Java Exceptions

SOftEng
http://softeng.polito.it

Motivation

- Report anomalies, by delegating error handling to higher levels
  - Called method might not know how to recover from an error
  - Caller of a method can handle error in a more appropriate way than the method itself
- Localize error handling code, by separating it from functional code
  - Functional code is more readable
  - Error code is collected together, rather than being scattered

Error signaling techniques

- Program termination
  - Abrupt termination of the execution
- Special value
  - Return a special value to indicate error
- Global status
  - A global variable contains error condition
- Exceptions
  - Throw an exception

Error handling: abort

- If a non-locally-remediable error occurs while method is executing, call `System.exit()`
- A method causing an unconditional program interruption in not very dependable (nor usable)

Error handling: special value

- If an error occurs while method is executing, 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

Error handling code

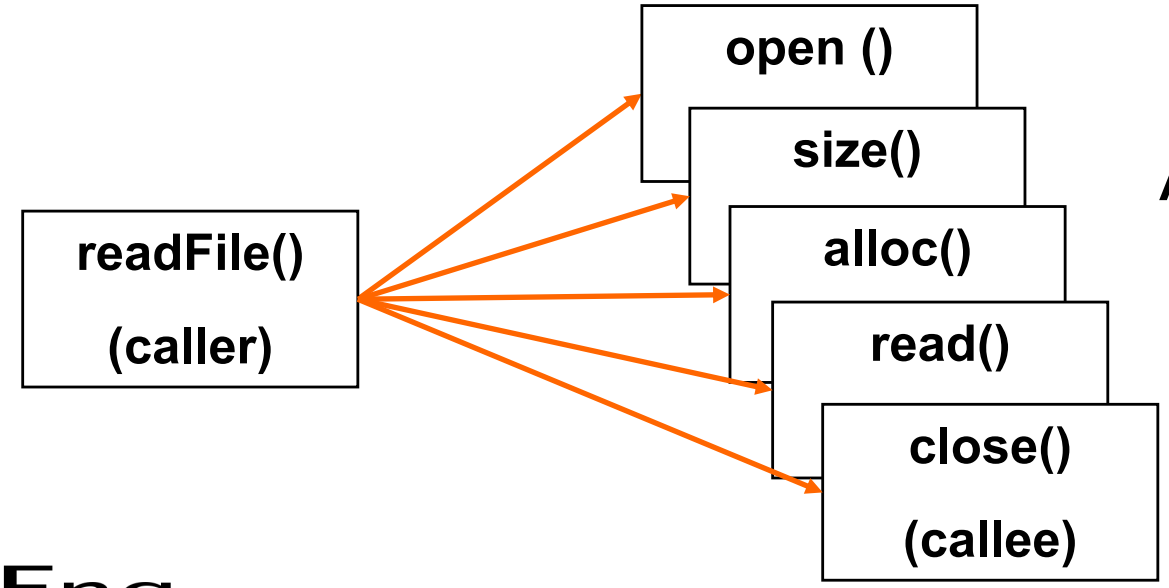
- Code is messy to write and hard to read

```
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

Special values (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 one must check specs of library calls (no homogeneity)

No error handling (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 (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

- The code causing/detecting 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 handle the exception
- In between, methods can
  - Ignore the exception (complete delegation)
  - Intercept and re-issue (partial delegation)

Syntax

- Java provides three keywords
  - **`Throw`**
    - Raises (generate) an exception
  - **`Try`**
    - Introduces code to watch for exceptions
  - **`Catch`**
    - Defines the exception handling code
- Java also defines a new type
  - **`Throwable`** (and **`Exception`**)

Generating Exceptions

1. Identify/Define an exception class
2. Declare/Mark the method as potential source of exception
3. Create an exception object
4. Throw upward the exception

Generation

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

                                                          (2)
class Stack<E>{
   public E pop() throws EmptyStack {

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

throws

- The method interface must declare exception type(s) generated within its body
  - Possibly more than one (list w/ commas)
- Either
  - generated and thrown by the method, directly
  - or generated by other methods called within the method and not caught

throw

- When an exception is thrown:
  - The execution of the current method is interrupted instantly
  - The code immediately following the throw statement is not executed
    - Similarly to a **`return`** statement
  - The 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

- If no exception is generated then the **`catch`** block is 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");
```

Execution flow

- If **`open()`** generates an exception, then **`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 probl. 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

- If `close()` fails, a file error is notified

```
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

- If **`read()`** fails, an I/O error is notified

```
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 block defining error-handling code is executed
- The more specific block is selected, according to the exception type
- Blocks must be ordered according to their “generality”

Exception handling

- When a fragment of code can possibly raise an exception, the exception must be handled / checked
- Handling can use different strategies:
  - Catch
  - Propagate
  - Catch and re-throw

Handling: catch

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

Handling: propagate

```
class Dummy {

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

}
```

Handling: propagate

- Exception not caught can be propagated untill the **`main()`** method and the JVM

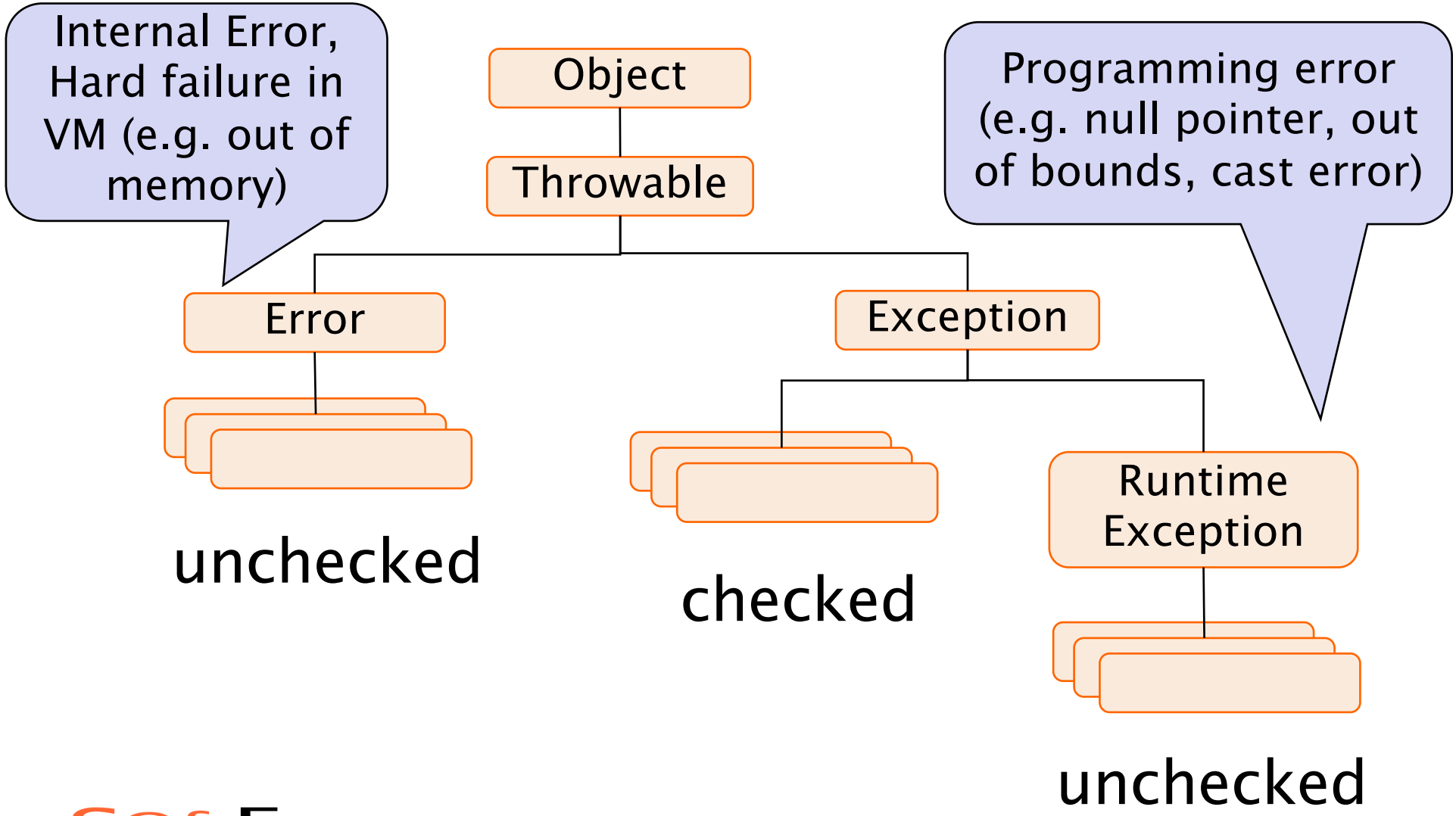
```
class Dummy {
    public void foo() throws FileNotFound {
        FileReader f = new FileReader("file.txt");
    }
}
```

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

Handling: re-throw

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

Exceptions hierarchy



Checked and unchecked

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

Main exception classes

- **Error**
  - OutOfMemoryError
- **Exception**
  - ClassNotFoundException
  - InstantiationException
  - NoSuchMethodException
  - IllegalAccessException
  - NegativeArraySizeException
- **RuntimeException**
  - NullPointerException
  - ClassCastException

Custom exceptions

- It is possible to define new types of exceptions
  - Represent anomalies specific for the application
  - Can be caught separately from the predefined ones
  - Must extend **`Throwable`** or one of its descendants
  - Most commonly they extend **`Exception`**

finally

- Keyword **`finally`** allows specifying actions that must be executed in any case
  - Dispose of 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
}
```

Wrap-up

- Exceptions provide a mechanism to handle anomalies and errors
- Allow separating “nominal case” code from exceptional case code
- Decouple anomaly detection from anomaly handling
- They are used pervasively throughout the standard Java library

Wrap-up

- Exceptions are classes extending the **`Throwable`** base class
- Inheritance is used to classify exceptions
  - **`Error`** represent internal JVM errors
  - **`RuntimeException`** represent programming error detected by JVM
  - **`Exception`** represent the usual application-level error

Wrap-up

- Exception must be checked by
  - Catching them with **`try{ } catch{ }`**
  - Propagating with **`throws`**
  - Catching and re-throwing (propagating)
- Unchecked exceptions can avoid mandatory handling
  - All exceptions extending **`Error`** and **`RuntimeException`**