Java Exceptions

Object Oriented Programming

SOftEng
http://softeng.polito.it

Version 2.2.2 - May 2016

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

Coding 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)

Coding without exceptions (II)

- If errors happen 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

Real problems

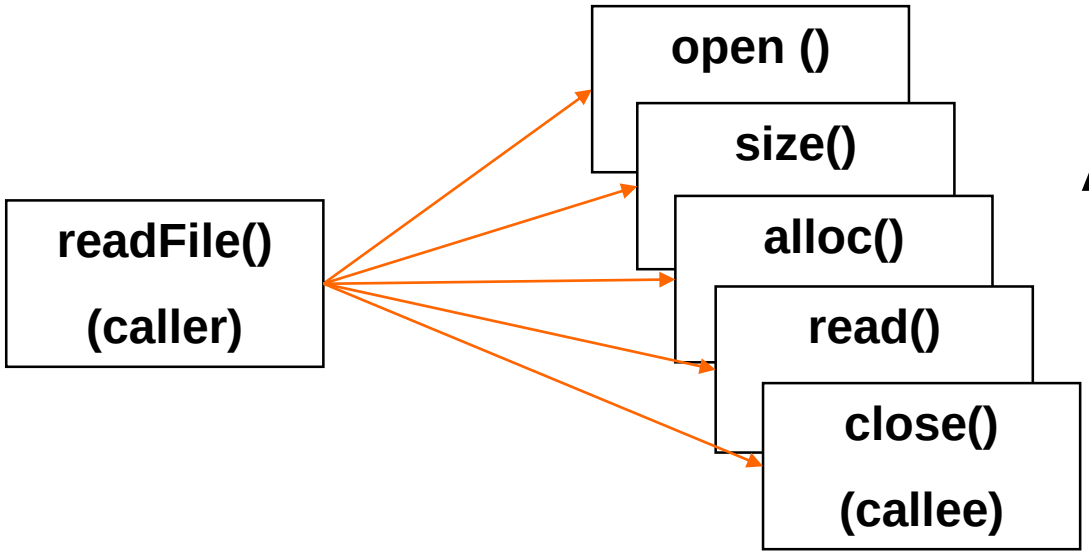
- Code is messier to write and harder to read

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

- Only the direct caller can intercept errors (no delegation to any upward method)
  - **Unless additional code is added**

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)

Wihtout error handling

```
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 handle the exception
- In between, methods can
  - Ignore the exception (complete delegation)
  - Intercept and re-issues (partial delegation)

Syntax

- Java provides three keywords
  - `throw`
    - Generates an exception
  - `try`
    - Contains code that may generate exceptions
  - `catch`
    - Defines the error handler
- We also need a new entity
  - `Exception` class

Generating Exceptions

1. Identify an exception class
2. Have the method generating the exception marked
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`

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

`throw`

- When an exception is thrown:
  - The execution of the current method is interrupted immediatelly
  - The code immediately following the **`throw`** is not exectuted
    - Similar 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 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

If **`close`** fails

- "*File error*" is printed
- Eventually program terminates with "*End*"

```
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:

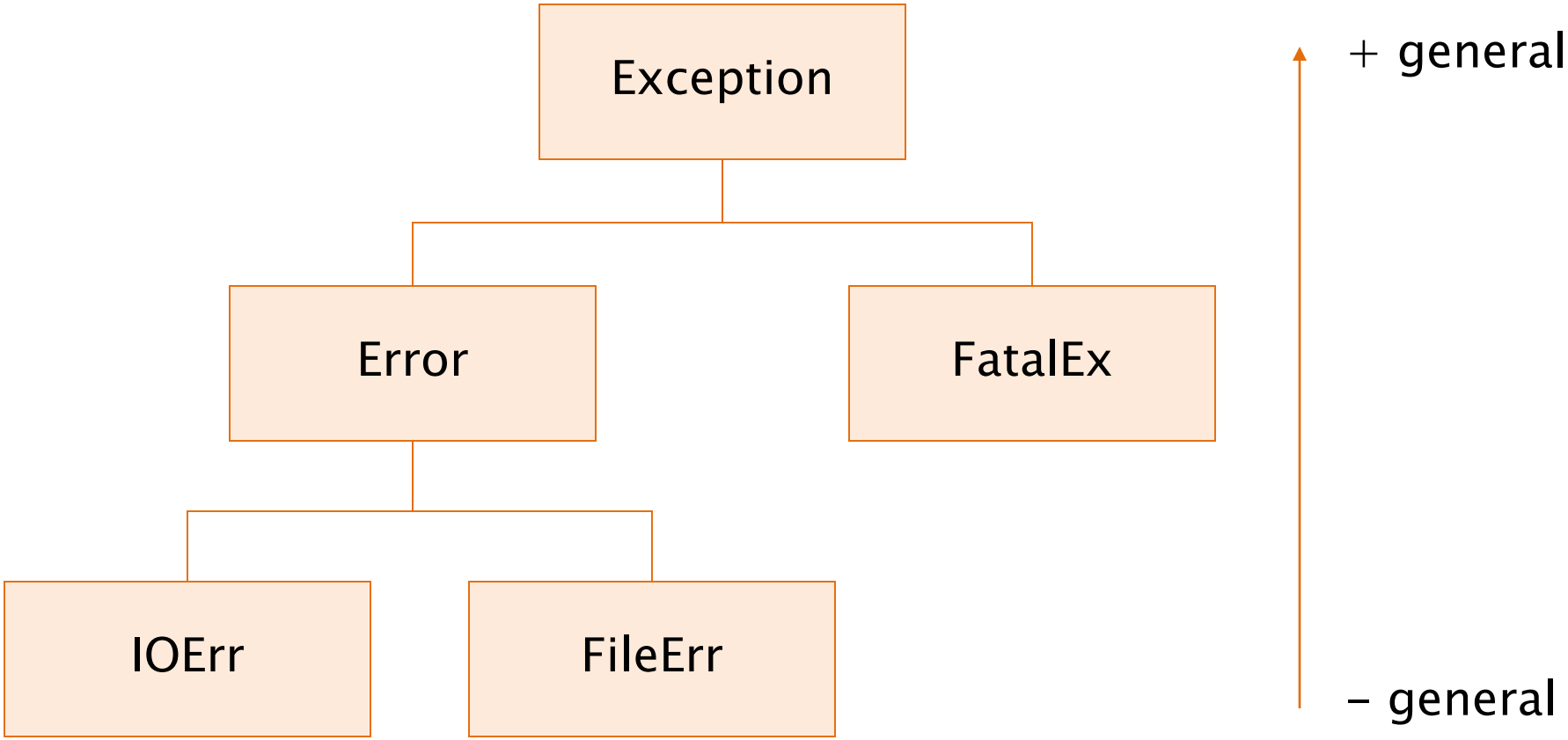
- "*I/O error*" is printed
- Eventually program terminates with "*End*"

```
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 most specific handler is selected, according to the exception type order
- Handlers are ordered according to their "generality"
  - From the most general (base classes) to the most specific (derived classes)
  - Most general are the first to be selected

Matching rules



Matching rules

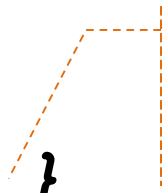
```
class Error    extends Exception{}
class IOErr    extends Error{}
class FileErr  extends Error{}
class FatalEx  extends Exception{}
```

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

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){ /*…*/ }
```

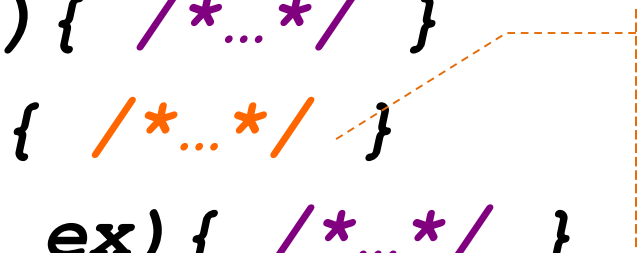


IOErr is generated

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){ /*…*/ }
```



Error or FileErr 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){ /*…*/ }
```

FatalEx is generated

Nesting

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

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

Exception checking

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

Checking: Catch

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

Checking: Propagate

```
class Dummy {

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

}
```

Checking: Propagate (cont'd)

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

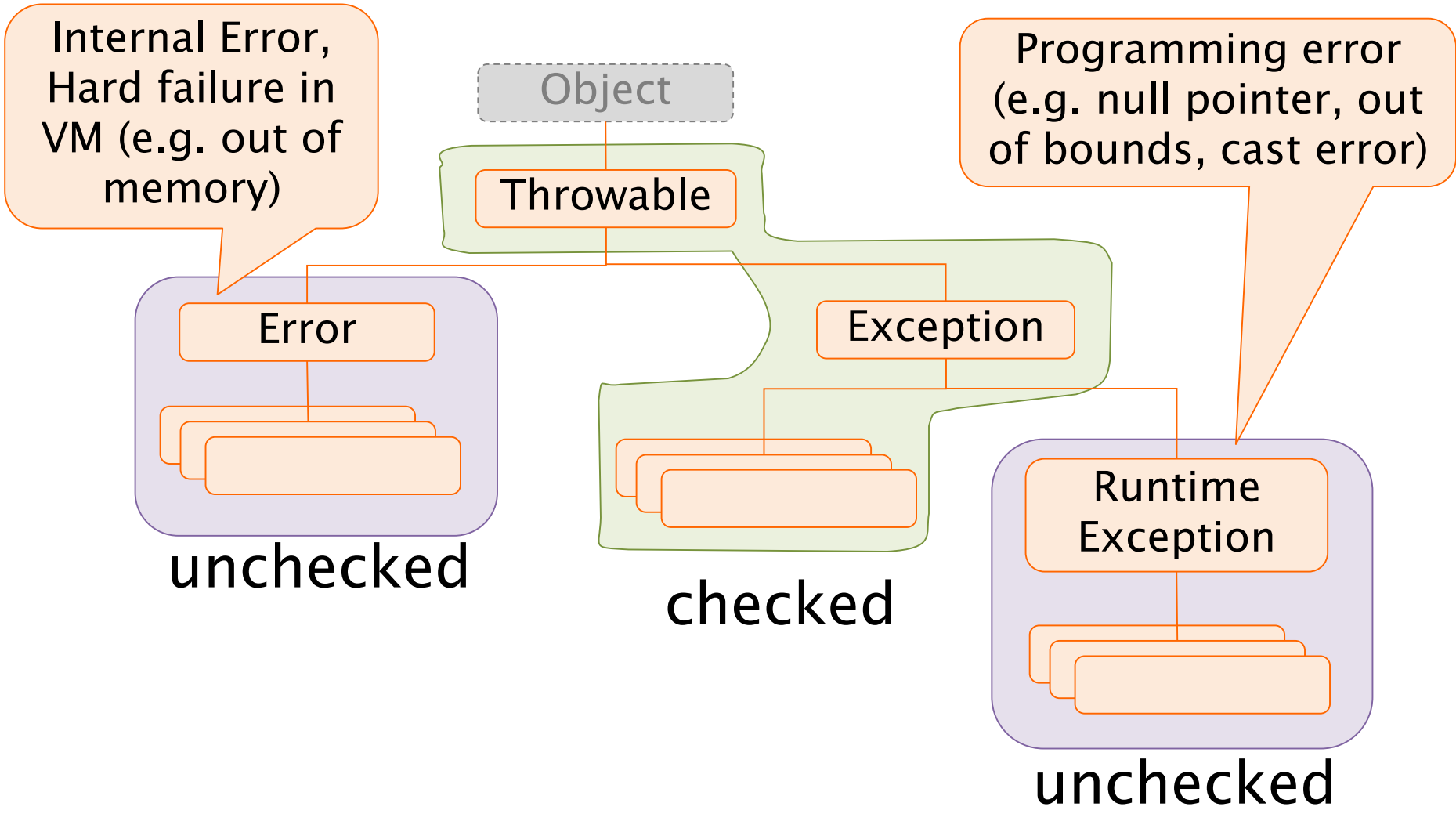
```
class Dummy {
    public void foo()
       throws FileNotFound {
       FileReader f = new FileReader("file");
    }
}
```

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

Checking: 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)
  - Generated by JVM
- Checked exceptions
  - Exceptions declared and checked
  - Generated with “throw”

Main Exception classes

- Error
  - `OutOfMemoryError`
- Exception
  - `ClassNotFoundException`
  - `InstantiationException`
  - `NoSuchMethodException`
  - `IllegalAccessException`
  - `NegativeArraySizeException`
  - `EmptyStackException`
- RuntimeException
  - `NullPointerException`

Application 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 extend **`Exception`**

finally

- The keyword **`finally`** allows specifying actions that must be executed in any case, e.g.:
  - 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
}
```

Testing exceptions

- Two main cases shall be checked:
- We expect an anomaly and therefore an exception should be rised
  - In this case the tests fails whether NO exception is detected
- We expect a normal behavior and therefore no exception should be raised
  - In this case the tests fails whether that exception in raised

Expected exception test

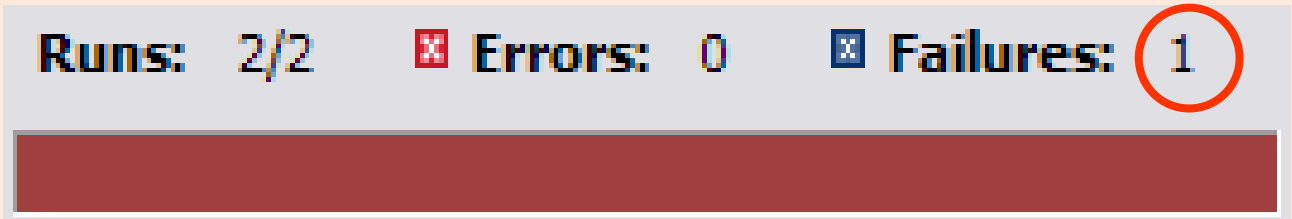
```
try{
   //  e.g. method invoked with “wrong” args
   obj.method(null);
   fail(“Method didn’t detected an anomaly");
}catch(PossibleException e){
   assertTrue(true); // OK
}
```

```
class TheClassUnderTest {
 public void method(String p)
throws PossibleException
   { /*... */ }
}
```

Unexpected exception test

```
try{
   //  e.g. method invoked with right args
   obj.method("Right Argument");
   assertTrue(true); // OK
}catch(PossibleException e){
 fail("Method should not raise except.");
}
```

Exception ➔ Failure



Unexpected exception test

```
public void testSomething()
      throws PossibleException {
   //  e.g. method invoked with right args
    obj.method("Right Argument");
}
```

Exception → Error

Summary

- 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

Summary

- 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

Summary

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