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

SWEN221: Software Development

05: Exceptions

David J. Pearce & Marco Servetto & Nicholas Cameron & James Noble & Petra Malik
Computer Science, Victoria University

Error Handling

```
/**
if n >= 0 return n! else return -1
*/
public static int factorial(int n) {
  if(n<0)return -1;//represents an error condition
  if(n==0)return 1;
  return n*factorial(n-1);
}
```

Error Handling

```
/**
if n >= 0 return n! else return -1
*/
public static int factorial(int n) {
  if(n<0)return -1;//represents an error condition
  if(n==0)return 1;
  return n*factorial(n-1);
}
```

- Pre-exception solution. Still used in some situations…
  - But, what about `Object ArrayList.get(int index)`?

Error Handling

```
/**
if n >= 0 return n! else return -1
*/
public static int factorial(int n) {
  if(n<0)return -1;//represents an error condition
  if(n==0)return 1;
  return n*factorial(n-1);
}
```

- Pre-exception solution. Still used in some situations…
  - But, what about `Object ArrayList.get(int index)`?
  - Relies on client to check for error
    - Amazing how often this is neglected. E.g. C `malloc()`
    - Does not promote *robust* programs

Error Handling

```
/**
if n >= 0 return n! else return -1
*/
public static int factorial(int n) {
  if(n<0)return -1;//represents an error condition
  if(n==0)return 1;
  return n*factorial(n-1);
}
```

- Pre-exception solution. Still used in some situations…
  - But, what about `Object ArrayList.get(int index)`?
  - Relies on client to check for error
    - Amazing how often this is neglected. E.g. C `malloc()`
    - Does not promote *robust* programs
  - Error-checking code often horribly entangled with normal code

Introducing Exceptions

A language construct designed to deal with

- errors
- exceptional behaviour

Sometimes they can be also useful for terminating complex computations.

Differences between:

- **return** new Foo();
- **throw** new Foo();

Introducing Exceptions

```
class ArrayList {
  public int size() {...}
  public Object get(int index) {
    if(0 <= index && index < size()) {
      ... }
    else {
      throw new ArrayIndexOutOfBoundsException();
}}}
```

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
catch(ArrayIndexOutOfBoundsException e) {
  // deal with error
}
```

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
catch(ArrayIndexOutOfBoundsException e) {
  // deal with error
}
```

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

What gets printed?

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

void foo() {
 try { bar(); }
 catch(IndexOutOfBoundsException e) {
   System.out.println("foo");}}

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

A: “foo”

B: “main”

What gets printed?

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

void foo() {
 try { bar(); }
 catch(IndexOutOfBoundsException e) {
   System.out.println("foo");}}

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

A: “foo”
B: “main”

- When an Exception is thrown
  - Control passes to enclosing try-catch block which matches exception type

Try-Catch Blocks

This try-catch block doesn't match

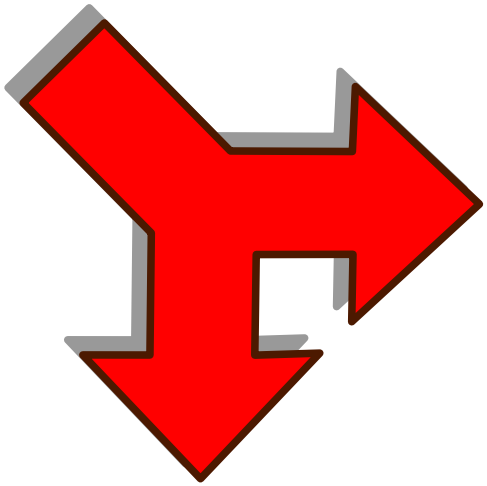
```
void bar() { throw new NullPointerException(); }

void foo() {
 try { bar(); }
 catch(IndexOutOfBoundsException e) {
   System.out.println("foo");}}

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

Control transferred here

- When an Exception is thrown
  - Control passes to enclosing try-catch block which matches exception type



Nesting Exceptions

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

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

- Exceptions are a **language** feature
- Chain of exceptions (`getCause()`) is a **library** feature, that is, a convention programmers use to propagate complex exceptions

Resource Handling

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

- This code has a problem

Finally

```
void foobar() {
  FileOutputStream tmp = new FileOutputStream("tmp.dat");
  try {
    ... // do some complicated computation
    ... // writing results to temporary file
  }
  catch(IOException e) {
    ... // report write 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. by normal execution, by 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*/ }
```

Write down a minimal example of code where the presence of finally produces a different behavior!

```
try {

}
catch(SomeException e) {

}
finally {

}
```

```
try {

}
catch(SomeException e) {

}
```

Is this a valid solution?

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

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

A: YES

B: NO

```
try {
  throw new SomeException();
}
catch(SomeException e) {
  System.out.println(“Hello”);
}
finally {
  System.out.println(“World”);
}
```

```
try {
 throw new SomeException();
}
catch(SomeException e) {
 System.out.println(“Hello”);
}

System.out.println(“World”);
```

Is this a valid solution?

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

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

A: YES

B: NO

```
try {
  throw new AnotherException();
}
catch(SomeException e) {
  System.out.println(“Hello”);
}
finally {
  System.out.println(“World”);
}
```

```
try {
  throw new AnotherException();
}
catch(SomeException e) {
 System.out.println(“Hello”);
}

System.out.println(“World”);
```

Is this a valid solution?

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

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

A: YES

B: NO

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

```
try {
 throw new SomeException();
}
catch(SomeException e) {
 throw e;
}

System.out.println("World");
```

Types of Exception

- Unchecked Exceptions

Subclasses of `RuntimeException` and `Error`

  - E.g. `NullPointerException` and `IndexOutOfBoundsException`

- Checked Exceptions
  - Subclasses of `Exception`, but not `RuntimeException`
  - e.g. IOException
  - Must be declared in a method's **throws** *clause*:
    - If it throws one, or doesn't catch one thrown by called method
    - Otherwise compile-time error

Checked Exceptions

```
void foo() throws IOException {
  ...
  throw new IOException(
    "Lost contact with the hard disk during the read process");
  }


void bar() { foo(); }                          //ERROR

void bar()throws IOException { foo(); }        //OK

void bar(){                                    //OK
  try{ foo(); }
  catch(IOException e){...}
  }
```

Checked Exceptions

- Why checked exceptions?
  - Signal recoverable problems
    - E.g. `FileNotFoundException` (in an interactive program) versus `NullPointerException` or `ArithmeticException`
    - Programs can **typically recover** from such errors
  - ***Force*** clients to deal with problem
    - Programmer cannot ignore possible error
    - Compile time errors are better than runtime errors!

Checked vs Unchecked

- Checked exceptions:
  - Signal recoverable problems / expected behaviour
- Unchecked exceptions:
  - Observed bug (not always a fatal one)

Deal with (Un)Checked exceptions

```
class MyCheckedException extends Exception{

  MyCheckedException(){ super(); }
  MyCheckedException(String m){ super(m); }
  MyCheckedException(String m, Throwable c){ super(m,c); }
}
...
try{...}
catch(MyCheckedException e){
  /*provide alternative behaviour*/
}
```

```
class MyUncheckedException extends RunTimeException{

  MyUncheckedException(){super();}
  MyUncheckedException(String m){super(m);}
  MyUncheckedException(String m, Throwable c){ super(m,c);}
}
...
try{...}
catch(MyUncheckedException e){
  /*resume the execution
  usually make sense only in a master slave pattern*/
}
```

Turn Checked Exceptions into Errors

- Checked exceptions means there is still (potential) hope for a solution
  - Sometimes, you know there is no hope
  - You know something just should not happen.
- Scenario (this really happened)
  - Designed and built “Simple Program Interpreter”
  - In first version of language no InputStatement
    - Statements don’t declare “**throws** IOException”
  - Added InputStatement, which reads input!
    - But, InputStream.read() throws IOException
    - What to do?

Problems with Checked Exceptions

- This is the situation:

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

- Options:
  - Declare Statement.execute() (+ all subclasses) throws IOException
  - 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) {}
  }
}
```

- `IOException` is still being “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();
    }
  }
}
```

`IOException` written on the “standard output”; and than “swallowed”!

Sadly, this is the eclipse “template”

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!

Just mark the error as being an observed bug.

(AssertionError would work too...)

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
  }
```

- New in Java7, is a compact and more elegant solution to close resources
  - Convenient alternative to finally
  - Any object that implements java.lang.AutoCloseable (includes java.io.Closeable) can be used as a resource
  - Not as flexible; for example here we do not delete our file, we only close it.

Further Reading …

- Good articles about Java Exceptions
  - 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