

# Chapter 1

## Exception Handling

Lecture slides for:

*Java Actually: A Comprehensive Primer in Programming*

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cengage Learning, 2008.

ISBN: 978-1-844480-933-2

<http://www.i.i.uib.no/~khalid/jac/>

*Permission is hereby granted to use these lecture slides in conjunction with the book.*

*Modified: 16/2/19*

## Overview

- 
- Program Execution
  - *Throw-and-catch* principle
  - Exception Propagation
  - Typical scenarios when using the try-catch statement
  - Using the throws clause with checked exceptions
  - Unchecked exceptions
-

## What is an exception?

- An *exception* in Java signals an error situation that can occur during program execution.
- Some examples of error situations:
  1. Programming errors -- occur because of logical errors.
    - Illegal array index
    - Integer division by 0
    - Method call with illegal parameters
    - Using a reference with the `null` value to access members of an object
  2. Runtime error -- programmer has little control over.
    - Opening a file that doesn't exist.
    - Read or write errors when using a file.
    - A network connection that goes down.

## Program execution

- A *program stack* is used to manage the execution of methods.
- An *element* (called the *stack frame*) on the program stack corresponds to a method call.
  - Each method call results in the creation of a new stack frame.
  - Stack frame contains various information, including storage for local variables.
  - When the method is complete, its frame is removed from the program stack.
  - The method that has the stack frame on the top of the program stack is executed.
- When returning from a method call, program execution continues with the method in the corresponding stack frame that now has been revealed on top of the program stack.
- During execution, all stack frames on the program stack at any given time, specify which methods are *active*, i.e., have not finished executing.

## Method Execution and Exception Propagation

- We shall use the following problem to illustrate the method of execution and exception propagation.

Calculate the speed when the distance and time are given.

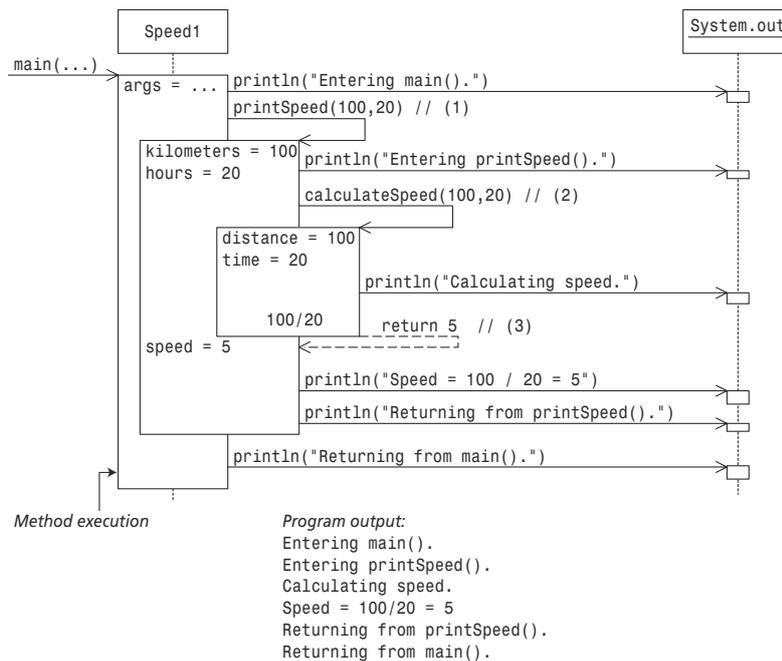
The program uses three methods:

1. `main()`
  2. `printSpeed()`
  3. `calculateSpeed()`
- Note integer division in the calculation of expression (`distance / time`).
    - Integer division by 0 is an *illegal* operation in Java.

## Method Execution (Program 10.1)

```
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)  
    }  
}
```

## Normal execution (Figure 10.1)



## Throw-and-catch principle

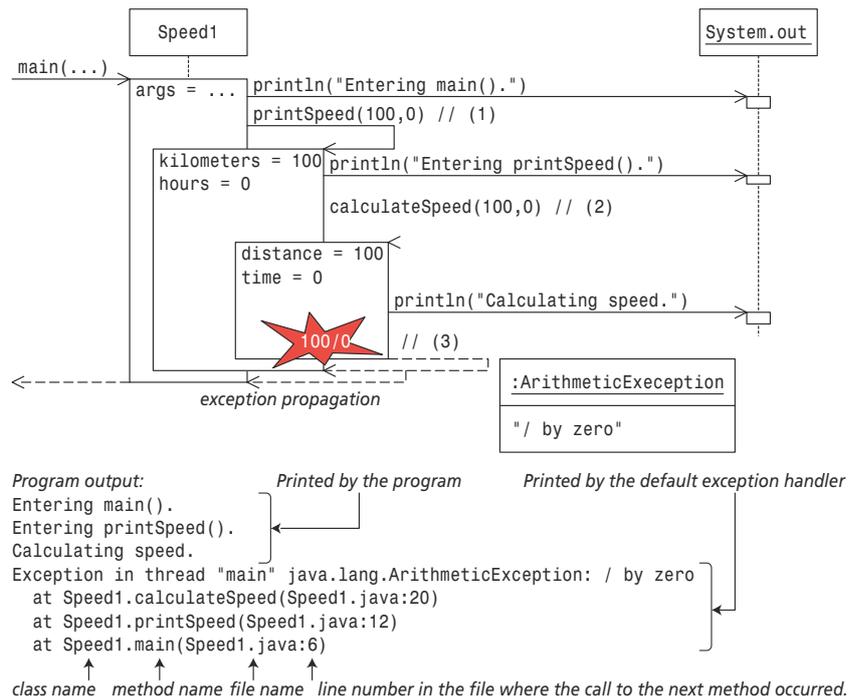
- An exception is *thrown* when an error situation arises during program execution, and is *caught* by an exception handler that handles it.
- Replace in Program 10.1:  

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

 with  

```
printSpeed(100, 0); // (1) Second parameter is 0.
```
- Execution of the program is illustrated in the sequence diagram in Figure 10.2.
- In (7) an ArithmeticException is thrown-exception, which is communicated back (*propagated*) through the stack frames of the program stack.

## Exception Propagation (integer division by 0) (Figure 10.2)

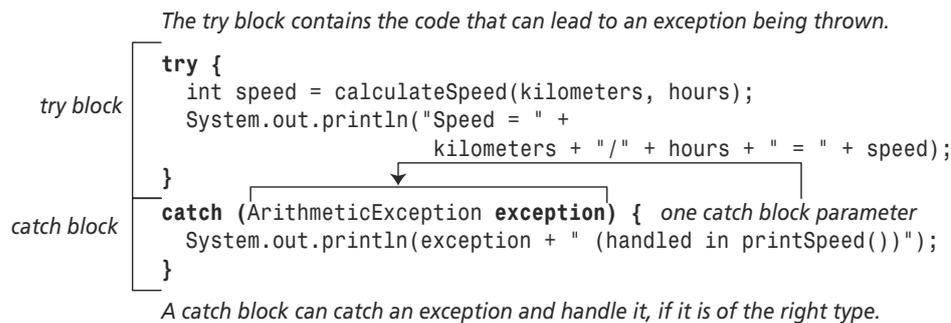


## Exception Propagation

- Program execution does not continue normally during unntakspropageringen, and the exception is not returned by the return statement.
- The exception is offered to active methods in turn.
  - If an active method does not catch the exception, the execution of the method is interrupted, i.e. its stack frame is removed from the program stack.
- When the exception propagates to the "top level", it is processed by a standard exception handler in the Java virtual machine.
- *Standard exception handler* generates a stack trace at the terminal.

## Exception Handling: try-catch

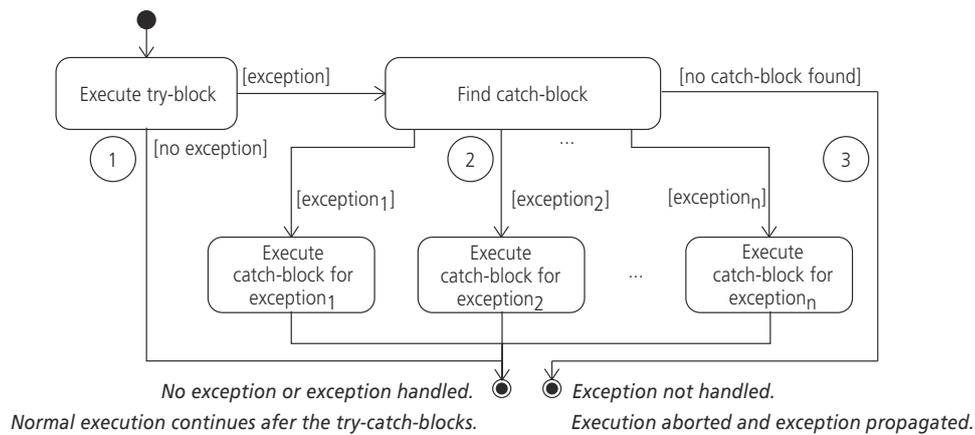
- A `try` block can contain arbitrary code, but the purpose is to add code that might throw an exception during execution.
- A `catch` block is an *exception handler*.
  - A `catch` block belongs with a `try` block.
  - An exception that can be thrown as a result of executing the code in the `try` block, can be caught and handled in a corresponding `catch` block.



## Typical scenarios when using a try-catch statement

1. The code in the `try` block is executed, and no exception is thrown.
  - Normal execution continues after the `try-catch` blocks.
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.
  - The execution of the `try`-block is cancelled, such that the actions in the rest of the `try` block are not executed.
  - Normal execution continues after the `try-catch` blocks.
3. The code in the `try` block is executed, and an exception is thrown, but no `catch` block is found to handle the exception.
  - The execution of the `try`-block is cancelled, such that the actions in the rest of the `try` block are not executed.
  - The exception is propagated.

## try-catch scenarios (Figure 10.4)



## try-catch scenario 1

- In Program 10.2, the `printSpeed()` method uses a try-catch statement, (2).
- The corresponding catch block, (4), is declared to catch exceptions of type `ArithmeticException`.
- Execution of Program 10.2 is shown in Figure 10.5.

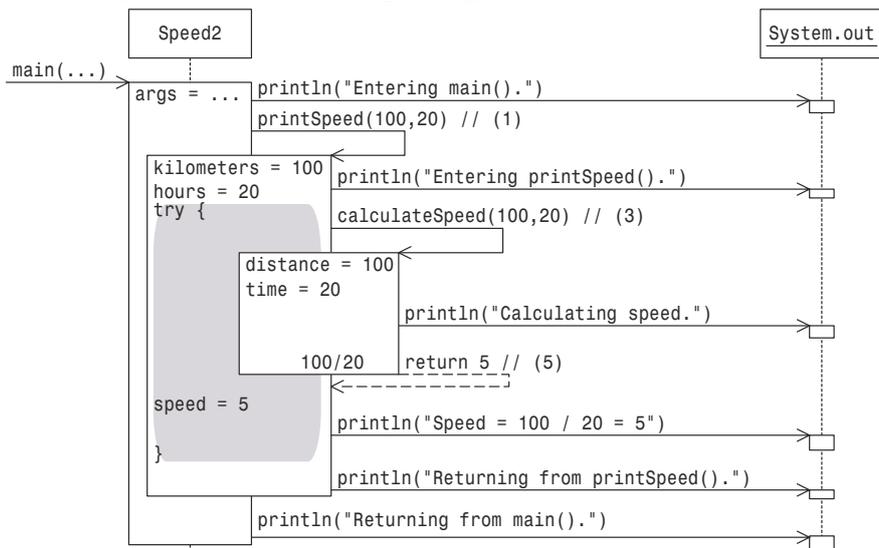
## Exception handling (Program 10.2)

```
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)
    }
}
```

## Exception handling (Figure 10.5) (scenario 1)

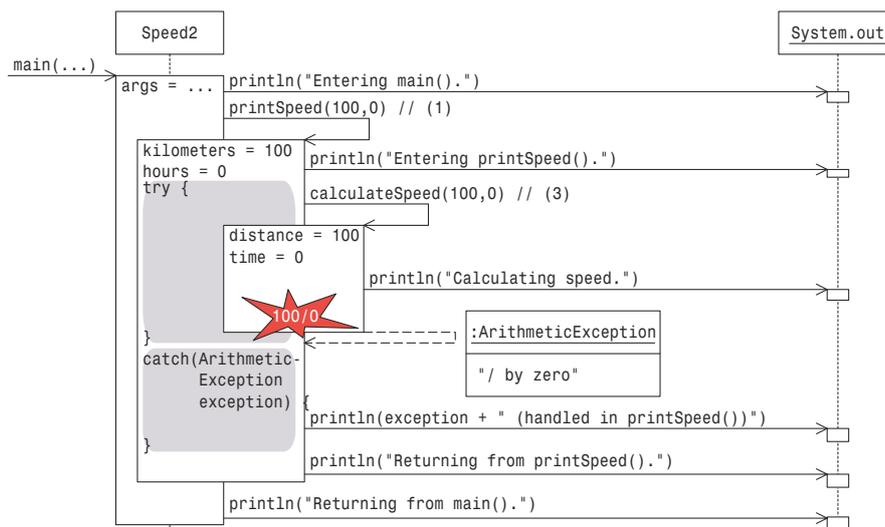


Program output:  
Entering main().  
Entering printSpeed().  
Calculating speed.  
Speed = 100/20 = 5  
Returning from printSpeed().  
Returning from main().

## try-catch scenario 2

- Replace in Program 10.2:  
`printSpeed(100, 20); // (1)`
- with  
`printSpeed(100, 0); // (1) Second parameter is 0.`
- Integer Division by 0 causes an `ArithmeticException` exception to be thrown at (5) in the method `calculateSpeed()`.
  - Execution of this method is interrupted and the exception is propagated.
  - It is caught by the catch block in method `printSpeed()`.
  - After handling the exception, normal execution of the program is restored.
  - Printout shows the normal execution of the `printSpeed()` method from this point.

## Exception handling (Figure 10.6) (scenario 2)



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

- Scenario 3 shows what happens when an exception is thrown during the execution of a try-block and there is no corresponding catch block to handle the exception.
- The program's execution behavior in Figure 10.7 shows that the integer division with 0 again leads to an `ArithmeticException`-exception, thrown at (7) in the method `calculateSpeed()`.
  - Execution of this method is interrupted and the exception is propagated.
- `ArithmeticException` is *not* caught by the catch block in `printSpeed()` method, since this catch block can only handle exceptions of type `IllegalArgumentException`.
- Execution of the `printSpeed()` method is interrupted, and the exception is propagated.
- The exception `ArithmeticException` is now caught by the catch block in method `main()` at (3).
  - After handling the exception in the catch block in method `main()`, normal execution of the program is restored.
  - Printout shows the normal execution of the method `main()` from this point.

### Exception handling (Program 10.3) (scenario 3)

```
public class Speed3 {
    public static void main(String[] args) {
        System.out.println("Entering main().");
        try {
            printSpeed(100,20);
        }
        catch (ArithmeticException exception) {
            System.out.println(exception + " (handled in main())");
        }
        System.out.println("Returning from main().");
    }

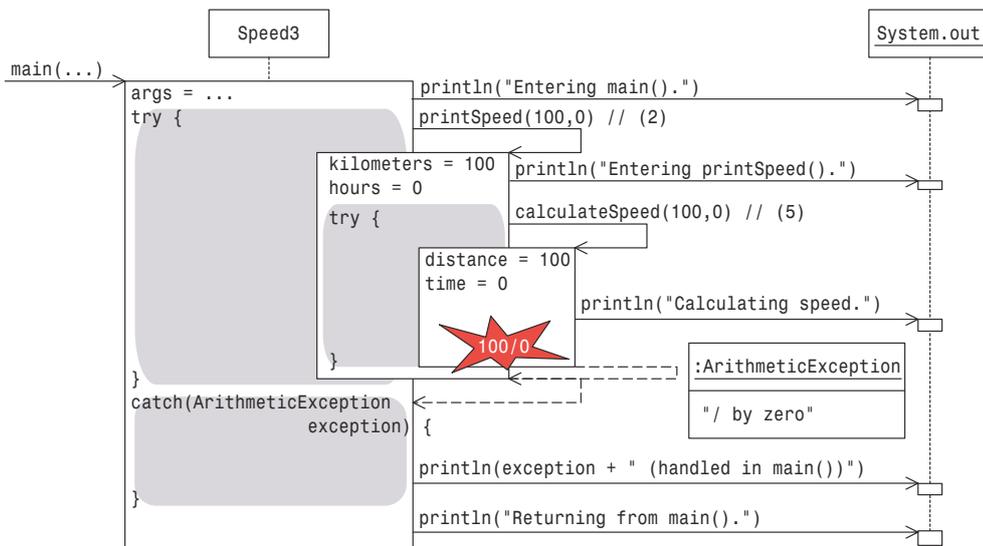
    private static void printSpeed(int kilometers, int hours) {
        System.out.println("Entering printSpeed().");
        try {
            int speed = calculateSpeed(kilometers, hours);
            System.out.println("Speed = " +
                kilometers + "/" + hours + " = " + speed);
        }
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

```

### Exception handling (Figure 10.7) (scenario 3)



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

## Handling of controlled exceptions

- The purpose of the exception handling is 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 something about it.
  - This results in applications that are not very *robust*.
- Use of controlled exception provides a strategy in which a method, that can throw such an exception, is forced to decide how the exception should be handled:
  1. Catch and handle the exception in a try-catch statement.
  2. Allow further propagation of the exception with a throws-clause specified in the method declaration.
- A throws clause is specified in the method header, between the parameter list and the method body:  
`... method name (...) throws exception class1, ..., exception classn {...}`

## Handling of controlled exceptions

- Given that the method foo() calls the method bar():  
`void bar () throws { /*...*/ } // K is a checked exception.`  
`void foo () { bar (); } // Compile-time error.`
- Solution:  
Either  
`void foo () throws K { bar (); } // Throws the exception further.`  
or:  
`void foo () {  
 try { bar (); }  
 catch (K k) { k.printStackTrace (); } // Catch and handle the exception.  
}`
- The compiler will check that a method that can throw a checked exception meets one of the above requirements.
- This means that any client of a method that can propagate a checked exception in a throws clause, must decide how this exception should be handled.
- If a checked exception specified in a throws clause propagates all the way to the top level, it will be processed by a standard exception handler in the normal way.

## Example: checked exceptions (Program 10.4)

- The method `calculateSpeed()` explicitly throws an `Exception` in an `if` statement, (6), using the `throw` statement.
  - This exception is further propagated in a `throws`-clause (5).
- The method `printSpeed()`, which calls the method `calculateSpeed()`, also chooses to propagate the exception in a `throws`-clause (4).
- The method `main()` that calls the `printSpeed()` method, chooses to catch and handle this exception in a `try-catch` block, (1) and (3).
  - Normal execution continues after the exception is handled.
- Attempts to exclude the `throws`-clauses will result in compile-time errors.

## Handling checked exceptions (Program 10.4)

```
public class Speed6 {  
  
    public static void main(String[] args) {  
        System.out.println("Entering main().");  
        try {  
            // printSpeed(100, 20);           //(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;
}
}

```

- Running the program with (2a):

```

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

```

- Running the program with (2b):

```

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

```

## Unchecked Exceptions

- Unchecked exceptions are exceptions that the compiler does not check, i.e. the compiler does not care whether the code deals with them or not.
- Unchecked exceptions are usually the result of programming errors and should be corrected.
- The unchecked `AssertionError` should never be caught as it thrown by the `assert` statement.
- See Table 10.2.