Chapter 11
Exceptions and Localization

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Handling Exceptions**
  - Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions.
- ✓ **Implementing Localization**
  - Implement localization using locales and resource bundles. Parse and format messages, dates, times, and numbers, including currency and percentage values.

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

In this chapter, we discuss these problems and solutions to them using exceptions, formatting, 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 specific 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.

Understanding Exceptions

A program can fail for just about any reason. Here are just a few possibilities:

- The code tries to connect to a website, but the Internet connection is down.
- You made a coding mistake and tried to access an invalid index in an array.
- One method calls another with a value that the method doesn't support.

As you can see, some of these are coding mistakes. Others are completely beyond your control. Your program can't help it if the Internet connection goes down. What it *can* do is deal with the situation.

The Role of 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.

As an example, think of Java as a child who visits the zoo. The *happy path* is when nothing goes wrong. The child continues to look at the animals until the program ends nicely. Nothing went wrong, and there were no exceptions to deal with.

This child's younger sister doesn't experience the happy path. In all the excitement, she trips and falls. Luckily, it isn't a bad fall. The little girl gets up and proceeds to look at more animals. She has handled the issue all by herself. Unfortunately, she falls again later in the day and starts crying. This time, she has declared that she needs help by crying. The story ends well. Her daddy rubs her knee and gives her a hug. Then they go back to seeing more animals and enjoy the rest of the day.

These are the two approaches Java uses when dealing with exceptions. A method can handle the exception case itself or make it the caller's responsibility.

Real World Scenario

Return Codes vs. Exceptions

Exceptions are used when "something goes wrong." However, the word *wrong* is subjective. The following code returns `-1` instead of throwing an exception if no match is found:

```
public int indexOf(String[] names, String name) {
   for (int i = 0; i < names.length; i++) {
      if (names[i].equals(name)) { return i; }
   }
   return -1;
}
```

While common for certain tasks like searching, return codes should generally be avoided. After all, Java provided an exception framework, so you should use it!

Understanding Exception Types

An exception is an event that alters program flow. Java has a `Throwable` class for all objects that represent these events. Not all of them have the word *exception* in their class name, which can be confusing. Figure 11.1 shows the key subclasses of `Throwable`.

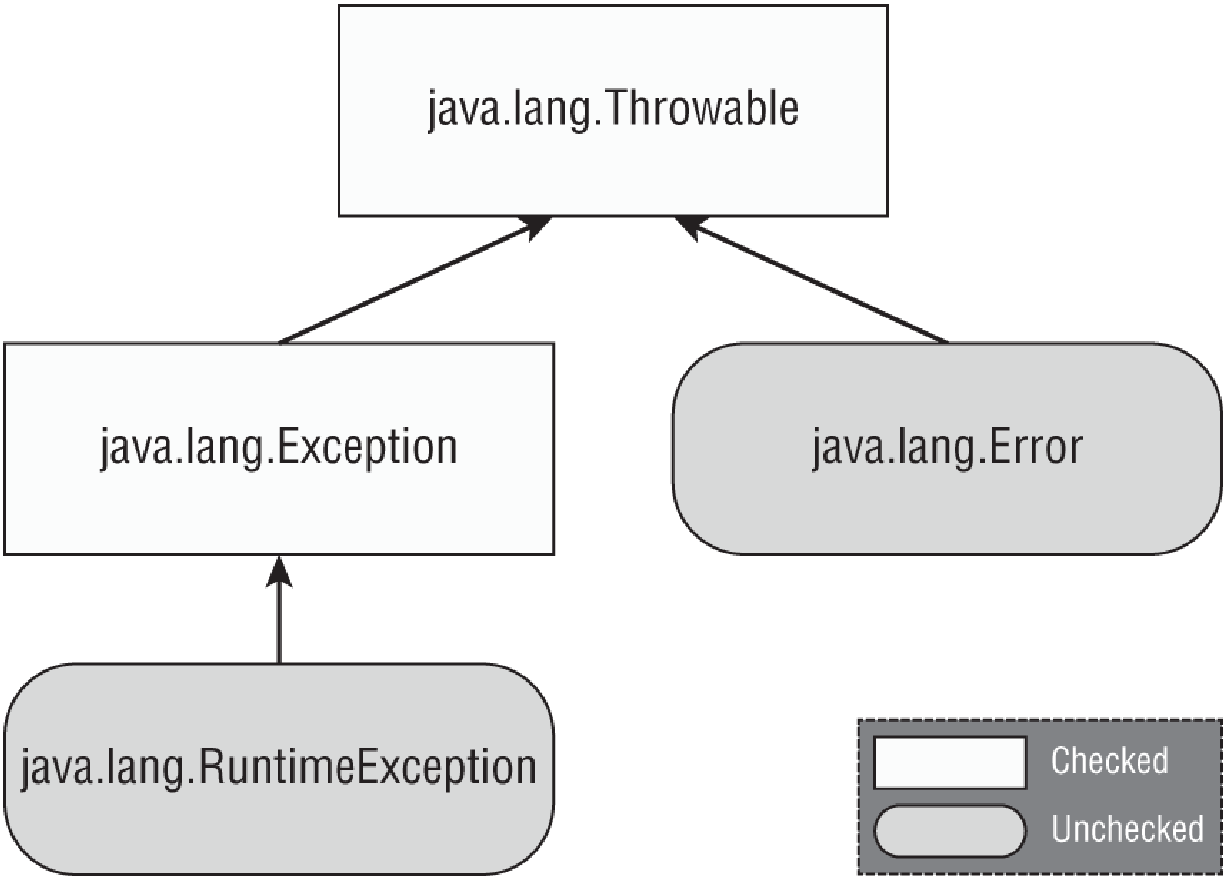

FIGURE 11.1 Categories of exception

Make sure you memorize Figure 11.1 for the exam! We'll cover the details in this chapter, but it is very likely to come up on the exam.

Checked Exceptions

A *checked exception* is an exception that must be declared or handled by the application code where it is thrown. In Java, checked exceptions all inherit `Exception` but not `RuntimeException`. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn't exist.

> **NOTE**
>
> Checked exceptions also include any class that inherits `Throwable` but not `Error` or `RuntimeException`, such as a class that directly extends `Throwable`. For the exam, you just need to know about checked exceptions that extend `Exception`.

Checked exceptions? What are we checking? Java has a rule called the handle or declare rule. The *handle or declare rule* means that all checked exceptions that could be thrown within a method are either wrapped in compatible `try` and `catch` blocks or declared in the method signature.

Because checked exceptions tend to be anticipated, Java enforces the rule that the programmer must do something to show that the exception was thought about. Maybe it was handled in the method. Or maybe the method declares that it can't handle the exception and someone else should.

Let's take a look at an example. The following `fall()` method declares that it might throw an `IOException`, which is a checked exception:

```
 void fall(int distance) throws IOException {
   if(distance> 10) {
      throw new IOException();
   }
}
```

Notice that you're using two different keywords here. The `throw` keyword tells Java that you want to throw an `Exception`, while the `throws` keyword simply declares that the method might throw an `Exception`. It also might not.

Now that you know how to declare an exception, how do you handle it? The following alternate version of the `fall()` method handles the exception:

```
 void fall(int distance) {
```

```
   try {
      if(distance> 10) {
         throw new IOException();
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Notice that the `catch` statement uses `Exception`, not `IOException`. Since `IOException` is a subclass of `Exception`, the `catch` block is allowed to catch it. We cover `try` and `catch` blocks in more detail later in this chapter.

Unchecked Exceptions

An *unchecked exception* is any exception that does not need to be declared or handled by the application code where it is thrown. Unchecked exceptions are often referred to as *runtime exceptions*, although in Java, unchecked exceptions include any class that inherits `RuntimeException` or `Error`.

> **TIP**
>
> It is permissible to handle or declare an unchecked exception. That said, it is better to document the unchecked exceptions callers should know about in a Javadoc comment rather than declaring an unchecked exception.

A *runtime exception* is defined as the `RuntimeException` class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Even though they do inherit the `Exception` class, they are not checked exceptions.

An unchecked exception can occur on nearly any line of code, as it is not required to be handled or declared. For example, a `NullPointerException` can be thrown in the body of the following method if the `input` reference is `null`:

```
 void fall(String input) {
   System.out.println(input.toLowerCase());
}
```

We work with objects in Java so frequently that a `NullPointerException` can happen almost anywhere. If you had to declare unchecked exceptions everywhere, every single method would have that clutter! Remember, the code will still compile if you declare a redundant unchecked exception.

Error and *Throwable*

`Error` means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive “disappeared” or the program ran out of memory. These are abnormal conditions that you aren’t likely to encounter and cannot recover from.

For the exam, the only thing you need to know about `Throwable` is that it’s the parent class of all exceptions, including the `Error` class. While you *can* handle `Throwable` and `Error` exceptions, it is not recommended you do so in your application code. When we refer to exceptions in this chapter, we generally mean any class that inherits `Throwable`, although we are almost always working with the `Exception` class or subclasses of it.

Reviewing Exception Types

Be sure to closely study everything in Table 11.1. For the exam, remember that a `Throwable` is either an `Exception` or an `Error`. You should not catch `Throwable` directly in your code.

TABLE 11.1 Types of exceptions and errors

Type	How to recognize	OK for program to catch?	Is program required to handle or declare?
Unchecked exception	Subclass of `RuntimeException`	Yes	No
Checked exception	Subclass of `Exception` but not subclass of `RuntimeException`	Yes	Yes
Error	Subclass of `Error`	No	No

Throwing an Exception

Any Java code can throw an exception; this includes code you write. Some exceptions are provided with Java. You might encounter an exception that was made up for the exam. This is fine. The question will make it obvious that this is an exception by having the class name end with `Exception`. For example, `MyMadeUpException` is clearly an exception.

> TIP
>
> It's common practice in Java to have exception classes end with the word `Exception`, but it is not required. You should follow this convention when creating your own exception classes, though!

On the exam, you will see two types of code that result in an exception. The first is code that's wrong. Here's an example:

```
String[] animals = new String[0];
System.out.println(animals[0]);  //
ArrayIndexOutOfBoundsException
```

This code throws an `ArrayIndexOutOfBoundsException` since the array has no elements. That means questions about exceptions can be hidden in questions that appear to be about something else.

NOTE

On the exam, some questions have a choice about not compiling and about throwing an exception. Pay special attention to code that calls a method on a `null` reference or that references an invalid array or `List` index. If you spot this, you know the correct answer is that the code throws an exception at runtime.

The second way for code to result in an exception is to explicitly request Java to throw one. Java lets you write statements like these:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

The `throw` keyword tells Java that you want some other part of the code to deal with the exception. This is the same as the young girl crying for her daddy. Someone else needs to figure out what to do about the exception.

throw vs. *throws*

Anytime you see `throw` or `throws` on the exam, make sure the correct one is being used. The `throw` keyword is used as a statement inside a code block to throw a new exception or rethrow an existing exception, while the `throws` keyword is used only at the end of a method declaration to indicate what exceptions it supports.

When creating an exception, you can usually pass a `String` parameter with a message, or you can pass no parameters and use the defaults. We say *usually* because this is a convention. Someone has declared a constructor that takes a `String`. Someone could also create an exception class that does not have a constructor that takes a message.

Additionally, you should know that an `Exception` is an `Object`. This means you can store it in an object reference, and this is legal:

```
var e = new RuntimeException();
throw e;
```

The code instantiates an exception on one line and then throws on the next. The exception can come from anywhere, even passed into a method. As long as it is a valid exception, it can be thrown.

The exam might also try to trick you. Do you see why this code doesn't compile?

```
throw RuntimeException();   // DOES NOT COMPILE
```

If your answer is that there is a missing keyword, you're absolutely right. The exception is never instantiated with the `new` keyword.

Let's take a look at another place the exam might try to trick you. Can you see why the following does not compile?

```
3: try {
4:    throw new RuntimeException();
5:    throw new ArrayIndexOutOfBoundsException();  // DOES NOT
COMPILE
6: } catch (Exception e) {}
```

Since line 4 throws an exception, line 5 can never be reached during runtime. The compiler recognizes this and reports an unreachable code error.

Calling Methods That Throw Exceptions

When you're calling a method that throws an exception, the rules are the same as those for handling an exception within the method. Do you see why the following doesn't compile?

```
class NoMoreCarrotsException extends Exception {}

public class Bunny {
   private void eatCarrot() throws NoMoreCarrotsException {}
```

```
    public void hopAround() {
        eatCarrot();  // DOES NOT COMPILE
    }
}
```

The problem is that `NoMoreCarrotsException` is a checked exception. Checked exceptions must be handled or declared. The code would compile if you changed the `hopAround()` method to either of these:

```
    // Option 1
    public void hopAround() throws NoMoreCarrotsException {
        eatCarrot();
    }

    // Option 2
    public void hopAround() {
        try {
            eatCarrot();
        } catch (NoMoreCarrotsException e) {
            System.out.print("sad rabbit");
        }
    }
```

You might have noticed that `eatCarrot()` didn't throw an exception; it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception.

The compiler is still on the lookout for unreachable code. Declaring an unused exception isn't considered unreachable code. It gives the method the option to change the implementation to throw that exception in the future. Do you see the issue here?

```
public class Bunny {
    private void eatCarrot() {}
    public void bad() {
        try {
            eatCarrot();
        } catch (NoMoreCarrotsException e) {  // DOES NOT COMPILE
            System.out.print("sad rabbit");
        }
    } }
```

Java knows that `eatCarrot()` can't throw a checked exception—which means there's no way for the `catch` block in `bad()` to be reached.

NOTE

When you see a checked exception declared inside a `catch` block on the exam, make sure the code in the associated `try` block is capable of throwing the exception or a subclass of the exception. If not, the code is unreachable and does not compile. Remember that this rule does not extend to unchecked exceptions or exceptions declared in a method signature.

Overriding Methods with Exceptions

When we introduced overriding methods in Chapter 6, “Class Design,” we included a rule related to exceptions. An overridden method may not declare any new or broader checked exceptions than the method it inherits. For example, this code isn’t allowed:

```
class CanNotHopException extends Exception {}

class Hopper {
   public void hop() {}
}

public class Bunny extends Hopper {
   public void hop() throws CanNotHopException {}  // DOES NOT COMPILE
}
```

Java knows `hop()` isn’t allowed to throw any checked exceptions because the `hop()` method in the superclass `Hopper` doesn’t declare any. Imagine what would happen if the subclasses’ versions of the method could add checked exceptions—you could write code that calls `Hopper`’s `hop()` method and not handle any exceptions. Then, if `Bunny` were used in its place, the code wouldn’t know to handle or declare `CanNotHopException`.

An overridden method in a subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them.

```
class Hopper {
   public void hop() throws CanNotHopException {}
}
public class Bunny extends Hopper {
   public void hop() {}  // This is fine
}
```

An overridden method not declaring one of the exceptions thrown by the parent method is similar to the method declaring that it throws an exception it never actually throws. This is perfectly legal. Similarly, a class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of a broader type.

Printing an Exception

There are three ways to print an exception. You can let Java print it out, print just the message, or print where the stack trace comes from. This example shows all three approaches:

```
public static void main(String[] args) {
   try {
      hop();
   } catch (Exception e) {
      System.out.println(e + "\n");
      System.out.println(e.getMessage()+ "\n");
      e.printStackTrace();
   }
}
private static void hop() {
   throw new RuntimeException("cannot hop");
}
```

This code prints the following:

```
java.lang.RuntimeException: cannot hop

cannot hop

java.lang.RuntimeException: cannot hop
   at Handling.hop(Handling.java:15)
   at Handling.main(Handling.java:7)
```

The first line shows what Java prints out by default: the exception type and message. The second line shows just the message. The rest shows a stack trace. The stack trace is usually the most helpful because it shows the hierarchy of method calls that were made to reach the line that threw the exception.

Recognizing Exception Classes

You need to recognize three groups of exception classes for the exam: `RuntimeException`, checked `Exception`, and `Error`. We look at common examples of each type. For the exam, you'll need to recognize which type of an exception it is and whether it's thrown by the Java Virtual Machine (JVM) or by a programmer. For some exceptions, you also need to know which are inherited from one another.

RuntimeException Classes

`RuntimeException` and its subclasses are unchecked exceptions that don't have to be handled or declared. They can be thrown by the programmer or the JVM. Common unchecked exception classes are listed in Table 11.2.

TABLE 11.2 Unchecked exceptions

Unchecked exception	Description
`ArithmeticException`	Thrown when code attempts to divide by zero.
`ArrayIndexOutOfBoundsException`	Thrown when code uses illegal index to access array.
`ClassCastException`	Thrown when attempt is made to cast object to class of which it is not an instance.
`NullPointerException`	Thrown when there is a `null` reference where an object is required.
`IllegalArgumentException`	Thrown by programmer to indicate that method has been passed illegal or inappropriate argument.
`NumberFormatException`	Subclass of `IllegalArgumentException`. Thrown when attempt is made to convert `String` to numeric type but `String` doesn't have appropriate format.

ArithmeticException

Trying to divide an `int` by zero gives an undefined result. When this occurs, the JVM will throw an `ArithmeticException`.

```
int answer = 11 / 0;
```

Running this code results in the following output:

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero
```

Java doesn't spell out the word *divide*. That's OK, though, because we know that `/` is the division operator and that Java is trying to tell you division by zero occurred.

The thread `"main"` is telling you the code was called directly or indirectly from a program with a `main` method. On the exam, this is all the output you will see. Next comes the name of the exception, followed by extra information (if any) that goes with the exception.

ArrayIndexOutOfBoundsException

You know by now that array indexes start with `0` and go up to one less than the length of the array—which means this code will throw an `ArrayIndexOutOfBoundsException`.

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);
```

This is a problem because there's no such thing as a negative array index. Running this code yields the following output:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
   Index -1 out of bounds for length 3
```

ClassCastException

Java tries to protect you from impossible casts. This code doesn't compile because `Integer` is not a subclass of `String`:

```
String type = "moose";
Integer number = (Integer) type;  // DOES NOT COMPILE
```

More complicated code thwarts Java's attempts to protect you. When the cast fails at runtime, Java will throw a `ClassCastException`.

```
String type = "moose";
Object obj = type;
Integer number = (Integer) obj;  // ClassCastException
```

The compiler sees a cast from `Object` to `Integer`. This could be OK. The compiler doesn't realize there's a `String` in that `Object`. When the code runs, it yields the following output:

```
Exception in thread "main" java.lang.ClassCastException:
   java.base/java.lang.String cannot be cast to
   java.lang.base/java.lang.Integer
```

Java tells you both types that were involved in the problem, making it apparent what's wrong.

NullPointerException

Instance variables and methods must be called on a non-`null` reference. If the reference is `null`, the JVM will throw a `NullPointerException`.

```
public class Frog {
   static String name;
   public void hop() {
      System.out.print(name.toLowerCase() + " is hopping");
   }
   public static void main(String[] args) {
      new Frog().hop();
   }
}
```

Remember from Chapter 5, “Methods,” that `static` variables are initialized as `null`. Running this code results in the following output:

```
Exception in thread "main" java.lang.NullPointerException:
   Cannot invoke "String.toLowerCase()" because "Frog.name" is
null
```

Notice anything special about this output? Java includes a feature called *Helpful NullPointerExceptions*, in which the JVM tells you the object reference that triggered the `NullPointerException`.

On instance and `static` variables, the JVM will tell you the name of the variable in the nice, easy-to-read format as we just saw. On local variables (including method parameters), it is not as friendly. Let’s try an example:

```
public class Frog {
   public void hop(String name) {
      System.out.print(name.toLowerCase() + " is hopping");
   }
   public static void main(String[] args) {
      new Frog().hop(null);
   }
}
```

This program prints the following:

```
Exception in thread "main" java.lang.NullPointerException:
   Cannot invoke "String.toLowerCase()" because "<parameter1>"
is null
```

Wait, what's <parameter1>? On method parameters it prints <parameterX>, while on local variables it prints <localX>, where X is the order in which the variable appears in the method. The reason for this is that the name of the variable is lost when the code is compiled.

Not very helpful is it? Fret not, there is a fix! If the class is *compiled* with the `-g:vars` argument, then the code will print the variable name at runtime:

```
javac -g:vars Frog.java
java Frog
```

This is a debug argument, meant to provide additional information in the case that the code is behaving unexpectedly. This code now prints the following at runtime:

```
Exception in thread "main" java.lang.NullPointerException:
   Cannot invoke "String.toLowerCase()" because "name" is null
```

Remember, this argument applies only to local variables and method parameters and has to be used when the code is compiled.

> **NOTE**
>
> If you're using an IDE such as Eclipse or IntelliJ, they often have the `-g:vars` parameter enabled by default.

IllegalArgumentException

`IllegalArgumentException` is a way for your program to protect itself. You want to tell the caller that something is wrong—preferably in an obvious way that the caller can't ignore so the programmer will fix the problem. Seeing the code end with an exception is a great reminder that something is wrong. Consider this example when called as `setNumberEggs(-2)`:

```
public void setNumberEggs(int numberEggs) {
   if (numberEggs < 0)
```

```
      throw new IllegalArgumentException("# eggs must not be
negative");
   this.numberEggs = numberEggs;
}
```

The program throws an exception when it's not happy with the parameter values. The output looks like this:

```
Exception in thread "main" java.lang.IllegalArgumentException:
   # eggs must not be negative
```

Clearly, this is a problem that must be fixed if the programmer wants the program to do anything useful.

NumberFormatException

Java provides methods to convert strings to numbers. When these are passed an invalid value, they throw a `NumberFormatException`. The idea is similar to `IllegalArgumentException`. Since this is a common problem, Java gives it a separate class. In fact, `NumberFormatException` is a subclass of `IllegalArgumentException`. Here's an example of trying to convert something non-numeric into an `int`:

```
Integer.parseInt("abc");
```

The output looks like this:

```
Exception in thread "main" java.lang.NumberFormatException:
   For input string: "abc"
```

For the exam, you need to know that `NumberFormatException` is a subclass of `IllegalArgumentException`. We cover more about why that is important later in the chapter.

Checked *Exception* Classes

Checked exceptions have `Exception` in their hierarchy but not `RuntimeException`. They must be handled or declared. Common checked exceptions are listed in Table 11.3.

TABLE 11.3 Checked exceptions

Checked exception	**Description**
`FileNotFoundException`	Subclass of `IOException`. Thrown programmatically when code tries to reference file that does not exist.
`IOException`	Thrown programmatically when problem reading or writing file.
`NotSerializableException`	Subclass of `IOException`. Thrown programmatically when attempting to serialize or deserialize non-serializable class.
`ParseException`	Indicates problem parsing input.

For the exam, you need to know that these are all checked exceptions that must be handled or declared. You also need to know that `FileNotFoundException` and `NotSerializableException` are subclasses of `IOException`. You'll see `ParseException` later in this chapter and the other three classes in Chapter 14, "I/O."

Error Classes

Errors are unchecked exceptions that extend the `Error` class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see the ones listed in Table 11.4.

TABLE 11.4 Errors

Error	**Description**
`ExceptionInInitializerError`	Thrown when `static` initializer throws exception and doesn't handle it
`StackOverflowError`	Thrown when method calls itself too many times (called *infinite recursion* because method typically calls itself without end)
`NoClassDefFoundError`	Thrown when class that code uses is available at compile time but not runtime

For the exam, you just need to know that these errors are unchecked and the code is often unable to recover from them.

Handling Exceptions

What do you do when you encounter an exception? How do you handle or recover from the exception? In this section, we show the various statements in Java that support handling exceptions.

Using *try* and *catch* Statements

Now that you know what exceptions are, let's explore how to handle them. Java uses a `try` statement to separate the logic that might throw an exception from the logic to handle that exception. Figure 11.2 shows the syntax of a *try statement*.

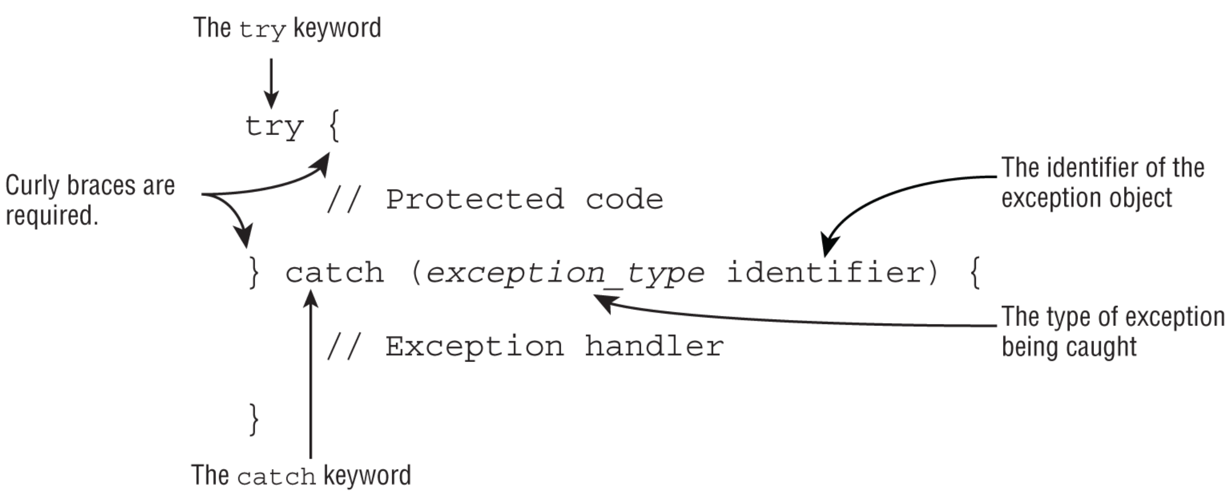

FIGURE 11.2 The syntax of a `try` statement

The code in the `try` block is run normally. If any of the statements throws an exception that can be caught by the exception type listed in the `catch` block, the `try` block stops running, and execution goes to the `catch` statement. If none of the statements in the `try` block throws an exception that can be caught, the *catch clause* is not run.

You probably noticed the words *block* and *clause* used interchangeably. The exam does this as well, so get used to it. Both are correct. *Block* is correct because there are braces present. *Clause* is correct because it is part of a `try` statement.

There aren't a ton of syntax rules here. The braces are required for `try` and `catch` blocks. In our example, the little girl gets up by herself the first time she falls. Here's what this looks like:

```
3:  void explore() {
4:     try {
5:        fall();
6:        System.out.println("never get here");
7:     } catch (RuntimeException e) {
8:        getUp();
9:     }
10:    seeAnimals();
11: }
12: void fall() {  throw new RuntimeException(); }
```

First, line 5 calls the `fall()` method. Line 12 throws an exception. This means Java jumps straight to the `catch` block, skipping line 6. The girl gets up on line 8. Now the `try` statement is over, and execution proceeds normally with line 10.

Now let's look at some invalid `try` statements that the exam might try to trick you with. Do you see what's wrong with this one?

```
try  // DOES NOT COMPILE
   fall();
catch (Exception e)
   System.out.println("get up");
```

The problem is that the braces `{}` are missing. The `try` statements are like methods in that the braces are required even if there is only one statement inside the code blocks, while `if` statements and loops are special and allow you to omit the braces.

What about this one?

```
try {  // DOES NOT COMPILE
   fall();
}
```

This code doesn't compile because the `try` block doesn't have anything after it. Remember, the point of a `try` statement is for something to happen if an exception is thrown. Without another clause, the `try` statement is lonely. As you see shortly, there is a special type of `try` statement that includes an implicit `finally` block, although the syntax is quite different from this example.

Chaining *catch* Blocks

For the exam, you may be given exception classes and need to understand how they function. Here's how to tackle them. First, you must be able to recognize if the exception is a checked or an unchecked exception. Second, you need to determine whether any of the exceptions are subclasses of the others.

```
class AnimalsOutForAWalk extends RuntimeException {}

class ExhibitClosed extends RuntimeException {}

class ExhibitClosedForLunch extends ExhibitClosed {}
```

In this example, there are three custom exceptions. All are unchecked exceptions because they directly or indirectly extend `RuntimeException`. Now we chain both types of exceptions with two `catch` blocks and handle them by printing out the appropriate message:

```
public void visitPorcupine() {
   try {
      seeAnimal();
   } catch (AnimalsOutForAWalk e) {  // first catch block
      System.out.print("try back later");
   } catch (ExhibitClosed e) {       // second catch block
      System.out.print("not today");
   }
}
```

There are three possibilities when this code is run. If `seeAnimal()` doesn't throw an exception, nothing is printed out. If the animal is out for a walk, only the first `catch` block runs. If the exhibit is closed, only the second `catch` block runs. It is not possible for both `catch` blocks to be executed when chained together like this.

A rule exists for the order of the `catch` blocks. Java looks at them in the order they appear. If it is impossible for one of the `catch` blocks to be executed, a compiler error about unreachable code occurs. For example, this happens when a superclass `catch` block appears before a subclass `catch` block. Remember, *we warned you to pay attention to any subclass exceptions!*

In the porcupine example, the order of the `catch` blocks could be reversed because the exceptions don’t inherit from each other. And yes, we have seen a porcupine be taken for a walk on a leash.

The following example shows exception types that do inherit from each other:

```
public void visitMonkeys() {
   try {
      seeAnimal();
   } catch (ExhibitClosedForLunch e) {  // Subclass exception
      System.out.print("try back later");
   } catch (ExhibitClosed e) {          // Superclass exception
      System.out.print("not today");
   }
}
```

If the more specific `ExhibitClosedForLunch` exception is thrown, the first `catch` block runs. If not, Java checks whether the superclass `ExhibitClosed` exception is thrown and catches it. This time, the order of the `catch` blocks does matter. The reverse does not work.

```
public void visitMonkeys() {
   try {
      seeAnimal();
   } catch (ExhibitClosed e) {
      System.out.print("not today");
   } catch (ExhibitClosedForLunch e) {  // DOES NOT COMPILE
      System.out.print("try back later");
   }
}
```

If the more specific `ExhibitClosedForLunch` exception is thrown, the `catch` block for `ExhibitClosed` runs—which means there is no way for the second `catch` block to ever run. Java correctly tells you there is an unreachable `catch` block.

Let’s try this one more time. Do you see why this code doesn’t compile?

```
public void visitSnakes() {
   try {
   } catch (IllegalArgumentException e) {
```

```
    } catch (NumberFormatException e) {  // DOES NOT COMPILE
    }
}
```

Remember we said earlier that you needed to know that `NumberFormatException` is a subclass of `IllegalArgumentException`? This example is the reason why. Since `NumberFormatException` is a subclass, it will always be caught by the first `catch` block, making the second `catch` block unreachable code that does not compile. Likewise, for the exam, you need to know that `FileNotFoundException` is a subclass of `IOException` and cannot be used in a similar manner.

To review multiple `catch` blocks, remember that at most one `catch` block will run, and it will be the first `catch` block that can handle the exception. Also, remember that an exception defined by the `catch` statement is only in scope for that `catch` block. For example, the following causes a compiler error since it tries to use the exception object reference outside the block for which it was defined:

```
public void visitManatees() {
    try {
    } catch (NumberFormatException e1) {
        System.out.println(e1);
    } catch (IllegalArgumentException e2) {
        System.out.println(e1);  // DOES NOT COMPILE
    }
}
```

Applying a Multi-catch Block

Often, we want the result of an exception that is thrown to be the same, regardless of which particular exception is thrown. For example, take a look at this method:

```
public static void main(String args[]) {
    try {
        System.out.println(Integer.parseInt(args[1]));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Missing or invalid input");
    } catch (NumberFormatException e) {
        System.out.println("Missing or invalid input");
    }
}
```

Notice that we have the same `println()` statement for two different `catch` blocks. We can handle this more gracefully using a *multi-catch* block. A multi-catch block allows multiple exception types to be caught by the same `catch` block. Let's rewrite the previous example using a multi-catch block:

```
public static void main(String[] args) {
   try {
      System.out.println(Integer.parseInt(args[1]));
   } catch (ArrayIndexOutOfBoundsException |
NumberFormatException e) {
      System.out.println("Missing or invalid input");
   }
}
```

This is much better. There's no duplicate code, the common logic is all in one place, and the logic is exactly where you would expect to find it. If you wanted, you could still have a second `catch` block for `Exception` in case you want to handle other types of exceptions differently.

Figure 11.3 shows the syntax of multi-catch. It's like a regular `catch` clause, except two or more exception types are specified, separated by a pipe. The pipe (`|`) is also used as the "or" operator, making it easy to remember that you can use either/or of the exception types. Notice how there is only one variable name in the `catch` clause. Java is saying that the variable named `e` can be of type `Exception1` or `Exception2`.

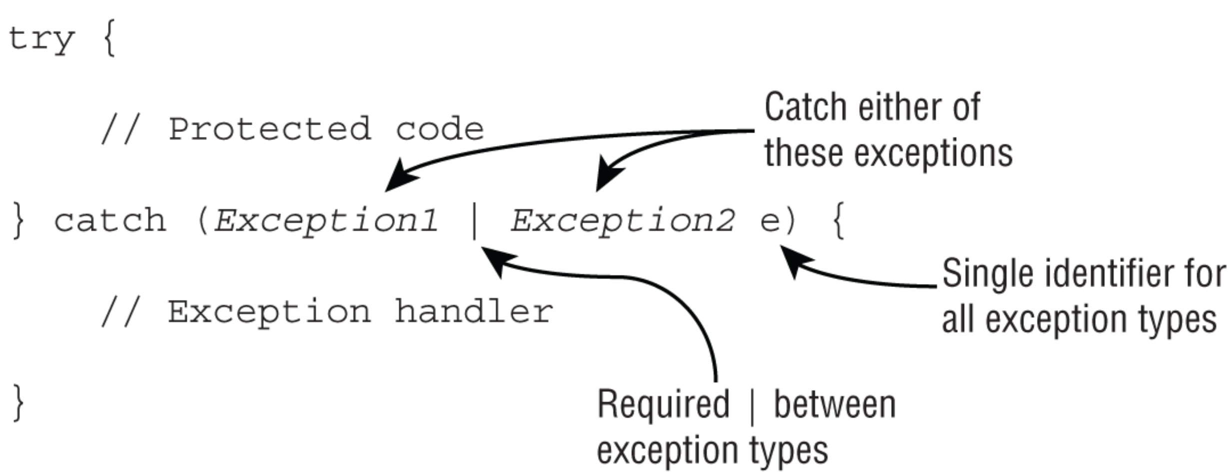

FIGURE 11.3 The syntax of a multi-catch block

The exam might try to trick you with invalid syntax. Remember that the exceptions can be listed in any order within the `catch` clause. However, the variable name must appear only once and at the end. Do you see why these are valid or invalid?

```
catch(Exception1 e | Exception2 e | Exception3 e)    // DOES
NOT COMPILE
```

```
catch(Exception1 e1 | Exception2 e2 | Exception3 e3) // DOES
NOT COMPILE
```

```
catch(Exception1 | Exception2 | Exception3 e)
```

The first line is incorrect because the variable name appears three times. Just because it happens to be the same variable name doesn't make it OK. The second line is incorrect because the variable name again appears three times. Using different variable names doesn't make it any better. The third line does compile. It shows the correct syntax for specifying three exception types.

Java intends multi-catch to be used for exceptions that aren't related, and it prevents you from specifying redundant types in a multi-catch. Do you see what is wrong here?

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

Specifying related exceptions in the multi-catch is redundant, and the compiler gives a message such as this:

```
The exception FileNotFoundException is already caught
   by the alternative IOException
```

Since `FileNotFoundException` is a subclass of `IOException`, this code will not compile. A multi-catch block follows rules similar to chaining `catch` blocks together, which you saw in the previous section. For example, both trigger compiler errors when they encounter unreachable code or duplicate exceptions being caught. The one difference between multi-catch blocks and chaining `catch` blocks is that order does not matter for a multi-catch block within a single `catch` expression.

Getting back to the example, the correct code is just to drop the extraneous subclass reference, as shown here:

```
try {
   throw new IOException();
} catch (IOException e) {}
```

Adding a *finally* Block

The `try` statement also lets you run code at the end with a *finally clause*, regardless of whether an exception is thrown. Figure 11.4 shows the syntax of a `try` statement with this extra functionality.

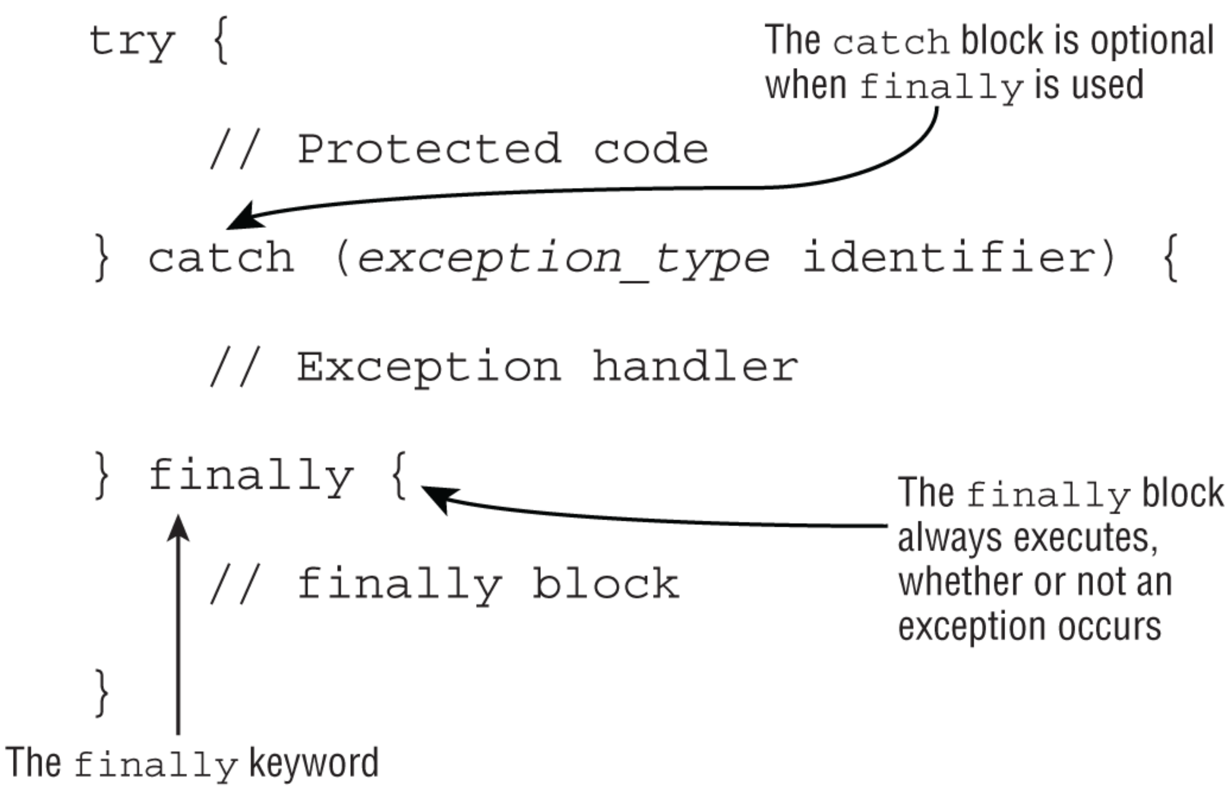

FIGURE 11.4 The syntax of a `try` statement with `finally`

There are two paths through code with both a `catch` and a `finally`. If an exception is thrown, the `finally` block is run after the `catch` block. If no exception is thrown, the `finally` block is run after the `try` block completes.

Let's go back to our young girl example, this time with `finally`:

```
void explore() {
   try {
      seeAnimals();
      fall();
   } catch (Exception e) {
      getHugFromDaddy();
```

```
18:    } finally {
19:       seeMoreAnimals();
20:    }
21:    goHome();
22: }
```

The girl falls on line 15. If she gets up by herself, the code goes on to the `finally` block and runs line 19. Then the `try` statement is over, and the code proceeds on line 21. If the girl doesn't get up by herself, she throws an exception. The `catch` block runs, and she gets a hug on line 17. With that hug, she is ready to see more animals on line 19. Then the `try` statement is over, and the code proceeds on line 21. Either way, the ending is the same. The `finally` block is executed, and execution continues after the `try` statement.

The exam will try to trick you with missing clauses or clauses in the wrong order. Do you see why the following do or do not compile?

```
25: try {  // DOES NOT COMPILE
26:    fall();
27: } finally {
28:    System.out.println("all better");
29: } catch (Exception e) {
30:    System.out.println("get up");
31: }
32:
33: try {  // DOES NOT COMPILE
34:    fall();
35: }
36:
37: try {
38:    fall();
39: } finally {
40:    System.out.println("all better");
41: }
```

The first example (lines 25–31) does not compile because the `catch` and `finally` blocks are in the wrong order. The second example (lines 33–35) does not compile because there must be a `catch` or `finally` block. The third example (lines 37–41) is just fine. The `catch` block is not required if `finally` is present.

Most of the examples you encounter on the exam with `finally` are going to look contrived. For example, you'll get asked questions such as what this code outputs:

```
public static void main(String[] unused) {
   StringBuilder sb = new StringBuilder();
   try {
      sb.append("t");
   } catch (Exception e) {
      sb.append("c");
   } finally {
      sb.append("f");
   }
   sb.append("a");
   System.out.print(sb.toString());
}
```

The answer is `tfa`. The `try` block is executed. Since no exception is thrown, Java goes straight to the `finally` block. Then the code after the `try` statement is run. We know that this is a silly example, but you can expect to see examples like this on the exam.

There is one additional rule you should know for `finally` blocks. If a `try` statement with a `finally` block is entered, then the `finally` block will always be executed, regardless of whether the code completes successfully. Take a look at the following `goHome()` method. Assuming an exception may or may not be thrown on line 14, what are the possible values that this method could print? Also, what would the return value be in each case?

```
12: int goHome() {
13:    try {
14:       // Optionally throw an exception here
15:       System.out.print("1");
16:       return -1;
17:    } catch (Exception e) {
18:       System.out.print("2");
19:       return -2;
20:    } finally {
21:       System.out.print("3");
22:       return -3;
23:    } }
```

If an exception is not thrown on line 14, then line 15 will be executed, printing `1`. Before the method returns, though, the `finally` block is executed, printing `3`. If an exception is thrown, then lines 15 and 16 will be skipped and lines 17–19 will be executed, printing `2`, followed by `3` from the `finally` block. While the first value printed may differ, the method always prints `3` last since it's in the `finally` block.

What is the return value of the `goHome()` method? In this case, it's always `-3`. Because the `finally` block is executed shortly before the method completes, it interrupts the `return` statement from inside both the `try` and `catch` blocks.

For the exam, you need to remember that a `finally` block will always be executed. That said, it may not complete successfully. Take a look at the following code snippet. What would happen if `info` was `null` on line 32?

```
31: } finally {
32:    info.printDetails();
33:    System.out.print("Exiting");
34:    return "zoo";
35: }
```

If `info` was `null`, then the `finally` block would be executed, but it would stop on line 32 and throw a `NullPointerException`. Lines 33 and 34 would not be executed. In this example, you see that while a `finally` block will always be executed, it may not finish.

System.exit()

There is one exception to "the `finally` block will always be executed" rule: Java defines a method that you call as `System.exit()`. It takes an integer parameter that represents the status code that is returned.

```
try {
   System.exit(0);
} finally {
   System.out.print("Never going to get here");  // Not
printed
}
```

`System.exit()` tells Java, "Stop. End the program right now. Do not pass Go. Do not collect $200." When `System.exit()` is called in the `try` or `catch` block, the `finally` block does not run.

Automating Resource Management

Often, your application works with files, databases, and various connection objects. Commonly, these external data sources are referred to as *resources*. In many cases, you *open* a connection to the resource, whether it's over the network or within a file system. You then *read*/*write* the data you want. Finally, you *close* the resource to indicate that you are done with it.

What happens if you don't close a resource when you are done with it? In short, a lot of bad things could happen. If you are connecting to a database, you could use up all available connections, meaning no one can talk to the database until you release your connections. Although you commonly hear about memory leaks causing programs to fail, a *resource leak* is just as bad and occurs when a program fails to release its connections to a resource, resulting in the resource becoming inaccessible. This could mean your program can no longer talk to the database—or, even worse, all programs are unable to reach the database!

For the exam, a *resource* is typically a file or database that requires some kind of stream or connection to read or write data. In Chapter 14, you will create numerous resources that will need to be closed when you are finished with them.

Introducing Try-with-Resources

Let's take a look at a method that opens a file, reads the data, and closes it:

```
public void readFile(String file) {
   FileInputStream is = null;
   try {
      is = new FileInputStream("myfile.txt");
      // Read file data
   } catch (IOException e) {
      e.printStackTrace();
   } finally {
      if(is != null) {
         try {
            is.close();
         } catch (IOException e2) {
            e2.printStackTrace();
         }
      }
```

```
19:     }
20: }
```

Wow, that’s a long method! Why do we have two `try` and `catch` blocks? Well, lines 7 and 14 both include checked `IOException` calls, and those need to be caught in the method or rethrown by the method. Half the lines of code in this method are just closing a resource. And the more resources you have, the longer code like this becomes. For example, you may have multiple resources that need to be closed in a particular order. You also don’t want an exception caused by closing one resource to prevent the closing of another resource.

To solve this, Java includes the *try-with-resources* statement to automatically close all resources opened in a `try` clause. This feature is also known as *automatic resource management*, because Java automatically takes care of the closing.

Let’s take a look at our same example using a try-with-resources statement:

```
public void readFile(String file) {
    try (FileInputStream is = new FileInputStream("myfile.txt")) {
        // Read file data
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Functionally, they are similar, but our new version has half as many lines. More importantly, though, by using a try-with-resources statement, we guarantee that as soon as a connection passes out of scope, Java will attempt to close it within the same method.

Behind the scenes, the compiler replaces a try-with-resources block with a `try` and `finally` block. We refer to this “hidden” `finally` block as an *implicit* `finally` block since it is created and used by the compiler automatically. You can still create a programmer-defined `finally` block when using a try-with-resources statement; just be aware that the implicit one will be called first.

Unlike garbage collection, resources are not automatically closed when they go out of scope. Therefore, 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.

Basics of Try-with-Resources

Figure 11.5 shows what a try-with-resources statement looks like. Notice that one or more resources can be opened in the `try` clause. When multiple resources are opened, they are closed in the *reverse* of the order in which they were created. Also, notice that parentheses are used to list those resources, and semicolons are used to separate the declarations. This works just like declaring multiple indexes in a `for` loop.

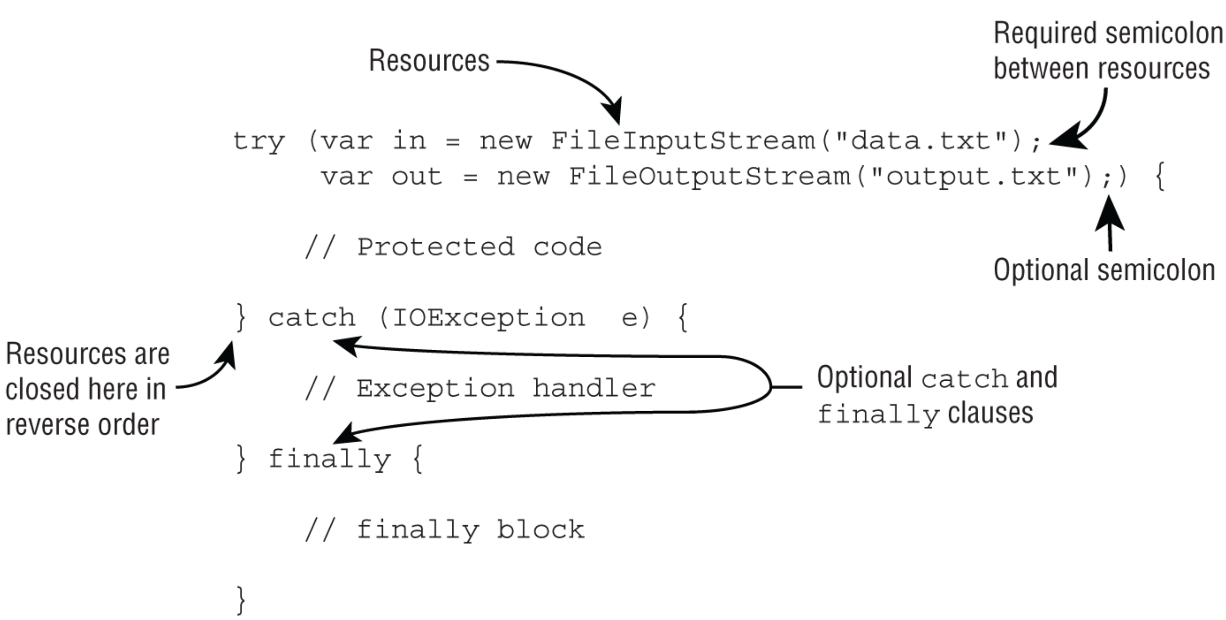

FIGURE 11.5 The syntax of a basic try-with-resources statement

What happened to the `catch` block in Figure 11.5? Well, it turns out a `catch` block is *optional* with a try-with-resources statement. For example, we can rewrite the previous `readFile()` example so that the method declares the exception to make it even shorter:

```
public void readFile(String file) throws IOException {
   try (FileInputStream is = new
FileInputStream("myfile.txt")) {
      // Read file data
   }
}
```

Earlier in the chapter, you learned that a `try` statement must have one or more `catch` blocks or a `finally` block. A try-with-resources statement differs from a `try` statement in that neither of these is required, although a developer may add both. For the exam, you need to know that the implicit `finally` block runs *before* any programmer-coded ones.

Constructing Try-with-Resources Statements

Only classes that implement the `AutoCloseable` interface can be used in a try-with-resources statement. 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.

Throughout the rest of this section, we use the following custom resource class that simply prints a message when the `close()` method is called:

```
public class MyFileClass implements AutoCloseable {
   private final int num;
   public MyFileClass(int num) { this.num = num; }
   @Override public void close() {
      System.out.println("Closing: " + num);
   } }
```

In Chapter 14, you 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`.

Declaring Resources

While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement. For example, the following do not compile:

```
try (MyFileClass is = new MyFileClass(1),  // DOES NOT COMPILE
   os = new MyFileClass(2)) {
}

try (MyFileClass ab = new MyFileClass(1),  // DOES NOT COMPILE
   MyFileClass cd = new MyFileClass(2)) {
}
```

The first example does not compile because it is missing the data type, and it uses a comma (`,`) instead of a semicolon (`;`). The second example does not compile because it also uses a comma (`,`) instead of a semicolon (`;`). Each resource must include the data type and be separated by a semicolon (`;`).

You can declare a resource using `var` as the data type in a try-with-resources statement, since resources are local variables.

```
try (var f = new BufferedInputStream(new
FileInputStream("it.txt"))) {
   // Process file
}
```

Declaring resources is a common situation where using `var` is quite helpful, as it shortens the already long line of code.

Scope of Try-with-Resources

The resources created in the `try` clause are in scope only within the `try` block. This is another way to remember that the implicit `finally` runs before any `catch`/`finally` blocks that you code yourself. The implicit close has run already, and the resource is 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 a nice feature. You can't accidentally use an object that has been closed. In a traditional `try` statement, the variable has to be declared before the `try` statement so that both the `try` and `finally` blocks can access it, which has the unpleasant side effect of making the variable in scope for the rest of the method, just inviting you to call it by accident.

Following Order of Operations

When working with try-with-resources statements, it is important to know that resources are closed in the reverse of the order in which they are created. Using our custom `MyFileClass`, can you figure out what this method prints?

```
public static void main(String… xyz) {
   try (MyFileClass bookReader = new MyFileClass(1);
         MyFileClass movieReader = new MyFileClass(2)) {
      System.out.println("Try Block");
      throw new RuntimeException();
   } catch (Exception e) {
      System.out.println("Catch Block");
   } finally {
      System.out.println("Finally Block");
   } }
```

While this example may look a bit convoluted in practice, questions like this are common on the exam. It output is as follows:

```
Try Block
Closing: 2
Closing: 1
Catch Block
Finally Block
```

For the exam, make sure you understand why the method prints the statements in this order. Remember, the resources are closed in the reverse of the order in which they are declared, and the implicit `finally` is executed before the programmer-defined `finally`.

Applying Effectively Final

While resources are often created in the try-with-resources statement, it is possible to declare them ahead of time, provided they are marked `final` or are effectively final. See Chapter 5 if you'd like to review what effectively final means.

The syntax uses the resource name in place of the resource declaration, separated by a semicolon (`;`). Let's try another example:

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

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
Closing: 5
Closing: 6
Closing: 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 is possible to write to the resource before the try-with-resources statement, it is not afterward.

Understanding Suppressed Exceptions

We conclude our discussion of exceptions with probably the most confusing topic: 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 to. Alternatively, we can allow the caller to deal with it.

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

```
public class JammedTurkeyCage implements AutoCloseable {
   public void close() throws IllegalStateException {
      throw new IllegalStateException("Cage door does not close");
   }
   public static void main(String[] args) {
      try (JammedTurkeyCage t = new JammedTurkeyCage()) {
         System.out.println("Put turkeys in");
      } catch (IllegalStateException e) {
         System.out.println("Caught: " + e.getMessage());
      }
   } }
```

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:       } }
```

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 and 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:       } }
```

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.

> **NOTE**
>
> If more than two resources throw an exception, the first one to be thrown becomes the primary exception, and the rest are grouped as suppressed exceptions. And since resources are closed in the reverse of the order in which they are declared, the primary exception will 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:       } }
```

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. This behavior existed before automatic resource management was added.

Formatting Values

We now shift gears a bit and talk about how to format data for users. In this section, we’re going to be working with numbers, dates, and times. This is especially important in the next section when we expand customization to different languages and locales. You may want to review Chapter 4, “Core APIs,” if you need a refresher on creating various date/time objects.

Formatting Numbers

In Chapter 4, you saw how to control the output of a number using the `String.format()` method. That’s useful for simple stuff, but sometimes you need finer-grained control. With that, we introduce the `NumberFormat` abstract class, which has two commonly used methods:

```
public final String format(double number)
public final String format(long number)
```

Since `NumberFormat` is an abstract class, we need the concrete `DecimalFormat` class to use it. It includes a constructor that takes a pattern `String`:

```
public DecimalFormat(String pattern)
```

The patterns can get quite complex. But luckily, for the exam you only need to know about two formatting characters, shown in Table 11.5.

TABLE 11.5 `DecimalFormat` symbols

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

These examples should help illuminate how these symbols work:

```
12: double d = 1234.567;
13: NumberFormat f1 = new DecimalFormat("###,###,###.0");
14: System.out.println(f1.format(d));  // 1,234.6
15:
16: NumberFormat f2 = new DecimalFormat("000,000,000.00000");
17: System.out.println(f2.format(d));  // 000,001,234.56700
18:
19: NumberFormat f3 = new DecimalFormat("Your Balance
$#,###,###.##");
20: System.out.println(f3.format(d));  // Your Balance
$1,234.57
```

Line 14 displays the digits in the number, rounding to the nearest 10th after the decimal. The extra positions to the left are omitted because we used `#`. Line 17 adds leading and trailing zeros to make the output the desired length. Line 20 shows prefixing a nonformatting character along with rounding because fewer digits are printed than available. Notice that the commas are automatically removed if they are used between `#` symbols.

As you shall see in the localization section of this chapter, there's a second concrete class that inherits `NumberFormat` called `CompactNumberFormat`, which you'll need to know for the exam.

Formatting Dates and Times

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

```
LocalDate date = LocalDate.of(2025, Month.OCTOBER, 20);
System.out.println(date.getDayOfWeek());  // MONDAY
System.out.println(date.getMonth());      // OCTOBER
System.out.println(date.getYear());       // 2025
System.out.println(date.getDayOfYear());  // 293
```

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

```
LocalDate date = LocalDate.of(2025, 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_T
IME));
```

The code snippet prints the following:

```
2025-10-20
11:12:34
2025-10-20T11:12:34
```

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:

```
date.format(DateTimeFormatter.ISO_LOCAL_TIME);  //
RuntimeException
time.format(DateTimeFormatter.ISO_LOCAL_DATE);  //
RuntimeException
```

Customizing the Date/Time Format

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, 2025 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.

> **NOTE**
>
> It's possible, albeit unlikely, to come across questions on the exam that use `SimpleDateFormat` rather than the more useful `DateTimeFormatter`. If you do see it on the exam used with an older `java.util.Date` object, just know that the custom formats that are likely to appear on the exam will be compatible with both.

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 11.6 includes the symbols you should be familiar with for the exam.

TABLE 11.6 Common date/time symbols

Symbol	Meaning	Examples
`y`	Year	`25`, `2025`
`M`	Month	`1`, `01`, `Jan`, `January`
`d`	Day	`5`, `05`
`H`	24 Hour	`15`
`h`	12 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`

TIP

You may find it difficult to remember the differences between upper and lower case letters in Table 11.6. As a tip, you can remember `H` is 24 hours and `h` is 12 hours because uppercase is bigger. Similarly, `M` is month and `m` is minute because `M` is bigger.

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

```
var dt = LocalDateTime.of(2025, Month.OCTOBER, 20, 6, 15, 30);

var formatter1 = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");
System.out.println(dt.format(formatter1));  // 10/20/2025 06:15:30

var formatter2 = DateTimeFormatter.ofPattern("MM_yyyy_-_dd");
System.out.println(dt.format(formatter2));  // 10_2025_-_20

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

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` were used instead, the code would complete successfully and print 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 11.7 shows which symbols you can use with each of the date/time objects.

TABLE 11.7 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(2025, Month.OCTOBER, 20, 6, 15, 30);
var formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy hh:mm:ss");

System.out.println(dateTime.format(formatter));  // 10/20/2025 06:15:30
System.out.println(formatter.format(dateTime));  // 10/20/2025 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 them 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 into multiple smaller formatters and then concatenate the results.

```
var dt = LocalDateTime.of(2025, 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, 2025 at 06:15` at runtime.

While this works, it could become difficult if a lot of text values and date symbols are 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.

```
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy ’at’
hh:mm");
System.out.println(dt.format(f));  // October 20, 2025 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 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! 2025, yay!
```

If you don't escape 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");  //
IllegalArgumentException
```

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: ");  //
IllegalArgumentException
```

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 4/1/25 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 that the proper data formatters are used. *Localization* means 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.

> **NOTE**
>
> 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 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 11.6 shows the two formats for `Locale` objects that you are expected to remember.

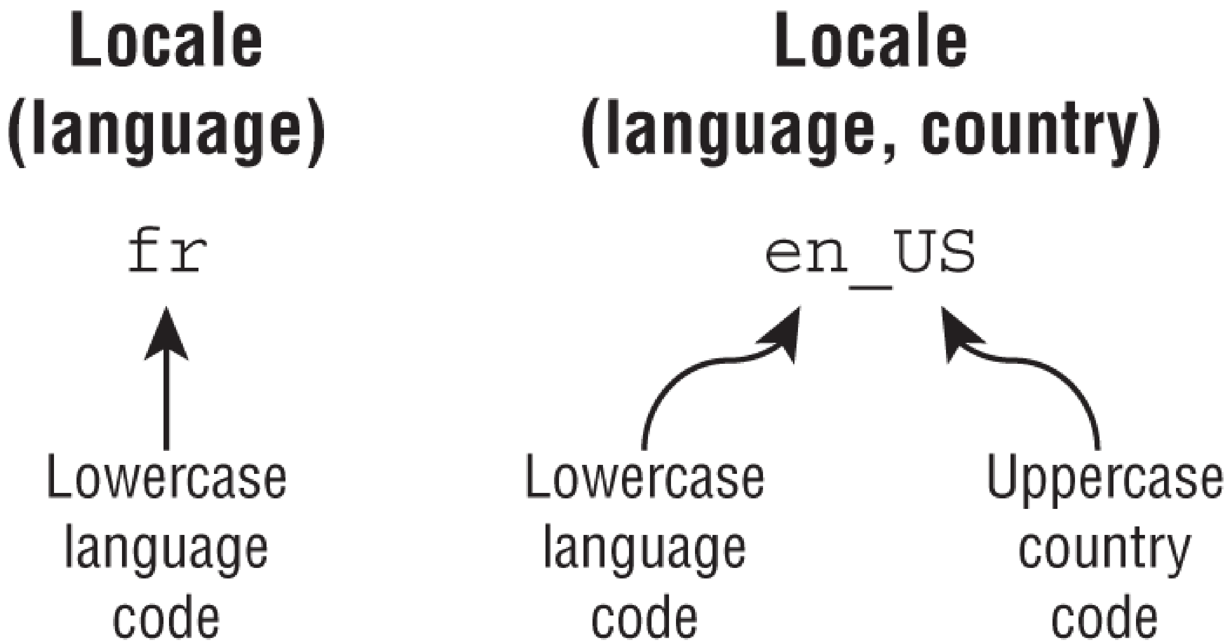

FIGURE 11.6 `Locale` formats

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`.

> **NOTE**
>
> 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 that 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 factory `Locale.of()` methods. You can pass just a language, or both a language and country:

```
System.out.println(Locale.of("fr"));        // fr
System.out.println(Locale.of("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. 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 the `Locale` at the end. This means 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();
```

There's actually a fourth way to create a `Locale` instance, using a `Locale` constructor, such as `new Locale("en")` or `new Locale("en", "US")`. These constructors are now deprecated, so use one of the three previous techniques instead.

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

```
System.out.println(Locale.getDefault());  // en_US
Locale locale = Locale.of("fr");
Locale.setDefault(locale);
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 11.8 shows the available factory methods.

TABLE 11.8 Factory methods to get a `NumberFormat`

Description	**Using default `Locale` and a specified `Locale`**
General-purpose formatter	`NumberFormat.getInstance()` `NumberFormat.getInstance(Locale locale)`
Same as `getInstance`	`NumberFormat.getNumberInstance()` `NumberFormat.getNumberInstance(Locale locale)`
For formatting monetary amounts	`NumberFormat.getCurrencyInstance()` `NumberFormat.getCurrencyInstance(Locale locale)`
For formatting percentages	`NumberFormat.getPercentInstance()` `NumberFormat.getPercentInstance(Locale locale)`
Rounds decimal values before displaying	`NumberFormat.getIntegerInstance()` `NumberFormat.getIntegerInstance(Locale locale)`
Returns compact number formatter	`NumberFormat.getCompactNumberInstance()` `NumberFormat.getCompactNumberInstance( Locale locale, NumberFormat.Style formatStyle)`

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.

TIP

The format classes are not thread-safe. Do not store them in instance variables or `static` variables. You learn more about thread-safety in Chapter 13, “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));  // 266,666

var gr = NumberFormat.getInstance(Locale.GERMANY);
System.out.println(gr.format(attendeesPerMonth));  // 266.666

var ca = NumberFormat.getInstance(Locale.CANADA_FRENCH);
System.out.println(ca.format(attendeesPerMonth));  // 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, this code outputs `$48.00`. On the other hand, when run with the default locale of `en_GB` for Great Britain, it outputs `£48.00`.

> **NOTE**
>
> 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!

Finally, the exam may have examples that show formatting percentages:

```
double successRate = 0.802;
var us = NumberFormat.getPercentInstance(Locale.US);
System.out.println(us.format(successRate));  // 80%

var gr = NumberFormat.getPercentInstance(Locale.GERMANY);
System.out.println(gr.format(successRate));  // 80 %
```

Not much difference, we know, but you should at least be aware that the ability to print a percentage is locale-specific for the exam!

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.

Let's look at an example. The following code parses a discounted ticket price with different locales. The `parse()` method 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 as 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`.

Formatting with *CompactNumberFormat*

The second class that inherits `NumberFormat` that you need to know for the exam is `CompactNumberFormat`. If you haven't seen it before, don't worry, we'll cover it in this section.

`CompactNumberFormat` is similar to `DecimalFormat`, but it is designed to be used in places where print space may be limited. It is opinionated in the sense that it picks a format for you, and locale-specific in that output can change depending on your location.

Consider the following sample code that applies a `CompactNumberFormat` to a group of locales, using a `static` import for `Style` (an enum with value `SHORT` or `LONG`):

```
var formatters = Stream.of(
   NumberFormat.getCompactNumberInstance(),
   NumberFormat.getCompactNumberInstance(Locale.getDefault(),
Style.SHORT),
   NumberFormat.getCompactNumberInstance(Locale.getDefault(),
Style.LONG),

   NumberFormat.getCompactNumberInstance(Locale.GERMAN,
Style.SHORT),
   NumberFormat.getCompactNumberInstance(Locale.GERMAN,
Style.LONG),

   NumberFormat.getNumberInstance());

formatters.map(s ->
s.format(7_123_456)).forEach(System.out::println);
```

The following is printed by this code when run in the `en_US` locale (line breaks added for readability):

```
7M
7M
7 million

7 Mio.
7 Millionen

7,123,456
```

Notice that the first two lines are the same. If you don't specify a style, `SHORT` is used by default. Next, notice that the values except the last one (which doesn't use a compact number formatter) are truncated. There's a reason it's called a compact number formatter! Also, notice that the short form uses common labels for large values, such as `K` for thousand. Last but not least, the output may differ for you when you run this, as it was run in an `en_US` locale.

Using the same formatters, let's try another example:

```
formatters.map(s ->
s.format(314_900_000)).forEach(System.out::println);
```

This prints the following when run in the `en_US` locale:

```
315M
315M
```

```
315 million

315 Mio.
315 Millionen

314,900,000
```

Notice that the third digit is automatically rounded up for the entries that use a `CompactNumberFormat`. The following summarizes the rules for `CompactNumberFormat`:

- First it determines the highest range for the number, such as thousand (`K`), million (`M`), billion (`B`), or trillion (`T`).
- It then returns up to the first three digits of that range, rounding the last digit as needed.
- Finally, it prints an identifier. If `SHORT` is used, a symbol is returned. If `LONG` is used, a space followed by a word is returned.

For the exam, make sure you understand the difference between the `SHORT` and `LONG` formats and common symbols like `M` for million.

> **NOTE**
>
> While certainly out of scope for the exam, some `CompactNumberFormat` instances will display more than three digits if the value is higher than the supported range. For example, using `Long.MAX_VALUE` will display seven digits (`9223372T`) in the previous example, as trillion (10^{12}) is the highest range that the instance will use.

`CompactNumberFormat` can also be used for parsing, although not always in the way you might expect! Consider this example:

```
var locale = Locale.of("en", "US");
var compact = NumberFormat.getCompactNumberInstance(
   locale, Style.SHORT);
System.out.println(compact.format(1_000_000));  // 1M
```

```
24: System.out.println(compact.parse("1M"));         // 1000000
25: System.out.println(compact.parse("1000000"));    // 1000000
26: System.out.println(compact.parse("1,000,000")); // 1
27: System.out.println(compact.parse("$1000000"));  //
ParseException
```

Lines 20-23 should look familiar. They print a million using the short format of `1M`. Lines 24 and 25 shows that the format is flexible in taking the original or shortened format. Line 26 might surprise you as Java stops at the initial punctuation and only prints `1`. By contrast, line 27 is a road too far. Java doesn't know what to do with the `$` and throws a `ParseException`.

Localizing Dates

Like numbers, date formats can vary by locale. Table 11.9 shows methods used to retrieve an instance of a `DateTimeFormatter` using the default locale.

TABLE 11.9 Factory methods to get a `DateTimeFormatter`

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

Each method in the table takes a `FormatStyle` parameter (or two) 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:

```
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(Locale.of("en", "US"));
   var italy = Locale.of("it", "IT");
   var dt = LocalDateTime.of(2025, Month.OCTOBER, 20, 15, 12,
34);

   // 10/20/25 --- 20/10/25
   print(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT),
dt, italy);

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

   // 10/20/25, 3:12 PM --- 20/10/25, 15:12
   print(DateTimeFormatter.ofLocalizedDateTime(
      FormatStyle.SHORT, FormatStyle.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 subdivides the underlying formatting options into distinct categories with the `Locale.Category` enum.

The `Locale.Category` enum is a nested element in `Locale` that supports distinct locales for displaying and formatting data. For the exam, you should be familiar with the two enum values in Table 11.10.

TABLE 11.10 `Locale.Category` values

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

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

```
public static void printCurrency(Locale locale, double money) {
   System.out.println(
      NumberFormat.getCurrencyInstance().format(money)
      + ", " + locale.getDisplayLanguage());
}

public static void main(String[] args) {
   var spain = Locale.of("es", "ES");
   var money = 1.23;

   // Print with default locale
   Locale.setDefault(Locale.of("en", "US"));
   printCurrency(spain, money);  // $1.23, Spanish

   // Print with selected locale display
   Locale.setDefault(Category.DISPLAY, spain);
   printCurrency(spain, money);  // $1.23, español

   // Print with selected locale format
   Locale.setDefault(Category.FORMAT, spain);
   printCurrency(spain, money);  // 1,23 €, español
}
```

The code prints the same data three times. First it prints the money variable and the language value of `spain` using the locale `en_US`. Then it prints it using the `DISPLAY` category of `es_ES`, while the `FORMAT` category remains `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(Locale.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.

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 zoos in even more countries are interested. We thought about which locales we need to support, and we came up with these four:

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

In the next sections, we create a resource bundle using properties files. 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 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 = Locale.of("en", "US");
17:    var france = Locale.of("fr", "FR");
18:    printWelcomeMessage(us);     // Hello, The zoo is open
19:    printWelcomeMessage(france); // Bonjour, Le zoo est ouvert
20: }
```

Lines 16 and 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.

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 = Locale.of("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 that 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, doing so is 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 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 11.11 shows what Java goes through when asked for resource bundle `Zoo` with the locale `Locale.of("fr", "FR")` when the default locale is U.S. English.

TABLE 11.11 Picking a resource bundle for French/France with default locale English/US

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

As another way of remembering the order of Table 11.11, 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 the language/country, followed by just the language.
3. Use the default resource bundle if no matching locale can be found.

> **NOTE**
>
> 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 11.11. What is the maximum number of files that Java would need to consider in order to find the appropriate resource bundle with the following code?

```
Locale.setDefault(Locale.of("hi"));
ResourceBundle rb = ResourceBundle.getBundle("Zoo",
Locale.of("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 11.12 shows how to do this.

TABLE 11.12 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 11.11, 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 `en_US` *only if there is* *no matching `fr_FR` or `fr` resource bundle*. 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 Québec (which has a default locale of French Canada) who has asked the program to provide information in English. What do you think this outputs?

```
Locale.setDefault(Locale.of("en", "US"));
var locale = Locale.of("en", "CA");
ResourceBundle rb = ResourceBundle.getBundle("Zoo",
locale);

System.out.print(rb.getString("hello"));
System.out.print(". ");
System.out.print(rb.getString("name"));
System.out.print(" ");
System.out.print(rb.getString("open"));
System.out.print(" ");
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.

After line 12, the resource bundle is selected, and Java will only consider files it finds that are part of this resource bundle, namely `Zoo_en_CA.properties`, `Zoo_en.properties`, and `Zoo.properties`, in this order.

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 hierarchy 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. 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 retrieve a non-existent property results in an exception:

```
System.out.print(rb.getString("close"));  //
MissingResourceException
```

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:

```
helloGreeting=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.

Suppose we have a resource bundle `rb`:

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

This will 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 9, "Collections and Generics," 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 print it. This is commonly referred to as providing a default, or a 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
```

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 that the resources are properly closed. Resources are closed at the conclusion of the `try` block, in the reverse of the 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.

Java includes a number of built-in classes to format numbers and dates. We reviewed how to create custom formatters for each. You should be able to read these custom formats when you encounter them on the exam.

Localization involves creating programs that adapt to change. 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. You need to know how to format number and date/time values based on locale, including the new `CompactNumberFormat` class.

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 requested 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

Understand the various types of exceptions. All exceptions are subclasses of `java.lang.Throwable`. Subclasses of `java.lang.Error` should never be caught. Only subclasses of `java.lang.Exception` should be handled in application code.

Differentiate between checked and unchecked exceptions. Unchecked exceptions do not need to be caught or handled and are subclasses of `java.lang.RuntimeException` or `java.lang.Error`. All other subclasses of `java.lang.Exception` are checked exceptions and must be handled or declared.

Understand the flow of a try statement. A `try` statement must have a `catch` or a `finally` block. Multiple `catch` blocks can be chained together, provided no superclass exception type appears in an earlier `catch` block than its subclass. A multi-catch expression may be used to handle multiple exceptions in the same `catch` block, provided one exception is not a subclass of another. The `finally` block runs last regardless of whether an exception is thrown.

Be able to follow the order of a try-with-resources statement. A try-with-resources statement is a special type of `try` block in which one or more resources are declared and automatically closed in the reverse of the order in which they are declared. It can be used with or without a `catch` or `finally` block, with the implicit `finally` block always executed first.

Be able to write methods that declare exceptions. Understand the difference between the `throw` and `throws` keywords and how to declare methods with exceptions. Know how to correctly override a method that declares exceptions.

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, factory method, or builder class.

Format dates, numbers, and messages. Be able to format dates, numbers, and messages into various `String` formats, and know how locale influences these formats. Know how the various number formatters (currency, percent, compact) differ. Be able to write a custom date or number formatter using symbols, including knowing how to escape literal values.

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 can be inserted on line 8 to make this code compile? (Choose all that apply.)

```
7: public void whatHappensNext() throws IOException {
8:    // INSERT CODE HERE
9: }
```

   A. `System.out.println("it’s ok");`

   B. `throw new Exception();`

   C. `throw new IllegalArgumentException();`

   D. `throw new java.io.IOException();`

   E. `throw new RuntimeException();`

   F. None of the above

2. Which statement about the following class is correct?