5COSC019W - OBJECT ORIENTED PROGRAMMING
Lecture 7: Exceptions

Dr Dimitris C. Dracopoulos

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.
- Throw an exception object to indicate failure:

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

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;
   }
   ...
}
```

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 runtime 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.

Example:

```
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);
```

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)
```

Example — Catching the Exception:

```
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;
    }
}
```

Example — Catching the Exception:

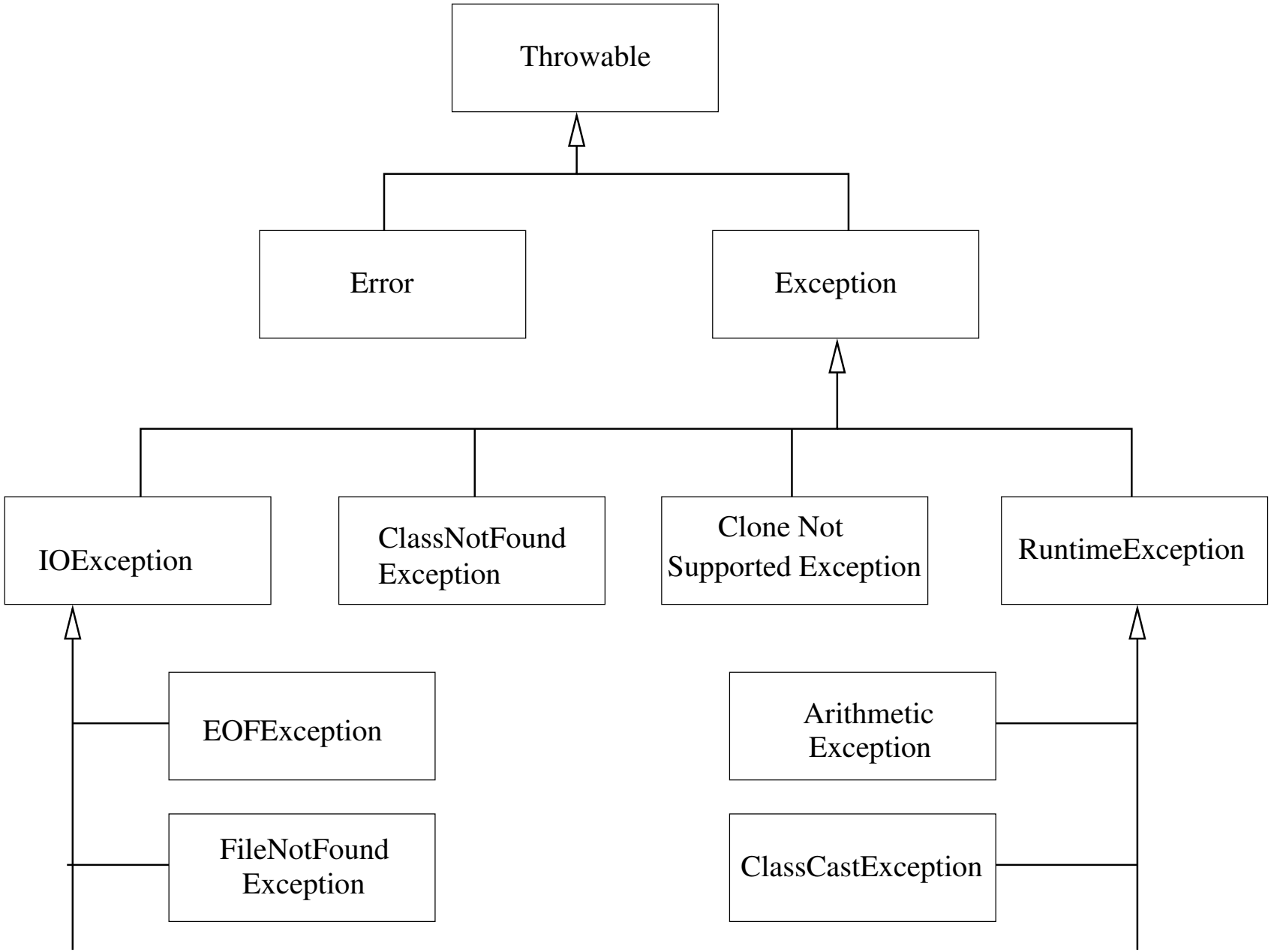
```
public class BankAccountExceptionTest2 {
    public static void main(String[] args) {
        BankAccount b1 = new BankAccount();
        b1.deposit(100.0);  // first deposit

        // 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());
    }
}
```

Classification of Exceptions



Classification of Exceptions (cont'ed)

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.

Classification of Exceptions (cont'ed)

- 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.

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
```

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(FileNotFoundException ex) {
  System.out.println("File not found: " +
                                        ex.getMessage());
    ex.printStackTrace();
}
```

A class for an Exception catches all the exceptions of this type and the exceptions which have derived from this exception.

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.

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();
   ...
}
```

The Finally Clause

Assume that an exception terminates the current method.
Then:

- Danger: Can skip over essential code

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".

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
   ...
}
```

Example:

```
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");
            }
        }
    }
```

Catching More than 1 Exception with 1 Catch Clause

Java 7 and newer:

- Use the vertical bar (|)

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

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.

The try-with-resources Statement

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

- 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 (;)

Designing your own Exceptions

The developer can create their own exceptions by extending the appropriate type of Exception.
Example:

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

- Make it an unchecked exception–programmer could have avoided it by checking the balance first
- 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);
    }
}
```

Designing your own Exceptions (cont'ed)

- 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**.

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, then the JVM will report you when the exception occurs and prints the `StackTrace`.

Exception Specification (cont'ed)

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

```java
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();
    }
}
```

Advantages in using Exceptions

1. Localisation of error handling in the code
   1.1 Separating Error-Handling Code from "Normal" Code
   1.2 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