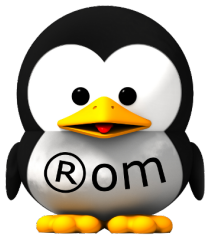
Java course

Jean-Paul COSAL



Section 1

Introduction

Who I am

Who you are

- Current project
- Previous experience with software development
- Expectations about this course?
- ...

Course Objectives

- Java Basics.
- Object Oriented Programming Basics
- Insights of work and processes in the Industry
- **Develop *interesting* skills**

Course Objectives

- Java Basics.
- Object Oriented Programming Basics
- Insights of work and processes in the Industry
- **Develop *interesting* skills**
  - Autonomy
  - Curiosity
  - *Discipline*
  - *Go further*
  - *Defend your cause*
  - ...
- *Be prepared to integrate the Industry*

Section 2

Day 1: Reminders

My first program

My first program

HelloWorld.java

My first program

HelloWorld.java

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

Java main features

- Object oriented language
- Portable
- Memory managed
- Comes with a rich development environment
- ...

Java basics

Variables and primitive types

byte, short, int, long, float, double, char, boolean

```
int myNumber; # Variable declaration
myNumber = 5; # Variable assignment

String s = "My String"; # Declaration and assignment
```

Conditionals

- Any expression that evaluates to `true` or `false`

```
boolean result;
result = 1 == 3; // a equal to b - false
result = 1 < 3; // true
result = 1 > 3; // false
```

- Compose expressions with `||` and `&&`
  - Important notice: **Lazy** evaluation
- Use `()` to enforce **precedence**

```
int a,b;
if (a > 1 && (a < 9 && b != 3)) /*do something*/;
```

Collections

A collection represents a group of objects

- *Vector* (1D array)

```
int[] a = new int[5];
a[0] = 5;
int[] b = {1, 2, 3};
```

- *Matrix* (nD array)

```
int[][] a = new int[2][2];
a[0][0] = 5;
int[][] b = { {1, 1}, {2, 2}};
```

Collections(cont.)

- Set and List

  Main interfaces of standard collections

```
boolean add(E e);
int size();
void clear();
Iterator<E> iterator();
...
```

Control structures

- if/else if/else
- switch
- for
- while
- do while
- for(each)

Functions

```
public int sum(int a, int b) {
    // Do something here
}
```

Mind:

- **The return value**
- **The arguments**

Exceptions

TBD

JUnit

TBD

Excercises

Introducing `Git`

- `SCM` main features
  - Backup
  - Keep track of code changes
  - Collaborative work
- Configuration Management main features

Excercises

Introducing `Git`

- `SCM` main features
  - Backup
  - Keep track of code changes
  - Collaborative work
- Configuration Management main features
  - Manage the lifecycle of a product
  - Version delivery management
- Git main feature
  - **distributed** `SCM`

Setting up `Git`

- Create an account on github.com
- Use preferably `ssh`
  - Create an ssh key (e.g. `ssh-keygen`)
  - Public key should be in `~/.ssh/id_rsa.pub`
  - Add ssh public key to your github account

Git cheat sheet

- `git clone <remote_repo>`
  - Get a working copy (*clone*) of a repository
- `git pull`
  - Synchronize, i.e. get changes, with remote repository
- `git push`
  - Publish your changes to remote repository
- `git branch <my_branch>`
  - Create branch `my_branch`
- `git checkout <my_branch>`
  - Switch to branch `my_branch`

Raw Java

- Mean function

$$\text{moy} : (x_1, x_2, \ldots, x_n) \mapsto \frac{1}{n} \sum_{i=1}^{n} x_i$$

  - Code
  - Compile (write a script)
  - Execute & test (write a script)

Introducing Eclipse

- Mean function
  - Create a project
  - Build & run
  - Unit test
- Matrix multiplication
  - `int[][] A = new int[m][n]`
  - `int[][] B = new int[n][o]`
  - `int[][] C = multiply(A, B)`

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \qquad i \in [1, m],\ j \in [1, o]$$

  - Implement and test

Using the Debugger

Main objective: **Inspect**

- Breakpoint
- Stepping
- Variables inspection

Section 3

Day 2: Object Oriented Programing

Before Object...

FIFO

Without using objects, implement a FIFO with the following properties:

- `int` elements
- `pop` function that return **and** remove the first element of the FIFO
- `push` function that add an element to the back (i.e. as last element) of the FIFO

Test the FIFO:

- Create a FIFO
- Add elements `{1,...,1000}` to the FIFO
- Remove the first 100 elements and check content

Modularity & Encapsulation

Modularity & Encapsulation

- Decompose a system into simpler subsystems to reduce overall complexity
- Module: subsystem **loosely** coupled to other subsystems.
- Encapsulation
  - technique used to favor subsystems' modularity
  - separate *interface* and *implementation*
  - data protection: data are not accessed directly, modules communicates via messages (methods)
  - responsibilities

Object

Object

An object is a module

An entity with:

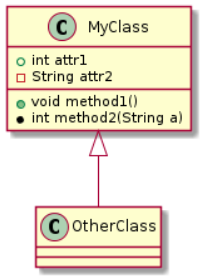
- data (state)
- functions/procedures (behaviour)

Class

Class

Type/structure (of an object) with:

- attributes
- methods



Visibility: public, protected, private

Constructor: initializes an object (instance)

Destructor: `finalize()`. cannot be called directly in `Java`

Class attributes/methods

Inheritance

Inheritance

A child class is a **specialization** of parent class.

- Access to parent members
- *enhancement* through new attributes and overrides
- Constructors
- Prevent class specialization with `final` keyword

Polymorphism

Polymorphism

- An instance belongs to its direct defining class and all its parent class

Abstract class and interface

- An *abstract class* is a class with (at least) one abstract (unimplemented) method.
- An *interface* lists a collection of *useable* methods (i.e contracts). A class **implements** an interface if it defines **all** the methods listed in the interface definition.

Object in a nutshell

Objects are used to **model** a problem with **concepts** relevant to what is to solve.

(Some) Golden rules:

- Analyse the problem to solve.
  - Understand what is at stake
  - Understand the domain
- Model the system with **relevant** concepts
  - e.g. Don't consider element of the *real world* irrelevant to the problem.
- Try to anticipate future changes...

Excercices

My first models with objects

- Cars
- Application handling Courses

Excercices

My first models with objects

- Cars
- Application handling Courses
- **HearthStone**

FIFO w/ objects

- Implement a FIFO using objects.
- **Test** your implementation.

Collections using interfaces

We want to be able to use FIFO, LIFO, LILO, ...

- Define an interface suiting the need above.
- Implement a FIFO, LIFO using implementing interface defined above.
- Test.

UML

Unified Modeling **Language**

13 diagrams (**structure** & **behavior**)

Class diagram

- Analysis and design of the static view of an application
- Describe responsibilities of a system
- The only diagram that maps directly mith oo languages
- Highlights
  - Meaningful class name
  - Relationships (e.g specialization, assoication, multiplicity...)
  - Favor clarity and keep only useful properties
  - Use notes when needed

Sequence diagram

- Used to visualize the sequence of calls in a system
- Highlights
  - Used to described the *workflow* of a complex functionality

State machine diagram

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

Use case diagram

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements and actors.

Case study: Bank agency

An bank agency needs an application to manage its customers' bank accounts.

- Define/precise the needs using a use case diagram.
- Model the application using a class diagram.
- Implement and test.

Section 4

Day 3: Reminders (cont.)

Generics

Why

- Factor/Re-use code with different input
- Benefit from type checking

Usage

- Declaring a generic type

```
class name<T1, T2, ..., Tn> { /* ... */ }
}
```

- Invoking and Instantiating a Generic Type

```
List<Integer> integerList;
```

Generic Methods

```
public class Util {
    public static <K, V> boolean compare(K p1, V p2) {
        // do compare
    }
}
```

Bounded Type Parameters

```
// A, B and C can be classes or interfaces
public class name<U extends A & B & C> {
}
```

Example

Container of updatable elements:

- If an element does not already **exist** in the container, it is added
- Otherwise, the copy in the container is **updated** with the info provided in the new copy

Threads and concurrency

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
– Javadoc

Using threads

Using threads

- Sub-classing `Thread` (and override the `run` method)

Using threads

- Sub-classing `Thread` (and override the `run` method)
- Implementing `Runnable`

Using threads

- Sub-classing `Thread` (and override the `run` method)
- Implementing `Runnable`
- Start a new Thread with:

Using threads

- Sub-classing `Thread` (and override the `run` method)
- Implementing `Runnable`
- Start a new Thread with:

Using threads

- Sub-classing `Thread` (and override the `run` method)
- Implementing `Runnable`
- Start a new Thread with:

```
// When sub-classing Thread
MyThread t = new MyThread(143);
t.start();

// When implementing runnable
new Thread(myRunnable).start();
```

Concurrency

Concurrency

- `synchronized` methods

Concurrency

- `synchronized` methods
- `synchronized` blocks

Concurrency

- `synchronized` methods
- `synchronized` blocks
- `wait()` and `notifyAll()`

Concurrency

- `synchronized` methods
- `synchronized` blocks
- `wait()` and `notifyAll()`

Concurrency

- `synchronized` methods
- `synchronized` blocks
- `wait()` and `notifyAll()`

Exercice: FIFOs

Implement a **synchronized** FIFO using an `ArrayList`, i.e. `add` and `remove` methods cannot be accessed simultaneously by multiple threads.

Swing

Introduction

Swing

Introduction

- Toolkits

Swing

Introduction

- Toolkits
- Base components (e.g. widgets, container, layouts)

Swing

Introduction

- Toolkits
- Base components (e.g. widgets, container, layouts)
- Event handlers

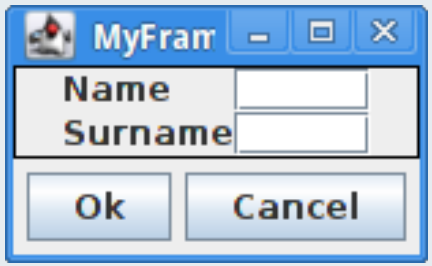
HelloSwing

HelloSwing

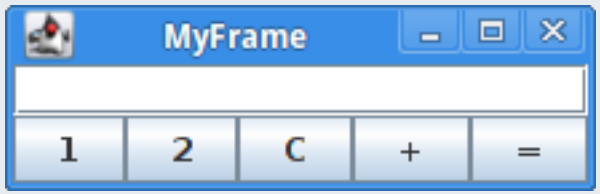
```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE
        );
        JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Layout basics



Case study: Calculator



Section 5

Day 4: Exam1

Section 6

Day 5: OOP (cont.)

The facts

- Requirements are **bound** to change and they **will**.
  - Taking into account **unforeseen** changes may be costly.
  - Software architecture and design must *strive* to **anticipate** changes
- *Design object-oriented software is hard.*
  - OOP is a *philosophy/framework*: no methods, some guidelines.
  - *Spaghetti* code.
  - Complexity grows with size. Maintainability decreases with complexity.

An answer

Design patterns aggregate all the experience from developpers who have been working on a **common** problem and offer a conceptual design solution.

Design patterns

Types

- Creational
- Structural
- Behavioral

Creational Patterns

Simple Factory

```
class SimpleFactory {
  public Product create(Object... createOptions);
}
```

Creational Patterns

Simple Factory

```
class SimpleFactory {
  public Product create(Object... createOptions);
}
```

Singleton

```
class Singleton {
  private static Singleton instance;
  public static Singleton getInstance() {
    return instance;
  }
}
```

Structural Patterns

Adapter

```
interface Target {
  targetRequest();
}
interface Source {
  sourceRequest();
}
class Adapter implements Target {
  private Source src;
  public targetRequest() {
    src.sourceRequest();
    // adapt result
  }
```

Behavioral Patterns

Observer

```
interface IObservable {
  void addObserver(Observer o);
  void notify();
}
interface Observer {
  void update();
}
```

Section 7

Day 6: Java 1.8 enhancements

Lambda

- A.k.a anonymous function
- Usage: ‘(arguments) -> {body}
- Example

```
x -> x * 2 // double
(x, y) -> x + y // sum
```

- Types of arguments and return value inferred (if omitted)
- Syntactic sugar only, no impact on runtime. Use with care.

Functional interface

- An interface with exactly **one** method
- Accepts lambda

- Example
  - Before Java 1.8

```
Thread t = new Thread(new Runnable() {
@Override
public void run(){
System.out.println("Running");
}
});
```

  - With Java 1.8

```
Thread t = new Thread(()->System.out.println("Running")
```

Stream API

- Enables filter/map/reduce like operations on collections.
- Example

```
int[] list = {1,5,8,7,6,9,4,2,3};
IntStream stream = Arrays.stream(list);
stream
   .map(i->2*i) // map
   .sum(); // reduce/collect
```