

OPI

Lecture 20

Exception handling

Kasper B. Graversen

Kasper Østerbye

Carsten Schuermann

IT University Copenhagen

Today's schedule

Goal:

- to enable you to write your own exceptions.
- to throw exceptions from your own code.

Contents:

- The programmers roles in exceptions handling
- Design principles for exception handling
 - Checked or unchecked exceptions
 - Fitting level of abstraction
 - Recovery strategies
 - How to get context information from the place the exception was thrown
- The try catch finally statement
- Exceptions and inheritance
- Where can exceptions be thrown, and what is the consequences of that?
- Efficiency of exception handling
- Call stack inspection

Motivation I

```
readFile {  
  open the file;  
  determine its size;  
  allocate that much memory;  
  read the file into memory;  
  close the file;  
}
```

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Motivation I

```
errorCodeType readFile {  
    initialize errorCode = 0;  
    open the file;  
    if (theFileIsOpen) {  
        determine the length of the file;  
        if (gotTheFileLength) {  
            allocate that much memory;  
            if (gotEnoughMemory) {  
                read the file into memory;  
                if (readFailed) {  
                    errorCode = -1;  
                }  
            } else {  
                errorCode = -2;  
            }  
        } else {  
            errorCode = -3;  
        }  
    }  
}
```

```
...  
}  
close the file;  
if (theFileDintClose && errorCode == 0) {  
    errorCode = -4;  
} else {  
    errorCode = errorCode and -4;  
}  
} else {  
    errorCode = -5;  
}  
return errorCode;  
}
```

Nice thought:

Let's separate code and error handling code
(note: no clean up code is presented)

Motivation II –

The two programmers in Exception handling

The programmer of the class

- Knows the implementation, and when an error situation occurs
- Do not know what to do with the error.
- Implements the class independent of any specific usage

The user of the class (You?)

- Knows that something might go wrong, but not where.
- Knows how to handle the error
- Uses the class in a specific context

Example

```
public static Class.forName(String className)  
    throws ClassNotFoundException
```

Parameters:

className - the fully qualified name of the desired class.

Returns:

the Class object for the class with the specified name.

Throws:

LinkageError - if the linkage fails

ExceptionInInitializerError - if the initialization provoked by this method fails

ClassNotFoundException - if the class cannot be located

The programmer of Class

- Finds out that a class does not exist, but does not know how you want to handle that.
- Finds out that the class depends on a class which cannot be loaded, but does not know how you want to handle that.
- Finds the class, but it cannot be initialized. How do you want to handle that?

usage example

```
class Stack {
    LinkedList elements = new LinkedList();
    public void push(Object o) {
        elements.add(o);
    }
    public Object pop() throws EmptyException {
        if(elements.size() == 0)
            throw new EmptyException();
        return elements.removeLast();
    }
    public class EmptyException
        extends Exception {
    }
}
```

class programmer

```
Stack s = new Stack();
try {
    System.out.println( s.pop() );
}
catch(Stack.EmptyException ee) {
    ...
}
```

class user

Checked exceptions

- **Checked by the compiler**

A method foo might declare that it throws one or more Exceptions.

If your code call foo, the compiler will check that:

- Either you enclose the call in a try-catch block, which catches the Exceptions mentioned in foo.
- Or your own code declares that it does too throws the same exceptions.

All exceptions are checked, except from subclasses of RuntimeException.

```
void myMethod() {
```

```
    ...
```

```
    try {
```

```
        ...
```

```
        Class cl = Class.forName("...");
```

```
        ...
```

```
    }
```

```
    catch(ClassNotFoundException e) {
```

```
        ...
```

```
    }
```

```
}
```

----- or -----

```
void myMethod() throws ClassNotFoundException {
```

```
    ...
```

```
    Class cl = Class.forName("...");
```

```
    ...
```

```
}
```


Unchecked exceptions

- **Not checked by the compiler**

A method `foo` might throw an unchecked exception without you knowing it.

If your code call `foo`, the compiler will not tell you that `foo` might throw an exception.

You are therefore likely not to handle that situation.

Rationale:

- Every method call `o.foo()` can throw a `NullPointerException` if `o` is null.
- It is not practical that correct code (that cannot fail) is polluted with error checks

```
try {  
    try {  
        Foo f = new Foo();  
    } catch (ClassNotFoundException cnfe) {...}  
    f.bar();  
} catch (NullPointerException e) {...}
```

- Similarly with arithmetic errors, array stores, class casts, and other.
- The rule of thumb is that if the pre-conditions for some method is broken, the method may throw an unchecked exception.

Checked or unchecked Exceptions?

Errors – use checked Exceptions

The provider (class programmer) is not in a position to ensure that the operation can be completed

- that the network is working
- a server is running
- the connection to the phone is not working

If the provider (class programmer) knows that this can happen, the client *must* deal with that potential problem.

Contract violations (pre conditions) – use unchecked Exceptions

The client did not ensure the pre-condition.

- used null argument
- tried to pop empty stack
- tried to read past end of array

If the provider finds out that the pre-condition is not satisfied, the provider method should throw an unchecked exception.

The try catch finally statements

```
try {  
    Statement0;  
}  
catch(ExceptionType1 ex1) {  
    Statement1;  
}  
catch(ExceptionType2 ex2) {  
    Statement2;  
}  
finally {  
    Statement3  
}
```

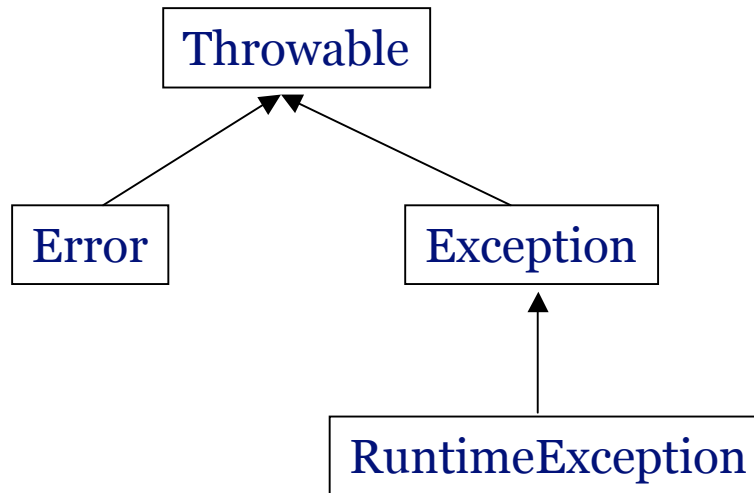
- If Statement0 terminates normally, catch statements are ignored.
- If Statement0 terminates abnormally by throwing an exception e of type E, the first catch which matches E is located (if any).
- If a corresponding type is found, e is bound to the corresponding ex_i, and Statement_i is executed.
- If no corresponding type is found the entire statement terminates with exception e of type E.
- **No matter** what happens in Statement0 or any of the catch statements, Statement3 is executed as the last statement.
- If statement0 throws exception ExceptionType1 and statement1 throws ExceptionType2 what happens?

How final is finally?

```
String pip() {  
    try {  
        foo();  
        return "Hans";  
    }  
    catch(FooException e) { System.out.println("123"); }  
    finally { return "Grete"; }  
}
```

- A Finally block will always be the last code to be executed in a method.
 - Thus, pip() will return "Grete".
 - "Grete" is returned if an exception is raised in the foo() call no matter which exception it is
- If the result of the try-catch part is to throw the exception e, the finally can throw an other exception instead.
 - Most certainly do not do this.... But Java let's you.
- Both cases is needed in rare circumstances.
- All finally blocks from the top of the stack to a matching catch block are executed before the matching catch is executed.

Java exception types hierarchy



An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch. *Most such errors are abnormal conditions.*

A method is not required to declare in its throws clause any subclasses of **Error** that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Direct Known Subclasses of Error:

`AnnotationFormatError`, `AssertionError`, `AWTError`, `CoderMalfunctionError`, `FactoryConfigurationError`, `LinkageError`, `ThreadDeath`, `TransformerFactoryConfigurationError`, `VirtualMachineError`

An orange L-shaped line that starts at the top right, goes left, then turns 90 degrees and goes down to the bottom left.

Design principles

Design principle: Fit level of abstraction

The exceptions thrown by an object should be defined in the same level of abstraction.

Positive examples:

`java.util.Stack` has a method `peek()`, which throws `EmptyStackException`.

`java.util.ArrayList` has a method `get(int index)` which throws `IndexOutOfBoundsException`

`java.lang.Class` has a method `getField(String name)` which throws `NoSuchFieldException`, `NullPointerException` `SecurityException`

– The exception should be informative

- E.g. `IndexOutOfBoundsException` should contain a reference to the `ArrayList` and the index which failed.

– It must be in the vocabulary of the class.

Negative example:

`javax.swing.JTextField` has a method `getText()` If the *underlying* document is null, will give a `NullPointerException`. Should have been `NoDocumentException`.

It has a method `getSelectedText()`, which can throw `IllegalArgumentException`. But there are no arguments!

Exception handling

There are three reasonable ways to deal with exceptions

1. Raise the exception to your level of abstraction (throw a new exception)
2. Try to recover (esp. network app.)
3. Die

1)

```
public Object pop() throws EmptyException {  
    try {  
        return elements.removeLast();  
    } catch (NoSuchElementException no) {  
        throw new EmptyException(this);  
    }  
}
```

2)

```
public Object popOr(Object default) {  
    try {  
        return elements.removeLast();  
    }  
    catch (NoSuchElementException no) {  
        return default;  
    }  
}
```

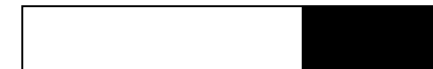
3)

```
public Object pop() throws EmptyException {  
    try {  
        return elements.removeLast();  
    }  
    catch (NoSuchElementException no) {  
        System.exit(1);  
    }  
}
```


Diagram of exceptions

:class::method

- A method call




- try part
- catch part
- finally part
- Each box represents a call stack entity
 - method call
 - try-catch-finally block
- When an exception of type T is thrown, one looks down through the call stack to find a try block which its black mark in the left side, and with a matching catch. The try block which has a matching catch is then marked in the middle, and the execution continues from there.
- If the try block has its black mark in the middle or in the right, it will be ignored when we look for a matching try block down the call stack.

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

- We try to
 - Use the graphic notation
 - Show the propagation down the call stack upon throwing and catching an exception
 - Skipping some code
 - Ensuring the execution of other code

Diagram of exceptions




```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

:ExTry::ExTry

:ExTry::main

Diagram of exceptions



```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

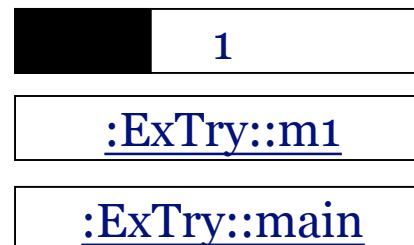


Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

:ExTry::m2

1

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

:ExTry::m3

:ExTry::m2

1

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

RuntimeException

:ExTry::m3

:ExTry::m2

1

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

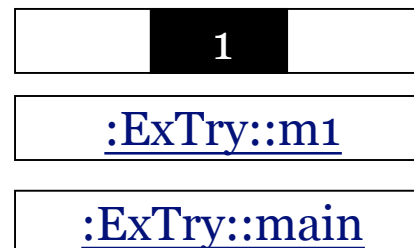


Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

:System.out::println

1

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

1

:ExTry::m1

:ExTry::main

Diagram of exceptions

```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```


:System.out::println

1

:ExTry::m1

:ExTry::main

Diagram of exceptions



```
class ExTry {  
    public static void main(String[] args) {  
        new ExTry().m1();  
    }  
    void m1() {  
        try { m2(); }  
        catch(Exception e) {  
            System.out.println(e);  
        }  
        finally {  
            System.out.println("hello world");  
        }  
    }  
    void m2() { m3(); System.out.println("foo");}  
    void m3() {  
        throw new RuntimeException();  
    }  
}
```

- Note we never printed “foo” on the screen
 - Would “hello world” be printed if our catch was declared to catch “IOException” instead?
- 

:ExTry::main

Diagram of exceptions


```
class D {  
→ void m() { throw new ExceptionA(1); }  
→ public static void main(String[] args) {  
→   D d = new D();  
→   try { d.m();}  
     catch(ExceptionA ea) {  
       d.m();  
     }  
} }
```

- We try to

- Use the graphic notation in a less terse way
- Show that try-blocks are only candidates for a match of an exception, if they have their black mark on the left.

ExceptionA

:D::m

 1

:D::main

Diagram of exceptions

```
class D {  
→ void m() { throw new ExceptionA(1); }  
  public static void main(String[] args) {  
    D d = new D();  
    try { d.m();}  
→    catch(ExceptionA ea) {  
→      d.m();  
    }  
  }  
}
```

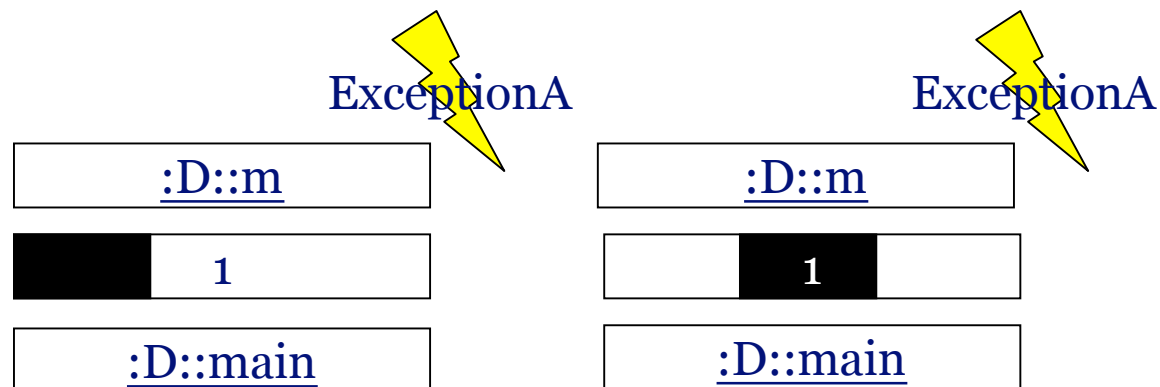
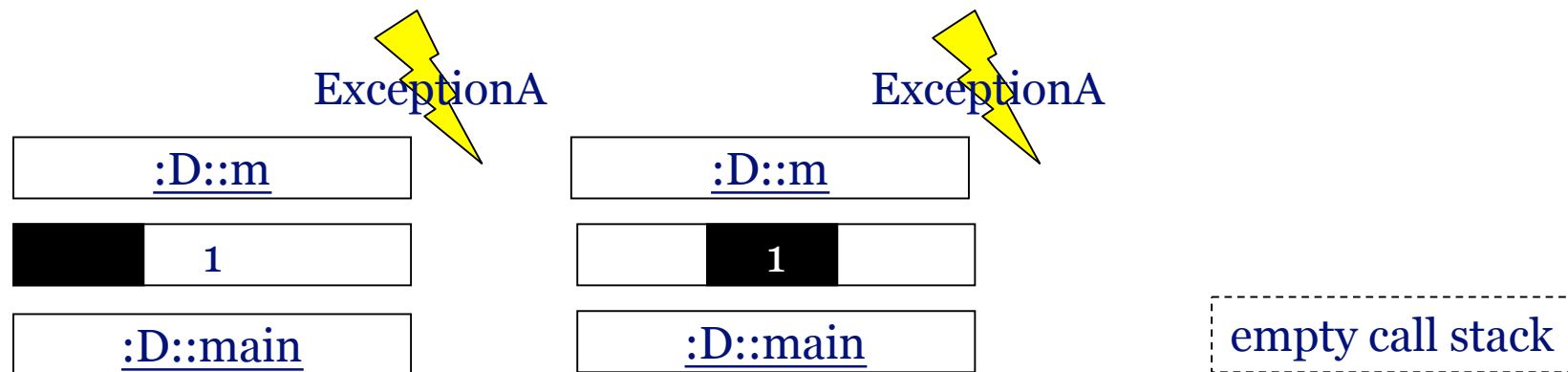


Diagram of exceptions

```
class D {  
    void m() { throw new ExceptionA(1); }  
    public static void main(String[] args) {  
        D d = new D();  
        try { d.m(); }  
        catch(ExceptionA ea) {  
            d.m();  
        }  
    }  
}
```



Other exception handling mechanisms

Return an error code:

List and String have a method called `indexOf(element)` which returns `-1` if the element is not in the collection/String.

Error status

The class `MediaTracker` maintains a list of images and voice objects to be loaded, and one can check an error code to see if anything went wrong.

Differences

- No propagation down the call stack
- No typed information (typically an integer with a certain value is used rather than a type)

Providing context information

```
class Stack {
    LinkedList elements = new LinkedList();
    public void push(Object o) {
        elements.add(o);
    }
    public Object pop() throws EmptyException {
        if (elements.size() == 0)
            throw new EmptyException(this);
        return elements.removeLast();
    }
    public class EmptyException
        extends Exception {
        Stack stack;
        EmptyException(Stack stack) {
            this.stack = stack;
        }
    }
}
```

```
Stack s1 = new Stack();
Stack s2 = new Stack();
Stack s3 = new Stack();
try {
    ...
    System.out.println( s1.pop() );
    System.out.println( s2.pop() );
    System.out.println( s3.pop() );
}
catch(Stack.EmptyException ee) {
    System.out.println("offending stack was: "
        + ee.stack );
}
```

An orange L-shaped line that starts at the top right, goes left, then turns down to the bottom left, framing the content area.

Perspectives on exceptions

Inheritance and checked exceptions

A method header specify which checked exceptions it throws.

The compiler checks to see that the body of the method does not throw any other exceptions

But what about inheritance, can we add or remove exceptions in the throw list?

```
class A {  
    public void foo() throws X {...}  
}  
class B extends A {  
    public void foo() throws X,Y {...} //legal??  
}  
class C extends A {  
    public void foo() {...} // legal??  
}
```

```
public void useA(A a) {  
    try {  
        a.foo();  
    }  
    catch(X x) {  
        ... handle x...  
    }  
}
```

The compiler will accept this code, because all checked exceptions thrown by foo are caught.

But if we call the useA method with an instance of a B, then a Y exception might be thrown, but not caught.

- *We are not allowed to add new exceptions when overriding a method.*
- *We are allowed to remove exceptions*

Inheritance: require less, promise more

Contracts

A contract is an agreement between the client C of a method M, and a provider P of that method M.

Can another provider R substitute P?

Slogan: *If R requires less and promises more, C will not be offended.*

E.g. If the post office ask for less postage and/or delivers the letters faster, we are happy.

If foo (from class A) has precondition P_A , then foo in B must have a weaker precondition P_B . (require less).

If foo in A has post condition Q_A , then foo in B can have a stronger post condition Q_B (promise more).

Exceptions

If foo in A can go wrong in cases X,Y,Z, then foo in B promise to fail only in X and Z (promise more – fewer exceptions to the contract). I will not be offended if the post office delivers also things which are too big.

Inheritance and catch blocks order

- Catch parts are searched from the top and down.
- When a match is successful, the code for that catch block is executed.
- The rest of the catch blocks are ignored.
- If present the finally block is executed.

```
class ExceptionA extends Exception
class ExceptionB extends Exception
class ExceptionC extends ExceptionA
...
void bar() throws ExceptionA {...}
```

- Can bar() throw exceptions of type ExceptionC ?
- Which of these foo() methods are correct?

```
public void foo1() {
    try { bar(); }
    catch(ExceptionA) { }
    catch(ExceptionB) { }
    catch(ExceptionC) { }
}

public void foo2() {
    try { bar(); }
    catch(ExceptionA) { }
    catch(ExceptionC) { }
    catch(ExceptionB) { }
}

public void foo3() {
    try { bar(); }
    catch(ExceptionB) { }
    catch(ExceptionC) { }
    catch(ExceptionA) { }
}
```

Where can exceptions be thrown

- In a method
 - terminates the execution of the method along with all other methods on the call stack from the top till a matching catch is found.
- In a constructor
 - terminates the construction of an object, and terminates all other methods on the call stack from the top till a matching catch is found
- In a static initializer
 - terminates the initialization of the class. However, it is still possible to create instances of that class!! Catching such an exception (outside the initializer) is extremely problematic, as class loading happens asynchronously and is not fully specified.
 - only unchecked exceptions may be thrown, why?
- In a catch or finally block
 - Terminates the rest of the execution of the code in the block and all other methods on the call stack from the top till a matching catch is found.
- Recall that finally blocks are always executed despite the catch block did not match the exception thrown.

Where to declare the Exception class

If the Exception is bound to a specific class make it a member class (inner class).

If the exception can occur in a range of different classes in your package, make it a public class in your package.

If one do not need the 'this' reference in instances of the Exceptions, make the exception class static.

Naming: If the class is name Foo, the exception need not be named Foo as well. E.g. The EmptyException was not called StackEmptyException.

Reason: In use situations, one will write
`catch(Stack.EmptyException ee)`

Exception handling

```
import java.io.*;
class ReaderWriter {
    public static void main(String[] args) throws IOException {
        Reader r1 = new FileReader("in1.txt");
        Reader r2 = new FileReader("in2.txt");
        Writer w = new FileWriter("out.txt");
        int ch;
        while((ch = r1.read()) != -1) w.write(ch);
        while((ch = r2.read()) != -1) w.write(ch);
        r1.close(); r2.close();
        w.close();
    }
}
```

- **Program description**

Merge two files into one destination file.

- if the first file is fully read and written but the second file fails, then keep the destination.
 - if the first file fails, remove the destination file and ignore processing the second.
- Our first try is an easy solution. The customer, however, is presented with low level error messages in the form of exceptions. not good!
 - and in case of a failure, we do not check if we should remove the destination file.

Exception handling

```
import java.io.*;
class ReaderWriter {
    public static void main(String[] args) {
        Reader r1 = null, r2 = null; // must be set to null!
        try {
            r1 = new FileReader("in1.txt");
            r2 = new FileReader("in2.txt");
            Writer w = new FileWriter("out.txt");
            int ch;
            while((ch = r1.read()) != -1) w.write(ch);
            while((ch = r2.read()) != -1) w.write(ch);
            w.close();
        }
        catch(FileNotFoundException e) {+ report to the user
            if(e.getSource() == r1) {
                r2.close(); new File("out.txt").delete(); }
            else r1.close();
        }
    }
}
```

- At the place of catching an exception, one cannot get hold of context information from where the exception is thrown.
- Most exceptions in java does not propagate information with them.
- `getSource()` is a **non-existing method**. We can only get information as strings from the exception.
 - We cannot write this code!
- We can check in the catch block that
 - `r1` and `r2 == null`
 - `r1 != null, r2 == null`
 - But then we make a high coupling between the order of execution and the error handling (ie. they are not separated at all).
 - Was not possible if we caught the exception elsewhere but within the method!

Exception handling

```
... while((ch = r1.read()) != -1) w.write(ch);
```

- `read()` can throw an `IOException`
 - Somehow the file cannot be read
 - access rights
 - EOF is reached
- But handling all cases of IO problems is really difficult – just look at how many direct subclasses there exist for `IOException`:
Direct subclasses of `IOException`: `CharsetException`, `CharacterCodingException`, `CharConversionException`, `ClosedChannelException`, `EOFException`, `FileLockInterruptedException`, `FileNotFoundException`, `HttpRetryException`, `IOException`, `InterruptedIOException`, `InvalidPropertiesFormatException`, `JMXProviderException`, `JMXServerErrorException`, `MalformedURLException`, `ObjectStreamException`, `ProtocolException`, `RemoteException`, `SaslException`, `SocketException`, `SSLException`, `SyncFailedException`, `UnknownHostException`, `UnknownServiceException`, `UnsupportedEncodingException`, `UTFDataFormatException`, `ZipException`
 - And for each exception one should understand the exception and determine (how is not always clear!) if it applies to the application in the domain it is used, and if we thus need a special error handling and supply specific information to the user.
- Likewise for `write()` ...
- For both cases, we must remember to close all three streams!

Exception handling

```
import java.io.*;
class ReaderWriter {
    public static void main(String[] args) {
        try {
            Reader r1 = null, r2 = null; Writer w = null;
            try {
                r1 = new FileReader("in1.txt");
                try {
                    r2 = new FileReader("in2.txt");
                    try {
                        w = new FileWriter("out.txt"); int ch;
                        try { while((ch = r1.read()) != -1) w.write(ch); }
                        catch(IOException e) { new File("out.txt").delete(); /* report deleting output */ }
                        try { while((ch = r2.read()) != -1) w.write(ch); }
                        catch(IOException e) { /* report keeping output */ }
                    }
                    catch(IOException e) { }
                    finally { w.close(); }
                }
                catch(FileNotFoundException e) { }
                finally { r2.close(); }
            }
            catch(FileNotFoundException e) { }
            finally { r1.close(); }
        }
        catch(IOException e) { /* if any close() fail... still we do not know which one! */ }
    }
}
```

- Reporting to the user may be separated, but low level clean up may be difficult to let external objects take care of.

Exception handling

```
import java.io.*;
class ReaderWriter {
    public static void main(String[] args) {
        try {
            Reader r1 = null, r2 = null; Writer w = null;
            try {
                r1 = new FileReader("in1.txt");
                try {
                    r2 = new FileReader("in2.txt");
                    try {
                        w = new FileWriter("out.txt"); int ch;
                        try { while((ch = r1.read()) != -1) w.write(ch); }
                        catch(IOException e) { new File("out.txt").delete(); /* report deleting output */ }
                        try { while((ch = r2.read()) != -1) w.write(ch); }
                        catch(IOException e) { /* report keeping output */ }
                    }
                    catch(IOException e) { /* report failure */ }
                    finally { w.close(); }
                }
                catch(FileNotFoundException e) { /* report failure */ }
                finally { r2.close(); }
            }
            catch(FileNotFoundException e) { /* report failure */ }
            finally { r1.close(); }
        }
        catch(IOException e) { /* if any close() fail... still we do not know which one! */ }
    }
}
```

- Proper clean up may easily entail almost as bad code as the code from the motivation showing intermixed business code and error handling code!
- Almost more difficult to read code in this style of programming! (but still less redundant than the if-else approach)
- green – business logic
- purple – code which may execute on error
- red – code which execute on error

An orange L-shaped line that starts at the top right, goes left, then turns down and goes to the bottom left.

Exceptions and efficiency

Exceptions and efficiency

In class System there is a method called `currentTimeMillis()`, which returns number of milliseconds since midnight January 1st 1970.

It can be used as a stopwatch to find out how long time it takes to execute a piece of java code:

```
int N = 100000000; // 100.000.000
long start = System.currentTimeMillis();
long time;
for (int i=0; i<N; i++)
    foo(7); // this is what we examine.
time = System.currentTimeMillis()-start;
System.out.println("Calling an empty method"
    + " takes " + (time*1000)/N + " μ seconds");
```

1. Without Exceptions : 0.00401 μ sec
2. With Exceptions : 4.797 μ sec
3. No ex, but finally : 0.0043 μ sec
using java 1.4

1. Without Exceptions : 0.004 μ sec
2. With Exceptions : 2.780 μ sec
3. No ex, but finally : 0.004 μ sec
using java 5

Notice, it is about **700-1000** times slower if the method throws an exception.

The for loop runs 100.000.000 times in the code to the left. That takes a few seconds on my laptop.

Beware: 1000 times a 2 seconds is about half an hour. Run the loop with a small number at first, and then increase by factors of 10.

Code for the timing experiment

1) No exception thrown

```
void pip1(int N) {  
    for (int i = 0; i<N; i++) {  
        try {  
            noUps();  
        }  
        catch(Exception e) { }  
    }  
}
```

2) An exception thrown

```
void pip2(int N) {  
    for (int i = 0; i<N; i++) {  
        try{  
            uups();  
        } catch(Exception e) { }  
    }  
}
```

3) No exception thrown, but with a finally

```
void pip3(int N) {  
    for (int i = 0; i<N; i++) {  
        try {  
            noUps();  
        }  
        catch(Exception e) { }  
        finally {}  
    }  
}
```

```
void uups() throws Exception {  
    throw new Exception("Oh no");  
}
```

```
void noUps() throws Exception { }
```


What takes time with exceptions

The uups method makes a new Exception object, and throws it.

It is making the Exception object which is expensive:

```
void uups() throws Exception {  
    throw new Exception("Oh no");  
}  
void noUups() throws Exception {  
    new Exception("Oh no");  
}
```

Now the timing results are:

Without Exceptions	: 2.643 micro sec
With Exceptions	: 2.764 micro sec
No ex, but finally	: 2.594 micro sec

That is a difference of ~7%

An alternative is to make the exception object ahead of time:

```
static Exception ohNo = new Exception("Oh no no");  
void uups1() throws Exception {  
    throw ohNo;  
}  
void uups2() throw Exception {  
    ohNo.fillStackTrace();  
    throw onNo;  
}
```

Then the timing results become:

With Exceptions1	: 0.1533 micro sec
With Exceptions2	: 0.2424 micro sec

This way exceptions are "only" 40-60 times slower, not 700 times.

Why is it so expensive to make Exception objects?

Exception is a subclass of Throwable.

A throwable object has a stacktrace.

The stack trace tells in which method the Throwable was created, and from which method it was called, and from where it was called...

Establishing the stack trace is done when a new Throwable is created (or any of its subclasses).

Setting the stack trace accounts for the remaining cost of throwing an exception.

The deeper the call stack is when you make the exception object, the more expensive it is to make the exception object.

Runtime reflection: StackTraceElement

The stack trace is an array of StackTraceElement, with the most recent method call being in the zero'th index.

The following method returns true if the method it was called from is named foo, false otherwise.

```
public static boolean amIFoo() {  
    Throwable t = new Throwable();  
    StackTraceElement[] trace = t.getStackTrace();  
    String methodName = trace[1].getMethodName();  
    return methodName.equals("foo");  
}  
  
public static void main(String[] args) {  
    System.out.println("Main:" + amIFoo() );  
    foo(); }  
  
public static void foo() {  
    System.out.println("Foo: " + amIFoo() );}
```

The API in brief:

String getClassName()

String getFileName()

int getLineNumber()

String getMethodName()

Notice, the methods does not return the reflection objects, but string names.

Only in recent version of Java is this possible (Java 1.4 and newer)