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
- Developers must remember value/meaning of special values for checking errors

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

The world without exceptions

- Alternatively, if a non locally remediable error happens within a method, we might be tempted to call System.exit(). Resulting code is not reusable!

```
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

- When we use return codes for handling errors
  - Code is messy to write and hard to read
  - Only the direct caller can intercept errors (no delegation to any upward method)

```
if (function() == ERROR_CODE) {
  // handle error
} else
  // proceed
}
```

An example, read a file into memory

- Open the file
- Determine file size
- Allocate (the needed amount of) memory
- Read the file into memory
- Close the file

All operations can fail!

First approach

- Short, readable BUT not reusable nor dependable

```
int loadFile()  {
    open the file;
    determine file size;
    allocate memory;
    read the file into memory;
    close the file;
    return  0;
}
```

Second approach

```
open the file;
if(operationFailed) return -1;

determine file size;
if(operationFailed) return -2;

allocate 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;
```

- Reusable, dependable BUT long and obscure
- A lot of error-detection and error-handling code
  - To detect errors we must check the specification of library calls
  - Each library has its own standards

Third approach, 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 (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)
```

Basic Concepts

- The code causing an error generates an exception
- At some point, up in the hierarchy of method invocations, a method might catch and handle the exception
- All methods can
  - Intercept the exception (no delegation)
  - Ignore the exception (complete delegation)
  - Intercept and generate a new exception (partial delegation)

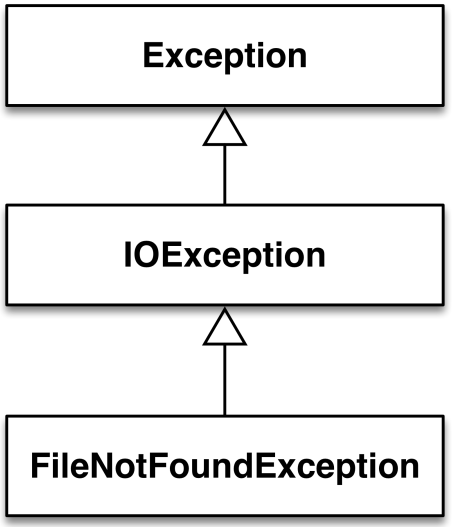
Syntax

- Java provides four keywords related with exceptions
  - Try
    - Contains code that may generate exceptions
  - Catch
    - Defines the error handler
  - Finally
    - Code block being executed regardless the catching phase
  - Throws
    - Mark a method as able to delegate exceptions
  - Throw
    - Generates a new exception

- There is also a class for representing exceptions
  - Exception class (java.lang.Exception)

Interception (catch)

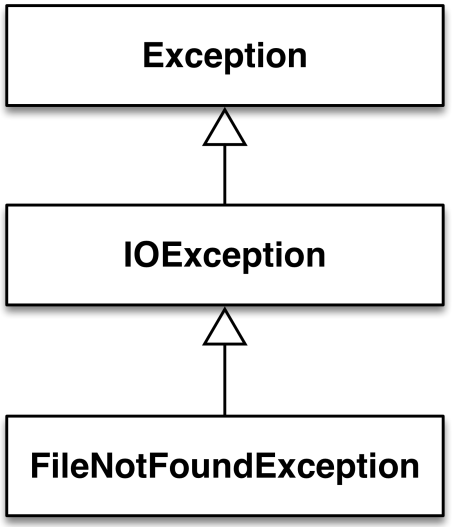
```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```



ING

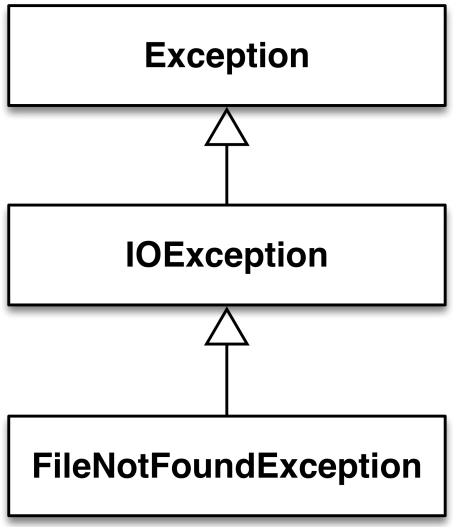
Interception (catch)

```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



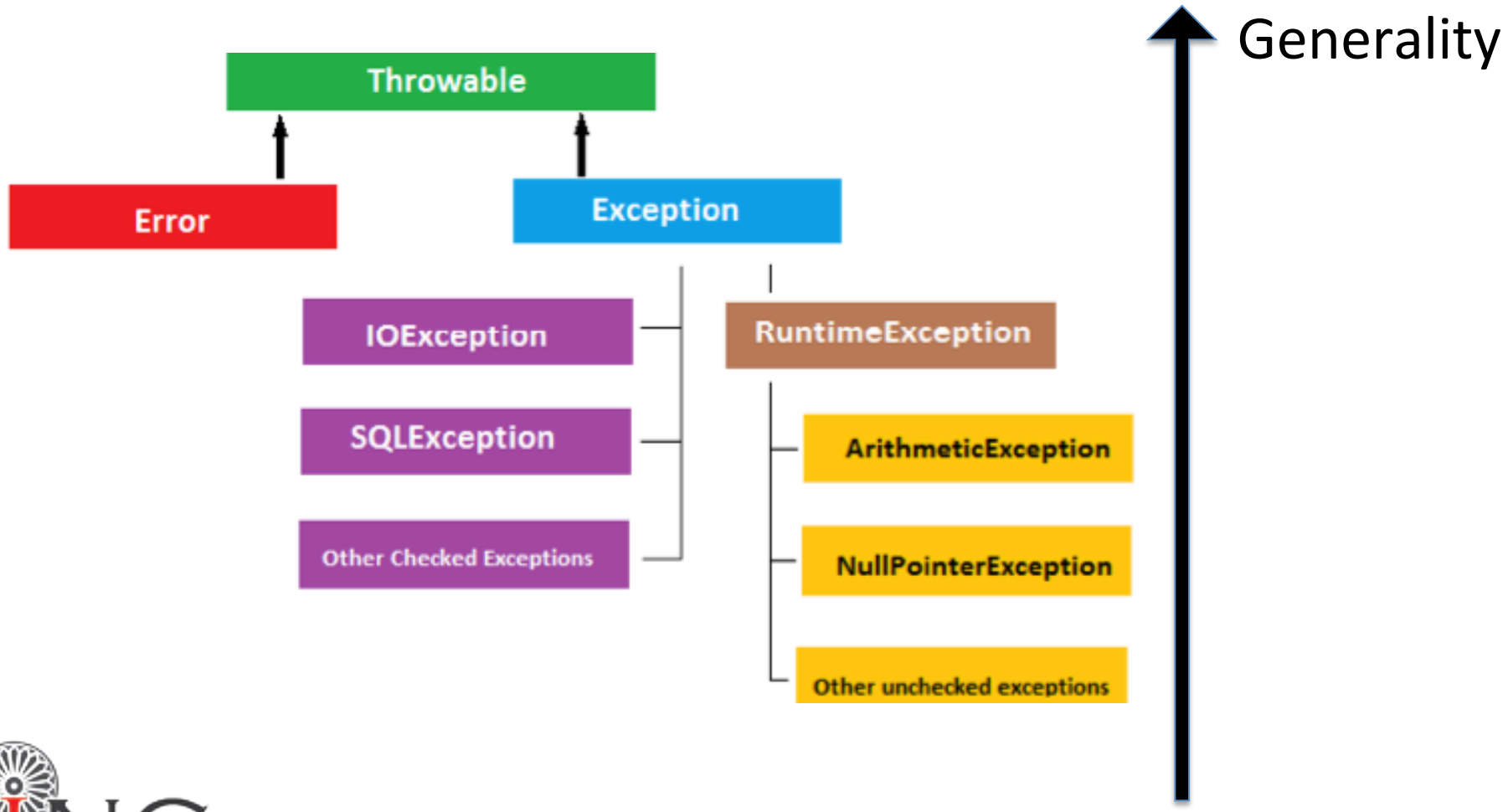
Interception (catch)

```
public static void main(String[] args) {
    char[] v = new char[256];
    try {
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```



ING

Matching Rules



ING

Delegation (throws)

- For enabling delegation, method interface must declare exception type(s) generated within its implementation (list with commas)

- Exception can be either generated:
  - by the method directly
  - by other methods called within the method and not caught

Delegation (throws)

```
public static void main(String[] args)
    throws IOException {
        char[] v = new char[256];
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
}
```

Delegation (throws)

```
Class Dummy {
    public void foo() throws IOException {
        char[] v = new char[256];
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    }
}

Class App {
    public static void main (String args[]) throws IOException {
        new Dummy().foo();
    }
}
```

Delegation (throws)

```
Class Dummy {
    public void foo() throws IOException {
        char[] v = new char[256];
        FileReader f = new FileReader("test.txt");
        f.read(v);
        f.close();
    }
}

Class App {
    public static void main (String args[]) {
        try {
            new Dummy().foo();
        } catch (IOException e) { // do something }
    }
}
```

Generation (throw)

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

Generation (throw)

```
public static void main(String[] args) {
    Queue<String> q = new LinkedList<String>();
    q.remove();
}
```

```
Exception in thread "main" java.util.NoSuchElementException
    at java.util.LinkedList.removeFirst(LinkedList.java:270)
    at java.util.LinkedList.remove(LinkedList.java:685)
    at main(Test.java:13)
```

Generation (throw)

```
public class LinkedList implements List, Queue {

  public Object removeFirst()
   throws NoSuchElementException {
    if (size() == 0) {
        throw(new NoSuchElementException());
   }
   …
  }
}
```

Generation (throw)

```
public class EmptyStackException extends Exception {}

 public Object removeFirst()
    throws EmptyStackException {
     if (size() == 0) {
         throw(new EmptyStackException());
    }
    …
  }
}
```

Partial delegation (catch, throw)

- Methods can intercept and exception, handle it eventually partially, and throw a new exception to be managed by their callers.
- The thrown exception can be either of the same type or of a different type

Partial delegation (catch, throw)

```
public static void main(String[] args)
throws IOException {
    char[] v = new char[256];
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
        throw (new IOException());
    }
}
```

Partial delegation (catch, throw)

```
public class MyException extends Exception {}

public static void main(String[] args)
throws IOException {
    char[] v = new char[256];
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
        throw (new MyException());
    }
}
```

Nesting

- Try/catch blocks can be nested (e.g., error handlers may generate new exceptions)

```
public static void main(String[] args) {
    try {
        FileReader f = new FileReader("test.txt");
    } catch (FileNotFoundException e) {
        try {
            FileWriter w = new FileWriter("log.txt");
            w.write("Error in opening test.txt");
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
}
```

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 {}`

Exceptions and loops

- For errors affecting a single iteration (or items of a collection!), 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){
        // discard iteration
    }
}
```

Exceptions and loops

- For 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.

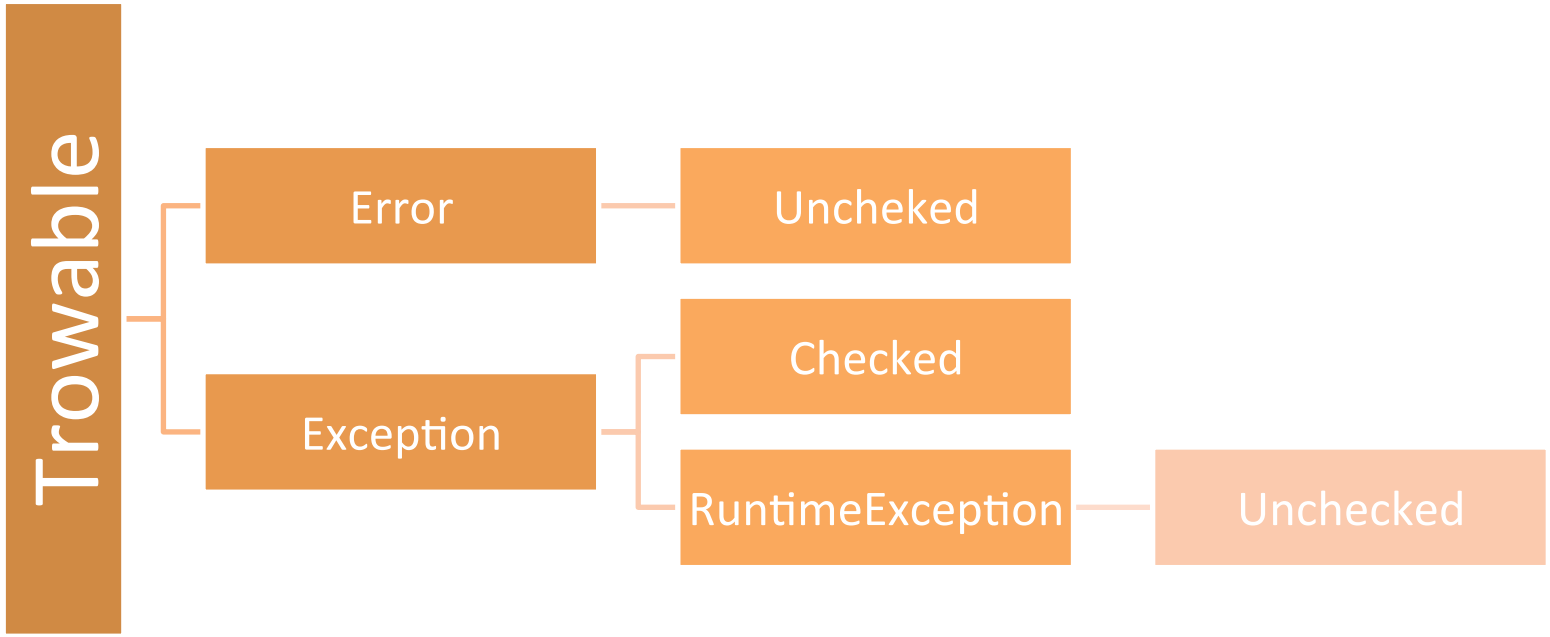
```
try{
    while(true){
        // potential exceptions
    }
} catch(Exception e){
    // discard whole loop
}
```

Finally

The JVM always executes the ***finally*** block regardless the outcome of try/ catch. Usually it is used for cleaning things up (e.g., closing files, connections)

```
public static void main(String[] args) {
    FileReader f;
    try {
        f = new FileReader("test.txt");
        f.read(v);
        f.close();
    } catch (IOException e) {
        // do something
    } finally {
        if (f != null) f.close();
    }
}
```

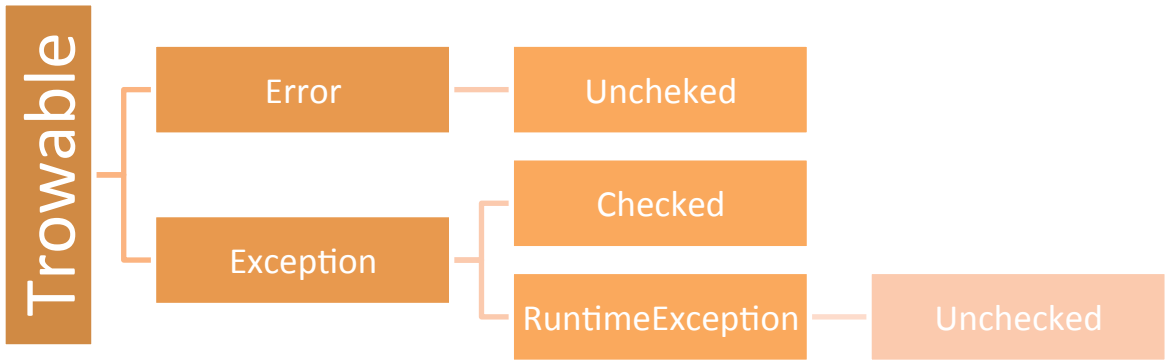
Exceptions and Errors



ING

Errors

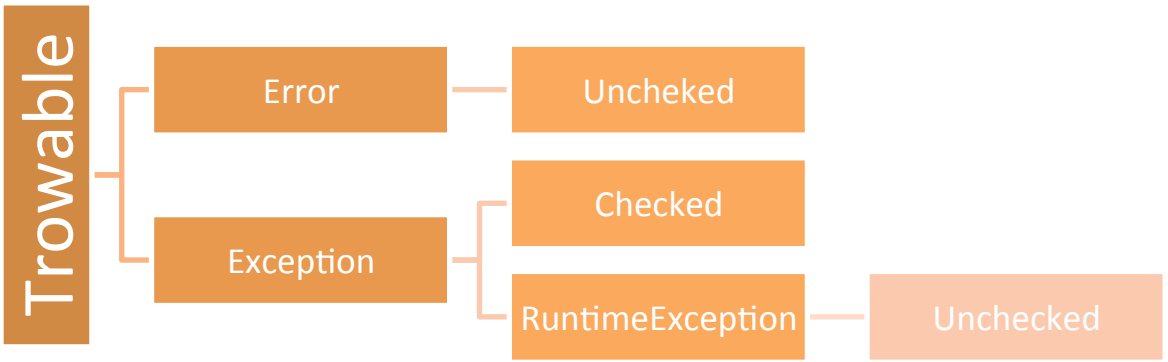
- An Error is a subclass of Throwable referring to serious issues that a reasonable application should not try to catch. Most of errors are truly abnormal conditions.
  - LinkageError indicates that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
  - VirtualMachineError indicates that the Java Virtual Machine is broken or has run out of resources



ING

Checked and Unchecked Exceptions

- Unchecked exceptions (Generated by JVM)
  - Their generation is not foreseen (can happen everywhere)
  - Need not to be managed with try/catch (not checked by the compiler). Examples are NullPointerException, ArrayIndexOutOfBound, …
- Checked exceptions
  - Exceptions manages with try/catch and checked by the compiler
  - Generated using throw. Examples are IOException, SQLException, ClassNotFoundException, …

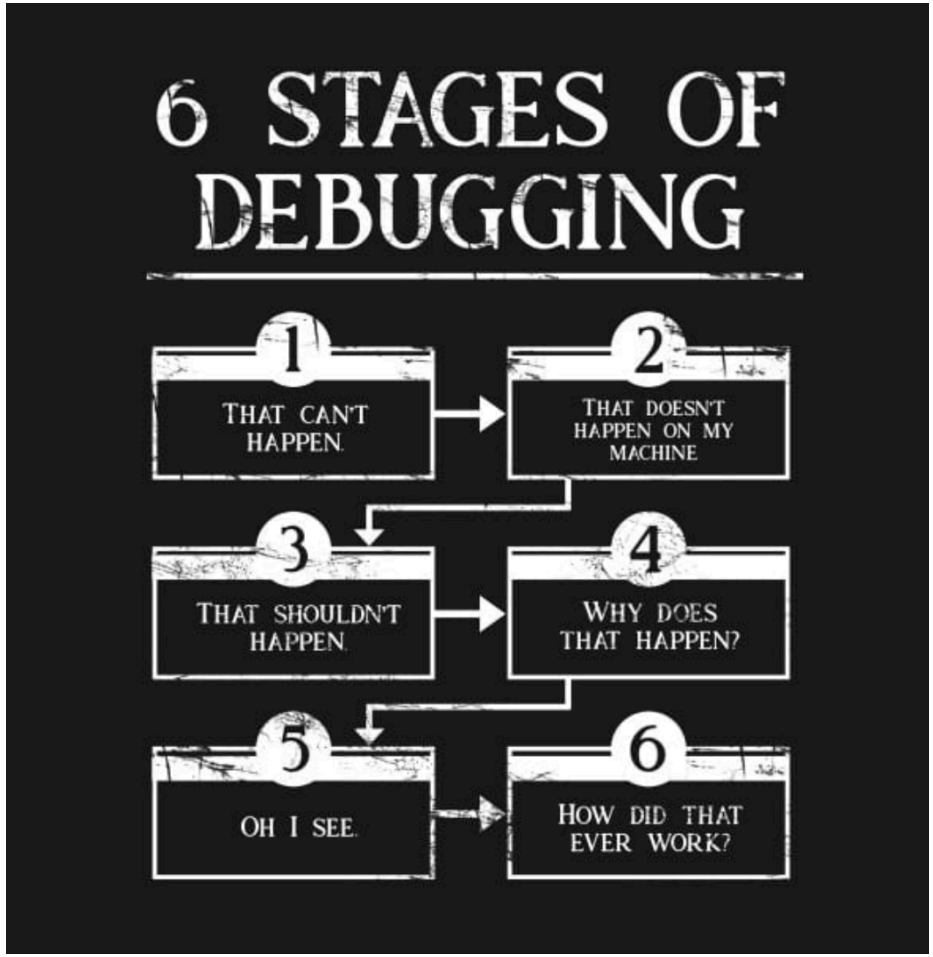


ING

Summarizing

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

Summarizing



ING