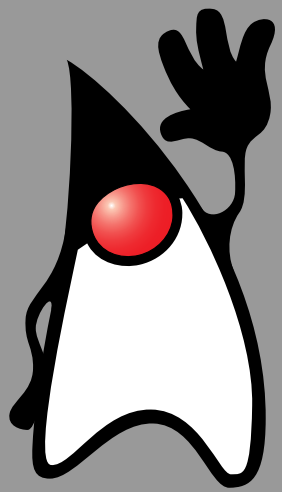
EPITA Première Année Cycle Ingénieur
Atelier Java - J1

Marwan Burelle

marwan.burelle@lse.epita.fr
http://www.lse.epita.fr



Plan

Plan

Plan

Plan

A Bit of History

- **1991:** creation of the JAVA project by James Gosling of SUN.
- **1995:** First public implementation (v1.0).
- **2006:** Java SE 6 first release (also known as Java 1.6), the actual plateform specification.
- Java's philosophy: *Write Once, Run Anywhere* (WORA)
- Goals: a Virtual Machine and an Object Oriented Language with a C/C++ like syntax.
- From November 2006 to may 2007 SUN moves Java to Open Source.

Present

- Oracle acquire Sun during 2009/2010 (provoking Gosling's departure)
- Java is one of the most used programming language and plateform nowadays
- Java is central in Google's Android (more than 50% of smartphones in third quarter of 2011 and growing fast)
- Java represents an important part of web-oriented applications (Java EE framework)

Java's situation

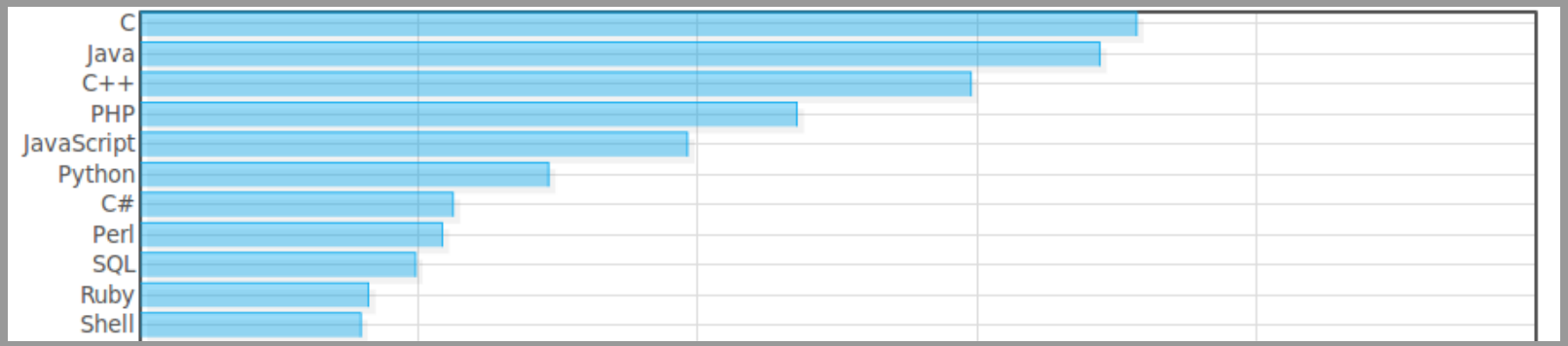


Figure: Programming Language Popularity - [1]

2012/04	Programming Language	Ratings
1	C	17,56%
2	Java	17,03%
3	C++	8,90%
TIOBE Programming Community Index for April 2012		

Figure: http://www.tiobe.com - [2]

Primary Goals

There were five primary goals in the creation of the Java language:

1. It should use the **object-oriented** programming methodology.
2. It should allow the same program to be executed on **multiple operating systems**.
3. It should contain built-in support for using computer **networks**.
4. It should be designed to execute code from **remote sources** securely.
5. It should be **easy to use** by selecting what were considered the good parts of other object-oriented languages.

Plan

Bytecode Compiling

The major innovation of Java (in the first place) was the bytecode compilation model and its Virtual Machine.

- Keep the whole framework *portable* across different platforms;
- Allows a quick widespread;
- Optimizations focus upon one *virtual architecture*;
- Introduce the notion of *code mobility*;

Write Once, Run Anywhere

- The JVM was intend to be platform independant;
- Normally, a program compiled for the JVM will run on any valid JVM on any OS on any CPU;
- *Mobile Code:* applications can send bytecode (classes or complete programs) to any working JVM;
- *Mobile Code classic example:* browser applet (classes executed inside a browser;)
- A lot of hardware configuration tools (such as *Line6 POD* utilities) benefits from WORA (such software tools can run on any computer with a JVM;)

Plan

Common Usage

- Applets: portable mobile code loaded into browser;
- Small applications with quick dev cycle and a high portability requirement;
- Hardware configuration tools or any kind of software provided with specific gears;
- Smooth and fast GUI oriented app (often free gadgets distributed over the net with an applet version;)
- Web services: the web server acts as a service provider, each request triggers code executions on the server (servelet model;)
- Modern Cell phones run application written in Java;
- The **Android** framework (**Google** phone OS) relies on Java for user apps.

Applets

- An applet is a small class with a limited environment;
- The bytecode only requires a JVM (and some classes;)
- The local security is enforced via a sandbox model;
- The applet have access to a limited set of local resources (sandbox) and can callback only its original provider;
- Applets offer more interactivity and expressivity than basic web pages (html and javascript;)

Web Programming

- Java describe a complete framework for “entreprise” applications (especially web applications)
- The whole platform defines a **multi-tiers, component-based, scalable and portable** applications ecosystem
- Java EE provides:
  - Network
  - Webservices infrastructures
  - XML integration
  - Page scripting and processing (JSP . . .)

Plan

2 Java OOP
- Everything is Object
- Everything in Object
- Classes
- Inheritance and Subtyping
- Late Binding
- Visibility Modifiers
- Interfaces
- Living without multiple-inheritance

Basic Programming

- Java's syntax is very similar to C/C++ syntax;
- Java offers some basic types (integer, float, boolean ...) which can be used directly (unboxed) or through object oriented paradigms (boxed;)
- Arrays are the only structured types outside objects (but they're also objects ...;)
- Java offers an exception mechanism (throw and catch;)

Example:

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

Plan

Example:

```
public class PlayWithInt {
  private int x;
  private Integer y;
  public PlayWithInt(int z) {
    y = new Integer (z);
    x = z;
  }
  public void print(){
    System.out.println(y.toString());
    System.out.println(Integer.toString(x));
  }
  public static void main(String[] args){
    PlayWithInt o = new PlayWithInt(1);
    o.print();
  }
}
```

Example:

```
public class PlayWithInt {

  // ...

  public void times(int n){
    y = y * n; // object or not ?
    x *= n;    // C like shortcut
  }

  // ...

}
```

Plan

Everything in Object

- No Code outside classes;
- Methods (static or not) are completely defined in classes;
- A Java file (`.java`) defines **one** public class, other classes in the file are private classes only visible from the public one;

Example:

```
public class PlayWithInt {
  // ...
}
private class IntBox {
  private int x;
  public IntBox(int z){x = z;}
}
```

Entering Into The Program

- Most programming languages offer a unique entry point (like the `main()` function in C or the `BEGIN ... END.` block in Pascal;)
- Some other languages executed any code inside the program (like OCaml;)
- In Java, the entry point is defined by the class given as parameter to the JVM;
- Any class can have a class method `main`, and thus can serve as entry point of the program;
- The `main` method exists without creating any instance of the class, if needed, such an instance should be explicitly created (using `new` constructor) inside the method;

Entry Point

Example:

```
public class HelloWorld {
  // Our Entry Point !
  public static void main(String[] args){
    // Do some clever things ...
    System.out.println("Hello World!");
  }
}
```

Plan

Class Definitions

Example:

```
// A public class
public class MyClasses {
  // Some properties
  private int some_int;
  private String a_string;
  // Constructors
  // Whithout parameters
  public MyClasses() {
    some_int = 0;
    a_string = "";
  }
  // Overloaded Constructors
  public MyClasses(int x, String s){
    some_int = x;
    a_string = s;
  }
}
```

Class Fields and Methods

- Methods and properties can exist without any object;
- These methods and properties are marked as `static`;
- `static` methods can be accessed directly without instantiating an object;
- `static` properties are shared among the class and its instance;
- The `main` method is an example of class method;

Class Fields and Methods

Example:

```
public class MyClass2 {
  public static int x = 0;
  public int y;
  public MyClass2(int z) { y = z; }
  public static void setX(int z){ x = z; }
  public static void main(String[] args){
    MyClass2 o = new MyClass2(1);
    System.out.println("x = "
      + Integer.toString(MyClass2.x)
      + " y = " + Integer.toString(o.y));
    MyClass2.setX(2);
    System.out.println("x = "
      + Integer.toString(o.x)
      + " y = " + Integer.toString(o.y));
  }
}
```

Class Fields and Methods

Example:

```
x = 0 y = 1
x = 2 y = 1
```

`MyClass2.x` refers property `x` in the class, thus `MyClass2.x` and `o.x` are the same *entity*.

Plan

Inheritance

- The inherited fields can be used directly, just like any other fields, provides that the field visibility authorises it.
- You can declare a field in the subclass with the same name as the one in the super-class, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the super-class.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the super-class, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the super-class, thus hiding it.
- You can declare new methods in the subclass that are not in the super-class.
- You can write a subclass constructor that invokes the constructor of the super-class, either implicitly or by using the keyword `super`.

Inheritance

Example:

```
public class Animal {
  public Integer members;
  public String scream;
  public Animal(int m, String s) {
    members = new Integer(m);
    scream  = new String(s);
  }
  public String toString() {
    return ("Your animal has "
      + members.toString()
      + " members and "
      + scream);
  }
}
```

Example:

```
public class Dog extends Animal {
  // A new fields
  private String breed;
  // The new constructor
  public Dog(String r){
    // We're using the parent constructor
    super(4,"barks");
    breed = new String(r);
  }
  // We override the toString method
  public String toString(){
    return ("Your dog is a "+breed);
  }
}
```

Subtyping (so-called polymorphism)

- Inheritance provide a form of **subtyping**: if `MyClass2` is a subclass of `MyClass` thus instances of `MyClass2` can be used where an instance of `MyClass` is awaited;
- Subtyping can be seen as way a of describing object by their behaviour: this object can be use here because it provides these methods;
- Interfaces extends the notion of subtyping (when a class implements an interface, it can be seen as of type of this interface;)
- In OOP, subtyping is often referred to as **polymorphism** while it does not correspond to ML nor System F like polymorphism;
- Real polymorphism (precisely F-bounded polymorphism) can be found using Generics (see later;)

Java Object Hierarchy

- Every classes implicitly extends the `Object` class.
- Since Java provides only simple inheritance, the set of relations of the classes defines a tree hierarchy.
- The `Object` class act as a *any* type, that is, every objects can be seen (and thus used) as an instance of `Object` class.
- `Object` class is (was thanks to *Generics*) often used like the `void*` type in C for unsafe generic containers.

Example:

```
class Box {
  private Object          content;
  public Object get() { return content; }
  public void set(Object o) { content = o; }
}
```

Plan

Method Resolution

- Method are defined in classes but are called from object (instance;)
- **Late Binding:** the method is selected dynamically:
  - If the method is not overridden or if the class of the object is not ambiguous, everything happen as usual;
  - When the method is overridden and the known class of the object (at compile time) was the parent, thanks to late binding, the executed method is the method of the instance rather than the method of the known class;

Late Binding

Example:

```
public static void printAnimal(Animal a){
  System.out.println(a.toString());
}
public static void main(String[] args){
  Animal a = new Animal(4,"meows");
  Dog    d = new Dog("Chihuahua");
  printAnimal(a);
  printAnimal(d);
}
```

```
Your animal has 4 members and meows
Your dog is a Chihuahua
```

Plan

Visibility

Methods and fields can have an access modifiers:

Access modifiers

Modifier	Class	Package	Subclass	World
`public`	yes	yes	yes	yes
`protected`	yes	yes	yes	no
none (default)	yes	yes	no	no
`private`	yes	no	no	no

Plan

Notions of interfaces

- Interfaces describe what an object must provide without describing how.
- It extends the types name strategy to provide features similar to structural typing:
  - To implements an interface object must provide the described methods.
  - Interface describe objects by their features rather than their class.
  - As input type, interfaces describe which methods are expected.
- Interfaces are useful tools of OOP (see section on containers.)

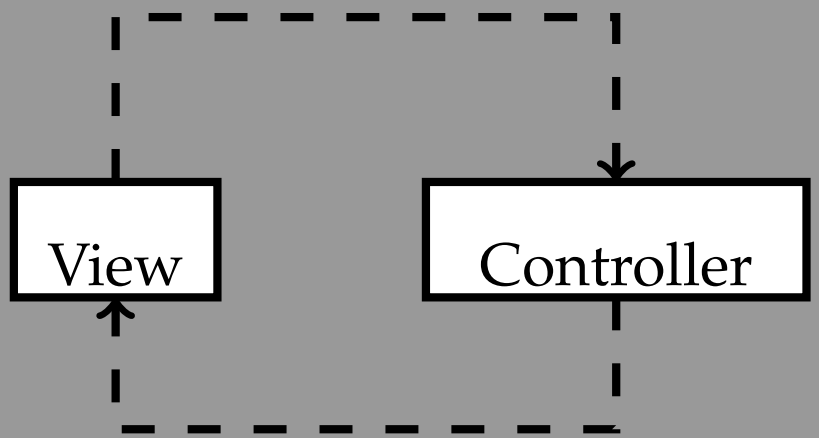
Interface syntax

```
public interface Printable {
    public void print();
}

public class PrintablePoint implements Printable {
    private int x;
    private int y;
    public PrintablePoint(int _x, int _y){
        x = _x;
        y = _y;
    }
    public void print(){
        System.out.println("("+x+","+y+")");
    }
}
```

Breaking circular dependencies

- Suppose you have two classes: `View` and `Controller`
- Each depends on the other (a `View` instance embeded a `Controller` instance which in turns embeded the previous `View` instance.)
- The two declaration depends on the other !



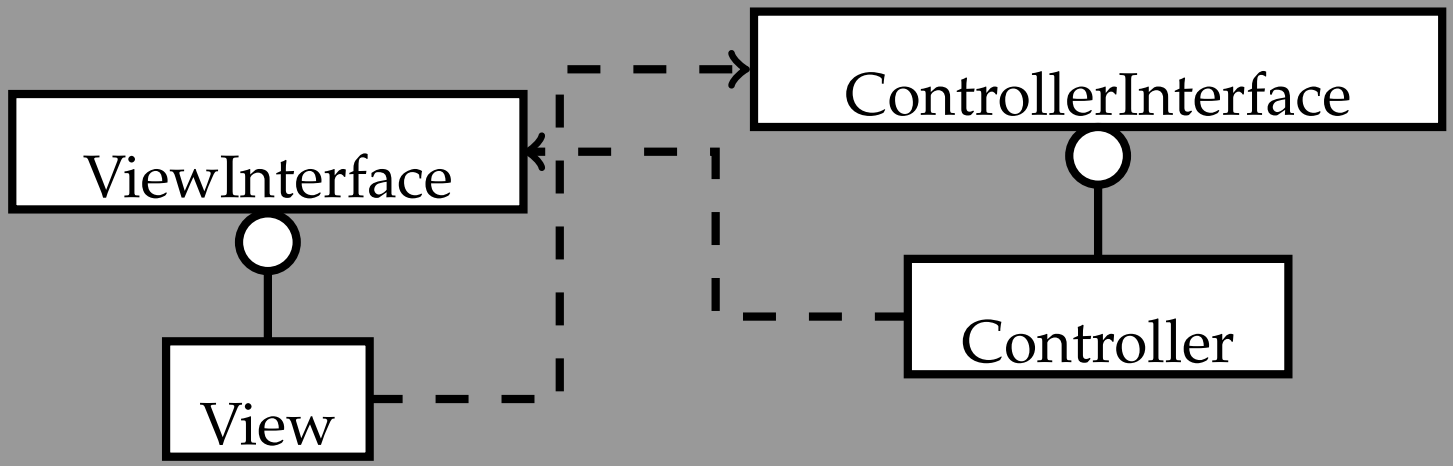
Breaking circular dependencies

```
public class View {
    private Controller engine;
    public View() {
        engine = new Controller(this);
    }
    /* ... */
}

public class Controller {
    private View gui;
    public Controller(View g) {
        gui = g;
    }
    /* ... */
}
```

Breaking circular dependencies

- **Using interfaces solves this issue quiet easily.**



Breaking circular dependencies

```
public class Controller implements ControllerInterface
{
    private ViewInterface gui;
    public Controller(){
        /* ... */
    }
    public void registerView(ViewInterface g)
    { gui = g; }
    /* ... */
}

public class View implements ViewInterface
{
    private ControllerInterface engine;
    public View(ControllerInterface e) {
        engine = e;
        e.registerView(this);
    }
    /* ... */
}
```

An other example of interface

- Java doesn't provide parameters decorations and methods' calls is always done *by address* since object are pointers.
- We may want to enforce immutability on an object passed as parameter, we will use interface for that.
- The strategy is quite simple: you just build an interface without the methods that can modify the object state.
- Your class is declared as *implementing* your restricted interface.
- All methods using object of your class, but that should not modify it, will use the interface and not class directly.

Constant Interfaces ...

```
// The interface
public interface ConstBox {
    public int get();
}

// The class
public class Box implements ConstBox {
    private int          data;
    public int get() { return data; }
    public void set(int x) { data = x; }
    Box(int x) { data = x; }
}
```

Constant Interfaces ...

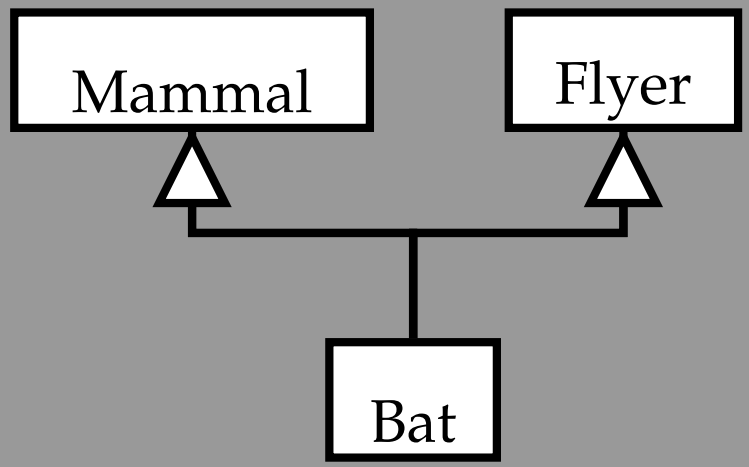
```
// Using ...
public class ExampleConstBox {
    public static int fact(ConstBox b) {
        int                 n = b.get(), res = 1;
        for (int i=2; i<=n; ++i) res *= i;
        // ERROR: b.set not defined
        b.set(res);
        return res;
    }
    public static void main(String[] args) {
        Box                 b = new Box(5);
        b.set(ExampleConstBox.fact(b));
        System.out.println("fact(5)=" + b.get());
    }
}
```

Plan

2 Java OOP
- Everything is Object
- Everything in Object
- Classes
- Inheritance and Subtyping
- Late Binding
- Visibility Modifiers
- Interfaces
- Living without multiple-inheritance

Good usage of multiple-inheritance

- Multiple-Inheritance offers the possibility to compose classes from different small classes.
- If we have a pool of small classes representing aspect and/or features, we can compose then to create more complex objects.
- Implementing a `Bat` class with `Mammal` and `Flyer` is quite straightforward:



Using aggregation

- One can *simulate* multiple-inheritance using aggregation.
- When inheritance is used to add *features*, it is quiet simple to *embeded* another class instance.

```
public class Bat extends Mammal {
    // integrate a Flyer into a Mammal
    Flyer        innerFlyer = new Flyer();

    public void flyTo(int x, int y){
        // wraper for the original Flyer's method
        innerFlyer.flyTo(x,y);
    }
}
```

Using interface and aggregation

- The previous `Bat` class is not a `Flyer` (while it offers the same methods)
- To enforce the fact that a `Bat` is a flying mammal, we should use interface:

```
public interface Flyable {
    public void flyTo(int x, int y);
}

public class Bat extends Mammal implements Flyable {
    // integrate a Flyer into a Mammal
    Flyer        innerFlyer = new Flyer();

    public void flyTo(int x, int y){
        // wrapper for the original Flyer's method
        innerFlyer.flyTo(x,y);
    }
}
```

Discussion

- Multiple-inheritance is often considered harmful:
  - One can ended-up with ugly and heavy objects filled with craps from other classes;
  - Mixing classes without a bit of control can ended-up with inconsistency: a bat is not *mammal bird*.
- A good practice is to build small classes providing few methods (with a corresponding interface if needed.)
- Using aggregation (or better, *mixin* if available) is a rather good substitute for multiple-inheritance since it enforces the features approach: a bat is mammal with a flying capability and so a bat provide a `flyTo` methods.

Inheritance *v.s.* Object Composition

- This is a classic topic of Object Oriented Programming: should we prefer inheritance or object composition ? Here are some *pros and cons*:
- **Inheritance:**
  - Improve simple code reuse;
  - Explicitly binds classes with a hierarchical organization.
  - Ties relations between objects to a statically fixed model
  - Increase (dramatically) the number of classes ...
- **Object Composition:**
  - Offers more controls
  - Let objects dynamically decide how they're connected
  - Increase code size and errors potentiality
  - Increase overhead in methods' calls.

Observer Pattern

```
public interface Subject {
    public State getState();
    public void  registerObserver(Observer o);
}

public interface Observer {
    public void update();
}

public class State {
    private int           val;
    public int getVal() { return val; }
    public State(int v) { val = v; }
}
```

Observer Pattern

```
public class ConcreteSubject implements Subject {
    private int         val;
    private int         nbobs;
    private Observer[]  obs;
    public ConcreteSubject() {
        val = 1;
        nbobs = 0;
    }
    public State getState() {
        State           s = new State(val);
        return s;
    }
    public void registerObserver(Observer o) {
        obs[nbobs] = o;
        nbobs++;
    }
    private void update() {
        for (int i = 0; i<nbobs; ++i)
            obs[i].update();
    }
    public void fact(int n) {
        if (n<2) {
            this.update();
        }
        else {
            val *= n;
            this.update();
            this.fact(n-1);
        }
    }
}
```

Observer Pattern

```
public class ConcreteObserver implements Observer {
    private Subject      subject;
    public ConcreteObserver(Subject s) {
      subject = s;
      subject.registerObserver(this);
    }
    public void update() {
      System.out.println(subject.getState().getVal());
    }
}
```

Plan

Plan

Memory Management

- Memory is implicitly managed by the runtime.
- No explicit destruction is needed.
- Memory can only be allocated through the `new` keyword.
- The garbage collector tries to free objects with persistent root.
- Object is never implicitly copied (you can still use and redefine the `clone` method.)
- An object is **always** a pointer, thus you never passed an object to a method, you pass a pointer to it (often called reference, while this pointer can be `NULL`.)

Plan

Annotations

- Annotations are a kind of tags that can be prepended to any definitions.
- The usual purpose of annotations is to enhance documentation.
- Annotations can also be used to enforce development model.
- Annotations can be used in conjunction with introspection to retrieve methods without knowing the class definition.
- Annotations can be used to mark deprecated methods, overriding methods . . .
- Java comes with a predefined set of annotations that can be extended.

Annotations: Basic Example

```
class MyClass {
    protected int x;
    public MyClass(int _x) { x = _x ; }
    public int get() { return x; }
    public void set(int _x) { x = _x ; }
}

class MyClass2 extends MyClass {
    public MyClass(int _x) { x = 2*_x ; }
    @Override
    public int get() { return (x/2); }
    @Override
    public void set(int _x) { x = 2*_x ; }
}
```

Annotations: userdefined

```java
import java.lang.annotation.*;

@Documented
@interface AuthorsHeaders {
  String author();
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  String[] keywords();
}
```

Annotations: userdefined

```
@AuthorsHeaders(
  author = "Marwan Burelle",
  date = "2011/04/06",
  currentRevision = 2,
   keywords = {"Example", "Java", "Annotations"}
)
public class ExempleAnnotation02 {
  private int           r;
  private int           n;
  private boolean       done;
  public ExempleAnnotation02(int x){
    n = x; r = 1; done = false;
  }
  public void doIt() {
    for (int i=1; i<=n; ++i) r *= i;
    done = true;
  }
  public int get() {
    if (!done) this.doIt();
    return r;
  }
  public static void main(String [] args){
    ExempleAnnotation02 o = new ExempleAnnotation02(5);
    System.out.println("fact(5) = " + o.get());
  }
}
```

Plan

Exception Mechanism

- Java provides a *classical* exception mechanism with *throw and catch* operations.
- As usual, when throwing (*raising*) an exception it interrupts the current execution block and backtrack the call stack until it reaches an handler or the main method.
- *Uncatched* exceptions interrupt the execution with a specific error messages.
- You can *catch* an exception at any point in the call stack, offering various possibilities of error recovery or proper error messages and clean exit.
- Of course, you can define your own exceptions.

Checked or not Checked ?

- Java provides two kinds of exceptions: checked exceptions and *unchecked* exceptions.
- When a method *could* throws a checked exception, it has to inform others: *i.e.* the declaration of the method will be decorated by a `throws` clause.
- Unchecked exceptions may be throws without any warn to others.
- Java provides guidelines in order to define which exceptions should be checked. In the general case, an exception is **always checked**, only specific *runtime* exceptions (such as the `NullPointerException`) or errors are *unchecked*.
- Both checked or unchecked exceptions can be catched.

Try and catch ...

So, to handle an exception, you can *catch* it using a `try` block:

`try/catch` block

```
try {
  // code that can throw exceptions
} catch (ExceptionType name) {
  // handling code
} catch (ExceptionType name) {
  // handling code
}
// ...
```

`ExceptionType` is the name of a class extending `Throwable` and `name` can be used in the code of the handler to refer (and thus exploit) the exception. If the tried code throws one of the catch exceptions, the corresponding code will be triggered.

finally block

We often need to execute some common code to all `catch` blocks, the `finally` block is the provided solution: it will always be triggered even when the method exits normally. It can be used to clean-up things when dying using exceptions but also to enforce you always go through some code at the end of methods.

finally block

```
finally {
  // Final code.
}
```

Exceptions Example

```
public class ExClass {
    private class MyEx extends Throwable {
        private String  toto;
        public MyEx(String s){
            toto = new String(s);
        }
        public String getToto() { return toto; }
    }

    private class MyEx2 extends Throwable {
        private String  toto;
        public MyEx2(String s){
            toto = new String(s);
        }
        public String getToto() { return toto; }
    }

    public void doTheJob() throws MyEx
    {
        MyEx            e = new MyEx("First one...");
        System.out.println("Ok throwing MyEx");
        throw e;
    }
```

Exceptions Example

```
    public void doTheJob2() throws MyEx2
    {
        MyEx2           e2 = new MyEx2("Second one...");
        try {
            this.doTheJob();
        } catch (MyEx e) {
            System.out.println("Catched: " + e.getToto());
            throw e2;
        }
        finally {
            System.out.println("The finally block.");
        }
    }

    public static void main(String[] args) throws MyEx2
    {
        ExClass         o = new ExClass();
        o.doTheJob2();
    }
}
```

Exceptions Example

```
Ok throwing MyEx
Catched: First one...
The finally block.
Exception in thread "main" ExClass$MyEx2
        at ExClass.doTheJob2(ExClass.java:27)
        at ExClass.main(ExClass.java:42)
```

Plan

Plan

Compiling Java Code

- `javac` is the java compiler;
- The simplest way of compiling a file is:

  ```
  > javac HelloWorld.java
  ```

- This compilation produces a file `MyClass.class`;
- If `MyClass` contains a static `main` method, it can be passed to the JVM:

  ```
  > java HelloWorld
  Hello World!
  ```

- Each class should have has its own file (in order to be *public*) and the file should be named after the class name;

The JVM

- `java` command launches the JVM;
- It takes a class (bytecode file with extension `.class`) as input and execute the `main()` method of this class;
- Can take compressed classes (`.jar` files;)
- Classes are found in the class path (through `CLASSPATH` environment variable or options;)

CLASSPATH

- `java` (the JVM) look for the given class in the `CLASSPATH`;
- `CLASSPATH` is an environment variable (similar to `PATH`) defining path to class files (or jar file;)
- When loading a class, every classes needed must be accessible using the `CLASSPATH`;
- This path could also be specified using the `-cp` option of `java`;
- When compiling a class every classes needed should be available in the class path also;
- Fortunately the *dot* (current directory) is in the class path by default;

Plan

Packages

- Classes can be organized in packages;
- If your class belongs to a package, the file `must` begin with the statement:

```
package mypackage;
```

- The compiler and the JVM expect to find classes of a given package in a directory named after the package name (under the class path;)
- The package names often show some kind of hierarchy (*i.e.* `java.awt.color`):
  - It does not reflect a real inclusion hierarchy: the package `java.awt.color` is not included in `java.awt`;
  - Each name separated by a dot denotes a directory name (thus classes of the package `java.awt.color` should be find in the directory `java/awt/color/` under the class path;)

Using Packages

- A component of a package (namely a class) can be accessed in several ways:
  - Using the full name: `mypackage.MyClass;`
  - Importing the component at the beginning of the file: `import mypackage.MyClass;`
  - Importing the whole package at the beginning of the file: `import mypackage.*;`
- Static methods or fields can be *imported* in order to prevent full name usage: `myStaticMethod()` instead of `MyClass.myStaticMethod();`
- Static import is done using: `import static mypackage.MyClass.myStaticMethod;`
- Since dotted hierarchy of packages does not reflect an inclusion hierarchy, importing for exemple `java.awt.*` will not import `java.awt.color`

Plan

Using the Java Documentation

- Java API comes with a complete documentation built from the source.
- This documentation is organized by packages, classes, interfaces . . .
- Each class/interface is described by a short explanation of its purpose, its parent, its constructors, its properties and its methods (inherited or not.)
- Javadoc for Java 1.6 API is available at:

`http://download.oracle.com/javase/6/docs/api/`

- Modern IDEs (such as Eclipse or Netbeans) smartly integrate this documentation to provide contextual help and completion.

Producing Documentation

- You can produce your own documentation from sources using the `javadoc` command.
- Javadoc works almost as doxygen or any other literate programming tools:
  - In each definition (classes, interfaces, methods . . .) you can add documentation;
  - Documentation is extract from special comments;
  - The whole infrastructure of your documentation is built upon your code organization;
  - Only public *entities* (classes, methods, . . .) are *documented*.
- Javadoc can also use annotations.
- As usual, javadoc produces an API documentation, that is a documentation for developers.

An example

```java
/**
 * This is a kind of HelloWorld classes with some "extension"
 */
public class HelloWorld {

    private int                 x;
    private int                 y;

    /**
     * The class constructor.
     * @param _x initial value for property x
     * @param _y initial value for property y
     */
    public HelloWorld(int _x, int _y) {
        x = _x;
        y = _y;
    }

    /**
     * The usual toString method
     */
    public String toString() {
        return ("Hello, World ("+x+","+y+")");
    }

    public static void main(String[] args) {
        HelloWorld        o = new HelloWorld(0,0);
        System.out.println(o.toString());
    }
}
```

Codes/jdocex01/index.html

Plan

5 Issues
- Performance
- Safe or Not ?
- Java *v.s.* C++

Plan

Is Java Slow ?

- Bytecode programs are often considered slower than native binary programs;
- The previous assumption is partially true: part of the execution time is lost in the VM;
- But, real programs spend quiet some times in I/O or waiting for system calls to complete (which takes as much time in both bytecode and native binary;)
- Since Java is Fully Object Oriented (without troll inside), it sometimes loose time and space confronted to a more low level language;
- Some restrictions of the language (mainly to remain almost pure OO) introduce unwanted overhead;
- Java also offers some other compilation methods (Just In Time or Native Compilation) for specific cases;

Plan

Typing Issues

- While Java is considered strongly typed there are still some possibility of Run-Time errors due to type unsafe expressions accepted by the compiler;
- Java permits dynamic explicit unsafe cast;
- Explicit allocation and usage of pointer (as opposed to ML like references) can lead to NULL pointer dereferencement error;
- Some typing rules does not preserves type safety in order to satisfy design choices rather than subject reduction (result of an expression should accept the same type as the expression itself);
- Java types system does not distinguish inheritance and subtyping, which is prove to be unsafe;

Security Model for Mobile Code

- The JVM uses a very strong security policy for mobile code (applets or any other kinds of loaded code from outside.)
- Mobile code is *sandboxed* and has very limited resources access.
- Each loaded piece of bytecode is *bound-checked* on stack (no code can access outside its stack frame.)
- Java security model is *back-proven* with strong theoretical foundations (Boxed Ambient, type checking . . .)
- Code analysis, restricted capabilities and sandboxing are completed by classical cryptographic signatures technics.
- *Mobile Code is probably the safer part of Java.*

False Safety

- A common mistake is to consider that Java code is safe because of the restrictive language model and its security policy.
- High Level design enforces good safety and eliminates most of dangerous mistakes, but does not guarantee full safety.
- Usually, lots of people think that smaller code is safer and thus that since high level of abstraction induces smaller code, it also induces safer code:

It's not true, high level of abstraction induces *smarter* code, that induces *smarter bugs*, which can have worce consequences and prove to be harder to solve.

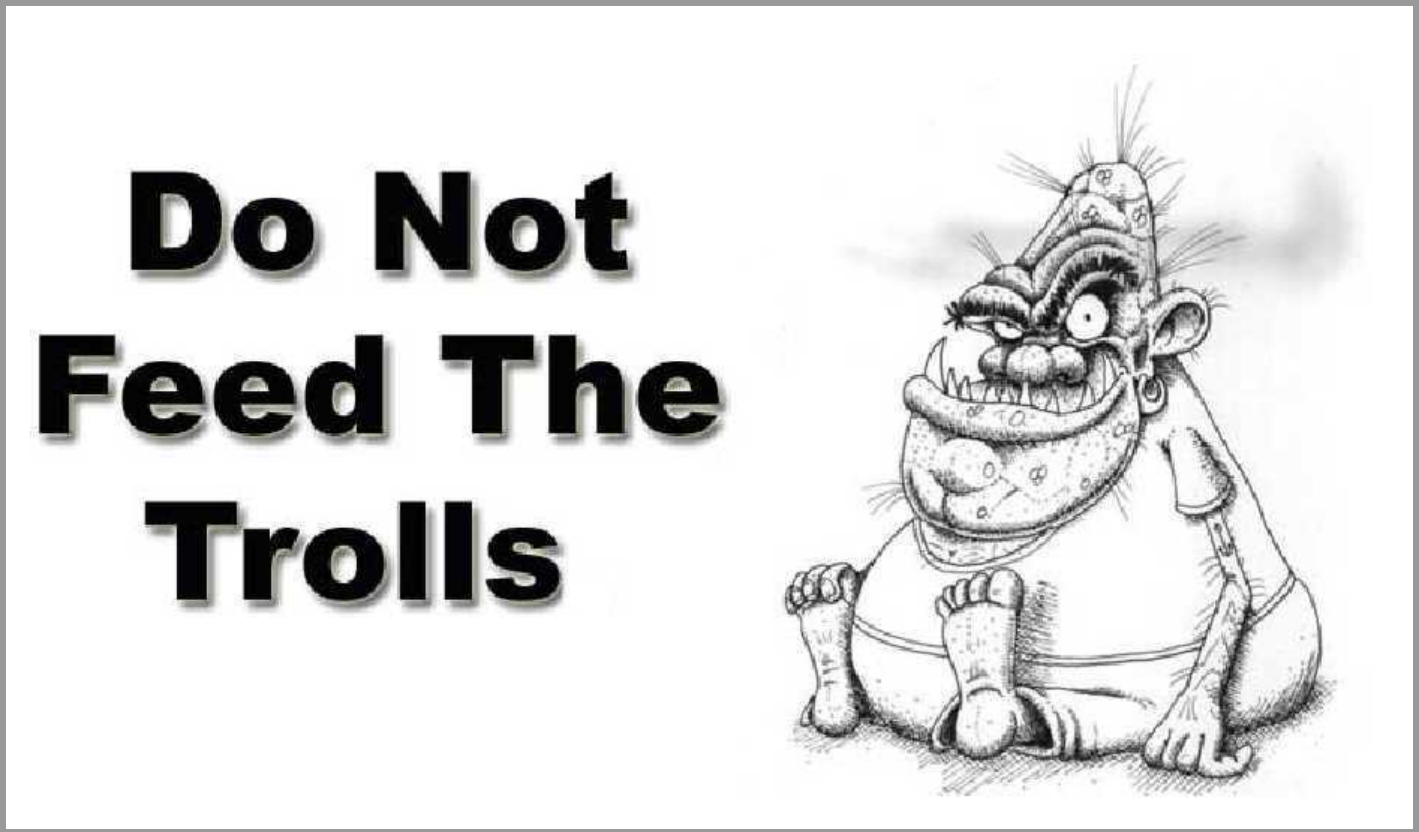
Don’t forget

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?”

Brian Kernighan, ”The Elements of Programming Style”, 2nd edition, chapter 2

Plan

Java *v.s.* C++



Java *v.s.* C++

- Simple inheritance
- No object as value (every objects are pointers)
- No static specialization
- No operators overloading (pure OOPL should not have operators anyway)
- Simpler inheritance model (no public nor private inheritance)

Plan

Biblio


Programming language popularity. http://www.langpop.com/.


Tiobe programming community index for april 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci, april 2012.