Exceptions Handling

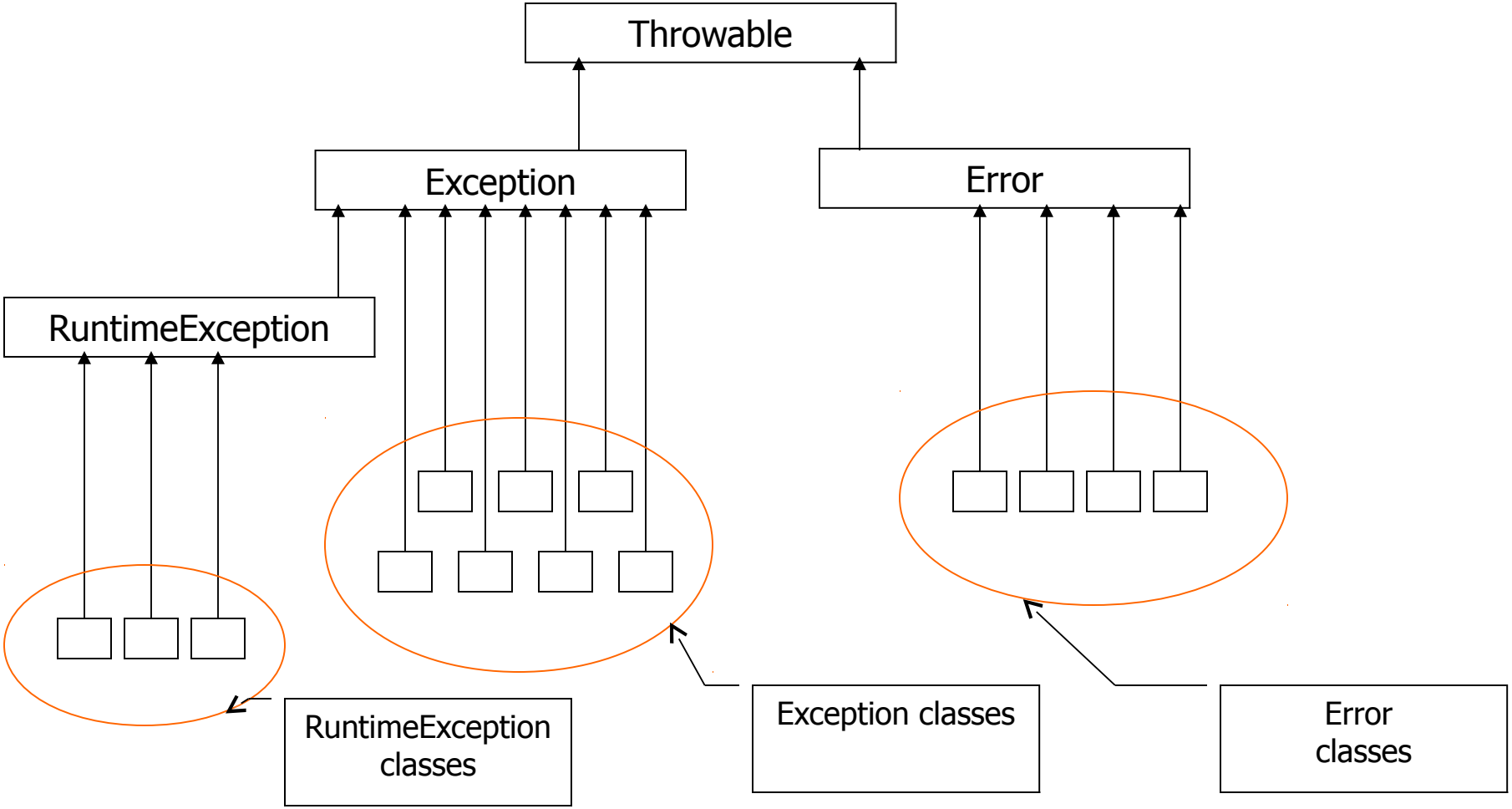
Introduction

- Each malfunction is described by exception, an object that was instantiated from a class that extends the `Throwable` class.
- The malfunction can be any situation that you chose to define as a malfunction (or others have decided for you).

Introduction

- The instantiated class can be one of the classes that were already declared as system classes or a new one.

The Exception Classes Hierarchy



The `Error` Classes

- Severe problems from which a recovery is difficult or impossible (e.g. running out of memory). The programmer isn't expected to handle them.
- In most cases, getting an error will required us to try and run the JVM in a different customized way.

The `RuntimeException` Classes

- Malfunctions that results from a design or from an implementation problem.

  `ArrayIndexOutOfBoundsException`

  `NullPointerException`

- Usually, runtime exceptions aren't handled and writing the program properly it is possible to avoid them.

The `Exception` Classes

- Difficulties during runtime that are usually caused by environmental effects, and can be handled. (e.g. `FileNotFoundException`)
- Programmers are encouraged to handle them.
- This group includes those classes that extend Exception without going through the `RuntimeException` class.

The `try` & `catch` Statement

- Each time a malfunction occurs, the relevant exception class is instantiated and the exception object is thrown.
- We can place the error prone code within a try & catch statement.

The `try` & `catch` Statement

```
try
{
        code that might not work properly
        and therefore might throw an exception
}
catch(AbcException e)
{
        code that will be invoked
        if the exception is caught
        in this catch
}
```

The `try` & `catch` Statement

- Within the try block we place the code that might throw an exception.
- The catch block is very similar to a method. Its declaration has one parameter, which gets the reference for the exception object if its type fits the parameter's type.

The `try` & `catch` Statement

- After the try block we can place more than one catch block. The first catch block that succeeds to catch the thrown exception is preformed.

Try & Catch Statement

```
try
{
        code that might not work properly
        and therefore might throw an exception
}
catch(AException e)
{

}
catch(BException e)
{

}
catch(CException e)
{

}
```

The `throw` Command

- Each piece of code that might throw an exception has the `throw` command within it or within one of the methods it calls.
- The `throw` command, followed by a reference to a new exception object, causes the exception object to be thrown.

The `throw` Command

- This exception might be caught within a try & catch statement.
- The execution never returns to the code line where the exception was thrown. The execution continues after the catch block that catches the exception.

The `throw` Command

```
public void doSomethiung()
{
     if(...)
          throw new SomethingException();
}
```

The Throwing Chain

- Given a method invocation that doesn’t succeed and an exception is thrown, if the method call wasn’t placed within a try & catch the exception will be transferred to the method from which the current method was called.

The Throwing Chain

```
public void aaa()
{
        …
        bbb();
        System.out.println("aaa ended");
}

public void bbb()
{
        …
        ccc();
        System.out.println("bbb ended");
}

public void ccc()
{
        …
        if(… )
                throw new SomethingException();
        System.out.println("ccc ended");
}
```

SomethingException Instence was thrown

SomethingException Instence was thrown

SomethingException Instence was thrown

The Finally Block

- When using a try & catch statement, it is possible to place after the last catch block a finally block.

```
try
{
          doSomething();
}
catch(AaaException e) {...}
catch(BbbException e) {...}
finally
{
          this block always executes
}
```

The Finally Block

- The finally block always executes, whether an exception is thrown or not and no matter whether the thrown exception is caught or not.
- The only case in which the finally block doesn't execute is when calling the `System.exit()` method.

The `throws` Keyword

- It is possible (sometimes necessary) to add `throws` to the method declaration, and announce – by doing so – that the method might throw an exception.

```
public void aaa() throws SomethingException
{
            ...
}
```

The 'Handle or Declare' Rule

- When a code segments might throw an exception that belongs to the Exception classes group it is necessary either placing the code within an appropriate try&catch block or add the appropriate throws declaration to the method definition.
- The compiler doesn’t compile code that doesn’t follow this rule.

Methods Overriding

- When you override a method, the new version can't throw exceptions (that belong to the third group) that weren't thrown by the overridden version.
- If the type of the exception thrown by the overriding method belongs to the third group and it extends the type of exception thrown by the overridden version it would compile successfully.

Sample for Exception Handling

```
class MyException extends Exception
{
     MyException(String msg)
     {
         super(msg);
     }
}

public class MyExceptionHandlingDemo
{
     public static void main(String args[])
     {
         int val = 123;
```

Sample for Exception Handling

```
try
{
        aaa(val);
}
catch(MyException e)
{
        e.printStackTrace();
}
finally
{
        System.out.println("finally always works");
}
```

Sample for Exception Handling

```
    val = -123;
    try
    {
            aaa(val);
    }
    catch(MyException e)
    {
            e.printStackTrace();
    }
    finally
    {
            System.out.println("finally always works");
    }
}
```

Sample for Exception Handling

```
static void aaa(int num) throws MyException
{
   System.out.println("aaa start");
   bbb(num);
   System.out.println("aaa end");
}

static void bbb(int val) throws MyException
{
   System.out.println("bbb start");
   ccc(val);
   System.out.println("bbb end");
}
```

Sample for Exception Handling

```
    static void ccc(int number) throws MyException
    {
       System.out.println("ccc start");
       if (number<0)
       {
          throw new MyException("negative number");
       }
       else
       {
          System.out.println("The log of " + number + " is "
               + Math.log(number));
       }
       System.out.println("ccc end");
    }
}
```

The try-with-resources Statement

- As of Java 7 we can code a try statement that declares one or more resources. Each resource is an object that must be closed after the program is finished.
- The try-with-resources statement ensures that each resource is closed at the end of the statement.

The try-with-resources Statement

- Each object considered as a resource must be instantiated from a class that implements `java.lang.AutoCloseable`.

The try-with-resources Statement

```
public class TryResourcesDemo
{
    public static void main(String[] args)
    {
        try
        {
            String str = readTextFile("bb.txt");
            System.out.println(str);
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
```

The try-with-resources Statement

```
    static String readTextFile(String path) throws IOException
    {
        StringBuilder builder = new StringBuilder();
        try (BufferedReader br = new BufferedReader(new FileReader(path)))
        {
            String str = null;
            while((str = br.readLine())!=null)
            {
                builder.append(str).append("\n");
            }
        }
        return builder.toString();
    }
}
```

The try-with-resources Statement

- If an exception is thrown both from the try block and from closing the resource the exception thrown from closing the resource will be suppressed.

The try-with-resources Statement

- We can define within the try-with-resource more than one resource. We should use '`;`' for separating their declarations.

```
try( InputStream is = new YoyoFilInputStream();
     OutputStream os = new YoyoOutputStream() )
{
     …
}
```

Handling Multiple Exceptions Types

- As with Java 7 we can handle more than one type of exception using a single catch block.

```
try
{
      …
}
catch(IOException | ClassNotFoundException e)
{
      …
}
```

Handling Multiple Exceptions Types

- When catch handles more than one type of exception the catch parameter is implicitly `final`. We won't be able to assign it with a new value.
- The bytecode created from compiling code that includes a catch statement that handles multiple types of exceptions will be smaller. There won't be any code duplicity.

Handling Multiple Exceptions Types

```java
package com.abelski.samples;

import java.io.DataOutputStream;

public class MultipleExceptionsCatch
{
    public static void main(String[] args)
    {
        try(FileOutputStream fos = new FileOutputStream("bb.txt");
            DataOutputStream dos = new DataOutputStream(fos))
        {
            dos.writeUTF(args[0]);
        }
        catch(ArrayIndexOutOfBoundsException | IOException e)
        {
            e.printStackTrace();
        }

    }

}
```

Handling Multiple Exceptions Types