

Tutorial: Week 18

Alistair Baron

a.baron@comp.lancs.ac.uk

C28, infolab

<http://www.comp.lancs.ac.uk/~barona>

Chuck Norris Factoids



- When Chuck Norris crosses the road, the cars have to look both ways.
- Chuck Norris is a stunt double for Optimus Prime.
- Chuck Norris beat the sun in a staring contest.

Exceptions

- When you'll use Exceptions:
 - When you have to!
 - Checked Exceptions in the Java API dictate your method must either catch the exception or throw it.
 - [http://java.sun.com/javase/6/docs/api/java/lang/Double.html#parseDouble\(java.lang.String\)](http://java.sun.com/javase/6/docs/api/java/lang/Double.html#parseDouble(java.lang.String))
 - When you want to make sure something “bad” hasn't happened.
 - Unchecked Exceptions in the Java API can be thrown if, for example, a parameter is invalid.
 - [http://java.sun.com/javase/6/docs/api/java/lang/String.html#matches\(java.lang.String\)](http://java.sun.com/javase/6/docs/api/java/lang/String.html#matches(java.lang.String))
 - You can also create your own exceptions to account for **exceptional** circumstances.
 - Slide later showing this.

Dealing with Exceptions

- Whenever you know a method could throw an exception you need to either:
 - use a *try-catch block*:

```
try {  
    double d = Double.parseDouble("12.00");  
}  
catch(NumberFormatException ex) {  
    //do something about it.  
}
```

- Declare that the calling method throws the exception:

```
public void formatString(String s) throws NumberFormatException {  
    //....  
}
```

Finally

- You can use a *finally-block* to declare code which should be executed no matter what.

```
try {  
    //....  
    if(fileString.matches(regex))  
        //.....  
}  
finally {  
    /* stuff here is always done whether or not exception is thrown, e.g.  
    closing I/O stream or database connection.  
}
```

- Note: it is legal (and sometimes useful) to have no catch statement, as long as a *finally-block* exists.

Using your own exceptions

- Dead easy:

```
public class MyException extends Exception {
    public MyException() {
        super();
        //anything else you want to do
    }
    public MyException(String msg) {
        super(msg);
        //anything else
    }
}
```

```
public void doSomething() throws MyException {
    If(something wrong) {
        throw new MyException("Something bad's happened");
    }
}
```

Questions

- What exception types can be caught by the following handler?

```
catch (Exception e) {  
}
```

Answer: This handler catches exceptions of type `Exception`; therefore, it catches any exception, as all exceptions extend `Exception`.

- What is wrong with using this type of exception handler?

Answer: This can be a poor implementation because you are losing valuable information about the type of exception being thrown and making your code less efficient. By catching each type of exception separately you can run different code dependant on the type of exception.

Questions

- Is there anything wrong with this exception handler as written?
Will this code compile?

```
try {  
    //do something  
}  
catch (Exception e) {  
}  
catch (ArithmeticException a) {  
}
```

Answer: This first handler catches exceptions of type `Exception`; therefore, it catches any exception, including `ArithmeticException`. The second handler could never be reached. This code will not compile.

An Example

```
public class ExceptionTest {
    public static void main(String[] args) {
        System.out.println("A");
        try {
            System.out.println("B");
            if(Integer.parseInt(args[0]) > 5)
                System.out.println("C");
            else
                System.out.println("D");

            System.out.println("E");
        }
        catch(NumberFormatException ex) {
            System.out.println("F");
        }
        catch(Exception ex) {
            System.out.println("G");
        }
        finally {
            System.out.println("H");
        }
        System.out.println("I");
    }
}
```

Test this code with different inputs on the command line: 10, “ten” and nothing. What output is given, can you understand why?

What happens if the second catch statement isn't present and no input is given?

What happens if you insert *return;* in one of the catch statements, and that exception is caught. Is the finally block called? What about the final *I*.

What if you include *System.exit(1)* in place of the above *return;*, what happens this time? Can you understand the difference? Please ask if this is puzzling.

Experiment with the code and see the result.