CHAPTER 10

Exception handling

LEARNING OBJECTIVES

By the end of this chapter you will understand the following:

- **How to use exception handling to create programs that are reliable and robust.**
- **How methods are executed by the JVM and how it handles error situations.**
- **Major scenarios of program execution when using the `try-catch` statement.**
- **How exceptions are thrown, propagated, caught and handled.**
- **The difference between checked and unchecked exceptions.**

INTRODUCTION

A program must be able to handle error situations gracefully when they occur at runtime. Java provides exception handling for this purpose. In this chapter, we take a closer look at how such error situations can occur and how they can be handled by the program. Further aspects of exception handling are discussed in Chapter 18.

10.1 What is an exception?

A program should be able to handle error situations that might occur at runtime. Error situations can be divided into two main categories:

1. *Programming errors*. For example, using an invalid index to access an array element, attempting to divide by zero, calling a method with illegal arguments or using a reference with the `null` value to access members of an object.
2. *Runtime environment errors*. For example, opening a file that does not exist, a read or write error when using a file, or a network connection going down unexpectedly.

Programming errors are error situations that occur because of *logical errors* in the program, while runtime environment errors are errors over which the program has little control. A program must be able to handle both kinds of errors. Ideally, programming errors should not occur, and the program should handle runtime environment errors gracefully.

An *exception* in Java signals that an error or an unexpected situation has occurred during program execution. *Exception handling* is the mechanism for dealing with such situations. It is based on the "*throw and catch*" principle. An exception is *thrown* when an error situation occurs during program execution. It is *propagated* by the JVM and *caught* by an *exception handler* that takes an appropriate action to *handle* the situation. This principle is embedded in the `try–catch` statement. All exceptions are objects and the Java standard library provides classes that represent different types of exceptions.

The examples in this chapter show how exceptions can be thrown, propagated, caught and handled. These examples are intentionally simplified to highlight the important concepts in exception handling.

10.2 Method execution and exception propagation

We will use Program 10.1 as our running example in this chapter, and will write several versions of it to illustrate exception handling. We want to calculate speed when distance and time are given. The program uses three methods:

1. The method `main()`, which calls the `printSpeed()` method with parameter values for distance and time, (1).
2. The method `printSpeed()`, which in turn calls the `calculateSpeed()` method, (2).
3. The method `calculateSpeed()`, which calculates the expression `(distance/time)` and returns the result in a `return` statement, (3).

Observe that integer division is performed in the evaluation of the expression `(distance/time)`, as both operands are integers, and that integer division by `0` is an *illegal* operation in Java. Attempt at integer division by `0` will result in a runtime error. (On the other hand, floating-point division in an expression like `10.0/0.0` will result in an infinitely large number, denoted by the constant `Double.POSITIVE_INFINITY`.)

Method execution

During program execution, the JVM uses a program stack to control the execution of methods (see Section 16.2 about stacks and Chapter 17 about recursion). Each stack frame on the program stack corresponds to one method call. Each method call results in the creation of a new stack frame that has storage for local variables (including parameters) in the method. The method whose stack frame is at the top of the program stack is the one currently being executed. When the method returns (i.e. has finished executing), its stack frame is removed from the top of the stack. Program execution continues in the method whose stack frame is now uncovered at the top of the program stack. This execution behaviour is called *normal execution*.

An important aspect of program execution is that if method `A()` calls method `B()`, method `A()` cannot continue its execution until method `B()` completes execution. At any given time during execution, the program stack will contain stack frames of methods that are *active*, i.e. methods that have been called but whose execution has not completed. If we at any given time print the information about all the active methods on the program stack, we obtain what is called a *stack trace*. The stack trace shows which methods are active while the current method at the top of the stack is executing.

Execution of Program 10.1 is illustrated in the sequence diagram in Figure 10.1. Execution of a method is shown as a box with local variables. The length of the box indicates how long a method is active. Just before the `return` statement in (3) is executed, we see that the program stack at this particular time contains three active methods: `main()`, `printSpeed()` and `calculateSpeed()`. The `return` statement in (3) returns the result from the `calculateSpeed()` method and completes the execution of this method. We see that the output from the program corresponds to the sequence of method calls in Figure 10.1.

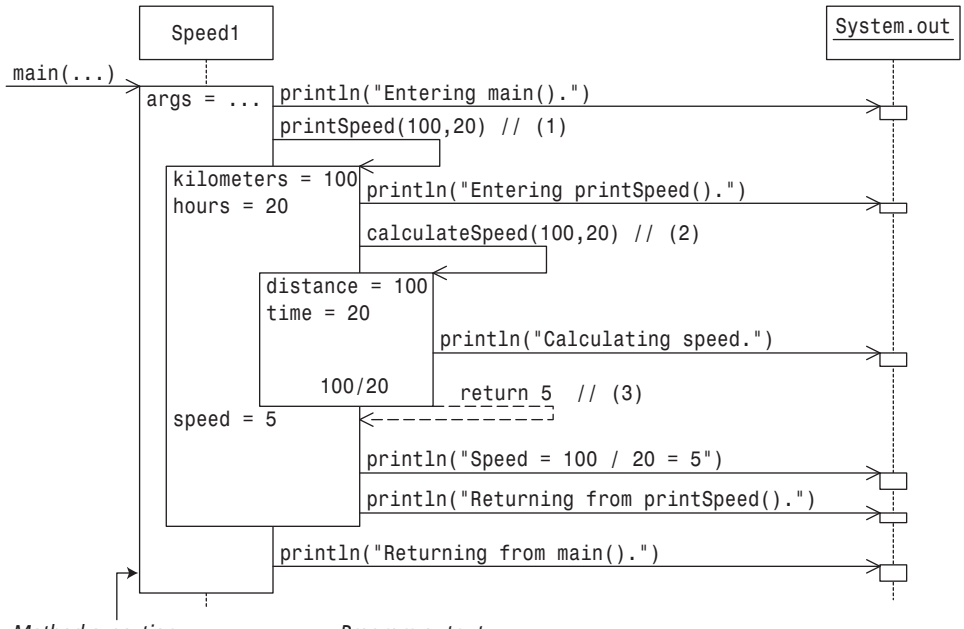
PROGRAM 10.1 Method entry and return

```
public class Speed1 {

  public static void main(String[] args) {
    System.out.println("Entering main().");
    printSpeed(100, 20);                                  // (1)
    System.out.println("Returning from main().");
  }

  private static void printSpeed(int kilometers, int hours) {
    System.out.println("Entering printSpeed().");
    int speed = calculateSpeed(kilometers, hours);        // (2)
    System.out.println("Speed = " +
                       kilometers + "/" + hours + " = " + speed);
    System.out.println("Returning from printSpeed().");
  }

  private static int calculateSpeed(int distance, int time) {
    System.out.println("Calculating speed.");
    return distance/time;                                 // (3)
  }
}
```

Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().
```

FIGURE 10.1 Method execution (Program 10.1)



Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().
```

BEST PRACTICE

Use of print statements, as illustrated in Program 10.1, is a useful debugging technique for tracking program execution.

Stack trace

If we replace the following call in the `main()` method from Program 10.1:

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

with

```
printSpeed(100, 0);      // (1) The second parameter is 0.
```

and run the program, we get the following output in the terminal window:

```
Entering main().
Entering printSpeed().
Calculating speed.
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

```
    at Speed1.calculateSpeed(Speed1.java:20)
    at Speed1.printSpeed(Speed1.java:12)
    at Speed1.main(Speed1.java:6)
```

The execution of the program is illustrated in the sequence diagram in Figure 10.2. We see that all goes well until the execution of the statement at (3), as corroborated by the output from the program. An error situation occurs at (3) during the evaluation of the expression `distance/time`, because the variable `time` has the value `0`. This error situation is signalled by *throwing* an `ArithmeticException` (i.e. an object of the class `ArithmeticException`), which is sent back (we say *propagated*) through the stack frames on the program stack. The JVM takes care of forwarding an exception to the active methods on the program stack in the right order.

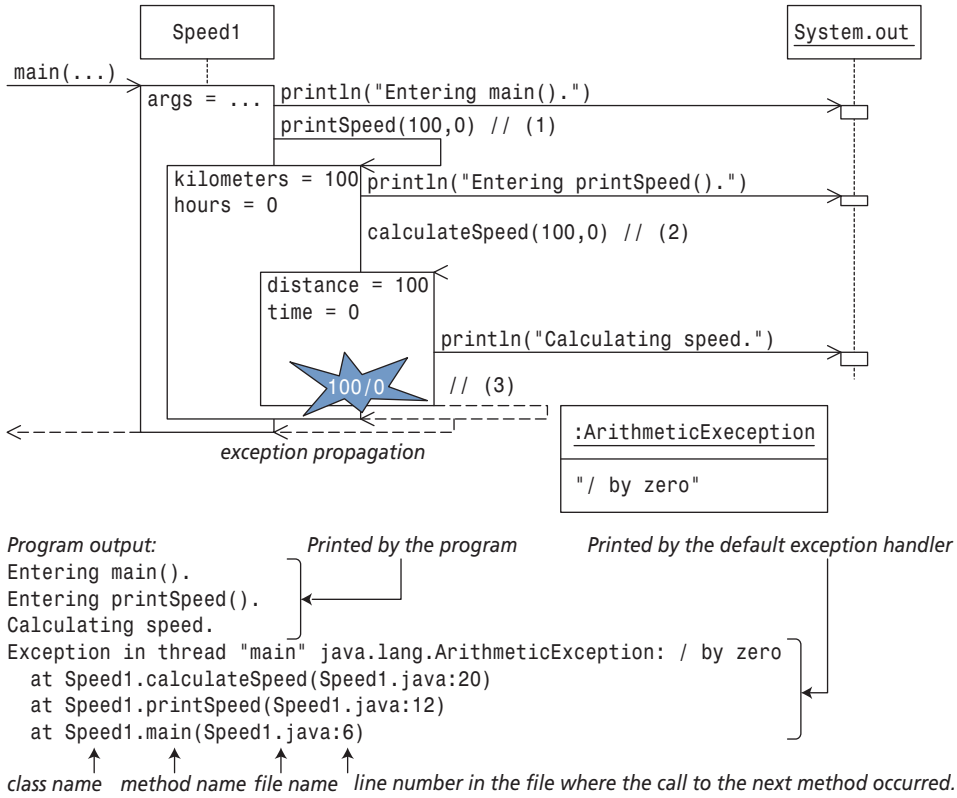
Exception propagation

Propagation of an exception takes place in the following way: the exception is offered to the method whose stack frame is at the top of the program stack, i.e. the method in which the exception occurred. In this case, it is the method `calculateSpeed()`. Because this method does not have any code to deal with this exception, the `calculateSpeed()` method is terminated and its stack frame at the top of the program stack is removed. The exception is next offered to the active method that is now at the top of the program stack: the `printSpeed()` method. It also does not have any code to handle the exception, and consequently it is also terminated and its stack frame removed. The exception is next offered to the `main()` method, which is also terminated and its stack frame removed, since it does not have any code for exception handling either. Now the exception has propagated to the *top level*, and here it is handled by a *default exception handler* in the JVM. This exception handler prints information about the exception, together with the stack trace at the time when the exception occurred (see Figure 10.2). The execution of the program is then terminated.

By comparing Figure 10.1 and Figure 10.2, we see that the error situation at runtime has great influence on the behaviour of the program. Program execution does not continue in the normal way under exception propagation, and the exception is not forwarded by any `return` statement. The execution of each active method on the program stack is successively terminated when an exception is propagated, unless an active method catches the exception and thereby stops its propagation.

For the terminal-based applications that we have developed so far, the program is terminated after the exception is handled by a default exception handler. For applications with a GUI (Chapter 20), the program continues after the exception has been handled by a default exception handler. In both cases, a default exception handler prints information from the program stack to the terminal window.

FIGURE 10.2 Exception propagation (integer division by 0)



10.3 Exception handling

In many cases, it is not advisable to let exceptions be handled by a default exception handler. The consequences of terminating program execution too early can be drastic: for example, data can be lost. The language construct `try–catch` can be used for exception handling in Java. Figure 10.3 shows a `try` block followed by a `catch` block. A `try` block consists of a block with the keyword `try` in front. A `try` block can contain arbitrary code, but normally it contains statements that can result in an exception being thrown during execution. A `catch` block is associated with a `try` block. A `catch` block constitutes an exception handler. An exception can be thrown as a result of executing the code in the `try` block, and this exception can be caught and handled in an associated `catch` block. The block notation, `{}`, is required for the `try` and `catch` blocks, even if a block only contains a single statement.

A `catch` block resembles a method declaration. The head of a `catch` block consists of the keyword `catch` and the declaration of a single parameter specifying which type of exceptions this `catch` block can handle. A `catch` block can contain arbitrary code, but its main purpose is to execute actions for handling the error situation represented by the exception caught by the `catch` block.

Since a `try` block is a local block, the local variables declared in the block can only be used in the block itself. The same is also true for a `catch` block, where the exception parameter is considered a local variable, as is the case for formal parameters of a method.

FIGURE 10.3 **try–catch statement**

The try block contains the code that can lead to an exception being thrown.

try block

```
try {
  int speed = calculateSpeed(kilometers, hours);
  System.out.println("Speed = " +
                     kilometers + "/" + hours + " = " + speed);
}
```

catch block

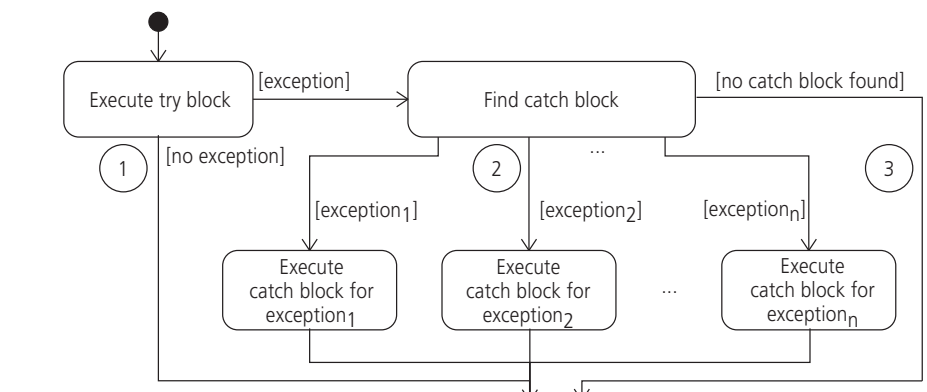
```
catch (ArithmeticException exception) {   one catch block parameter
  System.out.println(exception + " (handled in printSpeed())");
}
```

A catch block can catch an exception and handle it, if it is of the right type.

Figure 10.4 illustrates three typical scenarios when using the `try–catch` statement. These scenarios comprise the following situations during execution:

1. The code in the `try` block is executed, and no exception is thrown.
2. The code in the `try` block is executed and an exception is thrown. This exception is caught and handled in a corresponding `catch` block.
3. The code in the `try` block is executed, an exception is thrown, but no `catch` block is found for handling the exception.

FIGURE 10.4 **try–catch scenarios**



For scenarios 2 and 3 in Figure 10.4, the execution of the `try` block is terminated when an exception is thrown, skipping the rest of the `try` block.

For scenarios 1 and 2 in Figure 10.4, normal execution continues after the `try-catch` blocks, as there is no exception to be propagated. For scenario 3 the exception will be propagated as described in Section 10.2.

If the exception is caught and handled during its propagation, normal execution is resumed from that point onwards. This means that an exception can be handled in a different method from the one in which it was thrown, and a `catch` block is only executed if it catches an exception. The sequence in which statements are executed, and thereby the sequence of active methods on the program stack, is called the *runtime behaviour* of the program. It determines which exceptions can be thrown and how these are handled.

`try–catch` scenario 1: no exception

The method `printSpeed()` in Program 10.2 uses a `try–catch` statement to handle exceptions of the type `ArithmeticException`. This method calls the `calculateSpeed()` method in the `try` block at (2). The corresponding `catch` block, (4), is declared to catch exceptions of the type `ArithmeticException`. The handling of such an exception in the `catch` block consists of printing the exception. This type of exception can occur in the method `calculateSpeed()` during the evaluation of the arithmetic expression in (5). This means that if such an exception is thrown in the method `calculateSpeed()`, it will be caught in the method `printSpeed()`.

The runtime behaviour for Program 10.2 is shown in Figure 10.5. This behaviour corresponds to scenario 1 in Figure 10.4, where no exception occurs in the execution of the `try` block. The entire `try` block in the method `printSpeed()` is executed, while the `catch` block is skipped, as no exception is thrown in the method `calculateSpeed()`. We note that the execution of Program 10.2 shown in Figure 10.5 corresponds with the execution of Program 10.1 shown in Figure 10.1, and we get the same output in the terminal window in both programs.

PROGRAM 10.2 Exception handling

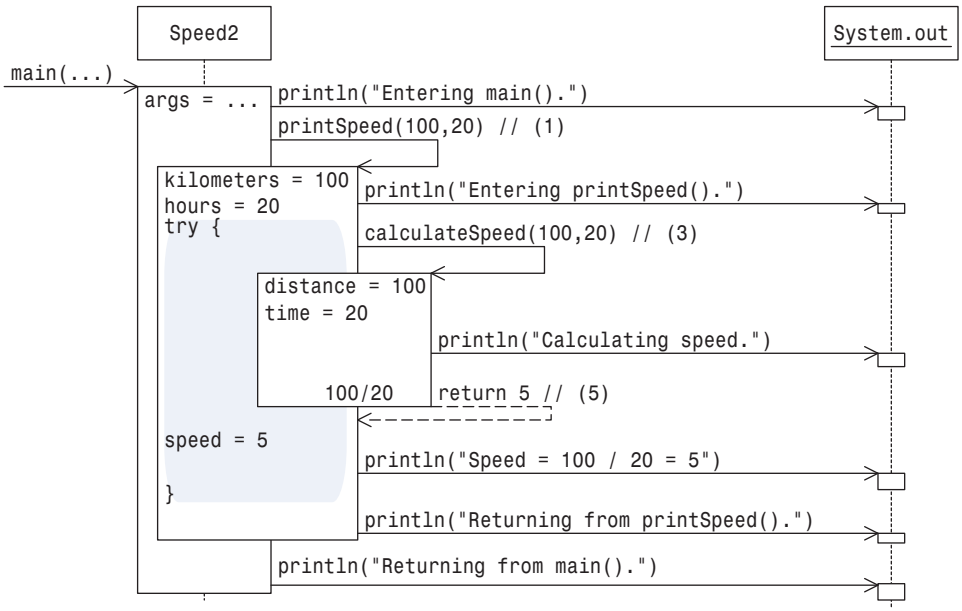
```
public class Speed2 {

  public static void main(String[] args) {
    System.out.println("Entering main().");
    printSpeed(100, 20);                                        // (1)
    System.out.println("Returning from main().");
  }

  private static void printSpeed(int kilometers, int hours) {
    System.out.println("Entering printSpeed().");
    try {                                                       // (2)
      int speed = calculateSpeed(kilometers, hours);            // (3)
      System.out.println("Speed = " +
                         kilometers + "/" + hours + " = " + speed);
    }
    catch (ArithmeticException exception) {                     // (4)
      System.out.println(exception + " (handled in printSpeed())");
    }
    System.out.println("Returning from printSpeed().");
  }
```

```
  private static int calculateSpeed(int distance, int time) {
    System.out.println("Calculating speed.");
    return distance/time;                                    // (5)
  }
}
```

Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().
```


FIGURE 10.5 Exception handling (Program 10.2): scenario 1 in Figure 10.4



Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5
Returning from printSpeed().
Returning from main().
```

`try–catch` scenario 2: exception handling

If we again replace the following call in the `main()` method from Program 10.2:

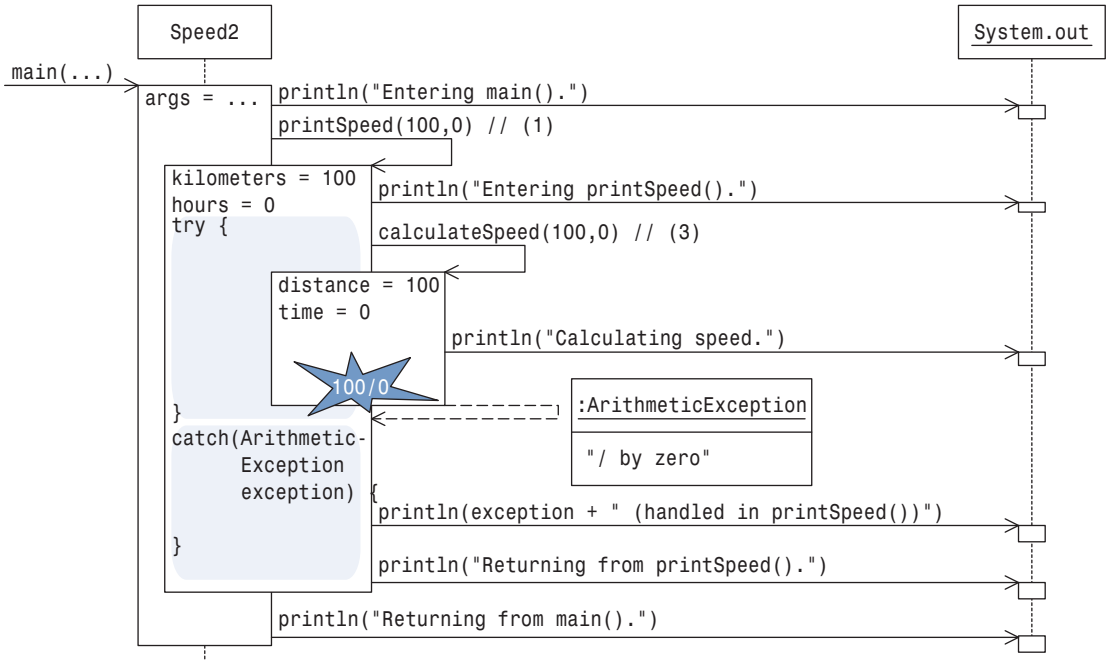
```
printSpeed(100, 20);     // (1)
```

with

```
printSpeed(100, 0);      // (1) Second parameter is 0.
```

the program behaviour is as shown in Figure 10.6. It corresponds to scenario 2 in Figure 10.4. Integer division by `0` results in an `ArithmeticException` being thrown at (5) in the `calculateSpeed()` method. The execution of this method is terminated and the exception propagated. It is caught by the `catch` block in the method `printSpeed()`. After handling the exception, normal execution of the program is resumed, as shown by the output in Figure 10.6.

FIGURE 10.6 Exception handling (Program 10.2): scenario 2 in Figure 10.4



Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmeticException: / by zero (handled in printSpeed())
Returning from printSpeed().
Returning from main().
```

`try–catch` scenario 3: exception propagation

Program 10.3 illustrates scenario 3 from Figure 10.3. Scenario 3 shows what can happen when an exception is thrown during the execution of a `try` block and no corresponding `catch` block is found to handle the exception. The scenario shows that the exception is propagated in the usual way, as we have seen in Section 10.2.

In Program 10.3, both the `main()` and the `printSpeed()` methods use the `try–catch` statement, at (1) and (4) respectively. The `main()` method has a `catch` block to catch an excep-

tion of the type `ArithmeticException`, (3), while the `printSpeed()` method has a `catch` block to catch an exception of the type `IllegalArgumentException`, (6).

The runtime behaviour of the program (Figure 10.7) shows that integer division by `0` again results in an `ArithmeticException` being thrown at (7) in the method `calculateSpeed()`. The execution of this method is terminated and the exception is propagated. It is *not* caught by the `catch` block in the method `printSpeed()`, since this `catch` block is declared to catch exceptions of the type `IllegalArgumentException`, not exceptions of the type `ArithmeticException`. The execution of the method `printSpeed()` is terminated (statements after (5) are not executed) and the exception is propagated further. The exception `ArithmeticException` is now caught by the `catch` block in the `main()` method at (3). After handling of the exception in the `catch` block in the `main()` method, normal execution of the program is resumed, as can be seen from the output in Program 10.3.

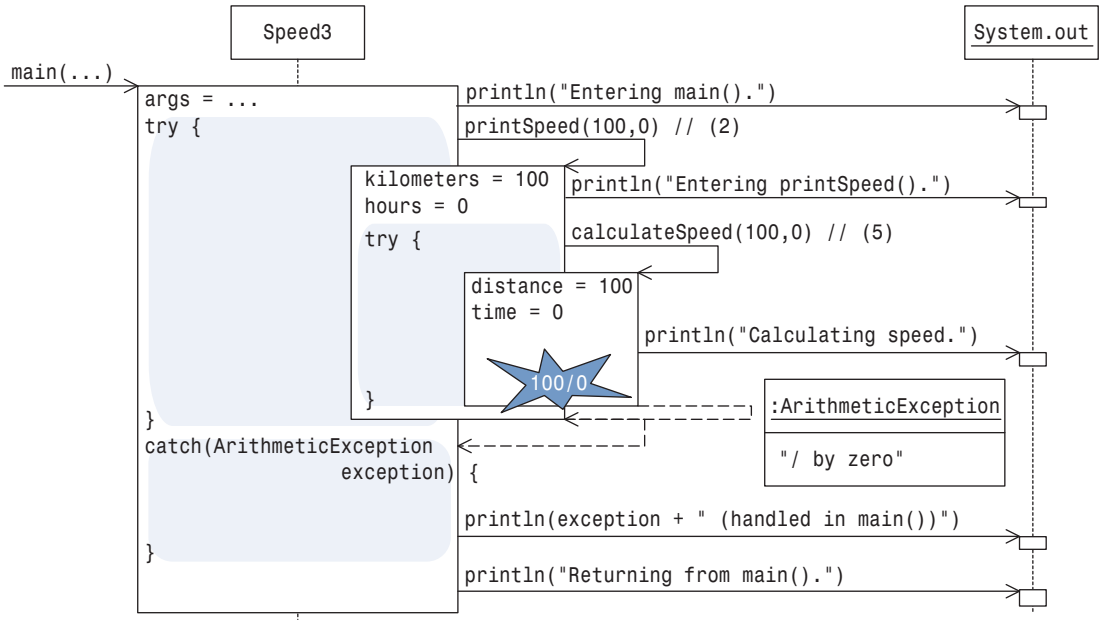
PROGRAM 10.3 Exception handling: scenario 3 in Figure 10.4

```
public class Speed3 {

  public static void main(String[] args) {
    System.out.println("Entering main().");
    try {                                                          // (1)
      printSpeed(100,20);                                          // (2)
    }
    catch (ArithmeticException exception) {                        // (3)
      System.out.println(exception + " (handled in main())");
    }
    System.out.println("Returning from main().");
  }

  private static void printSpeed(int kilometers, int hours) {
    System.out.println("Entering printSpeed().");
    try {                                                          // (4)
      int speed = calculateSpeed(kilometers, hours);               // (5)
      System.out.println("Speed = " +
                         kilometers + "/" + hours + " = " + speed);
    }
    catch (IllegalArgumentException exception) {                   // (6)
      System.out.println(exception + " (handled in printSpeed())");
    }
    System.out.println("Returning from printSpeed().");
  }

  private static int calculateSpeed(int distance, int time) {
    System.out.println("Calculating speed.");
    return distance/time;                                          // (7)
  }
}
```

Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmeticException: / by zero (handled in main())
Returning from main().
```


FIGURE 10.7 Exception handling (Program 10.3): scenario 3 in Figure 10.4



Program output:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.ArithmeticException: / by zero (handled in main())
Returning from main().
```

BEST PRACTICE

Start designing your exception handling strategy from the beginning, rather than tacking it on when the implementation is done.

BEST PRACTICE

Exception handling carries a performance penalty. For this reason, avoid placing a `try–catch` statement inside a loop.

10.4 Checked exceptions

Exception handling allows the program to deal with error situations during program execution. It is possible for a method that throws an exception to let the exception propagate further without doing anything about it. However, Java defines some special exceptions that a program cannot ignore when they are thrown. Such an exception is called a *checked exception*, because the compiler will complain if the method in which it can occur does not deal with it explicitly. Checked exceptions thus force the code in which they can occur to take explicit action to deal with them, resulting in programs that are *robust*, i.e. are able to handle error situations appropriately.

The Java standard library defines classes whose objects represent exceptions. Table 10.1 shows a selection of checked exceptions.

TABLE 10.1 Some checked exceptions

Checked exception class (in the `java.lang` package unless otherwise noted)	Description
`Exception`	This class represents the category of all checked exceptions.
`ClassNotFoundException`	Signals an attempt to load a class during execution, but the class cannot be found.
`java.io.IOException`	Signals an error during reading and writing of data. For example, the `read()` methods in the interface `InputStream` and the `write()` methods in the interface `OutputStream` throw this exception.
`java.io.EOFException`	Signals unexpected end of input. For example, the `read()` methods in the interface `InputStream` throw this exception.
`java.io.FileNotFoundException`	Signals an attempt to refer to a file that does not exist. For example, the constructors in the classes `FileInputStream`, `FileOutputStream` and `RandomAccessFile` throw this exception, if the file cannot be assigned.

The `Exception` class represents the category of all checked exceptions. The `ClassNotFoundException` is thrown when the class that a program uses cannot be found. You have mostly likely seen it when you misspelled the name of the class with the `main()` method on the command line while starting a program. The last three checked exceptions are in the `java.io` package and signal error situations that occur when dealing with files and streams (see Chapter 11 and Chapter 19). We will take a closer look at the main categories of exceptions in Section 18.1 on page 555.

Dealing with checked exceptions using the `throws` clause

A method that can throw a checked exception must satisfy *one* of the following two conditions. It must either:

1. Catch and handle the exception in a `try–catch` statement, as we have discussed earlier.
2. Allow further propagation of the exception with a `throws` clause specified in its method declaration, which we will discuss here.

A `throws` clause is specified in the method header, between the parameter list and the method body:

... *method name* `(...) throws` *exception class*$_1$`, ...,` *exception class*$_n$ `{ ... }`

Let us say that a method `B()` can throw a checked exception `K`, and that method `B()` chooses to propagate exception `K` further with the help of a `throws` clause in its method declaration. If a method `A()` calls method `B()`, then method `A()` must now take a stand on how to deal with exception `K`, because method `A()` can now indirectly throw exception `K` that it can receive from method `B()`. This means that each client of a method that can propagate a checked exception in a `throws` clause must decide how to deal with this exception. The compiler will check that a method that can throw a checked exception satisfies one of the two conditions listed above. If a checked exception that is specified in a `throws` clause is propagated to the top level (i.e. not caught by any `catch` block), it will be handled by a default exception handler in the usual way (Section 10.2).

Programming with checked exceptions

In Program 10.4, the `calculateSpeed()` method can throw a checked exception of the type `Exception` in an `if` statement:

```
if (distance < 0 || time <= 0)                          //(6)
  throw new Exception("distance and time must be > 0");
```

We can use a `throw` statement to throw an exception explicitly, by specifying the exception object to be thrown in the statement. In this case, we call the constructor of the exception class and pass a suitable message to explain the error situation. An object of the class `Exception`, with the string `"distance and time must be > 0"`, is thrown by the `throw` statement above. More details on using the `throw` statement can be found in Section 18.2 on page 557.

With the set-up in Program 10.4, the `calculateSpeed()` method must decide how to deal with a checked `Exception`. It chooses to throw this exception further in a `throws` clause,

(5). The `printSpeed()` method, which calls the `calculateSpeed()` method, must therefore take a stand on this checked exception as well. It also chooses to throw it further in a `throws` clause, (4). Since the `main()` method calls the `printSpeed()` method, the `main()` method must also decide what to do with this exception. The `main()` method chooses to catch and handle this exception in a `try–catch` block, (1) and (3). Any attempt to leave out this exception from the `throws` clauses at (5) and (6) will result in a compile-time error.

We will see several examples that use checked exceptions in the chapters on files and streams (Chapter 11 and Chapter 19).

PROGRAM 10.4 Handling checked exceptions

```
public class Speed6 {

  public static void main(String[] args) {
    System.out.println("Entering main().");
    try {                                                    //(1)
//    printSpeed(100, 20);                                   //(2a)
      printSpeed(-100,20);                                   //(2b)
    }
    catch (Exception exception) {                            //(3)
      System.out.println(exception + " (handled in main())");
    }
    System.out.println("Returning from main().");
  }

  private static void printSpeed(int kilometers, int hours)
                     throws Exception {                      //(4)
    System.out.println("Entering printSpeed().");
    double speed = calculateSpeed(kilometers, hours);
    System.out.println("Speed = " +
                       kilometers + "/" + hours + " = " + speed);
    System.out.println("Returning from printSpeed().");
  }

  private static int calculateSpeed(int distance, int time)
                    throws Exception {                       //(5)
    System.out.println("Calculating speed.");
    if (distance < 0 || time <= 0)                           //(6)
      throw new Exception("distance and time must be > 0");
    return distance/time;
  }
}
```

Output from Program 10.4 when (2a) is in the program and (2b) is commented out:

```
Entering main().
Entering printSpeed().
Calculating speed.
Speed = 100/20 = 5.0
Returning from printSpeed().
Returning from main().
```

Output from Program 10.4 when (2b) is in the program and (2a) is commented out:

```
Entering main().
Entering printSpeed().
Calculating speed.
java.lang.Exception: distance and time must be > 0 (handled in main())
Returning from main().
```


10.5 Unchecked exceptions

Since Java provides checked exceptions, it begs the question: does Java provide *unchecked exceptions*? The answer is "yes". Unchecked exceptions are exceptions typically concerning unforeseen errors, such as *programming errors*. Table 10.2 shows some common unchecked exceptions that you are likely to come across when you program in Java. We have already seen one example where an `ArithmeticException` and an `IllegalArgumentException` can be thrown (Program 10.3).

In contrast to checked exceptions, the compiler does *not* check whether unchecked exceptions can be thrown. This means that a method does not have to deal with unchecked exceptions – but it must then face the consequences if the program is terminated due to an unchecked exception.

The best solution for handling such situations is to correct the cause of the errors in the program so that they do *not* occur during program execution. For example, selection statements or assertions can be used to check the conditions that a method imposes on its actual parameters, before we call the method.

Since the compiler ignores unchecked exceptions and a method is not forced to handle them, such exceptions are usually not specified in the `throws` clause of a method.

TABLE 10.2 Some unchecked exceptions

Unchecked exception class (in the `java.lang` package)	Description
`RuntimeException`	This class represents one category of unchecked exceptions.
`NullPointerException`	Signals an attempt to use a reference that has the value `null`, i.e. the reference does not refer to an object. For example, the expression `new String(null)` throws this exception, since the parameter has the value `null`, instead of being a reference to an object.
`ArithmeticException`	Signals an illegal arithmetic operation, for example integer division with 0, e.g. `10/0`.

`ClassCastException`	Signals an attempt to convert an object's reference value to a type to which it does not belong. For example:
	`Object ref = new Integer(0);`
	`String str = (String) ref; // Integer is not String.`

`IllegalArgumentException`	Signals an attempt to pass an illegal actual parameter value in a method call.
`NumberFormatException`	Indicates a problem converting a value to a number, for example, an attempt to convert a string with characters that cannot constitute a legal integer, e.g. `Integer.parseInt("4u2")`.
`IndexOutOfBoundsException`	Signals than an index value is not valid.
`ArrayIndexOutOfBoundsException`	Signals than an index value is not valid. The index value in an array is either less than `0` or greater than or equal to the array length, e.g. `array[array.length]`.
`StringIndexOutOfBoundsException`	Signals than an index value is not valid. The index value in a string is either less than `0` or greater than or equal to the string length, e.g. `str.charAt(-1)`.
`AssertionError`	Indicates that the condition in an `assert` statement has evaluated to the value `false`, i.e. the assertion failed. See Section 3.4 on page 64 and Section 14.3 on page 398.

BEST PRACTICE

The `AssertionError` should never be caught in a `catch` block as it signals that an assertion about the program does not hold and, therefore, the logic of the program must be corrected.

10.6 Review questions

1. Execution handling in Java is built on the principle of _____ and _____.
2. The sequence in which actions are executed determines which exceptions are thrown and how these will be handled. This is called the _____ of the program.
3. Local variables for a method call are stored on a _____ during execution.
4. If a method `A()` calls a method `B()`, the execution of method _____ will complete before method _____.
5. What is meant by a method being active during execution?
6. What is wrong with the following `try–catch` statement?

```
try
  int i = Integer.parseInt("onetwothree");
catch (NumberFormatException x)
  System.out.println("Not a valid integer.");
```

7. Which statements are true about the `try–catch` statement?
   - **a** A `catch` block must have a corresponding `try` block.
   - **b** The parameter list of a `catch` block can be empty, then the `catch` block can catch any exception.
   - **c** The parameter list of a `catch` block always has only one parameter that specifies the type of exceptions the `catch` block can catch.
   - **d** A `try` block can contain code that does not throw an exception.
8. Which statements are true when an exception occurs?
   - **a** If the exception is not caught and handled by a `catch` block, it will be handled by a default exception handler.
   - **b** If the exception is caught and handled by a `catch` block, normal execution will continue.
   - **c** The execution of the `try` block is terminated, regardless of whether the exception is later caught or not.
   - **d** Normal execution can never be resumed after an exception has occurred in a `try` block.

9. A method can specify exceptions that it will rethrow in a _____ clause in the method header.

10. Which statements about checked exceptions are true?

    a Checked exceptions that can be thrown in a method must be listed in a `throws` clause if the exceptions are not caught and dealt with in the method body.

    b A method that calls another method that has a `throws` clause with checked exceptions need not deal with these exceptions.

    c A method that calls another method that has a `throws` clause with checked exceptions must explicitly deal with these exceptions.

11. Which statement is true about checked exceptions?

    a Checked exceptions that can be thrown in a method must be listed in a `throws` clause of the method if these exceptions are not handled by a `try–catch` statement in the method body.

    b A method that calls another method that has a `throws` clause with checked exceptions need not handle these exceptions.

    c Since the compiler checks the use of checked exceptions, such exceptions are not handled by a default exception handler.

12. Which exceptions will the following code throw during execution?

    a `String str = null; int in = str.length();`

    b `Object objRef = new String("aba"); Integer intRef = (Integer) objRef;`

    c `int[] array = new int[10]; array[array.length] = 100;`

    d `int j = Integer.parseInt("1two1");`

13. A method specifies checked exceptions it can throw to its caller in a _____ clause.

14. Which code will compile?

    a `try { }`

    b `try { } catch(Exception x) { }`

    c `catch(Exception y) { }`

    d `catch(Exception y) { } try { }`

10.7 Programming exercises

1. Write a class `Temperature` that converts temperature between Fahrenheit (`F`) and Celsius (`C`). The formulas for conversion between the two units are:
   - fahrenheit = (9.0 * celsius) / 5.0 + 32.0
   - celsius = (fahrenheit – 32.0) * 5.0 / 9.0

The program reads the degrees from the command line, for example:

```
> java Temperature 50.0 C
50.0 C is 122.0 F
> java Temperature 122.0 F
122.0 F is 50.0 C
```

The program should throw an exception of the type `NumberFormatException` if the number of degrees is not a legal floating-point number. The program should check the data from the command line and give suitable feedback.

2. Write a class `Area` that calculates the area of a rectangle, using the following formula:

*area = length * breadth*

The program reads the length (`L`) and the breadth (`B`) from the command line, for example:

```
> java Area L 10.5 B 5.0
Length: 10.5
Breadth: 5.0
Area: 52.5
> java Area B 5.0 L 10.5
Length: 10.5
Breadth: 5.0
Area: 52.5
```

The program should throw a `NumberFormatException` if the sides are not legal values. The program should check the information read from the command line and give suitable feedback.