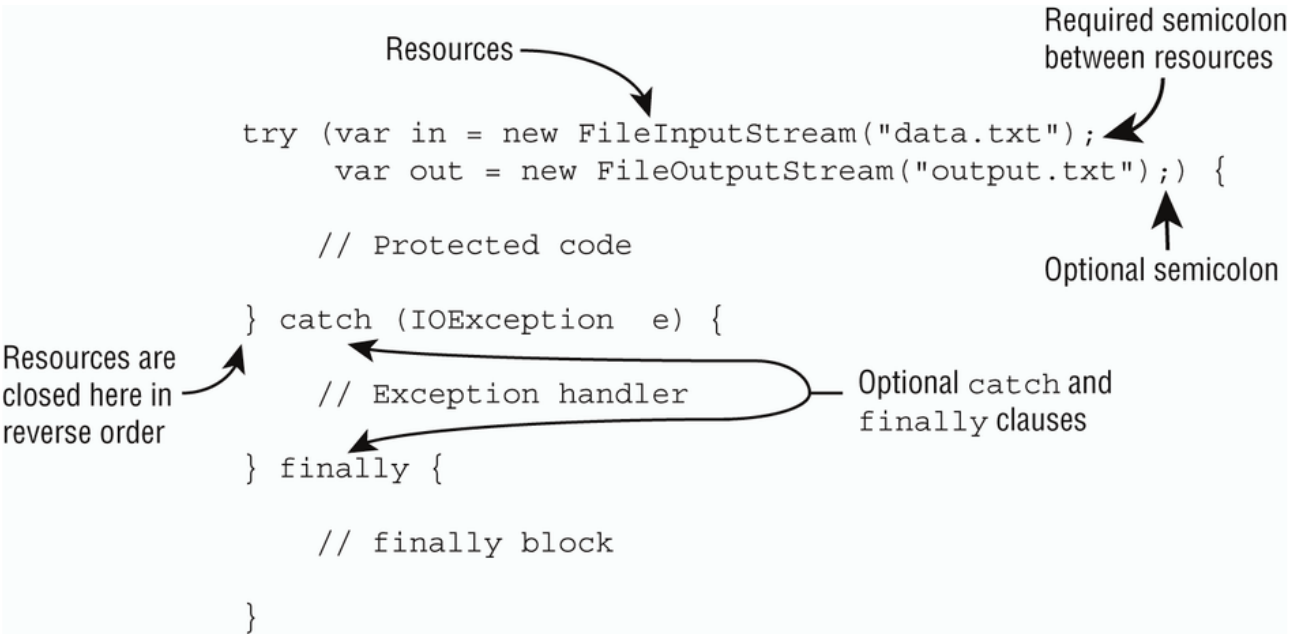
Chapter 16 - Exceptions Aasertions Localization



Examining Exception Categories

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

Checked exceptions

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

Inheriting Exception Classes

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

Adding Custom Constructors

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

```
1 public class CannotSwimException extends Exception {
2    public CannotSwimException() {
```

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

Automating Resource Management

Resource Management vs. Garbage Collection

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

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

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

Constructing Try-With-Resources Statements

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

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

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

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

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

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

The code prints the following at runtime:

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

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

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

The second rule you should be familiar with is: *a try-with-resources statement can include multiple resources, which are closed in the reverse order in which they are declared*. Resources are terminated by a semicolon (`;`), with the last one being optional.

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

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

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

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

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

Learning the New Effectively Final Feature

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

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

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

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

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

Understanding Suppressed Exceptions

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

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

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

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

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

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

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

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

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

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

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

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

match the `catch` clause, the exception is thrown to the caller.

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

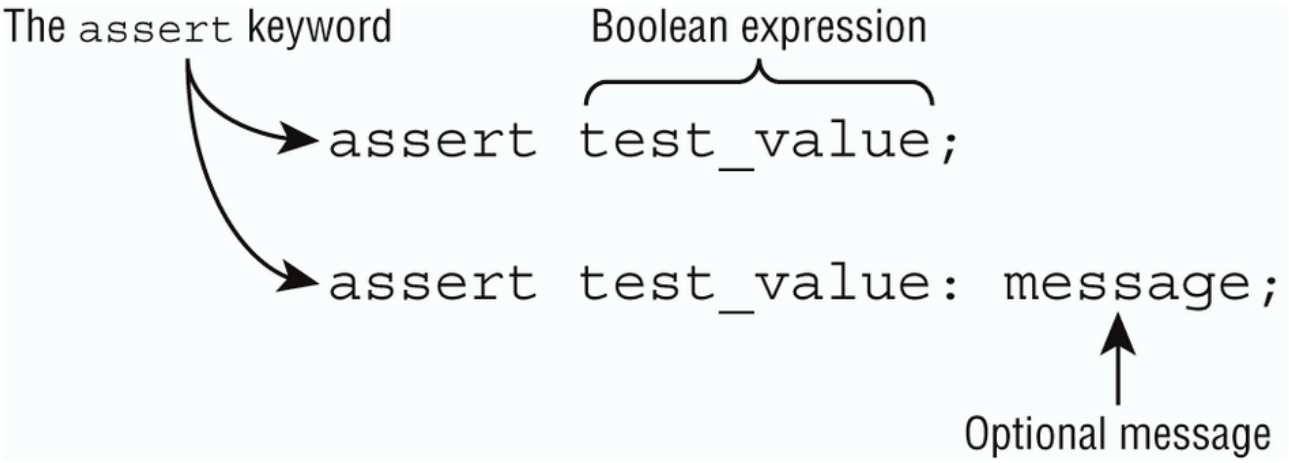
Keep in mind that **suppressed exceptions apply only to exceptions thrown in the `try` clause**. The following example does not throw a suppressed exception:

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

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

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

Declaring Assertions



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

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

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

- If assertions are disabled, Java skips the assertion and goes on in the code.

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

Enabling Assertions

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

```
1 java -enableassertions Rectangle
```

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

```
1 java -ea Rectangle
```

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

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

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

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

Disabling Assertions

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

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

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

Understanding Date and Time Types

TABLE 16.4 Date and time types

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

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

Formatting Dates and Times

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

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

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

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

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

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

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

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

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

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

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

Selecting a *format()* Method

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

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

```

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

Adding Custom Text Values

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

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

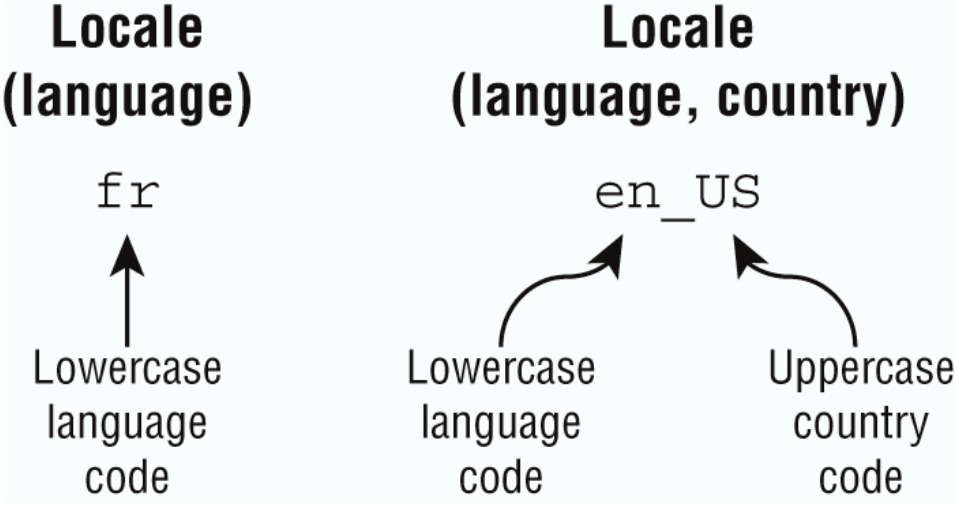
```
var dt = LocalDateTime.of(2020, Month.OCTOBER, 20, 6, 15, 30);

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

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

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

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

Supporting Internationalization and Localization



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

```
System.out.println(new Locale("fr"));       // fr
```

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

Localizing Numbers

TABLE 16.7 Factory methods to get a `NumberFormat`

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

Formatting Numbers

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

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

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

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

```

The output looks like this:

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

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.

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

```
String s = "40.45";

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

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

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

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

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

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

Writing a Custom Number Formatter

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

These examples should help illuminate how these symbols work:

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

Line 20 shows prefixing a nonformatting character (`$` sign) along with rounding because fewer digits are printed than available.

Localizing Dates

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

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

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

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

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

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

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

Specifying a Locale Category

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

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

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

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

Loading Properties with Resource Bundles

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

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

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

```
Zoo_en.properties
hello=Hello
open=The zoo is open

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

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

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

The `ResourceBundle` class also provides a `keySet()` method to get a set of all keys.

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);
```

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

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

Formatting Messages

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

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

Given a resource bundle `rb`:

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

```

that would then print the following:

```
Hello, Tammy and Henry
```

Using the *Properties* Class

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

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

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

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

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

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

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

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