3.6 Exceptions

Software Analysis and Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Exceptions: Table of contents

- **Introduction**
- Working scheme
- Types
- Creating new types of exceptions
- Exception handling
- Raising (throwing) exceptions

Introduction: Exceptions

- **What is an exception?**
  - An event that happens during the normal execution of a program, which is associated with an *exception object* notifying the event
  - It interrupts the normal flow of execution

- **When to use exceptions?**
  - Only in exceptional situations:
    - Errors during data type conversions
    - Physical limits (e.g., of the hard disk, memory)
    - Device failures
    - Programming errors
    - …

Example: With no error handling

```
public void readFile() {
    open file;
    compute file size;
    allocate required amount of memory;
    read file contents into memory;
    close file;
}
```

Handling errors using “C style”

```
public errorCodeType readFile() {
    initialize errorCode = 0;
    open file;
    if (file opened) {
        compute file size;
        if (size was obtained) {
            allocate that amount of memory;
            if (not enough memory) {
                read file contents into memory;
                if (read failure) errorCode = -1;
            }
            else errorCode = -2;
        }
        else errorCode = -3;
        close file;
        if (file not closed && errorCode == 0)
            errorCode = -4;
        else errorCode = errorCode & -4;
    }
    else errorCode = -5;
    return errorCode;
}
```

Using Exceptions

```
public void readFile() {
    try {
        open file;
        compute file size;
        allocate required amount of memory;
        read file contents into memory;
        close file;
    }
    catch (fileOpenFailed) { doSomething; }
    catch (sizeDeterminationFailed) { doSomething; }
    catch (memoryAllocationFailed) { doSomething; }
    catch (readFailed) { doSomething; }
    catch (fileCloseFailed) { doSomething; }
}
```

Exceptions

- Error → an *exception object* is generated
- An exception object contains:
  - Error type
  - Error message
  - Program state when the error happened
- Exceptions can be generated by code

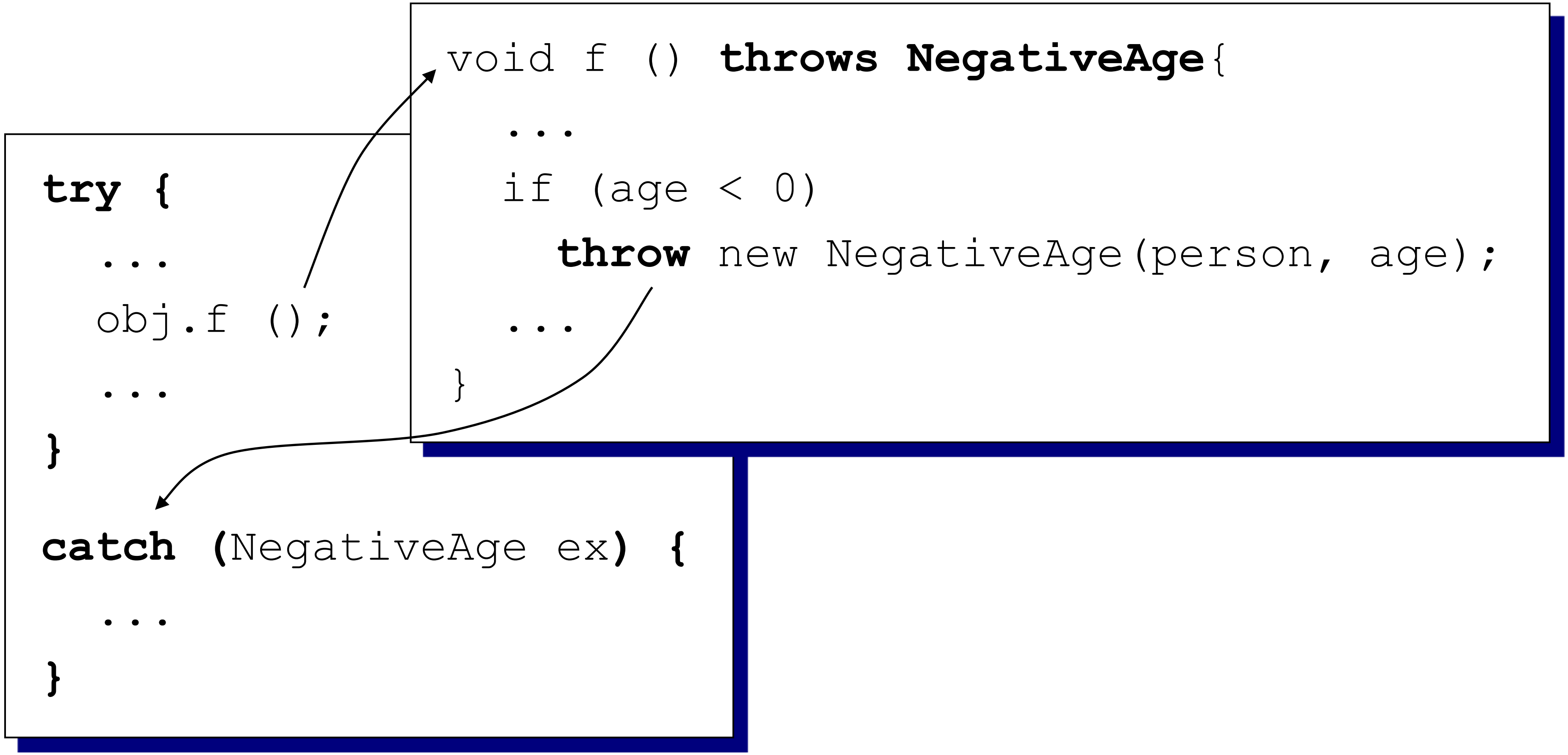
→ using **`throw`**

```
if (unexpected()) throw new Exception("error");
```

Handler blocks `try/catch/finally`

```
try {
    ...  // block protected by exception handlers declared below
}
catch (ExceptionType1 e1) {
    ...     // handling exception e1
}
catch (ExceptionType2 e2) {
    ...     // handling exception e2
}
...
catch (ExceptionTypeN eN) {
    ...     // handling exception e_N
}
finally {
    ...     // optional: if it exists it will be executed
}           // e.g., for resource deallocation
```

Example



```
try {
   ...
   obj.f ();
   ...
}
catch (NegativeAge ex) {
   ...
}
```

```
void f () throws NegativeAge{
   ...
   if (age < 0)
     throw new NegativeAge(person, age);
   ...
}
```

Example: Throwing Exceptions

```
class BankAccount {
   ...
   boolean blocked;
   ...
   void withdraw(long amount) throws UnderfundedException,
                                    BlockedAccountException {
      if (blocked)
         throw new BlockedAccountException(acctNumber);
      else if (amount > balance)
         throw new UnderfundedException(acctNumber, balance);
      else balance -= amount;
   }
}
```

Declaring exceptions as classes

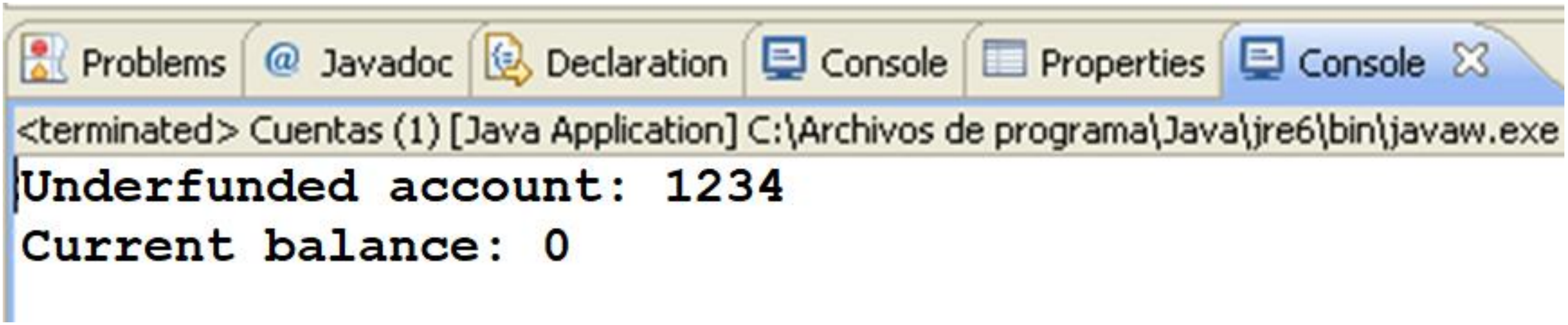
```
class UnderfundedException extends Exception {
    long acctNumber, balance;
    UnderfundedException(long num, long bal) {
        acctNumber = num; balance = bal;
    }
    public String toString () {
        return "Underfunded account: " + acctNumber
               + "\nCurrent balance: " + balance;
    }
}

class BlockedAccountException extends Exception {
    long acctNumber;
    BlockedAccountException(long num) { acctNumber = num; }
    public String toString () {
        return "Acct: " + acctNumber + " is blocked";
    }
}
```

Catching exceptions

Only the first catch block that is compatible will be executed

```
public static void main(String args[]) {
   try {
      new BankAccount(1234,0).withdraw(100000);
   }
   catch (UnderfundedException excep) {
      System.out.println(excep);
   }
   catch (BlockedAccountException excep) {
      System.out.println(excep);
   }
}
```

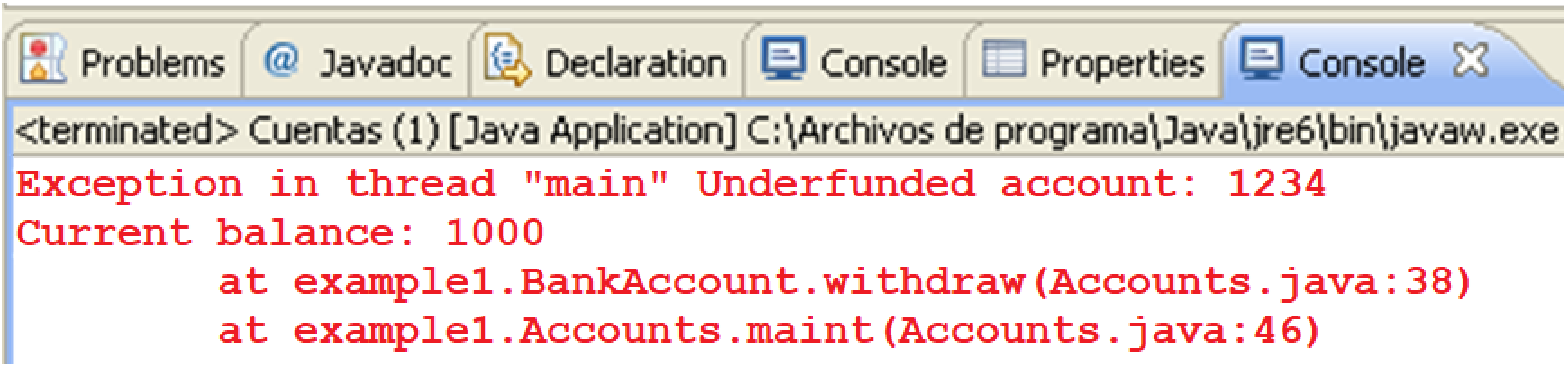


<terminated> Cuentas (1) [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe

```
Underfunded account: 1234
Current balance: 0
```

… and what they are not caught?

```
public static void main(String args[])
            throws  BlockedAccountException,
                    UnderfundedException {
   BankAccount account = new BankAccount(123,1000);
   cuenta.withdraw(2000);
}
```

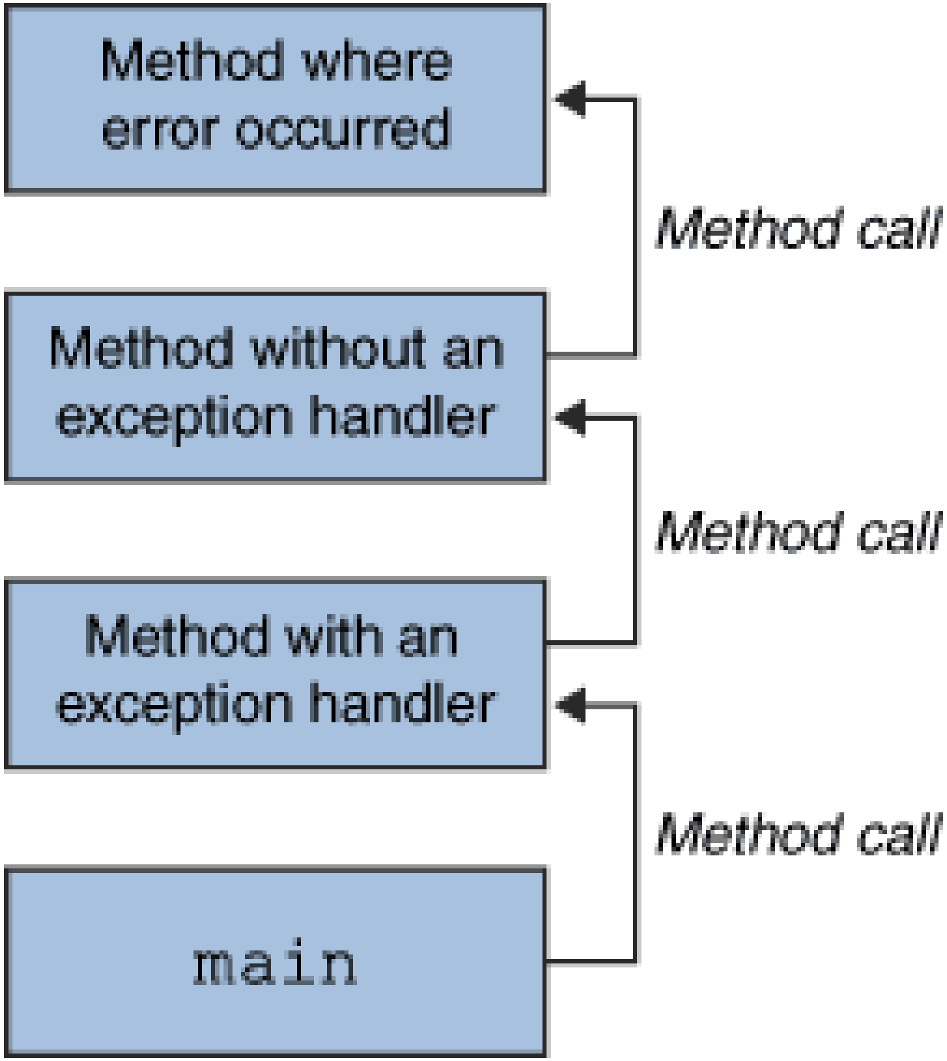


```
<terminated> Cuentas (1) [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe
Exception in thread "main" Underfunded account: 1234
Current balance: 1000
        at example1.BankAccount.withdraw(Accounts.java:38)
        at example1.Accounts.maint(Accounts.java:46)
```

Exceptions: Table of contents

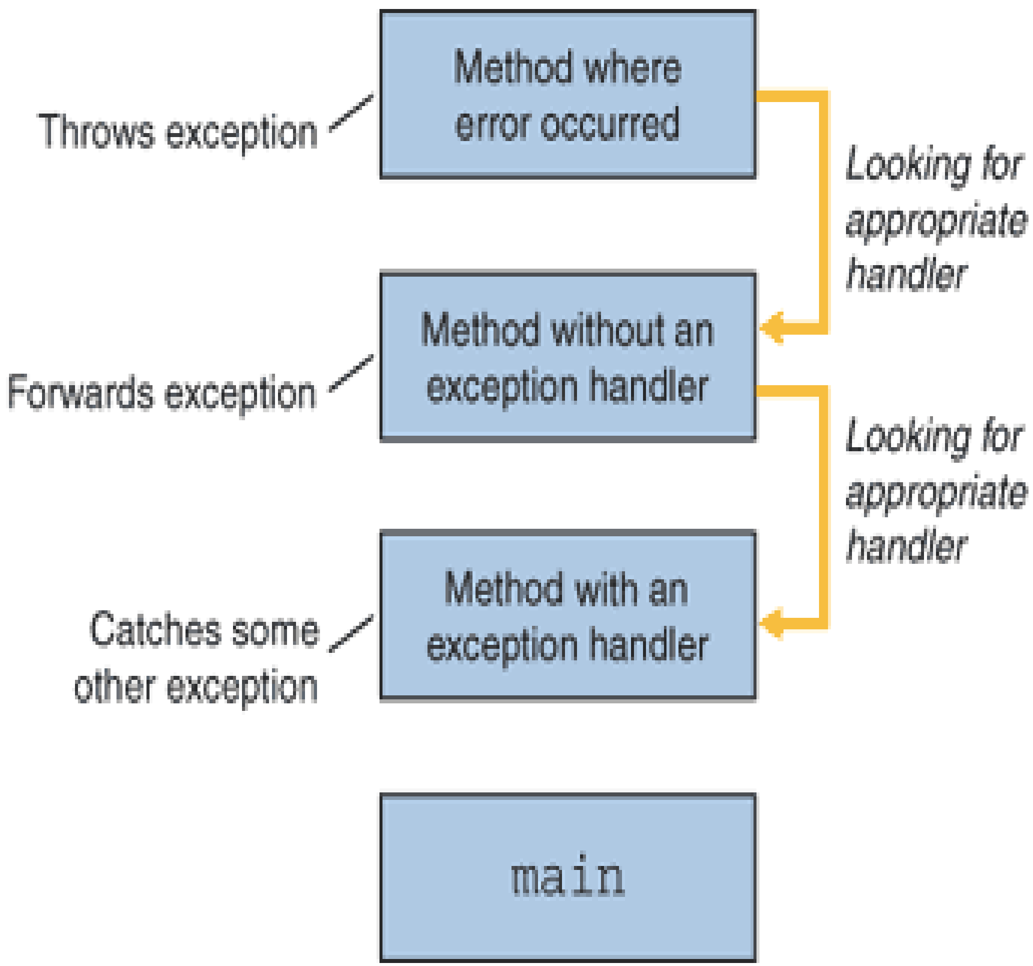
- Introduction
- **Working scheme**
- Types
- Creating new types of exceptions
- Exception handling
- Raising (throwing) exceptions

Execution stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

- Stack of pending calls in current thread
- The top method at the stack is where the exception was thrown
- The stack trace shows the code line number of each pending call

Handling/Capturing an exception


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

- There may be different handlers (*catch* clauses) for different types of exceptions (exception handlers)
- The search for the proper handler starts from the code nearest to the error condition
- The search continues by going deeper in the execution stack

Requirement: catch or throws

- In Java, a method must capture (*catch*) all exceptions that can be raised (thrown) in any of its calls
- Or alternatively, the method must declare that it also can raise the exceptions (*throws*)
- Otherwise, compilation will not succeed
- This does not apply to: ***Runtime Exceptions*** (e.g., division by zero)

Exceptions: Table of contents

- Introduction
- Working scheme
- **Types**
- Creating new types of exceptions
- Exception handling
- Raising (throwing) exceptions

Taxonomy

Origin/Checked	**Unchecked Exceptions**	**Checked Exceptions**
External Origin	External Errors	The source of the error is external (e.g., user, network, permissions, etc.)
Internal Origin	Internal Errors (Bugs/ Runtime Exceptions)	Make no sense as exceptions (condition with if)

Checked Exceptions

- *Checked* Exceptions
  - The most widely used, and the only ones requiring *catch* or *throws*
  - They represent exceptional events that the calling program should foresee and control
    - for example: *FileNotFoundException*
  - They are subclasses of ***Exception***
    - *RuntimeException* and its subclasses are *not checked exceptions* although they are subclases of *Exception.*

Unchecked exceptions

- They do not require *catch* or *throws*
- Exceptional events, but typically not foreseeable and hard to recover from
- Subtypes:
  - External errors:
    - Subclasses of *Error*
    - Example: *IOError* when a hard disk reading operation fails
  - Internal errors or RuntimeExceptions
    - Subclasses of *RuntimeException*
    - Example: *NullPointerException*

Hierarchy of exceptions and errors

- They all inherit from *Throwable* (a direct subclass of Object)
- The hierarchy helps to organize a variety of situations
- Class ***Error***
  - Typically not captured; they are severe errors:

    StackOverflowError, OutOfMemoryError, UnknownError, hardware failure,

    JVM intenal errors, errors program loading and initialization, …
- Class ***Exception***
  - being separated from errors, allows us to capture and treat them globally:

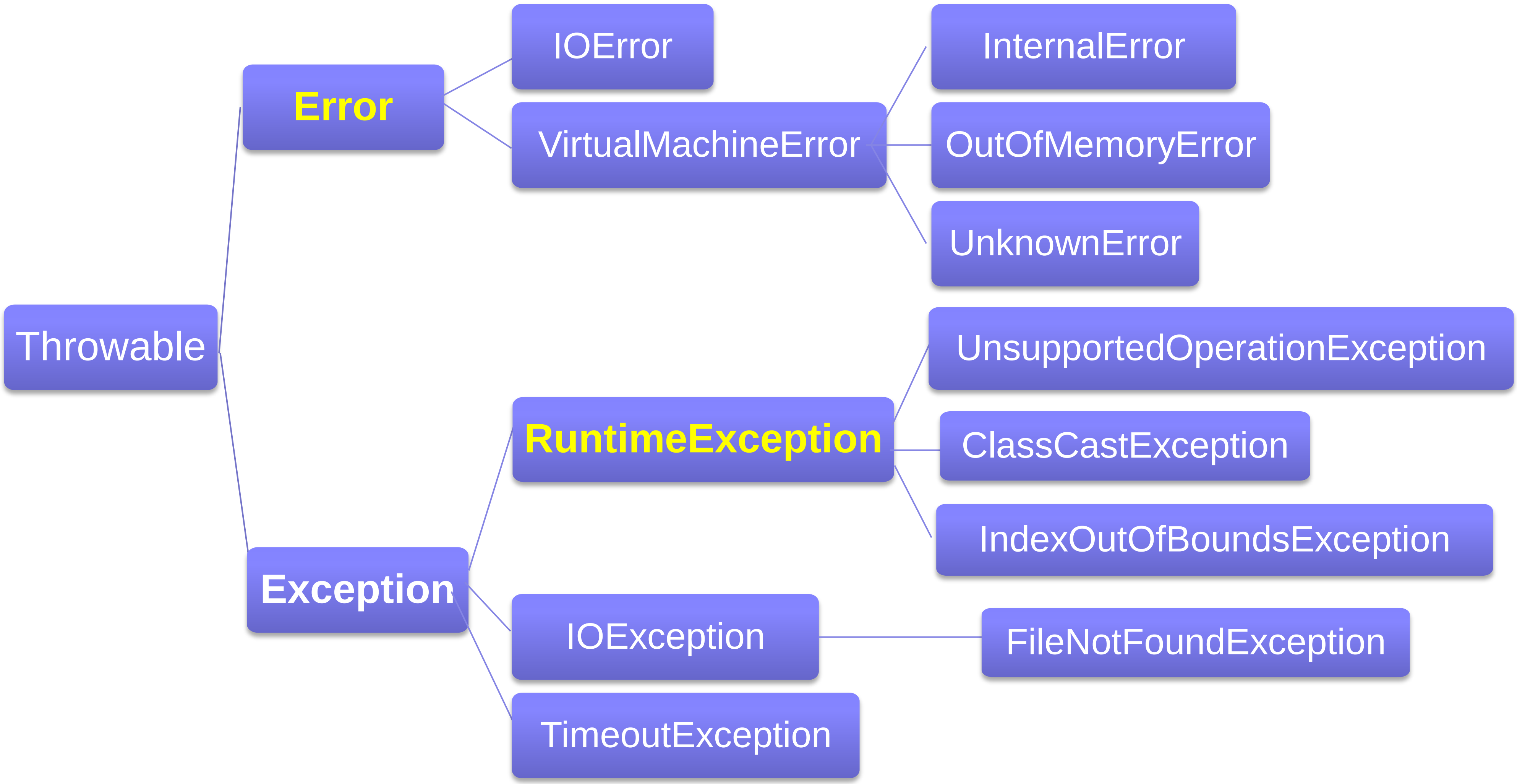
    `try {…} catch (Exception e) { /*` *handle any Exception* **`*/ }`**
- Class ***RuntimeException*** (direct subclass of Exception)

  Not a subclass of Error, since their origin is *not* external:

  Programming faults, typically not captured, but debugged when they happen

  They are special cases of Exception: **they do not require `catch` nor `throws`**

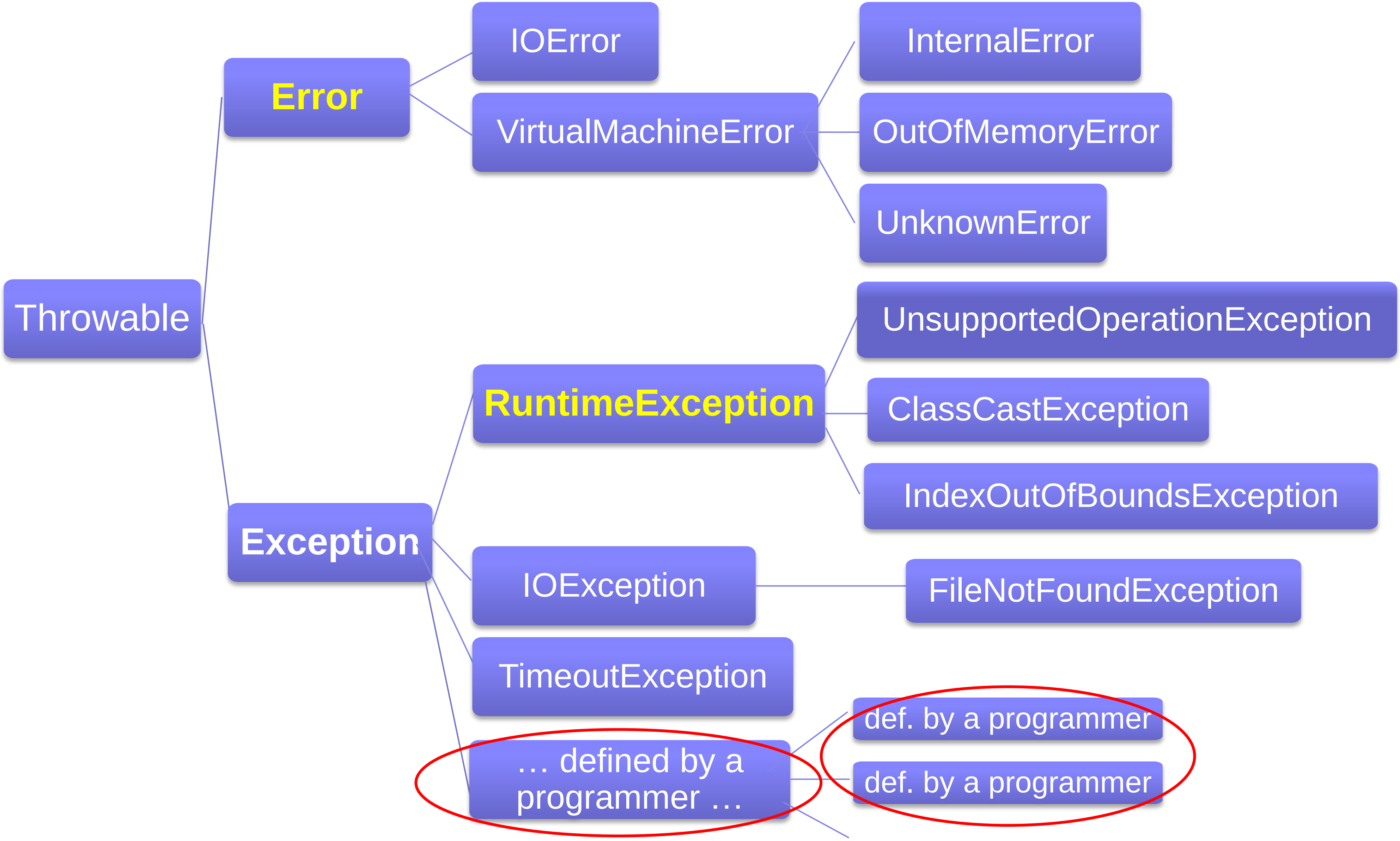
Hierarchy of exceptions and errors



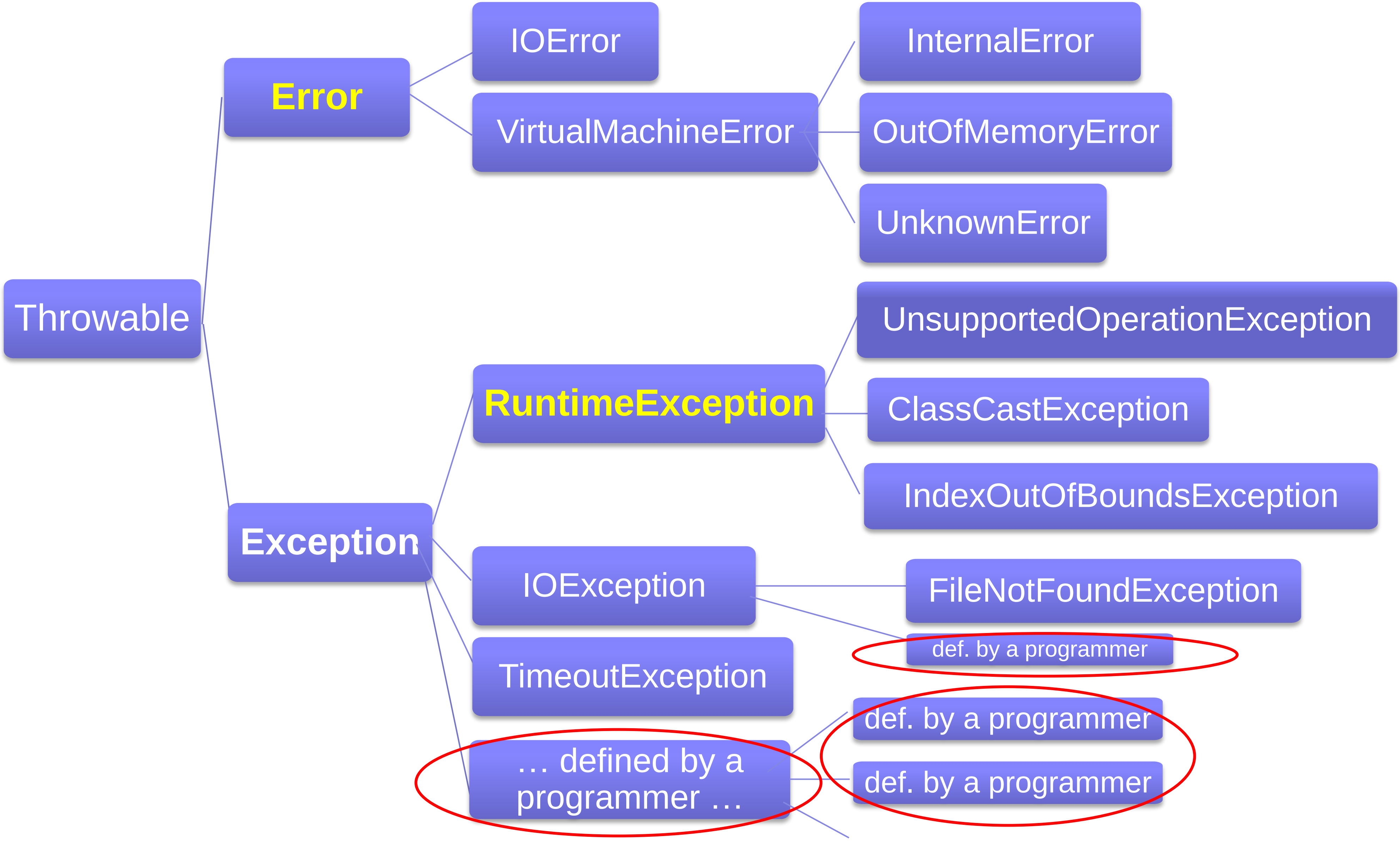
Exceptions: Table of contents

- Introduction
- Working scheme
- Types
- **Creating new types of exceptions**
- Exception handling
- Raising (throwing) exceptions

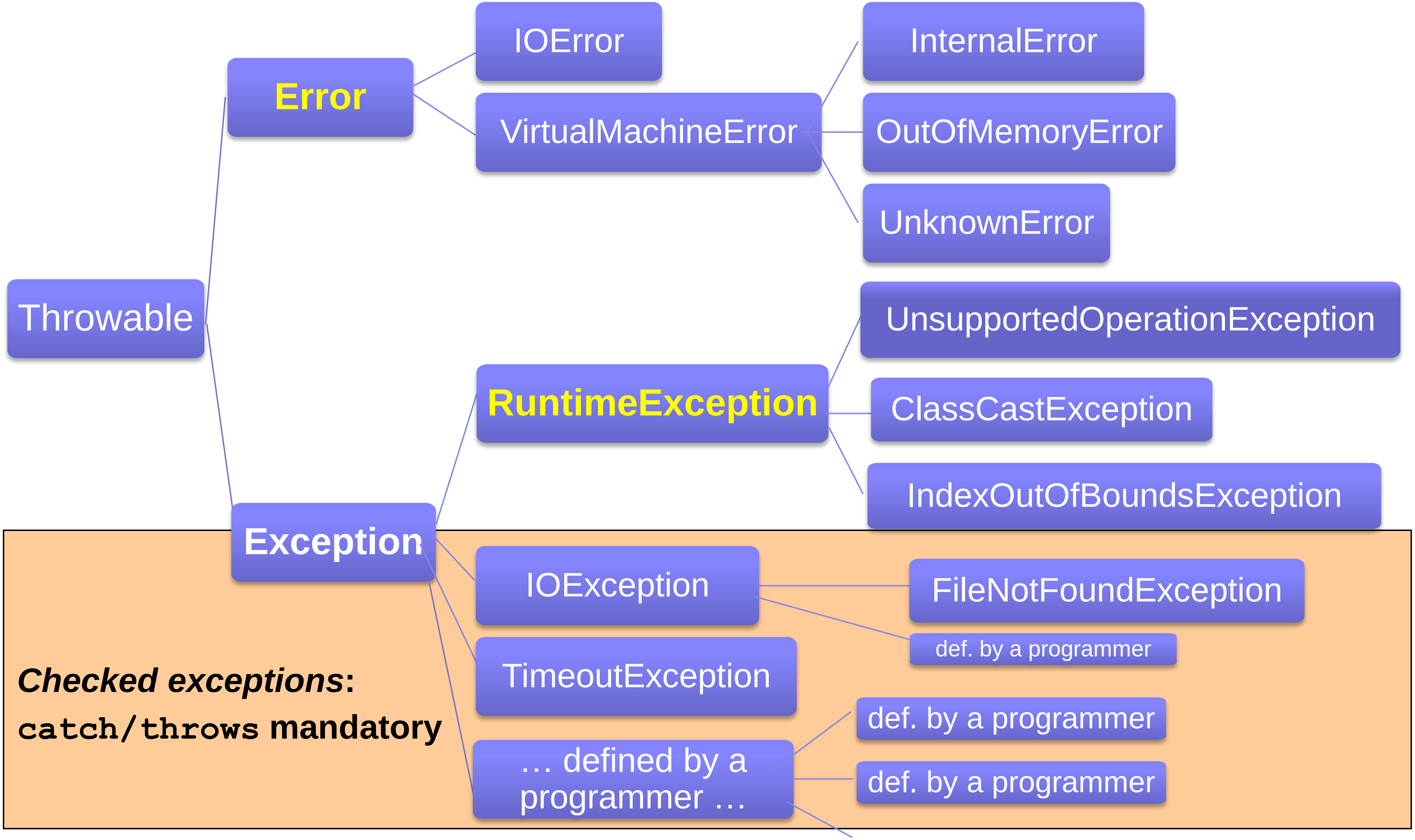
Extending exceptions and errors



Extending exceptions and errors



Extending exceptions and errors



Throwable

Error

IOError

VirtualMachineError

InternalError

OutOfMemoryError

UnknownError

Exception

RuntimeException

UnsupportedOperationException

ClassCastException

IndexOutOfBoundsException

IOException

FileNotFoundException

def. by a programmer

TimeoutException

… defined by a programmer …

def. by a programmer

def. by a programmer

Checked exceptions:

`catch/throws` **mandatory**

Creating new exceptions

- Basic rule:
  - Can we *reasonably* expect that a *client program* will be able to recover from the exception?
    - Yes → Checked exception
    - No → Unchecked (Error or RuntimeException)
  - **Do not** use subclasses of RuntimeException simply to avoid the requierement that your methods declare a `throws` for the exceptions they do not capture and handle.

Exceptions: Table of contents

- Introduction
- Working scheme
- Types
- Creating new types of exceptions
- **Exception handling**
- Raising (throwing) exceptions

Example: Exception Handling

```
File file = new File("data.txt");
int ch;
StringBuffer strContent = new StringBuffer("");
FileReader fin = new FileReader(file);
while ((ch = fin.read()) != -1)
  strContent.append((char)ch);
fin.close();
```

Exception handling:

- The main code changes little or nothing
  - We avoid complicating it with details related to checking exceptional cases
- The code for treating errors and exceptions is located at the end of each block in which they may happen

Example: Exception Handling

```
File file = new File("data.txt");
int ch;
StringBuffer strContent = new StringBuffer("");
try {
  FileReader fin = new FileReader(file);
  while ((ch = fin.read()) != -1)
    strContent.append((char)ch);
  fin.close();
} …
```

Possible exceptions

- Specific ones
  - When accessing “data.txt”
- General ones
  - Reading any data

Example: Exception Handling

```
File file = new File("data.txt");
int ch;
StringBuffer strContent = new StringBuffer("");
try {
  FileReader fin = new FileReader(file);
  while ((ch = fin.read()) != -1)
    strContent.append((char)ch);
  fin.close();
}
catch (FileNotFoundException e)
{
  System.err.println("File " + file.getAbsolutePath() +" could not be found.");
  throw new MyException();  // throw our own exception
}
```

Example: Exception Handling

```
File file = new File("data.txt");
int ch;
StringBuffer strContent = new StringBuffer("");
try {
  FileReader fin = new FileReader(file);
  while ((ch = fin.read()) != -1)
    strContent.append((char)ch);
  fin.close();
}
catch (FileNotFoundException e)
{
  System.err.println("File " + file.getAbsolutePath() +" could not be found.");
  throw new MyException();  // throw our own exception
}
// the compiler still signals an error:
// unhandled exception type IOException
```

Example: Exception Handling

```
File file = new File("data.txt");
int ch;
StringBuffer strContent = new StringBuffer("");
try {
  FileReader fin = new FileReader(file);
  while ( (ch = fin.read()) != -1)
    strContent.append((char)ch);
  fin.close();
}
catch (FileNotFoundException e) {
  System.err.println("File " + file.getAbsolutePath() +" could not be found.");
  throw new MyException();  // throw our own exception
}
catch (IOException e) {
  System.out.println("Exception while reading the file" + e.getMessage());
}
```

Proper ordering of `catch` clauses

- Exceptions will be captured following the order of the *catch* clauses. Hence:
  - □ Include first the more specific exceptions (those that are not subclasses of the previous ones)
  - □ Then the more general exceptions (some of which may be subclases of those in a previous catch)
  - □ The compiler verifies this and signal ordering errors (i.e., a general exception followed by a more specific one makes the latter handling code unreachable)

- In the previous example:
  - □ *FileNotFoundException* (more specific case) is handled by throwing a new exception from our own exception class using *throw* (without s, no the same as *throws*)
  - □ *IOException* stands for a more general case

Improvemts in `catch` since JDK 7

- We can combine the handler of several types of exceptions in a single `catch`

```
catch (FileNotFoundException | SecurityException ex) {
  logger.log(ex);
  throw new MiExcepcion(ex);
  // Note SecurityException is not mandatory in
  // PrintWriter(File f) since it´s unchecked exception
}
```

Post-exception treatment and finalization

```
private List<Integer> aList;
PrintWriter output = null;
...
try {
      System.out.println("Start Try");
      output = new PrintWriter(new FileWriter("d/out.txt"));
      for (int i=0; i<=aList.size(); i++) {
        output.println("aList[" + i + "] = " + aList.get(i));
      }
} catch (FileNotFoundException e) {
    ...
} catch (IOException e) {
    ...
} finally {
  if (output!=null) output.close();
}
```

The block `finally` *always* executes, regardless of whether or not any exception actually happened.

It is usefull to deallocate resources and to end the process properly.

Block `finally`

- The block *finally* executes after exiting from the block try
  - In the example, the variable *output* is declared outside the *try* block so that it can be accessed from the *finally* block

- In the *try* block we could have *return*, *break*, *continue*, or even other exceptions nor foreseen ...

- The *finally* block is always executed
  - We can also use a *try* with *finally* even if no exceptions are expected.

- **Note**: The *finally* block could produce new exceptions, and so in that case it may not execute completely

Example 2: copying a file (without `throws`)

```
void copyWithRisk() {
     InputStream in = null;
     OutputStream out = null;
     ...
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     ...
          in.close();
          out.close();
     }
  }
```

Their declaration in `java.io` *indicates that they may throw* `IOException`

It does not compile unless we declare, using **`throws`**, that copyWithRisk may produce exceptions of type **`IOException`**, or catch the **`IOExceptions`** inside the method

Example 2: copying a file (without `try`)

```
void copyWithRisk() throws IOException {
     InputStream in = null;
     OutputStream out = null;
     ...
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     ...
          in.close();
          out.close();
     }
  }
```

Now it compiles without errors, but it is not a good idea to ignore all exceptions and internal errors and simply passing them up by declaring them in **`throws`**

Example 2: copying a file (try without catch)

```
void copyWithRisk() throws IOException {
     InputStream in = null;
     OutputStream out = null;
     try {
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     } finally {
          in.close();
          out.close();
     }
  }
```

Though this compiles, it is not a good solution to use a **try/finally** with no **catch** clauses.

It prevents reading/writing when a file did not open correctly, but if a file did not open and we try to close it, there will be a **NullPointerException**

Example 2: copying a file (`try` with `catch`)

```
void copyWithRisk() throws IOException {
     InputStream in = null;
     OutputStream out = null;
     try {
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     } catch (IOException e) {
          // handle exception e
     } finally {
      if (in != null) in.close();
      if (out != null) out.close();
     }
  }
```

The **`catch`** clauses are needed to to handle the exceptions before the execution jumps to **`finally`**

We also avoid a **`NullPointerException`** when closing the file

Example 2: copying a file (needs a throws)

```
void copyWithRisk() throws IOException {
     InputStream in = null;
     OutputStream out = null;
     try {
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     } catch (IOException e) {
          // handle exception e
     } finally {
      if (in != null) in.close();
      if (out != null) out.close();
     }
  }
```

In spite of having **catch** clauses

we still need a **throws**

Because the calls to **close** inside the **finally** may also produce **IOException**

Example 2: copying a file (`try` within `finally`)

```
...
} finally {
     if (in != null) {
       try {
             in.close();
       } catch (IOException e) {
          // handle the exception that close may throw

       }
     // we could do the same for out.close
} // end of first try/finally
```

An exception thrown by **`close`** means a severe input/output error, so there is little we can do to recover from it, just inform about the error. But at least we no longer need to declare that our method **`throws IOException`**

(It seems too much exception handling for such a simple program, but our purpose here was to learn all we can do and when we should do what).

Example 2: copying a file (needs no `throws`)

```
void copyWithoutRisk() {
     InputStream in = null;
     OutputStream out = null;
     try {
          in = new FileInputStream("in.txt");
          out = new FileOutputStream("out.txt");
          byte[] buffer = new byte[1024];
          int n;
          while ((n = in.read(buffer)) >= 0) {
             out.write(buffer, 0, n);
          }
     } catch (IOException e) {
          // handle the exception e
     } finally {
         closeIgnoringExceptions(in);
         closeIgnoringExceptions(out);
     }
  }
```

Example 2: copying a file (`close` without `throws`)

```
void closeIgnoringExceptions(Closeable c) {
  if (c != null) {
        try {
             c.close();
        } catch (IOException e) {
           // handle the exception that close may throw
        }
  }
}
```

Interface **`Closeable`** requires the methdos:

```
    void close() throws IOException
```

`FileInputStream` and **`FileOutputStream`** implement **`Closeable`**

Improved `try` since JDK 7

- When dealing with resources, an improved syntax can be used to ensure that resources are automatically closed
- It avoids using `finally` when its only purpose is closing resources, and resource variables do not need to be declared outside `try` just to initialise them to `null`
- The resource is required to implement the `AutoCloseable` interface
- The exceptions supressed within a try-with-resources can be obtained with `Throwable.getSuppresed()`.

```
try (PrintWriter output =
        new PrintWriter(new FileWriter("d/out.txt"))){
...
}
catch (...){
...
}
```

Exceptions: Table of contents

- Introduction
- Working scheme
- Types
- Creating new types of exceptions
- Exception handling
- **Raising (throwing) exceptions**

Declaring which exception may be thown

- The clause **`throws`** is used when declaring a method or a constructor, just before its implementation block (if present, i.e., not abstract)

```
public void writeList()
              throws IOException, ArrayIndexOutOfBoundsException {
}
```

- It may also be used in interface methods and abstract methods
- Unchecked exceptions do not need to be included in **`throws`**

```
public void escribeLista() throws IOException{
}
```

We keep only *IOException*, since *ArrayIndexOutOfBoundsExceptio*n is an unchecked exception

Throwing exceptions

- It is done with `throw objectException`
- `objectExcepcion` created with `new ExceptionClass()`

  (constructors with parameters can be used)
- We often use subclasses of `Exception`
- Subclasses of `Error` are not typically captured or thrown (severe errors within the virtual machine, etc.)

```
public Object pop() {
    Object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    --size;
    return obj;
}
```

Example implementing pop()
with an exception thrown

Chaining exceptions together

- It happens when an exception handler creates a new exception and throws it (including the former as cause of the latter)
- **`Throwable.getCause()`** returns the cause
- **`Throwable.initCause(Throwable)`** sets an exception cause (but it can only be set once)
- More often, the new exception is created with a constructor like:

```
Throwable(String mensaje, Throwable causa)
```

The **`cause`** of **`MyException`** is …

```
try {
     /* load a database */
} catch (IOException e) {
   throw new MyException("Error opening database", e);
}
```

Dealing with details of exceptions

- Print the execution trace

```
catch(Exception e){
  e.printStackTrace();}
}
```

- Get access to the pending calls in the stack

```
StackTraceElement[] stackElemets = e.getStackTrace();
for (StackTraceElement se : stackElemets ) {
   System.err.println(se.getFileName()+": "
                     +se.getLineNumber()+"\t"
                     +se.getMethodName+"()")
}
```

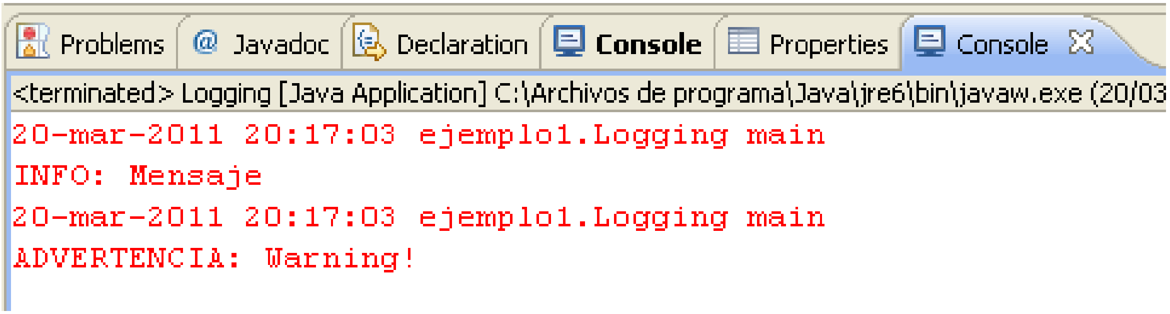
Logging Service

- Used to save errors or debugging messages in a file, using the application configuration
- We can have a global logger of the system, one per application, or for a specific package
- Hierarchical system, names separated by dots
  - Ej: `Logger logger= Logger.getLogger(“java.net”);`
- logger.log(Level, message [, objet o associated exception])
  - Level (from more to less severe): `SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST`
  - Ej: `logger.log(Level.WARNING, “Time out, retrying...”, exception);`

Logging Service

```java
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Logging {
  public static void main(String[] arg) throws IOException {
    Logger log = Logger.getLogger(BankAccount.class.getName());
    log.addHandler(new FileHandler("out.xml"));
    log.setLevel(Level.INFO);
    log.info("Mensaje");
    log.warning("Warning!");
  }
}
```



```
<terminated> Logging [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.exe (20/03
20-mar-2011 20:17:03 ejemplo1.Logging main
INFO: Mensaje
20-mar-2011 20:17:03 ejemplo1.Logging main
ADVERTENCIA: Warning!
```

Logging Service

File out.xml

```
<?xml version="1.0" encoding="windows-1252"
standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2011-03-20T20:17:03</date>
  <millis>1300648623093</millis>
  <sequence>0</sequence>
  <logger>ejemplo1.BankAccount</logger>
  <level>INFO</level>
  <class>ejemplo1.Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Mensaje</message>
</record>
<record>
  <date>2011-03-20T20:17:03</date>
  <millis>1300648623140</millis>
  <sequence>1</sequence>
  <logger>ejemplo1.BankAccount</logger>
  <level>WARNING</level>
  <class>ejemplo1.Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Warning!</message>
</record>
</log>
```

When not to use exceptions?

- They are no substitute for *normal conditional statements*
- Example: *lazy* initialization of a list

```
List<Element> aList; // without initialization
Element x;
try {
  aList.add(x);
}
catch (Exception e){
  aList= new ArrayList<Element>();
  aList.add(x);
}
```

```
// Better with an if
if (aList == null) { aList = new ArrayList<Element>(); }
aList.add(x);
```

Advantages of using exceptions

- Separation of normal execution flow from error handling code
- Facilitate the error flow through thestack of pending calls

  → They are handled where they can be controlled better
- They allow us to group errors by their type and category
- Each method must declare which exceptions it may throw (only checked exceptions)

Exercise

- Using exceptions, add error control to the exercise of the directories:
  - Detect if null is pass to method add
  - Detect a cyclic dependency in method add