Exceptions and File IO

1DV506 - Lecture 9

Dr Jonas Lundberg

Jonas.Lundberg@lnu.se

December 17, 2019

Agenda

- Example: An integer list
- Exceptions
- Checked and Unchecked Exceptions
- File Input/Output

Reading Instructions: Chapter 12 (except 12.9, 12,12, 12.13)

A Growing Collection of Integers

```
IntList list = new IntList();

for (int i=1;i<=20;i++) // Add 20 integers 10,20,30, ... ,200
    list.add( i*10 );

System.out.println("Size: "+list.size());
System.out.println("Content: "+list.toString());
System.out.println("Integer at position 5: "+list.get(5));
System.out.println("Contains 100: "+list.contains(100));
```

Note:

- We don't assign any size to `IntList`.
- It will be able to handle any number of integers.
- It behaves like an `ArrayList<Integer>`
- Q: How to create a collection like this?
- A: Use an array and change to a bigger one if the first gets full.

IntList (Part 1)

```
public class IntList {
    private int size = 8;          // Current array size
    private int count = 0;         // Number of added element
    private int [] data;           // Integer storage

    public IntList () { data = new int[size]; }

    public int size () { return count; }
    public int get(int pos) { return data[pos]; }  // Index check!?

    public String toString () {
        StringBuilder buf = new StringBuilder() ;
        for (int i=0;i<count;i++)
            buf.append(data[i] + " ");
        return buf. toString ();
    }

    ...     // More methods on next slide
}
```

IntList (Part 2)

```
public class IntList {
    ...

    public void add(int n) {
        if (count == size) // time to grow?
            resize ();

        data[count] = n;    // store integer n
        count++;
    }

    /* Doubles the size of the array */
    private void resize () {
        int [] tmp = new int[2*size]; // increase array size
        for (int i=0;i<size;i++)
            tmp[i] = data[i];          // copy array content

        data = tmp;         // update data
        size = 2*size;      // and size
    }
}
```

`IntList` – Error Handling

Method `get(int pos)` does not work if index `pos` is out of range ⇒ less than zero or larger than `size-1`

```
public class IntList {
    private int count = 0;      // Number of added element
    private int [] data;        // Integer storage
    ...
    public int get(int pos) { return data[pos]; }   // Might Crash!
    ...
}
```

Alternative `get(int pos)` implementation

```
public int get(int pos) {
    if (pos < 0 || pos >= count) {
        System.err.println("List index "+pos+" out of range!");
        return -99;             // We must return something!
    }
    else
        return data[pos];       // OK!
}
```

Neither approach is satisfying. We can do better!

Exception Example

```
public static void main(String[] args) {

   int p = 7, q = 0;              // Zero denominator
   try {                          // Enter dangerous area
      int res = divide(p, q);
      System.out.println(p+"/"+q+" = "+res);
   }
   catch (ArithmeticException e) {        // Catches and handles error
      System.out.println("Exception: "+e.getMessage());
   }
}


private static int divide(int a, int b) {    // Computes a/b
   if (b == 0)
      throw new ArithmeticException("Dividing by Zero!");  // Throws exception
   else
      return a / b;
}
```

Exceptions: Basics

- Java handles all errors and abnormal conditions using *exceptions*.
- An exception is an object that encapsulates information about an error.
- Error ⇒ program *throws* (or *raises*) an exception.
  (e.g., `throw new ArithmeticException()`)
- ⇒ execution halts immediately
- ⇒ call stack is unwounded until an appropriate enclosing exception handler is found. (e.g., `catch(ArithmeticException exc) {...}`)
- No enclosing exception handler ⇒ JVM catches exception, abruptly terminates program, and prints a stack trace
- Advantages
  - Uniform handling of all abnormal conditions
  - Separation of responsibilities:
    The programmer identifies problems and raises exceptions.
    The client (or user) determines how to handle the problem
    (ignore and continue, recover, try again, exit, ...).

List with Improved Error Handling

```
public class IntList {
    private int count = 0;        // Number of added element
    private int [] data;          // Integer storage
    ...
    public int get(int pos) {
        if (pos < 0 || pos >= count)  // Index out of range
            throw new IndexOutOfRangeException("Index out of range: " + pos)
        else
            return data[pos];     // OK!
    }
}
```

- The Java documentation of the class library contains a lot of information about what `exceptions` are thrown by different methods. It is then possible for the user to deal with the error without knowing the source code.

An Unspoken Contract

Background

- The programmer can't know how a user wants to deal with an error.
- Different users and situations ⇒ different types of error handling.

An Unspoken Contract

- The programmer is responsible for identifying errors and to notify the user by raising an exception.
- The user/client decides how to handle the exception.

Example: The method `get(int pos)` in the class `IntList`

- The programmer finds the faulty index (outside the range) ⇒

```
throw new IndexOutOfRangeException("Index out of range: " + pos);
```

- The user of the list can (if he/she likes) catch and handle the error

```
try {
   get() gets called
} catch (IndexOutOfRangeException exc) {
   the error is handled
}
```

Another Example

```
int[] a = new int[5];
try {
   a[99] = 24;    // => out-of-range exception
}
catch (IndexOutOfRangeException e) { // Handle this type of error
   System.out.println(e.getMessage());
}
catch (Exception e) {  // Handles any kind of error
   System.out.println(e.getMessage());
}
finally { // Always executed
    /* Save what is possible. Close database connections,
       networks and so on */
}
```

- Repeated `catch` ⇒ the first suitable is used.
- `Exception` is the base class for all exceptions ⇒ handles everything
- The `finally` clause is always executed.

Read input repeatedly

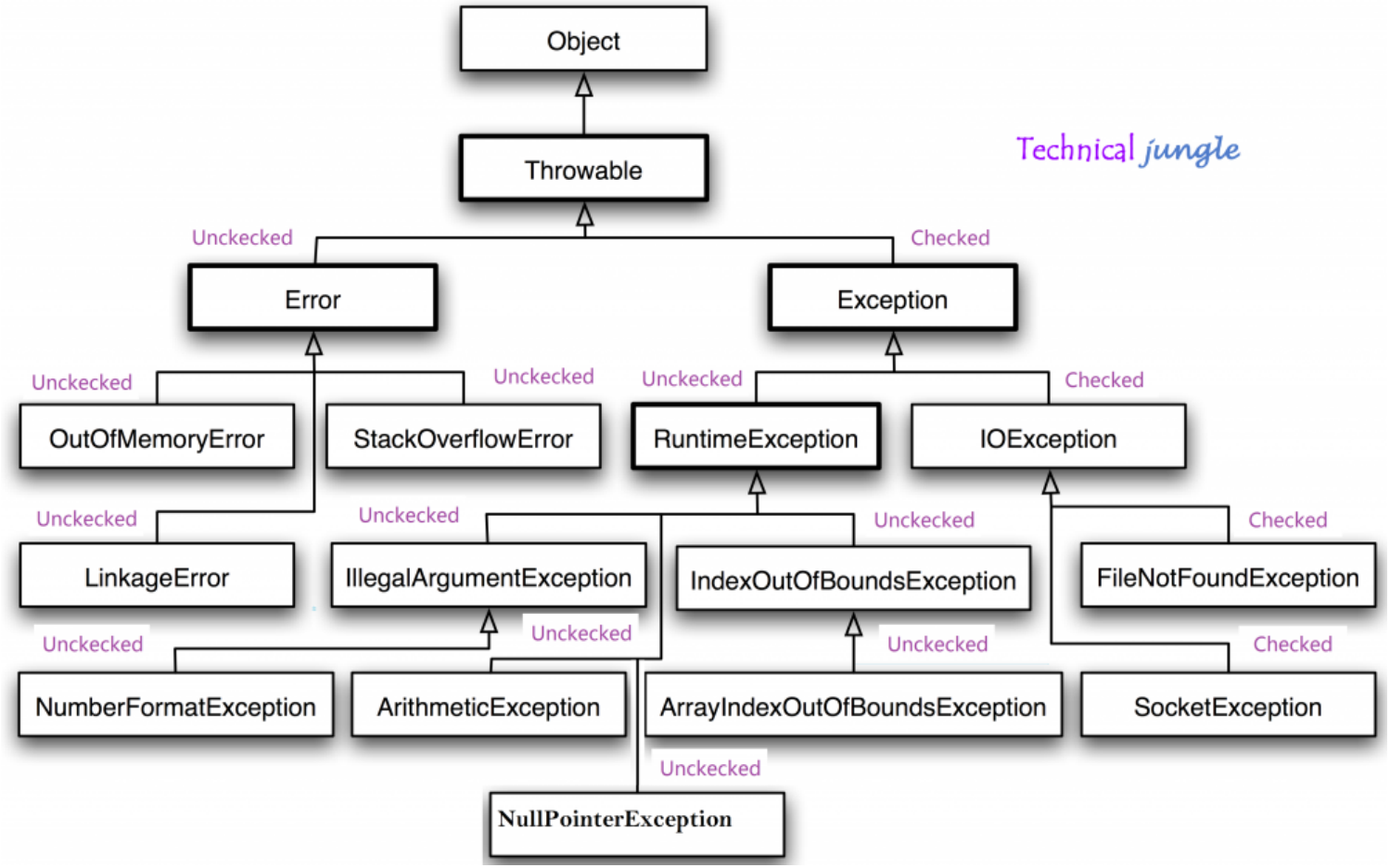
```
public static void main(String[] args) {
   int n = readInteger("Enter an integer: ");
   System.out.println(n);
}

public static int readInteger(String userInstruction) {
   Scanner sc = new Scanner(System.in);
   do {
      try {
         System.out.print(userInstruction);
         int n = sc.nextInt();
         sc.close();
         return n;
      }
      catch (InputMismatchException ime) {
         System.out.println("Input incorrect, try again!");
         sc.nextLine();   // Clear input stream
      }
   } while(true);   // Repeat forever!
}
```

`sc.nextInt()` throws an `InputMismatchException` if input entered is not an integer.

Hence, `readInteger(...)` do not return until user have entered an integer.

The Exception Hierarchy

Java has a large number of predefined exceptions. Many are available in the `java.lang` package but others are in related packages. For example, `IOException` is in `java.io`. Some are checked, other are unchecked.



Unchecked vs Checked

Java has two types of exceptions

- Unchecked
  - Not needed to be handled.
  - If it is not handled in the program, the JVM will catch it and terminate execution.
  - Inherits from `java.lang.RuntimeException`.
  - Many available exceptions to chose from (see slide 13)
  - None suitable? Throw a `RuntimeException` with a suitable error message
- Checked
  - Must be handled or passed on.
  - Pass on ⇒ no local handling, the method sends the responsibility to the calling method.

```
public void readFile(String path) throws IOException {
    ...       // Do something that might
    ...       // generate an IOException
}
```

Handle or Pass on (IO details later)

Read from file might raise a checked `IOException` that must be handled

```
public static void main(String[] args) {
    try {
        String text = readFile("C:\\Temp\\SmallText.txt"); // Might raise IOException
        ...                                                // Do something with text
    }
    catch (IOException ioe) { ioe.printStackTrace(); }
}

private static String readFile(String path) throws IOException { ... }
```

Alternatively

```
public static void main(String[] args) throws IOException {
    String text = readFile("C:\\Temp\\SmallText.txt");      // Might raise IOException
    ...
}
```

- **Note:** The `throws IOException` in the `main` method head ⇒ our program is not catching any IOExceptions ⇒ we delegate the catching to the JVM
- Not catching checked exceptions is poor programming practice ⇒ Don't do it!

Stack Traces

A catch block printing a stack trace

```
try {
    ...
    int n = list.get(100);
    ...
} catch (RuntimeException e) {
    e.printStackTrace();        // Prints a stack trace
}
```

Example output for a stack trace

```
1:  java.lang.IndexOutOfBoundsException: Index = 100, Upper boundary = 10
2:      at data_structures.IntList.checkIndex(IntList.java:96)
3:      at data_structures.IntList.get(IntList.java:50)
4:      at data_structures.ListMain.testList(ListMain.java:54)
5:      at data_structures.ListMain.main(ListMain.java:19)
```

- Line 1: Exception type and error message
- Line 2-5: A trace showing where error took place (line 96 in IntList.java)
- ... and the call sequence (starting in `main()`) that lead us to the error is ListMain.main(line:19) → ListMain.testList(line:54) → IntList.get(line:50) → IntList.checkIndex(line:96)

Stack Traces - Example

```
1:
2: public class TestException {
3:     public static void main(String[] args) { callMethodOne(); }
4:
5:     public static void callMethodOne() { callMethodTwo(); }
6:     public static void callMethodTwo() { callMethodThree(); }
7:
8:     public static void callMethodThree() {int result = 19/0; }
9: }
```

Example output for a stack trace

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestException.callMethodThree(TestException.java:8)
    at TestException.callMethodTwo(TestException.java:6)
    at TestException.callMethodOne(TestException.java:5)
    at TestException.main(TestException.java:3)
```

The exception (unchecked) in this case is handled by the JVM

A 10 Minute Break!!

ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ

File input/Output (I/O)

Read from and write to a text file is easy

```
StringBuilder text = new StringBuilder();

try {                                                    // Read text from file
    File file = new File("C:\\Temp\\input.txt");
    Scanner scan = new Scanner(file);          // Connect Scanner to file
    while (scan.hasNext()) {
        String str = scan.nextLine();          // End-of-Line not included
        text.append(str+"\n");                 // End-of-Line added
    }
     scan.close();
} catch (IOException e) {  e.printStackTrace();  }

try {                                                          // Save text in file
    File outFile = new File("C:\\Temp\\output.txt");
    PrintWriter printer = new PrintWriter(outFile);
    printer.print(text);     // Save text in file
    printer.close();
} catch (IOException e) { e.printStackTrace(); }
```

The Class `java.io.File`

- An object of type `File` represents a file (or directory) on your hard drive
- `... new File("C:\\Temp\\output.txt")` ⇒ connects to the file `output.txt`
- From the `java.io.File` documentation
  - `boolean exists()`: Tests whether the file/directory exists.
  - `boolean canRead()`: Tests whether the application can read the file.
  - `boolean canWrite()`: Tests whether the application can modify the file.
  - `boolean isDirectory()`: Tests whether the file is a directory.
  - `boolean isFile()`: Tests whether the file is a normal file.
  - `File[] listFiles()`: Returns an array of files in the directory denoted by this file.
  - `File getParentFile()`: Returns the parent directory.
  - ... and many more methods
- You specify a file by providing a **file path**
  - `File file = new File("C:\\Temp\\input.txt");  // PC`
  - `File file = new File("/Users/jlnmsi/Temp/input.txt"); // Mac`

  Notice that the Windows standard path separator `\` must be escaped as `\\`

Read and Write Text

Read text from file

```
File   file  = new File("C:\\Temp\\input.txt");          // PC Path
Scanner scan = new Scanner(file);
while (scan.hasNext()) {
    String  line  = scan.nextLine();          // Read one line
    // String  word = scan.next();            // Read one word
    ...
}
```

Write text to file

```
File  outFile  = new File("C:\\Temp\\output.txt");
PrintWriter  printer  = new PrintWriter(outFile);     // Creates  file  if  not  existing
printer . print (text);     // Save  text  in  file
printer . close ();
```

Both `Scanner` and `PrintWriter` come from the package `java.io`.
Both operations might generate `IOException` ⇒ must be handled with try/catch.
There are many library classes related to file IO. For example, to handle binary or audio files. We will only work with text files in this course.

Assignment 4 and Next Lecture

Assignment 4

- Will soon be publsihed
- Covers only this lecture ⇒ very short!
- Deadline: January 19

Next Lecture

- English: January 9
- Växjö/Swedish: January 8
- Kalmar/Distance: January 7
- An old exam will be presented ⇒ rehearsal, no new material

Written Exam: January 11

Merry Christmas!!!