# Exception Handling 7

Chapter Topics

- Understanding program execution and the exception handling facility
- Recognizing the inheritance hierarchy of the `Throwable` class and its subclasses `Exception`, `RuntimeException`, and `Error` that define exception classes in Java
- Distinguishing between checked and unchecked exceptions
- Defining customized exception types
- Understanding the `try` - `catch` - `finally` construct and the control flow paths through it
- Using multiple `catch` clauses with the `try` statement
- Creating and throwing exceptions programmatically with the `throw` statement
- Using the `throws` clause to specify checked exceptions in the method header to propagate exceptions
- Using uni- `catch` and multi- `catch` clauses with the `try` statement to catch exceptions
- Rethrowing exceptions and handling chained exceptions
- Using the `try` -with-resources statement and implementing the `AutoCloseable` interface
- Handling suppressed exceptions

| Java SE 17 Developer Exam Objectives | |
|---|---|
| [4.1] Handle exceptions using try/catch/finally, try-with-resources, and multi-catch blocks, including custom exceptions | §7.2, p. 375<br>§7.3, p. 375<br>§7.6, p. 397<br>§7.7, p. 407 |

| Java SE 11 Developer Exam Objectives | |
|---|---|
| [4.1] Handle exceptions in the Java program by using try/catch/finally clauses, try-with-resource, and multi-catch statements | §7.3, p. 375 |

| | |
|---|---|
| | §7.6, p. 397<br>§7.7, p. 407 |
| [4.2] Create and use custom exceptions | §7.2, p. 375 |

An exception in Java signals the occurrence of an event that disrupts the normal flow of program execution. Such an event typically occurs due to violation of some semantic constraint of the Java programming language during execution—for example, a requested file cannot be found, an array index is out of bounds, or a network link failed. Inserting explicit checks in the code for such events can easily result in less comprehensible code. Java provides an exception handling mechanism for systematically dealing with such events.

## 7.1 Stack-Based Execution and Exception Propagation

The exception mechanism is built around the *throw-and-catch* paradigm. To *throw* an exception is to signal that an unexpected event has occurred. To *catch* an exception is to take appropriate action to deal with the exception. An exception is caught by an *exception handler*, and the exception need not be caught in the same context in which it was thrown. The runtime behavior of the program determines which exceptions are thrown and how they are caught. The throw-and-catch principle is embedded in the `try-catch-finally` construct (p. 375).

Several threads can be executing at the same time in the JVM (§22.2, p. 1369). Each thread has its own *JVM stack* (also called a *runtime stack*, *call stack*, or *invocation stack* in the literature) that is used to handle execution of methods. Each element on the stack is called an *activation frame* or a *stack frame* and corresponds to a method call. Each new method call results in a new activation frame being pushed on the stack, which stores all the pertinent information such as the local variables. The method with the activation frame on the top of the stack is the one currently executing. When this method finishes executing, its activation frame is popped from the top of the stack. Execution then continues in the method corresponding to the activation frame that is now uncovered on the top of the stack. The methods on the stack are said to be *active*, as their execution has not completed. At any given time, the active methods on a JVM stack make up what is called the *stack trace* of a thread's execution.

Example 7.1 is a simple program to illustrate method execution. It calculates the average for a list of integers, given the sum of all the integers and the number of integers. It uses three methods:

- The method `main()` calls the method `printAverage()` with parameters supplying the total sum of the integers and the total number of integers, (1).
- The method `printAverage()` in turn calls the method `computeAverage()`, (3).
- The method `computeAverage()` uses integer division to calculate the average and returns the result, (7).

**Example 7.1** *Method Execution*

Click here to view code image

```
public class Average1 {

  public static void main(String[] args) {
    printAverage(100, 20);                                          // (1)

    System.out.println("Exit main().");                             // (2)
  }

  public static void printAverage(int totalSum, int totalCount) {
    int average = computeAverage(totalSum, totalCount);             // (3)
    System.out.println("Average = " +                               // (4)
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");                     // (5)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");                       // (6)
    return sum/count;                                               // (7)
  }
}
```

Output of program execution:

Click here to view code image

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Execution of **Example 7.1** is illustrated in **Figure 7.1**. Each method execution is shown as a box with the local variables declared in the method. The height of the box indicates how long a method is active. Before the call to the method `System.out.println()` at (6) in **Figure 7.1**, the stack trace comprises the three active methods: `main()`, `printAverage()`, and `computeAverage()`. The result `5` from the

method `computeAverage()` is returned at (7) in **Figure 7.1**. The output from the program corresponds with the sequence of method calls in **Figure 7.1**. As the program terminates normally, this program behavior is called *normal execution*.

If the method call at (1) in **Example 7.1**

**Click here to view code image**

```
printAverage(100, 20);                                  // (1)
```

is replaced with

**Click here to view code image**

```
printAverage(100, 0);                                   // (1)
```

and the program is run again, the output is as follows:

**Click here to view code image**

```
Computing average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

**Figure 7.2** illustrates the program execution when the method `printAverage()` is called with the arguments `100` and `0` at (1). All goes well until the `return` statement at (7) in the method `computeAverage()` is executed. An error event occurs in calculating the expression `sum/number` because integer division by 0 is an illegal operation. This event is signaled by the JVM by *throwing* an `ArithmeticException` (**p. 372**). This exception is *propagated* by the JVM through the JVM stack as explained next.

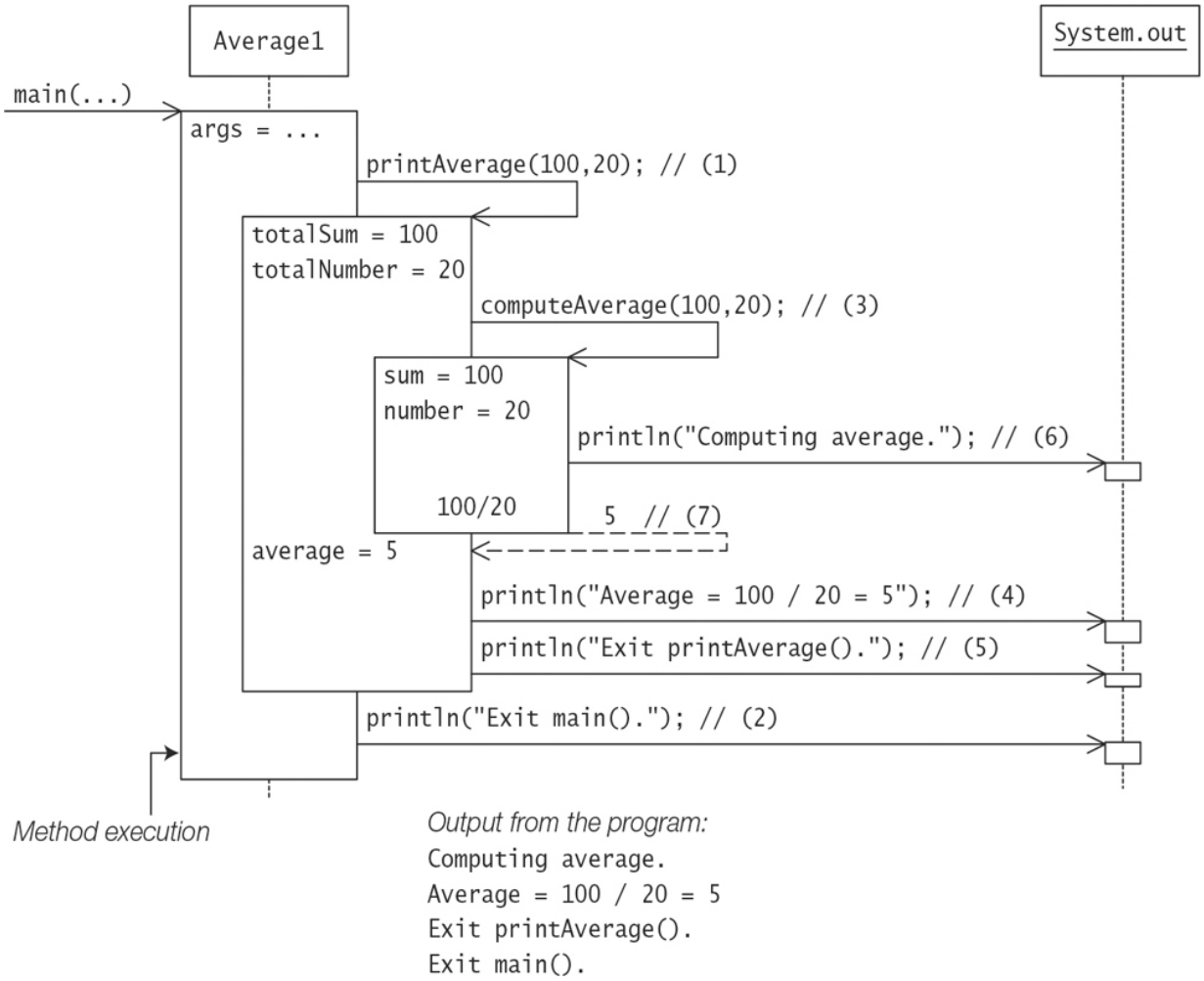



**Figure 7.1** *Normal Method Execution*

Figure 7.2 illustrates the case where an exception is thrown and the program does not take any explicit action to deal with the exception. In Figure 7.2, execution of the `computeAverage()` method is suspended at the point where the exception is thrown. The execution of the `return` statement at (7) never gets completed. Since this method does not have any code to deal with the exception, its execution is likewise terminated abruptly and its activation frame popped. We say that the method *completes abruptly*. The exception is then offered to the method whose activation is now on the top of the stack (`printAverage()`). This method does not have any code to deal with the exception either, so its execution completes abruptly. The statements at (4) and (5) in the method `printAverage()` never get executed. The exception now propagates to the last active method (`main()`). This does not deal with the exception either. The `main()` method also completes abruptly. The statement at (2) in the `main()` method never gets executed. Since the exception is not *caught* by any of the active methods, it is dealt with by the main thread's *default exception handler*. The default exception handler usually prints the name of the exception, with an explanatory message, followed by a printout of the stack trace at the time the exception was thrown. An uncaught exception, as in this case, results in the death of the thread in which the exception occurred.

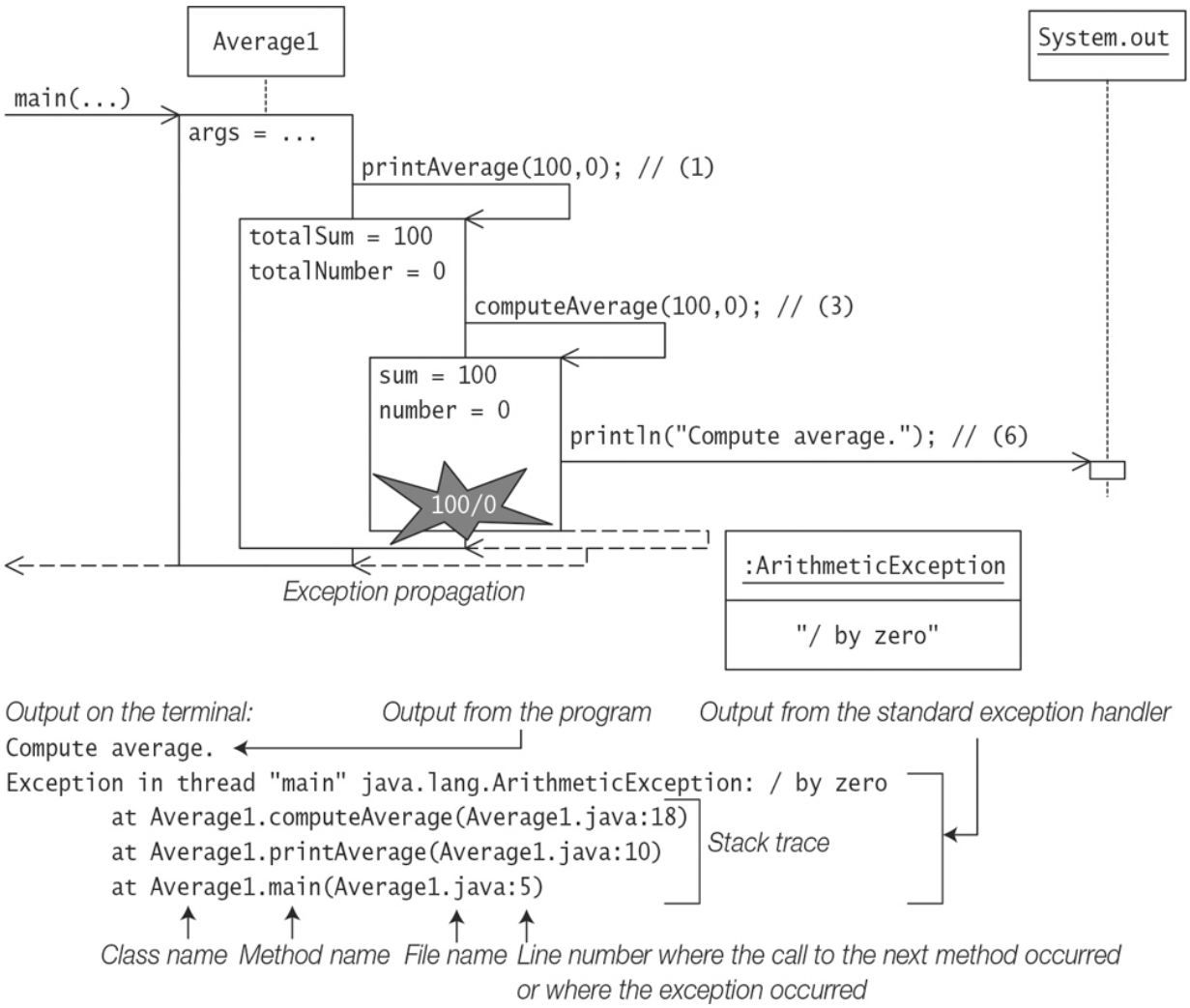



**Figure 7.2** *Exception Propagation*

If an exception is thrown during the evaluation of the left-hand operand of a binary expression, then the right-hand operand is not evaluated. Similarly, if an exception is thrown during the evaluation of a list of expressions (e.g., a list of actual parameters in a method call), evaluation of the rest of the list is skipped.

If the line numbers in the stack trace are not printed in the output as shown previously, use the following command to run the program:

**Click here to view code image**

```
>java -Djava.compiler=NONE Average1
```

## 7.2 Exception Types

Exceptions in Java are objects. All exceptions are derived from the `java.lang.Throwable` class. **Figure 7.3** shows a partial hierarchy of classes derived from the `Throwable` class. The two main subclasses `Exception` and `Error` constitute the main categories of *throwables*, the term used to refer to both exceptions and errors. **Figure 7.3** also shows that not all exception classes are found in the `java.lang` package.

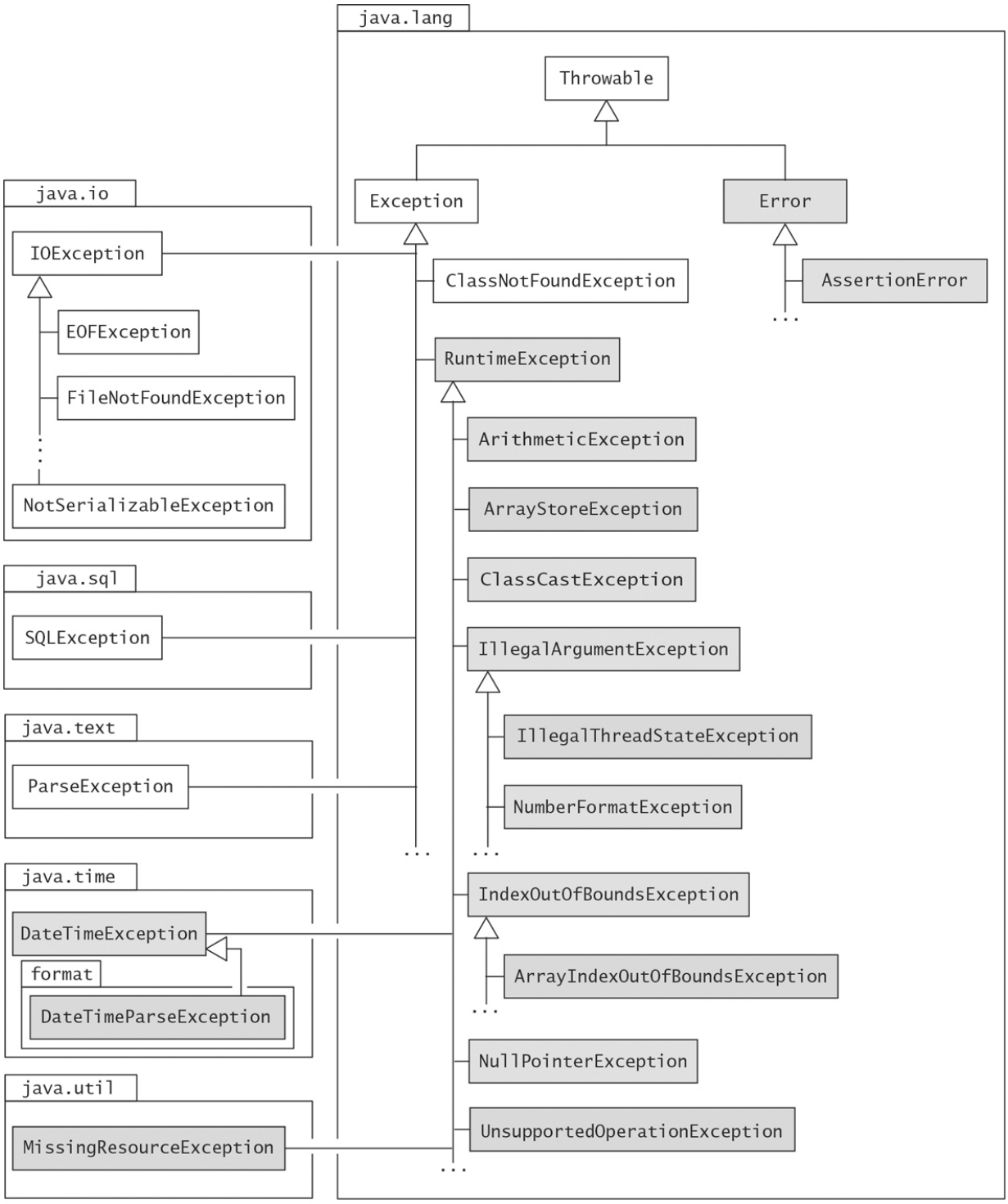



**Figure 7.3** *Partial Exception Inheritance Hierarchy*

All throwable classes in the Java SE Platform API at least define a zero-argument constructor and a one-argument constructor that takes a `String` parameter. This parameter can be set to provide a *detail message* when an exception is constructed. The purpose of the detail message is to provide more information about the actual exception.

---

```
Throwable()
Throwable(String msg)
```

The first constructor constructs a throwable that has `null` as its detail message. The second constructor constructs a throwable that sets the specified string as its detail message.

Most exception types provide analogous constructors.

---

The class `Throwable` provides the following common methods to query an exception:

---

```
String getMessage()
```

Returns the detail message.

```
void printStackTrace()
```

Prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the JVM stack when the exception was thrown. The stack trace can also be written to a `PrintStream` or a `PrintWriter` by supplying such a destination as an argument to one of the two overloaded `printStackTrace()` methods. Any *suppressed exceptions* associated with an exception on the stack trace are also printed (**p. 415**). It will also print the *cause* of an exception (which is also an exception) if one is available (**p. 405**).

```
String toString()
```

Returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the `getMessage()` method.

---

In dealing with throwables, it is important to recognize *situations* in which a particular throwable can occur, and the *source* that is responsible for throwing it. By *source* we mean:

- The *JVM* that is responsible for throwing the throwable, or
- The throwable that is explicitly thrown *programmatically* by the code in the application or by any API used by the application.

In further discussion of exception types, we provide an overview of situations in which selected throwables can occur and the source responsible for throwing them.

### The `java.lang.Exception` Class

The class `Exception` represents exceptions that a program would normally want to catch. Its subclass `java.lang.RuntimeException` represents many common programming errors that can manifest at runtime (see the next subsection). Other subclasses of the `Exception` class, excluding the `RuntimeException` class, define what are known as *checked exceptions* (**p. 374**) that particularly aid in building robust programs. Some common checked exceptions are presented below.

**Click here to view code image**

`java.lang.ClassNotFoundException`

The class `ClassNotFoundException` is a subclass of the `Exception` class that signals that the JVM tried to load a class by its string name, but the class could not be found. A typical example of this situation is when the class name is misspelled while starting program execution with the `java` command. The source in this case is the JVM throwing the exception to signal that the class cannot be found.

`java.io.IOException`

The class `IOException` is a subclass of the `Exception` class that represents I/O-related exceptions that are found in the `java.io` package (`EOFException, FileNotFoundException`, `NotSerializableException`). **Chapter 20**, **p. 1231**, and **Chapter 21**, **p. 1285**, provide ample examples of contexts in which I/O-related exceptions can occur.

`java.io.EOFException`

The class `EOFException` is a subclass of the `IOException` class that represents an exception that signals that an *end of file* (EOF) or end of stream was reached unexpectedly when more input was expected—that is, there is no more input available. Typically, this situation occurs when an attempt is made to read input from a file when all data from the file has already been read, often referred to as *reading past EOF*.

**Click here to view code image**

`java.io.FileNotFoundException`

The class `FileNotFoundException` is a subclass of the `IOException` class that represents an exception that signals an attempt to open a file by using a specific pathname failed—in other words, the file with the specified pathname does not exist. This exception can also occur when an I/O operation does not have the required permissions for accessing the file.

Click here to view code image

`java.io.NotSerializableException`

The class `NotSerializableException` is a subclass of the `IOException` class that represents an exception that signals that an object does not implement the `Serializable` interface that is required in order for the object to be serialized (§20.5, p. 1261).

`java.sql.SQLException`

The class `SQLException` is a subclass of the `Exception` class that represents an exception that can provide information about various database-related errors that can occur. Chapter 24 provides examples illustrating such situations.

`java.text.ParseException`

The class `ParseException` is a subclass of the `Exception` class that represents an exception that signals unexpected errors while parsing. Examples of parsing date, number, and currency where this exception is thrown can be found in §18.5, p. 1116.

### The `java.lang.RuntimeException` Class

Runtime exceptions are all subclasses of the `java.lang.RuntimeException` class, which is a subclass of the `Exception` class. As these runtime exceptions are usually caused by program bugs that should not occur in the first place, it is usually more appropriate to treat them as faults in the program design and let them be handled by the default exception handler.

Click here to view code image

`java.lang.ArithmeticException`

This exception represents situations where an illegal arithmetic operation is attempted, such as integer division by 0. It is typically thrown by the JVM.

Click here to view code image

`java.lang.ArrayIndexOutOfBoundsException`

Java provides runtime checking of the array index value, meaning out-of-bounds array indices. The subclass `ArrayIndexOutOfBoundsException` of the `RuntimeException` class represents exceptions thrown by the JVM that signal out-of-bound errors specifically for arrays—that is, an error in which an invalid index is used to access an element

in the array. The index value must satisfy the relation 0 ≤ *index value* < *length of the array* (§3.9, p. 120).

Click here to view code image

`java.lang.ArrayStoreException`

This exception is thrown by the JVM when an attempt is made to store an object of the wrong type into an array of objects. The array store check at runtime ensures that an object being stored in the array is assignment compatible with the element type of the array (§5.7, p. 260). To make the array store check feasible at runtime, the array retains information about its declared element type at runtime.

`java.lang.ClassCastException`

This exception is thrown by the JVM to signal that an attempt was made to cast a reference value to a type that was not legal, such as casting the reference value of an `Integer` object to the `Long` type (§5.11, p. 269).

Click here to view code image

```
java.lang.IllegalArgumentException
```

The class `IllegalArgumentException` represents exceptions thrown to signal that a method was called with an illegal or inappropriate argument. For example, the `ofPattern(String pattern)` method in the `java.time.format.DateTimeFormatter` class throws an `IllegalArgumentException` when the letter pattern passed as an argument is invalid (§18.6, p. 1134).

Click here to view code image

`java.lang.IllegalThreadStateException`

The class `IllegalThreadStateException` is a subclass of the `IllegalArgumentException` class. Certain operations on a thread can only be executed when the thread is in an appropriate state (§22.4, p. 1380). This exception is thrown when this is not the case. For example, the `start()` method of a `Thread` object throws this exception if the thread has already been started (§22.3, p. 1370).

Click here to view code image

`java.lang.NumberFormatException`

The class `NumberFormatException` is a subclass of the `IllegalArgumentException` class that is specialized to signal problems when converting a string to a numeric value if the format of the characters in the string is not appropriate for the conversion. This exception is thrown programmatically. The numeric wrapper classes all have methods that throw this exception when conversion from a string to a numeric value is not possible (§8.3, p. 434).

Click here to view code image

```
java.lang.NullPointerException
```

This exception is typically thrown by the JVM when an attempt is made to use the `null` value as a reference value to refer to an object. This might involve calling an instance method using a reference that has the `null` value, or accessing a field using a reference that has the `null` value.

This programming error has made this exception one of the most frequently thrown exceptions by the JVM. The error message issued provides helpful information as to *where* and *which* reference raised the exception. However, inclusion of variables names in the message can be a potential security risk.

Click here to view code image

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.toLowerCase()" because "msg" is null
    at StringMethods.main(StringMethods.java:162)
```

Click here to view code image

`java.lang.UnsupportedOperationException`

This exception is thrown programmatically to indicate that an operation invoked on an object is not supported. Typically, a class implements an interface, but chooses not to provide certain operations specified in the interface. Methods in the class corresponding to these operations throw this exception to indicate that an operation is not supported by the objects of the class.

The API documentation of the `java.util.Collection` interface (§15.1, p. 783) in the Java Collections Framework states that certain methods are *optional*, meaning that a concrete collection class need not provide support for such operations, and if any such operation is not supported, then the method should throw this exception.

`java.time.DateTimeException`

In the Date and Time API, the class `DateTimeException` represents exceptions that signal problems with creating, querying, and manipulating date-time objects (§17.2, p. 1027).

Click here to view code image

`java.time.format.DateTimeParseException`

In the Date and Time API, the class `DateTimeParseException` is a subclass of the `DateTimeException` class that represents an exception that signals unexpected errors when parsing date and time values (§18.6, p. 1127).

Click here to view code image

`java.util.MissingResourceException`

This exception is thrown programmatically, typically by the lookup methods of the `java.util.ResourceBundle` class (§18.3, p. 1104). Get methods on resource bundles typically throw this exception when no resource can be found for a given key. Static lookup methods in this class also throw this exception when no resource bundle can be found based on a specified base name for a resource bundle. Resource bundles are discussed in §18.3, p. 1102.

### The `java.lang.Error` Class

The class `Error` and its subclasses define errors that are invariably never explicitly caught and are usually irrecoverable. Not surprisingly, most such errors are signaled by the JVM. Apart from the subclass mentioned below, other subclasses of the `java.lang.Error` class define different categories of errors.

The subclass `VirtualMachineError` represents virtual machine errors like stack overflow (`StackOverflowError`) and out of memory for object allocation (`OutOfMemoryError`). The subclass `LinkageError` represents class linkage errors like missing class definitions (`NoClassDefFoundError`). The subclass `AssertionError` of the `Error` class is used by the Java assertion facility.

### Checked and Unchecked Exceptions

Except for `RuntimeException`, `Error`, and their subclasses, all exceptions are *checked* exceptions. A checked exception represents an unexpected event that is not under the control of the program—for example, a file was not found. The compiler ensures that if a method can throw a checked exception, directly or indirectly, the method must either

catch the exception and take the appropriate action, or pass the exception on to its caller (p. 388).

Exceptions defined by the `Error` and `RuntimeException` classes and their subclasses are known as *unchecked* exceptions, meaning that a method is not obliged to deal with these kinds of exceptions (shown with gray color in **Figure 7.3**). Either they are irrecoverable (exemplified by the `Error` class), in which case the program should not attempt to deal with them, or they are programming errors (exemplified by the `RuntimeException` class and its subclasses) and should usually be dealt with as such, and not as exceptions.

### Defining Customized Exceptions

Customized exceptions are usually defined to provide fine-grained categorization of error situations, instead of using existing exception classes with descriptive detail messages to differentiate among the various situations. New customized exceptions are usually defined by either extending the `Exception` class or one of its checked subclasses, thereby making the new exceptions checked, or extending the `RuntimeException` subclass or one of its subclasses to create new unchecked exceptions.

Customized exceptions, as any other Java classes, can declare fields, constructors, and methods, thereby providing more information as to their cause and remedy when they are thrown and caught. The `super()` call can be used in a constructor to set pertinent details about the exception: a detail message or the cause of the exception (p. 405), or both. Note that the exception class must be instantiated to create an exception object that can be thrown and subsequently caught and dealt with. The following code sketches a class declaration for an exception that can include all pertinent information about the exception. Typically, the new exception class provides a constructor to set the detail message.

Click here to view code image

```
public class EvacuateException extends Exception {
  // Fields
  private Date date;
  private Zone zone;
  private TransportMode transport;

  // Constructor
  public EvacuateException(Date d, Zone z, TransportMode t) {
    // Call the constructor of the superclass, usually passing a detail message.
    super("Evacuation of zone " + z);
    // ...
```

```
    }
    // Methods
    // ...
}
```

Several examples in subsequent sections illustrate exception handling.

### 7.3 Exception Handling: `try`, `catch`, and `finally`

The mechanism for handling exceptions is embedded in the `try`-`catch`-`finally` construct, which has the following basic form:

Click here to view code image

```
try {                                       // try block
  statements
} catch (exception_type1 parameter1) {        // uni-catch clause
  statements
}
...
  catch (exception_typen parameter_n) {       // uni-catch clause
  statements

} finally {                                 // finally clause
  statements
}
```

A few aspects about the syntax of this construct should be noted. For each `try` block, there can be zero or more `catch` clauses (i.e., it can have *multiple* `catch` clauses), but only one `finally` clause. The `catch` clauses and the `finally` clause must always appear in conjunction with a `try` block, and in the right order. A `try` block must be followed by at least one `catch` clause, or a `finally` clause must be specified—in contrast to the `try`-with-resources statement where neither a `catch` nor a `finally` clause is mandatory (p. 407). In addition to the `try` block, each `catch` clause and the `finally` clause specify a block, `{ }`. The block notation is mandatory.

Exceptions thrown during execution of the `try` block can be caught and handled in a `catch` clause. Each `catch` clause defines an exception handler. The header of the `catch` clause specifies exactly one exception parameter. The exception type must be of the `Throwable` class or one of its subclasses; otherwise, the code will not compile. The type of the exception parameter of a `catch` clause is specified by a *single* exception type in the syntax shown earlier, and such a `catch` clause is called a *uni-*`catch` clause.

A `finally` clause is always executed, regardless of the cause of exit from the `try` block, or whether any `catch` clause was executed at all. The two exceptions to this scenario are if the JVM crashes or the `System.exit()` method is called. **Figure 7.4** shows three typical scenarios of control flow through the `try-catch-finally` construct.

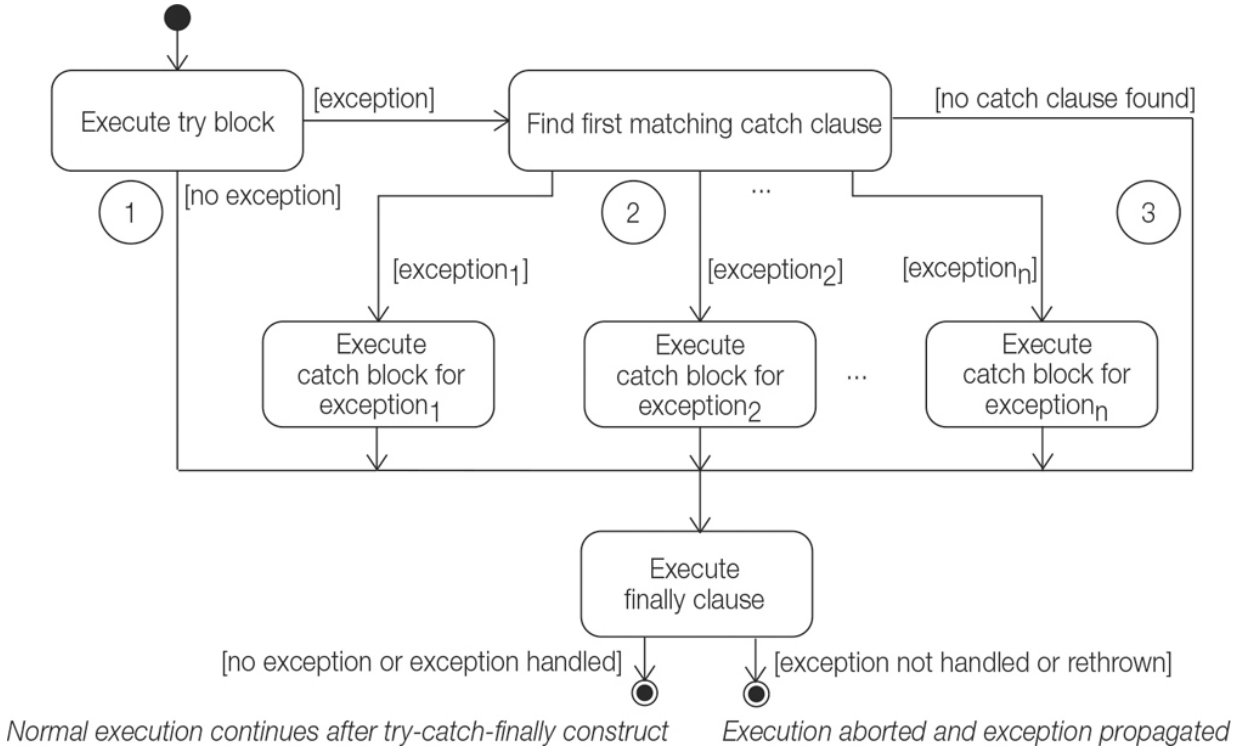



**Figure 7.4** *The* `try-catch-finally` *Construct*

The `try` block, the `catch` clause, and the `finally` clause of a `try-catch-finally` construct can contain arbitrary code, which means that a `try-catch-finally` construct can be nested in any block of the `try-catch-finally` construct. However, such nesting can easily make the code difficult to read and is best avoided, if possible.

### The `try` Block

The `try` block establishes a context for exception handling. Termination of a `try` block occurs as a result of encountering an exception, or from successful execution of the code in the `try` block.

The `catch` clauses are skipped for all normal exits from the `try` block when no exceptions are thrown, and control is transferred to the `finally` clause if one is specified (Scenario 1 in **Figure 7.4**).

For all exits from the `try` block resulting from exceptions, control is transferred to the `catch` clauses—if any such clauses are specified—to find a matching `catch` clause (Scenario 2 in **Figure 7.4**). In the case when no `catch` clause is specified, control is transferred to the mandatory `finally` clause. If no `catch` clause matches the thrown exception, control is transferred to the `finally` clause if one is specified (Scenario 3 in **Figure 7.4**).

### The `catch` Clause

Only an exit from a `try` block resulting from an exception can transfer control to a `catch` clause. A `catch` clause can catch the thrown exception only if the exception is assignable to the parameter in the `catch` clause. The code of the first such `catch` clause is executed, and all other `catch` clauses are ignored.

On exit from a `catch` clause, normal execution continues unless there is any uncaught exception that has been thrown and not handled. If this is the case, the method is aborted after the execution of any `finally` clause and the exception propagated up the JVM stack.

It is important to note that after a `catch` clause has been executed, control is always transferred to the `finally` clause if one is specified. This is true as long as there is a `finally` clause, regardless of whether the `catch` clause itself throws an exception.

In **Example 7.2**, the method `printAverage()` calls the method `computeAverage()` in a `try`-`catch` construct at (4). The `catch` clause is declared to catch exceptions of type `ArithmeticException`. The `catch` clause handles the exception by printing the stack trace and some additional information at (7) and (8), respectively. Normal execution of the program is illustrated in **Figure 7.5**, which shows that the `try` block is executed but no exceptions are thrown, with normal execution continuing after the `try`-`catch` construct. This corresponds to Scenario 1 in **Figure 7.4**.

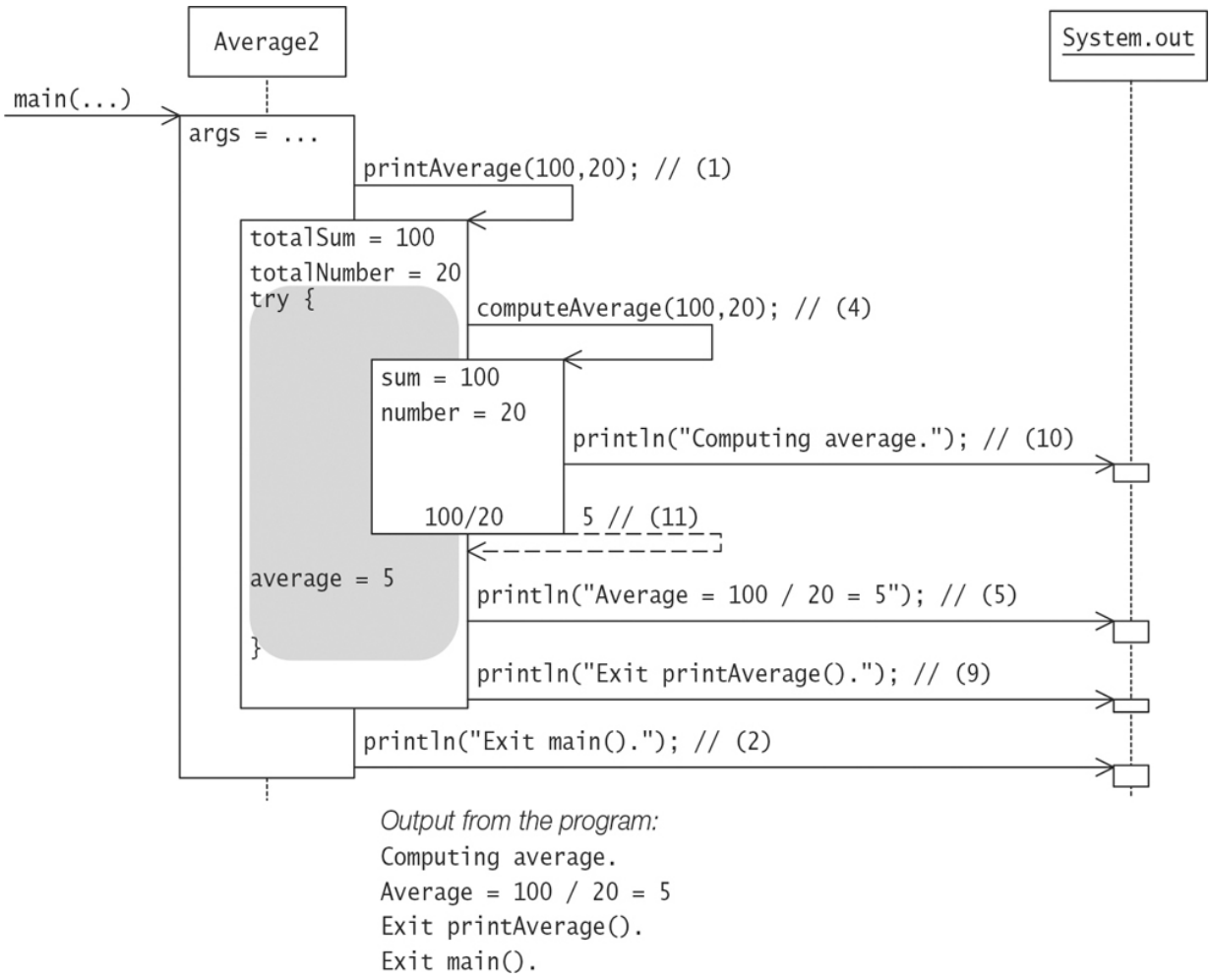



**Figure 7.5** *Exception Handling (Scenario 1)*

### Example 7.2 *The* `try-catch` *Construct*

Click here to view code image

```
public class Average2 {

  public static void main(String[] args) {
    printAverage(100, 20);                                          // (1)
    System.out.println("Exit main().");                             // (2)
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                           // (3)
      int average = computeAverage(totalSum, totalCount);           // (4)
      System.out.println("Average = " +                             // (5)
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                              // (6)
      ae.printStackTrace();                                         // (7)
      System.out.println("Exception handled in printAverage().");   // (8)
    }
    System.out.println("Exit printAverage().");                     // (9)
  }
```

```
    public static int computeAverage(int sum, int count) {
      System.out.println("Computing average.");                    // (10)
      return sum/count;                                            // (11)
    }
  }
```

Output from the program, with call `printAverage(100, 20)` at (1):

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Output from the program, with call `printAverage(100, 0)` at (1):

**Click here to view code image**

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:23)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

However, if we run the program in **Example 7.2** with the following call at (1):

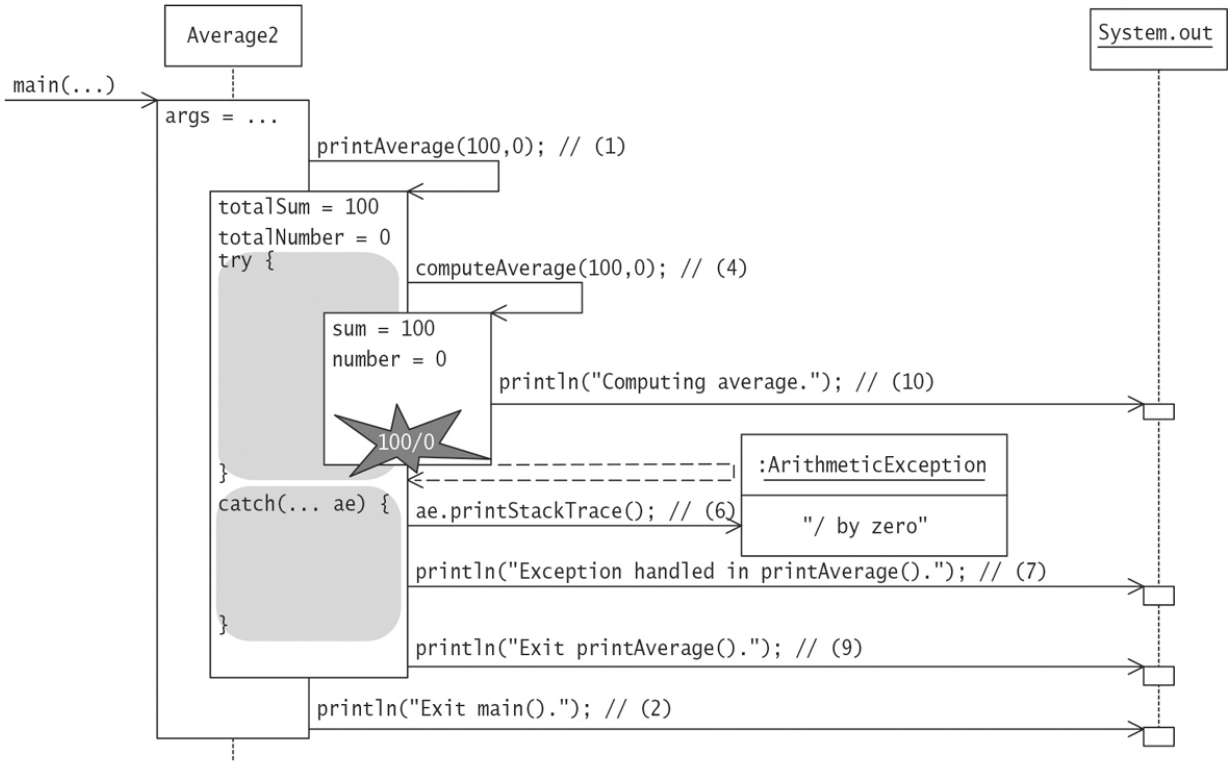
```
printAverage(100, 0)
```

an `ArithmeticException` is thrown by the integer division operator in the method `computeAverage()`. In **Figure 7.6** we see that the execution of the method `compute-Average()` is stopped and the exception propagated to method `printAverage()`, where it is handled by the `catch` clause at (6). Normal execution of the method continues at (9) after the `try-catch` construct, as witnessed by the output from the statements at (9) and (2). This corresponds to Scenario 2 in **Figure 7.4**.

In **Example 7.3**, the `main()` method calls the `printAverage()` method in a `try-catch` construct at (1). The `catch` clause at (3) is declared to catch exceptions of type `ArithmeticException`. The `printAverage()` method calls the `computeAverage()` method in a `try-catch` construct at (7), but here the `catch` clause is declared to catch exceptions of type `IllegalArgumentException`. Execution of the program is illustrated in **Figure 7.7**, which shows that the `ArithmeticException` is first propagated to the

`catch` clause in the `printAverage()` method. Because this `catch` clause cannot handle this exception, it is propagated further to the `catch` clause in the `main()` method, where it is caught and handled. Normal execution continues at (6) after the exception is handled.

In **Example 7.3**, the execution of the try block at (7) in the printAverage() method is never completed: The statement at (9) is never executed. The `catch` clause at (10) is skipped. The execution of the `printAverage()` method is aborted: The statement at (13) is never executed, and the exception is propagated. This corresponds to Scenario 3 in **Figure 7.4**.




*Output from the program:*

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:23)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

**Figure 7.6** *Exception Handling (Scenario 2)*

### Example 7.3 *Exception Propagation*

Click here to view code image

```
public class Average3 {

  public static void main(String[] args) {
    try {                                                    // (1)
      printAverage(100, 0);                                  // (2)
    } catch (ArithmeticException ae) {                       // (3)
```

```
        ae.printStackTrace();                                          // (4)
        System.out.println("Exception handled in main().");            // (5)
      }
      System.out.println("Exit main().");                              // (6)
    }

    public static void printAverage(int totalSum, int totalCount) {
      try {                                                            // (7)
        int average = computeAverage(totalSum, totalCount);            // (8)
        System.out.println("Average = " +                              // (9)

            totalSum + " / " + totalCount + " = " + average);
      } catch (IllegalArgumentException iae) {                         // (10)
        iae.printStackTrace();                                         // (11)
        System.out.println("Exception handled in printAverage().");    // (12)
      }
      System.out.println("Exit printAverage().");                      // (13)
    }

    public static int computeAverage(int sum, int count) {
      System.out.println("Computing average.");                        // (14)
      return sum/count;                                                // (15)
    }
}
```

Output from the program:

**Click here to view code image**

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:28)
        at Average3.printAverage(Average3.java:16)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

The scope of the exception parameter name in the `catch` clause is the body of the `catch` clause—that is, it is a local variable in the body of the `catch` clause. As mentioned earlier, the type of the exception object must be *assignable* to the type of the argument in the `catch` clause. In the body of the `catch` clause, the exception object can be queried like any other object by using the parameter name.

The `javac` compiler complains if a `catch` clause for a superclass exception shadows the `catch` clause for a subclass exception, as the `catch` clause of the subclass exception will never be executed (a situation known as *unreachable code*). The following ex-

ample shows incorrect order of the `catch` clauses at (1) and (2), which will result in a compile-time error at (2): The superclass `Exception` will shadow the subclass `ArithmeticException`.

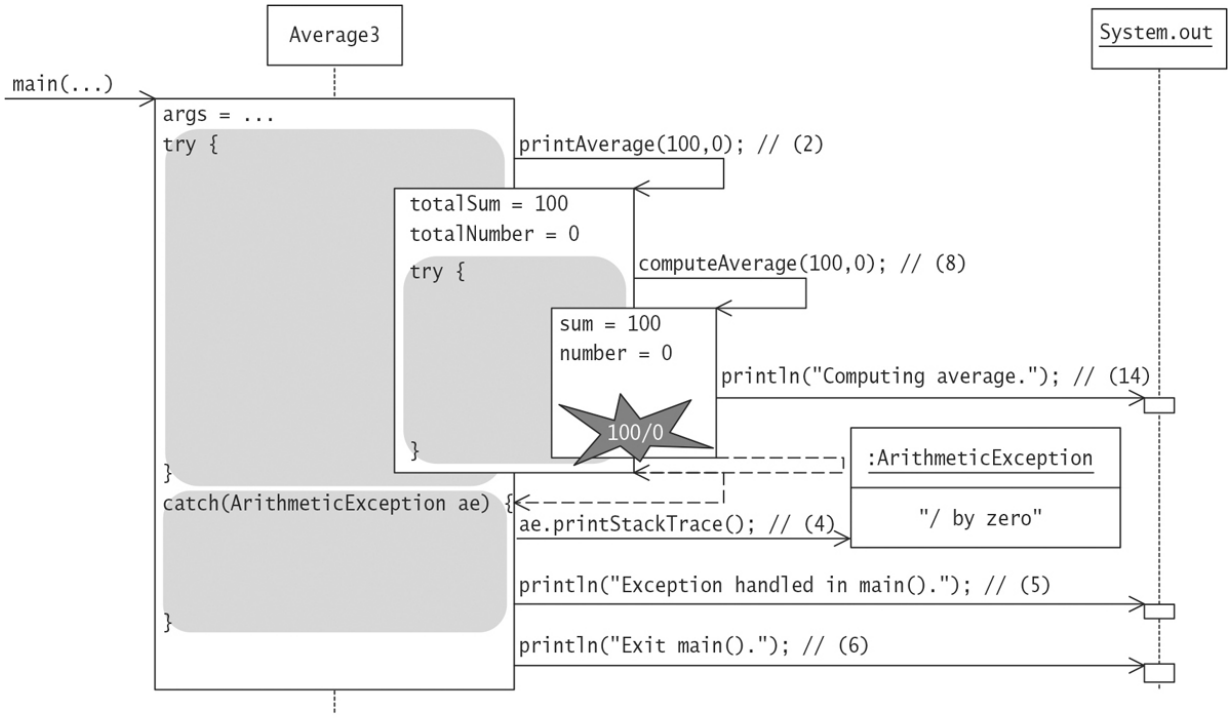
**Click here to view code image**

```
try {
  // ...
} catch (Exception e) {              // (1) Superclass shadows subclass
  System.out.println(e);
} catch (ArithmeticException e) {    // (2) Compile-time error: Unreachable code
  System.out.println(e);
}
```

The compiler will also flag an error if the parameter of the `catch` clause has a *checked* exception type that cannot be thrown by the `try` block, as this would result in unreachable code.

**Click here to view code image**

```
try {
  throw new ArithmeticException();      // IOException never thrown in try block
} catch (IOException e) {               // Compile-time error: Unreachable code
  System.out.println(e);
}
```




*Output from the program:*

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:28)
        at Average3.printAverage(Average3.java:16)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

**Figure** 7.7 *Exception Handling (Scenario 3)*

### The `finally` Clause

If the `try` block executes, then the `finally` clause is guaranteed to be executed, regardless of whether any `catch` clause was executed, barring the two special cases (JVM crashes or the `System.exit()` method is called). Since the `finally` clause is always executed before control transfers to its final destination, the `finally` clause can be used to specify any clean-up code (e.g., to free resources such as files and network connections). However, the `try`-with-resources statement provides a better solution for handling resources, and eliminates the use of the `finally` clause in many cases (p. 407).

A `try`-`finally` construct can be used to control the interplay between two actions that must be executed in the correct order, possibly with other intervening actions. In the code below, the operation in the `calculateAverage()` method (called at (2)) is dependent on the success of the `sumNumbers()` method (called at (1)). The `if` statement at (2) checks the value of the `sum` variable before calling the `calculateAverage()` method:

Click here to view code image

```
int sum = 0;
try {
```

```
    sum = sumNumbers();                    // (1)
    // other actions
  } finally {
    if (sum > 0) calculateAverage();       // (2)
  }
```

This code guarantees that if the `try` block is entered, the `sumNumbers()` method will be executed first, and later the `calculateAverage()` method will be executed in the `finally` clause, regardless of how execution proceeds in the `try` block. We can, if desired, include any `catch` clauses to handle any exceptions.

If the `finally` clause neither throws an exception nor executes a control transfer statement like a `return` or a labeled `break`, the execution of the `try` block or any `catch` clause determines how execution proceeds after the `finally` clause (**Figure 7.4**, **p. 376**).

- If no exception is thrown during execution of the `try` block or the exception has been handled in a `catch` clause, normal execution continues after the `finally` clause.
- If there is any uncaught exception (either because no matching `catch` clause was found or because the `catch` clause threw an exception), the method completes abruptly and the exception is propagated after the execution of the `finally` clause.

The output of **Example 7.4** shows that the `finally` clause at (4) is executed, regardless of whether an exception is thrown in the `try` block at (2). If an `ArithmeticException` is thrown, it is caught and handled by the `catch` clause at (3). After the execution of the `finally` clause at (4), normal execution continues at (5).

**Example 7.4** *The* `try-catch-finally` *Construct*

**Click here to view code image**

```
public class Average4 {

  public static void main(String[] args) {
    printAverage(100, 20);                                    // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                     // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                        // (3)
```

```
        ae.printStackTrace();
        System.out.println("Exception handled in printAverage().");
      } finally {                                                    // (4)
        System.out.println("Finally done.");
      }
      System.out.println("Exit printAverage().");                   // (5)
    }

    public static int computeAverage(int sum, int count) {
      System.out.println("Computing average.");
      return sum/count;
    }
  }
```

Output from the program, with the call `printAverage(100, 20)` at (1):

**Click here to view code image**

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Exit printAverage().
Exit main().
```

Output from the program, with the call `printAverage(100, 0)` at (1):

**Click here to view code image**

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average4.computeAverage(Average4.java:24)
        at Average4.printAverage(Average4.java:10)
        at Average4.main(Average4.java:4)
Exception handled in printAverage().
Finally done.
Exit printAverage().
Exit main().
```

On exiting from the `finally` clause, if there is any uncaught exception, the method completes abruptly and the exception is propagated as explained earlier. This is illustrated in **Example 7.5**. The method `printAverage()` is aborted after the `finally` clause at (3) has been executed, as the `ArithmeticException` thrown at (4) is not caught by any method. In this case, the exception is handled by the default exception handler. Notice the difference in the output from **Example 7.4** and **Example 7.5**.

**Example 7.5** *The* `try-finally` *Construct*

Click here to view code image

```
public class Average5 {

  public static void main(String[] args) {
    printAverage(100, 0);                                          // (1)
    System.out.println("Exit main().");
  }

  public static void printAverage(int totalSum, int totalCount) {
    try {                                                          // (2)
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } finally {                                                    // (3)
      System.out.println("Finally done.");
    }
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;                                              // (4)
  }
}
```

Output from the program:

Click here to view code image

```
Computing average.
Finally done.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average5.computeAverage(Average5.java:21)
        at Average5.printAverage(Average5.java:10)
        at Average5.main(Average5.java:4)
```

If the `finally` clause executes a control transfer statement, such as a `return` or a labeled `break`, this control transfer statement determines how the execution will proceed—regardless of how the `try` block or any `catch` clause was executed. In particular, a value returned by a `return` statement in the `finally` clause will supercede any value returned by a `return` statement in the `try` block or a `catch` clause.

Example 7.6 shows how the execution of a control transfer statement such as a `return` in the `finally` clause affects the program execution. The first output from the program shows that the average is computed but the value returned is from the `return` statement at (3) in the `finally` clause, not from the `return` statement at (2) in the `try` block. The second output shows that the `ArithmeticException` thrown in the `computeAverage()` method and propagated to the `printAverage()` method is suppressed by the `return` statement in the `finally` clause. Normal execution continues after the `return` statement at (3), with the value `0` being returned from the `printAverage()` method.

If the `finally` clause throws an exception, this exception is propagated with all its ramifications—regardless of how the `try` block or any `catch` clause was executed. In particular, the new exception overrules any previously uncaught exception (p. 415).

**Example 7.6** *The* `finally` *Clause and the* `return` *Statement*

Click here to view code image

```
public class Average6 {

  public static void main(String[] args) {
    System.out.println("Value: " + printAverage(100, 20));          // (1)
    System.out.println("Exit main().");
  }

  public static int printAverage(int totalSum, int totalCount) {
    int average = 0;
    try {
      average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
      return average;                                               // (2)
    } finally {
      System.out.println("Finally done.");
      return average*2;                                             // (3)
    }
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    return sum/count;
  }
}
```

Output from the program, with call `printAverage(100, 20)` at (1):

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Value: 10
Exit main().
```

Output from the program, with call `printAverage(100, 0)` at (1):

```
Computing average.
Finally done.
Value: 0
Exit main().
```

## 7.4 The `throw` Statement

Earlier examples in this chapter have shown how an exception can be thrown implicitly by the JVM during execution. Now we look at how an application can programmatically throw an exception using the `throw` statement. This statement can be used in a method, a constructor, or an initializer block. The general format of the `throw` statement is as follows:

Click here to view code image

```
throw object_reference_expression;
```

The compiler ensures that the type of the *object reference expression* is a `Throwable` or one of its subclasses. This ensures that a `Throwable` will always be propagated. At runtime a `NullPointerException` is thrown by the JVM if the *object reference expression* evaluates to `null`.

A detail message is often passed to the constructor when the exception object is created.

Click here to view code image

```
throw new ArithmeticException("Integer division by 0");
```

Propagation of a programmatically thrown exception is no different from one thrown implicitly by the JVM. When an exception is thrown, normal execution is suspended. The JVM proceeds to find a `catch` clause that can handle the exception.

The search starts in the context of the current `try` block, propagating to any enclosing `try` blocks and through the JVM stack to find a handler for the exception. Any associated `finally` clause of a `try` block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level. If a handler is found, normal execution resumes after the code in its `catch` clause has been executed, barring any rethrowing of an exception.

In **Example 7.7**, an exception is thrown by the `throw` statement at (4) as the `count` value is `0`. This exception is propagated to the `printAverage()` method, where it is caught and handled by the `catch` clause at (1). Note that the `finally` clause at (2) is executed, followed by the resumption of normal execution, as evident from the output of the print statement at (3).

**Example** 7.7 *Throwing Exceptions Programmatically*

**Click here to view code image**

```
public class Average7 {

  public static void main(String[] args) {
    printAverage(100, 0);            // Calling with 0 number of values.
  }

  public static void printAverage(int totalSum, int totalCount) {
    System.out.println("Entering printAverage().");
    try {
      int average = computeAverage(totalSum, totalCount);
      System.out.println("Average = " +
          totalSum + " / " + totalCount + " = " + average);
    } catch (ArithmeticException ae) {                              // (1)
      ae.printStackTrace();
      System.out.println("Exception handled in printAverage().");
    } finally {                                                     // (2)
      System.out.println("Finally in printAverage().");
    }
    System.out.println("Exit printAverage().");                     // (3)
  }

  public static int computeAverage(int sum, int count) {
    System.out.println("Computing average.");
    if (count == 0)
      throw new ArithmeticException("Integer division by 0");       // (4)
    return sum/count;
  }
}
```

Output from the program:

Click here to view code image

```
Entering printAverage().
Computing average.
java.lang.ArithmeticException: Integer division by 0
      at Average7.computeAverage(Average7.java:26)
      at Average7.printAverage(Average7.java:11)
      at Average7.main(Average7.java:5)
Exception handled in printAverage().
Finally in printAverage().
Exit printAverage().
```

## 7.5 The `throws` Clause

A `throws` clause can be specified in a method or a constructor header to declare any checked exceptions that can be thrown by a statement in the body of a method or a constructor. It is declared immediately preceding the body of the method or the constructor.

Click here to view code image

```
... throws ExceptionType1, ExceptionType2,..., ExceptionTypen { /* Body */ }
```

Each $ExceptionType_i$ is an *exception type* (i.e., a `Throwable` or one of its subclasses), although usually only checked exceptions are specified. The compiler enforces that if a checked exception can be thrown from the body of the method or the constructor, then either the type of this exception or a supertype of its exception type is specified in the `throws` clause of the method or the constructor. The `throws` clause can specify unchecked exceptions, but this is seldom done and the compiler does not enforce any restrictions on their usage.

The `throws` clause is part of the contract that a method or a constructor offers to its clients. The `throws` clause can specify any number of exception types in any order, even those that are not thrown by the method or the constructor. The compiler simply ensures that any checked exception that can actually be thrown in the method or constructor body is covered by the `throws` clause. Of course, any caller of the method or constructor cannot ignore the checked exceptions specified in the `throws` clause.

In a method or a constructor, a checked exception can be thrown directly by a `throw` statement, or indirectly by calling other methods or constructors that can throw a checked exception. If a checked exception is thrown, the code must obey the following

rule (known by various names: *catch-or-declare rule, handle-or-declare rule, catch-or-specify requirement*):

- Either use a `try` block and catch the checked exception in a `catch` block and deal with it
- Or explicitly allow propagation of the checked exception to its caller by declaring it in the `throws` clause

Note that catching and dealing with a checked exception does not necessarily imply resumption of normal execution. A `catch` clause can catch the checked exception and choose to throw some other exception or even the same exception that is either unchecked or declared in the `throws` clause (**p. 401**). This rule ensures that a checked exception will be dealt with, regardless of the path of execution. This aids development of robust programs, as allowance can be made for many contingencies.

In **Example 7.8**, a new checked exception is defined, where the checked exception class `IntegerDivisionByZero` extends the `Exception` class. The method call at (2) in the `try` block at (1) results in the `printAverage()` method at (6) to be executed. The method call at (7) results in the `computeAverage()` method at (8) to be executed.

In the `if` statement at (9), the method `computeAverage()` throws the checked exception `IntegerDivisionByZero`. Neither the `computeAverage()` method nor the `printAverage()` method catches the exception, but instead throws it to the caller, as declared in the `throws` clauses in their method headers at (6) and (8). The exception propagates to the `main()` method. Since the `printAverage()` method was called from the context of the `try` block at (1) in the `main()` method, the exception is successfully caught by its `catch` clause at (3). The exception is handled and the `finally` clause at (4) is executed, with normal execution resuming from (5). If the method `main()` did not catch the exception, it would have to declare this exception in a `throws` clause. In that case, the exception would end up being handled by the default exception handler.

**Example 7.8** *The* `throws` *Clause*

**Click here to view code image**

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

**Click here to view code image**

```
// File: Average8.java
public class Average8 {
  public static void main(String[] args) {
    try {                                                          // (1)
      printAverage(100, 0);                                        // (2)
    } catch (IntegerDivisionByZero idbz) {                         // (3)
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                    // (4)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                            // (5)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {                               // (6)
    int average = computeAverage(totalSum, totalCount);            // (7)
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {                               // (8)
    System.out.println("Computing average.");
    if (count == 0)                                                // (9)
      throw new IntegerDivisionByZero();
    return sum/count;                                              // (10)
  }
}
```

Output from the program:

Click here to view code image

```
Computing average.
IntegerDivisionByZero: Integer Division By Zero
        at Average8.computeAverage(Average8.java:27)
        at Average8.printAverage(Average8.java:17)
        at Average8.main(Average8.java:5)
Exception handled in main().
Finally done in main().
Exit main().
```

As mentioned earlier, the exception type specified in the `throws` clause can be a super-class of the actual exceptions thrown—that is, the exceptions thrown must be assigna-

ble to the type of the exceptions specified in the `throws` clause. If a method or a constructor can throw a checked exception, then the `throws` clause must declare its exception type or a supertype of its exception type; otherwise, a compile-time error will occur. In the `printAverage()` method, the superclass `Exception` of the subclass `IntegerDivisionByZero` could be specified in the `throws` clause of the method. This would also entail that the `main()` method either catch an `Exception` or declare it in a `throws` clause.

**Click here to view code image**

```
public static void main(String[] args) throws Exception {
  /* ... */
}
public static void printAverage(int totalSum, int totalCount) throws Exception {
  /* ... */
}
```

It is generally considered bad programming style to specify exception superclasses in the `throws` clause when the actual exceptions thrown are instances of their subclasses. It is also recommended to use the `@throws` tag in a Javadoc comment to document the checked exceptions that a method or a constructor can throw, together with any unchecked exceptions that might also be relevant to catch.

### Overriding the `throws` Clause

A subclass can *override* a method defined in its superclass by providing a new implementation (§5.1, p. 196). What happens when a superclass method with a list of exceptions in its `throws` clause is overridden in a subclass? The method declaration in the subclass need not specify a `throws` clause if it does not throw any checked exceptions, and if it does, it can specify only *checked* exception classes that are already in the `throws` clause of the superclass method, or that are subclasses of the checked exceptions in the `throws` clause of the superclass method. As a consequence, an overriding method can have more number of exceptions, but it *cannot* allow *broader* checked exceptions in its `throws` clause than the superclass method does. Allowing broader checked exceptions in the overriding method would create problems for clients who already deal with the exceptions specified in the superclass method. Such clients would be ill prepared if an object of the subclass threw a checked exception they were not prepared for. However, there are no restrictions on specifying *unchecked* exceptions in the `throws` clause of the overriding method. The preceding discussion also applies to overriding methods from an interface that a class implements.

In the code below, the method `compute()` at (1) in superclass `A` is overridden correctly at (2) in subclass `B`. The `throws` clause of the method at (2) in subclass `B` specifies

only one checked exception ( `ThirdException` ) from the `throws` clause at (1) and adds the more specific subclass exception ( `SubFirstException` ) of the superclass exception ( `FirstException` ) that is specified in the `throws` clause at (1). An unchecked exception ( `NumberFormatException` ) is also specified in the throws clause at (2). The checked exceptions in the `throws` clause at (2) are covered by the checked exceptions specified in the `throws` clause at (1). Unchecked exceptions are inconsequential in this regard. The subclass `C` does not override the `compute()` method from class `A` correctly, as the `throws` clause at (3) specifies an exception ( `FourthException` ) that the overridden method at (1) in class `A` cannot handle.

**Click here to view code image**

```
// New exception classes:
class FirstException    extends Exception { }
class SecondException   extends Exception { }
class ThirdException    extends Exception { }
class FourthException   extends Exception { }
class SubFirstException extends FirstException { }

// Superclass
class A {
  protected void compute()
      throws FirstException, SecondException, ThirdException { /* ... */ }  // (1)
}

// Subclass
class B extends A {
  @Override
  protected void compute()
      throws ThirdException, SubFirstException, NumberFormatException {     // (2)
    /* ... */
  }
}
//Subclass
class C extends A {
  @Override
  protected void compute()                              // Compile-time error at (3)
      throws FirstException, ThirdException, FourthException { /* ... */ }  // (3)
}
```

Usage of checked and unchecked exceptions in different contexts is compared in **Table 7.1**.

**Table 7.1** *Comparing Checked and Unchecked Exceptions*

| Context | Checked exceptions | Unchecked exceptions |
|---|---|---|
| The `throws` clause | Can include any checked exception, but when overridden it cannot specify new checked exceptions. | Can include any unchecked exception, whether it is overridden or not. |
| The `try` block | Can throw any checked exception. | Can throw any unchecked exception. |
| The `catch` clause | Can only catch a checked exception that is thrown by the `try` block. A `catch` clause can always be used to catch an `Exception`. | Can catch any unchecked exception, whether or not it is thrown by the `try` block. |
| The `throw` statement | Can throw any checked exception. | Can throw any unchecked exception. |
| The *catch-or-declare* rule | Applies only to checked exceptions. | Does not apply to unchecked exceptions. |

## Review Questions

7.1 Which digits, and in which order, will be printed when the following program is run?

Click here to view code image

```
public class DemoClass {
  public static void main(String[] args) {
    int k=0;
    try {
      int i = 5/k;
    } catch (ArithmeticException e) {
      System.out.println("1");
    } catch (RuntimeException e) {
      System.out.println("2");
      return;
    } catch (Exception e) {
```

```
      System.out.println("3");
    } finally {
      System.out.println("4");
    }
    System.out.println("5");
  }
}
```

Select the one correct answer.

**a.** The program will only print `5`.

**b.** The program will only print `1` and `4`, in that order.

**c.** The program will only print `1`, `2`, and `4`, in that order.

**d.** The program will only print `1`, `4`, and `5`, in that order.

**e.** The program will only print `1`, `2`, `4`, and `5`, in that order.

**f.** The program will only print `3` and `5`, in that order.

7.2 Which of the following statements are true about the following program?

**Click here to view code image**

```
public class Exceptions {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    } finally {
      System.out.println("The end");
    }
  }
}
```

Select the two correct answers.

**a.** If run with no program arguments, the program will produce no output.

**b.** If run with no program arguments, the program will print `The end`.

**c.** The program will throw an `ArrayIndexOutOfBoundsException`.

**d.** If run with one program argument, the program will simply print the specified argument.

**e.** If run with one program argument, the program will print the specified argument followed by `The end`.

7.3 Which of the following statements are true? Select the two correct answers.

**a.** If an exception is not caught in a method, the method will terminate and normal execution will resume.

**b.** An overriding method must declare that it throws the same exception classes as the method it overrides.

**c.** The `main()` method of a program can declare that it throws checked exceptions.

**d.** A method declaring that it throws an exception of a certain class may throw instances of any subclass of that exception class.

**e.** The `finally` clause is executed if, and only if, an exception is thrown in the corresponding `try` block.

7.4 Which statement is true about the following code?

Click here to view code image

```
class A extends Throwable {}

class B extends A {}

public class RQ6A20 {
  public static void main(String[] args) throws A {
    try {
      action();
    } finally {
      System.out.println("Done.");
    } catch (A e) {

      throw e;
    }
  }

  public static void action() throws B {
    throw new B();
  }
}
```

Select the one correct answer.

**a.** The `main()` method must declare that it throws `B`.

**b.** The `finally` clause must follow the `catch` clause in the `main()` method.

**c.** The `catch` clause in the `main()` method must declare that it catches `B` rather than `A`.

**d.** A single `try` block cannot be followed by both `catch` and `finally` clauses.

**e.** The declaration of class `A` is not valid.

7.5 Which statement is true about the following code?

Click here to view code image

```
public class Calculator {
  public static int average(int... values) {
    int result = 0;
    for (int i = 0; i < values.length; i++) {
      result += values[i];
    }
    return result/values.length;
  }
  public static void main(String[] args) {
    int value = 1;
    try {
      value = average();
    } catch (ArithmeticException e) {
      System.out.print("error");
    }
    System.out.print(value);
  }
}
```

Select the one correct answer.

**a.** The program will print `error0`.

**b.** The program will print `error1`.

**c.** The program will print `error`.

**d.** The program will print `0`.

**e.** The program will print `1`.

**f.** The program will terminate by throwing an exception.

**g.** The program will fail to compile.

**7.6** What will be the output of the following program?

**Click here to view code image**

```
public class PlayerException extends Exception {
  public PlayerException(String message) { super(message); }
}
```

**Click here to view code image**

```
public class Player {
  public static String reaction (String action)  throws PlayerException {
    if (action == null || action.isEmpty()) {
      throw new PlayerException("Invalid action");
    }
    return ">" + action;
  }
  public static void main(String[] args) {
    String message = null;
    try {
      message = reaction(message);
    } catch (PlayerException e) {
      message = e.getMessage();
    }
    System.out.print(message);
  }
}
```

Select the one correct answer.

**a.** `message`

**b.** `>message`

**c.** `>null`

**d.** `>`

**e.** `Invalid action`

**f.** `>Invalid action`

7.7 What will be the output of the following program?

**Click here to view code image**

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws Exception {

    if (name == null) throw new FileNotFoundException("invalid file name");
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

**a.** `IO error: invalid file name finally`

**b.** `IO error: invalid file name the end`

**c.** `IO error: invalid file name finally the end`

**d.** `Other error: invalid file name finally`

**e.** `Other error: invalid file name the end`

**f.** `Other error: invalid file name finally the end`

**g.** `finally the end`

**h.** `the end`

**i.** Nothing will be printed.

**j.** An uncaught exception stack trace will be printed.

7.8 What will be the output of the following program?

Click here to view code image

```
import java.io.*;
public class FileReader {
  public static void readFile(String name)  throws Exception {

    if (name != null) throw new FileNotFoundException("invalid file name");
  }
  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage()+" ");
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage()+" ");
    } finally {
      System.out.print("finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

**a.** `IO error: invalid file name finally`

**b.** `IO error: invalid file name the end`

**c.** `IO error: invalid file name finally the end`

**d.** `Other error: invalid file name finally`

**e.** `Other error: invalid file name the end`

**f.** `Other error: invalid file name finally the end`

**g.** `finally the end`

**h.** `the end`

**i.** Nothing will be printed.

**j.** An uncaught exception stack trace will be printed.

7.9 What will be the output of the following program?

Click here to view code image

```
import java.io.*;
public class FileReader {
  public static void readFile(String name) throws IOException {

    if (name == null) {
      throw new NullPointerException("invalid file name");
    } else {
      throw new IOException("file read not implemented");
    }
  }

  public static void main(String[] args) {
    String file = null;
    try {
      readFile(file);
    } catch (IOException e) {
      System.out.print("IO error: " + e.getMessage());
    } catch (Exception e) {
      System.out.print("Other error: " + e.getMessage());
      return;
    } finally {
      System.out.print(" finally");
    }
    System.out.print(" the end");
  }
}
```

Select the one correct answer.

**a.** `IO error: file read not implemented finally`

**b.** `IO error: file read not implemented the end`

**c.** `IO error: file read not implemented finally the end`

**d.** `Other error: invalid file name finally`

**e.** `Other error: invalid file name the end`

**f.** `Other error: invalid file name finally the end`

**g.** `finally the end`

**h.** `the end`

**i.** Nothing will be printed.

**j.** An uncaught exception stack trace will be printed.

### 7.6 The Multi-`catch` Clause

**Example 7.9** uses a `try` block that has multiple uni-`catch` clauses. This example is based on **Example 7.8**. The sum of the values and the number of values needed to calculate the average are now read as program arguments from the command line at (2) and (3), respectively. The example shows a `try` statement at (1) that uses three uni-`catch` clauses: at (5), (6), and (7). In a *uni-*`catch` clause, a *single* exception type is specified for the `catch` parameter.

In **Example 7.9**, the method `printAverage()` is only called at (4) if there are at least two consecutive integers specified on the command line. An unchecked `ArrayIndexOutOfBoundsException` is thrown if there are not enough program arguments, and an unchecked `NumberFormatException` is thrown if an argument cannot be converted to an `int` value. The astute reader will notice that the code for handling these two exceptions is the same in the body of the respective `catch` clauses. In order to avoid such code duplication, one might be tempted to replace the two `catch` clauses with a single `catch` clause that catches a more *general* exception, for example:

**Click here to view code image**

```
catch (RuntimeException rte) { // NOT RECOMMENDED!
  System.out.println(rte);
  System.out.println("Usage: java Average9 <sum of values> <no. of values>");
}
```

This is certainly not recommended, as specific exceptions are to be preferred over general exceptions, not the least because a more general exception type might unintentionally catch more exceptions than intended.

**Example 7.9** *Using Multiple* `catch` *Clauses*

**Click here to view code image**

```
// File: IntegerDivisionByZero.java
public class IntegerDivisionByZero extends Exception {
```

```
  IntegerDivisionByZero() { super("Integer Division by Zero"); }
}
```

Click here to view code image

```
// File: Average9.java
public class Average9 {
  public static void main(String[] args) {
    try {                                                    // (1)
      int sum         = Integer.parseInt(args[0]);           // (2)
      int numOfValues = Integer.parseInt(args[1]);           // (3)
      printAverage(sum, numOfValues);                        // (4)
    } catch (ArrayIndexOutOfBoundsException aioob) {         // (5) uni-catch
      System.out.println(aioob);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>");
    } catch (NumberFormatException nfe) {                    // (6) uni-catch
      System.out.println(nfe);
      System.out.println("Usage: java Average9 <sum of values> <no. of values>");
    } catch (IntegerDivisionByZero idbz) {                   // (7) uni-catch
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                              // (8)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                      // (9)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    int average = computeAverage(totalSum, totalCount);
    System.out.println("Average = " +
        totalSum + " / " + totalCount + " = " + average);
    System.out.println("Exit printAverage().");
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    System.out.println("Computing average.");
    if (count == 0)
      throw new IntegerDivisionByZero();
    return sum/count;
  }
}
```

Running the program:

Click here to view code image

```
>java Average9 100 twenty
java.lang.NumberFormatException: For input string: "twenty"
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

Running the program:

Click here to view code image

```
>java Average9 100
java.lang.ArrayIndexOutOfBoundsException: 1
Usage: java Average9 <sum of values> <no. of values>
Finally done in main().
Exit main().
```

The *multi-* `catch` clause provides the solution, allowing specific exceptions to be declared and avoiding duplicating the same code for the body of the `catch` clauses. The syntax of the multi- `catch` clause is as follows:

Click here to view code image

```
catch (exception_type1|exception_type2|...|exception_typek parameter) {
  statements
}
```

The multi- `catch` clause still has a single *parameter*, but now a list of exception types, delimited by the vertical bar ( `|` ), can be specified as the types for this parameter. This list defines a *union of alternatives* that are the exception types which the multi- `catch` clause can handle. The *statements* in the body of the multi- `catch` clause will be executed when an object of any of the specified exception types is caught by the multi- `catch` clause.

The multiple `catch` clauses at (5) and (6) in **Example 7.9** have been replaced with a multi- `catch` clause at (5) in **Example 7.10**:

Click here to view code image

```
catch (ArrayIndexOutOfBoundsException |                  // (5) multi-catch
       NumberFormatException ep) {
  System.out.println(ep);
  System.out.println("Usage: java Average10 <sum of values> <no. of values>");
}
```

The multi-`catch` clause in **Example 7.10** is semantically equivalent to the two uni-`catch` clauses in **Example 7.9**, and we can expect the same program behavior in both examples.

**Example 7.10** *Using the Multi-*`catch` *Clause*

Click here to view code image

```
// File: Average10.java
public class Average10 {
  public static void main(String[] args) {
    try {                                                       // (1)
      int sum = Integer.parseInt(args[0]);                      // (2)
      int numOfValues = Integer.parseInt(args[1]);              // (3)
      printAverage(sum, numOfValues);                           // (4)
    } catch (ArrayIndexOutOfBoundsException |                   // (5) multi-catch
             NumberFormatException ep) {
      System.out.println(ep);
      System.out.println("Usage: java Average10 <sum of values> <no. of values>");
    } catch (IntegerDivisionByZero idbz) {                      // (6) uni-catch
      idbz.printStackTrace();
      System.out.println("Exception handled in main().");
    } finally {                                                 // (7)
      System.out.println("Finally done in main().");
    }
    System.out.println("Exit main().");                         // (8)
  }

  public static void printAverage(int totalSum, int totalCount)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }

  public static int computeAverage(int sum, int count)
      throws IntegerDivisionByZero {
    // See Example 7.9.
  }
}
```

A few remarks are in order regarding the alternatives of a multi-`catch` clause. There should be no subtype–supertype relationship between any of the specified exception types in the alternatives of a multi-`catch` clause. The following multi-`catch` clause will not compile, as `ArrayIndexOutOfBoundsException` is a subtype of `IndexOutOfBoundsException`:

Click here to view code image

```
catch (IndexOutOfBoundsException |                  // Compile-time error!
       ArrayIndexOutOfBoundsException e) {
  // ...
}
```

The parameter of a multi-`catch` clause is also considered to be *implicitly* `final`, and therefore cannot be assigned to in the body of the multi-`catch` clause. In a uni-`catch` clause, the parameter is considered to be *effectively* `final` if it does not occur on the left-hand side of an assignment in the body of the uni-`catch` clause.

**Click here to view code image**

```
try {
  // Assume appropriate code to throw the right exceptions.
} catch (NumberFormatException |
         IndexOutOfBoundsException e) {    // Parameter is final.
  e = new ArrayIndexOutOfBoundsException();// Compile-time error!
                                           // Cannot assign to final parameter e.
} catch (IntegerDivisionByZero idbz) {     // Parameter is effectively final.
  idbz.printStackTrace();
} catch (IOException ioe) {                // Parameter is not effectively final.
  ioe = new FileNotFoundException("No file.");
}
```

Disallowing any subtype–supertype relationship between alternatives and the parameter being `final` in a multi-`catch` clause or effectively `final` in a uni-`catch` clause allows the compiler to perform precise exception handling analysis.

The compiler also generates effective bytecode for a *single* exception handler corresponding to all the alternatives in a multi-`catch` clause, in contrast to generating bytecode for *multiple* exception handlers for uni-`catch` clauses that correspond to the multi-`catch` clause.

**Rethrowing Exceptions**

Rethrowing an exception refers to throwing an exception in the body of a `catch` clause. The `catch` clause catches an exception, but then throws this exception or another exception in its body. This allows an exception to be partially handled when it is caught the first time, and then again when the rethrown exception is caught later. Typically, the first exception handler is a common handler for the situation and the later exception handler is a more specific one.

Exception parameters that are explicitly, implicitly, or effectively `final` in `catch` clauses allow the compiler to perform improved analysis of exception handling in the

code, especially when it comes to rethrowing exceptions.

For the examples in this section, it is important to keep in mind that the exception type `IOException` is the supertype of both `EOFException` and `FileNotFoundException`.

**Example 7.11** illustrates how the compiler is able to identify unreachable code by precise analysis of rethrown exceptions that are either `final` or effectively `final`. The body of the `try` statement at (1) can only throw a `FileNotFoundException` that is caught by the `catch` clause at (3). This exception is effectively `final` in the `catch` clause at (3), as no assignment is made to it in the body of the `catch` clause. This exception is rethrown in the nested `try` statement at (4), but the `catch` clause at (6) of this `try` statement can only catch an `EOFException`. Since parameter `ex` is effectively `final`, it can only denote a `FileNotFoundException`, never an `EOFException`. The `catch` clause at (6) is unreachable, and the compiler flags an error.

**Example 7.11** *Precise Rethrowing of Exceptions*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionAnalysis {
  public static void main(String[] args) throws IOException {
    try {                                  // (1)
      throw new FileNotFoundException(); // (2)
    } catch (IOException ex) {             // (3)
      try {                          // (4) Nested try statement
        throw ex;                    // (5) Can only rethrow FileNotFoundException
      } catch (EOFException se) {  // (6) Compile-time error: clause unreachable
        System.out.println("I am unreachable.");
      }

    }
  }
}
```

**Example 7.12** illustrates how `final` or effectively `final catch` parameters allow more precise exceptions to be specified in the `throws` clause of a method. The thing to note is that the parameter `e` in the `catch` clause at (5) is not effectively `final`, as an assignment is made to the parameter at (6). All bets are off when the parameter is not `final` and the exception is rethrown. The `throws` clause must specify the same type as the type of the `catch` parameter, as shown at (3a). This has consequences for the

caller method `main()`. Its `try` statement at (1) must include the `catch` clause at (2) to catch an `IOException` as well, or the compiler will flag an error about an uncaught checked exception. The contract of the `checkIt()` method allows for all exceptions that are either `IOException` or its subtypes.

If the assignment statement at (6) is commented out, the `catch` parameter `e` is effectively `final` in the `catch` body. The compiler can deduce that the rethrown exception can only be of type `FileNotFoundException` or `EOFException`. The `throws` clause of the `checkIt()` method can be made more specific, as at (3b). Note that the type of the `catch` parameter is the supertype `IOException` of the subtypes specified in the `throws` clause at (3b). Commenting out the assignment statement at (6) and uncommenting the more precise `throws` clause at (3b) has consequences for the caller method `main()` as well. The `catch` clause at (2) becomes unreachable, and the compiler issues a *warning*. Note also that the type of the exception parameter `e` at (5) is `IOException`, which is the supertype of the exception types specified in the `throws` clause. However, static analysis by the compiler is able to confirm that the exception parameter `e` can only denote objects of either `FileNotFoundException` or `EOFException`, but not of supertype `IOException`.

**Example 7.12** *Precise* `throws` *Clause*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class MorePreciseRethrow {
  public static void main(String[] args) {                          // (1)
    try {
      checkIt(1);
    } catch (FileNotFoundException fnfe) {
      System.out.println("Check that the file exits.");
    } catch (EOFException eofe) {
      System.out.println("Check the contents of the file.");
    } catch (IOException ioe) { // (2) mandatory with (3a), but compiler warning
                                //     that clause is unreachable with (3b).
      System.out.println("This should never occur.");
    }
  }

  public static void checkIt(int value) throws IOException {       // (3a)
//public static void checkIt(int value)                            // (3b)
//    throws FileNotFoundException, EOFException {
    try {                                                           // (4)
```

```
      switch (value) {
        case 1:
          throw new FileNotFoundException("File not found");
        case 2:
          throw new EOFException("End of file");
        default:
          System.out.println("OK");
      }
    } catch (IOException e) {                                    // (5)
      System.out.println(e.getMessage());
      e = new EOFException("End of file");         // (6) not effectively final,
                                                   //     requires (3a).
                                                   //     When commented out,
                                                   //     can use (3b).
      throw e;
    }
  }
}
```

Program output with (3a) and (6) uncommented, and (3b) commented out:

**Click here to view code image**

```
File not found
Check the contents of the file.
```

Program output with (3a) and (6) commented out, and (3b) is uncommented:

**Click here to view code image**

```
File not found
Check that the file exits.
```

In summary, a `throw` statement in the body of a `catch` clause can throw a `final` or an effectively `final` exception parameter that has exception type `E` if *all* of the following conditions are satisfied:

- Exception type `E` can be thrown in the body of the `try` statement with which the `catch` clause is associated.
- Exception type `E` is assignment compatible with any of the exception types declared for the parameter in the `catch` clause.
- Exception type `E` is not assignment compatible with any of the exception types declared for the parameters in any preceding `catch` clause in the same `try` statement.

In **Example 7.13**, the `throw` statements at (1), (2), and (3) all try to rethrow an exception that is effectively `final` in the body of the `catch` clause.

- The `throw` statement at (1) in the `main()` method satisfies all the conditions. The `try` block throws an exception of the right type (`EOFException`). The exception thrown (`EOFException`) is assignment compatible with the type declared for the parameter in the `catch` clause (`IOException`). There is no preceding `catch` clause that handles the exception (`EOFException`).
- The `throw` statement at (2) in the `rethrowA()` method cannot throw the exception, as the first condition is not satisfied: The `try` block does not throw an exception of the right type (`EOFException`). The compiler flags an *error* for the `catch` clause that is unreachable.
- The `throw` statement at (3) in the `rethrowB()` method cannot throw the exception, as the third condition is not satisfied: A preceding `catch` clause can handle the exception (`EOFException`). The compiler flags a *warning* for the `catch` clause which is unreachable.

**Example 7.13** *Conditions for Rethrowing Final Exceptions*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class RethrowMe {
  public static void main(String[] args) throws EOFException {
    try {
      switch (1) {
        case 1: throw new FileNotFoundException("File not found");
        case 2: throw new EOFException("End of file");
        default: System.out.println("OK");
      }
    } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe);
    } catch (IOException ioe) {
      throw ioe;                                                    // (1)
    }
  }

  public static void rethrowA() throws EOFException {
    try {
      // Empty try block.
    } catch (EOFException eofe) { // Compile-time error: exception not thrown
                                  //                      in try block.
      throw eofe;                                                   // (2)
```

```
    }
  }

  public static void rethrowB() throws EOFException {
    try {
      throw new EOFException("End of file");
    } catch (EOFException eofe) {
      System.out.println(eofe);
    } catch (IOException ioe) {   // Compile-time warning: unreachable clause
      throw ioe;                                                    // (3)
    }
  }
}
```

### Chaining Exceptions

It is common that the handling of an exception leads to the throwing of another exception. In fact, the first exception is the cause of the second exception being thrown. Knowing the *cause* of an exception can be useful, for example, when debugging the application. The Java API provides a mechanism for *chaining* exceptions for this purpose.

The class `Throwable` provides the following constructors and methods to handle chained exceptions:

---

**Click here to view code image**

```
Throwable(Throwable cause)
Throwable(String msg, Throwable cause)
```

Sets the throwable to have the specified cause, and also specifies a detail message if the second constructor is used. Most exception types provide analogous constructors.

**Click here to view code image**

```
Throwable initCause(Throwable cause)
```

Sets the cause of this throwable. Typically called on exception types that do not provide a constructor to set the cause.

```
Throwable getCause()
```

Returns the cause of this throwable, if any; otherwise, it returns `null`.

---

**Example 7.14** illustrates how a chain of cause-and-effect exceptions, referred to as the *backtrace*, associated with an exception can be created and manipulated. In the method `chainIt()`, declared at (2), an exception is successively caught and associated as a *cause* with a new exception before the new exception is thrown, resulting in a chain of exceptions. This association is made at (3) and (4). The `catch` clause at (1) catches the exception thrown by the method `checkIt()`. The causes in the chain are successively retrieved by calling the `getCause()` method. Program output shows the resulting back-trace: which exception was the cause of which exception, and the order shown being reverse to the order in which they were thrown, the first one in the chain being thrown last.

**Example 7.14** *Chaining Exceptions*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionsInChain {
  public static void main(String[] args) {
    try {
      chainIt();
    } catch (Exception e) {                                  // (1)
      System.out.println("Exception chain: " + e);
      Throwable t = e.getCause();
      while (t != null) {
        System.out.println("Cause: " + t);
        t = t.getCause();
      }
    }
  }

  public static void chainIt() throws Exception {            // (2)
    try {
      throw new FileNotFoundException("File not found");
    } catch (FileNotFoundException e) {
      try {
        IOException ioe = new IOException("File error");
        ioe.initCause(e);                                    // (3)
        throw ioe;
      } catch (IOException ioe) {
        Exception ee = new Exception("I/O error", ioe);      // (4)
        throw ee;
      }
    }
```

```
  }
}
```

Output from the program:

**Click here to view code image**

```
Exception chain: java.lang.Exception: I/O error
Cause: java.io.IOException: File error
Cause: java.io.FileNotFoundException: File not found
```

A convenient way to print the backtrace associated with an exception is to invoke the `printStackTrace()` method on the exception. The `catch` clause at (1) in **Example 7.14** can be replaced with the `catch` clause at (1') below that calls the `printStackTrace()` method on the exception.

**Click here to view code image**

```
...
try {
  chainIt();
} catch (Exception e) {                                  // (1')
  e.printStackTrace();                                   // Print backtrace.
}
...
```

The refactoring of **Example 7.14** will result in the following analogous printout of the backtrace, showing as before the exceptions and their causes in the reverse order to the order in which they were thrown:

**Click here to view code image**

```
java.lang.Exception: I/O error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:23)
      at ExceptionsInChain.main(ExceptionsInChain.java:8)
Caused by: java.io.IOException: File error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:19)
      ... 1 more
Caused by: java.io.FileNotFoundException: File not found
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:16)
      ... 1 more
```

## 7.7 The `try`-with-resources Statement

Normally, objects in Java are automatically garbage collected at the discretion of the JVM when they are no longer in use. However, *resources* are objects that need to be explicitly closed when they are no longer needed. Files, streams, and database connections are all examples that fall into this category of objects. Such resources also rely on underlying system resources for their use. By closing such resources, any underlying system resources are also freed, and can therefore be recycled. Resource leakage (i.e., failure to close resources properly) can lead to performance degradation as resources get depleted.

Best practices recommend the following idiom for using a resource:

- Open the resource—that is, allocate or assign the resource.
- Use the resource—that is, call the necessary operations on the resource.
- Close the resource—that is, free the resource.

Typically, all three steps above can throw exceptions, and to ensure that a resource is always closed after use, the recommended practice is to employ a combination of `try`-`catch`-`finally` blocks. The first two steps are usually nested in a `try` block, with any associated `catch` clauses, and the third step is executed in a `finally` clause to ensure that the resource, if it was opened, is always closed regardless of the path of execution through the `try`-`catch` blocks.

**Example 7.15** shows a naive approach to resource management, that can result in resource leakage. It uses a `BufferedReader` associated with a `FileReader` to read a line from a text file (§20.3, p. 1251). The example follows the idiom for resource usage. As we can see, an exception can be thrown from each of the steps. If any of the first two steps throw an exception, the call to the `close()` method at (4) will never be executed to close the resource and thereby any underlying resources, as execution of the method will be terminated and the exception propagated.

**Example 7.15** *Naive Resource Management*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class NaiveResourceUse {
  public static void main(String[] args)
```

```
        throws FileNotFoundException, EOFException, IOException {

    // Open the resource:
    var fis = new FileReader(args[0]);             // (1) FileNotFoundException
    var br = new BufferedReader(fis);

    // Use the resource:
    String textLine = br.readLine();               // (2) IOException

    if (textLine != null) {
      System.out.println(textLine);
    } else {
      throw new EOFException("Empty file.");       // (3) EOFException
    }

    // Close the resource:
    System.out.println("Closing the resource.");
    br.close();                                    // (4) IOException
  }
}
```

Running the program:

Click here to view code image

```
>java NaiveResourceUse EmptyFile.txt
Exception in thread "main" java.io.EOFException: Empty file.
      at NaiveResourceUse.main(NaiveResourceUse.java:20)
```

Running the program:

Click here to view code image

```
>java NaiveResourceUse Slogan.txt
Code Compile Compute
Closing the resource.
```

**Example 7.16** improves on **Example 7.15** by explicitly using `try` - `catch` - `finally` blocks to manage the resources. No matter how the `try` - `catch` blocks execute, the `finally` block at (4) will always be executed. If the resource was opened, the `close()` method will be called at (5). The `close()` method is only called on the `BufferedReader` object, and not on the `FileReader` object. The reason is that the `close()` method of the `BufferedReader` object implicitly calls the `close()` method of the `FileReader` object associated with it. This is typical of how the `close()` method of

such resources works. Also, calling the `close()` method on a resource that has already been closed normally has no effect; the `close()` method is said to be *idempotent*.

**Example 7.16** *Explicit Resource Management*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TryWithoutARM {
  public static void main(String[] args) {
    BufferedReader br = null;
    try {
      // Open the resource:
      var fis = new FileReader(args[0]);             // (1) FileNotFoundException
      br = new BufferedReader(fis);

      // Use the resource:
      String textLine = br.readLine();               // (2) IOException

      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");       // (3) EOFException
      }
    } catch (FileNotFoundException | EOFException e) {
      e.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    } finally {                                      // (4)
      if (br != null) {
        try {
          System.out.println("Closing the resource.");
          br.close();                                // (5) IOException
        } catch(IOException ioe) {
          ioe.printStackTrace();
        }
      }
    }
  }
}
```

Running the program:

Click here to view code image

```
>java TryWithoutARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithoutARM.main(TryWithoutARM.java:20)
Closing the resource.
```

Running the program:

Click here to view code image

```
>java TryWithoutARM Slogan.txt
Code Compile Compute
Closing the resource.
```

A lot of boilerplate code is required in explicit resource management using `try-catch-finally` blocks, and the code can get tedious and complex, especially if there are several resources that are open and they all need to be closed explicitly. The `try-` with-resources statement takes the drudgery out of associating `try` blocks with corresponding `finally` blocks to ensure proper resource management. Any resource declared in the header of the `try`-with-resources statement will be automatically closed, regardless of how execution proceeds in the `try` block. The compiler expands the `try`-with-resources statement (and any associated `catch` or `finally` clauses) into basic `try-catch-finally` blocks to guarantee this behavior. It implicitly generates a `finally` block that calls the `close()` method of each resource declared in the header of the `try`-with-resources statement. There is no need to call the `close()` method of the resource, let alone provide any explicit `finally` clause for this purpose.

The `close()` method provided by the resources declared in a `try`-with-resources statement is the implementation of the sole `abstract` method declared in the `java.lang.AutoCloseable` interface. In other words, these resources must implement the `AutoCloseable` interface (p. 412).

The syntax of the `try`-with-resources statement augments the `try` statement with a `try` *header* that comprises a list of *resource declaration statements* separated by a semicolon ( `;` ).

Click here to view code image

```
try ( resource_declaration_statement1; ... ; resource_declaration_statementm ) {
  statements
} catch (exception_type1 |...| exception_typek parameter1) { // multi-catch
  statements
```

```
}
...
  catch (exception_type_n parameter_n) {                    // uni-catch
  statements
} finally {                                                 // finally
  statements
}
```

The syntax shown above is for the *extended* `try`-with-resources statement, which has explicit `catch` and/or `finally` clauses associated with it. The *basic* `try`-with-resources statement has no explicit `catch` or `finally` clauses associated with it.

We will use **Example 7.17** to illustrate the salient features of the `try`-with-resources statement. The extended `try`-with-resources statement at (1) declares and initializes two resources in its `try` header:

**Click here to view code image**

```
try (var fis = new FileReader(args[0]);       // (1) FileNotFoundException
     var br = new BufferedReader(fis)) {
  // ...
}
```

Note that no calls should be made to the `close()` method of a resource variable that is declared and initialized in the `try` header. Any `catch` clause or `finally` clauses associated with an extended `try`-with-resources statement will be executed *after* all the declared resources have been closed. When running **Example 7.17** with an empty file, the `EOFException` is caught and handled *after* the two resources denoted by the references `br` and `fis` have been closed.

The scope of a resource variable is the `try` block—that is, it is a local variable in the `try` block. Local variable type inference with `var` can be used with advantage in declaring resources in this context (§3.13, p. 142). Also, the resource variables are implicitly `final`, guaranteeing that they cannot be assigned to in the `try` block, thus ensuring that the right resources will be closed when the `close()` method is called after the execution of the `try` block. The closing order of the resources is the reverse order in which they are declared in the `try` header—that is, the resource declared first in the `try` header is closed last after the execution of the `try` block. In **Example 7.17**, the `close()` method is first called on the `BufferedReader` object denoted by the reference `br` as it is declared last, followed by a call to the `close()` method of the `FileReader` object denoted by the reference `fis`. Note how the idempotent behavior of the `close()` method in the `FileReader` object comes into play: It is first called implicitly by the `close()` method of the `BufferedReader` object, and then again by

virtue of the resource declaration statement in the `try` header. The `try-` with-resources statement at (1) can be easily rewritten to avoid this redundant call, where now only the `BufferedReader` resource is declared, ensuring that the `close()` method of the `FileReader` resource will be called only once when the `BufferedReader` resource is closed:

Click here to view code image

```
try (var br = new BufferedReader(new FileReader(args[0]))) {
  // ...
}
```

**Example 7.17** *Using* `try` *with Automatic Resource Management (ARM)*

Click here to view code image

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.stream.Stream;

public class TryWithARM {
  public static void main(String[] args) {
    try (var fis = new FileReader(args[0]);        // (1) FileNotFoundExeception
         var br = new BufferedReader(fis)) {
      String textLine = br.readLine();             // (2) IOException
      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");     // (3) EOFException
      }
    } catch (FileNotFoundException | EOFException e) {
      e.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Running the program:

Click here to view code image

```
>java TryWithARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithARM.main(TryWithARM.java:15)
```

Running the program:

Click here to view code image

```
>java TryWithARM Slogan.txt
Code Compile Compute
```

### Concise `try`-with-resources Statement

The header of the `try`-with-resources statement can be made less verbose by factoring out the resource declarations from the header. This refactoring involves *declaring the resources preceding* the `try` statement and specifying instead the *resource variables* in the header. The resources must be `AutoCloseable` (to provide the `close()` method) as before, but in addition must also be *either* `final` *or effectively* `final` to ensure that the right resource is closed.

Click here to view code image

```
final ACResource1 acr1 = new ACResource1();     // (1)
ACResource2 acr2 = new ACResource2();           // (2) Effectively final at (3)
// ...
try (acr1; acr2) {                              // (3) Resource variables.
  // Use the resources.
} // Both resources closed on exit.
```

The variable names of the `AutoCloseable` resources declared at (1) and (2) are specified in the header of the `try`-with-resources statement at (3). Apart from the syntactic difference, the semantics of the `try`-with-resources statement are not changed in any way. The resources `acr1` and `acr2` are guaranteed to be closed no matter how the execution of the `try` statement at (3) proceeds.

However, care must be taken if any resource declarations proceeding the `try`-with-resources statement can throw exceptions. For example, if the declaration of `acr2` at (2) above throws an exception, the execution will be interrupted and the resource `acr1` created at (1) will not be closed—leading to resource leakage. One solution is to move the declaration of `acr2` into the header of the `try`-with-resources statement as shown below at (3). The list in the header can be a mix of resource variables and resource declarations. This setup will ensure that both resources will be closed regardless of how the execution in the `try` statement proceeds. The `try`-with-resources statement can

be augmented with the necessary `catch` blocks to catch any exceptions thrown during its execution, as shown at (4).

Click here to view code image

```
final ACResource1 acr1 = new ACResource1();              // (1)
// ...
try (acr1; ACResource2 acr2 = new ACResource2()) {       // (3)
  // Use the resources.
} catch (Exception ex) {                                 // (4)
  ex.printStackTrace();
} // Both resources closed on exit.
```

### Implementing the `AutoCloseable` Interface

A declared resource must provide the `close()` method that will be called when the resource is to be closed. This is guaranteed by the fact that the resource must implement the `java.lang.AutoCloseable` interface which specifies the `close()` method. The compiler only allows resource declarations or resource variables in the `try` header that are `AutoCloseable`. Many classes in the Java API implement the `AutoCloseable` interface: byte input streams, byte output streams, readers and writers in the `java.io` package (Chapter 20, p. 1231), and database connections, query statements, and data result sets in the `java.sql` package (Chapter 24, p. 1511).

The abstract `void` method `close()` in the `java.lang.AutoCloseable` interface is declared to throw an `Exception`. Files and streams implement the `java.io.Closeable` interface which extends the `AutoCloseable` interface, but the abstract method `close()` in the `Closeable` interface is specified to throw an `IOException`, a subtype of `Exception`.

Click here to view code image

```
interface AutoCloseable {                        // java.lang package
  void close() throws Exception;
}

interface Closeable extends AutoCloseable {      // java.io package
  void close() throws IOException;               // Mandatory idempotent
}
```

What is important is that any class implementing the `AutoCloseable` interface can override the `throws` clause of the abstract `close()` method. The implementation of the `close()` method can choose to throw any `Exception`, meaning either an

`Exception` or subtypes of `Exception`, but also throw no exception at all if the method cannot fail.

An implementation of the `close()` method of the `Closeable` interface must be idempotent. There is no such requirement for implementations of the `close()` method of the `AutoCloseable` interface, but it is highly recommended that they are idempotent.

In **Example 7.18**, the class `Gizmo` implements the `AutoCloseable` interface. The class uses the field `closed` to keep track of whether the gizmo is closed or not. The `close()` method at (2) reports whether the gizmo is already closed; if not, it sets the `closed` flag to `true` and throws an unchecked `IllegalArgumentException`. The `close()` method is idempotent, as it will only report that the gizmo is already closed once it has been called. The class `Gizmo` also has the method `compute()` at (3), that throws an unchecked `ArithmeticException`.

The class `GizmoTest` uses the `Gizmo` class. It declares and initializes a `Gizmo` in a `try-` with-resources statement at (4), and calls its `compute()` method. The output shows that the unchecked `ArithmeticException` thrown by the `compute()` method is caught by the `catch` clause at (5), after the `close()` method has been called on the resource—any `catch` clauses and `finally` clause explicitly specified with an extended `try` -with-resources statement are executed *after* the declared resources have been closed.

**Example 7.18** *Implementing the* `AutoCloseable` *Interface*

Click here to view code image

```
// File: Gizmo.java
public class Gizmo implements AutoCloseable {
  private boolean closed = false;                 // (1) Closed if true
  @Override
  public void close() {                           // (2) Idempotent
    System.out.println("Enter: close()");
    if (closed) {
      System.out.println("Already closed");
    } else {
      closed = true;
      System.out.println("Gizmo closed");
      System.out.println("Throwing IllegalArgumentException in close()");
      throw new IllegalArgumentException("thrown in close()"); // Suppressed
    }
    System.out.println("Exit: close()");     // Only executed if already closed.
  }

  public void compute() {                          // (3)
```

```
    System.out.println("Enter: compute()");
    System.out.println("Throwing ArithmeticException in compute()");
    throw new ArithmeticException("thrown in compute()");
  }
}
```

Click here to view code image

```
// File: GizmoTest.java
public class GizmoTest {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {             // (4)
      myGizmo.compute();
    } catch (Exception ex) {                      // (5)
      System.out.println("Printing stack trace in catch clause of main():");
      ex.printStackTrace();
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

Click here to view code image

```
// File: GizmoTest2.java
public class GizmoTest2 {
  public static void main(String[] args) {
    try (var myGizmo = new Gizmo()) {
      myGizmo.compute();
    } catch (Exception ex) {                                        // (6)
      System.out.println("Exception caught in the catch clause of main():\n\t"
                         + ex);
      System.out.println("Printing suppressed exceptions "
                         + "in the catch clause of main():");
      Throwable[] supressedEx = ex.getSuppressed();                 // (7)
      for (Throwable t : supressedEx) {
        System.out.println("\t" + t);
      }
    } finally {
      System.out.println("Finally: Done in main()");
    }
  }
}
```

Output from running GizmoTest:

Click here to view code image

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Printing stack trace in catch clause of main():
java.lang.ArithmeticException: thrown in compute()
      at Gizmo.compute(Gizmo.java:21)
      at GizmoTest.main(GizmoTest.java:5)
      Suppressed: java.lang.IllegalArgumentException: thrown in close()
          at Gizmo.close(Gizmo.java:13)
          at GizmoTest.main(GizmoTest.java:6)
Finally: Done in main()
```

Output from running GizmoTest2:

Click here to view code image

```
Enter: compute()
Throwing ArithmeticException in compute()
Enter: close()
Gizmo closed
Throwing IllegalArgumentException in close()
Exception caught in the catch clause of main():
      java.lang.ArithmeticException: thrown in compute()
Printing suppressed exceptions in the catch clause of main():
      java.lang.IllegalArgumentException: thrown in close()
Finally: Done in main()
```

### Suppressed Exceptions

The program output from `GizmoTest` in **Example 7.18** shows that the `ArithmeticException` was thrown in the `compute()` method called in the `try` block at (4) *before* the `IllegalArgumentException` was thrown by the implicit call to the `close()` method. Since only one exception can be propagated, the `IllegalArgumentException` thrown last would mask the `ArithmeticException` and the `IllegalArgumentException` would be propagated. However, the stack trace shows that this is not the case.

Exceptions thrown from the body of a `try`-with-resources statement are given preferential treatment over exceptions thrown by the `close()` method when called implicitly to close a declared resource. Exceptions thrown in the `try` block pertain to the program logic and should not be masked, but at the same time any exceptions thrown

in the `close()` method should not be ignored. A solution is provided through *suppressed exceptions*. The class `Throwable` provides the following methods to handle such exceptions:

---

Click here to view code image

```
void addSuppressed(Throwable exception)
```

Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.

```
Throwable[] getSuppressed()
```

Returns an array containing all of the exceptions that were suppressed in order to deliver this exception.

---

An exception `s` that is associated with an exception `e` through the method call `e.addSuppressed(s)` is called a *suppressed exception*. The idea is that if exception `s` was thrown during the propagation of exception `e`, then exception `s` is suppressed and the propagation of exception `e` continues. When exception `e` is caught, it is possible to retrieve all its suppressed exceptions by the method call `e.getSuppressed()`.

From the stack trace printed by `GizmoTest` in **Example 7.18**, we see that the `IllegalArgumentException` was thrown in the `close()` method and is implicitly associated with the `ArithmeticException` that was thrown earlier in the `compute()` method—that is, the `IllegalArgumentException` was suppressed and the `ArithmeticException` was propagated.

In the `catch` clause at (6) of `GizmoTest2` in **Example 7.18**, the exception thrown and the suppressed exception are printed explicitly. The method call `ex.getSuppressed()` at (7) returns an array of `Throwable` containing the suppressed `IllegalArgumentException`. Of course, suppressing any exception from the `close()` method is only warranted if an exception is thrown in the body of the `try`-with-resources statement.

The compiler expands the `try`-with-resources statement into code (essentially a combination of basic `try`-`catch`-`finally` blocks with the necessary control flow) that ensures proper closing of the declared resources, and handling of suppressed exceptions.

### 7.8 Advantages of Exception Handling

*Robustness* refers to the ability of a software system to respond to errors during execution. A system should respond to unexpected situations at runtime in a responsible way. Applications that provide the user with frequent cryptic messages with error codes or that repeatedly give the user the silent treatment when something goes wrong can hardly be considered robust.

The exception handling mechanism in Java offers the following advantages that facilitate developing robust applications in Java:

- *Separation of exception handling code*
  The code for handling error situations can be separated from the code for the program logic by using the exception handling constructs provided by the language. Code that can result in error situations is confined in the `try` block, and their handling in the `catch` clause.
- *Transparent exception propagation*
  Propagation of a checked exception in the JVM stack cannot be ignored by an active method. The method must comply with the catch-or-declare requirement: Either catch and handle the exception, or propagate it by declaring it in the method's `throws` clause. Error situations causing exception propagation are thus always detected, and can be caught and remedied.
- *Exception categorization and specialization*
  The exception and error classes in the Java SE Platform API are organized in an inheritance hierarchy (**Figure 7.3**, **p. 369**). Classes higher in this hierarchy represent *categories* of exceptions and errors (`Exception`, `RuntimeException`, `IOException`, `Error`), whereas classes lower in this hierarchy represent more *specific* exceptions and errors (`NullPointerException`, `FileNotFoundException`, `AssertionError`). The `try-catch` construct allows flexibility in catching and handling exceptions. A `catch` clause can specify an exception category for coarse-grained exception handling, as the exception category class will subsume its more specific exception subclasses, or it can specify a more specific exception class for fine-grained exception handling. Best practice dictates that fine-grained exception handling be used.

#### Review Questions

**7.10** Which of these methods in the class will compile?

**Click here to view code image**

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
```

```
import java.io.FileReader;
import java.io.IOException;

public class ARMy {

  public static void methodA(String filename) throws FileNotFoundException {// (1)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodB(String filename)throws IOException {            // (2)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
  }

  public static void methodC(String filename)throws FileNotFoundException { // (3)
    var fis = new FileReader(filename);
    try (var br = new BufferedReader(fis)) { }
    catch (IOException ioe) { }
  }

  public static void methodD(String filename) throws FileNotFoundException {// (4)
    try (var fis = new FileReader(filename);
         var br = new BufferedReader(fis)) { }
  }

  public static void methodE(String filename) throws IOException {           // (5)
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodF(String filename) {                              // (6)
    try (FileReader fis = null; BufferedReader br = null) {
    }
  }

  public static void methodG(String filename)
                             throws IOException {                            // (7)
    try (FileReader fis = null; BufferedReader br = null) {
      fis = new FileReader(filename);
      br = new BufferedReader(fis);
    }
  }
}
```

Select the three correct answers.

**a.** The method at (1).

**b.** The method at (2).

**c.** The method at (3).

**d.** The method at (4).

**e.** The method at (5).

**f.** The method at (6).

**g.** The method at (7).

**7.11** What will the following program print when run?

**Click here to view code image**

```
import java.io.EOFException;
import java.io.IOException;

public class TryTwisting {
  public static void main(String[] args) {
    try {
      justDoIt();
    } catch (Exception t1) {
      System.out.println(t1);
      for (Throwable t : t1.getSuppressed())
        System.out.println("Suppressed: " + t);
    }
  }

  public static void justDoIt() throws Exception {
    IOException t2 = null;
    try {
      t2 = new IOException();
      throw t2;
    } finally {
      try {
        throw new EOFException();
      } catch (Exception t3) {
        t2.addSuppressed(t3);
      }
    }
  }
}
```

Select the one correct answer.

**a.** `java.io.EOFException`

**b.** `java.io.IOException`

**c.** `java.io.Exception`

**d.**

**Click here to view code image**

```
java.io.EOFException
Suppressed: java.io.IOException
```

**e.**

**Click here to view code image**

```
java.io.EOFException
Suppressed: java.io.Exception
```

**f.**

**Click here to view code image**

```
java.io.Exception
Suppressed: java.io.IOException
```

**g.**

**Click here to view code image**

```
java.io.Exception
Suppressed: java.io.EOFException
```

**h.**

**Click here to view code image**

```
java.io.IOException
Suppressed: java.io.EOFException
```

**i.**

Click here to view code image

```
java.io.IOException
Suppressed: java.io.Exception
```

7.12 What will make the following program compile and run without throwing an exception?

Click here to view code image

```
public class Truck implements AutoCloseable {
  public void close() {
    throw new Exception("Cannot close.");
  }

  public void load() {
    System.out.println("Loading truck.");
  }

  public static void main(String[] args) {
    try (var tl = new Truck()) {
      tl.load();
    }
  }
}
```

Select the one correct answer.

**a.** The program will compile and run without reporting any exception.

**b.** Add a `throws Exception` clause to the `close()` method.

**c.** Add a `throws Exception` clause to the `main()` method.

**d.** Add a `throws Exception` clause to both the `close()` and the `main()` methods.

**e.** Add a `catch (Exception e) {}` clause to the `try` statement in the `main()` method.

**f.** Add a `throws Exception` clause to the `close()` method and add a `catch (Exception e) {}` clause to the `try` statement in the `main()` method.

**g.** None of the above

7.13 Which one of these methods will fail to compile? Select the three correct answers.

**a.**

Click here to view code image

```
void task1(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException | FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

**b.**

Click here to view code image

```
void task2(int value) throws IOException {
  try {
    switch (value) {
      case 1: throw new EOFException();
      case 2: throw new FileNotFoundException();
    }
  } catch (EOFException e) {
    e = new IOException();
    throw e;
  } catch (FileNotFoundException e) {
    e = new IOException();
    throw e;
  }
}
```

**c.**

Click here to view code image

```
void task3() throws IOException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    e = new IOException();
    throw e;
```

```
  }
}
```

**d.**

Click here to view code image

```
void task4() throws FileNotFoundException {
  try {
    throw new FileNotFoundException();
  } catch (Exception e) {
    throw e;
  }
}
```

**e.**

Click here to view code image

```
void task5() throws Exception {
  try {
    throw new FileNotFoundException();
  } catch (IOException e) {
    throw e;
  }
}
```

7.14 What will be the output of the following program?

Click here to view code image

```
public class Resource implements AutoCloseable {
  public void action() { System.out.print("action "); }
  public void close() throws Exception { System.out.print("closure "); }
}
```

Click here to view code image

```
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (Exception ex) {
      System.out.print("error ");
```

```
    }
    System.out.print("the end ");
  }
}
```

Select the one correct answer.

**a.** action closure the end

**b.** action the end

**c.** action error the end

**d.** `action closure error the end`

**7.15** What will be the output of the following program?

**Click here to view code image**

```
import java.io.IOException;
public class Resource implements AutoCloseable {
  public void action() throws IOException {
    throw new IOException("action error ");
 }
  public void close() throws Exception {
    throw new Exception("closure error ");
 }
}
```

**Click here to view code image**

```
import java.io.IOException;
public class Test {
  public static void main(String[] args) {
    try (Resource r  = new Resource()) {
      r.action();
    } catch (IOException ex) {
      System.out.print("IO ");
      System.out.print(ex.getMessage());
    } catch (Exception ex) {
      System.out.print("Other ");
      System.out.print(ex.getSuppressed()[0].getMessage());
    }
    System.out.print("the end ");
  }
}
```

Select the one correct answer.

**a.** `IO action error Other closure error the end`

**b.** `IO action error the end`

**c.** `Other closure error IO action error the end`

**d.** `IO action error`

**e.** `IO action error Other closure error`

7.16 Which of the following statements is true?

Select the one correct answer.

**a.** Implicit and explicit `finally` blocks cannot coexist with the same `try`-with-resources statement.

**b.** The implicit `finally` block is executed before any explicit `finally` block.

**c.** Any explicit `finally` block is executed before the implicit `finally` block.

**d.** The execution order of the implicit and any explicit `finally` blocks is undetermined.