Programming Paradigms 159.272

Exception Handling

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Java tutorial, lesson on exceptions
   http://docs.oracle.com/javase/tutorial/essential/exceptions/

Overview

- declaring exceptions
- errors vs exceptions
- runtime (unchecked) exception
- handling exceptions
- stack traces
- implementing exception classes
- exception chaining
- documenting exceptions

Exceptions

- an **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- in general, exception occur in programs and there is no way to write (complex) software that does not cause exceptions
- Java offers **an exception handling framework** that can be used to deal with exceptions when they have occurred
- exceptions are not always related to errors in the program, but can also be caused by hardware failure (such as a broken network connections)

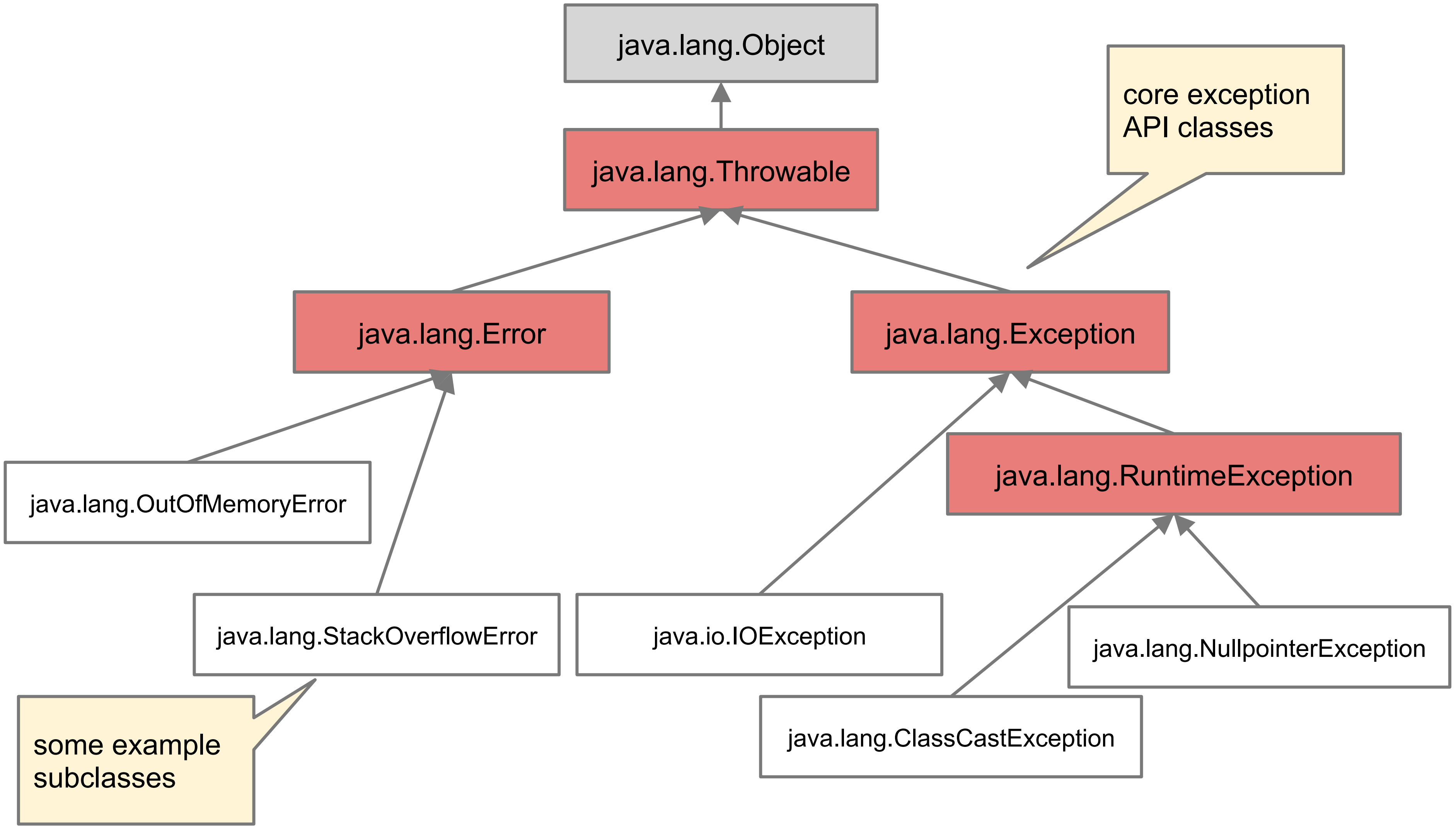
Errors

- **errors** are similar to exceptions
- errors usually indicate that the JVM encounters a serious problem
- examples - errors that occur when the JVM has problems to allocate memory:
  - OutOfMemoryError - the JVM cannot allocate heap for a new object to be created
  - StackOverflowError - recursion is used but not correctly terminated

Throwables

- exceptions and errors are represented by classes within the normal Java class hierarchy
- the root of the exception hierarchy is `java.lang.Throwable`
- the root class of all errors is `java.lang.Error`
- the root class of all exceptions is `java.lang.Exception`
- there is a tree of special exception classes, the root of these classes is `java.lang.RuntimeException`

The Exception/Error Class Hierarchy


java.lang.Object
core exception API classes
java.lang.Throwable
java.lang.Error
java.lang.Exception
java.lang.OutOfMemoryError
java.lang.RuntimeException
java.lang.StackOverflowError
java.io.IOException
java.lang.NullpointerException
java.lang.ClassCastException
some example subclasses

Declaring Exceptions

- methods and constructors can declare one or many exceptions
- these are the exceptions that can be expected when these methods or constructors are called
- these exceptions are part of the interface of these methods or constructors, and the compiler will enforce certain constraints based on the declared exceptions

```
public List readObjectsFromCSV(File csv) throws java.io.IOException {
        // read objects from a CSV file
        ...
}

public void writeObjectsToHTML(File html,List objects) throws
java.io.IOException {
    // write objects to an html file
        ...
}
```

Catch Or Specify

- the compiler enforces the **catch or specify** requirement
- when methods or constructors declaring exceptions are called, the caller must either:
  - declare the same or a more general exception
  - or handle the exception

Catching Exceptions

- when catching an exception, a **try-catch block** is used
- the **catch block** contains code that is executed when an exception occurs
- there can be **multiple catch blocks**, each handling one particular type of exception
- if a catch block catches an exception of type A, it also catches exceptions instantiating subclasses B of A (because all B's are also A's !)

Catching Exceptions ctd

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput
                this.readObjectsFromCSV(csvInput);
    }
    catch (IOException exception) {
        System.out.println("conversion failed: an exception has occurred");
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        // read objects from a CSV file
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
    // write objects to an html file
        ...
}
```

try block

catch block

- the catch block will be executed if the respective files cannot be accessed
- for instance, if there is not enough space on the harddrive to write to the htmlOutput file

Catching Exceptions: Superclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput);
                this.readObjectsFromCSV(csvInput);
    }
    catch (Exception exception) {
        System.out.println("conversion failed: an exception has occurred");
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- note: `java.io.IOException` is a subclass of `java.lang.Exception`

Catching Exceptions: Superclasses

```
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput);
                this.readObjectsFromCSV(csvInput);
    }
    catch (Exception exception) {
        System.out.println("conversion failed: an exception has occurred");
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- this still works: if we catch all exceptions, we also catch all io exceptions !
- it is possible to catch on `Throwable` (superclass of `Exception` and `Error`) !

Catching Exceptions: Subclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput);
                this.readObjectsFromCSV(csvInput);
    }
    catch (FileNotFoundException exception) {
        System.out.println("conversion failed: an exception has occurred");
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- note: `java.io.FileNotFoundException` is a subclass of `java.io.IOException`

Catching Exceptions: Subclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput);
                this.readObjectsFromCSV(csvInput);
    }
    catch (FileNotFoundException exception) {
        System.out.println("conversion failed: an exception has occurred");
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- this will **not work - compilation fails**
- the called methods may throw IOExceptions that are not FileNotFoundException - and those would not be handled!

Catching Multiple Exceptions

- **from Java 7** , catching multiple exceptions is supported
- this makes it possible to write more compact and concise code
- instead of having two catch blocks for two exception types X1 and X2, only one block is needed

```
try {...}
catch (X1 ex1) {..}
catch (X2 ex2) {..}
```

```
try {...}
catch (X1|X2 ex) {..}
```

finally blocks

- the optional **finally block** is executed after the try block has been exited or exceptions have been handled
- there are some exceptions: for instance, the finally block is not executed when the JVM crashes (with an error) when either block is executed

finally block ctd

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
         try {
                  List objects = this.readObjectsFromCSV(csvInput);
                  this.readObjectsFromCSV(csvInput);
     }
     catch (IOException exception) {
         System.out.println("conversion failed: an exception has occurred");
     }
     finally {
         System.out.println("done");
     }
}

public List readObjectsFromCSV(File csv) throws IOException {
         ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
         ...
}
```

finally block

Dealing with Exceptions

- console printing is not a good exception handling strategy
- it is better to use the dedicated error console `System.err` instead of `System.out`
- still better: use a full logging framework to log exception details
- exceptions support two methods that are useful:
  - `getMessage()` - this retrieves a description of the exception that has occured
  - `printStackTrace()` - prints information about the exception to `System.err`

Dealing with Exceptions ctd

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) {
        try {
                List objects = this.readObjectsFromCSV(csvInput);
                this.readObjectsFromCSV(csvInput);
    }
    catch (IOException exception) {
        System.err.println(exception.getMessage());
        exception.printStackTrace();
    }
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

Reading Stack Traces

```
java.io.FileNotFoundException: nofile (No such file or directory)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:138)
  at java.io.FileReader.<init>(FileReader.java:72)
  at nz.ac.massey.cs.pp.exceptionhandling.CSV2HTMLConverter1.readObjectsFromCSV(CSV2HTMLConverter1.java:26)
  at nz.ac.massey.cs.pp.exceptionhandling.CSV2HTMLConverter1.main(CSV2HTMLConverter1.java:15)
```

- the stack trace contains information about the method call stack that has led to the exception
- note that the source code lines are referenced here
- the invocation of `main` has led to an invocation of `readObjectsFromCSV` (in line 15), which has then failed to execute in line 26
- the problem was an attempt to read from a file ("nofile") that does not exist

Specifying Exceptions

- instead of catching exceptions, exceptions can be specified as well
- this means that the **responsibility** to deal with the exception **is delegated** to the called of the method or constructor

Specifying Exceptions

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) throws
IOException {

    List objects = this.readObjectsFromCSV(csvInput);
        this.readObjectsFromCSV(csvInput);
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

Specifying Exceptions: Superclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) throws
Exception {

    List objects = this.readObjectsFromCSV(csvInput);
        this.readObjectsFromCSV(csvInput);
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

Specifying Exceptions: Superclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) throws
Exception {

    List objects = this.readObjectsFromCSV(csvInput);
        this.readObjectsFromCSV(csvInput);
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- this still works: `convertCSV2HTML` declares that it may throw an exception (any instance of `Exception`), and any instance of `IOException` is also an instance of `Exception`

Specifying Exceptions: Subclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) throws
FileNotFoundException {

    List objects = this.readObjectsFromCSV(csvInput);
        this.readObjectsFromCSV(csvInput);
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- note: `java.io.FileNotFoundException` is a subclass of `java.io.IOException`

Specifying Exceptions: Subclasses

```
import java.io.*; import java.util.List;
...
public void convertCSV2HTML (File csvInput,File htmlOutput) throws
FileNotFoundException {

    List objects = this.readObjectsFromCSV(csvInput);
        this.readObjectsFromCSV(csvInput);
}

public List readObjectsFromCSV(File csv) throws IOException {
        ...
}

public void writeObjectsToHTML(File html,List objects) throws IOException {
        ...
}
```

- this will **not be compiled**
- `convertCSV2HTML` declares that it may throw a `FileNotFoundException`, but the called methods may generate other types of `IOExceptions` as well

Runtime Exceptions

- **runtime exceptions** represent problems that occur when a program is executed that cannot be (easily) anticipated, neither by the programmer, nor by the compiler
- often runtime exception indicate failed pre-conditions (object not initialised, array too small, wrong state, ..)
- runtime exceptions are direct or indirect subclasses of `java.lang.RuntimeException`
- the catch-or-specify requirement does **not apply** to runtime exceptions

Common Runtime Exceptions

```
Object object = null;
object.toString();
```

Common Runtime Exceptions

```
Object object = null;
object.toString();
```

- an instance of **java.lang.NullpointerException** is thrown when a method is invoked on an uninitialised object reference

Common Runtime Exceptions ctd

```
Object aDate = new Date();
String aString = (String)aDate;
```

Common Runtime Exceptions ctd

```
Object aDate = new Date();
String aString = (String)aDate;
```

- an instance of **java.lang.ClassCastException** is thrown when a runtime type cast fails

Common Runtime Exceptions ctd

```
String[] anArrayOfStrings = new String[10];
String aString = anArrayOfStrings[11];
```

Common Runtime Exceptions ctd

```
String[] anArrayOfStrings = new String[10];
String aString = anArrayOfStrings[42];
```

- an instance of **java.lang.ArrayIndexOutOfBoundsException** is thrown when a non-existing array slot is accessed

Defining Exceptions

- defining exception is easy: subclass `Exception` of one of its subclasses
- usually, exceptions have only the state defined in the superclasses (in `Throwable`)
- this means that only the constructors have to be implemented
- code generators can be used
- in Eclipse:
  Source > Generate Constructors from Superclass

Example: CSVException

- use case: when parsing tabular data (CSV format), we expect a fixed number of columns in each row
- we can use a custom exception type to deal with dirty data
- we regard this as an IO (input/output) problem, so we subclass `java.io.IOException`

```
John,Smith,Computer Science
Tim,Taylor,Software Engineering
Kate,Wilson
Harry,Brown,Information Technology
```

there is a missing data value in row 3 - we can use an exception to deal with this situation

Example: CSVException ctd

```
public class CSVException extends java.io.IOException {
        public CSVException(String message, Throwable cause) {
                super(message, cause);
        }
        public CSVException(String message) {
                super(message);
        }
        public CSVException() {
                super("Problems parsing CSV file");
        }
}
```

the message is passed as constructor parameter

- note that it is usually **not necessary** to implement instance variables and methods in exception classes !

Throwing an Exception

```
public List readObjectsFromCSV(File csv) throws IOException {
          ...
          throw new CSVException("Exception parsing CSV file");
          ..
}
```

- note that `CSVException` is a subclass of `IOException` !
- therefore the compiler will accept this

Exception Chaining

- the constructor `CSVException(String message, Throwable cause)` can be used for **exception chaining**
- this feature is used when an exception is caused by another exception (its "cause")
- this makes sure that an exception retains a reference to its cause
- when the stack trace is printed, the stack trace of the cause is printed as well (recursively)

Exception Chaining

```
public List readObjectsFromCSV(File csv) throws IOException {
         String[] row = ...;
         try {
                  String value = row[2];
     }
     catch (ArrayIndexOutOfBoundsException x) {
                  throw new CSVException("Exception parsing CSV
file",x);
     }
         ..
}
```

- for instance, if a row in a CSV file is accessed as an array, and we expect 3 columns of data in each array, then accessing a faulty row (with only 2 values) would cause an `ArrayIndexOutOfBoundsException`
- this can be mapped to a `CSVException` using chaining
- all information about the original exception is retained (= the stack race is available, and will be appended to the stack trace of the new exception)

Documenting Exceptions

- it is good style to document the exceptions
- there are special tags that can be used (`@throws` and `@exception`)
- this tags can be used in the javadoc tool to generate web sites from documented java program code
- for instance, this site is generated using javadoc: http://docs.oracle.com/javase/7/docs/api/overview-summary.html

Documenting Exceptions

```
/**
 * Read data from a CSV file, and return them as list.
 * @param csv an input text file, must conform to the CSV syntax rules
 * @return a list of objects, each row in csv will be converted into an
 * an object that is an element of this list
 * @throws java.io.IOException thrown if the file is not accessible, or is
 * not a valid CSV file
 */


public List readObjectsFromCSV(File csv) throws IOException {
         ..
}
```