Java Exceptions

SoftEng
http://softeng.polito.it

Version June 2009

Motivation

- Report errors, by delegating error handling to higher levels
- Callee might not know how to recover from an error
- Caller of a method can handle error in a more appropriate way than the callee

- Localize error handling code, by separating it from functional code
- Functional code is more readable
- Error code is centralized, rather than being scattered

The world without exceptions (I)

- If a non locally remediable error happens while method is executing, call `System.exit()`

- A method causing an unconditional program interruption in not very dependable (nor usable)

The world without exceptions (II)

- If errors happen while method is executing, we return a special value
- Special values are different from normal return value (e.g., null, -1, etc.)

- Developer must remember value/meaning of special values for each call to check for errors
- What if all values are normal?
  - double pow(base, exponent)
  - pow(-1, 0.5); //not a real

Real problems

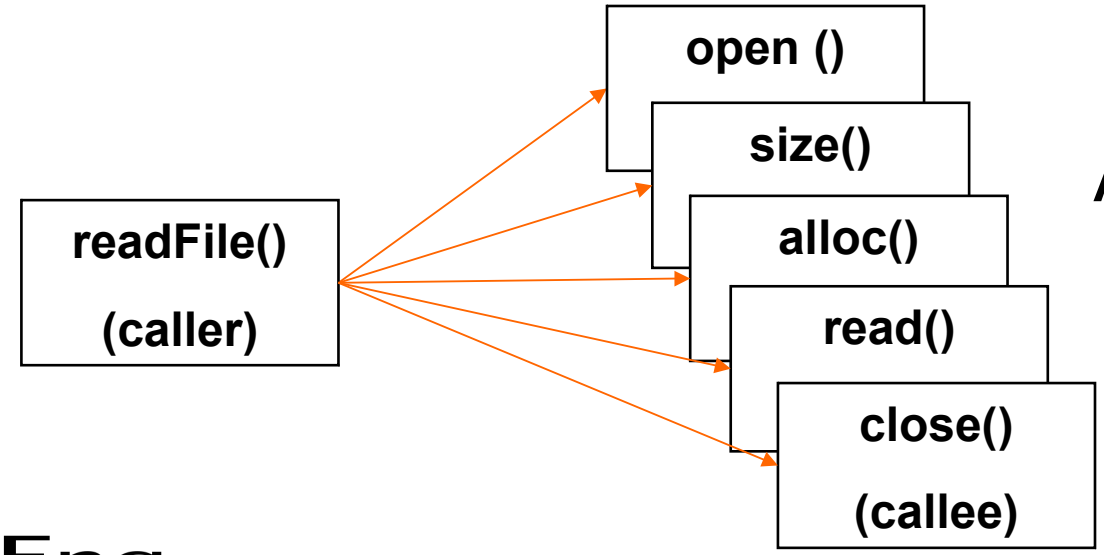
- Code is messier to write and harder to read

```
if( somefunc() == ERROR ) // detect error
    //handle the error
else
    //proceed normally
```

- Only the direct caller can intercept errors (no delegation to any upward method)

Example – Read file

- open the file
- determine file size
- allocate that much memory
- read the file into memory
- close the file



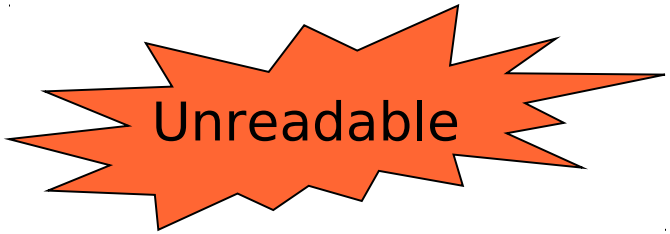
All of them can fail

Correct (but boring)

```
int readFile {
   open the file;
   if (operationFailed)
      return -1;
   determine file size;
   if (operationFailed)
      return -2;
   allocate that much memory;
   if (operationFailed) {
      close the file;
      return -3;
   }
   read the file into memory;
   if (operationFailed) {
      close the file;
      return -4;
   }
   close the file;
   if (operationFailed)
      return -5;
   return 0;
}
```

Lots of error-detection and error-handling code

To detect errors we must check specs of library calls (no homogeneity)



Wrong (but quick and readable)

```
int readFile {

    open the file;
    determine file size;
    allocate that much memory;
    read the file into memory;
    close the file;

    return 0;
}
```

Which one would YOU use ?

☺

Using exceptions (nice)

```
try {
        open the file;
        determine file size;
        allocate that much memory;
        read the file into memory;
        close the file;
} catch (fileOpenFailed) {
        doSomething;
} catch (sizeDeterminationFailed) {
        doSomething;
} catch (memoryAllocationFailed) {
        doSomething;
} catch (readFailed) {
        doSomething;
} catch (fileCloseFailed) {
        doSomething;
}
```

Basic concepts

1. The code causing the error will generate an exception
   - Developers code
   - Third-party library
1. At some point up in the hierarchy of method invocations, a caller will intercept and stop the exception
2. In between, methods can
   - Ignore the exception (complete delegation)
   - Intercept without stopping (partial delegation)

Syntax

- Java provides three keywords
  - Try
    - Contains code that may generate exceptions
  - Catch
    - Defines the error handler
  - Throw
    - Generates an exception

- We also need a new entity
  - Exception class

Generation

1. Declare an exception class
2. Mark the method generating the exception
3. Create an exception object
4. Throw upward the exception

Generation

```
// java.lang.Exception                                    (1)
public class EmptyStack extends Exception {
}

class Stack{                                              (2)
   public Object pop() throws EmptyStack {

      if(size == 0) {
         Exception e = new EmptyStack();                  (3)
         throw e;                                         (4)
      }
      ...
   }
}
```

throws

- Method interface must declare exception type(s) generated within its implementation (list with commas)

- Either generated and thrown by method, directly
- Or generated by other methods called within the method and not caught

throw

- Execution of current method is interrupted immediatelly
- Catching phase starts

Interception

- Catching exceptions generated in a code portion

```
try {
   // in this piece of code some
   // exceptions may be generated
   stack.pop();
   ...
}
catch (StackEmpty e) {

   // error handling
   System.out.println(e);
   ...
}
```

Execution flow

- `open` and `close` can generate a `FileError`
- Suppose `read` does not generate exceptions

```
System.out.print("Begin");

File f = new File("foo.txt");
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print("Error");
}

System.out.print("End");
```

Execution flow

- No exception generated

```
System.out.print("Begin");

File f = new File(“foo.txt”);
try{
   f.open();
   f.read();
   f.close();
}catch(FileError fe){
   System.out.print("Error");
}

System.out.print("End");
```

Execution flow

- `open()` generates an exception
- `read()` and `close()` are skipped

```
System.out.print("Begin");

File f = new File("foo.txt");
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print("Error");
}

System.out.print("End");
```

Multiple catch

- Capturing different types of exception is possible with different catch blocks

```
try {
  ...
}
catch(StackEmpty se) {
   // here stack errors are handled
}
catch(IOException ioe) {
   // here all other IO problems are handled
}
```

Execution flow

- `open` and `close` can generate a `FileError`
- `read` can generate a `IOError`

```
System.out.print("Begin");

File f = new File("foo.txt");
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print("File err");
}catch(IOError ioe){
  System.out.print("I/O err");
}

System.out.print("End");
```

Execution flow

- `close` fails
- “File error” is printed

```
System.out.print("Begin");

File f = new File(“foo.txt”);
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print(“File err”);
}catch(IOError ioe){
  System.out.print(“I/O err”);
}

System.out.print("End");
```

Execution flow

- `read` fails
- "I/O error" is printed

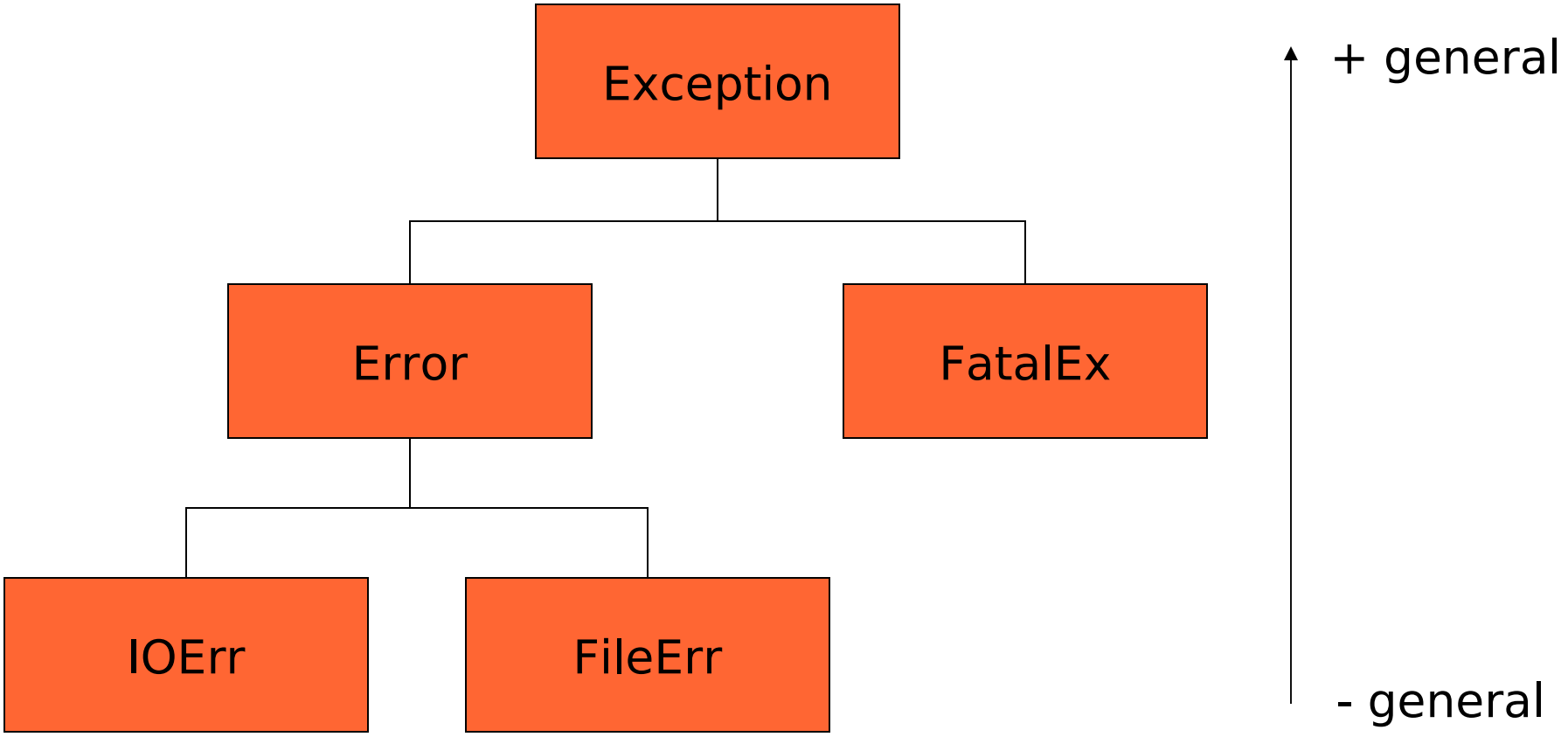
```
System.out.print("Begin");

File f = new File("foo.txt");
try{
  f.open();
  f.read();
  f.close();
}catch(FileError fe){
  System.out.print("File err");
}catch(IOError ioe){
  System.out.print("I/O err");
}

System.out.print("End");
```

Matching rules

- Only one handler is executed
- The more specific handler is selected, according to the exception type

- Handlers must be ordered according to their “generality”

Matching rules



Matching rules

```
class Error   extends Exception{}
class IOErr   extends Error{}
class FileErr extends Error{}
class FatalEx extends Exception{}
```

```
try{ /*…*/ }
catch(IOErr ioe){ /*…*/ }
catch(Error er){ /*…*/ }
catch(Exception ex){ /*…*/ }
```

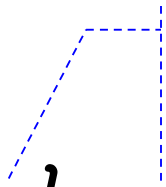
- general

+ general

Matching rules

```
class Error    extends Exception{}
class IOErr    extends Error{}
class FileErr  extends Error{}
class FatalEx  extends Exception{}
```

```
try{ /*…*/ }
catch(IOErr ioe){ /*…*/ }
catch(Error er){ /*…*/ }
catch(Exception ex){ /*…*/ }
```



IOErr is generated

Matching rules

```
class Error    extends Exception{}
class IOErr    extends Error{}
class FileErr  extends Error{}
class FatalEx  extends Exception{}
```

```
try{ /*…*/ }
catch(IOErr ioe){ /*…*/ }
catch(Error er){ /*…*/ }
catch(Exception ex){ /*…*/ }
```

Matching rules

```
class Error    extends Exception{}
class IOErr    extends Error{}
class FileErr  extends Error{}
class FatalEx  extends Exception{}

try{ /*…*/ }
catch(IOErr ioe){ /*…*/ }
catch(Error er){ /*…*/ }
catch(Exception ex){ /*…*/ }
```

FatalEx is generated

Nesting

- Try/catch blocks can be nested
- E.g. error handler may generate new exceptions

```
try{ /* Do something */ }
catch(…){
   try      { /* Log on file */ }
   catch(…){ /* Ignore */      }
}
```

Generate and catch

- When calling code, which possibly raises an exception, the caller can

1. Catch
2. Propagate
3. Catch and re-throw

[1] Catch

```
class Dummy {
  public void foo(){
    try{
      FileReader f;
      f = new FileReader("file.txt");
    } catch (FileNotFound fnf) {
       // do something
    }
  }
}
```

[2] Propagate

```
class Dummy {

  public void foo() throws FileNotFound{
    FileReader f;
    f = new FileReader("file.txt");
  }

}
```

[2] Propagate (cont'd)

- Exception not caught can be propagated till main() and VM

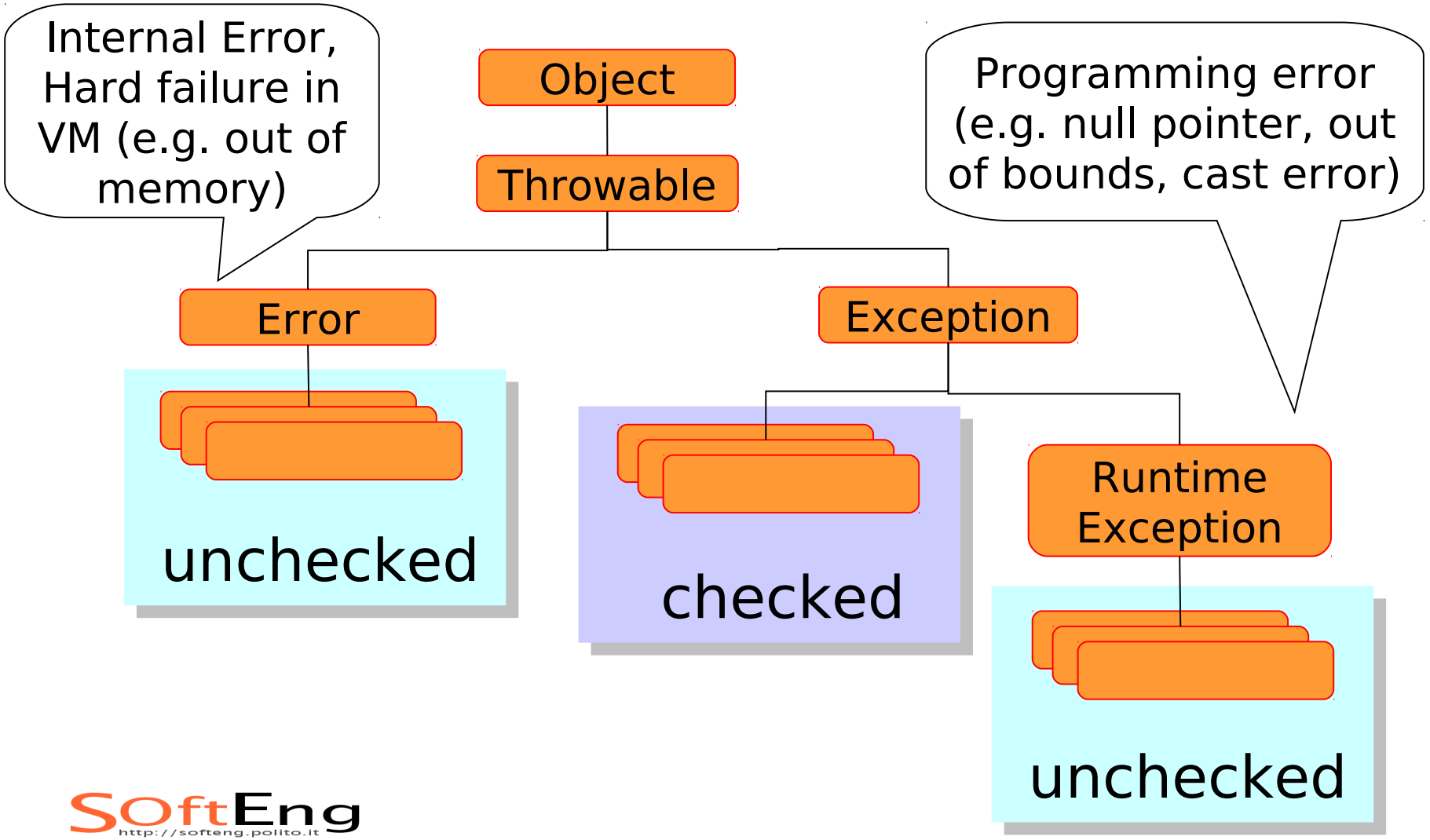
```
class Dummy {
   public void foo() throws FileNotFound {
      FileReader f = new FileReader("file.txt");
   }
}
```

```
class Program {
   public static
   void main(String args[]) throws FileNotFound {
      Dummy d = new Dummy();
      d.foo();
}
```

[3] Re-throw

```
class Dummy {
  public void foo(){
    try{
      FileReader f;
      f = new FileReader("file.txt");
    } catch (FileNotFound fnf) {
       // handle fnf, e.g., print it
       throw fnf;
    }
  }
}
```

Exceptions hierarchy



Custom exceptions

- It is possible to define new types of exceptions
- If the ones provided by the system are not enough...
- Just sub-classing Throwable or one of its descendants

Checked and unchecked

- Unchecked exceptions
  - Their generation is not foreseen (can happen everywhere)
  - Need not to be declared (not checked by the compiler)
  - Generated by JVM
- Checked exceptions
  - Exceptions declared and checked
  - Generated with “throw”

finally

- The keyword finally allows specifying actions that must be always executed
  - Dispose resources
  - Close a file

After all catch branches (if any)

```
MyFile f = new MyFile();
if (f.open("myfile.txt")) {
    try {
        exceptionalMethod();
    } finally {
        f.close();
    }
}
```

Exceptions and loops (I)

- For errors affecting a single iteration, the try-catch blocks is nested in the loop.
- In case of exception the execution goes to the catch block and then proceed with the next iteration.

```
while(true){
   try{
     // potential exceptions
   }catch(AnException e){
     // handle the anomaly
   }
}
```

Exceptions and loops (II)

- For serious errors compromising the whole loop the loop is nested within the try block.
- In case of exception the execution goes to the catch block, thus exiting the loop.

```
try{
    while(true){
    // potential exceptions
    }
}catch(AnException e){
    // print error message
}
```