INTRODUCTION

Programming Principles II

Topics

Exceptions

Kinds of exceptions

Exceptions

The following code

```
class TestClass {
        public static void main(String args[]) {
                int i = 5 / 0;
                System.out.println("Some other operation");
        }
}
```

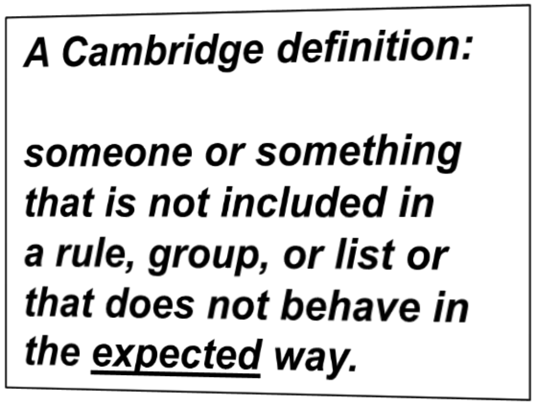
compiles successfully but during the runtime will print the following:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestClass.main(TestClass.java:3)
```

Exceptions

Exception is a state that is not implicitly considered in the code. When exception happens, we say that *program throws an exception*.

Exception (short for 'exceptional event') is an event that disrupts the normal, planned flow of instructions.



A Cambridge definition:

someone or something that is not included in a rule, group, or list or that does not behave in the expected way.

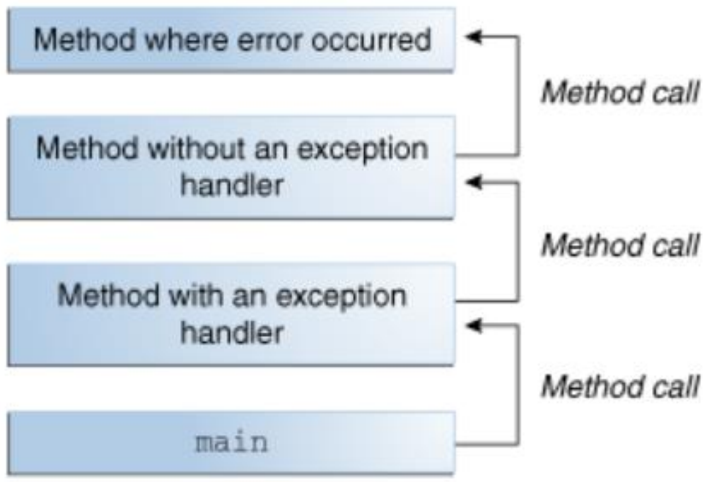
Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

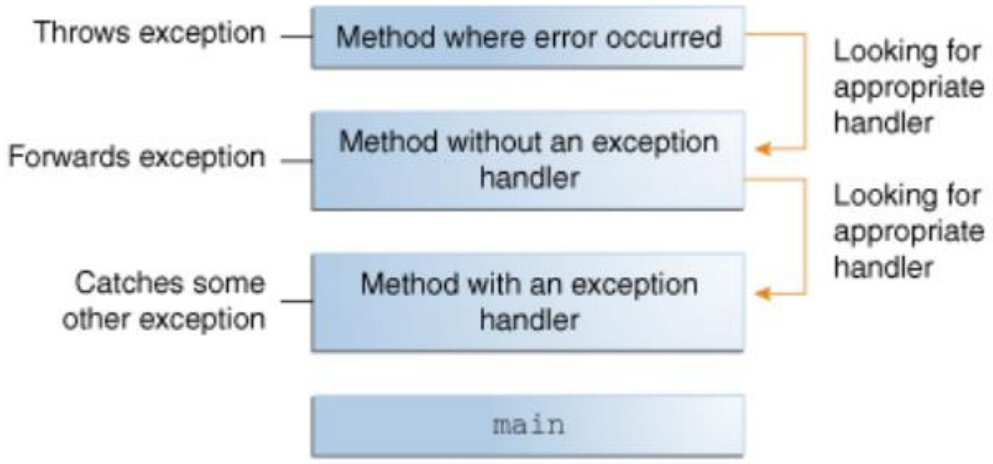
Exceptions

There are different types of exceptions. Some of them:

- When a mathematical operation related error happens, it throws *ArithmeticException*
- When program tries to access an array element with the index that greater than the (*array size – 1*) it throws *ArrayIndexOutOfBoundsException*
- When a method on un-instantiated object (reference) is called it throws *NullPointerException*
- etc.

Exceptions (procedure)


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exceptions

Your code should follow the *Catch or Specify requirement*. I.e., if a piece of code might raise an exception it might be enclosed by either:

- A *try* block following a catch statement which will handle the very exception
- (if there is no handler) the *method* which specifies that it can throw an exception

Kinds of exceptions

In Java there are three kinds of exceptions:

- Checked exceptions
  - follow the **Catch or Specify Requirement.**
  - checked during the compile time.

By default, all the exceptions, except 'Errors' and 'RuntimeException's.

- Runtime exceptions
  - **DO NOT** follow the **Catch or Specify Requirement**.
  - checked during the runtime.

Exceptions indicated by **RuntimeException** class and its subclasses

- Errors
  - **DO NOT** follow the **Catch or Specify Requirement.**
  - exceptions that are external to the application, usually cannot be handled.

Exceptions indicated by **Error** class and its subclasses

Questions

Topics

Handling exceptions

try-catch-finally

Handling exceptions - try-catch

Unplanned program terminations can be prevented by considering the exception case.

To follow the **Catch** or Specify Requirement, try-catch block can be used.

try-catch block helps defining the main code in *try {}* block and put the reaction to the Exception in the *catch() {}* block.

```
class TestClass {
        public static void main(String args[]) {
                try {
                        int i = 5 / 0;
                } catch (ArithmeticException ae) {
                        System.out.println("Error happened. Reason: "+ ae.toString());
                }
                System.out.println("Some other operation");
        }
}
```

Catching multiple exceptions

In Java it’s possible to catch more than one exception in each block. In this case, the exception classes shall be separated with the (pipe) “**|**” character:

```
try {
   // some operations;
   }
catch (NullPointerException npe) {
   System.out.println("Error type 1 : "+ npe.toString());
   }
catch (ArithmeticException | IOException  ex) {
   System.out.println("Error type 2 : "+ ex.toString());
   }
```

try-catch-finally

The general syntax of try/catch is shown below.

Note that, a single try can have several catch blocks – each for different exception types.

The finally block that is not mandatory, **is executed in any case**.

```
try {
   // the operation is here
}catch (ExceptionType1 ex1) {
   // What to do if ExceptionType1 happened?
}catch (ExceptionType2 ex2) {
   // What to do if ExceptionType2 happened?
}finally {
   // This part will execute in any case, in the end.
}
```

try-catch-finally - Example

```
int [] arr  = { 0, 2, 3, 4, 5};
int a = args[0]; int b = args[1];

try {
   System.out.println("Result :" + a / b);
}catch (ArithmeticException ex1) {
   System.out.println("Arithmetic exception: "+ ex1.toString());
}catch (ArrayIndexOutOfBoundsException ex2) {
   System.out.println("Array index: "+ ex2.toString());
}finally {
   arr = new int[]{ 1, 2, 3};
   System.out.println("Array is reset");
}
```

try-catch-finally - Example

Since all exceptions extend the **Exception** class, all possible exception catching can be replaced with one catch of the **Exception** (**or Throwable**):

```
int [] arr  = { 0, 2, 3, 4, 5};
int a = args[0]; int b = args[1];

try {
   System.out.println("Result :" + a / b);
}catch (Exception ex) {
   System.out.println("An exception: "+
ex.toString());
}finally {
   arr = { 1, 2, 3};
   System.out.println("Array is reset");
}
```

finally

Note: If the JVM exits while the try or catch code is being executed, then the finally block may not execute.

Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

Why use finally?

Why have finally, if we can do the operation after the try/catch block?

```
try {
    // some action
    }
catch (Exception ex) {
    System.out.println("Error : "+ex);
    }
finally {
    System.out.println("Finally");
    }
```

```
try {
    // some action
    }
catch (Exception ex) {
    System.out.println("Error : "+ex);
    }
System.out.println("Finally");
```

Why use finally?

If there is a return, break, continue or any other java keyword that changes the sequential execution of code within the try or catch block, the statements inside the finally block will still be executed.

```
try {
     // some action
     if (condition)
         return “File is empty”;
     }
catch (Exception ex) {
     System.out.println("Error : "+ex);
     return “Cannot read a file”;
     }
finally {
     // Will be executed in any case!
     // Free resources, close file,…
     }
```

Questions

Topics

Throwing exceptions

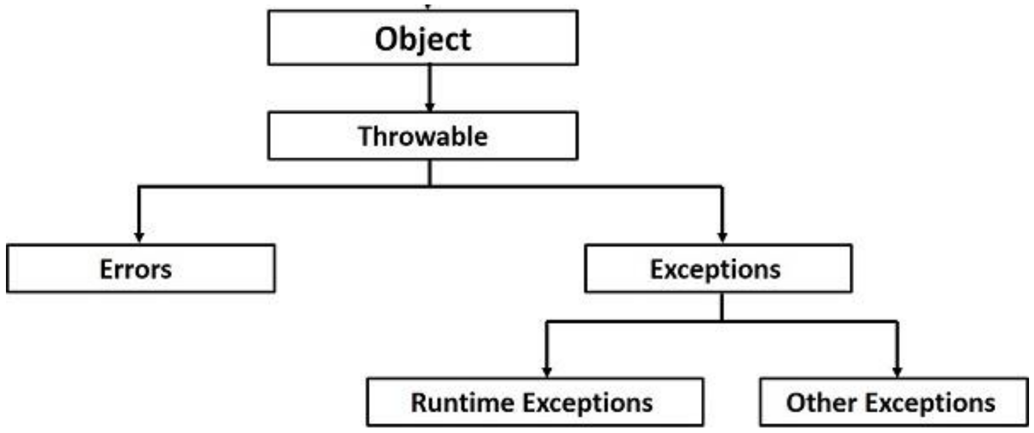
Specifying exceptions

Exception hierarchy

All exception types are subclasses of the built-in class **Throwable**.

There are two direct subclasses of **Throwable:**

- **Exception** that is used for exceptional conditions in user programs. There is an important subclass of **Exception**, called **RuntimeException** which is used to represent runtime errors.
- **Error** defines exceptions that are related to errors happened in Java Virtual Machine, not in Java programs itself.



Throwing an exception

In Java, programmer can also raise an exception by using ***throw*** keyword:

```
throw  ExceptionObj;
```

```
if ( a < 0 ) {
	ArithmeticException ae = new ArithmeticException(“Negative number!”);
	throw ae;
	}
```

or

```
if ( a < 0 ) {
	throw new ArithmeticException(“Negative number!”);
	}
```

Throwable methods

Return Type	Method and Description
String	getMessage() Returns the detail message string of this throwable.
void	printStackTrace() Prints this throwable and its backtrace to the standard error stream.
void	printStackTrace(PrintStream s) Prints this throwable and its backtrace to the specified print stream.
void	printStackTrace(PrintWriter s) Prints this throwable and its backtrace to the specified print writer.
String	toString() Returns a short description of this throwable.

See more: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html

Using ***throws***

If the operation throws an exception different than ***RuntimeException*** or ***Error***, then the compiler will force you to handle this exception (have a try-catch block for it).

```
import java.io.*;

class ExTest {
	public static void main(String args[]) {
		File f = new File("Hello.txt");
		FileReader reader = new FileReader(f);
	}
}
```

Catch or **Specify** Requirement

```
ExTest.java:6: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
		FileReader reader = new FileReader(f);
```

Using *throws*

Adding throws keyword to your method with the list of exceptions will remove the responsibility of handling exceptions:

returnType methodName(params) throws Exception1, Exception2,… {

```
import java.io.*;

class ExTest {
        public static void main(String args[]) throws FileNotFoundException{
                File f = new File("Hello.txt");
                FileReader reader = new FileReader(f);
        }
}
```

Homework (non-graded)

Write a program that reads input using System.in.read():

- Write a program and analyze the error.
- Handle the exception with try catch.
- Use throws and remove the exception handling

Questions

Topics

Defining custom exceptions

Creating custom exceptions

When you are selecting a type of Exception to use among all are available to you, if you do not find a type which has the very behavior you are seeking, you might create your own exception types.

To create a custom exception, it is sufficient to create a class that extends the ***Exception*** class (or any of its subclasses).

To have a better output, it is recommended to override the necessary Throwable methods (at least toString()).

It is also a good practice to name your exception with “Exception” ending, like ***SomeCustomerException****,* for better readability.

Creating custom exceptions

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

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

Using custom exceptions

```java
public static void createUser(String username, String password)
            throws InvalidUsernameException, InvalidPasswordException {
     if (username == null || username.length() < 3)
         throw new InvalidUsernameException("Username cannot contain
                                       less than three symbols");

     if (!Character.isAlphabetic(username.charAt(0)))
         throw new InvalidUsernameException("Username cannot start
                                   with non-alphabetical character");

     if (password.length() < 8)
         throw new InvalidPasswordException("The length of the
                                   password cannot be less than 8.");
}
```

Handling custom exceptions

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

       try {
            createUser("demo_user", "12345");
        } catch (InvalidUsernameException e) {
            e.printStackTrace();
        } catch (InvalidPasswordException e) {
            e.printStackTrace();
        }

   }
```

Homework (non-graded)

Create a ***Student*** class with *id, name and surname* fields. Create getters and setters for these fields.

Create a custom exception class called ***StudentNameException***.

In your student class's setName() and setSurname() throw a StudentNameException if the provided values are less than 2 symbols or null.

Questions