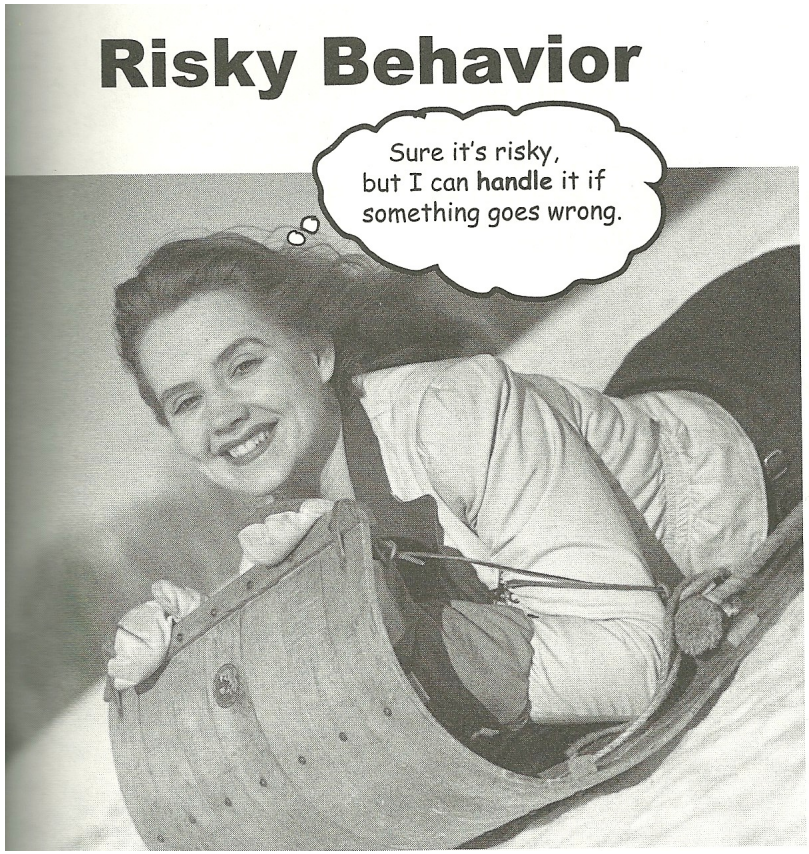
HBO Graduaat Informatica Optie Programmeren

Exception Handling

Chapter Overview

- Exception models and Java philosophy
- Java exception hierarchy
- Detecting, indicating and handling exceptions
- Releasing resources after an error
- Creating custom exception classes

Introduction



Introduction

- Objectives
  - To write programs capable of performing without failure under a wide range of conditions
  - To introduce the Java exception hierarchy
- Chapter content
  - Why is error handling so important?
  - Java syntax for detecting, indicating and handling exceptions; creating custom exceptions
  - Cleaning up after a problem
- Practical content
  - Familiarization with exception handling
- Summary

Exception Handling

- Philosophy: Badly formed code will not be run
- Exception models:
  - Resumption (like in Visual Basic): attempt to correct and resume execution
  - Termination (Java, C++, ...): you assume the error is so critical that there's no way to get back to where the exception occurred

Exception Handling

- Exceptional condition:
  - a problem that prevents the continuation of the method or scope that the program is in
- Reaction:
  - pass the problem (with the necessary information) to a higher context

Exception Handling

- Throwing an exception
  1. An `Exception` object is created
  2. The current path of execution is stopped
  3. The reference for the `Exception` object is ejected from the current context
  4. The *exception handling* mechanism looks for the right place to continue the program

Exception Handling

- Throwing an exception: example

```
public static Object[] toObjectArray(Object source) {
   if (source instanceof Object[]) {
      return (Object[]) source;
   }
   if (source == null) {
      return new Object[0];
   }
   if (!source.getClass().isArray()) {
      throw new IllegalArgumentException("Source is not an array: " + source);
   }
   int length = Array.getLength(source);
   if (length == 0) {
      return new Object[0];
   }
   Class wrapperType = Array.get(source, 0).getClass();
   Object[] newArray = (Object[]) Array.newInstance(wrapperType, length);
   for (int i = 0; i < length; i++) {
      newArray[i] = Array.get(source, i);
   }
   return newArray;
}
```

Exception Handling

- Catching an exception:
  - The caller of a method which potentially can throw an exception can either
    - be prepared to catch the exception, or
    - declare it can throw this exception in its turn

Exception Handling

- Catching an exception: example

```
...
try {
  Context ctx = new InitialContext();
  dataSource = (DataSource) ctx.lookup(dataSourceJndiName);
} catch (ClassCastException cce) {
  ...
} catch (NamingException ne) {
  ...
}
...
```

- `lookup()` can end up with a `NamingException`
- The casting can lead to a `ClassCastException`

- After the try keyword, catch is mandatory
- One catch per exception type
- Notice the argument for each catch block

Exception Handling

- Multi catching: example

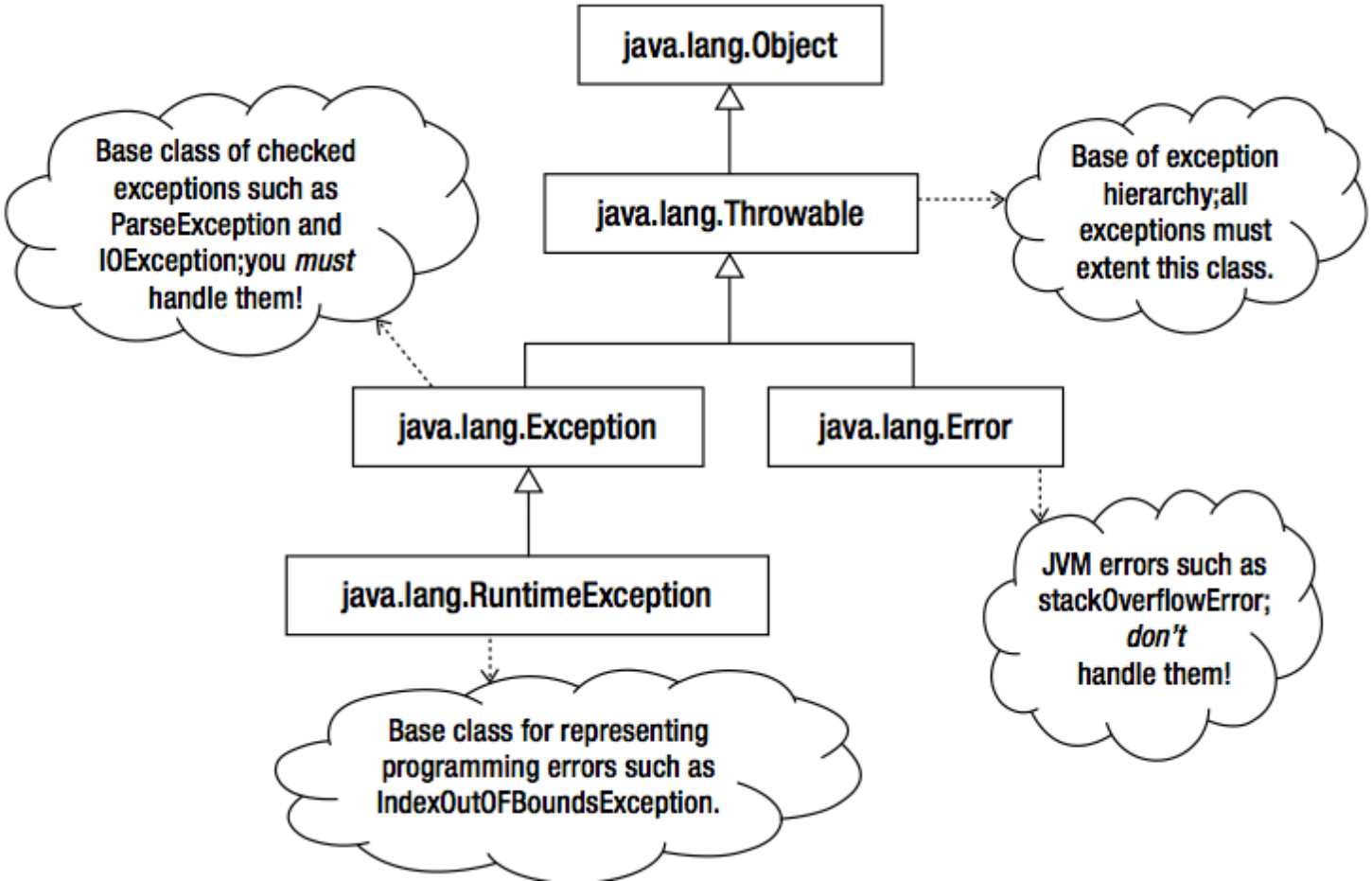
```
...
try {
  Context ctx = new InitialContext();
  dataSource = (DataSource) ctx.lookup(dataSourceJndiName);
} catch (ClassCastException cce | NamingException exc ) {
  ...
}
...
```

- As of JDK 7
- In a multi-catch block, you cannot combine catch handlers for two exceptions that share a base- and derived-class relationship. You can only combine catch handlers for exceptions that do not share the parent-child relationship between them.

Exception Handling

- Exception hierarchy

RuntimeExceptions are NOT checked by the compiler. They're known as (big surprise here) "unchecked" exceptions. You can throw, catch, and declare RuntimeExceptions, but you don't have to, and the compiler won't check.



Exception Handling

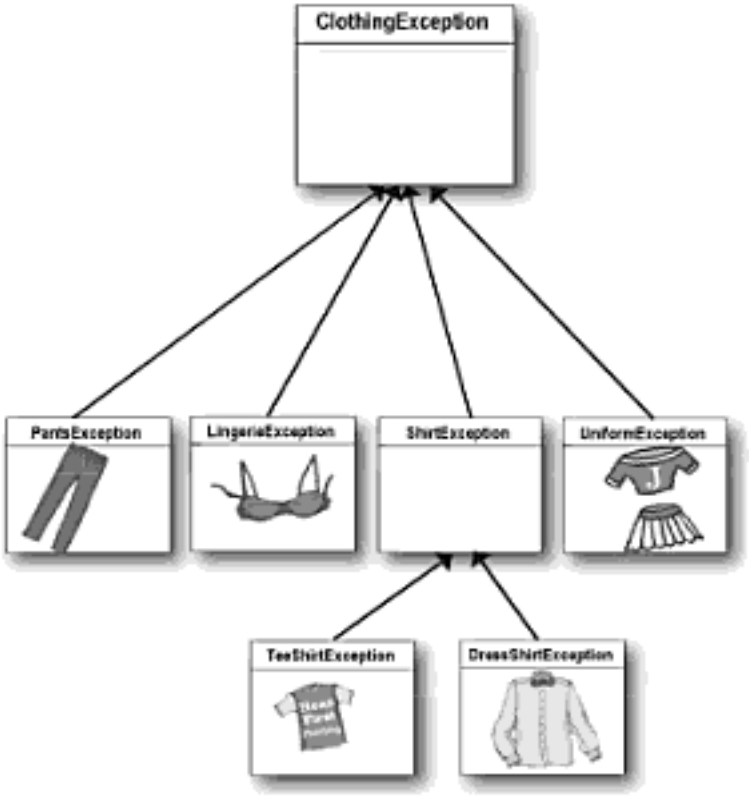
- A method can thrown multiple exceptions

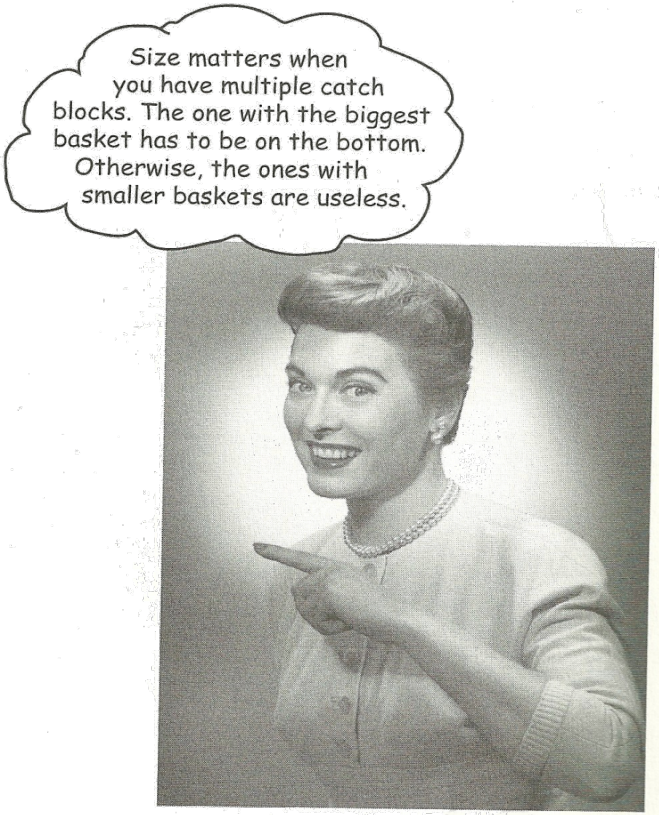
```
public class Laundry {

   public void doLaundry() throws PantsException, LingerieException {
      //code that could throw either exception
   }

}
```

```
Laundry laundry = new Laundry();

try {
   laundry.doLaundry();
} catch (PantsException pex) {
   //recovery code
} catch (LingerieException lex) {
   //recovery code
}
```

Exception Handling

- Exceptions can be polymorphic





- But multiple catch blocks must be ordered from smallest to biggest

Exception Handling

- The `Error` type: indicates a serious problem and should not be caught by the program (example: `OutOfMemoryError`)
- The `RuntimeException` subtree of the hierarchy: you don't need to catch or declare that your method throws this kind of exception
- All other exceptions:
  - Must be caught or
  - must be specified in the method API if they could be thrown by the method

Exception Handling

- Creating your `Exception` type:
  - The JDK exception hierarchy can't foresee all the errors you might want to report, so you can create your own, to denote a special problem that your library might encounter.
  - Inherit from an existing exception class that is close in meaning to your new exception.
  - One constructor argument is useful for exceptions that are little more than wrappers for other throwables

Exception Handling

- Creating your `Exception` type – example:

```
public class ServiceLocatorException extends Exception {

    public ServiceLocatorException() {
        super();
    }

    public ServiceLocatorException(String message) {
        super(message);
    }

    public ServiceLocatorException(Throwable cause) {
        super(cause);
    }

    public ServiceLocatorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Exception Handling

- Indicating a potential risk of exception:
  - A method that throws an exception (except for `RuntimeException` and its subclasses) must add a `throws` clause in the method definition:

```
public static void sleep(long millis) throws InterruptedException {}
```

Exception Handling

- Indicating a potential risk: example

```
import javax.naming.*;
import javax.sql.*;

public class ServiceLocator {

   public static DataSource getDataSource(String dataSourceJndiName)
         throws ServiceLocatorException {
      DataSource dataSource = null;
      try {
         Context ctx = new InitialContext();
         dataSource = (DataSource) ctx.lookup(dataSourceJndiName);
      } catch (ClassCastException cce) {
         throw new ServiceLocatorException(cce);
      } catch (NamingException ne) {
         throw new ServiceLocatorException(ne);
      }
      return dataSource;
   }
}
```

Exception Handling

- Rethrowing an Exception
  - Sometimes you'll want to rethrow the exception that you just caught,
    - particularly when you use `Exception` to catch any exception
    - or when you are unable to do something useful with it

```
 catch(Exception e) {
System.err.println("Problem ...");
throw e;
}
```

  - It's also possible to rethrow a different exception from the one you caught.

Exception Handling

- Some `Exception` methods:

operator	Description
`printStackTrace()`	Prints this throwable and its backtrace to the standard error stream (the exception message plus information added through the `fillInStackTrace()` method
`getMessage()`	Returns the detail message string of this throwable.
`getCause()`	Returns the cause of this throwable or null if the cause is nonexistent or unknown (via a constructor that contains a Throwable argument)
`getLocalizedMessage()`	Creates a localized description of this throwable. Subclasses may override this method in order to produce a locale-specific message
`getStackTrace()`	Programmatic access to the stack trace

Exception Handling

- Cleaning up with `finally`:
  - Sometimes you want to execute some code whether or not an exception is thrown within a `try` block
  - Releasing resources (network and database connections, ...) other than memory (the garbage collector handles this in any case)
  - The `finally` block is optional and comes after all `catch` blocks; if present, it will always be executed

Exception Handling

- Cleaning up resources: example

```
try {
   dataSource = ServiceLocator.getDataSource(DATA_SOURCE);
   conn = dataSource.getConnection();
   stmt = conn.createStatement();
   rs = stmt.executeQuery("select * from citycodes_tbl");
   while (rs.next()) {
     ...
   }
 } catch (Exception e) {
   ...
 } finally {
   try {
     if(rs != null){rs.close();}
     if(stmt != null){stmt.close();}
     if(conn != null){conn.close();}
   } catch(Exception e) {
     System.out.println(e);
   }
 }
 return cityList;
}
```

Exception Handling

- Restrictions
  - Only the exceptions of the base class can be thrown in an overridden method. Note: constructors don't have this restriction.
  - A derived-class method doesn't have to throw the exceptions of the base class. The base-class should do it.
  - When you upcast a class to its base class, the compiler forces you to catch the exceptions for the base type.
  - No overloading of a method based on the exceptions.

Exercise

Summary

- Java includes an exception handling feature based on the termination principle.
- You can detect, indicate and handle exceptional situations with a few keywords.
- Java provides an exception hierarchy that you can use or extend.