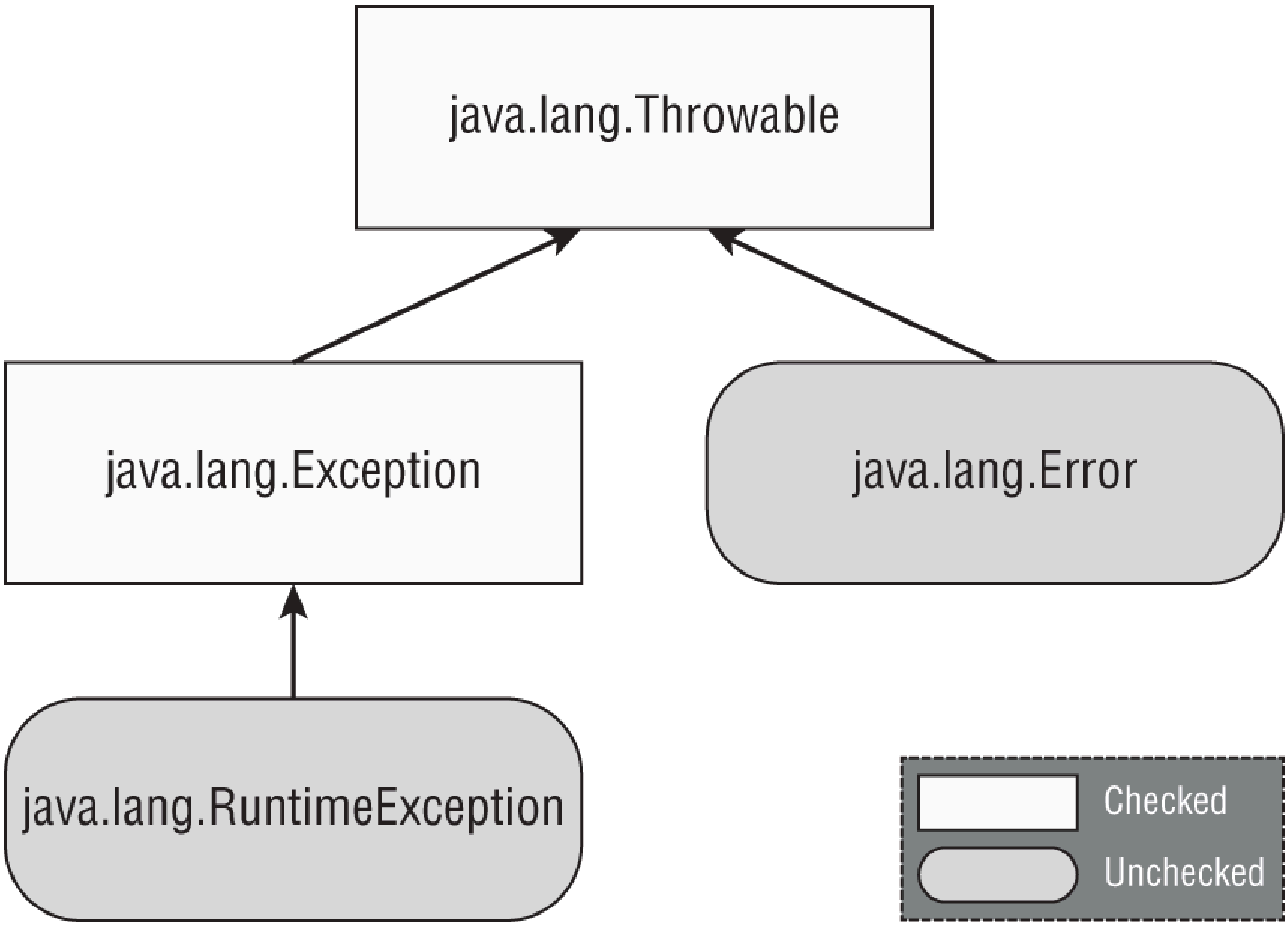
Capitolul 11 - Exceptions and Localization

Exceptions

Is Java way of saying "I give up. I don't know what to do. You deal it".
Is an event that alters program flow.



Checked exceptions

Is an exception that must be declared or handled by the application code where it's thrown.

Handle / 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.

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

-----------------------------------

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

Unchecked exception

Is any exception that does not need to be declared or handled by the application code it is thrown.
Known also like *runtime exceptions*.

Runtime exception

Is defined as the `RuntimeException` class and its subclasses.
Tend to be unexcepted but not necessarily fatal.

Error

Means something went so horrible wrong that your program should not attempt to recover from it.

Throwable

Parent class of all exceptions, including the `Error` class.

Reviewing exceptions types

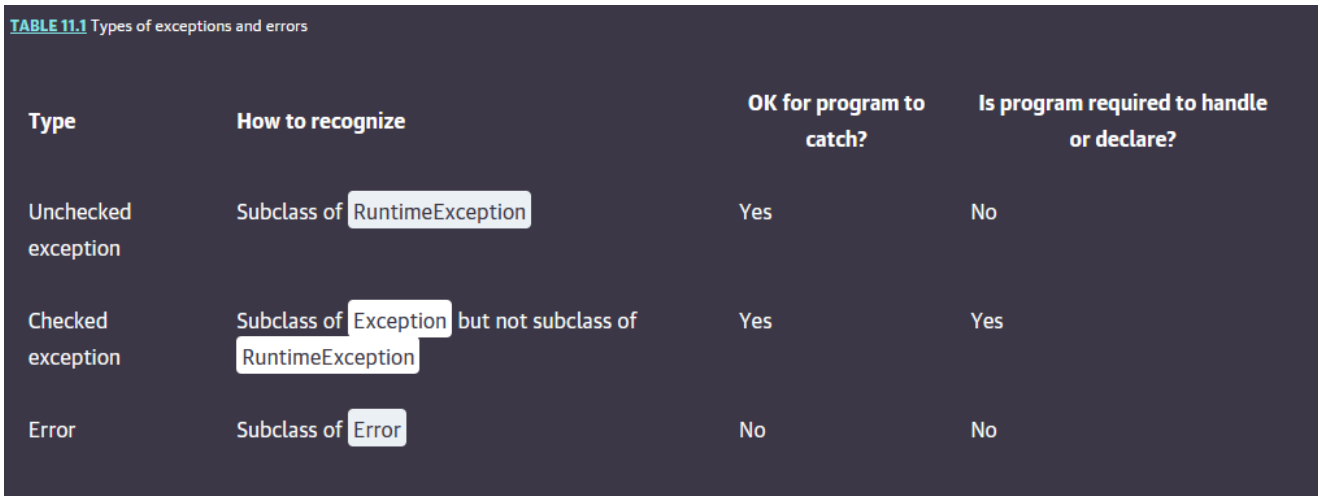


TABLE 11.1 Types of exceptions and errors

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

Calling methods that throw exceptions

```
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");
    }
}
```

Overriding methods with Exceptions

```
class CanNotHopException extends Exception {}

class Hopper {
   public void hop() {}
}

public class Bunny extends Hopper {
   public void hop() throws CanNotHopException {}  // DOES NOT
COMPILE
}
// reason → hop() isn't allowed to throw any checked exceptions
because the hop() method in the superclass Hopper doesn't declare
any.
```

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

```
public static void main(String[] args) {
    try {
        hop();
    } catch (Exception e) {
        System.out.println(e + "\n"); //print all object
        System.out.println(e.getMessage()+ "\n"); //print the
message
        e.printStackTrace(); //print where the stack trace
comes
    }
}
private static void hop() {
    throw new RuntimeException("cannot hop");
}
```

Exception classes

RuntimeException

> Are unchecked exceptions that don't have to be handled or declared. they can be thrown by the programmer or the JVM.

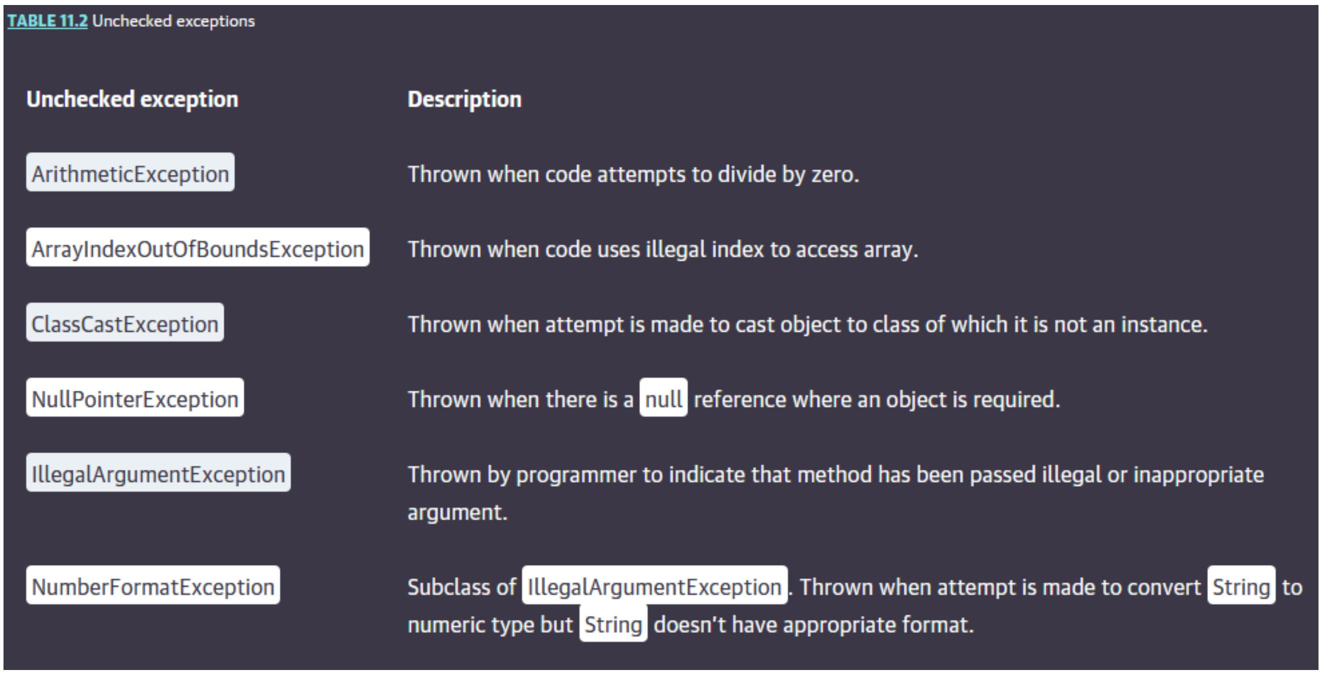


TABLE 11.2 Unchecked exceptions

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.

Checked *Exceptions* classes

> Checked exceptions have `Exception` in their hierarchy but not `RuntimeException`.
> They must be handled or declared.

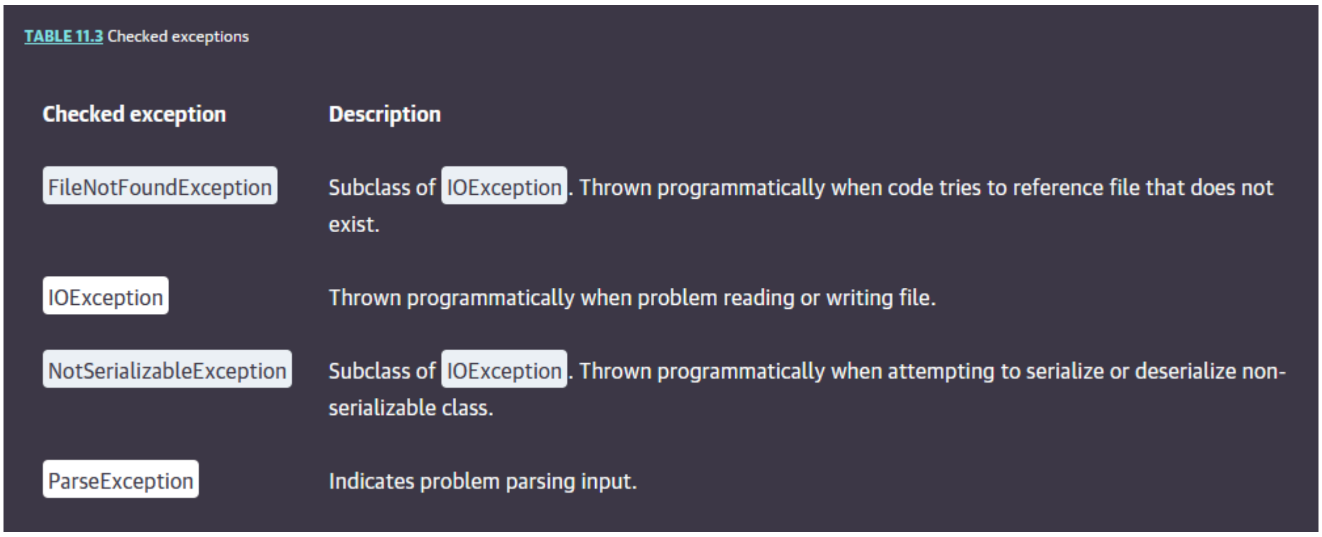


TABLE 11.3 Checked exceptions

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.

Error classes

> Are unchecked exceptions that extend the `Error` class.
> They are thrown by the JVM and should not be handled or declared.

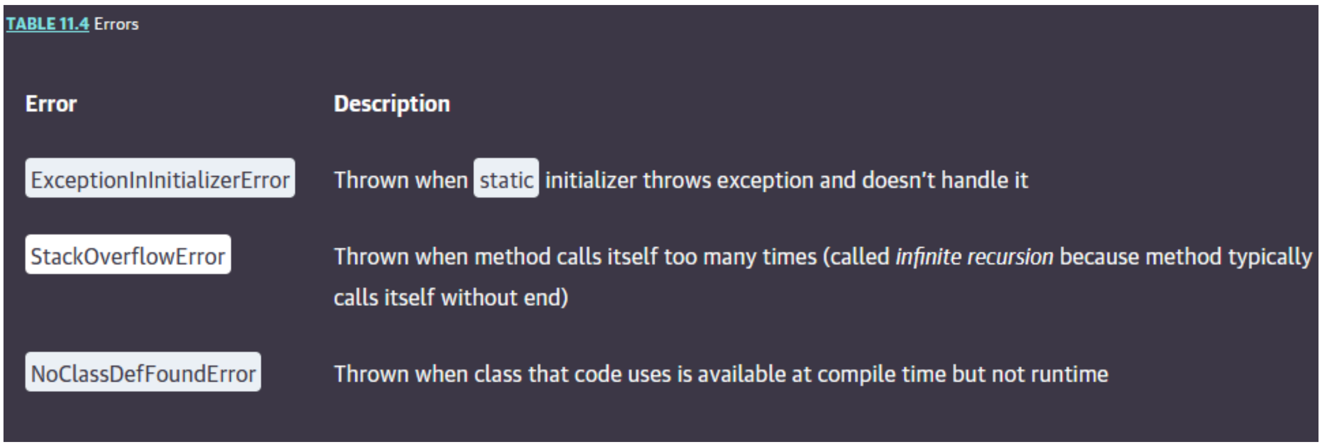
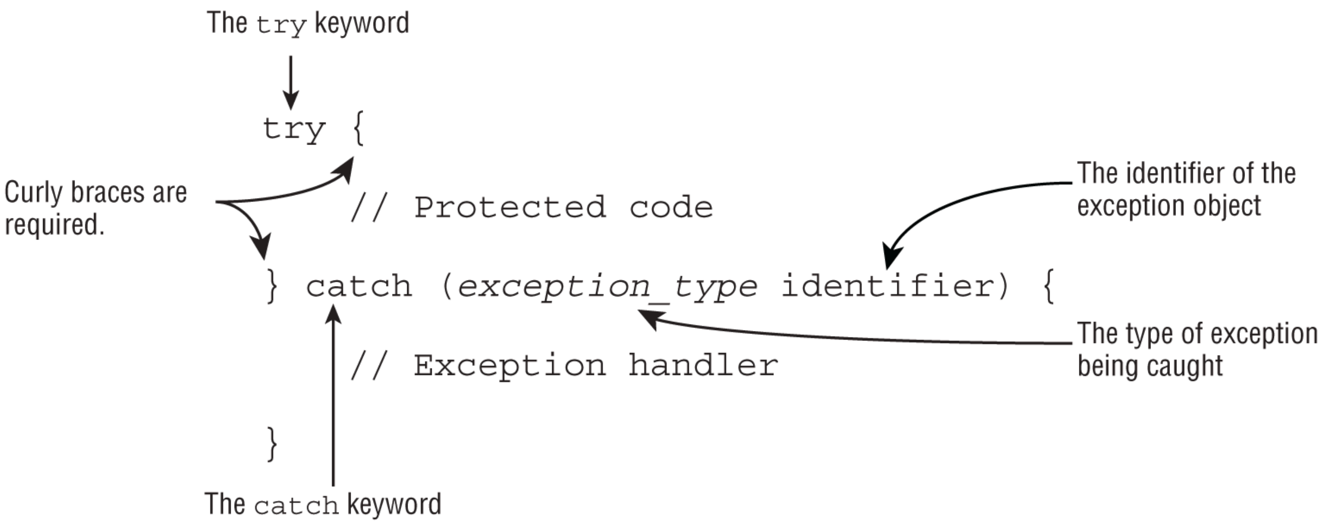


TABLE 11.4 Errors

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

Handling Exceptions

`try` and `catch` statements


The try keyword
Curly braces are required.
The identifier of the exception object
The type of exception being caught
The catch keyword

```
try {
   // Protected code
} catch (exception_type identifier) {
   // Exception handler
}
```

```
3:  void explore() {
4:      try {
        fall();
        System.out.println("never get here");
    } catch (RuntimeException e) {
        getUp();
    }
    seeAnimals();
}
void fall() {  throw new RuntimeException(); }
```

Chaining *catch* Blocks

```
class AnimalsOutForAWalk extends RuntimeException {}

class ExhibitClosed extends RuntimeException {}

class ExhibitClosedForLunch extends ExhibitClosed {}

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");
   }
}
```

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.

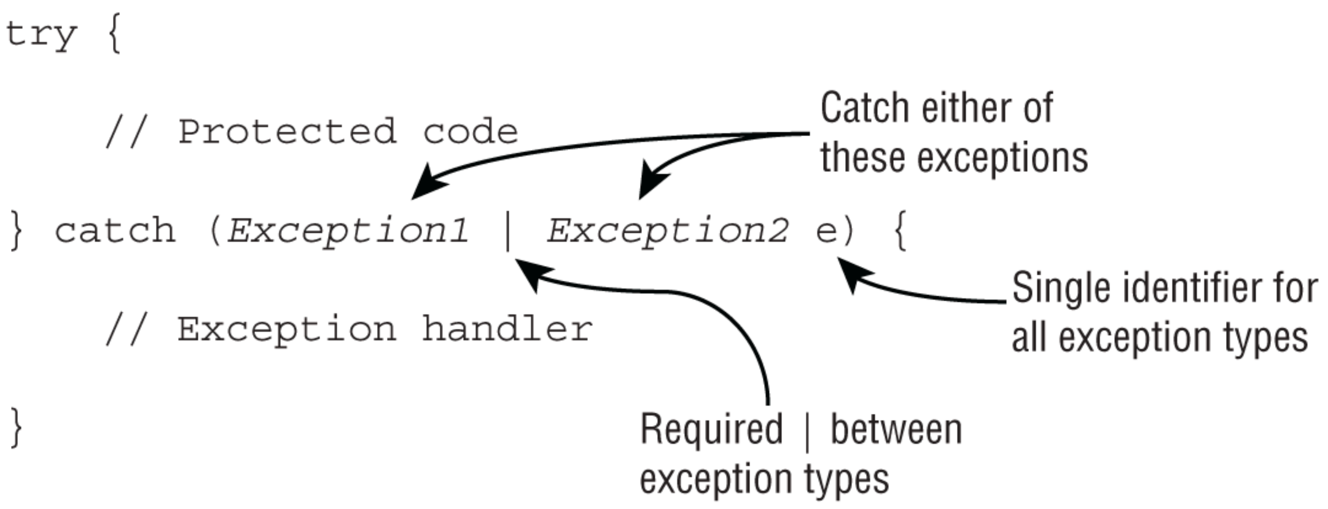
```
public void visitMonkeys() {
   try {
      seeAnimal();
   } catch (ExhibitClosedForLunch e) {   // Subclass exception
      System.out.print("try back later");
   } catch (ExhibitClosed e) {           // Superclass exception
      System.out.print("not today");
   }
}
```

```java
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");
   }
}
```

Remember that an exception defined by the `catch` statement is only in scope for that `catch` block.

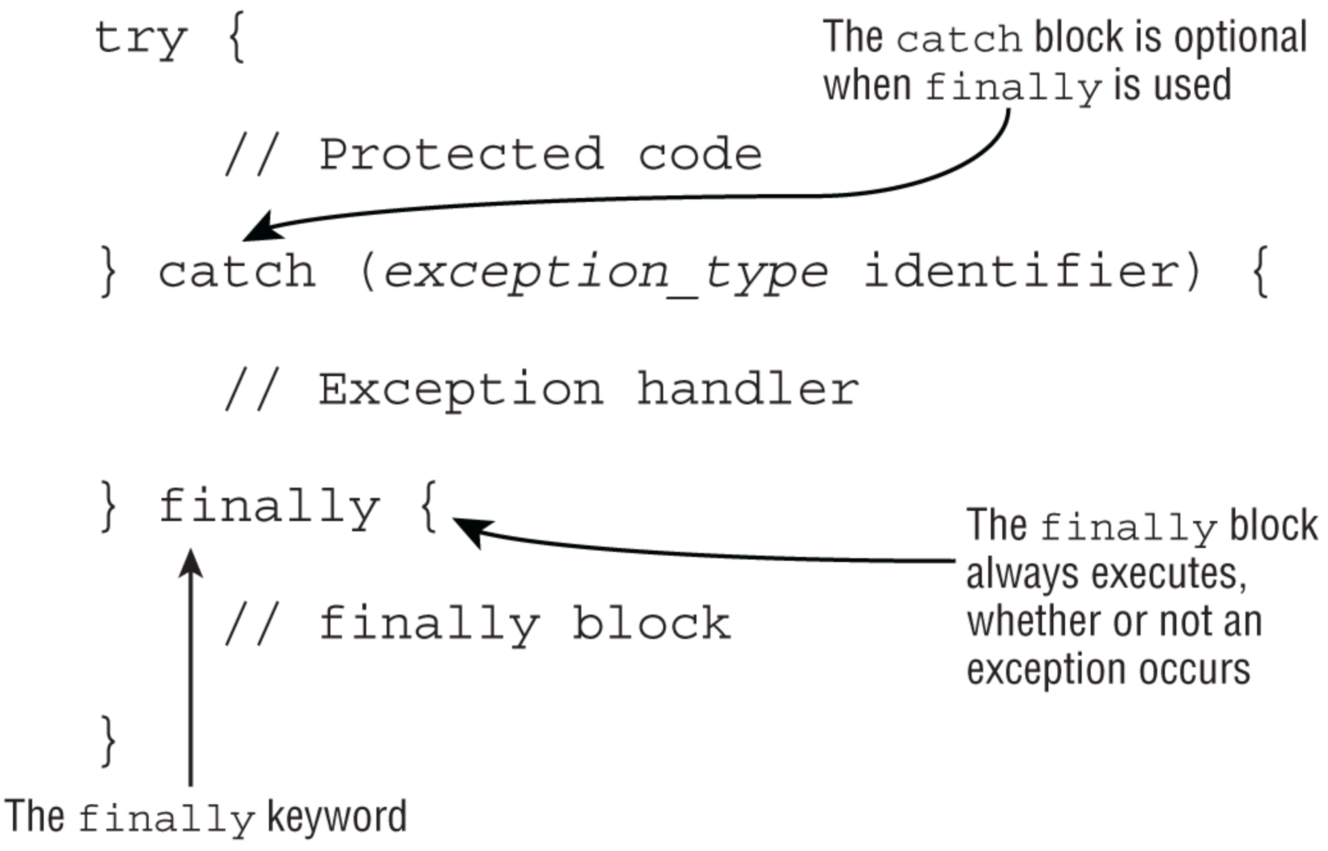
Applying a Multi-catch block

```java
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");
   }
}

---putem face asta---

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");
   }
}
```


try {
// Protected code
} catch (Exception1 | Exception2 e) {
// Exception handler
}
Catch either of these exceptions
Single identifier for all exception types
Required | between exception types

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

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

Adding a *finally* Block


try {
// Protected code
} catch (exception_type identifier) {
// Exception handler
} finally {
// finally block
}
The catch block is optional when finally is used
The finally block always executes, whether or not an exception occurs
The finally keyword

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.

```
12: void explore() {
13:     try {
14:         seeAnimals();
15:         fall();
16:     } catch (Exception e) {
17:         getHugFromDaddy();
18:     } finally {
19:         seeMoreAnimals();
20:     }
21:     goHome();
22: }
```

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.

```
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.

```
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(0)`

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.

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

Automating Resource Management

> External data sources are referred to as resources.

Introducing Try-with-resources

```java
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();
         }
      }
```

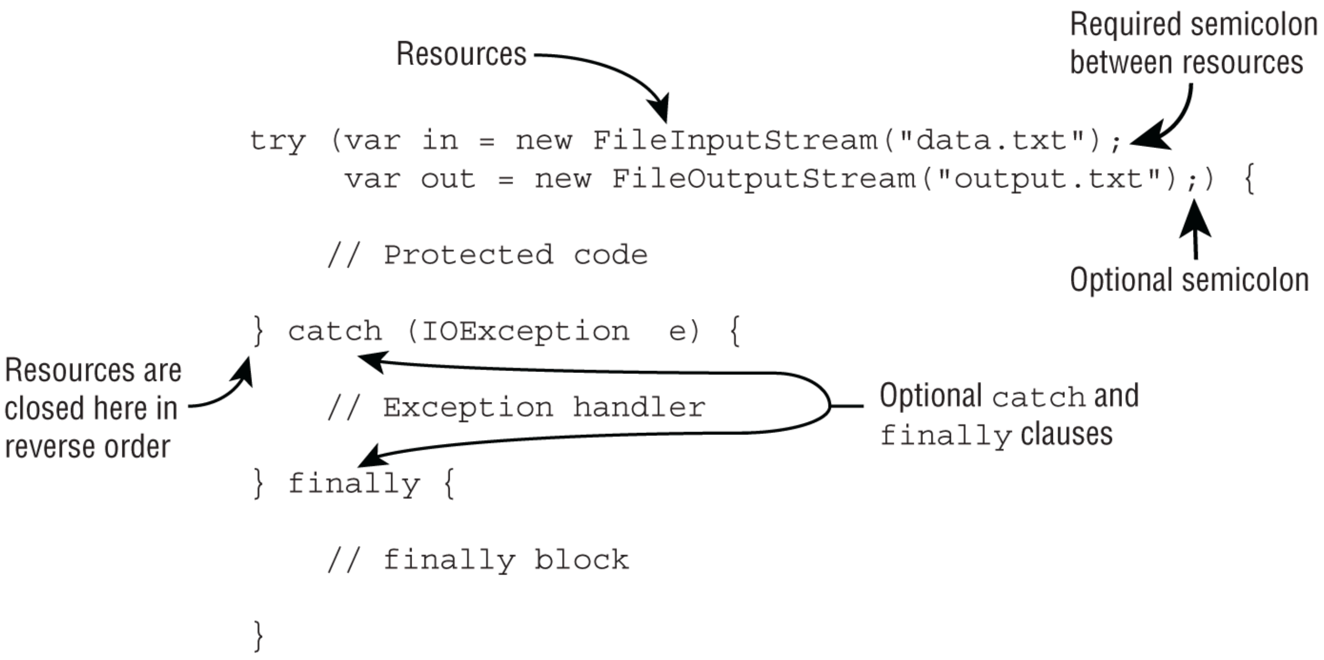
```
    }
}
```

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.

```
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.

Basics of Try-with-resources


Resources
Required semicolon between resources
Optional semicolon
Resources are closed here in reverse order
Optional catch and finally clauses

```
try (var in = new FileInputStream("data.txt");
     var out = new FileOutputStream("output.txt");) {

    // Protected code

} catch (IOException  e) {

    // Exception handler

} finally {

    // finally block

}
```

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.

Catch block is *optional* with a try-with-resources statement.

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

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

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

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

Declaring resources

> While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement.
> You can declare a resource using `var` as the data type in a try-with-resources statement, since resources are local variables.

Scope of Try-of-resources

```java
try (Scanner s = new Scanner(System.in)) {
    s.nextLine();
} catch(Exception e) {
    s.nextInt(); // DOES NOT COMPILE
} finally {
    s.nextInt(); // DOES NOT COMPILE
}
```

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.

```
try (Scanner s = new Scanner(System.in)) {
   s.nextLine();
} catch(Exception e) {
   s.nextInt(); // DOES NOT COMPILE
} finally {
   s.nextInt(); // DOES NOT COMPILE
}
```

Following Order of Operations

```
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");
   } }
```

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

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. The syntax uses the resource name in place of the resource declaration, separated by a semicolon (`;`).

```
public static void main(String… xyz) {
   final var bookReader = new MyFileClass(4);
   MyFileClass movieReader = new MyFileClass(5);
   try (bookReader;
         var tvReader = new MyFileClass(6);
         movieReader) {
      System.out.println("Try Block");
```

```
    } finally {
        System.out.println("Finally Block");
    } }
```

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

Understanding Suppressed Exceptions

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.

Formatting values

Format numbers

For this, we'll use `NumberFormat` abstract class with following 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.

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

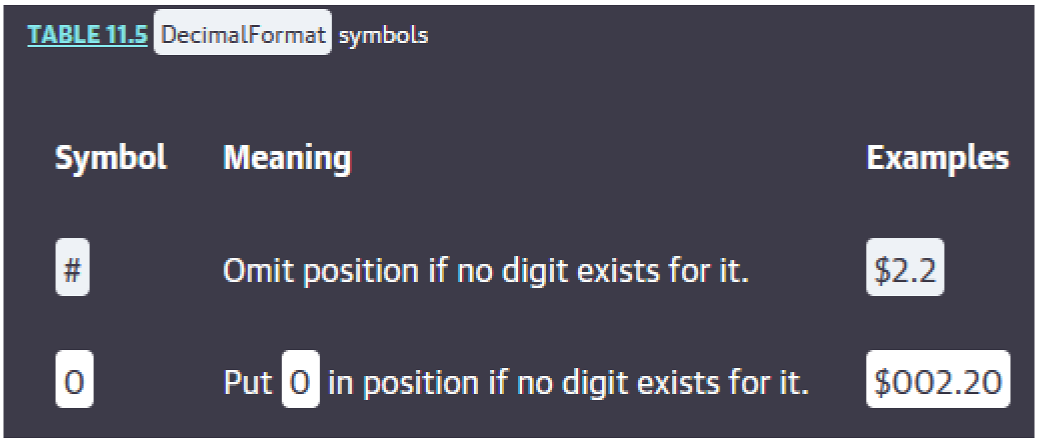


TABLE 11.5 DecimalFormat symbols

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

```
double d = 1234.567;
NumberFormat f1 = new DecimalFormat("###,###,###.0");
```

```
System.out.println(f1.format(d));  // 1,234.6

NumberFormat f2 = new DecimalFormat("000,000,000.00000");
System.out.println(f2.format(d));  // 000,001,234.56700

NumberFormat f3 = new DecimalFormat("Your Balance $#,###,###.##");
System.out.println(f3.format(d));  // Your Balance $1,234.57
```

Formatting Dates and Times

`DateTimeFormatter`

```
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_TIME));
```

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

The `DateTimeFormatter` will throw an exception if it encounters an incompatible type.

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.

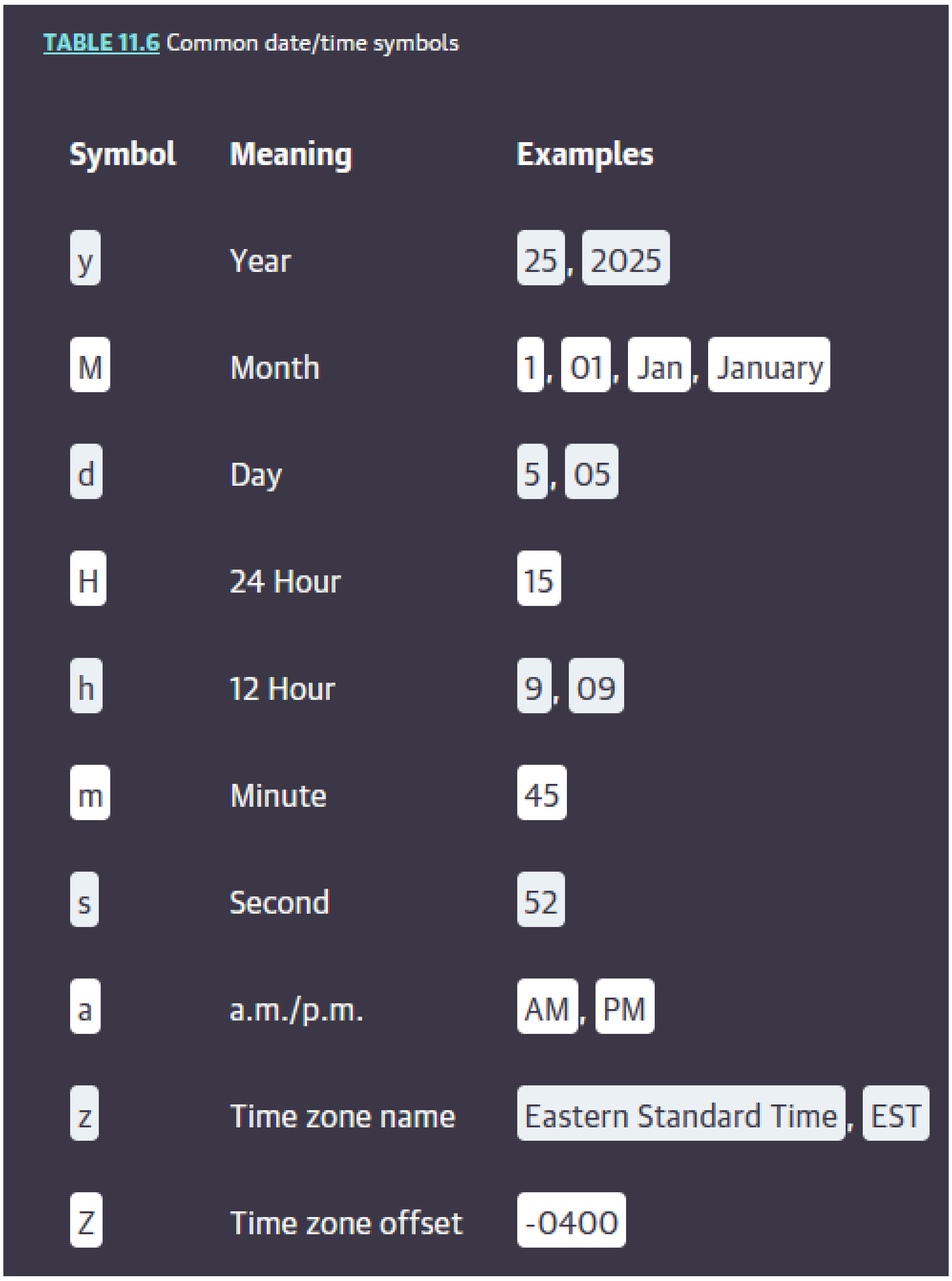


TABLE 11.6 Common date/time symbols

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`

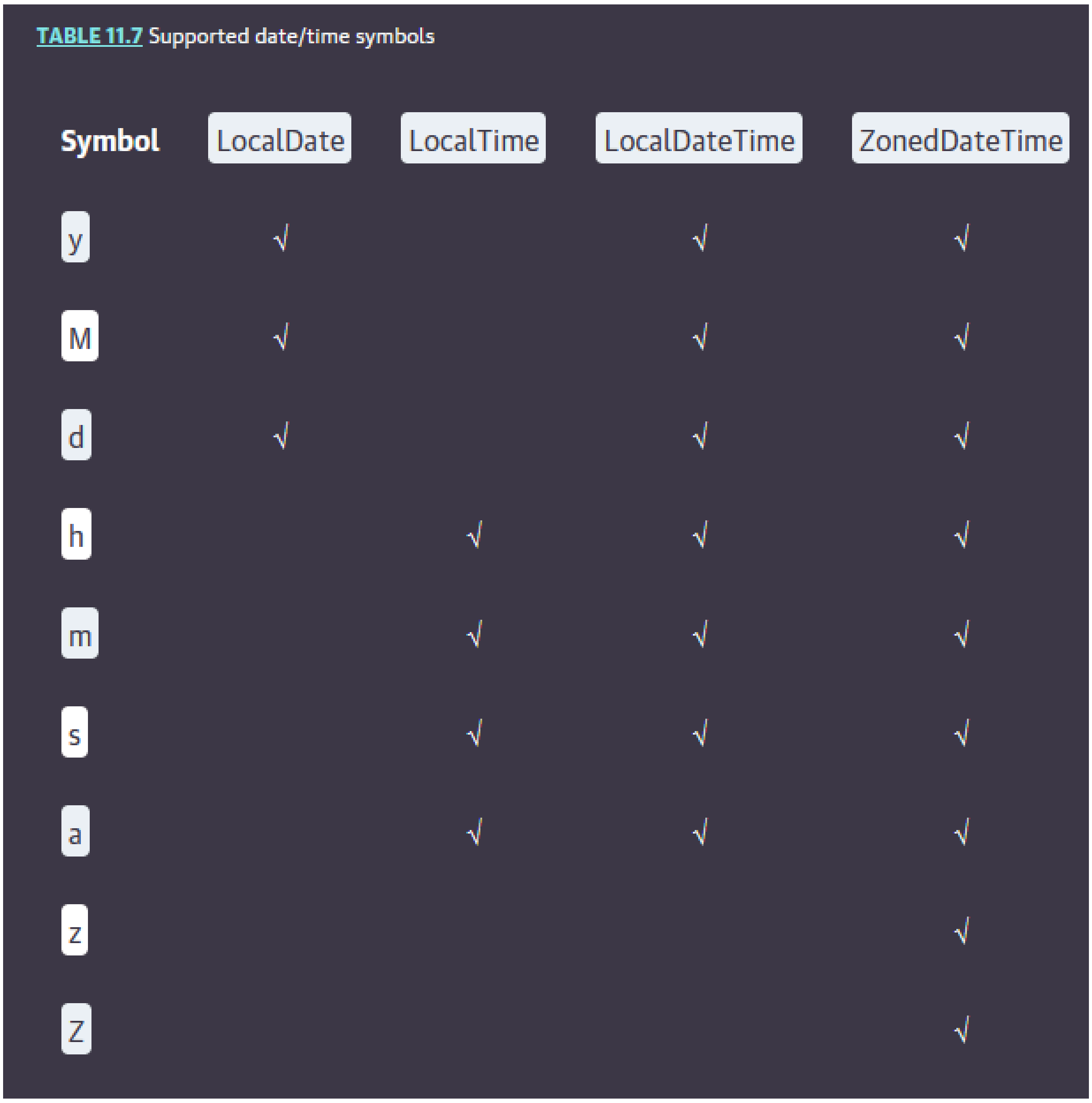


TABLE 11.7 Supported date/time symbols

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

Selecting a *format()* Method

```
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
```

Adding Custom text values

```
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));
---------------------------

var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at' hh:mm");
System.out.println(dt.format(f));  // October 20, 2025 at 06:15

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!
```

Supporting Internationalization and Localization

Internationalization

Process of designing your program so it can be adapted.

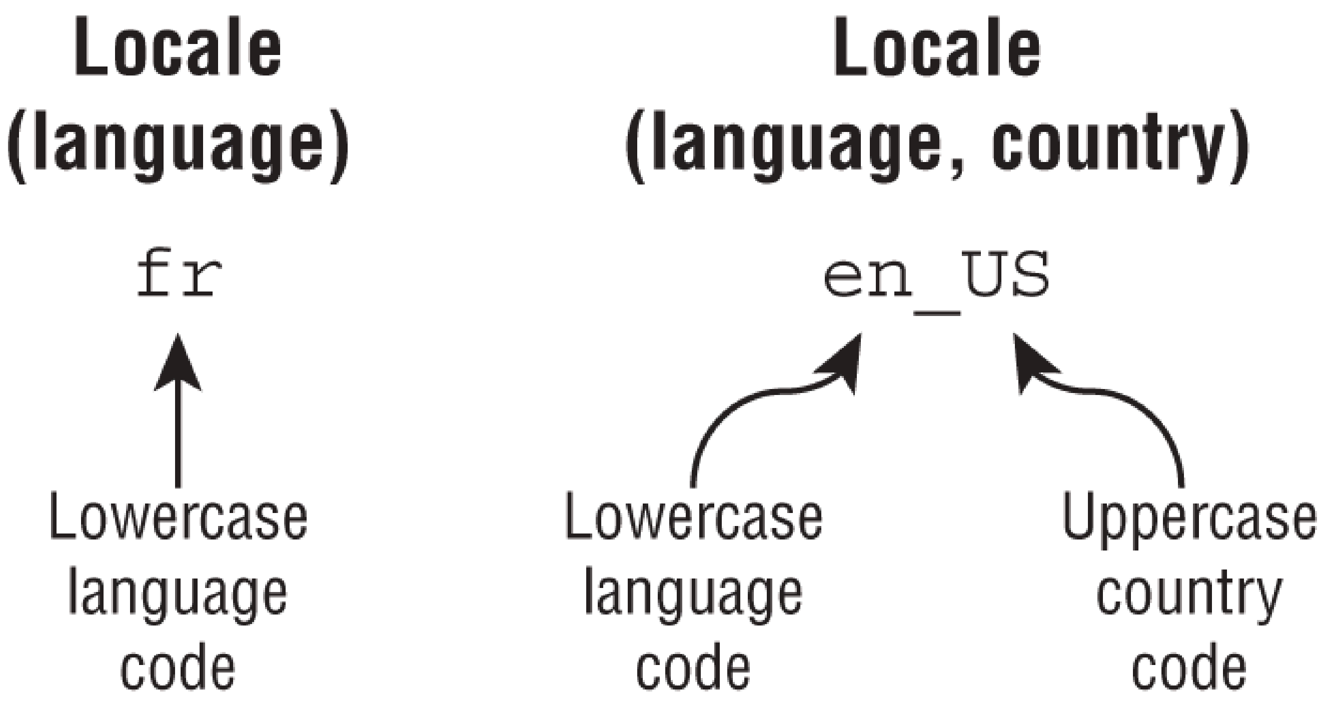
Localization

Supporting multiple locales or geographic regions.

Picking a Locale

> The `Locale` class is in the `java.util` package. The first useful `Locale` to find is the user's current locale.

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



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

----------------

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

-----------------------

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

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

//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
```

Localizing Numbers

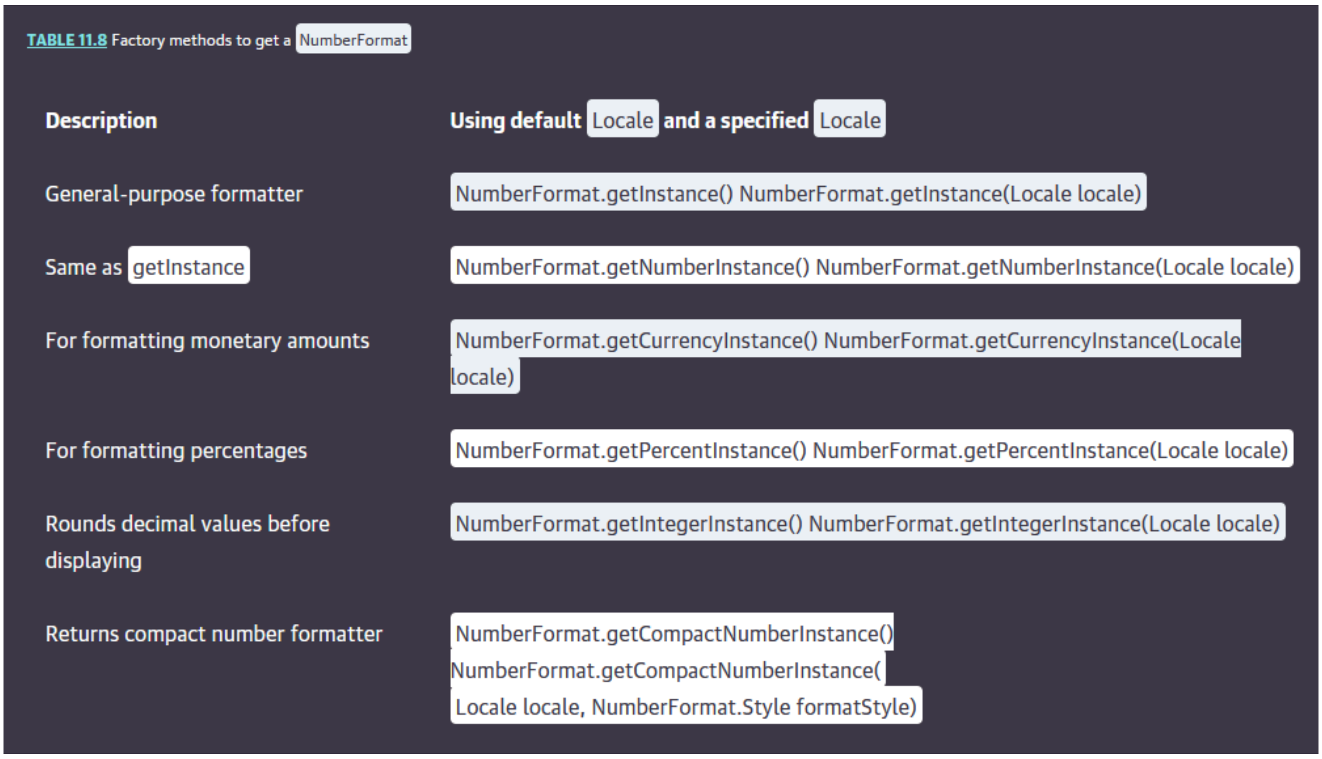


TABLE 11.8 Factory methods to get a NumberFormat

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)

Formatting Numbers

```
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


-------------


double price = 48;
var myLocale = NumberFormat.getCurrencyInstance();
System.out.println(myLocale.format(price));


--------------------
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 %
```

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.

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
----

//The parse() method is also used for parsing currency.

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

Formatting with *CompactNumberFormat*

CompactNumberFormat

Similar with `DecimalFormat`, but it's designed to be used in places where print space may be limited.

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);

--------------------
7M
7M
7 million

7 Mio.
7 Millionen

7,123,456
```

Localizing Dates

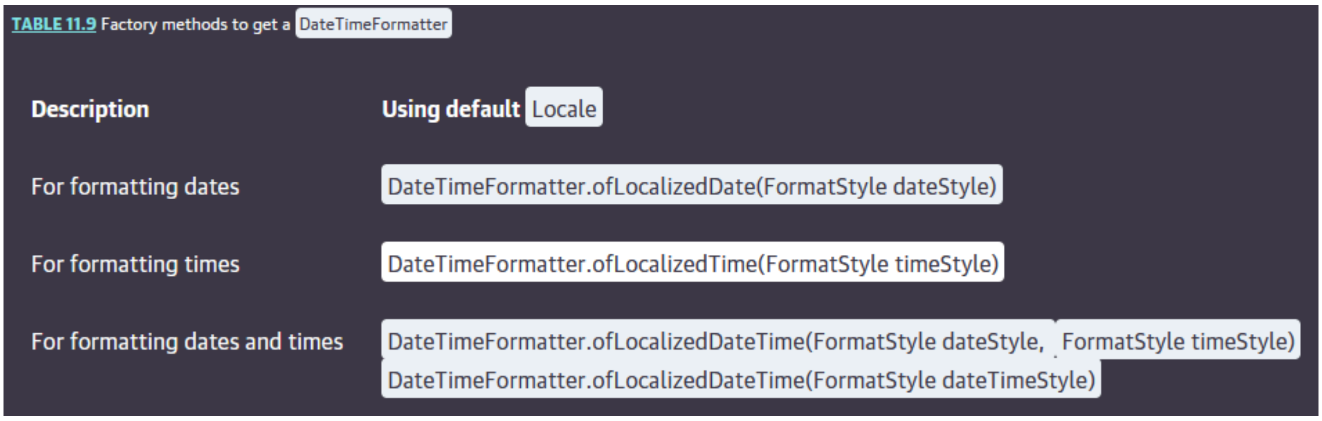


TABLE 11.9 Factory methods to get a DateTimeFormatter

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)

```
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);
}
```

Specifying a Locale Category

The `Locale.Category` enum is a nested element in `Locale` that supports distinct locales for displaying and formatting data.

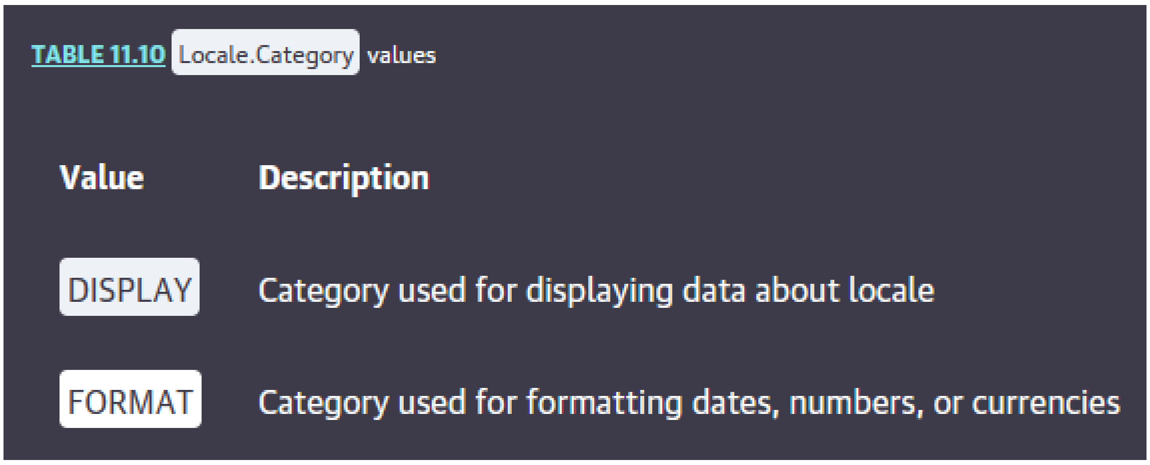


TABLE 11.10 `Locale.Category` values

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

```
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
}
```

Loading Properties with Resource Bundles

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.

Creating a Resource Bundle

```
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 = Locale.of("en", "US");
    var france = Locale.of("fr", "FR");
    printWelcomeMessage(us);      // Hello, The zoo is open
    printWelcomeMessage(france); // Bonjour, Le zoo est ouvert
}
```

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);

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

Picking a Resource Bundle

```
ResourceBundle.getBundle("name"); //default locale
ResourceBundle.getBundle("name", locale); // specify locale
```

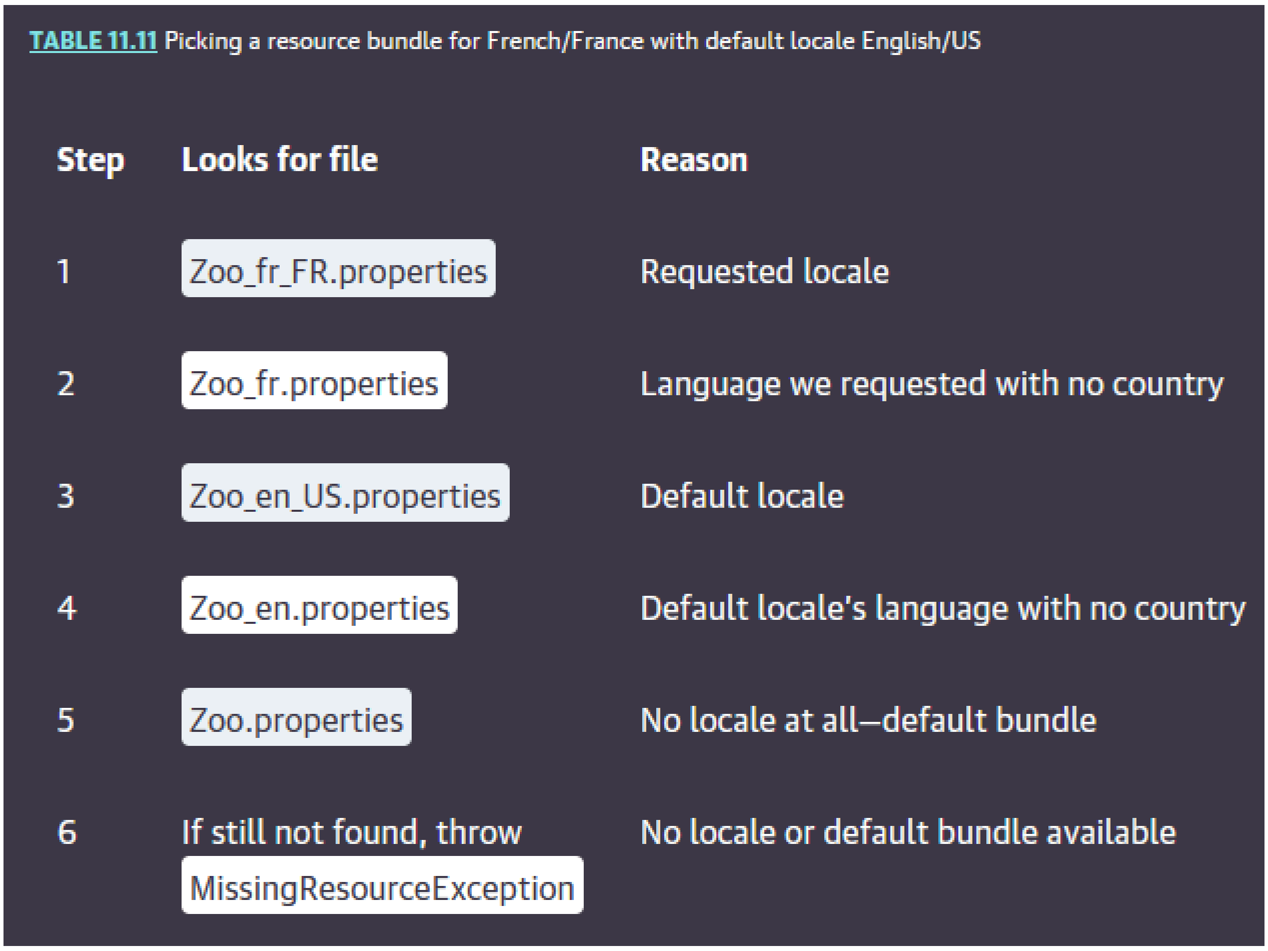


TABLE 11.11 Picking a resource bundle for French/France with default locale English/US

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, 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.

Selecting Resource Bundle Values

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.

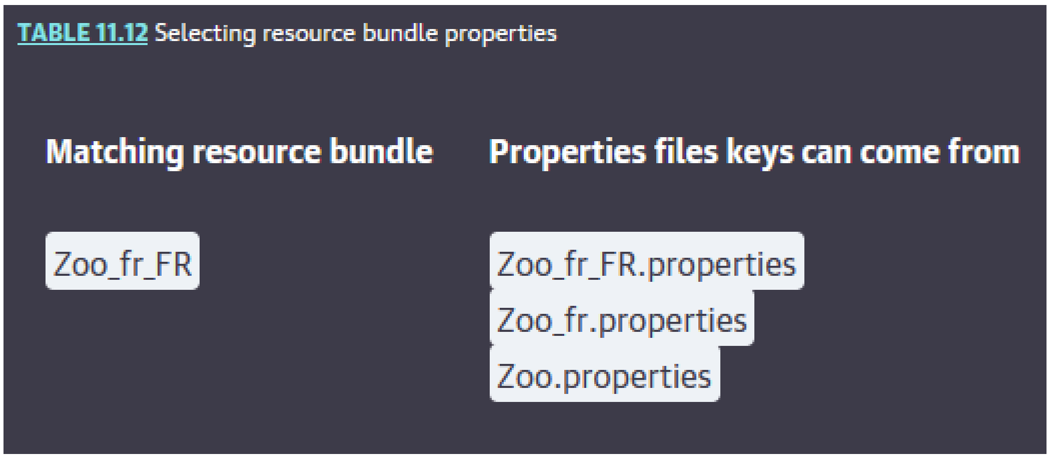


TABLE 11.12 Selecting resource bundle properties

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*.

```
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
```

```
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"));

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

Formatting Messages

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.

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

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

```
Hello, Tammy and Henry
```

Using the *Properties* Class

```
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
```