Java 101

Luca Telloli
luca.telloli@upf.edu

Pablo Martín Castagnaro
pablo.castagnaro@upf.edu

Summary

- Java
  - Strings
  - String equality
  - File I/O
  - Exception handling
  - Access to resources
  - Packages
  - Interfaces and their implementations
  - Unit testing
- Maven
  - Project structure
  - Compilation
- Other common errors
  - Hardcoded paths
  - Non-minimal imports

Python -> Java

If you come from Python, there are several resources to learn Java for Python Programmers.

- Java4Python
- Java for Python Developers. Youtube video
- And many others! Check Google for more

A list of problems with possible solutions

Java: Appending to Strings

- String is Java are immutable
- Using the operator + takes two string and generate a third string
- Each operation takes two existing chunks for memory and allocates a third chunk where the result is stored
- Accumulating on a string within a loop means that each new chunk is larger than the previous one ==> Performance decreases rapidly!

Test: Strings

```
List<Integer> ints = Arrays.asList(10, 100, 1000, 10000, 100000);
ints.forEach(n -> {
   String str = "";
   long start = System.currentTimeMillis();
   for(int i = 0; i < n; i++) {
       str += "aaa";
   }
   long stop = System.currentTimeMillis();
   System.out.println("Done in ms: " + (stop - start));
   System.out.println("AVG: " + ((stop - start) * 1.0/n));
   System.out.println("String: " + str.length());
});
```

```
String: 30
Done in ms: 0
AVG of 10 steps : 0.0

String: 300
Done in ms: 0
AVG of 100 steps : 0.0

String: 3000
Done in ms: 11
AVG of 1000 steps : 0.011

String: 30000
Done in ms: 286
AVG of 10000 steps : 0.0286

String: 300000
Done in ms: 11606
AVG of 100000 steps : 0.11606
```

Solution: Appending to String

- Use Java `StringBuilders`!
- https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html

```
StringBuilder buffer = new StringBuilder("aaa");
for (int i = 0; i < 100; i++)
   buffer.append("aaa");
System.out.println(buffer.toString());
```

For the objectives of Lab 1, it is not necessary to use string buffers. Why? Because You don't want to accumulate data in memory if you can flush it to file right away!

Problem: String Equality

- Java Strings are objects
- The == operator with objects check the object reference in memory, not the content!
- For checking content equality, you should use the `.equals()` method, which for strings implement equality
- Keep in mind that, for any generic class, == implements reference equality unless you override it!

```
String a = "123";
String b = "123";
String c = new String("123");

// TRUE, same object, same reference in memory
System.out.println(a == a);

// Surprisingly TRUE, compiler's smart: sees we're using the same string
System.out.println(b == a);

// FALSE, as expected, since we forced new memory allocation
System.out.println(c == a);

// TRUE, the content is the same
System.out.println(c.equals(a));
```

Problem: Reading/writing from/to files

- Java provides different ways of accessing files
  - The classic InputStream/OutputStream hierarchy
  - The more modern Reader/Writer hierarchy
- If you wrap a Reader into a BufferedReader or an InputStream into a BufferedInputStream, you can read files by line, which is a common way to access files in data-oriented applications
- Same for Writer & BufferedWriter, OutputStream & BufferedOutputStream
- You should generally prefer the use of a Reader/Writer vs InputStream/OutputStream. Reader has full support for 16 bit characters (Unicode, which includes, for instance, EMOJIs)
- Reader/Writers/InputStreams/OutputStreams should always be explicitly closed
  - (*Optional*) You could also investigate as an alternative the so called "r"

Example using Readers/Writers

```
FileReader reader = new FileReader("/some/file.txt");
BufferedReader bReader = new BufferedReader(reader);

String line = bReader.readLine(); // Read one line of content

FileWriter writer = new FileWriter("/some/other/file.txt");
BufferedWriter bWriter = new BufferedWriter(writer);

bWriter.write(line);    // Write one line of content

bReader.close(); // Close buffered reader and enclosed reader
bWriter.close(); // Close buffered writer and enclosed writer
```

Problem: Returning a List<String> (buffering output in general)

- Similarly to the big string problem, is the idea of accumulating lines in memory as a collection of strings
- The application is reading and writing onto files, so you should keep in memory only the minimal amount necessary from moving data from the source to the destination
- For the purpose of the assignment, the minimal amount is... 1 line! Why? Because you either write that line to the current output file, or you close the current and open a new output file

Problem: Handling exceptions

- Exceptions are a way to signal a deviation from an expected behavior
- Exceptions are of two types:
  - Checked: you need to explicitly catch or rethrow them within your code. (Ex: IOException)
  - Unchecked: you're not forced to catch them unless you need to for your particular flow
- If you catch an exception, you're generally into two possible situation:
  1. You can bring the flow of execution back to normality, by performing some actions within the catch block
  2. You can't bring the flow of execution back to normality (for instance, a file is missing), so you should terminate the program
- If you catch an exception and do nothing in the catch block (or just a printout), you're in fact making your flow of execution such that whatever comes next is ok

Example: Handling exceptions

```
try {

   FileReader fileReader = new
FileReader(f.toString());
   BufferedReader bufferedReader = new
BufferedReader(fileReader);

   // ... Do STUFF HERE...

   // Close input file
   bufferedReader.close();
} catch(FileNotFoundException ex) {
   ex.printStackTrace();
} catch(IOException ex) {
   ex.printStackTrace();
}
```

- In the code on the left, we're catching all file related exceptions, common when: a files is not found, the content of the stream could not be read, etc..
- We're printing the stacktrace, but not doing anything
- Whatever is after the try will think that the files have been read correctly and the flow was *correct* or *corrected*

Solution 1 (tentative): Handling exceptions

```
try {

   FileReader fileReader = new
FileReader(f.toString());
   BufferedReader bufferedReader = new
BufferedReader(fileReader);

   // ... Do STUFF HERE...

   // Close input file
   bufferedReader.close();
} catch(IOException ex) {
   ex.printStackTrace();
   System.exit(1);
}
```

- We don't know how to fix the exception so we terminate the execution after printing the stacktrace
- We signal the abnormality by returning an exit code different than 0
- Not the best solution because we don't explicitly close the resources we're using (the readers, etc)
- Exiting makes sense only in the main class

Solution 1 (better): Handling exceptions

```
int exitStatus = 0;
BufferedReader bufferedReader;

try {
   FileReader fr = new FileReader("aFile");
   bufferedReader = new BufferedReader(fr);

   // ... Do STUFF HERE...

   // Close input file
} catch(IOException ex) {
   ex.printStackTrace();
   exitStatus = 1;
} finally {
  // Closing resources, can also generate an
exception :(
   bufferedReader.close();
}
System.exit(exitStatus);
```

- We explicitly signal an abnormal exit condition at the end of the flow
- We include a finally block where we close resources. The finally block always gets executed, also if there's no exception in the flow
- The actions in the catch block should possibly not generate other exceptions
- If it does (`.close()` does) we're forced to make the code less readable or re-throw the exception

Solution 2 (easier): Propagating exceptions

```
void doSomething() throws IOException {
     BufferedReader bufferedReader;

     try {
        FileReader fr = new
          FileReader(f.toString());
        bufferedReader = new BufferedReader(fr);

        // ... Do STUFF HERE...

        // Close input file
     } catch(IOException ex) {
        ex.printStackTrace();
        throw ex;
     } finally {
       // Close resources
        bufferedReader.close();
     }
}
```

- Solution 2: we rethrow exceptions.
- The client of my class will have to take care of the IOExceptions
- We can concentrate all the exceptions in the main and capture them there
- This is what was suggested in the text of the assignment

Solution 3 (even easier): Propagating exceptions

```
void doSomething() throws IOException {
     BufferedReader bufferedReader;

     try {
        FileReader fr = new
          FileReader(f.toString());
        bufferedReader = new BufferedReader(fr);

        // ... Do STUFF HERE...

        // Close input file
     } finally {
       // Close resources
        bufferedReader.close();
     }
}
```

If we just re-throw exceptions, then we can even skip the catch block here ;)

Solution 3.b (even more easy): Propagating exceptions

```
void doSomething() throws IOException {

     try(FileReader fr = new FileReader(f.toString());
         BufferedReader br = new BufferedReader(fr);) {


        // ... Do STUFF HERE...

        // Close input file
     } // Resources will be automatically closed
}
```

Using a "try-with-resource" block, we can avoid the finally. The resource will be closed anyway at the end.

Accessing resources in Java

Resource: any piece of static data that the application should use during execution

Through the global `ClassLoader` class	Access *system* resources. *Not very portable* :-(	`ClassLoader.getSystemResource(...)`
		
		`ClassLoader.getSystemResourceAsStream(...)`

Through a class' class loader	Access *local* resources, including JAR resources	`this.class.getResource()`
		`AnyClass.class.getResource()`
		
		`this.class.getResourceAsStream()`
		`AnyClass.class.getResourceAsStream()`

Java packages

A Java Package is used to group related classes definitions

Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (packages created by the developer)

Syntax:

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

Java packages. User-defined packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders in your machine:

```
└── src
    └── mypack
        └── MyPackageClass.java
```

```java
package mypack;

class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }

}
```

Use the **package** command to create a user-defined package.

Java Interfaces and their implementations

- An **interface** in the Java programming language is an abstract type that is used to specify a behavior that classes must implement.
- Interfaces are declared using the `interface` keyword, and may only contain method signature and constants.
- A class implementing an interface must implement all the methods declared in the interface with exact same signature (name + parameters) as declared in the interface.

Java Interfaces and their implementations. Practical example

An interface for vehicles

```
interface Vehicle {
  public void moveTo(int x, int y);
}
```

```
public class Car implements Vehicle {
  //Implementing the method
  public void moveTo(int x, int y){
      this.position = drive_to(x, y);
  }
}
```

Two implementations for `Vehicle`:

```
public class Boat implements Vehicle {
  private final float length;
  public Boat(float length) { this.length = length; }
  //Implementing the method
  public void moveTo(int x, int y){
      this.position = sail_to(x, y);
  }
}
```

Get an instance of a `Vehicle`:

```
Vehicle car = new Car()
Vehicle boat = new Boat(1.4)
```

Java Interfaces and their implementations

- Class vs Interface

Class	Interface
In a class, you can instantiate variable and create an object	In an interface, you can’t instantiate variable and create an object
Class can contain concrete methods (with their implementation)	(From Java 1.8+) An interface can contain default methods implemented
The access specifiers used with classes are private, protected and public	In an interface, the only allowed access specifier is public.

Unit testing

Motivation: given a method, test relevant cases to check if it works correctly

Junit is most well known Java library for writing tests. More info, documentation and good practices available at https://junit.org/

A test class is a common Java class where methods:

- return void
- are annotated with `@Test`
- verify expectations

A common tests does the following:

- Get an instance of the class to test, plus relevant accessory classes
- Call the method to test with specific input
- Verify assertions

Unit testing: example

1. Example calculator

2. Example test class

```java
public class Calculator {

     public long add(int a, int b) = {
          return a + b;
     }

     public long subtract(int a, int b) = {
          return a - b;
     }
}
```

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    private Calculator c = new Calculator();

    @Test //each test should be annotated with this
    public void shouldSumTwoNumbers() {
      long sum = c.add(3,4);
      assertEquals("Sum is not as expected", 7, sum);
    }

    @Test
    public void shouldReturnOriginalValueIfAddZero() {
      long sum = calculator.add(3,0);
      assertEquals("Sum is not as expected", 3, sum);
    }

    [more tests here...]
}
```

Problem: Project Structure

- A Maven project, created from scratch, has a standard structure
- See also: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html
- Make sure your project has this structure (double check on GitHub)
- A good starting point is the already mentioned: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

```
/src
  /main
    /java // your code goes here
    …
    /resources
  /test  // your tests go here
    /java
    …
    /resources // your test files go here
pom.xml // your dependencies go here
```

Problem: Project compiles

- Make sure your project compiles:
  - `mvn clean package` within your project folder
- If it doesn't compile:
  - Read the compiler error, it should be relatively clear
- Check that your project supports Java 8
  - You can modify the following lines in the `<properties>` section of pom.xml
  - `<maven.compiler.source>1.8</maven.compiler.source>`
  - `<maven.compiler.target>1.8</maven.compiler.target>`

Problem: Hardcoding paths

- Some added *.txt* at the end of each input argument (I guess they might have got confused by the wording “text files” in the assignment)
- Some assumed that the input files are located in a fixed directory
- The above or similar assumptions make the application obscure and less portable!
- The arguments will either be:
  - Paths relative to the current directory
  - Full paths
- Solution: You should assume no prefix and no extension: the user will either specify the path of existing files or the app will fail

Problem: Non minimal imports

- Imports for Amazon Web Services:
  - `aws-java-sdk` brings in all JARs for all services of Amazon Web Services, but in Lab 1 we're only using S3 from AWS
  - `aws-java-sdk-s3` is the correct library to include
- Solution: include the correct dependency: you should see the number of dependencies reduce drastically (and the size of the shaded JAR consequently)