IT602: Object-Oriented Programming

Lecture - 10

# Exception Handling

Arpit Rana

24th Feb 2022

## Exception

In Java, an ***exception*** signals the occurrence of an error situation due to the ***violation of some semantic constraint*** of the Java programming language.

- A requested file cannot be found,
- An array index is out of bounds, or
- A network link failed

## Stack-based Execution

Several *threads* can be executing at the same time in the JVM.

- Each *thread* has its own ***JVM stack*** to handle the execution of methods
- Each element on the JVM stack is called an ***activation frame*** which corresponds to a method call
- An activation frame on top of the JVM stack is the one currently executing and is popped as the execution finishes.

At any given time, the active methods on a JVM stack make up a stack trace of a thread’s execution.

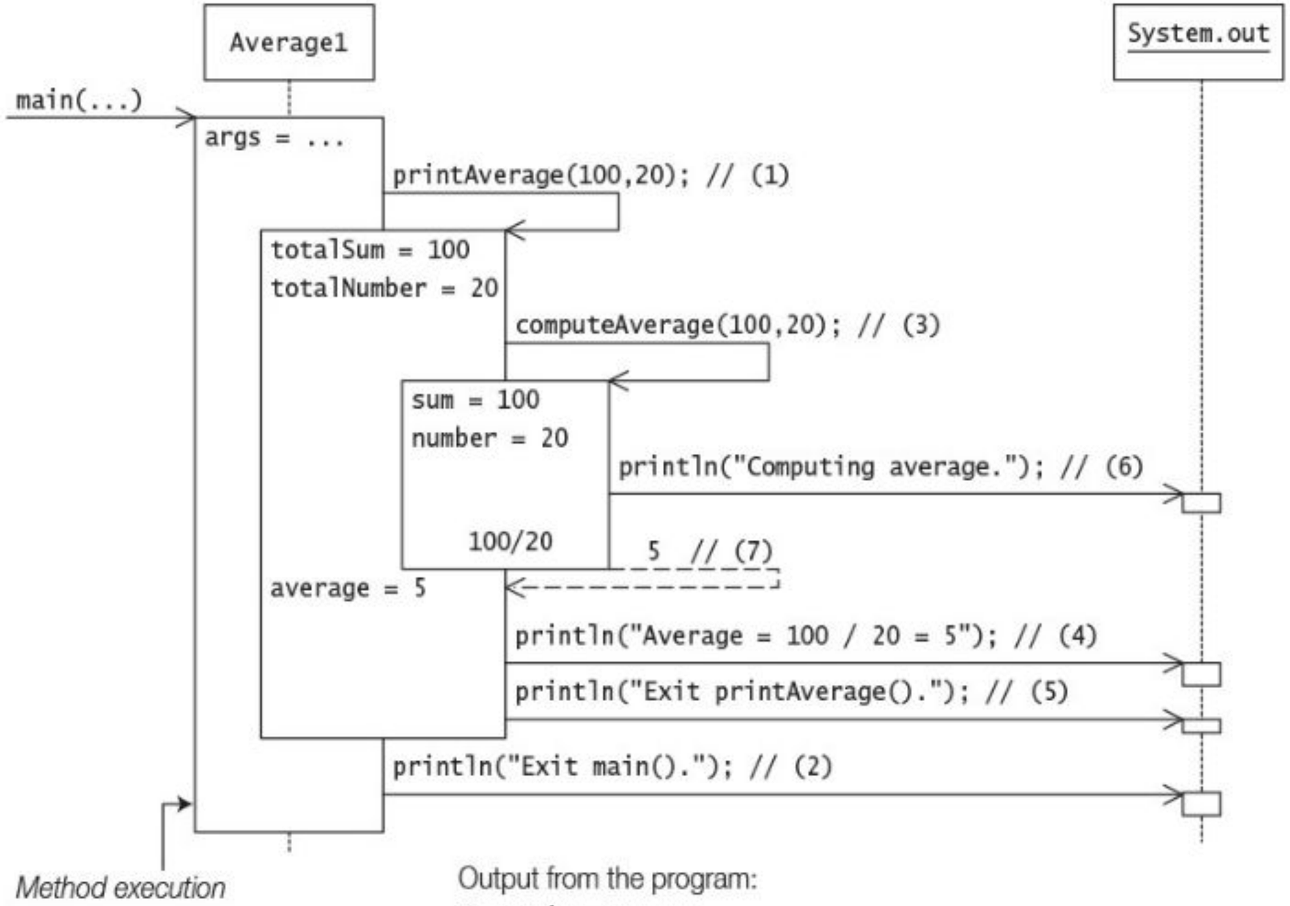
## Stack-based Execution

```
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 totalNumber) {
    int average = computeAverage(totalSum, totalNumber);                 // (3)
    System.out.println(“Average = ” +                                    // (4)
        totalSum + ” / ” + totalNumber + ” = ” + average);
    System.out.println(“Exit printAverage().”);                          // (5)
  }

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


Average1
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
computeAverage(100,20); // (3)
sum = 100
number = 20
println("Computing average."); // (6)
100/20
5 // (7)
average = 5
println("Average = 100 / 20 = 5"); // (4)
println("Exit printAverage()."); // (5)
println("Exit main()."); // (2)
Method execution

Output from the program:

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

## Stack-based Execution

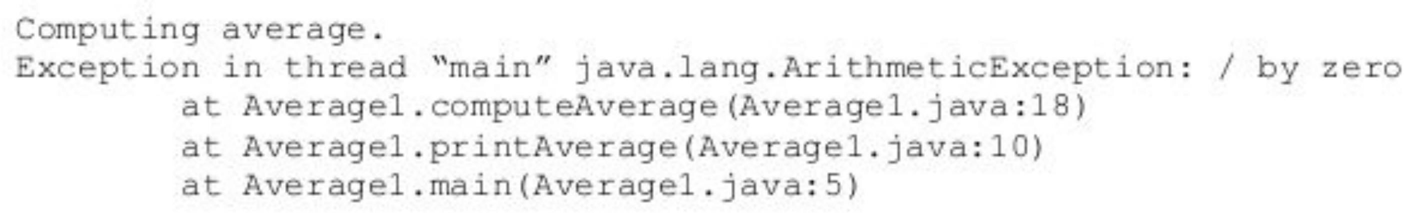
If the method call at (1) in the previous example i.e.

```
    printAverage(100, 20)
```

is replaced with

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

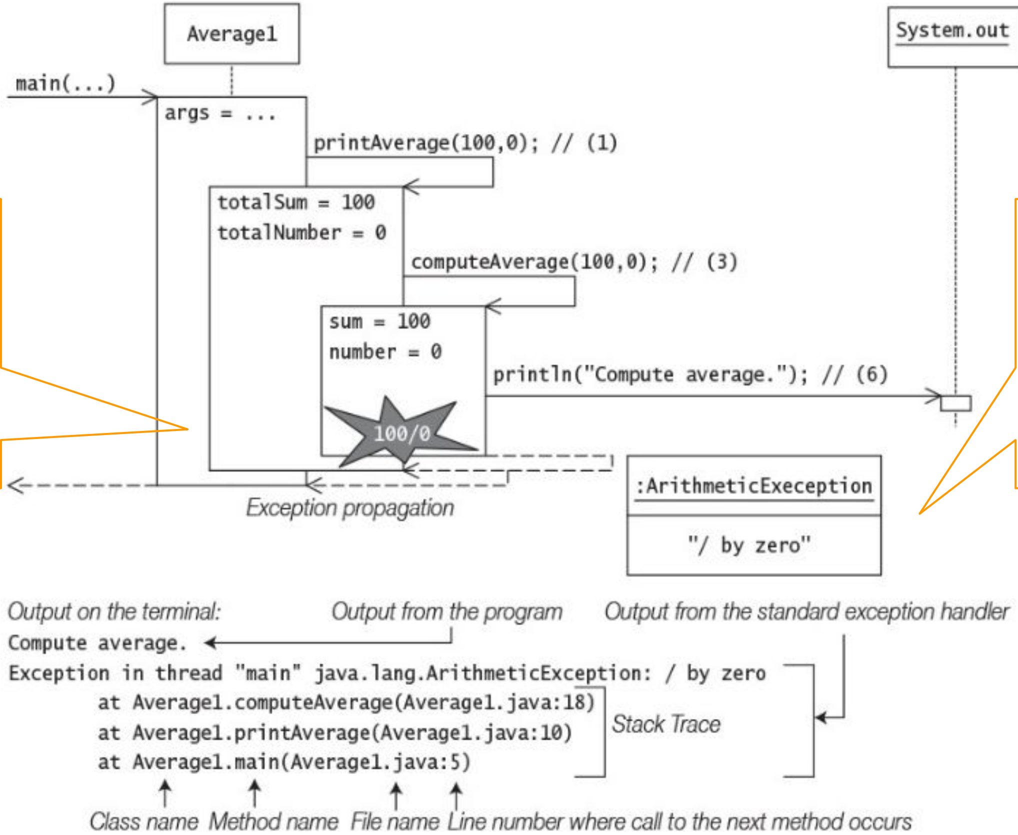
and the program is run again, the output is as follows.



```
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)
```

## Exception Propagation

An exception is ***thrown*** to signal that an unexpected condition has occurred.



```
Compute 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)
```

An exception is ***caught*** to take an appropriate action to deal with the exception.

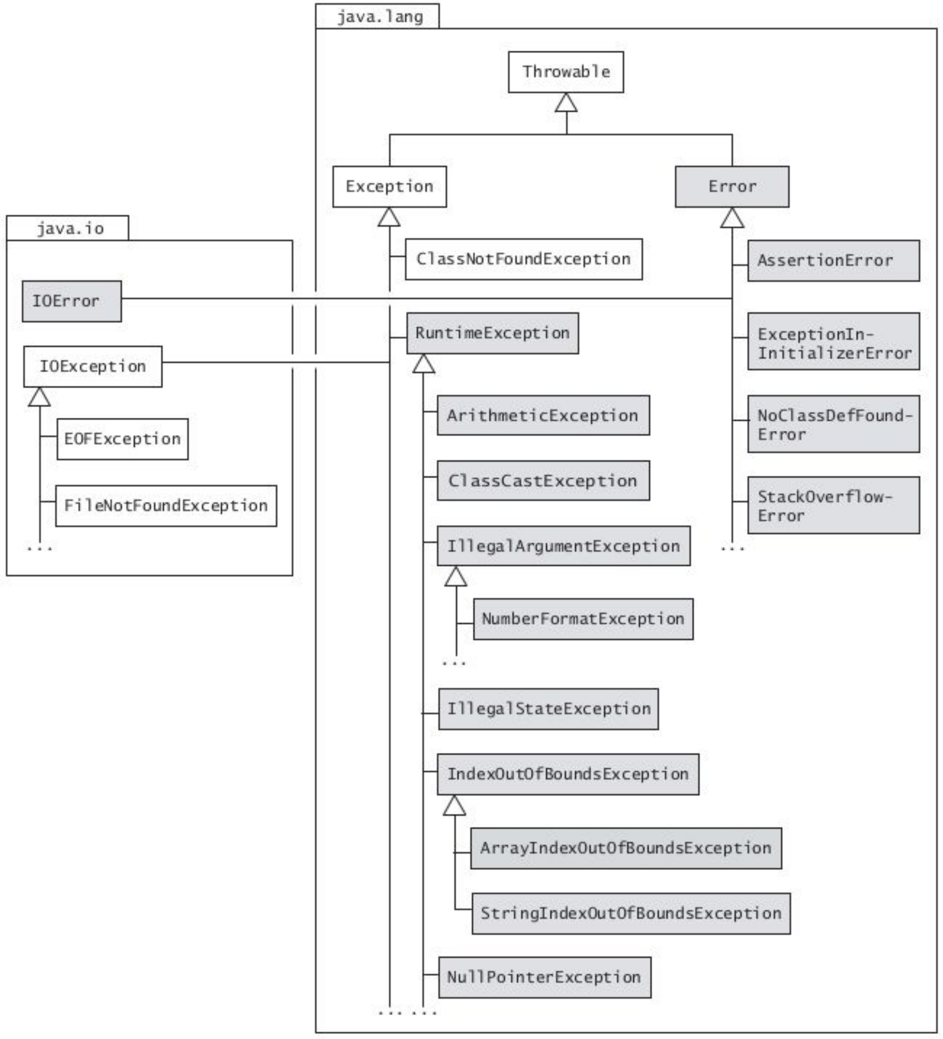
## Exception Types

Exceptions in Java are objects.

All exceptions are derived from the `java.lang.Throwable` class.

It has the following common methods:

- String getMessage()
- void printStackTrace()
- String toString()



Classes that are shaded (and their subclasses) represent unchecked exceptions.

## The Exception Class

The class Exception represents exceptions that a program would normally want to catch. It has the following subclasses among others.

- `ClassNotFoundException`
- `RuntimeException`
- `IOException` (e.g. `FileNotFoundException`)
- `SQLException`

## The ClassNotFoundException Class

The subclass `ClassNotFoundException` signals that -

- the JVM tried to load a class by its string name, but the class could not be found.
- e.g. class name is misspelled

## The RuntimeException Class

The runtime exceptions are usually caused by program bugs (faults in the program design)

We should let them be handled by the default exception handler.

- `ArithmeticException`: represents an illegal arithmetic operation, e.g. division by 0.
- `ArrayIndexOutOfBoundsException`: indicates an error in which an invalid index is used to access an element in the array.
- `ClassCastException`: signals that an attempt was made to cast a reference value to a type that was not legal,
  e.g. casting an `Integer` object to the `Long` type.
- `IllegalArgumentException`: is thrown to indicate that a method was called with an inappropriate argument.

## The RuntimeException Class

The runtime exceptions are usually caused by program bugs (faults in the program design)

We should let them be handled by the default exception handler.

- `NumberFormatException` (extends `IllegalArgumentException`): is thrown while converting a string to a numeric value and the format of the characters in the string is not appropriate.
- `NullPointerException`: is thrown when an attempt is made to use the `null` value as a reference value to refer to an object.

## The Error Class

The Error class define errors that are never explicitly caught and are usually irrecoverable.

These errors are signaled by the JVM.

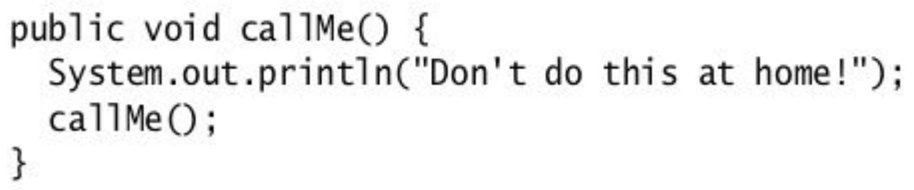
- `NoClassDefFoundError`: indicates that an application needs a class, but no definition of the class could be found. The reasons could be -
    - the name of the class might be misspelled in the command line,
    - the CLASSPATH might not specify the correct path, or
    - the class file with the bytecode is no longer available.

## The Error Class

The Error class define errors that are never explicitly caught and are usually irrecoverable.

These errors are signaled by the JVM.

- `StackOverflowError`: indicates that the JVM stack has no more room for new method activation frames.
    - This situation can occur when method execution in an application recurses too deeply.



```
public void callMe() {
  System.out.println("Don't do this at home!");
  callMe();
}
```

## Checked and Unchecked Exceptions

### Checked Exceptions

Except for `RuntimeException, Error`, and their subclasses, all exceptions are called ***checked*** exceptions.

- The compiler ensures that if a method can throw a checked exception, directly or indirectly, the method must explicitly deal with it.
- The method must either catch the exception and take the appropriate action, or pass the exception on to its caller.

## Checked and Unchecked Exceptions

### Unchecked Exceptions

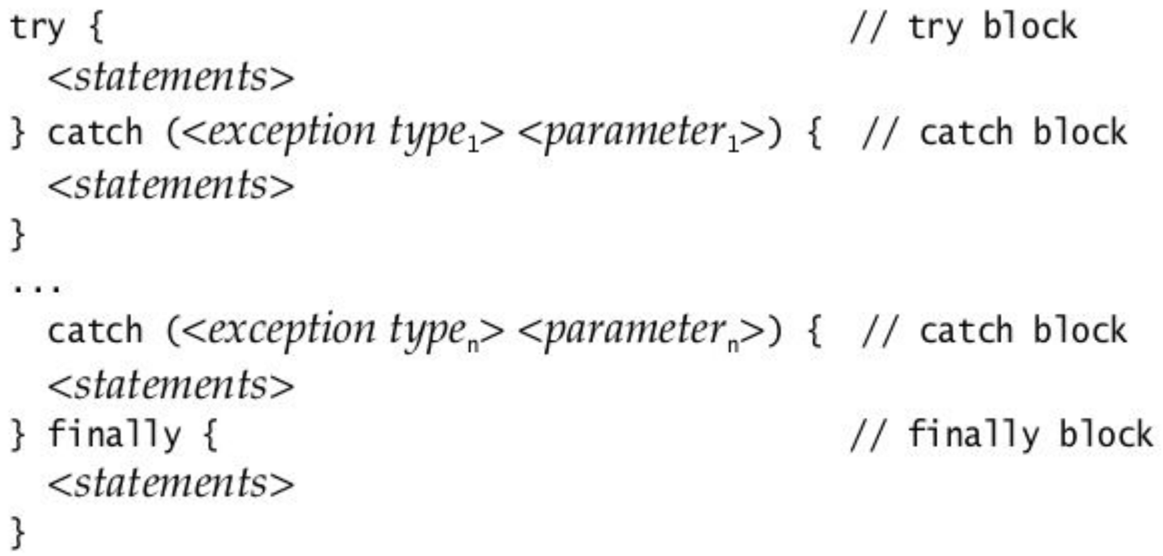
Exceptions defined by `Error` and `RuntimeException` classes and their subclasses are known as ***unchecked*** exceptions.

It means that a method is not obliged to deal with these kinds of exceptions.

- They are either irrecoverable (exemplified by the `Error` class) and the program should not attempt to deal with them, or
- They are programming errors (exemplified by the `RuntimeException` class)

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

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



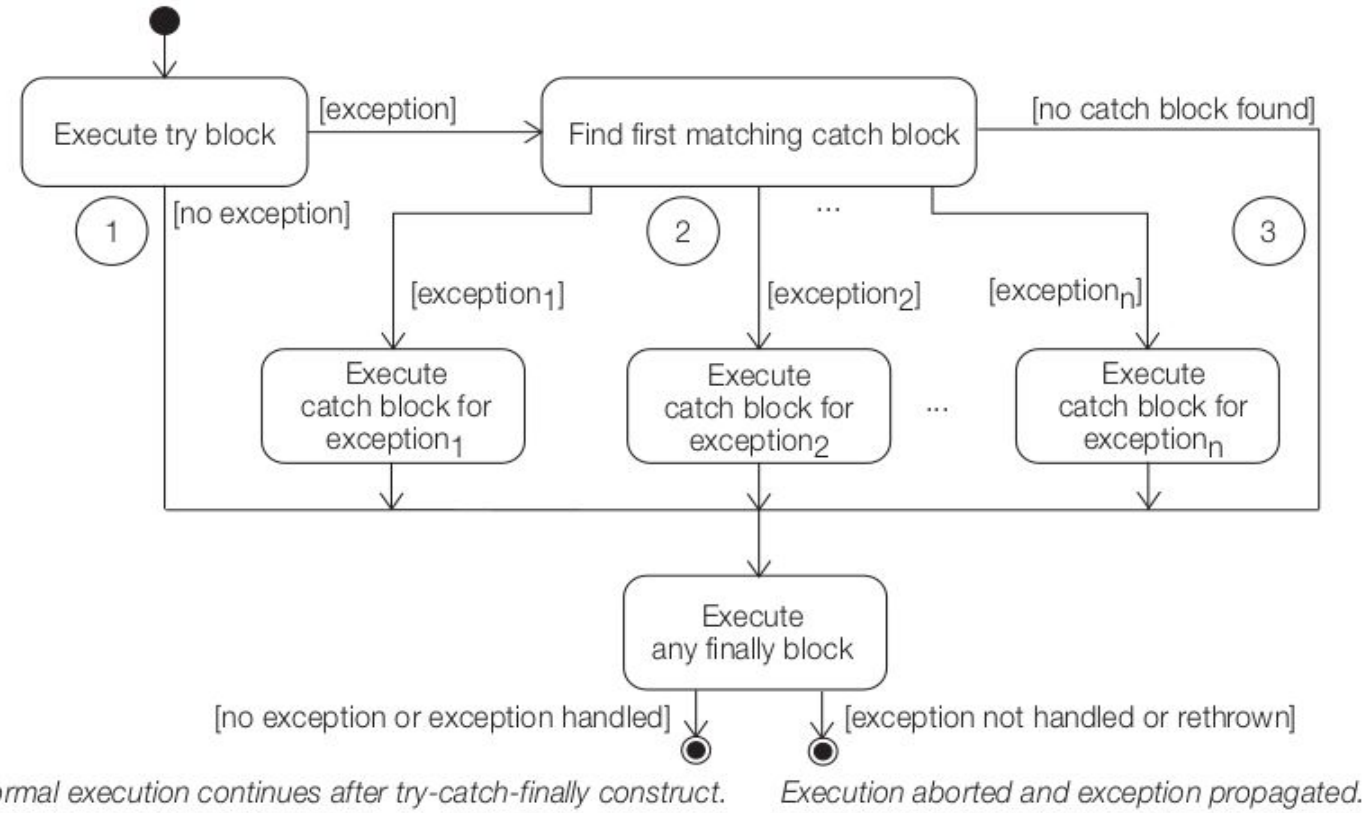
```
try {                                              // try block
  <statements>
} catch (<exception type_1> <parameter_1>) {   // catch block
  <statements>
}
...
  catch (<exception type_n> <parameter_n>) {   // catch block
  <statements>
} finally {                                        // finally block
  <statements>
}
```

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

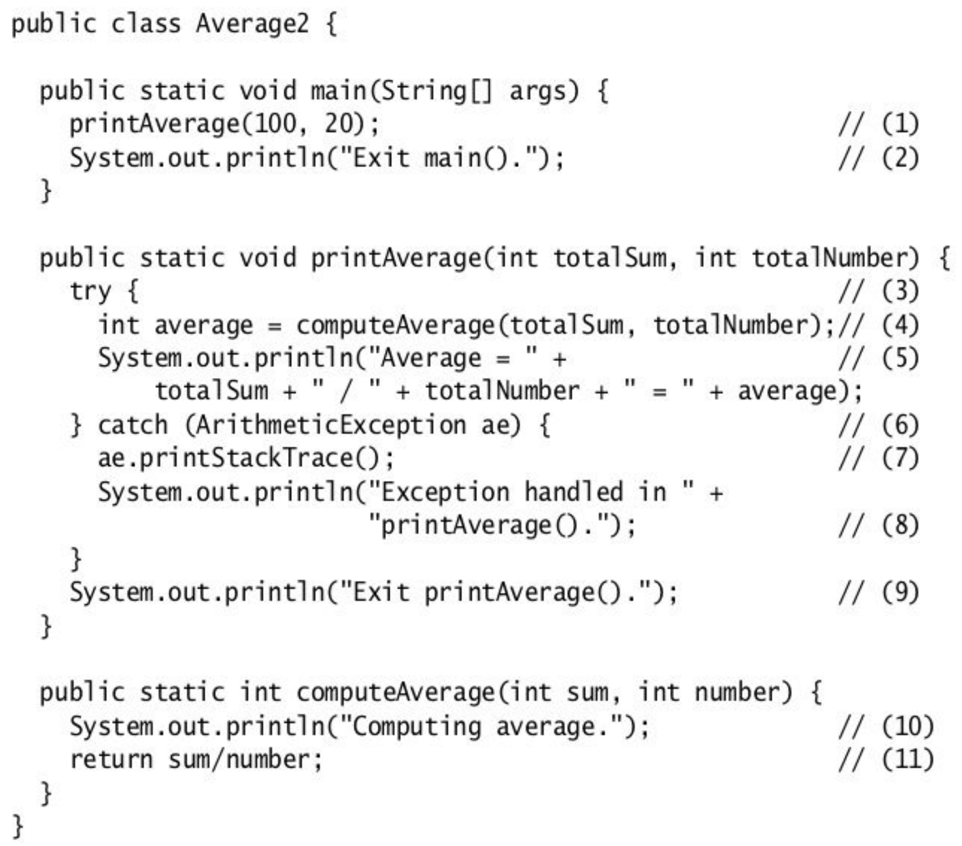
The mechanism for handling exceptions is embedded in the `try-catch-finally` construct, which has the following general form:

- For each `try` block there can be zero or more `catch` blocks, but only one `finally` block.
- Each `catch` block defines an exception handler.
- The header of the `catch` block takes exactly one argument, which is the exception the block is willing to handle.
- The exception must be of the `Throwable` class or one of its subclasses.
- A `finally` block is guaranteed to be executed, regardless of the cause of exit from the `try` block, or whether any `catch` block was executed.

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


Execute try block
[exception]
Find first matching catch block
[no catch block found]
1
[no exception]
2
...
3
[exception$_1$]
[exception$_2$]
[exception$_n$]
Execute catch block for exception$_1$
Execute catch block for exception$_2$
...
Execute catch block for exception$_n$
Execute any finally block
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct.
Execution aborted and exception propagated.

## The `try-catch` construct



```
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 totalNumber) {
    try {                                                    // (3)
      int average = computeAverage(totalSum, totalNumber);// (4)
      System.out.println("Average = " +                      // (5)
          totalSum + " / " + totalNumber + " = " + 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 number) {
    System.out.println("Computing average.");                // (10)
    return sum/number;                                       // (11)
  }
}
```

## The `try-catch` construct

```
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)
  }
```

## The `try-catch` construct

```
  public static void printAverage(int totalSum, int totalNumber) {
    try {                                                        // (7)
      int average = computeAverage(totalSum, totalNumber);// (8)
      System.out.println("Average = " +                          // (9)
          totalSum + " / " + totalNumber + " = " + 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 number) {
    System.out.println("Computing average.");                    // (14)
    return sum/number;                                           // (15)
  }
}
```

## The order of `catch` block

The **`javac`** compiler also complains if a `catch` block for a superclass exception shadows the `catch` block for a subclass exception, as the `catch` block of the subclass exception will never be executed.

The following example shows incorrect order of the `catch` blocks at (1) and (2), which will result in a *compile time error*:

- The superclass `Exception` will shadow the subclass `ArithmeticException.`

```
...
// Compiler complains
catch (Exception e) {                        // (1) superclass
  System.out.println(e);
} catch (ArithmeticException e) {            // (2) subclass
  System.out.println(e);
}
...
```

## The `try-finally` construct

```
public class Average6 {

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

  public static int printAverage(int totalSum, int totalNumber) {
    int average = 0;
    try {                                                      // (3)
      average = computeAverage(totalSum, totalNumber);         // (4)
      System.out.println("Average = " +                        // (5)
          totalSum + " / " + totalNumber + " = " + average);
      return average;                                          // (6)
    } finally {                                                // (7)
      System.out.println("Finally done.");
      return average*2;                                        // (8)
    }
  }

  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                  // (9)
    return sum/number;                                         // (10)
  }
}
```

## The `throw` Statement

We have seen how an exception can be thrown implicitly by the JVM during execution.

The `throw` statement is used to programmatically throw an exception.

`throw` *<object reference expression>*;

- The compiler ensures that the *<object reference expression>* is of the type `Throwable` class or one of its subclasses.
- At runtime a `NullPointerException` is thrown by the JVM if the *<object reference expression>* is `null`.

## The `throw` Statement

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

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

- When an exception is thrown, normal execution is suspended.
- The runtime system proceeds to find a `catch` block that can handle the exception.
- If no handler is found, then the exception is dealt with by the default exception handler at the top level.

## The `throw` Statement

```
public class Average7 {

  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 " +          // (5)
      "main().");
    } finally {
      System.out.println("Finally in main().");             // (6)
    }
    System.out.println("Exit main().");                     // (7)
  }

  public static void printAverage(int totalSum, int totalNumber) {
    try {                                                   // (8)
      int average = computeAverage(totalSum, totalNumber);  // (9)
      System.out.println("Average = " +                     // (10)
          totalSum + " / " + totalNumber + " = " + average);
    } catch (IllegalArgumentException iae) {                // (11)
      iae.printStackTrace();                                // (12)
      System.out.println("Exception handled in " +          // (13)
      "printAverage().");
    } finally {
      System.out.println("Finally in printAverage().");     // (14)
    }
```

## The `throw` Statement

```
      System.out.println("Exit printAverage().");                 // (15)
    }

    public static int computeAverage(int sum, int number) {
      System.out.println("Computing average.");
      if (number == 0)                                              // (16)
        throw new ArithmeticException("Integer division by 0");    // (17)
      return sum/number;                                            // (18)
    }
  }
```

**Output of the Program:**

```
Computing average.
Finally in printAverage().
java.lang.ArithmeticException: Integer division by 0
        at Average7.computeAverage(Average7.java:35)
        at Average7.printAverage(Average7.java:19)
        at Average7.main(Average7.java:6)
Exception handled in main().
Finally in main().
Exit main().
```

## The `throws` Clause

If a checked exception is thrown in a method, it must be handled in one of three ways:

- By using a `try` block and catching the exception in a handler and dealing with it.
- By using a `try` block and catching the exception in a handler, but throwing another exception that is either unchecked or declared in its `throws` clause.
- By explicitly allowing propagation of the exception to its caller by declaring it in the `throws` clause of its method header.

## The `throws` Clause

A `throws` clause can be specified in the method header.

```
... someMethod(...)
  throws <ExceptionType_1>, <ExceptionType_2>,..., <ExceptionType_n> { ... }
```

It explicitly allows propagation of the exception to its caller.

## The `throws` Clause

```
public class Average8 {
  public static void main(String[] args) {
    try {                                                    // (1)
      printAverage(100, 0);                                  // (2)
    } catch (IntegerDivisionByZero idbze) {                  // (3)
      idbze.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 totalNumber)
  throws IntegerDivisionByZero {                             // (6)

    int average = computeAverage(totalSum, totalNumber);
    System.out.println("Average = " +
        totalSum + " / " + totalNumber + " = " + average);
    System.out.println("Exit printAverage().");              // (7)
  }
```

## The `throws` Clause

```
  public static int computeAverage(int sum, int number)
  throws IntegerDivisionByZero {                             // (8)

    System.out.println("Computing average.");
    if (number == 0)                                         // (9)
      throw new IntegerDivisionByZero("Integer Division By Zero");
    return sum/number;                                       // (10)
  }
}

class IntegerDivisionByZero extends Exception {              // (11)
  IntegerDivisionByZero(String str) { super(str); }          // (12)
}
```

**Output of the Program:**

```
Computing average.
IntegerDivisionByZero: Integer Division By Zero
        at Average8.computeAverage(Average8.java:33)
        at Average8.printAverage(Average8.java:22)
        at Average8.main(Average8.java:7)
Exception handled in main().
Finally done in main().
Exit main().
```

IT602: Object-Oriented Programming

**Next lecture -**

OOP Concepts