Introduction

In this lecture, we introduce the main themes of the course. We start by reviewing the concept of architecture neutral, then briefly describe the Java class as unit of programming, summarize the main features of Java, and conclude with a few examples, to give a taste of Java.

1.1 Design goals of Java programming language

The Java programming language was designed to fulfill the requirements of application development for a variety of network devices and embedded systems, in the context of heterogeneous distributed environments. The language is *imperative* and *object-oriented* with syntax similar to C++, but simple and elegant, removing all the unnecessary complexities of C++. The Java platform architecture is neutral and portable.

For example, to accommodate the diversity of devices and operating environments, the Java compiler generates an *architecture neutral* intermediate format called *bytecodes*.

Example of java bytecodes:

```
cafe babe 0000 0034 001d 0a00 0600 0f09
0010 0011 0800 120a 0013 0014 0700 1507
0016 0100 063c 696e 6974 3e01 0003 2829
```

Java is also a *statically typed* language, which means that every variable and expression has a type that is known at compile time. Java is a language that has only two data types: *primitives* and *references*. Furthermore, primitive data always have the same size, regardless of the underlying running platform. For instance, the *byte* data type is always an *8-bit signed two's complement* integer.

1.2 Classes and objects

In Java, a *class* is the fundamental unit of programming. It defines the template based on which objects can be built. Classes are discussed in detail in the next chapter. We assume the reader is familiar with object-oriented paradigm and understands basic concepts, such as class and object.

For example, in Java, a class could be defined using the Java keyword `class` followed by a user defined name (e.g. **`Empty`**) and open and closed curly brackets. If one types in an editor the following text and saves it in a file called **`Empty.java`**, one has the simplest Java program (*).

Introduction

```
class Empty {
}
```

To compile the Java code, one needs the **`javac`** compiler. Using a shell of the operating system, if one types

```
javac Empty.java
```

a new file is generated. The file will have the name: **`Empty.class`** This is the bytecodes file created by the Java compiler.

Empty.class file:

```
cafe babe 0000 0034 000d 0a00 0300 0a07
000b 0700 0c01 0006 3c69 6e69 743e 0100
0328 2956 0100 0443 6f64 6501 000f 4c69
6e65 4e75 6d62 6572 5461 626c 6501 000a
536f 7572 6365 4669 6c65 0100 0a45 6d70
7479 2e6a 6176 610c 0004 0005 0100 0545
6d70 7479 0100 106a 6176 612f 6c61 6e67
2f4f 626a 6563 7400 2000 0200 0300 0000
0000 0100 0000 0400 0500 0100 0600 0000
1d00 0100 0100 0000 052a b700 01b1 0000
0001 0007 0000 0006 0001 0000 0001 0001
0008 0000 0002 0009
```

To run a Java program one needs a Java Virtual Machine (JVM) called **`java`**. The JVM comes from the Java Development Kit (JDK) and needs to be installed on the machine one is using to develop and run Java programs. More on the installation of the Java platform in the next chapter. To execute the above Java program (*), within the operating system shell, one should type the following command:

```
java Empty
```

Since the `Empty` class is incomplete, the execution will generate the following error:

```
Error: Main method not found in class Empty, please define the main method as:
public static void main(String[] args)                              (**)
```

The error is self explanatory: we are missing the implementation of the **`main`** method, which is the entry point for the execution of any Java program.

1.3 Java platform features

To take full advantage of the power of the Java platform when building applications, three programming paradigms are discussed in this book: objective-oriented, functional, and reactive programming.

In the first part of this course, we will study the *language basics*. We will introduce the basic data types, starting with the built-in types; then we will discuss user defined type by studying:

- Classes and Interfaces (nested and inner classes) in Java
- Inheritance and Polymorphism
- Annotations
- Packages
- Generics
- Exception
- Threads
- Basic Java I/O

In the second part of this course, we will study fundamental *design patterns* in Java, such as creational, structural and behaviour design patterns:

- Singleton and Factory
- Adapter and Composite
- Observer and Visitor

Java as a *functional programming* language will be examined in the third part of the book by studying the Java Collection through:

- Functional Interfaces
- Lambda Expressions
- Streams

To build Java applications, we need to use specific Java frameworks. As an example, in the fourth part of this book, we will examine the *reactive programming* paradigm in Java by studying *RxJava* as a framework for writing asynchronous, concurrent, and resilient applications.

1.4 A taste of Java

Before we dive into all these concepts, to have a better understanding of where we are going, we shall illustrate some features of the language through small examples.

By tradition, the first program for any new programming language must print "Hello World!". Here is the Java version of such a program:

Hello World program

A program that prints "Hello World" is, by tradition, the first program that one develops when learning a new programming language. Here is our "Hello World" program in Java:

```
/*************************************************************************
 *  Compilation:  javac First.java
 *  Execution:    java First
 *
 *  Prints "Hello World!" to the terminal
 *
 *  > java First
 *  Hello World!
 *************************************************************************/

public class First {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Our program defines a class called **First** and implements the **main** method. The method is the entry point for the program execution. Therefore, the signature of the method cannot be changed. The keywords **static** and **public** are interchangeable, but they are mandatory. The only parameter of the **main** method is an array of strings whose elements are used as the command-line arguments for our program. Starting with Java version 5.0, one could pass an unspecified number of arguments to a method. Java treats the variable-length argument list as an array. Therefore, it is also correct to define the **main** method as:

```
public static void main(String ... args)
```

A class could have many overloaded **main** methods. However, if there is no **main** method with the signature described above, the interpreter will issue an error at runtime (see **).

Introduction

Filter string program

Now, let us consider an example of lambda expression in Java. There is a in **`java.util`** package a public interface called **`Predicate<T>`**. The interface represents a predicate (a boolean-valued function) of one argument. The functional interface has a functional method called **`test(Object)`**.

We will use this interface to filter a collection of type list of strings. The list is composed of programming language names. We want to print all the languages whose name length is greater than a given value.

The important method of class **`Filter`** is called **`filter`**. It takes two arguments: the *list* that needs to be filtered and the *predicate* based on which the filter is executed. The method returns the filtered list.

```
public static <T> List<T> filter(List<T> data, Predicate<T> predicate)
```

The main method creates the list of strings and invokes the filter method. The actual arguments are the **`languages`** as a list of strings and the **`predicate`** object implemented as a lambda expression:

```
filter(languages, s -> s.length() > 3)
```

Finally, the **`main`** method prints the result (i.e., the filtered list).

```
/*************************************************************************
 *  Compilation:  javac Filter.java
 *  Execution:    java Filter
 *
 *  Filter a list of strings
 *  The filter is a predicate defined by the condition string length > 3
 *
 *  For the list: "C++", "Java", "Python" the result of executing
 *
 *  > java Filter
 *  [Java Python]
 *************************************************************************/

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Filter {

    /**
     * Filter a data structure of type list
     *
     * @param  data The list that needs to be filtered
     * @param  predicate Functional interface
     *                  whose functional method is test(Object)
     * @return The filtered list
     */
    public static <T> List<T> filter(List<T> data, Predicate<T> predicate) {

        List<T> list = new ArrayList<>(0);

        for (T t : data)
            if (predicate.test(t))
                list.add(t);

        return list;
    }

    public static void main(String[] args) {

        // defines the list of strings
        List<String> languages = Arrays.asList("C++", "Java", "Python");

        // filter the list of strings on the predicate length > 3
        List<String> result = filter(languages, s -> s.length() > 3);

        // prints the filtered list
        System.out.println(result);
    }
}
```

The advantage of using lambda expressions is major. It allows us to pass not only values, but also behaviors (e.g., the predicate). This enables us to dramatically raise the abstraction level.

Filter stream program

As a last example, to illustrate the evolution of Java programming language, we will use `Streams` API. By defining `Streams` as `Monads`, Java becomes a language, where the functional paradigm becomes a reality. What the functional programming did to Java is it closed the gap between a program's "domain intent" (the problem we want to solve) and the computation performed to carry out the intent.

For instance, as an extension to the previous example, we would like to process a list of programming language names, so that we can print with capital letters, sorted alphabetically, all the programming language names that contain the letter "a."

```java
/**********************************************************************
 *  Compilation:  javac StreamFilter.java
 *  Execution:    java StreamFilter
 *
 *  Filter a list of strings, change selected string to uppercase,
 *  sort and print the string
 *
 *  For the list: "C++", "Java", "Python", "Haskell" the result of executing
 *
 *  > java StreamFilter
 *  HASKELL
 *  JAVA
 **********************************************************************/

import java.util.Arrays;
import java.util.List;

public class StreamFilter {

    public static void main(String[] args) {

        List<String> ls = Arrays.asList("C++", "Java", "Python", "Haskell");

        ls.stream()                               // convert list to stream
                .filter(s -> s.contains("a"))    // filter strings
                .map(String::toUpperCase)        // change them to uppercase
                .sorted()                        // sort them alphabetically
                .forEach(System.out::println);   // print the selected language
    }
}
```

1.5 Chapter resources

Java API documentation is the most important resource for Java developers. It is freely available from: http://docs.oracle.com/javase/8/docs/api/
More about the Java Platform, Standard Edition version 8, (the version available at the time of writing), can be found at http://docs.oracle.com/javase/8/

1.6 Exercises

1. Write a program that prints the command-line arguments.

   For example, if your program is `Exercise_1`, then when you run the program `java Exercise_1 Alice Bill`, your program must print `Alice Bill`.

2. Write a program that prints the command-line arguments in reverse order.

For example, if your program is `Exercise_2`, then when you run the program `java Exercise_2 Alice Bill`, your program must print `Bill Alice`.

3. Develop a static method called `start`. The method takes as an argument an integer and prints the character '*' as many times as the integer value indicates. It returns nothing (i.e., `void` type). Write a program that prints the following pattern:

```
*
**
***
****
*****
```

using the `start` method. Write java documentation for the `start` method.

4. The following program will be executed as `java Exercise_4 2 0 1 7`

```java
public class Exercise_4 {

    public static void main(String ... args) {
        for (String s:args)
            System.out.print(s + " ");
    }
}
```

What does it print?

a. Compilation error
b. It prints nothing
c. It prints 2 0 1 7

1.6 A challenging exercise

5. Write a program that prints only the command-line arguments that have only four characters and all four are digits.

For example, if your program is `Exercise_5`, then when you run the program `java Exercise_5 23455 1a34 34cd 9876`, your program must print `9876`.

Hints:
Use the methods `match` and `length` from the class `String`

`public boolean matches(String regex)`
Tells whether or not this string matches the given regular expression.
`public int length()`
`Returns the length of this string. The length is equal to the number of Unicode code units in the string.`

Example: `string.matches("[0-9]+")`