Introduction to Java programming

Java collections, sorting and IO

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- This presentation some core Java APIs essential to almost any program:
  - Collections and sorting
  - File IO (New IO!)
  - Regular expressions
- We'll also encounter a prime OO design rule: program against interfaces, not implementations

PART ONE

Collections

The Collections API

- Storing variables **collectively** is nice; doing so in Arrays is usually not so nice - simply because they are static
- The Java Collections API offers a wealth of other ways to store data
- The Comparable and Comparator interfaces are used to do custom sorting of objects
- Finally (not a collection but essential to show you), we will see a nice way to deal with changeable String data: class StringBuilder

The old way: arrays

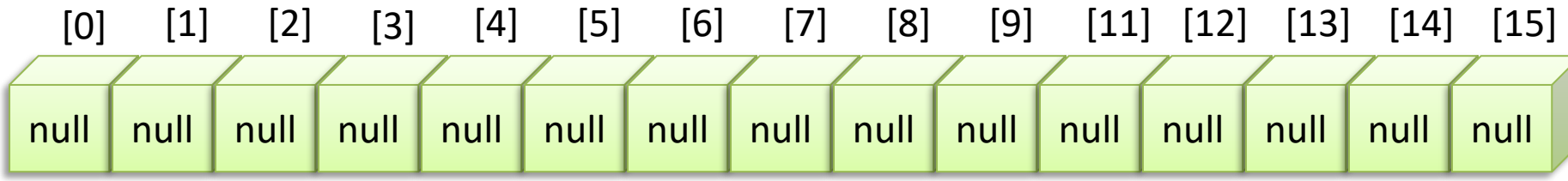
- In a previous lecture, you have seen the most basic collection type: array
- Arrays store a fixed number of variables
- But suppose you want to do something like this:

```
Sequence[] readSequences(String fileName) {
  //open file
  //read sequences
  //return the sequences as an array of Sequence
  //objects
}
```

You usually don't know how many sequences there are in a file!

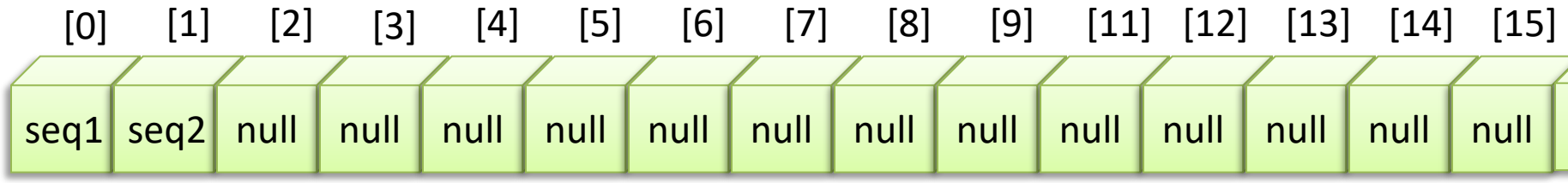
- So, you simply create a very large array that is as large as the biggest sequence file you know:

```
String[] readSequences( String fileName ){
  String[] sequences = new String[10000];
  //open file
  //read sequences
  //return the sequences as an arrays of Strings
}
```



Arrays: memory wasted or overflowed

- when you read only 1 or 2 sequences you will have wasted all this memory space
- when somebody invents a new sequencing machine that generates 1 million sequences in one run, you have a big problem!
  actually, as you should know, these machines already exist



ArrayList to the rescue

- There is a nice class that will really make your day: `java.util.ArrayList`

```
ArrayList<Sequence> seqs = new ArrayList<>();
```

What's that thing with the pointy brackets?

- All collection classes require you to specify the type of objects it will hold
- This ***type specification*** is done between the pointy brackets
- This mechanism is called ***generics***.

```
ArrayList<Sequence> sequences =
                          new ArrayList<>();
```

This declares the ArrayList to hold **ONLY** objects of type Sequence **AND ITS SUBTYPES**

ArrayList type declaration

- You in (very) old code or online examples, you may see the type declaration (<Sequence>) omitted, but this is not a good idea.

```
/*with type declaration; the correct Java7+ way to do it*/
ArrayList<Sequence> sequences = new ArrayList<>();
//fill it and fetch them
Sequence seq1 = sequences.get(0);

/*without type declaration; will give a compiler warning*/
ArrayList sequences = new ArrayList();
//fill it and fetch them
Sequence seq1 = (Sequence)sequences.get(0);
```

If you omit the type declaration, you have to do a type cast, otherwise ArrayList will return an object of type Object!

Class ArrayList must be imported

- You have to import it because its not in the `java.lang` core API:

```
package nl.bioinf.io;
import java.util.ArrayList;
class SequenceReader {
    ArrayList<Sequence> readSeqs(String file) {
    ArrayList<Sequence> sequences
        = new ArrayList<>();
    //fill the list with sequences
    return sequences;
  }
}
```

ArrayList methods

- To fill it and later traverse it, you do NOT use array-type indexing with square brackets. Here are a few example usages (check the others at your favourite site http://docs.oracle.com/javase/8/docs/api/):

```
ArrayList<Sequence> sequences = new ArrayList<>();
Sequence seq1 = new Sequence("GATTGCTCGGGATACC") ;
seq1.setName("seq1");
sequences.add(seq1);
//add more
System.out.println(sequences.toString());
System.out.println("list size is: " + sequences.size());
System.out.println("second sequence is: " + sequences.get(1));
sequences.clear();
```

This calls toString() on all list elements!

```
$ java ArrayDemo
[Sequence[name=seq1 length=16], Sequence[name=seq2 length=17]]
list size is: 2
second sequence is: Sequence[name=seq2 length=17]
```

ArrayList traversal

- ArrayLists and *for*-loops are seen together as often as new lovers

```
ArrayList<Sequence> sequences = fillMyList();

for(int i = 0; i < sequences.size(); i++){
  Sequence seq = sequences.get(i);
  //do something with the sequence
  //when you care about position
}

For(Sequence seq : sequences){
  //do something with the sequence
  //without caring where you are
}
```

Converting an array to an ArrayList

- Suppose you have a (fixed size) array and want to convert it into an ArrayList, for more flexibility:

```
//create object array
Element[] array = {new Element(1),
                   new Element(2),
                   new Element(3)};
//create ArrayList copy of the array
ArrayList<Element> arrayList = new ArrayList<>();
arrayList.addAll(Arrays.asList(array));
```

HashMap

- Of course, Java also supports **key-and-value** storage
- The Java class **`java.util.HashMap`** is well suited for holding collections of this type

HashMap

- Java HashMap objects also have to be instantiated with type declaration: one for the key and one for the value

```
HashMap<String, Sequence> seqs = new HashMap<>();
```

Now you have a HashMap that can hold Sequence objects that are accessible by a String key

Using a HashMap

- Here are some frequently used methods available in class HashMap. (See also http://docs.oracle.com/javase/8/docs/api/)

```
import java.util.HashMap;
...
  HashMap<String, Sequence> sequences = new HashMap<>();
  Sequence seq1 = new Sequence("GATTGCTCGGGATACC") ;
  seq1.setName("seq1");
  sequences.put(seq1.getName(), seq1);
  if(sequences.containsKey("seq1")){
    Sequence s1 = sequences.get("seq1");
  }
  System.out.println(sequences.toString() );
```

```
$ java HashMapDemo
{seq1=Sequence[name=seq1 length=16]}
```

HashMap and the hashCode() and equals() methods

- HashMap uses the hashCode() and equals() methods to store and retrieve values
- The next few slides cover these
- ***Never ever*** change the data you use for generating the hashcode: your objects will be lost in the collection:

```
HashSet<Element> elements = new HashSet<>();
Element one = new Element("One", 1);
elements.add(one);
System.out.println(elements.contains(one));
one.setName("OneChanged");
System.out.println(elements.contains(one));
elements.add(one);
System.out.println(elements.size());
```

```
$ java HashSetTester
true
false
2
```

Class Object

- In Java, *every* class extends from class Object, directly or indirectly.
- Class Object is dealt with in detail in the next presentation, but we'll have a peek at its definition here, for the sake of equals() and hashCode()

Class object

Object

```
public boolean equals(Object obj)
public int hashCode()
public String toString()
public final Class<?> getClass()
public final void notify()
public final void notifyAll()
public final void wait()
public final void public wait(
                    long timeout)
public final void wait(
                    long timeout,
                    int nanos)
```

These are the methods defined in class Object. All Java classes inherit these. You will have to **override** some of these to have real meaning. Others you can't touch: these are marked **final**

The grey ones have to do with multithreading

Methods equals() and hashCode()

- equals() and hashCode() have connected purposes
- They are especially important in “Hashed” collections
- **equals()** should return a boolean indicating whether two objects are *logically similar*
- The equality test operator **==** returns whether two objects are *the same* (the exact same memory address)

A naïve equals() implementation

- The most simple (and naïve) implementation for a class Cell would be

```
public boolean equals(Object other){
    return (Object)this == other;
}
```

This is actually the default implementation - how it is implemented in class Object

- This is of course correct, but when two cells have the same size and the same maxSize, you could also consider the two logically equivalent.
- This is shown in the next slide

A specific equals() implementation

- A more realistic implementation for class Cell would be

```
public boolean equals( Object other ){
    return (
        other instanceof Cell
        && this.getSize() == ((Cell)other).getSize()
        && this.getMaxSize() == ((Cell)other).getMaxSize());
}
```

Have a look at that nice instanceof keyword!

Requirements for equals() implementation

The equals() method must exhibit the following properties:

- ***Symmetry***: For two references, a and b, a.equals(b) if and only if b.equals(a) as well
- ***Reflexivity***: For all non-null references, a.equals(a)
- ***Transitivity***: If a.equals(b) and b.equals(c), then a.equals(c)
- ***Consistency with hashCode()***: Two equal objects must have the same hashCode() value

Use of hashCode()

- Method hashCode() is used for data structures in which hashing plays a role: HashMaps, HashSets etc.
- The hashCode() method should return an int value, so this

  ```
  public int hashCode(){return 1;}
  ```

  is a valid implementation, albeit a very ineffective one

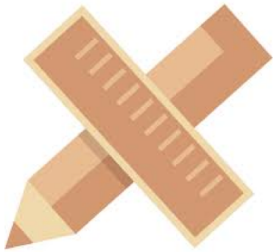
The hashCode() contract

- When it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer
- If two objects are equal according to the equals(Object) method, then the hashCode of each of the two objects must be the same integer.
- If two objects are unequal according to the equals(java.lang.Object) method, then the hashCode of the two objects may be differing. (However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables)
- *source: Java API on class Object*

Use of hashCode()

- This hashCode() method could be a more effective one:

```
public int hashCode(){
    return this.maxSize
                * this.size;
}
```



Design rules

- Always implement equals() and hashCode(), when objects are going to live in collections (which is quite often!)
- Always implement both equals **and** hashCode; never only one!

- And have your IDE generate it for you...

Primitives as objects: autoboxing

- You may have noticed the strange use of the primitive names in **`HashMap<Character, Double>`**.
- This is not a mistake; Java collections can **NOT** hold primitive values, only Objects.
- That's why every primitive type has an Object counterpart to represent it
- You can simply use it in a HashMap without this conversion: Java provides **autoboxing** for this.

```
molecularWeights.put('M', 149.21);
```

The **`char`** value 'M' is automatically boxed in an object of type **`Character`** and the **`double`** value 149.21 is boxed in type **`Double`**

Object representatives of primitives

- The classes Double, Integer, Character etc have other uses, as you'll see later. Here is a small sample

```
int max = Integer.MAX_VALUE;
Integer.parseInt("1100110", 2);
Integer i = new Integer(42);
i.compareTo(22);
```

Other Collection types

- There are ***many*** other collection types in the Java Collections API, each with their specific uses:
  - HashSet: like HasMaps, but without the values
  - LinkedList: when you do a lot of insertions/deletions to your list
  - Stack: you always have the last added element on top
  - Queue: for ... Queues
  - SortedMap…
- Check them out before automatically using ArrayList!

PART TWO

Sorting

Sorting Collections

- Sorting is something you do quite often
- For instance, a list of sequences you may want to sort on the name
- Sometimes, however, you may want to sort the same list based on sequence length, or the percentage of ‘A’ nucleotides
- As you will see, this is no problem in Java

Class java.util.Collections

- Java class **`Collections`** provides many methods that operate on ... collections
- It is a funny class because it cannot be instantiated into an object
- Among the more important methods available in class Collections are two variants of sort()

Collections.sort()

- Two flavours of Collections.sort()

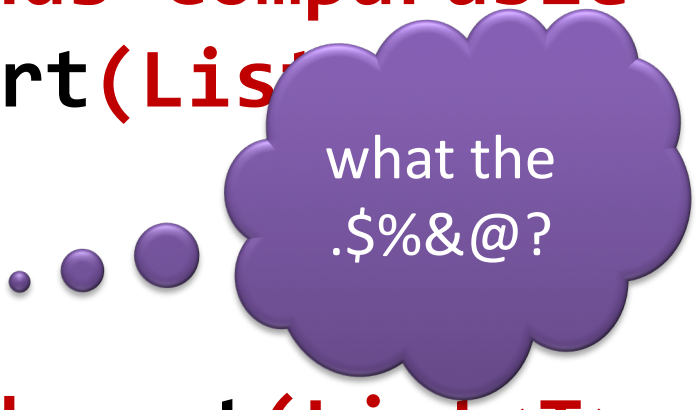
```
public static <T extends Comparable
  <? super T>> void sort(List<T>
  list)
```

```
public static <T> void sort(List<T>
  list, Comparator<? super T> c)
```

Collections.sort()

- Two flavours of Collections.sort()

```
public static <T extends Comparable
  <? super T>> void sort(Lis
  list)
```




```
public static <T> void sort(List<T>
  list, Comparator<? super T> c)
```

Collections.sort() dissected

```
public static
<T extends Comparable <? super T>>
void sort
(
 List<T> list
)
```

This says: pass sort() a List holding some objects of type (T) that implements the Comparable interface. ArrayList implements the List interface, so that is no problem. Only challenge here is making Sequence implement the Comparable **interface**.

What is an interface?

- An interface is like a Class ***contract***
- When you say a class implements an interface, it will be **forced** to fullfill the contract specified in the interface
- The contract specifies the methods **signatures** that come with the contract, but not how they are implemented (the method body)

Why an interface?

- A class can **extend** only one other class but can **implement** many interfaces

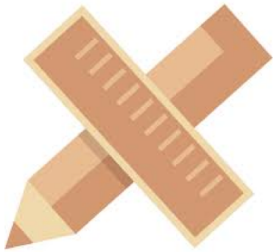
```
class Element extends Object
  implements Comparable, Doable,
  Serializable, Cloneable{
...
}
```

Interfaces in Java 8

- Actually, from Java 8 on, interfaces ***can*** have method implementations
- These are called ***default*** methods
- One of the restrictions with these is that they can not depend on member variables

So what does it add?

- A class that implements an interface can be dealt with in a IS_A manner, as if it ISA instance of that class
- Thus, a objects of a class that implements Comparable can be handled as Comparable instances (but only the Comparable part is visible)



Design rule

- Program against interfaces*, not implementations

(Make your code dependent on generic types, not specific types)

* interface means here: interface, abstract class or supertype

interface java.lang.Comparable

- Interface Comparable is defined in package *java.lang* and it defines this contract:

```
public interface Comparable<T> {
   public int compareTo(T other);
}
```

Every class implementing this interface (signing the Comparable contract) will **HAVE** to implement this method!

No method body, there is only a signature

The int return value of compareTo()

- The compareTo() methods must return an int
- What is the interpretation of this value?
- In short

```
this.compareTo(that)
```

should return

- a negative int if **this < that**
- 0 if **this == that**
- a positive int if **this > that**

Implementing an interface
=
signing the contract

- You may still be a bit confused (all normal people are at this point) so let's look at some code to illustrate things
- First declare class Sequence to be a Comparable one:

```
public class Sequence implements Comparable<Sequence>{
   public void getLength() {return this.seq.length();}
   //rest of the class code

}
```

Declare class Sequence to be Comparable (for Sequence objects). You are now committing yourself to implementing all methods defined in the interface (contract) Comparable

Living up to the contract (naively)

- implement compareTo( )

```
/**implements Comparable.compareTo(), sorts by length,
 * shortest length will come first */
public int compareTo(Sequence other){
  if(this.getLength() < other.getLength()){
    return -1;
  }
  else if(this.getLength() > other.getLength()){
    return 1;
  }
  else{ return 0; }
}
```

This implementation will sort on sequence length

Living up to the contract (more) efficiently

```
/**implements Comparable.compareTo(), sorts by length,
 * shortest length will come first */
public int compareTo(Sequence other){
    return this.getLength() - other.getLength();
}
```

This is functionally the same but more efficient and more comprehensive

Living up to the contract correctly

```
/**implements Comparable.compareTo(), sorts by length,
 * shortest length will come first */
public int compareTo(Sequence otherSequence){
  return Integer.compare(
       this.getLength(), other.getLength());
}
```

This is the implementation that will not break at the boundaries of integer size

Go for a test drive

```
ArrayList<Sequence> sequences = new ArrayList<Sequence>();
Sequence seq1 = new Sequence("GATTGCTCGGGATA") ;
seq1.setName("seq1");
sequences.add( seq1 );
Sequence seq2 = new Sequence("GFTTWEDYLPPGSQA") ;
seq2.setName("seq2");
sequences.add( seq2 );
Sequence seq3 = new Sequence("AGTCATGG") ;
seq3.setName("seq3");
sequences.add( seq3 );
System.out.println( "list before sorting " );
System.out.println( sequences.toString() );
Collections.sort( sequences );
System.out.println( "list after sorting" );
System.out.println( sequences.toString() );
```

```
list before sorting
[Sequence[name=seq1 length=14], Sequence[name=seq2 length=15], Sequence[name=seq3 length=8]]
list after sorting
[Sequence[name=seq3 length=8], Sequence[name=seq1 length=14], Sequence[name=seq2 length=15]]
```

Alternative sorting

- Usually objects based have some default sorting property (e.g. Last name)
- Sometimes you want to sort on other properties, preferably without changing compareTo()
- This is where the second version of sort() comes in

Collections.sort() using Comparator

- This is the second version again:

```
public static <T> void sort
(List<T> list, Comparator<? super T> c)
```

This version of sort() requires two arguments: A List of objects of a certain type (T) and a Comparator object that is able to compare objects of this type T

java.util.Comparator

- Comparator is an interface again, not a real class
- You have to implement it (usually in a dedicated class with only one responsibility: comparing two objects)

```
import java.util.Comparator;

public class SequenceNameComparator
      implements Comparator<Sequence> {
  //class code
}
```

interface Comparator is in package java.util You have to import it

Commit yourself to the contract of comparing Sequence objects

java.util.Comparator

- Now implement the single method defined in the contract:

```
import java.util.Comparator;

public class SequenceNameComparator
                        implements Comparator<Sequence> {
  public int compare (Sequence first, Sequence second) {
    return first.getName().compareTo(second.getName());
  }

}
```

Easy: simply **delegate** to the compareTo() method provided by class String. Class String knows very well how to compare Strings alphabetically!

Comparator test drive

```
ArrayList<Sequence> sequences = new ArrayList<Sequence>();
Sequence seq1 = new Sequence("GATTGCTCGGGATA") ;
seq1.setName("dna1");
sequences.add( seq1 );
Sequence seq2 = new Sequence("GFTTWEDYLPPGSQA") ;
seq2.setName("pp1");
sequences.add( seq2 );
Sequence seq3 = new Sequence("AGTCATGG") ;
seq3.setName("dna2");
sequences.add( seq3 );
System.out.println( "list before sorting " );
System.out.println( sequences.toString() );
Collections.sort( sequences, new SequenceNameComparator() );
System.out.println( "list after sorting" );
System.out.println( sequences.toString() );
```

```
list before sorting
[Sequence[name=dna1 length=14], Sequence[name=pp1 length=15], Sequence[name=dna2 length=8]]
list after sorting
[Sequence[name=dna1 length=14], Sequence[name=dna2 length=8], Sequence[name=pp1 length=15]]
```

Anonymous inner classes

- You do NOT have to create a separate class file for each Comparator.
- The next slides show some alternative ways to create interface implementers without creating separate class files.
- These are all called (anonymous) inner classes

Anonymous inner class (version 1)

```
Collections.sort(sequences,
  new Comparator<Sequence>() {
    public int compare(
      Sequence first, Sequence second) {
        return first.
                  getName().
                  compareTo(second.getName());
    }
  }
);
```

Yes, you can! Define a brand new class without a name within an ordinary method call and construct a new instance of it. This is called an **anonymous local inner class**

Anonymous inner class (version 2)

- Create a factory method that serves the right comparator

```
public Comparator<Sequence> getNameComparator() {
  return new Comparator<Sequence>() {
    public int compare(Sequence first, Sequence second) {
      return first.getName().compareTo(second.getName());
    }
  };
}
```

Again an example of an anonymous local inner class. The point is, nobody ever needs to know the type of your implementer; knowing that the interface has been implemented is enough!

Summary on collections

- You have seen some essential elements of real-world Java
- Along the way, you have met a integral element of Java programming: interfaces. If you stick with Java, you will see many more and also define your own
- Remember that the Java Collections framework has many more collection types that you have not seen here

PS on sorting in Java 8

- In Java 8, sorting has become a bit simpler with lambdas (for some, who like Lambdas)
- E.g. refer to https://www.mkyong.com/java8/java-8-lambda-comparator-example/
-

PART THREE

Accessing and processing files

Introduction

- Almost all applications work on external data – often files, databases and internet resources
- Here we will only have a look at files
- Since accessing external data is inherently risky, we need to discuss Exceptions as well
- We will also have a look at pattern matching using regular expressions because this is simply the best place to do that

Representing the file system

- In Java 7+, you use the Path and Paths classes to represent files and directories:

```
Path path = Paths.get("/Users/michiel/Desktop/test.txt");
System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0, 2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

```
toString: /Users/michiel/Desktop/test.txt
getFileName: test.txt
getName(0): Users
getNameCount: 4
subpath(0,2): Users/michiel
getParent: /Users/michiel/Desktop
getRoot: /
```

Here, the file does not exist yet! There is only a path definition

Creating files

- To create a file you type
  - Files.createFile(path);
- When you are not sure whether it already exists:

```
if(! Files.exists(path)){
    try {
        Files.createFile(path);
    } catch (IOException ex) {
        System.err.format("IOException: %s%n", ex);
    }
}
```

The try/catch will come back to haunt you later!

File properties and manipulations

- class Files offers a wealth of File manipulating and testing methods
- Here are a few

```
Files.isDirectory(path);
Files.isReadable(path);
Files.isWritable(path);
Files.exists(path);
try {
    Files.deleteIfExists(path);
} catch (IOException ex) {
    System.err.format("IOException: %s%n", ex);
}
```

Writing to file

- To write to a file, you need BufferedWriter or a PrintWriter

```
Charset charset = Charset.forName("US-ASCII");
String s = "Hello File";
try (BufferedWriter writer = Files.newBufferedWriter(
        path,
        charset,
        StandardOpenOption.APPEND)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

This will open in append mode! Omit this argument if you want to overwrite

StandardOpenAction enum values

- **APPEND** If the file is opened for WRITE access then bytes will be written to the end of the file rather than the beginning
- **CREATE** Create a new file if it does not exist.
- **CREATE_NEW** Create a new file, failing if the file already exists.
- **READ** Open for read access
- **WRITE** Open for write access.

PrintWriter

- Use a PrintWriter instance when you want easy string formatting and correct end-of line characters

```
try (PrintWriter writer = new PrintWriter(Files.newBufferedWriter(
        path,
        charset,
        StandardOpenOption.CREATE))) {
    writer.println(s);
    writer.printf("dit is een %s string met nummer: %f", "test", 42.0);
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Reading from file

- Reading is very similar to writing:

```
try (BufferedReader reader = Files.newBufferedReader(
        path,
        Charset.defaultCharset())) {
    String lineFromFile;
    while ((lineFromFile = reader.readLine()) != null) {
        System.out.println(lineFromFile);
    }
} catch (IOException exception) {
    System.out.println("Error while re
}
```

This will read lines from the file while there are still any left

Opening and reading from file pre-Java7

- Because you'll probable encounter this legacy code, here is the corresponding code from Java 6 and earlier

```
File input = new File("C:…\\myFile.txt");
BufferedReader br = null;
try{
    String line = null;
    br = new BufferedReader(new FileReader(input));
    while( (line = br.readLine()) != null ){
        System.out.println(line);
    }
    br.close();
} catch (IOException e){
    e.printStackTrace();
}
```

Try…and catch

- Since accessing external data is error-prone, everything you want to do with these sources could throw you an Exception
- Exceptions are return values that do not follow normal program flow; they are instantiated when something goes wrong
- Risky code should always be placed within a try/catch block (or throw an exception)
- We'll expand on this concept a bit

Generating an Exception

- Forcing an exception, using this code, will generate an error resulting in a ***stack trace***:

```
public static void main(String[] args) {
	foo();
}
public static void foo(){
	bar();
}
public static void bar(){
	int x = 42/0;
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionDemo.bar(ExceptionDemo.java:16)
at ExceptionDemo.foo(ExceptionDemo.java:12)
at ExceptionDemo.main(ExceptionDemo.java:8)
```

The stack

- A call stack is a data structure that stores information about the active subroutines of a computer program
- The current executing function is always on top
- When an exception or error occurs, you will see a ***traceback*** through this stack, from the top (where the error occurred) back to the main() function
- This gives you information about the origin and nature of the error

Dividing a number by zero will raise an ArithmeticException

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionDemo.bar(ExceptionDemo.java:16)
at ExceptionDemo.foo(ExceptionDemo.java:12)
at ExceptionDemo.main(ExceptionDemo.java:8)
```

This was done at line 16 in method bar() of class ExceptionDemo

Exception basics

- An Exception object is created when something exceptional occurs and the program can not run its normal course
- The Exception object is thrown from this point down the call stack ***unless it is caught***
- If you know something risky is going on, you can
  - **Catch it yourself**: place this code in a try/catch structure

```
try{ /*the risky thing*/ }
catch(Exception e){ /*solve the problem*/ }
```

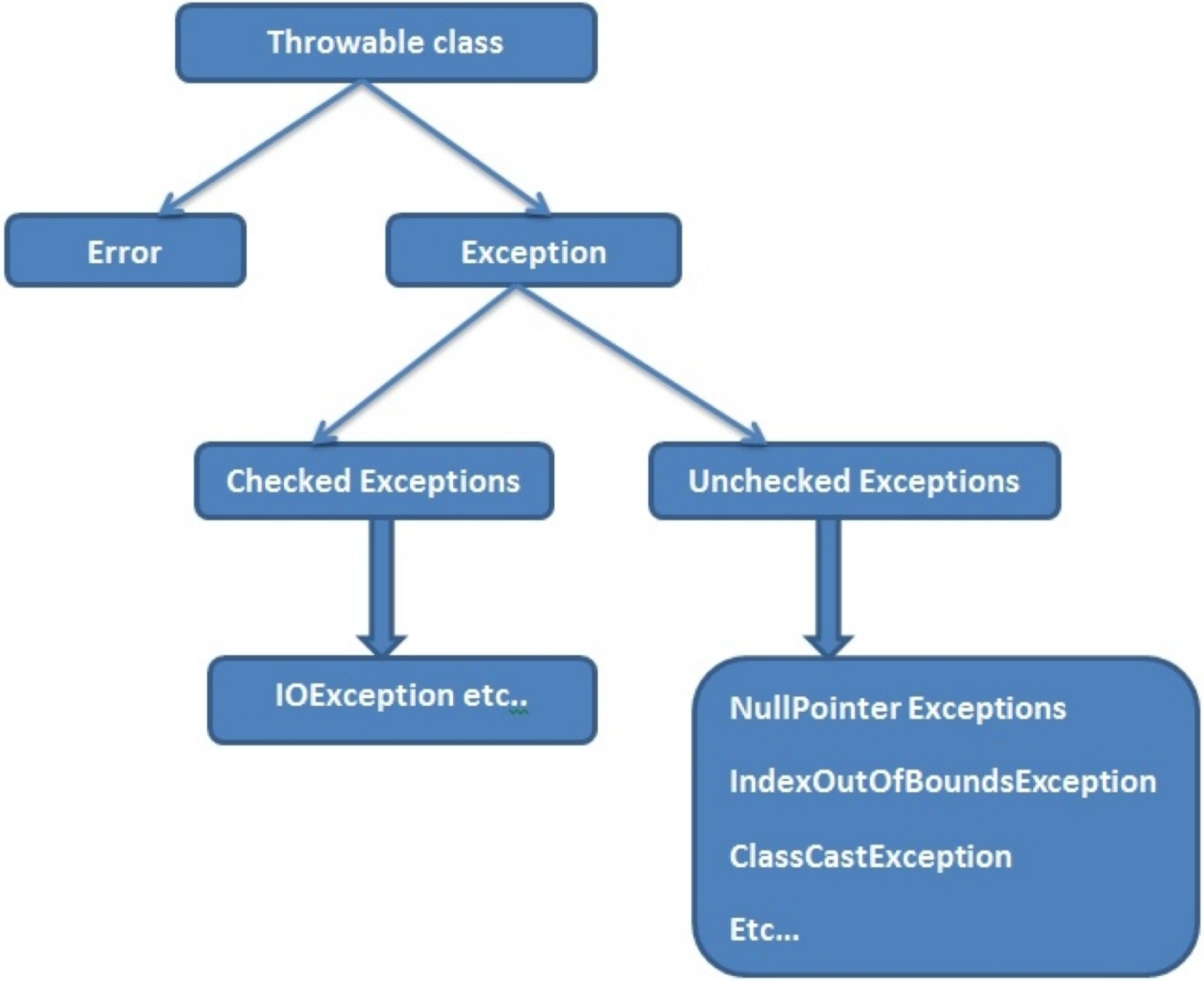
  - **Make other code solve the problem**: place a throws clause in the method signature:

```
public void doRisky() throws Exception{
    /*method body*/
}
```

Exception types

- There is a distinction between *checked* and *unchecked* exceptions
  - ***Checked*** exceptions *must* be explicitly caught or thrown, or the code will not compile
  - ***Unchecked*** exceptions, like the ArithmeticException of the previous example, need not (but may) be explicitly caught or thrown

Exception types



Exception hierarchy

- The Java language knows many Errors and Exceptions, all inheriting from class Throwable
- Catching a supertype means also catching all of its descendants!
- You can of course always make your own subclass from any member of this tree.

All try/catch elements

- Here all syntactic elements involved in the Exception mechanism

```
try{
      /*the risky thing*/
} catch(FileNotFoundException fnfe |
      EndOfFileException) {
      /*solve some file problems*/
} catch(IOException) {
      /*solve all other IO problems*/
} finally {
      /*do things that need to be done, no
      matter what, like closing a file or
      database connection*/
}
```

This is a multicatch (new in Java 7)

Subtypes should always precede the supertype!

try with resources (Java 7+)

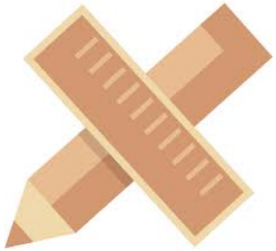
- The try/catch syntax for file reading and writing warrants some extra attention

```
try(<OPEN RESOURCE>){}
catch(<RESOURCE EXCEPTION>){}
```

- This form assures that a resource will be closed() after the try{} block exits, with or without error

Programming with exceptions

- Here you saw the exception ***syntax*** introduced
- Programming with correct handling of exceptions warrants an entire course itself but unfortunately there is no time for that.



Design rules

- Never ever let exceptions pass silently!
- Deal with exceptions at an appropriate location
- Nowadays, the trend is to prefer unchecked exceptions over checked exceptions (see discussion at http://www.yegor256.com/2015/07/28/checked-vs-unchecked-exceptions.html)

Exceptions summary

- All checked exceptions must be dealt with, either with a try/catch structure or with a throws clause in the method signature
- A *try* block can be followed by multiple *catch* clauses, as long as you go from a specific type to a general one, or stay at the same level in the exception class tree
- A checked exception *must* be caught somewhere in the call stack
- You can use Java's exception types or create your own
- PS: Never ever try to *catch* an Error!

PART THREE

Regular expressions and their usage

Parsing file content the easy way (1)

- Often, your data will be well-structured and parsing them is simple
- Just use the String.split() method, together with some conversion methods for creating ints and doubles and such from strings
- These are the employee data we are going to process:

```
Name    Age     Function        Salary*1000
JohnDoe         27      Programmer      23.67
JaneSmith       31      Manager         42.00
JoanDonnely     38      CEO     89.87
RoseBanner      18      secretary       11.78
```

empl_data.csv is a tab-separated file. Note the first line is a header and not real data!

Parsing file content the easy way (2)

- This is a typical data processing example

```
String line = null;
int lineNumber = 0;
br = Files.newBufferedReader(path, Charset.defaultCharset());
While ((line = br.readLine()) != null) {
        lineNumber++;
        if (lineNumber == 1) continue;     //skips first header line
        String[] elements = line.split("\t");    //split on tabs
        String name = elements[0];
        int age = Integer.parseInt(elements[1]); //convert to int
        String function = elements[2];
                                                 //convert to double
        double salary = Double.parseDouble(elements[3]) * 1000;
        System.out.println("name=" + name
                           + "; age=" + age
                           + "; function=" + function
                           + "; salary=" + salary );
}
```

Parsing text with Regular Expressions

- Regular expressions are used to search for patterns in text
- For instance, if you want to find occurrences of zip codes, you could use this:
  [a-zA-Z]{4} ?[0-9]{2}
- This part only introduces the Java regex API
- It is NOT a tutorial on regex!

- PS, I borrowed some examples from http://www.vogella.com

What is a regex pattern

- A regular expression is used to describe a pattern that is not literal, i.e. different strings could match the pattern
- Note: backslashes have meanings both in String literals and in regex. They denote special characters, but also negate the special meaning
- Thus, to match a literal backslash, your regex will be "\\\\" !!

Character classes

Character class	Description
.	Any character
[]	Set definition, eg [a-z] matches all lowercase characters; [aAbB1234] matches an a, A, b, B or numbers 1, 2 ,3, 4
[^]	Negated set definition, eg [^a-z] matches anything BUT lowercase characters
X|Z	Matches X or Z (either one will suffice)
^regex	Anchors regex at beginning of line
regex$	Anchors regex at end of line
\d \D	Any digit, [0-9]; any non-digit, [^0-9]
\s \S	Any whitespace character; any non-whitespace character
\w \W	A word character, short for [a-zA-Z_0-9]; same but negated
\b	Word boundary

Regex quantifiers

Quantifier	Description
{x}	Occurs exactly x times
{x, }	Occurs at least x times
{, x}	Occurs at the most x times
*	Occurs zero or more times; same as {0, }
+	Occurs one or more times; same as {1, }
?	Occurs zero or one time; same as {0, 1}
*?	Non-greedy: "?" after a quantifier makes it a *reluctant quantifier*, it tries to find the smallest match.

Grouping

- Using parentheses, you can group parts of your regex
- They can be used to
  - Retrieve or substitute parts of a regex
  - Apply quantifiers to groups
- Via the $ you can refer to a group. $1 is the first group, $2 the second, etc.

String, Pattern and Matcher

- For Java regex, these classes are related to regular expression matching (and replacement):
  - java.lang.String
  - java.util.regex.Pattern
  - java.util.regex.Matcher
- Many of the common tasks can be performed using the String class only

String methods involving regex

Method	Description
s.matches(regex)	Tells whether or not this string matches the given regular expression.
s.split(regex)	Splits this string around matches of the given regular expression
s.split(regex, limit)	Idem, but the limit parameter controls the number of times the pattern is applied
s.replaceAll(regex, replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement
s.replaceFirst(regex, replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement.

Regex examples with String

```
String EXAMPLE_TEST = "This is the <titel>example</title> "
     + "string which , I'm going to use for pattern matching.";

//split on whitespaces
String[] splitString = (EXAMPLE_TEST.split("\\s+"));

// Removes whitespace between a word character and . or ,
String pattern = "(\\w)(\\s+)([\\.,])";
System.out.println(EXAMPLE_TEST.replaceAll(pattern, "$1$3"));

// Extract the text between the two title elements of
pattern = "(?i)(<title.*?>)(.+?)(</title>)";
String updated = EXAMPLE_TEST.replaceAll(pattern, "$2");

// prints true if the string contains a number less then 300
System.out.println(s.matches("[^0-9]*[12]?[0-9]{1,2}[^0-9]*"));
```

Pattern and Matcher

- The classes java.util.regex.Pattern and java.util.regex.Matcher are used for more advanced regex tasks than what String offers
- Moreover, they are more efficient; String methods are not optimized for performance while these are
- Here is a typical example of their use

Pattern and Matcher in action

```
String resEnz = "HincII (GTYRAC)";
String seq = "ACAGTCGACTGATCGGGTTGACCCATAGCTACGGTGAAAGCTT";
String hinc2Pattern = "GT([CT][AG])AC";
// Create a Pattern object
Pattern r = Pattern.compile(hinc2Pattern);
// Now create matcher object.
Matcher m = r.matcher(seq);
// Report all matches
while(m.find()) {
    System.out.println("Found occurrence of " + resEnz
       + " at pos " + (m.start()+1)
       + ": " + m.group(0)
       + "; ambiguous bases are " + m.group(1) );}
```

```
Found occurrence of HincII (GTYRAC) at 4: GTCGAC; ambiguous bases are CG
Found occurrence of HincII (GTYRAC) at 17: GTTGAC; ambiguous bases are TG
```