COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

8. Exception/event handling

Aims

…

Objectives

At the end of this lecture you will understand how to use exceptions in Java. Also we consider issues involving concurrency and threads.

8. Exception/event handling

Outline

- Errors and exceptions

Readings

…

Runtime problems

What types of error can occur when a Java program is running?

User input errors Typos, unexpected key sequence, syntax errors (eg invalid URL).

Device errors Network failure, read file error or write file error.

Exceeded limits Out of memory.

Programmer error Invalid array access, null reference, division by 0.

Exceptions vs. errors

- Java distinguishes between ***checked*** exceptions and ***unchecked*** ones (ie, the handling of them is *checked* or *unchecked* at compile time)

- ***Checked*** Unusual situation which should be taken into account when developing a program (e.g. an exception is thrown if wrong URL is used).

- ***Unchecked*** condition which probably should be reported but is not worth explicitly handling (e.g. array out of bounds).

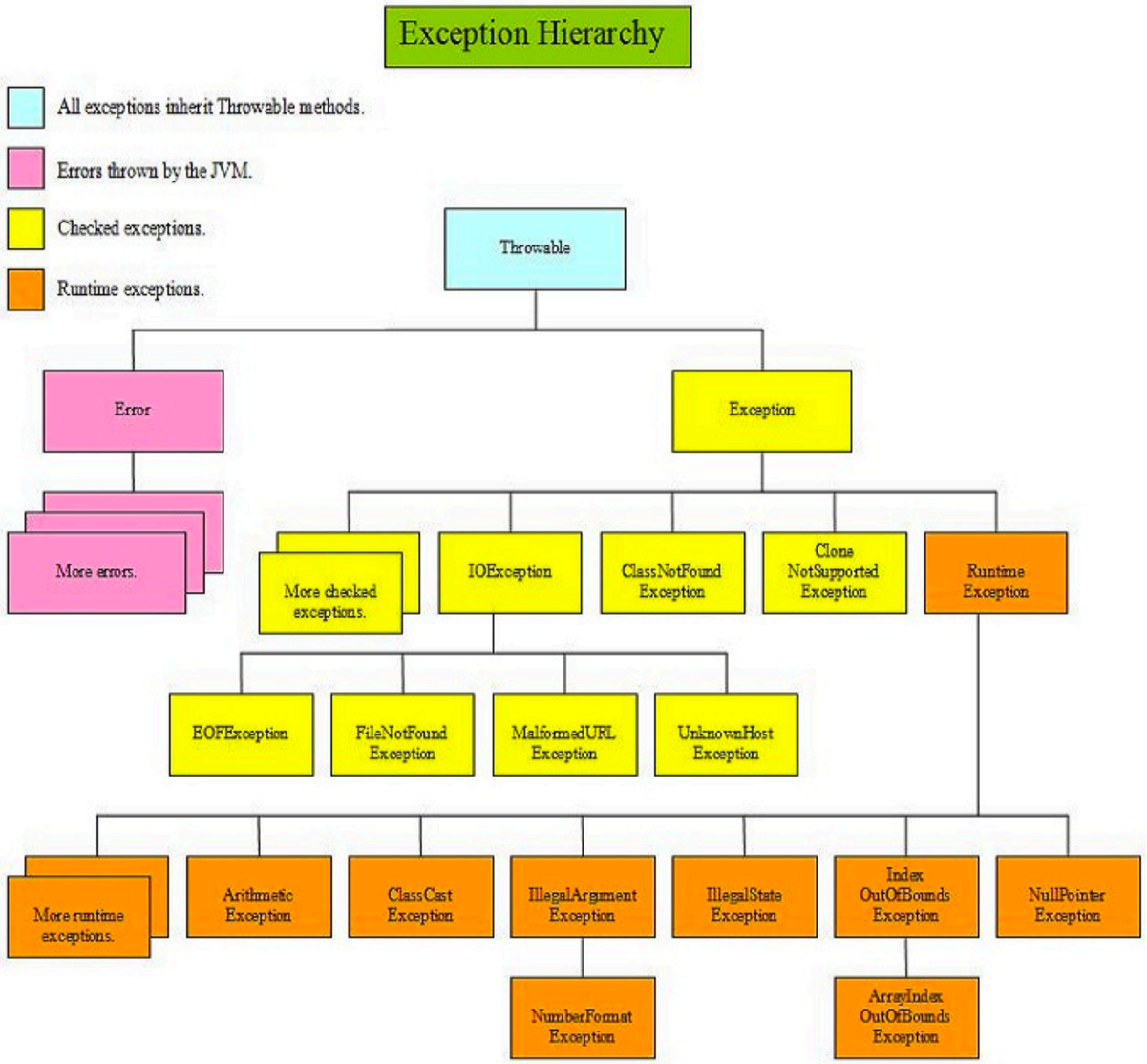
```
int[] myArray = new int[14];
for (int i=0; i<14; i++ ) {
     if (myArray[i+1] >1500 & myArray [i] >1500) {
        …
```

Exceptions

- The *author* of a library class can detect run-time errors but (in general) does not know what to do about them.
- The *user* of a library may know how to cope with such errors but cannot detect them (they may only be detected by library code) or else they would have been handled in the user's code.
- The notion of an *exception* is to help deal with such problems: if a method cannot deal with an error it ***throw****s* an exception hoping that its caller (direct or indirect) can deal with it.
- A function that is willing to handle the problem can ***catch*** the exception.

Java exception hierarchy

- Exception objects are instances of the Throwable class.
- Error: Internal errors, exceeded limits, thrown by system (not by programmer). You can't recover from these, just exit (as) gracefully (as possible).
- Exception: This is the hierarchy on which we focus



Java exception hierarchy

- **RuntimeException –** are not anticipated by the compiler**.** Includes things like null pointer access, bad array access, and bad cast. Programming errors generally create these.

- **Unchecked Exceptions** Error and RuntimeException (& of course their subclasses) are *unchecked*. The rest are *checked exceptions*.

- **Checked Exceptions** include IOException and FileNotFoundException – if your method may throw them, you have to declare them using the `throws` keyword. The compiler checks that they will be handled correctly.

Exception objects

The exception object contains:

* name
* description of the exception, and
* current state of the program where exception has occurred.

Creating the Exception Object and passing it to the run-time system is called ***throwing an Exception***.

Throwing exceptions

- Methods that can throw a checked exception must advertise the fact, e.g.:

```
// constructor for java.net.URI
public URI(String str) throws URISyntaxException
```

- If something goes wrong with the constructor (due to a bad argument) a URISyntaxException object is *thrown.*

- A method must declare a **throws** clause if:
  - it calls a method that throws a checked exception which is not caught, or
  - it explicitly throws a checked exception.

- Unchecked exceptions do not need to be advertised.

- The compiler will let you know if you forget to advertise a checked exception.

Throwing exceptions

- An exception is an object of a class in the Exception hierarchy
- You can throw it by choosing the appropriate Exception class, making an object of that class, and throwing the object.

```
import javax.sound.sampled.*;

public void openAudioFile(String str)
    throws UnsupportedAudioFileException {
        boolean fileOK;

            . . .
        if(! fileOK) {
            throw new UnsupportedAudioFileException("file not OK");
        }
        // otherwise carry on
        // . . .
```

Catching exceptions

- Catching an exception requires a bit more code
- Use a try/catch block:

```
try {
  // code that may throw an exception
}
catch (ExceptionType e) {
  // exception handling code
}
finally {
  // close any open files or similar resources
}
```

- If any code in the `try` block throws an exception of the class (or subclass) specified in the catch, then:
  - The program skips the remainder of the try block.
  - The handler code in the catch block is executed, and can refer to the exception object by its variable.
  - The `finally` block is always executed last, can be used to free resources.

Guidelines

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and run-time exceptions to indicate programming errors
- Document all exceptions thrown by each method
- Include useful debugging information in detail messages
- Never do these:

```
try {
  while(true) // use an exception to exit loop!
    f(a[++i]);
}
catch(ArrayIndexOutOfBoundsException e)
```

```
try {
  //. . .
}
catch(SomeException)
  // Empty catch block. . . very suspicious!
```

Catching exceptions

- The handling code may simply inform the user of the error, after which code execution proceeds e.g.

```
// catch
String numStr = JOptionPane.showInputDialog("Enter a number");
int number = 0;
try {
    number = Integer.parseInt(numStr);
}
catch (NumberFormatException e){
    System.out.println("Invalid number");
}
```

- The handler code may however try to fix the problem causing the exception. For example, if the exception is a FileNotFoundException then the handler code may ask the user to check the spelling of the file name.
- If an exception is thrown and not caught, then the program will terminate with a message to the console.

Example

```
public class TestException  {
   public static void main(String[] args) {
     String[] p = new String[4];
     try {
       p[0] = "John Lennon";
       p[2] = "Paul McCartney";
       p[4] = "George Martin";
       p[1] = "Ringo Starr";
       p[3] = "George Harrison";
     }
     catch (ArrayIndexOutOfBoundsException e) {
       System.out.println("There were only  four Beatles");
     }
     for (int i=0; i<p.length; i++) {
       System.out.println(p[i]);
      }
   }
}
```

```
There were only four
Beatles
John Lennon
null
Paul McCartney
null
```

Catching multiple exceptions

```
import java.util.Random;

public class TestException2 {
    public static void main(String[] args) {
        String[] p = new String[4];

        try {
           Random r = new Random();
           int divisor = r.nextInt(2);
           double x = 1/divisor;
           p[0] = "John Lennon";
           p[2] = "Paul McCartney";
           p[4] = "George Martin";
           p[1] = "Ringo Starr";
           p[3] = "George Harrison";
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("There were only four Beatles");
        }
       catch (ArithmeticException e) {
            System.out.println("Divide by zero");
       }
       for (int i=0; i<p.length; i++) {
           System.out.println(p[i]);
       }
  }
}
```

This randomly produces either 0 or 1 (with equal probability).

When zero, an ArithmeticException is thrown when we attempt the division. When one, we get the array index exception as before.

Output from this program is either:
There were only four Beatles
or
Divide by zero

Catching multiple exceptions

- The try/catch syntax provides a tidy way of handling more than one exception.
- We never throw both exceptions in a single run, since flow of control is thrown to catch block and then to the end of the entire try/catch block.
- We could catch both exceptions in a single statement since ArithmeticException and ArrayIndexOutOfBoundsException are both subclasses of Exception:

```
catch (Exception e) {
  // do something
}
```

- This may on occasion be appropriate (e.g., if the exceptional behaviour will be handled by exiting the program) but is generally considered to be lazy programming.

Using the Exception instance

- We have access to the Exception instance (object) and can throw it again (to some outer block of exception-handling code), and then print diagnostic information.
- This can be done using the exception's **`printStackTrace`** method:

```
catch (ArithmeticException e) {
    System.out.println("Divide by zero");
    e.printStackTrace();
}
```

- This gives useful information (particularly if the exception occurs in deeply nested code), e.g.

```
Divide by zero
java.lang.ArithmeticException: / by zero
at TestException2.main(TestException2.java:9)
But lets continue anyway
```

Exception ‘bubbling up’ (throwing up?)

- If we don’t provide an appropriate handler (or omit the try/catch block altogether) the exception is thrown to the method that invoked the method in which it occurred.
- If the invoking method has an appropriate handler, the exception is handled at that point otherwise the process repeats until the top-level (i.e., main) method is reached.

```
public class TestException3 {
    public static void main(String[] args) {
     try { method1(); }
        catch (ArithmeticException e) {
           System.out.println(“divide by zero”);
           e.printStackTrace();
        }
     }
     public static void method1() { double index = 5/0; }
}
```

- Produces

```
divide by zero
java.lang.ArithmeticException: / by zero
at TestException3.method1(TestException3.java:11)
at TestException3.main(TestException3.java:4)
```

Finally

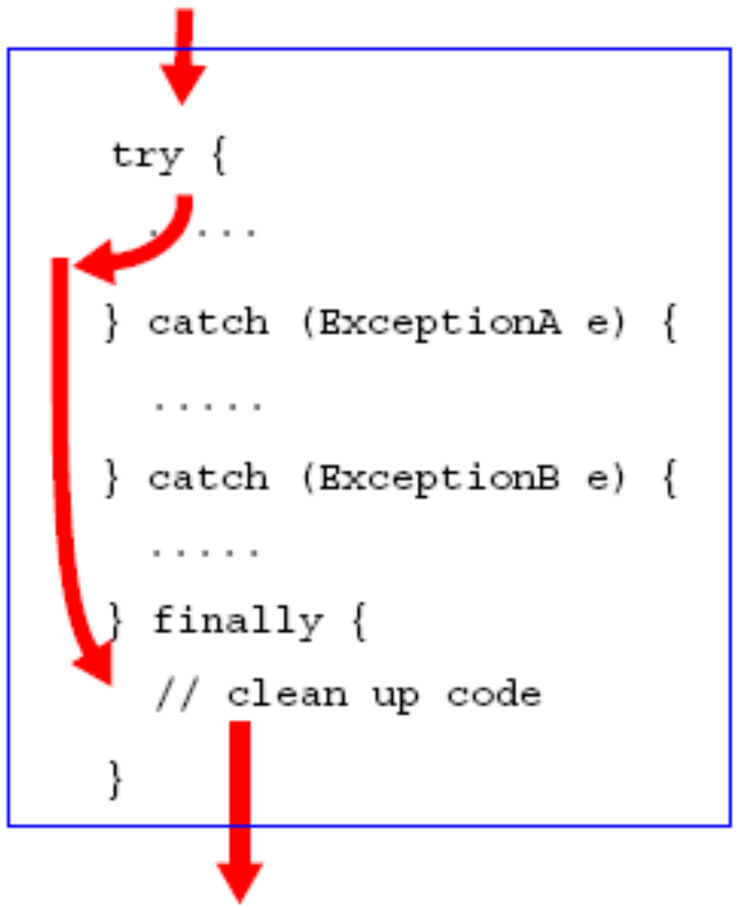
- We often need to ‘clean up’ after throwing an exception, e.g., to close files, database/network connections, other resources.
- Example: display “File not found. Select another one? (Y/N)”
- To avoid duplicating such code in different catch blocks, Java provides an optional ***finally*** block which is executed whether or not an exception is thrown and caught.

```
try {
  // as before
}
catch (AgeException e) {
  // as before
}
finally {
  System.out.println(”try-catch block completed");
}
```

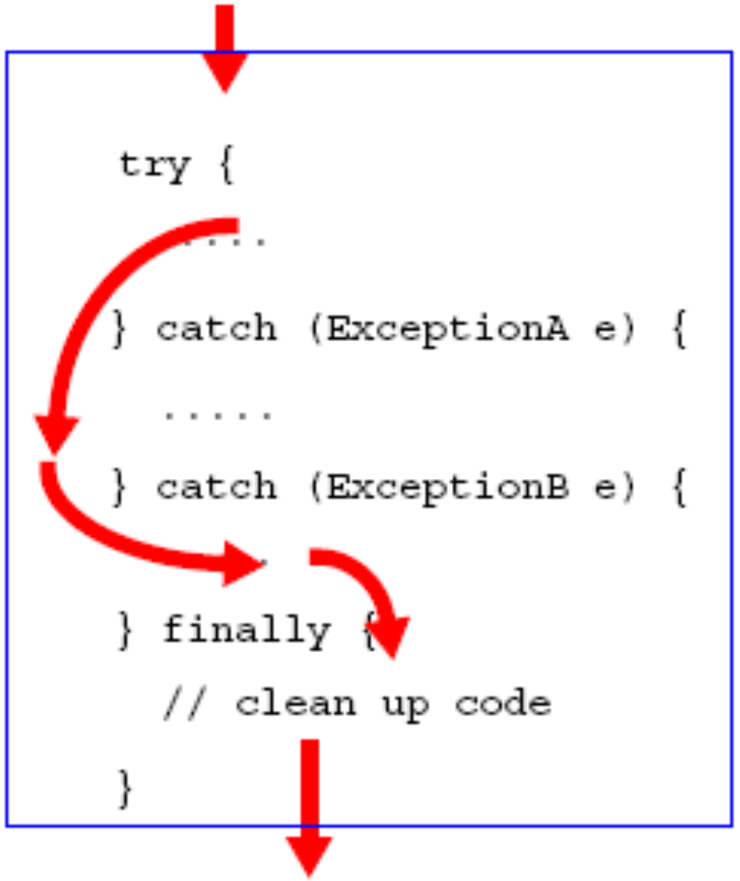
Finally

```
try {
  // as before
}
catch (ArrayIndexOutOfBoundsException e) {
  // as before
}
finally {
  System.out.println(”try-catch block completed");
}
```

Flow of control for `try/catch/finally`

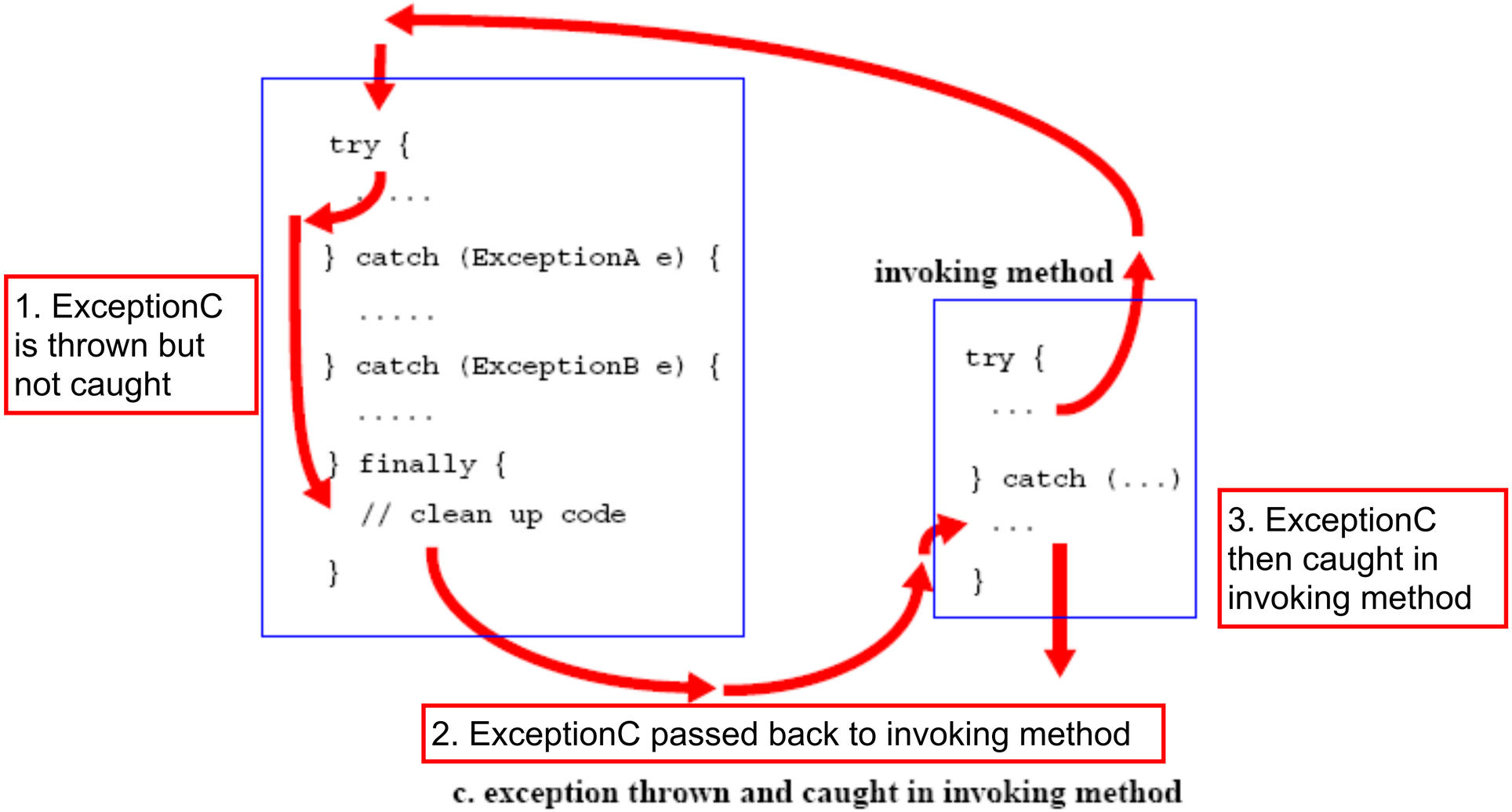


a. no exception thrown



b. exception thrown and caught locally

Flow of control for `try/catch/finally`



c. exception thrown and caught in invoking method

Example: EasyReader

The `sheffield.EasyReader` class was designed for basic use in the first week of the course, but your code calling it can't tell a "real" 0 from an error:

```
public int readInt() {
  int x = 0;
  try {
    x = (Integer(readString())).intValue();
  }
   catch (Exception e) {
     error("invalid integer number");
  }
  return x;
}
```

Example: EasyReader

The sheffield.EasyReader class could be improved for more advanced use by forwarding the exception to the calling code:

```
public int readInt() throws IOException {
  // The improved version of readString()
  // can throw IOException, all the way from
  // BufferedReader.read()

  // NumberFormatException is unchecked and can be
  // passed on to the calling code automatically

  return (Integer(readString())).intValue();
}
```

Example: EasyReader

Depending on how you want to use it, `sheffield.EasyReader` could be improved further by detecting unparseable strings and converting to a checked exception:

```
public int readInt() throws IOException {
  // The improved version of readString()
  // can throw IOException, all the way from
  // BufferedReader.read()

  // NumberFormatException is unchecked and can be
  // passed on to the calling code automatically

  try {
    String s = readString();
    int x = (Integer(s)).intValue();
  }
  catch (NumberFormatException e) {
    throw new IOException("Not a number " + s, e);
  }
  return x;
}
```

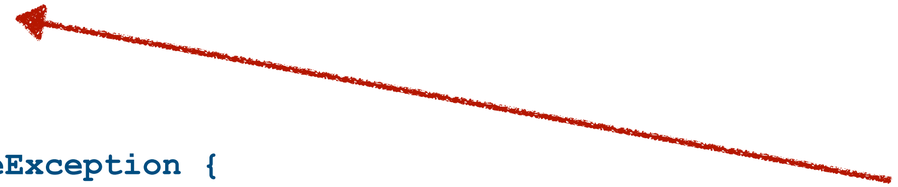
Example: A hierarchy of age exceptions

```
class AgeException extends Exception {
   public AgeException() {
          super();
   }
   public AgeException(String s) {
          super(s);
   }
}

class TooYoungException extends AgeException {
   public TooYoungException() {
          super();
   }
   public TooYoungException(String s) {
          super(s);
   }
}

class TooOldException extends AgeException {
   public TooOldException() {
          super();
   }
   public TooOldException(String s) {
          super(s);
   }
}
```

Default parameterless constructor



Constructor with String parameter, which will form an error message

Example: A hierarchy of age exceptions

```
try {
  Scanner keyboard = new Scanner(System.in);
  int age = keyboard.nextInt();
  if (age>70) {
    throw new TooOldException("Too old to rock and roll");
  }
  if (age<16)
       throw new TooYoungException("Too young");
  }
  catch (AgeException e) {
     e.printStackTrace();
}
```

Output for age of 2

```
TooYoungException: Too young
at TrivialApplication.main(AgeApplication.java)
```

Output for age of 90

```
TooOldException: Too old to rock and roll
at TrivialApplication.main(AgeApplication.java)
```