new Locale("qw");`
D. `new Locale("wp", "VW");`
E. `Locale.create("zp");`
F. `Locale.create("FF");`
G. The code does not compile regardless of what is placed in the blank.

14. Which of the following is true when creating your own exception class?
    A. One or more constructors must be coded.
    B. Only custom checked exception classes may be created.
    C. Only custom unchecked exception classes may be created.
    D. Custom `Error` classes may be created.
    E. The `toString()` method must be coded.
    F. None of the above
15. Which of the following can be inserted into the blank to allow the code to compile and run without throwing an exception? (Choose all that apply.)

```
var f = DateTimeFormatter.ofPattern("hh o'clock");
System.out.println(f.format(_____________.now()));
```

    A. `ZonedDateTime`
    B. `LocalDate`
    C. `LocalDateTime`
    D. `LocalTime`
    E. The code does not compile regardless of what is placed in the blank.
    F. None of the above
16. Which of the following command lines cause this program to produce an error when executed? (Choose all that apply.)

```
public class On {
   public static void main(String[] args) {
      String s = null;
      int check = 10;
      assert s != null : check++;
   }
}
```

A. `java –da On`
B. `java –ea On`
C. `java -da -ea:On On`
D. `java -ea -da:On On`
E. The code does not compile.

17. Which of the following statements about resource bundles are correct? (Choose all that apply.)
    A. All keys must be in the same resource bundle to be used.
    B. A resource bundle is loaded by calling the `new ResourceBundle()` constructor.
    C. Resource bundle values are always read using the `Properties` class.
    D. Changing the default locale lasts for only a single run of the program.
    E. If a resource bundle for a specific locale is requested, then the resource bundle for the default locale will not be used.
    F. It is possible to use a resource bundle for a locale without specifying a default locale.
18. What is the output of the following code?

```
import java.io.*;
public class FamilyCar {
   static class Door implements AutoCloseable {
      public void close() {
         System.out.print("D");
   } }
   static class Window implements Closeable {
      public void close() {
```

```
            System.out.print("W");
            throw new RuntimeException();
      } }
      public static void main(String[] args) {
         var d = new Door();
         try (d; var w = new Window()) {
            System.out.print("T");
         } catch (Exception e) {
            System.out.print("E");
         } finally {
            System.out.print("F");
   } } }
```

A. `TWF`
B. `TWDF`
C. `TWDEF`
D. `TWF` followed by an exception.
E. `TWDF` followed by an exception.
F. `TWEF` followed by an exception.
G. The code does not compile.

19. Suppose that we have the following three properties files and code. Which bundles are used on lines 8 and 9, respectively?

```
Dolphins.properties
name=The Dolphin
age=0

Dolphins_en.properties
name=Dolly
age=4

Dolphins_fr.properties
name=Dolly

5: var fr = new Locale("fr");
6: Locale.setDefault(new Locale("en", "US"));
7: var b = ResourceBundle.getBundle("Dolphins", fr);
8: b.getString("name");
9: b.getString("age");
```

A. `Dolphins.properties` and `Dolphins.properties`

B. `Dolphins.properties` and `Dolphins_en.properties`

C. `Dolphins_en.properties` and `Dolphins_en.properties`

D. `Dolphins_fr.properties` and `Dolphins.properties`

E. `Dolphins_fr.properties` and `Dolphins_en.properties`

F. The code does not compile.

G. None of the above

20. Fill in the blanks: When formatting text data, the _____ class supports parametrized `String` values, while the _____ class has built-in support for missing values.

A. `TextFormat`, `Properties`

B. `MessageFormat`, `Properties`

C. `Properties`, `Formatter`

D. `StringFormat`, `Properties`

E. `Properties`, `TextFormat`

F. `Properties`, `TextHandler`

G. None of the above

21. Which changes, when made independently, allow the following program to compile? (Choose all that apply.)

```
1: public class AhChoo {
2:    static class SneezeException extends Exception {}
3:    static class SniffleException extends SneezeException {}
4:    public static void main(String[] args) {
5:       try {
6:          throw new SneezeException();
7:       } catch (SneezeException | SniffleException e) {
8:       } finally {}
9:    } }
```

A. Add `throws SneezeException` to the declaration on line 4.

B. Add `throws Throwable` to the declaration on line 4.

C. Change line 7 to `} catch (SneezeException e) {`.

D. Change line 7 to `} catch (SniffleException e) {`.

E. Remove line 7.

F. The code compiles correctly as is.

G. None of the above

22. What is the output of the following code?

```
LocalDateTime ldt = LocalDateTime.of(2020, 5, 10, 11, 22, 33);
var f = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(ldt.format(f));
```

A. `3/7/19 11:22 AM`

B. `5/10/20 11:22 AM`

C. `3/7/19`

D. `5/10/20`

E. `11:22 AM`

F. The code does not compile.

G. A runtime exception is thrown.

23. Fill in the blank: A class that implements _____ may be in a try-with-resources statement. (Choose all that apply.)

A. `AutoCloseable`

B. `Resource`

C. `Exception`

D. `AutomaticResource`

E. `Closeable`

F. `RuntimeException`

G. `Serializable`

24. What is the output of the following method if `props` contains `{veggies=brontosaurus, meat=velociraptor}`?

```
private static void print(Properties props) {
   System.out.println(props.get("veggies", "none")
      + " " + props.get("omni", "none"));
}
```

A. `brontosaurus none`

B. `brontosaurus null`

C. `none none`

D. `none null`

E. The code does not compile.

F. A runtime exception is thrown.

25. What is the output of the following program?

```
public class SnowStorm {
   static class WalkToSchool implements AutoCloseable {
      public void close() {
         throw new RuntimeException("flurry");
      }
   }
   public static void main(String[] args) {
      WalkToSchool walk1 = new WalkToSchool();
      try (walk1; WalkToSchool walk2 = new WalkToSchool()) {
         throw new RuntimeException("blizzard");
      } catch(Exception e) {
         System.out.println(e.getMessage()
            + " " + e.getSuppressed().length);
      }
      walk1 = null;
   }
}
```

A. `blizzard 0`

B. `blizzard 1`

C. `blizzard 2`

D. `flurry 0`

E. `flurry 1`

F. `flurry 2`

G. None of the above

26. Which of the following are true of the code? (Choose all that apply.)

```
4: private int addPlusOne(int a, int b) {
5:    boolean assert = false;
6:    assert a++> 0;
7:    assert b> 0;
8:    return a + b;
9: }
```

A. Line 5 does not compile.
B. Lines 6 and 7 do not compile because they are missing the `String` message.
C. Lines 6 and 7 do not compile because they are missing parentheses.
D. Line 6 is an appropriate use of an assertion.
E. Line 7 is an appropriate use of an assertion.

27. What is the output of the following program?

```
import java.text.NumberFormat;
import java.util.Locale;
import java.util.Locale.Category;
public class Wallet {
   private double money;
   // Assume getters/setters/constructors provided

   private String openWallet() {
      Locale.setDefault(Category.DISPLAY,
         new Locale.Builder().setRegion("us"));
      Locale.setDefault(Category.FORMAT,
         new Locale.Builder().setLanguage("en"));
      return NumberFormat.getCurrencyInstance(Locale.GERMANY)
         .format(money);
   }
   public void printBalance() {
      System.out.println(openWallet());
   }
   public static void main(String… unused) {
      new Wallet(2.4).printBalance();
   }
}
```

A. `2,40 €`
B. `$2.40`
C. `2.4`
D. The output cannot be determined without knowing the locale of the system where it will be run.
E. The code does not compile.

F. None of the above