A Java introduction

SDK, FC

UMR AMAP

15/12/2009

Plan

History

- James Gosling and Sun Microsystems
- Java, May 20, 1995
- Java 1 $\longrightarrow$ Java 6
- GPL with classpath exception since 2006

Characteristics

- Object Oriented
- Interpreted (need a virtual machine)
- Portable (Linux, Mac, Windows)
- Dynamic (introspection)
- Static typing (checks during compilation)
- Simplier than C++ (memory management, pointers, headers...)

Programming environment

Java Environnment

- JRE (Java runtime environment)
- JDK (Java Development Kit)
- Jave SE (Standard edition)
- Java EE (Enterprise edition $\rightarrow$ Web)
- Java ME (Micro edition)

IDE - Editor

- Eclipse, NetBeans : Full IDE (completion, compilation...)
- Simple Editors : Notepad++, TextPad, Scite (coloration, indentation)

Installation

Windows

- Download and install the last JDK (Java SE 6)
- Environment variable (Computer → properties)
  - Add to PATH variable the `bin` directory
  - eg : "C :/Program Files/Java/jdk1.6.0_version/bin"
- Install editor TextPad or Notepad++

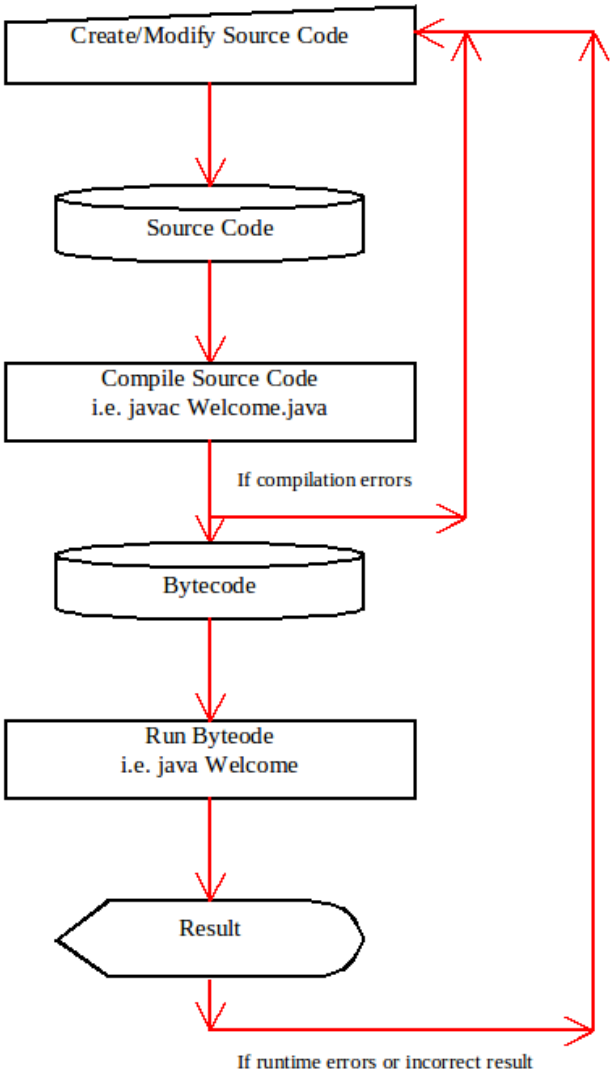
Linux

- `sudo apt-get install sun-java6-jdk`
- `sudo apt-get remove openjdk-6-jdk`
- Editor : use scite or gedit

Java application

- Java programs written in files with extension **".java"**
- Applications are **.java** files with **public static void main(...)** method
- Compile and run Java application
  - Run the compiler on a **.java** file : `javac package/MyProgram.java`
  - $\Rightarrow$ Produce a file of Java byte code : **MyProgram.class**
  - Run the interperter on a **.class** file : `java package.MyProgram`
- the tools **javac java** are part of the *JDK*

Workflow


Create/Modify Source Code
Source Code
Compile Source Code
i.e. javac Welcome.java
If compilation errors
Bytecode
Run Byteode
i.e. java Welcome
Result
If runtime errors or incorrect result

Hello world

```java
package hello;


/**
 * Application Hello world
 */
public class HelloWorld {

  /** Main Funcion */
  static public void main(String[] args) {

    // print to screen
    System.out.println("Hello world !");

  }
}
```

- Print to console : System.out.println(“a string”);
- Commands finish with a **;**
- Comments : // or /* */
- Package : hello = directory
- `java hello.HelloWorld`

Create a new application

- Copy directory
- Rename directory and file :
  - **class Name and file name should be identical**
  - The package is a namespace and corresponds to the directories
  - **package name and directory should be identical**

Exercice : Create a new application “package1.App1”

Variable & Simple type

Variable

- A variable has a type and hold a **value** or an **Object**
- A variable name starts with lowercase, eg : myVariable ;

- **boolean** : true or false
- **int** : signed integer 32 bits
- **long** : signed integer 64 bits
- **float** : floating point 32 bits
- **double** : floating point 64 bits
- **Object** : a generic object

initialization

- by default : 0
- int aVariable = 2 ;
- float anOtherVariable = 3.56f ;
- Use **final** for constants : final float CONSTANT = 0.1f ;

Operators

Arithmetic

- +, -, *, /, %
- myInt++, otherInt−− ;
- +=, -=, *=, /= ...
- Precedence : use parenthesis ()

division

- 2 / 3 different of 2.0 / 3.0
- Division par zero → Infinity or NaN

Exercise : arithmetic.DivideByZero

Math

```
import java.lang.Math
```

- Constante : Math.PI, Math.E
- Math.abs(), Math.pow(), Math.sqrt(), Math.exp(), Math.log(),
- Math.cos(), Math.tan()...
- Math.toDegrees(), Math.toRadians()

Javadoc : Java Math

Exercice : Calculate triangle hypothenus with pythagore

Simple Arrays

- One or 2 dimensions arrays
- Dynamic allocation : **new** keyword
- **null** if not initialized
- Not resizable
- Access with [] operator
- Index begin at 0

```
String[] array1a = new String[12];
String[] array1b = {"toto", "tata", "titi"};

int size = 4;
double[] array1c = new double[size];
double[][] array2 = new double[4][6];

array1a[11] = "a␣string";
array2[0][2] = 3d;
array2[3][5] = 1d;
System.out.println(array2[4][6]); // index error
```

Error during execution

- Try to use an object not initialized
- Try to acces to a non exisiting element in array

```
double[] array = null;
System.out.println(array.length);
array = new double[10];
System.out.println(array[10]);
```

- Exceptions are raised and stop the program

Exceptions

- Error can be catched anywhere in the program
- Specific exceptions can be raised

```
double[] array = null;

try {
  System.out.println(array.length);

} catch (Exception e) {

  System.err.println(e);
}


try {
  double result = array[10];

} catch(Exception e) {
  result = 0.;
}

System.out.println("Result : " + result);

// if(result < 0) { throw new Exception("result cannot be negative"; }
```

Complex types

- Java manipulates **Objects**
- An object is a concept (e.g a particular data structure)
- Objects are instantiated with the keyword **new**
- A variable contains a **reference** to an object
- Affectation is a reference copy (the variable point to the same object)
- Objects are destroyed when there is no reference on it
- By default an object variable is set to **null**
- Objects have properties and methods accessible with the **.** operator

```
MyObject o1, o2; // null;
o1 = new MyObject();
o1.value;
o1.sum();
o2 = o1;
o1 = null;
o2 = null; // Object will be destroyed by the garbage collector
```

Collections

- Collections are dynamic containers
- Associated to a specific type
- Specific behaviour : list, set, map..
- Simple types (int, float, double...) are not objects ⇒ need a wrapper
- int ⇒ Integer
- float ⇒ Float
- double ⇒ Double
- **Collection class** has method to sort, shuffle, reverse, ... collections

See javadoc : Collection java 1.6

```
Collection<String> c1;
Collection<Integer> c2;
Collection<Double> c2;
```

ArrayList

- An **ordered** collection of variable size
- Growth automatically
- **get**, **add** and **set** method
- Insertion is slow
- For a stack or a queue, use **LinkedList**

```
List<String> l = new ArrayList<String>();
l.add("toto");
l.add("tata");
l.add("titi", 1); // insert at index 1
l.set(1, "tutu"); // replace at index 1

l.size(); // 3
l.get(0); // "toto"
...
```

HashSet

- An **unordered** collection **without duplicated** elements
- **contains** method is fast

```
Set<String> s = new HashSet<String>();
s.add("toto");
s.add("tata");
s.add("titi");
s.add("toto"); // false

s.size(); // 3
s.contains("tutu"); // false
...
```

HashMap

- Map **associate** a key with an object
- a key should be unique (like in a Set)

```
Map<String, Color> m = new HashMap<String, Color>();
m.put("black", new Color(0, 0, 0));
m.put("red", new Color(1, 0, 0));
m.put("green", new Color(0, 1, 0));
m.put("blue", new Color(0, 0, 1));

m.get("red"); // return a color object
m.containsKey("black"); // true
m.keySet(); // set of keys
```

Boolean arithmetic

- boolean v = true ;
- ! && ||
- <= >= < > == !=

- == and **equals()** : reference vs contents
- != and **!equals()**
- use () for precedence

if else

- Simple condition
- Can be combined

```
// simple if
if(condition) {
   block
}

// complex if
if (condition) {
   block1
} else if(othercondition) {
   block2
} else {
   block3
}


if(v1 && !v2) {
    System.out.println("v1 and not v2");
}
```

while

- Loop with condition

```
int counter = 0;
while(counter < 100 ) {
    System.out.println("counter : " + counter);
    counter++;
}

int counter2 = 0;
int sum = 0;
while(counter2 < 100) {
  sum += Math.random();
  if( sum > 20) { break; }

  counter += 1;
}
```

for

- Loop with index

```
// With an array
int[] array = new array[12];

int sum = 0;
for(int i=0; i<array.length; i++) {
  sum += array[i];
}
```

- Loop in a collection

```
// With a list
List<Integer> l = new ArrayList<Integer>();
int sum = 0;
for(Integer v : l) {
  sum += v;
}

// With a Map
Map<String, Object> m = new HashMap<String, Object>();
for(String key : m.keySet()) {
  System.out.println(m.get(key));
}
```

Exercice : do the sum of a double array

Enumeration

- A **enumeration** is a type with a limited number of value
- Type checking
- Better than using integer or constante values

```
// Type declaration
enum StopLight {red, amber, green};

// Use type
StopLigth s;
```

static Functions

- Like a mathematical function
- $\Rightarrow$ **Reuse a block of code**
- Inputs Parameters : type + variable name
- Output : return type

```
static double aFunction(double param1, double param2) {
  return param1 * param1 + param2;
}
```

- param1 and param2 are the *parameters*
- their names have a local scope : they are only available in the function

```
double a = 2;
double b = 3;
double result = aFunction(a, b); // aFunction(2,3)
System.out.println(result);
```

Functions (2)

- a class can contain several functions
- if no parameters ⇒ use **()**
- if no return type ⇒ use **void**

```
public class MyClass {

  public static double function1(double param1, double param2) {
    ...
  }

  public static void function2() {
    ...
  }


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

Read a file

- **Scanner** tool parses a list of token

```
import java.util.Scanner;

...
// in a function
Scanner sc = new Scanner(new File("filename.txt"));

while(sc.hasNext()) {

  String s = sc.next(); // read a string
  int i = sc.nextInt(); // read an int
  float f = sc.nextFloat(); // read a float

}

// Read in the shell
Scanner sc2 = new Scanner(System.in);
System.out.print("Input a string : ");
String s = sc2.next();
```

Exercices

- Create a file with a list of number : *numbers.txt*
- Create a application which read the file in an **ArrayList of Float**

  `ArrayList<Float>`
- Compute the mean (and the variance) of the data
- Put the code in a function
- Use args argument in *main* to pass the filename in the command line
- Use the function to compute the mean for multiple files
- Try to put strings in the file and recover it
- Remove minimal and maximal values in the mean
- ...

Object Oriented programming

- Java manipulates Objects
- E.g : a list, a number, a car, a species, a plant, a plot...

Concepts

- **Class** : a type of object
- **Instance** : a real object
- **Instance data** : the data an object contains
- Data are different for each instanciated object
- **Methods** : the functions associated to an object

How to use Object

- Object are instanciated with the **new** keyword
- We can **instantiate several objects** of the same type

```
Tree t1 = new Tree("Spruce");// initialization with parameter
Tree t2 = new Tree("Larch");
Tree t3 = new Tree("Fir");

// each instance has its own data
t1.getSpeciesName(); // "Spruce"
t2.getSpeciesName(); // "Larch"
t3.getSpeciesName(); // "Fir"

// Properties
t3.height = 2;
System.out.println(t1.height);

// Methods
t2.setSpeciesName("Spruce");

// t2 != t1 but t2.equals(t1)

// affectation
t3 = t1; // t3 and t1 point to the same object t3 == t1
t3.getSpeciesName(); // Spruce
```

Class

- A class is a type description
- It contains data and methods (members)

```
public class Tree {

  public double height;
  private String speciesName;

  // Constructors
  public Tree(String n) {
    speciesName = n;
  }

  // Accessors
  public void setSpeciesName(String n) {
    speciesName = n;
  }
  public String getSpeciesName() { return speciesName; }
}
```

Class...

Access control

- **Public** : no access restriction
- **Private** : only accessible in the class method
- **Protected** : accessible in the class methods and in the subclasses

Constructor

- Called when the object is created
- Can take parameters or not
- Can be declined in different versions
- No return type

Exercices

tree/Tree.java

- Add a private data : *diameter*
- Add corresponding accessor
- Add a method which return the volume of the trunk (we consider a cylinder)
- Create an application which use this class and this function

This

- **this** keyword represents the current object
- Usefull if there is ambiguity with the names

```
public class Tree {

  ...
  /** Volume of the trunk */
  public double getVolume() {
    double volume;
    double radius = this.diameter / 2d;
    volume = radius * radius * 2d * Math.PI * this.height;

    return volume;
  };
  ...
}
```

static

- **static** members are associated to the type
- **static** datas are shared by all instance
- **static** methods are not attached to an instance

```
class Tree {
  ...
  static public int shared

  static public double getSurface(double diameter) {
    double radius = diameter / 2d;
    double surface = radius * radius * 2d * Math.PI;

    return surface;
  }
  ...
}

Tree.getSurface(2.)
Tree.shared
```

Inheritance

- We can reuse a class to make more specific classes
- e.g : a tree with crown, a tree with leaves, etc...
- Inheritance corresponds to a **“is a”** relation
- A sub-class has all the data and methods of its parent
- All class inherit from the class **Object**
- Multiple inheritance is not allowed

Inheritance

```
// Tree.java
class Tree { // extends Object

  protected double height;

  /** Constructor */
  public Tree(double h) {
    height = h;
  }

  public double getHeight();

}

// CrownTree.java
class CrownTree extends Tree {

  protected double crownHeigth;
  protected double crownDiameter;

  /** Constructor */
  public CrownTree(double h, double ch, double cd) {
    super(h); // parent constructor
    crownHeigth = ch;
    crownDiameter = cd;

  }

  public double getCrownVolume() {
    ...

  }
```

instanceof

- All class inherit from **Object**
- **instanceof** tests the type of an object

```
Tree t = new Tree(10d);
CrownTree ct = new CrownTree(10d, 4d, 4d);

ct instanceof CrownTree; // true
ct instanceof Tree; // true
ct instanceof Object; // true

t instanceof Tree; // true
t instanceof CrownTree; // false
```

Cast

- We can use the parent type in the code ("is a" relation)
- we use a cast, when we have to change the type to a subtype
- a cast can fail

```
Tree t1 = new Tree(10d);
Tree t2 = new CrownTree(10d, 4d, 4d);

t1.getCrownVolume(); // fail : not a crowntree
t2.getCrownVolume(); // fail : java think it is a Tree

CrownTree ct1 = (CrownTree) t2; //ok
ct1.getCrownVolume(); // ok

CrownTree ct1 = (CrownTree) t1; //fail : t1 is not a CrownTree
```

Override

- Method can be overriden in sub-class
- Use annotation *@Override*

```
class Tree {

  /** Tree growth */
  public void growth(int nbYear) {
    diameter += nbYear * dbhInc;
  }

}

/** Tree with crown */
class CrownTree extends Tree {

  /** Tree growth */
  public void growth(int nbYear) {
    super.growth(nbYear);
    crownDiameter += nbYear * crownInc;
  }
}
```

Interface

- An **interface** is a kind of contract
- A class can implement several interface (multiple inheritance)
- An interface do not provide any implementation
- An interface is a type

```
// Growthable.java
interface Growthable {

  public void growth(int nbyear);
}

// Tree.java
class Tree implements Growthable, Cloneable {

  public void growth(int nbyear) {
    // do growth
  }

  public Object clone() {
    // do clone...
  }
}

...
Growthable g = new Tree(...);
g.growth(10);
g.getDiameter(); // error : java don't know if it is a Tree
```

Abstract class

- An **abstract class** is an incomplete class
- It cannot be instantiated
- Like an interface but with some implementation
- Can be useful to share common methods in an inheritance graph

```
class abstract AbstractTree {

  private double height;
  private double diameter;

  public double getVolume() {

    double volume;
    double radius = this.diameter / 2d;
    return = radius * radius * 2d * Math.PI * this.height;

  }

  abstract int getNbLeaves();

}
```

Abstract class

- Each sub-class implements abstract methods

```
class TreeWithNbLeaves extends AbstractTree {

  private int nbLeaves;

  @Override
  int getNbLeaves() {
    return nbLeaves;
  }


}


class TreeWithListOfLeaves extends AbstractTree {

  private List<Leaf> leaves;

  @Override
  int getNbLeaves() {
    return leaves.size();
  }
}
```

Polymorphism

- **Polymorphism** means that we can use an object without knowing its exact type
- The correct function will be called

```
List<AbstractTree> l = new ArrayList<AbstractTree>();

l.add(new TreeWithNbLeaves());
l.add(new TreeWithListOfLeaves());

for(AbstractTree t : l) {
  System.out.println("nb␣leaves␣:␣" + t.getNbLeaves());
}
```

Exercices

In the tree package

- Create the sub-class *CrownTree*
- Add a method to compule the volume of the CrownTree (use @Override)
- Create 2 sub-classes *SphericTree* and *ConicTree*
- Override the volume function
- Create a class Forest containing a list of trees
- Add function to build a random forest
- Compute the volume of the forest

Java library

See javadoc : http ://java.sun.com/javase/6/docs/api/

- GUI
- Math
- Data Structure (Collections)
- Input Output
- Networking
- MultiThreading
- Database
- Intropection

Apache commons libraries

See web : http ://commons.apache.org/

- Math
- IO
- Loggin
- CLI
- Collections
- ...

Some links

- Sun's tutorials : http ://java.sun.com/docs/books/tutorial/
- Coding conventions :
  http ://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- Resources on Capsis web site : http ://capsis.cirad.fr
- Millions of books. . .