Java Exceptions

Object-Oriented Programming

SOftEng
http://softeng.polito.it

Version 3.0.0

Licensing Note

Motivation

- Report anomalies, by delegating error handling to higher levels
  - Methods detecting anomalies might not be able to recover from an error
  - Caller method can handle errors more suitably than the detecting method itself
- Localize error handling code by separating it from operating code
  - Operating code is more readable
  - Error handling code is collected in a single place, instead of being scattered

Anomalies in programs

- Detection
  - Check conditions revealing an anomaly
- Signaling
  - Inform the caller about the anomaly
- Dispatch
  - Receive and redirect the anomaly signal
- Handling
  - Perform operation to address an anomaly

Error signaling techniques

- Program abort (handling)
  - Abrupt termination of the execution
- Special value
  - Return a special value to indicate error
- Global status
  - Global variable contain error reports
- Exceptions
  - Throw an exception

Error signaling/handling: abort

- If a non-remediable error happens, call `System.exit()`
  - Abort program execution, VM does not perform any cleanup or resource release
  - A method causing an unconditional program interruption is not very dependable (nor usable)
- NEVER EVER CALL **`System.exit()`** in your programs
  - Makes it untestable

Error signaling: special value

- If an error happens, return a special value
- Special values are distinct from normal values returned

```
pb.find("non-exist");
```

null

```
"ABCD".indexOf("F");
```

-1

```
Math.pow(-1, 0.5);
```

NaN

- What if special values are normal?
  - `"" + null`

"null"

Error handling code

- Code is messy to write and hard to read

```
if( someMethod() == ERROR ) // acknowledge
    //handle the error
else
    //proceed normally
```

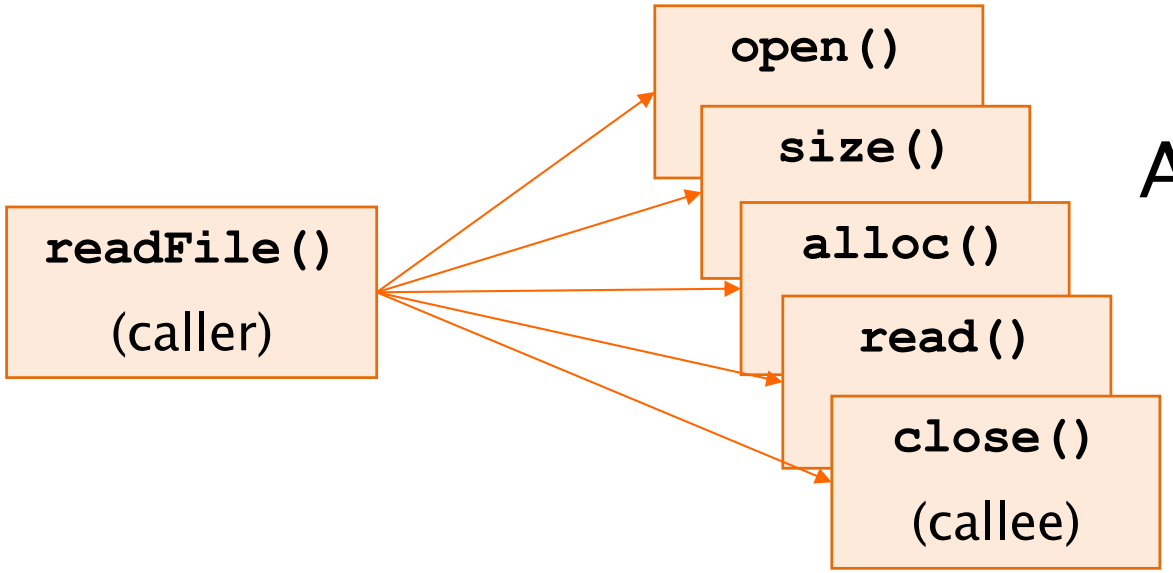
- Only the direct caller can intercept errors
  - no simple delegation to any upward method
  - unless further additional code is added
- Developer must remember value/meaning of special values to check for errors

Global error variable

- In C many function set the global variable **`errno`** to signal that an error occurred during an operation
  - See: http://man7.org/linux/man-pages/man3/errno.3.html
- In Java, such error signaling approach is never used

Example – Read file

- open the file
- determine file size
- allocate that much memory
- read the file into memory
- close the file



Any of them can fail

No error handling

```
int readFile() {

    open();
    int n = size;
    alloc(n);
    read();
    close();

    return 0;
}
```

Special return code

```
int readFile() {
   open();
   if (operationFailed)
      return -1;
   int n=size();
   if (operationFailed)
      return -2;
   alloc();
   if (operationFailed) {
      close the file;
      return -3;
   }
   read();
   if (operationFailed) {
      close the file;
      return -4;
   }
   close();



   return 0;
}
```

Lots of error-detection and error-handling code

To detect errors we must check specs of library calls (no homogeneity)

No formal verification that we do not forget any checks

Using exceptions

```
try {
    open();
    int n = size;
    alloc(n);
    read();
    close();
} catch (fileOpenFailed) {
        doSomething;
} catch (sizeDeterminationFailed) {
        doSomething;
} catch (memoryAllocationFailed) {
        doSomething;
} catch (readFailed) {
        doSomething;
} catch (fileCloseFailed) {
        doSomething;
}
```

Basic concepts

- The code detecting the the error will throw an exception, it can be either
  - 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, dispatching methods can
  - Relay the exception (complete delegation)
  - Intercept and re-throw (partial delegation)

Syntax

- Java provides four keywords
  - `throw`
    - Throws an exception
  - `throws`
    - Declare a potential exception
  - `try`
    - Introduces code to watch for exceptions
  - `catch`
    - Defines the exception handling code
- It also defines a new type
  - `Throwable` class

Generating Exceptions

1. Identify/define an exception class
2. Declare some methods as potential sources of exception
3. In the methods:
   a. Check condition, and if verified
   b. Create an exception object
   c. Throw the exception

Generation

```
public class EmptyStack extends Exception{}
```
(1)

```
public class Stack{
    public int pop() throws EmptyStack {  (2)

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

Operator `throw`

- Performs the exception throw
- When an exception is thrown, the execution of the current method is interrupted immediately
  - The code immediately following the **`throw`** statement is not executed
  - Like a **`return`** statement
- The catching phase starts

Declaration `throws`

- If a method might generate an exception, it must must declare it in its signature
  - All exception type(s) are listed after the **`throws`** keyword
- Allow checking dispatching by caller
- Must declare exception thrown both
  - directly by the method, or
  - by called methods and relayed

Exception dispatching

- When a fragment of code can possibly generate an exception, the exception must be dispatched:
  - Relay the exception and let it propagate up the call stack
    - Method has a **`throws`** declaration,
  - Catch, stop the exception, and handle it
    - Code enclosed in **`try{}catch(){}`** statement
  - Catch, partially handle, and re-throw

Run-time catching phase

- Once an exception is thrown the normal execution is suspended
- The thrown exception *"walks back"* the call stack until either:
  - It is caught by one of the methods
  - It overtakes **`main()`**
    - In this case the JVM prints the exception (and the full stack trace) and terminates execution

Relay

```
class Dummy {
   Stack st;
   public int foo() throws EmptyStack{
      int v = st.pop();
      return v + 1;
   }
}
```

Not executed in case of an exception

Relay

- Exception not caught can be relayed until the **`main()`** method and the JVM

```
class Dummy {
  Stack st;
  public int foo() throws EmptyStack{
    int v = st.pop();
    return v + 1;
  }
}
```

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

Catch and handle

```
class Dummy {
   Stack st;
   public int foo(){
      try{
         int v = st.pop();
         return v + 1;
      } catch (StackEmpty se) {
          // do something
      }
      return 0; // default value
} }
```

Not executed in case of an exception

Note: all paths must end with a return

Catch and re-throw

```
class Dummy {
   Stack st;
   public void foo() throws EmptyStack{
      try{
         int v = st.pop();
         return v + 1;
      } catch (StackEmpty se) {
         // intermediate handling
         throw se;
      }
} }
```

Not executed in case of an exception

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 – no exception

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 – exception

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");
```

EXCEPTION CLASSES

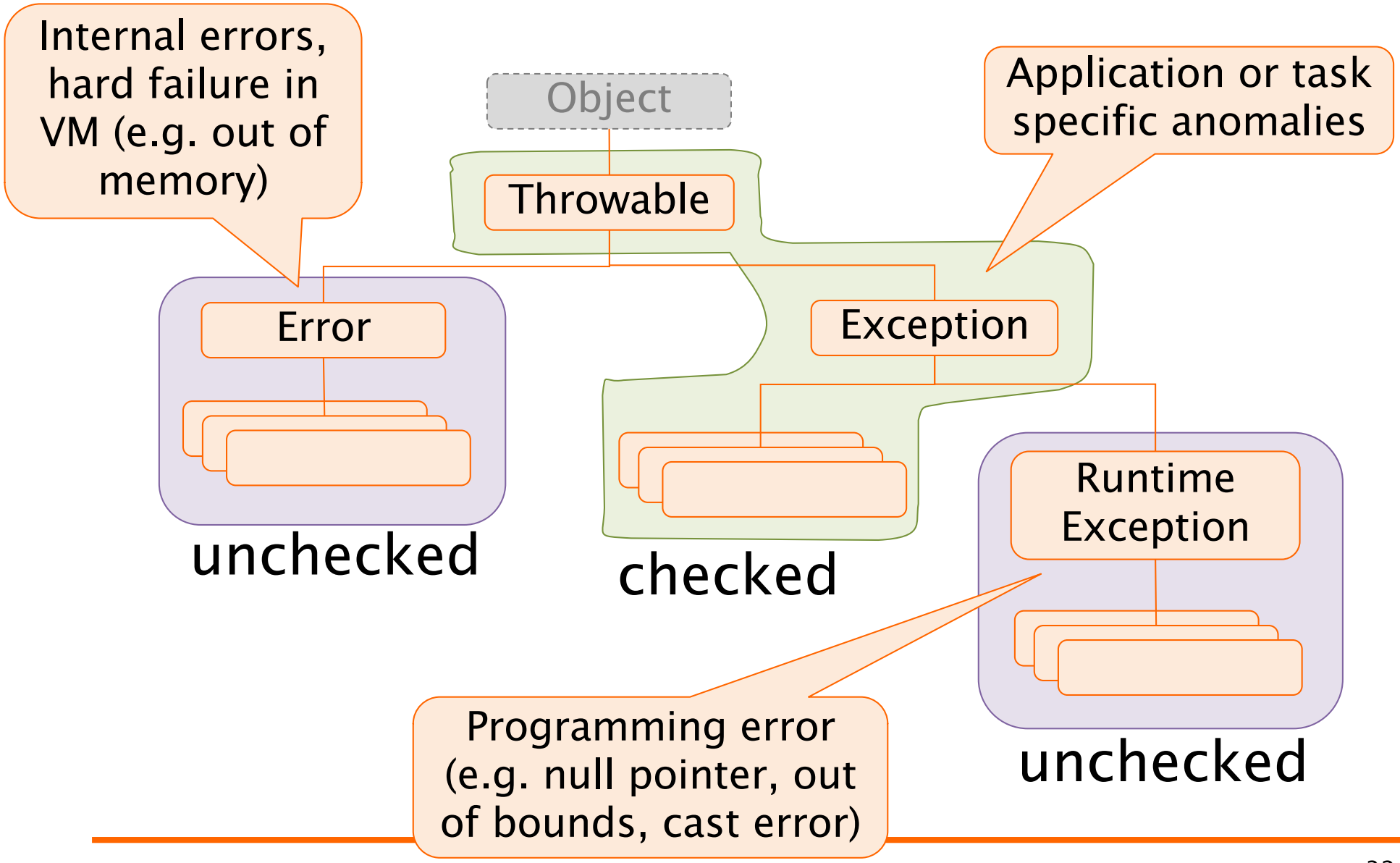
Class **`Throwable`**

- Exception classes must extend class **`Throwable`**
- Contains a snapshot of the call stack
- May contain a message string
  - provides information about the anomaly
- May also contain a `cause`
  - another exception that caused this one to be thrown

Class **Throwable**

- `getMessage()`
  - returns the error message associated with the exception
- `getCause()`
  - Return a possible other exception that caused this one
- `printStackTrace()`
  - Prints the stack trace until the place when the exception was created
  - The traces is automatically filled in by Throwable constructor
  - This is the method called by JVM on uncaught exceptions

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
  - Typically generated by JVM
- Checked exceptions
  - Exceptions must be declared
    - checked by the compiler
  - Generated with **`throw`**

Exception classes examples

- **`Error`**
  - **`OutOfMemoryError`**
- **`Exception`**
  - **`ClassNotFoundException`**
  - **`InstantiationException`**
  - **`IOException`**
  - **`InterruptedException`**
- **`RuntimeException`**
  - **`NullPointerException`**
  - **`ClassCastException`**

Application specific exceptions

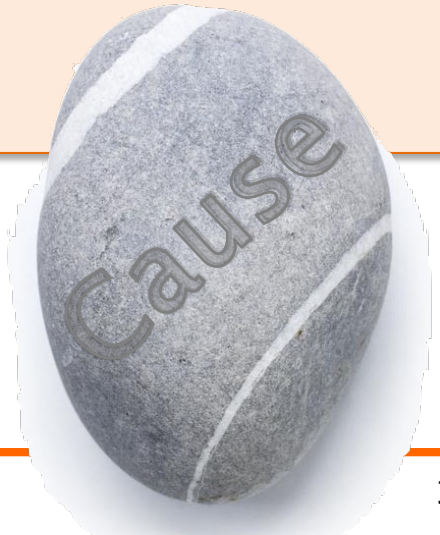
- Represent anomalies specific for the application
- Usually extend **`Exception`**
- Can be caught separately from the predefined ones
  - Allow more fine-grained control than using just **`Exception`**

Application specific exceptions

- Exceptions are like stones
  - When they hit you, they first matters because they exists and are thrown, then for their message

```
class Stone
extends Throwable
{}
```

```
class MsgStone
extends Exception {
public MsgStone(String m){
   super(m); }
}
```



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
   } // and continue with next iteration
}
```

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){ // exit the loop and …
    // handle the anomaly
}
```

Unchecked and loop

```
String[] strings =
{"1","2","III","4","V","6"};
int sum = 0;
for(String s : strings) {
  sum += Integer.parseInt(s);
}
System.out.println("Sum: " + sum);
```

```
NumberFormatException: For input string: "III"
```

Unchecked and loop

```
try{
   int sum = 0;
   for(String s : strings) {
      sum += Integer.parseInt(s);
   }
   System.out.println("Sum: " + sum);
}catch(Exception e){
   System.err.println("Error!");
}
```

Error!
No sum computed

Unchecked and loop

```
int sum = 0;
for(String s : strings) {
   try{
      sum += Integer.parseInt(s);
   }catch(NumberFormatException e){
      System.err.println("Wrong: "+s);
   }
}
System.out.println("Sum: " + sum);
```

```
Wrong III
Wrong V
```

```
Sum: 13
```

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 */ }
}
```

Unchecked and loop

```
sum = 0;
for(String s : strings) {
   try {
      sum += Integer.parseInt(s);
   }catch(NumberFormatException nfe) {
      try {
         sum += parseRoman(s);
      }catch(NumberFormatException re) {
         System.err.println("Wrong " + s);
} } }
System.out.println("Sum: " + sum);
```

Sum: 21

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
}
```

Matching rules

- Only one handler is executed
  - The first one matching the thrown exception
  - A **`catch`** matches if the thrown exception is **`instanceof`** the **`catch`**'s exception class
- Catch blocks must be ordered by their "generality"
  - From the most specific (derived classes) to the most general (base classes)
  - Placing the more general first would obscure the more specific, making them unreachable

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:

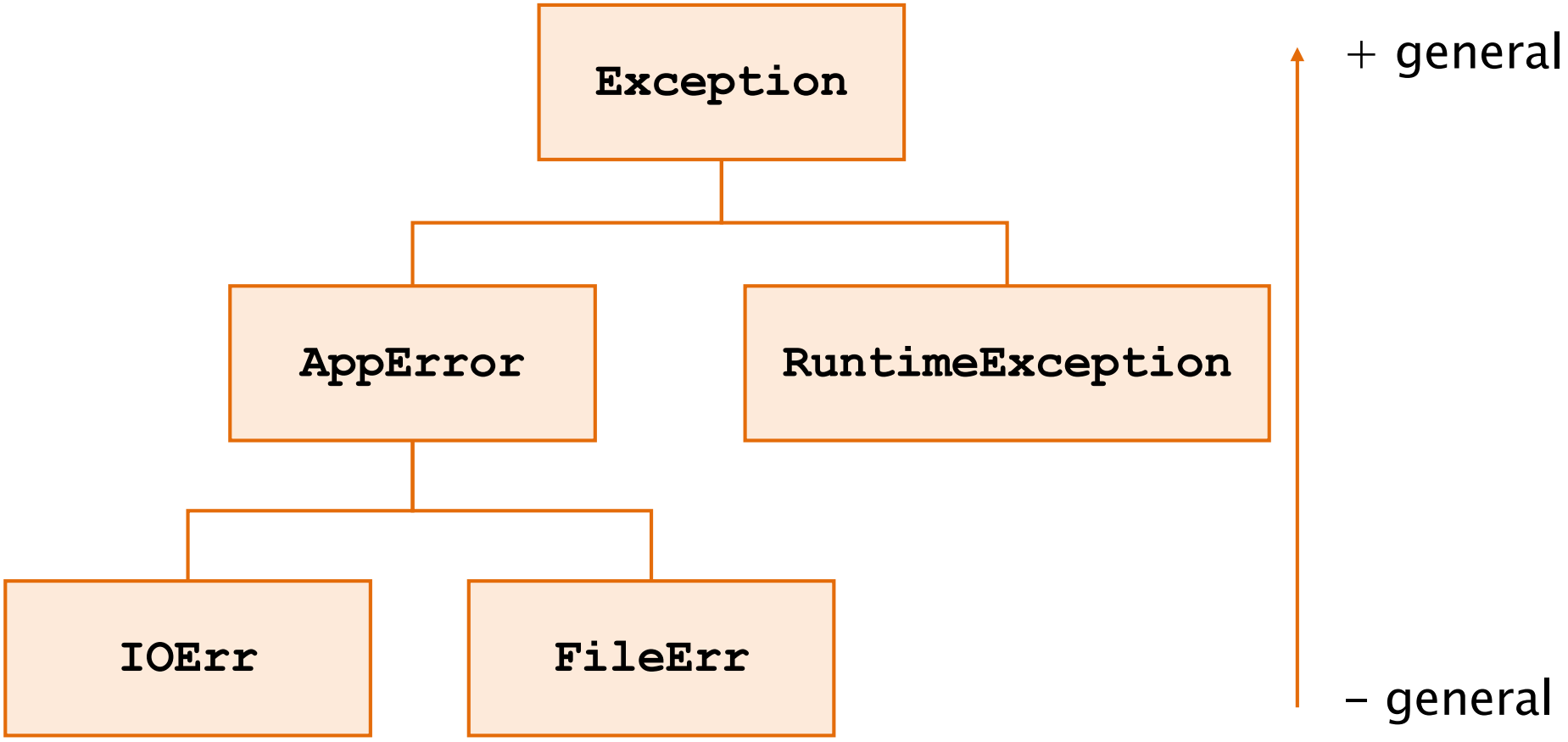
- "*I/O error*" is printed
- Eventually program terminates with "*End*"

```
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 example



Matching rules example

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

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

– general

+ general

Keyword `finally`

- The keyword **`finally`** introduces a code block that is executed in any case
  - No exception
  - Caught exception
  - Uncaught exception
    - Both checked and unchecked
  - Does not work in case of **`System.exit()`**
- Can be used to
  - Dispose of resources
  - Close a file

Keyword finally

```
MyFile f = new MyFile();
if (f.open("myfile.txt")) {
    try {
        exceptionalMethod();
    }catch(IOException e){
      //…
    } finally {
        f.close();
    }
}
```

After all catch branches (if any)

Critical Resources

- Some objects consume OS resources
  - E.g. input/output streams, db connections, etc.
- Such resources are limited
  - E.g., a program can open only a given number of files at once
- Such objects should be closed as soon as possible to free shared and limited resources

Close and exceptions

```
String readFile(String path)
                          throws IOException{
  FileReader fr = new FileReader(path);

  int ch = fr.read();
  // …
  fr.close();
  return String.valu    (ch);
}
```

What happens in case of exception in **`read()`** ?

Catch and close

```
String readFile(String path)
                          throws IOException {
  FileReader fr = new FileReader(path);
  try {
    int ch = fr.read();
    // …
    fr.close();
    return String.valueOf(ch);
  } catch(IOException e){
    fr.close();
    throw e;
  }
}
```

Complex and does not close in case of other exceptions

Finally close

```
String readFile(String path)
                          throws IOException {
  FileReader fr = new FileReader(path);
  try {
    int ch = fr.read();
    // …
    fr.close();
    return String.valueOf(ch);
  }finally {
    if(fr!=null) fr.close();
    throw e;
  }
}
```

Executed in any case before exiting the method

Try-with-resource

```
String readFile(S
    try(
    FileReader fr = new FileReader(path))){
      int ch = fr.read();
      // …
      fr.close();
      return String.valueOf(ch);
    }
}
```

Works since **FileReader** implements **Autocloseable**

More compact and readable form, equivalent to the one with finally

```
public interface AutoCloseable{
  public void close();
}
```

Summary

- Exceptions provide a mechanism to manage 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 dispatched by
  - Catching them with **`try{ }catch{ }`**
  - Relaying with **`throws`**
  - Catching and re-throwing
- Unchecked exception can avoid mandatory dispatching
  - All exceptions extending **`Error`** and **`RuntimeException`**
- The **`finally`** blocks can be used to execute some code in any possible case
  - the try-with-resource construct makes it easier to use them