5COSC019W - OBJECT ORIENTED DEVELOPMENT

Exceptions

Dr Dimitris C. Dracopoulos

email: d.dracopoulos@westminster.ac.uk

1 Exceptions

Exceptions offer a way to change the program flow control when an error or unexpected event happens.

- Exceptions notify the user of an error.
- Exceptions cannot be overlooked.
- Exceptions localise the error handling in only a few areas of the code. Exceptions are sent to an exception handler, not necessarily the calling method.
- Exceptions can facilitate error recovery when this is possible.
- An exception object is thrown to indicate failure:

```
if (failure)
{
   XxxException e = new XxxException(. . .);
   throw e;
}
```

where `XxxException` is an exception class (used defined or existing in the library).

An Example of Throwing an Exception

```
public class BankAccount {
   public void withdraw(double amount) {
      if (amount > balance)
         throw new IllegalArgumentException(
               "Amount exceeds balance");
      balance = balance - amount;
   }
   ...
}
```

2 What happens when an exception is thrown

The following happen when an exception is thrown:

1. An exception object is created to record the details that went wrong.
2. The run time system changes the normal flow of control, to search back up the call chain, for a place where code which handles the specific type of exception exists.

   That place can be in the same method, in the method that called the one where the exception occurred, or in the method that called the method where the exception occurred and so on. If the exception propagates to the point (top) where the program started and no handling for that exception is found, the program terminates with a message.

Example:

The following program does not have any code which handles (catches) the exception which can be possibly thrown in the `withdraw` method. Therefore, if such an exception occurs, the program will terminate and the rest of the code will not be executed.

```
class BankAccount {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > balance)
            throw new IllegalArgumentException("Amount exceeds balance");

        balance = balance - amount;
   }
}

public class BankAccountExceptionTest {
    public static void main(String[] args) {
        BankAccount b1 = new BankAccount();
        b1.deposit(100.0);  // first deposit
        b1.withdraw(200.0);

        System.out.println("Depositing 300.0");
        b1.deposit(300.0);  // second deposit
    }
}
```

When the code is run, it displays:

```
Exception in thread "main" java.lang.IllegalArgumentException: Amount exceeds balance
        at BankAccount.withdraw(BankAccountExceptionTest.java:10)
        at BankAccountExceptionTest2.main(BankAccountExceptionTest.java:21)
```

The second deposit will never take place as the program exits at the point where the exception is thrown, i.e. before calling `deposit` with 300.0 as an argument.

Example — Catching the Exception:

The previous example can be modified so as to catch the exception and report an error to the user.

```
class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > balance)
            throw new IllegalArgumentException("Amount exceeds balance");

        balance = balance - amount;
   }
}

public class BankAccountExceptionTest2 {
    public static void main(String[] args) {
        BankAccount b1 = new BankAccount();
        b1.deposit(100.0);  // first deposit

        // handle the exception if thrown
        try {
            b1.withdraw(200.0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println("*** Withdraw failed! ***");
            System.out.println(ex.getMessage());
        }
        System.out.println("New balance: " + b1.getBalance());

        System.out.println("Depositing 300.0");
        b1.deposit(300.0);  // second deposit
```

```
        System.out.println("New balance: " + b1.getBalance());
    }
}
```

The program displays:

```
*** Withdraw failed! ***
Amount exceeds balance
New balance: 100.0
Depositing 300.0
New balance: 400.0
```

Although the exception is thrown, in this case it is caught and the message "Withdraw failed" is displayed. The rest of the statements following the `catch` block will be executed this time and the code does not terminate abnormally.

3 Classification of Exceptions

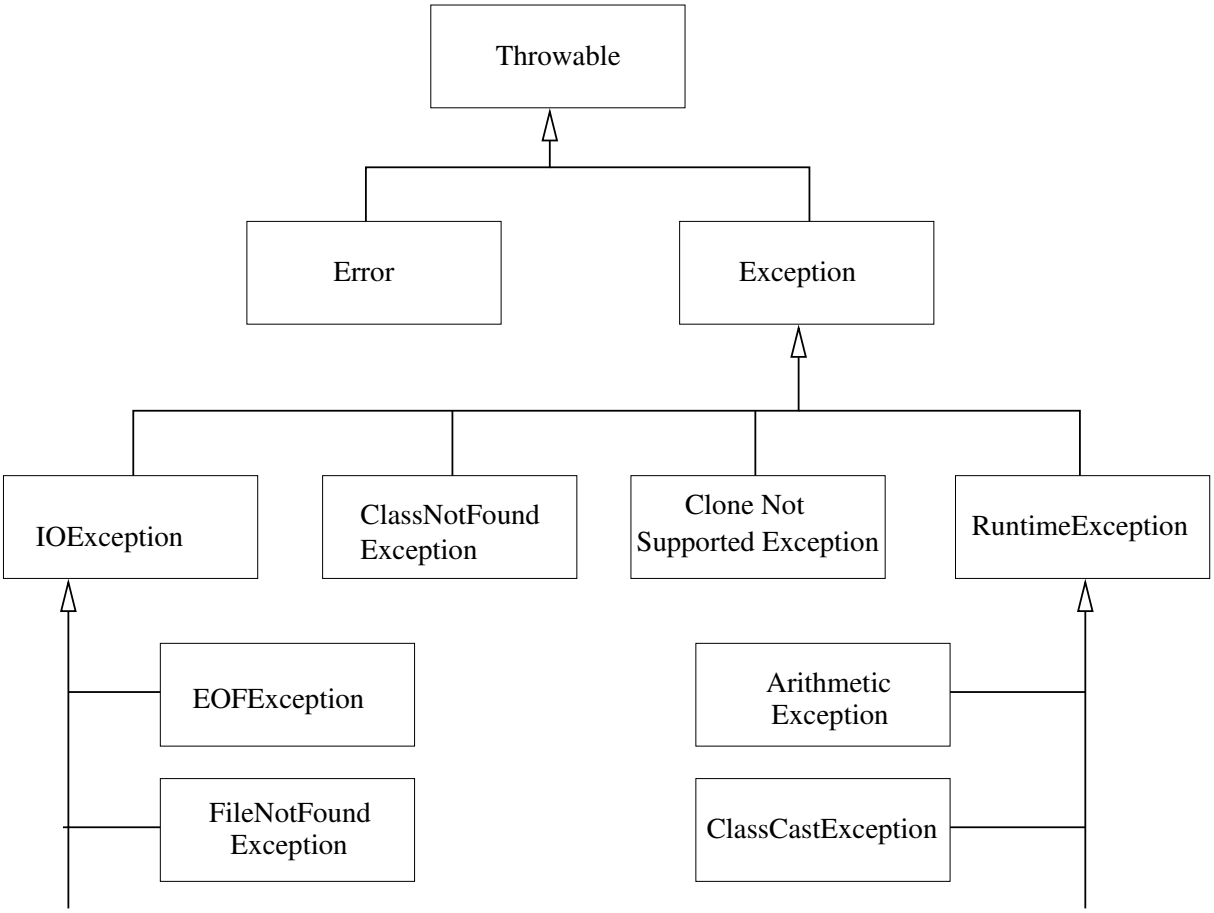

Figure 1: Exception hierarchy.

An exception object is always an instance of a class derived from `Throwable`.

All exceptions happen at run time (therefore the name `RuntimeException` was not properly chosen).

- A `RuntimeException` happens because you made a programming error. Any other exception occurs because a bad thing, such as I/O error, happened to your otherwise good program.

  Such problems occur because:

  - A bad cast
  - An out-of-bounds array access.
  - A null pointer access.

- Exceptions which do not inherit from `RuntimeException` include:

  - Trying to read past the end of a file.
  - Trying to open a malformed URL.

- The `Error` hierarchy describes internal errors and resource exhaustion inside the Java run time (e.g. heap space exhaustion). These exceptions are rare and they should not be caught.

4 Checked vs Unchecked Exceptions

The Java Language Specification calls any exception that derives from the class `Error` or `RuntimeException` an *unchecked* exception.

- A method must *declare* or *catch* all the checked exceptions it throws.
- Otherwise the compiler will give an error message.

  Example: When an I/O operation does not catch or declare as thrown the IOException, the compiler issues:

```
javac myProgram.java

unreported exception java.io.IOException;
must be caught or declared to be thrown
```

Note that an unchecked exception can (but does not have to) be caught, however the compiler will not warn you against uncaught unchecked exceptions (see the first `BankAccount` example in Section 2).

5 How to Catch an Exception (Checked or Unchecked)

```
String filename = "myfile";
FileReader reader = null;
try {
```

```
            reader = new FileReader(filename);
            Scanner sc = new Scanner(reader);
        }
        catch(IOException ex) {
            System.out.println("File not found: " + ex.getMessage());
            ex.printStackTrace();
        }
```

A particular exception class catches all the exceptions of this type and the exceptions which have derived from this exception. For example, in the above code the `catch` block will catch any thrown `IOException` exception or any exception which belongs to a class derived from `IOException` (e.g. `FileNotFoundException`).

However, as a matter of good practice, programs should attempt to catch the more "specific" exceptions, as these indicate specific errors which could be treated in a particular manner.

6 The Flow of Control using Exceptions

Here is what happens when an exception is specified (i.e. when a try block is used inside a code):

1. Statements in try block are executed.
2. If no exceptions occur, catch clauses are skipped.

   3a If exception of matching type occurs, execution jumps to catch clause.

   3b If exception of another type occurs, it is thrown to the calling method.
   The exception can propagate up to `main` method, if none of the other calling methods handle it. If `main` doesn't catch an exception, the program terminates with a stack trace.

7 The General Syntax of a Try Block

```
try
{
   statement
   statement
   ...
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   ...
}
catch (ExceptionClass exceptionObject)
```

```
{
   statement
   statement
   ...
}
...
```

Example:

```
BufferedReader console = null;
try {
   console = new BufferedReader(new InputStreamReader(System.in));
   System.out.println("What is your name?");
   String name = console.readLine();
   System.out.println("Hello,"+name +"!");
}
catch (IOException exception) {
   exception.printStackTrace();
   System.exit(1);
}
catch (Exception ex) {
   ex.printStackTrace();
   ...
}
```

8 The Finally Clause

Assume that an exception terminates the current method.

Then:

- Danger: Can skip over essential code

For example:

```
BufferedReader in;
in = new BufferedReader(new FileReader(filename));
in.read();
in.close();
```

- Must execute in.close() even if exception happens.
- Use finally clause for code that must be executed ”no matter what”.

8.1 How the Finally clause works

The `finally` clause is:

- Executed even the `try` block comes to normal end.
- Executed if a statement in try block throws an exception, before exception is thrown out of try block.
- Can also be combined with catch clauses.

```
try {
   statement
   statement
   ...
}
catch (ExceptionClass exceptionObject) {
   statement ...
}
finally {
   statement
   ...
}
```

8.2 Example:

```
import java.util.*;
import java.io.*;

public class ConsoleReadExample {
    public static void main(String[] args) {
        BufferedReader console = null;
        try {
            console = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("What is your name?");
            String name = console.readLine();
            System.out.println("Hello,"+name +"!");
        }
        catch (IOException exception) {
            exception.printStackTrace();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
        finally {
            try {
                if (console != null)
                    console.close();
```

```
            }
            catch(IOException ex) {
                System.out.println("close() failed");
            }
        }
    }
}
```

Note that the code inside `finally` is enclosed in an additional `try` block. This is required in this specific example, because the `close()` method inside `finally` may throw an `IOException` exception. The compiler will report an error if the inner `try` is omitted, because `IOException` is a checked exception.

9 Catching More than 1 Exception with 1 Catch Clause

In Java 7 and newer:

- Use the vertical bar (|)

```
catch (IOException|SAXException ex) {
    // .. do something, e.g.
    logger.log(ex);
}
```

10 The `try-with-resources` Statement

Java 7 and newer:

A `try` associated with one or more resources.

- A *resource* is an object that must be closed after the program is finished with it.
- Any object that implements the `java.lang.AutoCloseable` or `java.io.Closeable` interfaces.

The resource will be closed whether the `try` method completes normally or interrupted because of an exception being thrown.

```
String readLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

In this example, the resource declared in the `try`-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the `try` keyword. Since the class `BufferedReader` implements the interface `java.lang.AutoCloseable` (from Java 7 onwards) the object will be closed regardless of whether the try statement completes normally or abruptly, because of an `IOException` which might be thrown from the `readLine` method of `BufferedReader`.

- A `try-with-resources` can be associated with `catch` and `finally` blocks.
- Any `catch` and/or `finally` blocks will be executed after the declared resource(s) are closed.
- You can specify more than one "closeable" resources if you separate them with a semicolon, the statement separator (;)

11 Designing your own Exceptions

The developer can create his/her own exceptions by extending the appropriate type of Exception. For example:

```
if (amount > balance)
   throw new InsufficientFundsException(. . .);
```

In this case, the following decisions are made for the new user defined exception called `InsufficientFundException`:

- Make it an unchecked exception. The programmer could have avoided it by checking the balance first, therefore there is no need to enforce catching the exception by the compiler.
- Extend RuntimeException.
- Supply two constructors (one takes an argument, explanation of why the exception occurred).

Example:

```
public class InsufficientFundsException extends RuntimeException {
   public InsufficientFundsException() {
   }

   public InsufficientFundsException(String reason) {
      // call the corresponding constructor of the parent Exception
      super(reason);
   }
}
```

Note that the second constructor calls the parent constructor, i.e. the constructor of class `RuntimeException`.

- If the client code can reasonably be expected to recover from an exception, make it a **checked exception**.
- If the client code cannot do anything to recover make it an **unchecked exception**.

12 Exception Specification

If a method does not catch a *checked* exception (whether a checked user-defined exception or a checked exception available by the language) then it should declare it as thrown (if not the compiler will spot this error and force you to change your code):

```
public void read(BufferedReader in) throws IOException {
    value = Double.parseDouble(in.readLine());
    name =in.readLine();
}
```

If a method declares an Exception as thrown, then it is the responsibility of the caller of that method to catch the exception. If the caller declares this as also thrown, the exception propagates up to the `main()` method. If this does not catch the exception (but it simply declares as "thrown" as well), then the JVM will report you when the exception occurs and prints the `StackTrace`.

Example:

```
import java.io.*;

public class ExceptionDeclarationExample {
    double value;
    String name;

    public void read(BufferedReader in) throws IOException {
          System.out.print("Enter a value: ");
          value = Double.parseDouble(in.readLine());
          System.out.print("Enter a name: ");
          name = in.readLine();
    }

    public void print() {
        System.out.println("value=" + value);
        System.out.println("name=" + name);
    }

    public static void main(String[] args)  {
        ExceptionDeclarationExample e = new ExceptionDeclarationExample();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        try {
```

```
            e.read(br);
        }
        catch (IOException ioex) {
            System.out.println("read() method failed");
            ioex.printStackTrace();
        }

        e.print();
    }
}
```

Important note: *Specifying an exception does not result in an overhead in your program! Only when the exception actually occurs you will have an overhead cost..*

13 Advantages in using Exceptions

1. Localisation of error handling in the code
   (a) Separating Error-Handling Code from "Normal" Code
   (b) Propagating Errors Up the Call Stack: a method can propagate an error to a caller method if it cannot/don't want to deal with it itself.
2. Grouping and Differentiating Error Types