



Victoria University
of Wellington, New Zealand
*Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa*



SWEN221 Software Development Exceptions

Thomas Kuehne

Victoria University

(slides modified from slides by David J. Pearce &
Nicholas Cameron & James Noble & Petra Malik)

Error Handling

- What if there is an error?

```
public static int readFromFile (File f) {  
    if(!f.exists()) return -1; // return error code  
    ...  
}
```

- This may be fine in many situations, **but**
 - failure checking pollutes the standard scenario.
 - what about `Object ArrayList.get(int index)`?
 - relies on client to actively check for error codes.
 - does not promote robust programs.

Introducing Exceptions

A language construct designed to deal with

- errors
- exceptional behaviour

Exceptions allow problems to be dealt with

- gracefully
- in a client-specific manner

Introducing Exceptions

```
class ArrayList {  
    public int size() {...}  
    public Object get(int index) {  
        if(0 <= index && index < size()) { ... }  
        else throw new ArrayIndexOutOfBoundsException();  
    }  
}  
...  
ArrayList v = new ArrayList();  
  
try {  
    v.get(0); // error occurs here  
} catch(ArrayIndexOutOfBoundsException e) {  
    // deal with error here  
}
```

- Exceptions signal *exceptional behaviour*
 - Method can terminate *normally* by returning result
 - Or *abruptly*, by throwing an exception

What gets printed?

```
void b() { throw new NullPointerException(); }

void a() {
    try { b();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("a");
    }
}

public static void main(...) {
    try { a();
    } catch (NullPointerException e) {
        System.out.println("main");
    }
}
```

A: "a"
B: "main"

- When an Exception is thrown, control passes to the enclosing try-catch block that matches the exception type

Nesting Exceptions

- Exceptions might have an associated **message** and/or a **cause**
- **Methods:** `getMessage()`, `getCause()`, `getStackTrace()`, **etc.**

```
try {  
    lowLevelOp();  
} catch (LowLevelException e) {  
    throw new HighLevelException("Explanation...", e);  
}
```

Nesting Exceptions

- Exceptions are a **language** feature
- “Chain of exceptions” is a **programming style**; a convention that programmer can use to propagate complex exceptions

Resource Handling

```
void compute() {  
    FileOutputStream tmp = new FileOutputStream("tmp.dat");  
  
    try {  
        ... // do some complicated computation  
        ... // write results to temporary file  
  
        tmp.close();  
        new File("tmp.dat").delete(); // delete temporary file  
    } catch(IOException e) {  
        ... // report error and return  
    }  
}
```

- This code has a problem

Finally

```
void compute() {  
    FileOutputStream tmp = new FileOutputStream("tmp.dat");  
  
    try {  
        ... // do some complicated computation  
        ... // write results to temporary file  
    } catch(IOException e) {  
        ... // report error and return  
    } finally {  
        tmp.close();  
        new File("tmp.dat").delete(); // delete temporary file  
    }  
}
```

- Finally clause

- gets executed regardless of how try-block exited (e.g. normal execution, caught exception or uncaught exception)
- useful for “cleaning up” allocated resources

What is the difference?

```
try { /*a*/ }  
catch(SomeException e) { /*b*/ }  
/*c*/
```

```
try { /*a*/ }  
catch(SomeException e) { /*b*/ }  
finally { /*c*/ }
```

```
try {  
  
} catch(SomeException e) {  
  
}  
finally {  
  
}
```

```
try {  
  
} catch(SomeException e) {  
  
}
```

Do we get the same output?

```
try { /*a*/ }  
catch (SomeException e) { /*b*/ }  
/*c*/
```

```
try { /*a*/ }  
catch (SomeException e) { /*b*/ }  
finally { /*c*/ }
```

```
try {  
    throw new SomeException();  
}  
catch (SomeException e) {  
    System.out.println("Problem");  
}  
finally {  
    System.out.println("Cleaned");  
}
```

```
try {  
    throw new SomeException();  
}  
catch (SomeException e) {  
    System.out.println("Problem");  
}  
  
System.out.println("Cleaned");
```

A: YES
B: NO

Do we get the same output?

```
try { /*a*/ }  
catch (SomeException e) { /*b*/ }  
/*c*/
```

```
try { /*a*/ }  
catch (SomeException e) { /*b*/ }  
finally { /*c*/ }
```

```
try {  
    throw new AnotherException();  
}  
catch (SomeException e) {  
    System.out.println("Problem");  
}  
finally {  
    System.out.println("Cleaned");  
}
```

```
try {  
    throw new AnotherException();  
}  
catch (SomeException e) {  
    System.out.println("Problem");  
}  
  
System.out.println("Cleaned");
```

A: YES
B: NO

Do we get the same output?

```
try { /*a*/ }  
catch(SomeException e) { /*b*/ }  
/*c*/
```

```
try { /*a*/ }  
catch(SomeException e) { /*b*/ }  
finally { /*c*/ }
```

```
try {  
    throw new SomeException();  
}  
catch(SomeException e) {  
    throw e;  
}  
finally {  
    System.out.println("Cleaned");  
}
```

```
try {  
    throw new SomeException();  
}  
catch(SomeException e) {  
    throw e;  
}  
  
System.out.println("Cleaned");
```

A: YES
B: NO

Types of Exceptions

- Unchecked Exceptions

Subclasses of `RuntimeException` and `Error`

- e.g., `NullPointerException` and `IndexOutOfBoundsException`
- do not require explicit declaration / catching

- Checked Exceptions

- Subclasses of `Exception`, but not `RuntimeException`
- e.g., `IOException`
- must be declared in a **method's throws clause**:
 - compile-time error, unless all thrown exceptions – even those caused by called methods – are caught or declared

Checked Exceptions

```
void a() throws IOException {  
    ...  
    throw new IOException("Lost access to file");  
}
```

```
void b() { a(); } // ERROR
```

```
void b() throws IOException { a(); } // OK
```

```
void b() { // OK  
    try {  
        a();  
    } catch(IOException e) {...}  
}
```

Checked Exceptions

- Why checked exceptions?
 - Signal recoverable problems
 - e.g. `FileNotFoundException` (interactive application)
versus `NullPointerException` **or** `ArithmeticException`
 - programs can **typically recover** from such errors
 - **Force** clients to deal with the problem
 - programmer cannot ignore potential errors
 - compile time errors are better than runtime errors!

Checked vs Unchecked

- Checked exceptions:
 - signal recoverable problems / expected abnormalities
- Unchecked exceptions:
 - make exception handling / declarations feasible

Turn Checked Exceptions into Errors

- Scenario (this really happened)
 - programmed “Simple Program Interpreter”
 - the first version of the language had no InputStatement
 - hence, no need for declaring “**throws** IOException”
 - added InputStatement, which reads input
 - But, InputStream.read() throws IOException
 - What to do?

Problems with Checked Exceptions

- When facing the following:

```
abstract class Statement {  
    public abstract void execute();  
}  
class InputStatement extends Statement {  
    public void execute() {  
        InputStream input = ...;  
        input.read(); // throws IOException  
    }  
}
```

- The options are to
 - declare `Statement.execute()` (+ all subclasses) **throws `IOException`**, or
 - deal with Exception in `InputStatement` somehow

What not to do!

```
abstract class Statement {  
    public abstract void execute();  
}  
  
class InputStatement extends Statement {  
    public void execute() {  
        InputStream input = ...;  
        try { input.read(); } // throws IOException  
        catch(Exception e) {}  
    }  
}
```

- all exceptions (including `RuntimeException`) are “swallowed”

What not to do!

```
abstract class Statement {  
    public abstract void execute();  
}  
  
class InputStatement extends Statement {  
    public void execute() {  
        InputStream input = ...;  
        try { input.read(); } // throws IOException  
        catch(IOException e) {}  
    }  
}
```

- better, but IOExceptions are still “swallowed”

What not to do!

```
abstract class Statement {  
    public abstract void execute();  
}  
  
class InputStatement extends Statement {  
    public void execute() {  
        InputStream input = ...;  
        try { input.read(); } // throws IOException  
        catch(IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- better still, but this should just be a useful default to be replaced by proper handling

Turn Checked Exceptions into Errors

```
abstract class Statement {  
    public abstract void execute();  
}  
  
class InputStatement extends Statement {  
    public void execute() {  
        InputStream input = ...;  
        try { input.read(); } // throws IOException  
        catch(IOException e) { throw new Error(e); }  
    }  
}
```

- Exception **rethrown** as unchecked exception

Try with resource

```
try (FileOutputStream tmp = new FileOutputStream("out.dat")) {  
    ... // do some complicated computation  
    ... // writing results to file  
}  
catch (IOException e) {  
    ... // report write error and return  
}
```

- Since Java7: Shortcut solution to close resources
 - convenient alternative to finally
 - any object that implements `java.lang.AutoCloseable` can be used as a resource
 - Not as flexible; e.g., here we do not have the option of deleting the file, we'd only close it.

Further Reading ...

- <http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html>
- <http://www.octopull.demon.co.uk/java/ExceptionalJava.html>
- <http://www.artima.com/intv/handcuffs.html>
- <http://www.mindview.net/Etc/Discussions/CheckedExceptions>