```

    // Print with default locale
    Locale.setDefault(new Locale("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 language of the `spain` and `money` variables using the locale `en_US`. Then it prints it using the `DISPLAY` category of `es_ES`, while the `FORMAT` category 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(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 four:

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

In the next sections, we 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.

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

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 = new Locale("en", "US");
ResourceBundle rb = ResourceBundle.getBundle("Zoo", us);
rb.keySet().stream()
   .map(k -> k + ": " + rb.getString(k))
   .forEach(System.out::println);
```

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

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

Real World Scenario

Loading Resource Bundle Files at Runtime

For the exam, you don't need to know where the properties files for the resource bundles are stored. If the exam provides a properties file, it is safe to assume 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 `new Locale("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(new Locale("hi"));
ResourceBundle rb = ResourceBundle.getBundle("Zoo", new Locale("en"));
```

The answer is three. They are listed here:

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

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

Selecting Resource Bundle Values

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

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

The program prints the following:

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

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

Line 14 doesn't find a match for the key `hello` in `Zoo_en_CA.properties`, so it goes up the hierarchy to `Zoo_en.properties`. Line 16 doesn't find a match for `name` in either of the first two properties files, so it has to go all the way to the top of the 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: `Zoo_en_CA.properties`, `Zoo_en.properties`, and `Zoo.properties`. Even when the property wasn't found in `en_CA` or `en` resource bundles, the program preferred using `Zoo.properties` (the default resource bundle) rather than `Zoo_en_US.properties` (the default locale).

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

Formatting Messages

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

For example, suppose that we had this property defined:

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

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

Suppose we have a resource bundle `rb`:

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