Java Exceptions

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

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?

The world without exceptions

```
List<Integer> list = new ArrayList<Integer>();
public int get(int i) {
  if (list.size() <= i) {
    return -1;
  return list.get(i)
}
```

What does -1 mean?

Is it an error or a negative value from the list?

Need to find documentation!

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 is not usable in real-world!

The world without exceptions

```
List<Integer> list = new ArrayList<Integer>();
public int get(int i) {
  if (list.size() <= i) {
    System.exit();        // Never do this!!
  return list.get(i)
}
```

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)

```
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; }
```

Using Exceptions

- Exceptions delegate 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

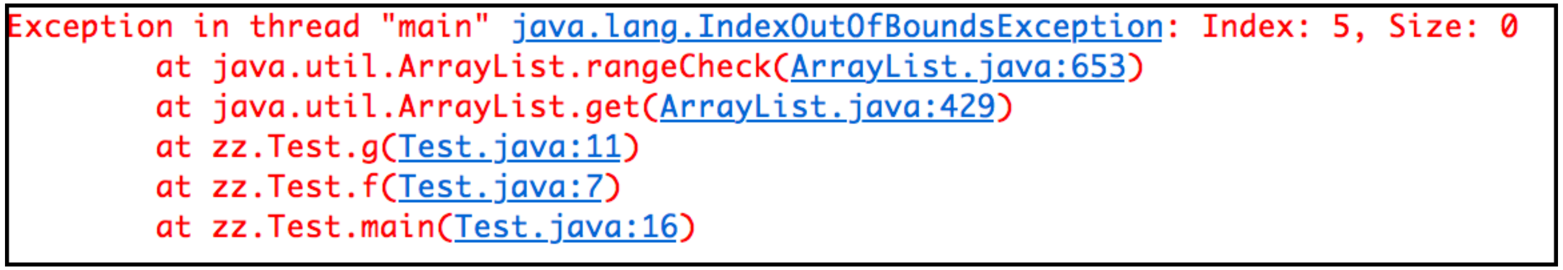
- Exceptions separate error handling from functional code
  - Functional code is more readable
  - Error code is centralized, rather than being scattered

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)

Stack trace

```
public class Test {
    public void f(int i) {
        g(i);
    }
    public void g(int i) {
        new ArrayList().get(i);
    }
    public static void main(String[] args) {
        new Test().f(5);
    }
}
```



```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 5, Size: 0
	at java.util.ArrayList.rangeCheck(ArrayList.java:653)
	at java.util.ArrayList.get(ArrayList.java:429)
	at zz.Test.g(Test.java:11)
	at zz.Test.f(Test.java:7)
	at zz.Test.main(Test.java:16)
```

Syntax

- Java provides three keywords
  - Try
    - Contains code that may generate exceptions
  - Catch
    - Defines the error handler
  - Throw
    - Generates an exception
  - Throws
    - Mark a method as able to convey exceptions

- We also need a new entity
  - Exception class

Interception

```
try   {
   AudioSystem.getAudioInputStream(
      new FileInputStream(“music.wav”));
} catch  (IOException e)   {
   // error handling
   System.out.println(e);
   ...
}
```

Interception

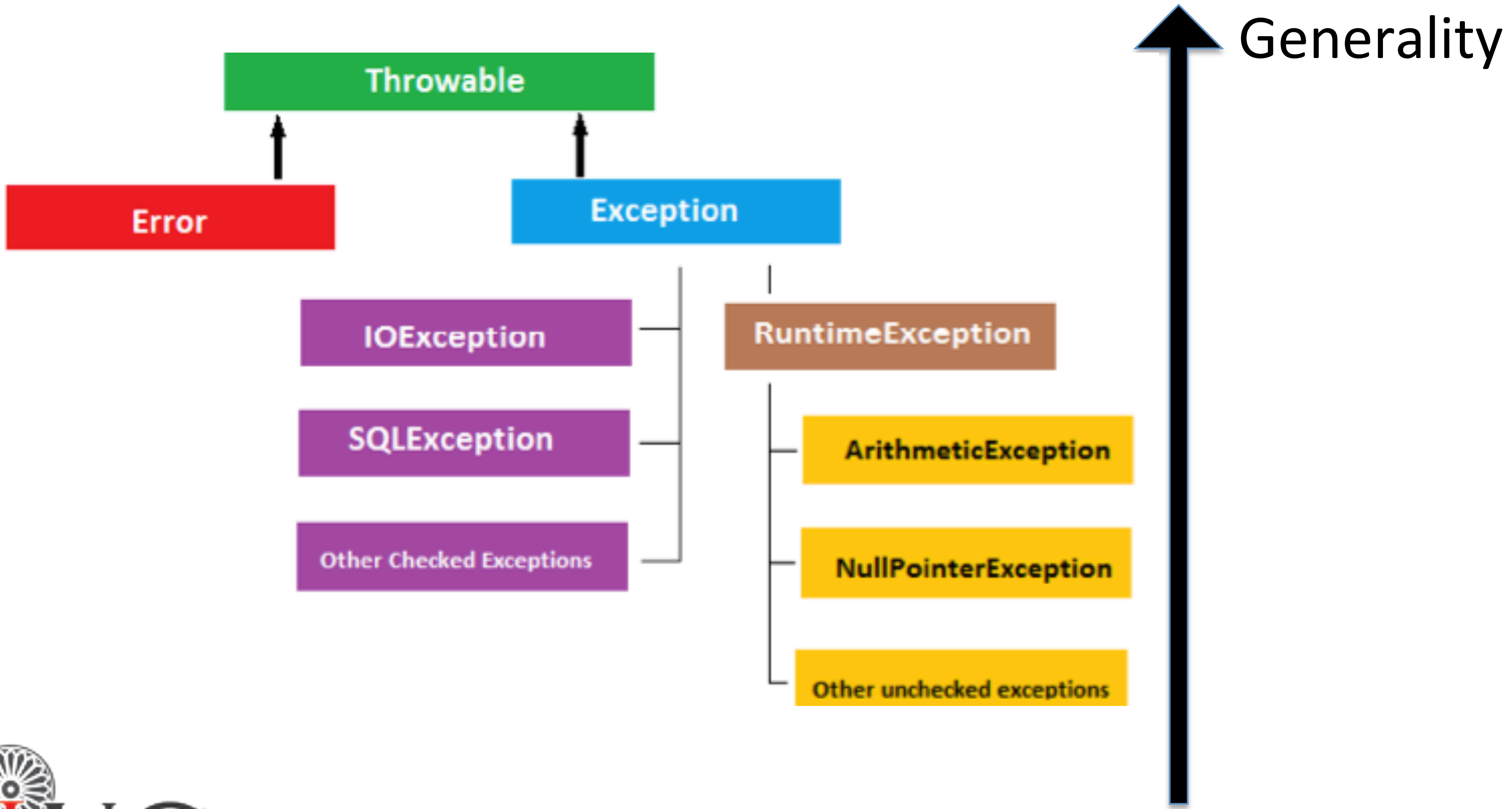
```
try {
    AudioSystem.getAudioInputStream(
        new FileInputStream(“music.wav”));
} catch (IOException e01) {
    //  error  handling
    System.out.println(e);
    ...
} catch(UnsupportedAudioFileException e02) {
    //  error  handling
    System.out.println(e);
    ...
}
```

Interception

```
try{
    f.read();
} catch(EOFException e01) {
    //
} catch(IOException e02) {
    //
} catch(Exception e03) {
    //
}
```

Only one handler is executed! Handlers must be ordered according to their “generality”. More specific first!

Matching Rules



A complete example

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

Generation

- (Eventually) Declare an exception class
- Mark the method generating the exception with throws
- Throw upward a new exception object

Generation

```
public class EmptyStack extends Exception {}

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

throw

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

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

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

ING

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(). When an exception is not caught in main() execution is halted!

```
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();
     }
}
```

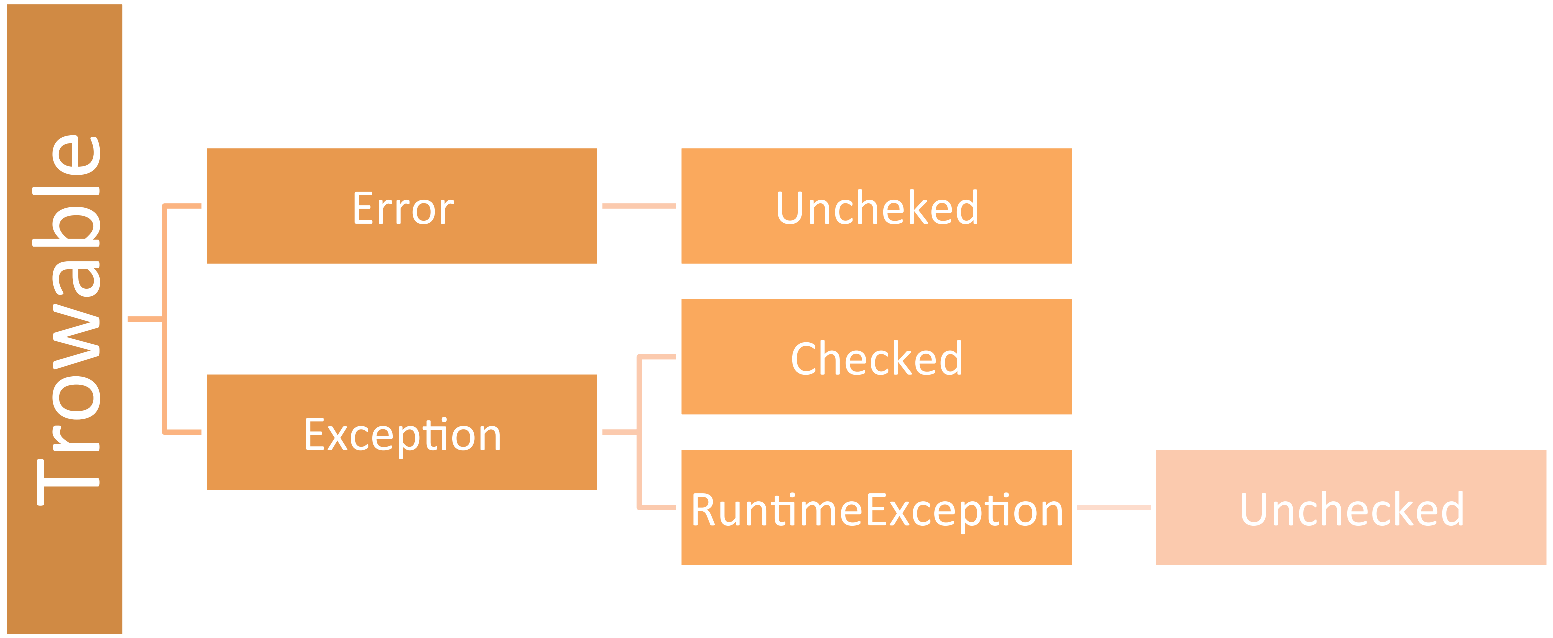
Catch and re-throw

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

Custom Exception

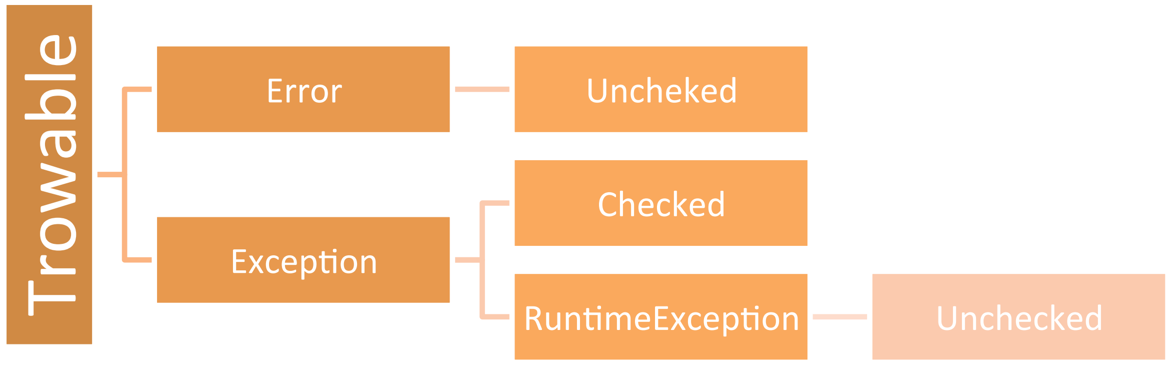
- It is possible to define new types of exceptions if the ones provided by the system are not enough…
- Subclass Throwable or Exception
  - `public class EmptyStack extends Exception {}`

Checked and Unchecked



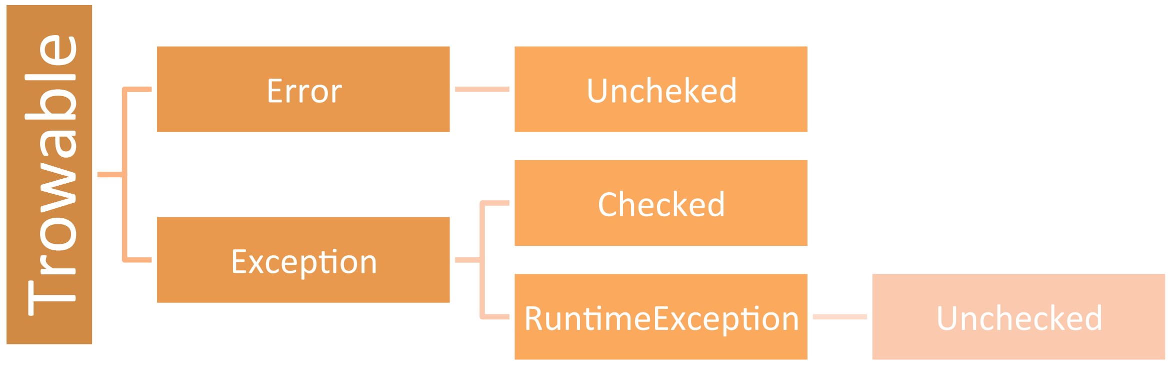
Checked and Unchecked

- Unchecked exceptions (Generated by JVM)
  - Their generation is not foreseen (can happen everywhere)
  - Need not to be declared (not verified by the compiler)
  - NullPointerException, ArrayIndexOutOfBound, …
- Checked exceptions
  - Exceptions declared and checked
  - Generated with “throw”
  - IOException, SQLException, ClassNotFoundException, …



Error

- 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.
  - LinkageError - Subclasses of LinkageError indicate that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
  - VirtualMachineError - Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.



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(Exception 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 ***finally*** block regardless the outcome of try/catch. Usually it is used for cleanup (e.g., closing files, connections, …).

```
FileReader f = new FileReader (“foo.txt”);
try {
    f.open();
    f.read();
    f.close();
} catch (IOException e) {
    System.out.println(“something went wrong!”);
} finally {
    if (out != null) out.close();
}
```