Java Exceptions

Università di Modena e Reggio Emilia
Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

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

- Separates error handling from functional code
- Functional code is more readable
- Error code is centralized, rather than being scattered

The world without exceptions

- 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

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

- 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-world problems

- Code is messier to write and harder to read
- Only the direct caller can intercept errors (no delegation to any upward method)

```
if ( func() == ERROR)
  // handle error
else
  // proceed
```

An example, file to memory copy

- Open the file open()
- Determine file size size()
- Allocate that much memory allocate()
- Read the file into memory read()
- Close the file close()

All of them can fail!

Correct (but long and obscure)

```
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 short and readable)

```
int  readFile  {
   open the file;
   determine file size;
  allocate that much memory;
   read the file into memory;
   close the file;
return  0;
}
```

Using Exceptions

```
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

- The code causing the error will generate an exception
  - Developers code
  - Third-party library
- At some point up in the hierarchy of method invocations, a caller will intercept and stop the exception
- 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

- Declare an exception class
- Mark the method generating the exception
- Create an exception object
- Throw upward the exception

Generation

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

public class Stack {
  public Object pop() throws EmptyStack {
    if (size == 0) {
       throw( new EmptyStack(); )
    }
    …
  }
}
```

throws

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

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

throw

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

Interception

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

Interception

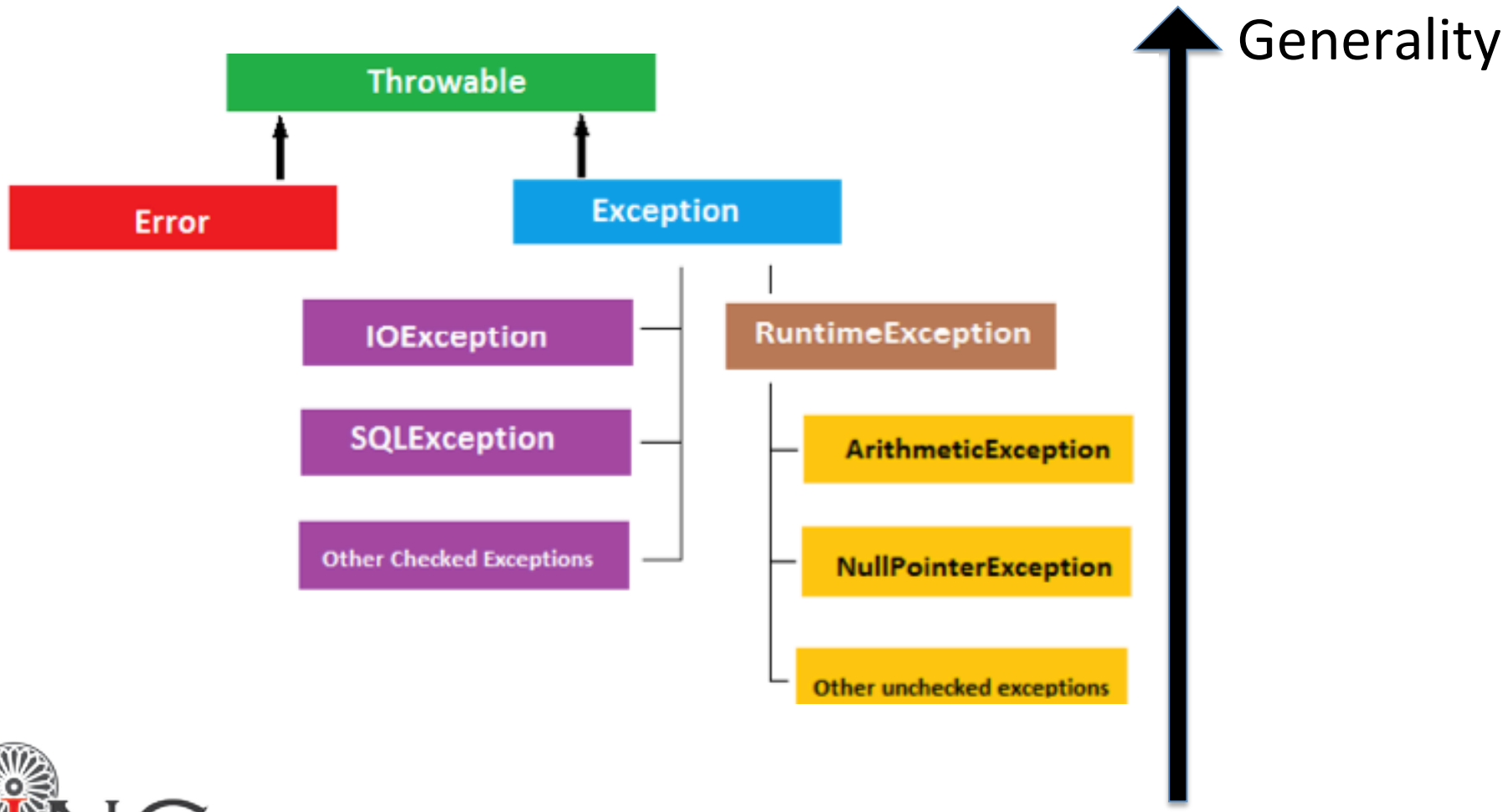
```
try  {
   ...
}
catch(StackEmpty  se)  {
   // here stack  errors  are  handled
}
catch(IOException ioe)
   // here all  other  IO problems are
handles
}
```

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

java.lang.Object

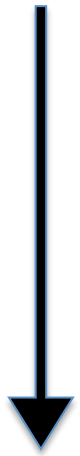
java.lang.Throwable

java.lang.Exception

java.io.IOException

java.io.EOFException

```
try   { /*…*/ }
Catch(EOFException e01)   { /*…*/}
catch(IOException e02) { /*…*/}
catch(Exception e03) { /*…*/}
```



Generality

ING

A complete example

```
File f = new File("foo.txt");
try {
    f.open();
    f.read();
    f.close();
} catch (IOException e) {
    System.out.println("something went wrong!");
}
```

Nesting

- Try/catch blocks can be nested (e.g., error handlers 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
  - Catch
  - Propagate
  - Catch and re-throw

Catch

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

Propagate

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

Propagate

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

```
Class Dummy {
     public void foo() throws FileNotFound {
          FileReader f;
          f = new FileReader("file.txt");
     }
}
Class App {
     public static void main (String args[]) throws FileNotFound  {
          Dummy d = new Dummy();
          f.foo();
     }
}
```

Catch and re-throw

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

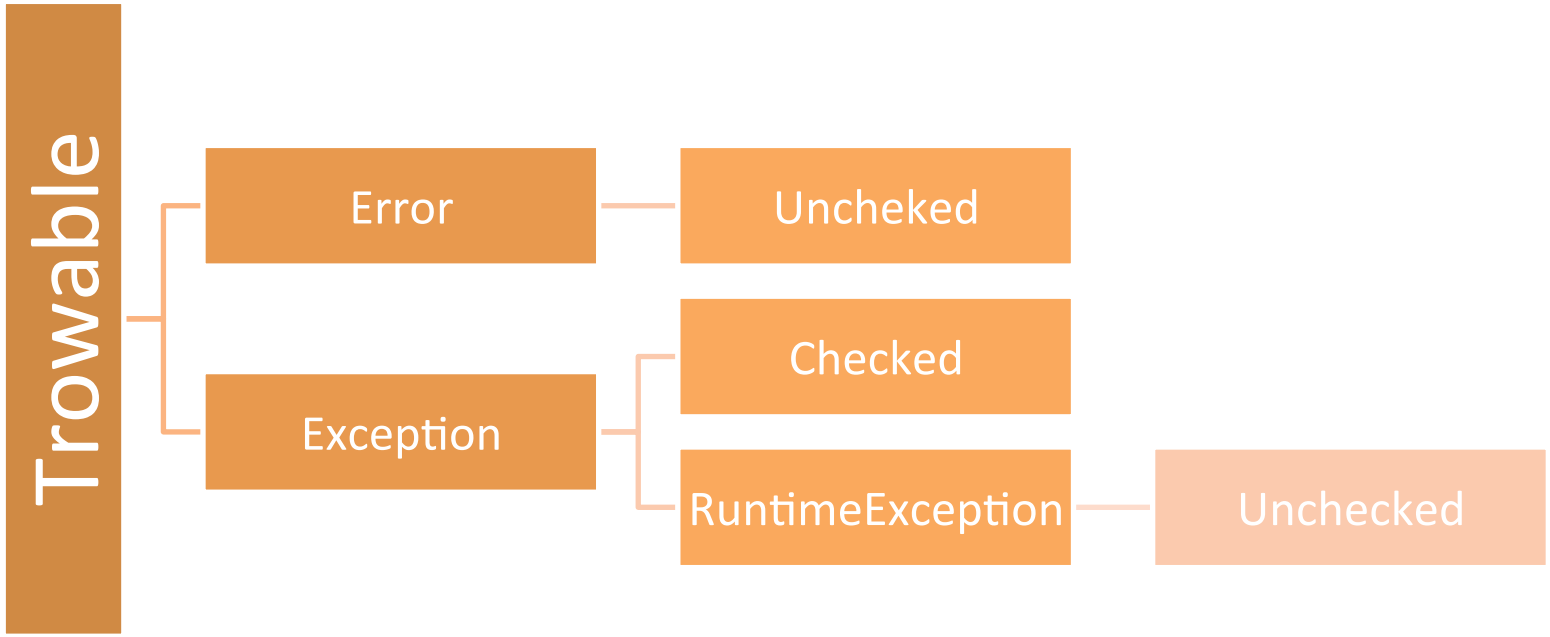
Custom Exception

- 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”

Checked and Unchecked



ING

Exceptions and loops

- 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

- 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
}
```

Finally

The runtime system always executes the statements within the ***finally*** block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

```
File f = new File("foo.txt");
try {
      f.open();
      f.read();
      f.close();
} catch (IOException e) {
      System.out.println("something went wrong!");
} finally {
      if (out != null) {
            System.out.println("Closing file!");
            out.close();
      } else {
            System.out.println("File not open!");
      }
}
```