Lecture Notes for JAVA

Krmoll's
email: {andrey,molli}@loria.fr
web: http://www.loria.fr/~molli,andrey
ESIAL, Université de Nancy I,
IUT de Dijon, Université de Bourgogne

August 26, 1998

Contents

First steps in Java

Krmoll's
email: {andrey,molli}@loria.fr
web: http://www.loria.fr/~{molli,andrey}
August 26, 1998

Duke is saying ...

... Hello

Slide 1

Contents

- Introduction
- Language: classes, inheritance, polymorphism, Packages...
- Threads
- Advanced Window Toolkit (AWT)
- Applets
- Perspective-Conclusions

Slide 2

1 Introduction

Books (IMHO)

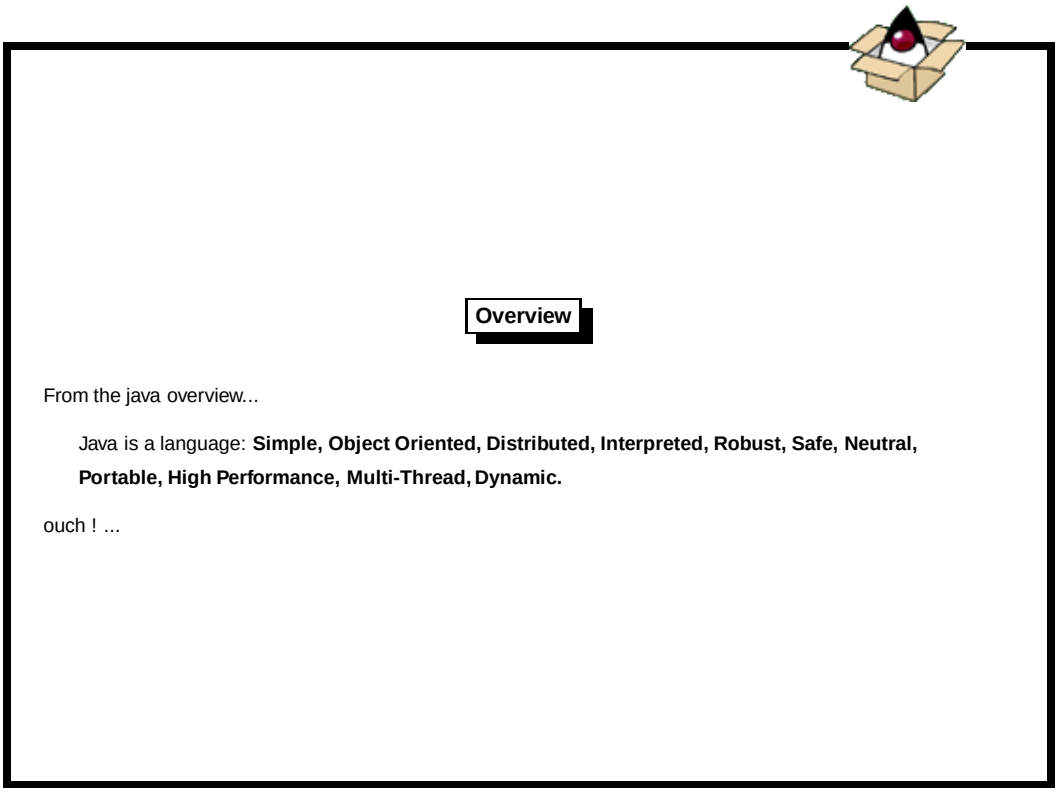
Good books :

- for programmers : “Java in Nutshell” [Fla96].
- for details about language → “the Java Language Specification” [JG96]
- the paper version (800 pages) of tutorial

Bad books :

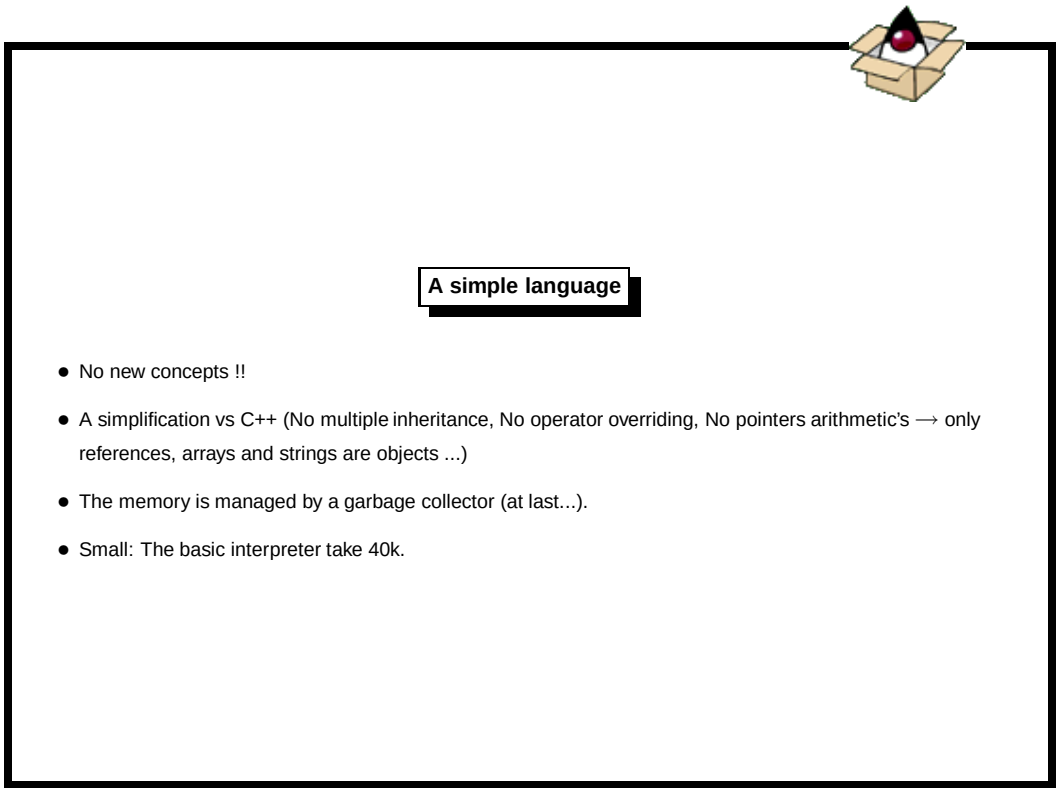
- “The Java Programming Language” [Arn96].
- “The Java Application Programming Interface” (Same publishers).

Slide 3



Slide 4

Ceci est repris de de [Mic95, Jam95]. Les différents points avancés par les auteurs sont discutables et vont être d'ailleurs discutés dans les transparents qui suivent.



Slide 5

En fait la jvm de base + code des bibliothèques de bases = 175 Ko (jdk 1.01).

An Object Oriented Language

Objects paradigms

- Inheritance (simple)
- Data Hiding (à la C++)
- Polymorphism (but “virtual” free ;-).
- No Genericity (using cast operator...)

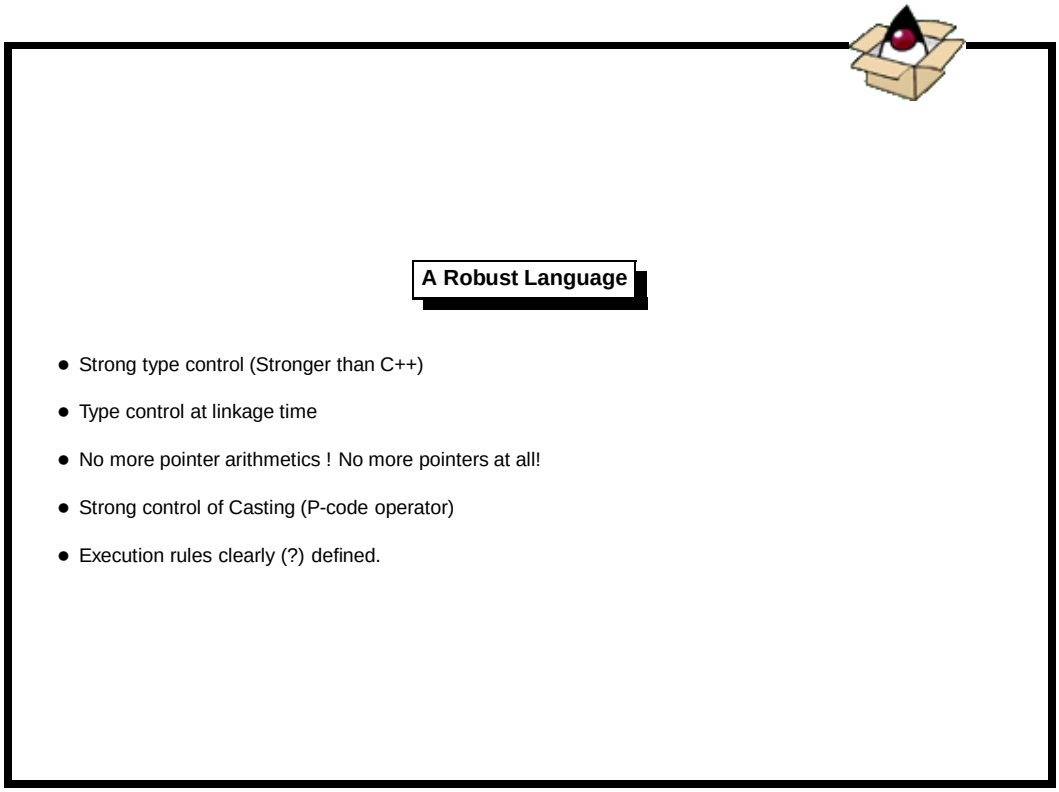
Slide 6

A Distributed Language

- A socket package
- Very simple access to HTTP and FTP protocols
- Opening an URL is nearly as difficult as opening a local file
- Really Distributed ?
- Remote Method Invocation (RMI) is allowed in JDK1.1.
- CORBA interface.

Slide 7

- On peut quand même attendre un peu plus d'un système distribué ...
- En fait on pourait aussi parler de "système distribuable" (effet du P-code, de la jvm). C'est vrai : le succès (?) de java grandemment lié à la facilité de :
  - distribuer le support : charger le jdk, portage jdk
  - distribuer des applications : chargements par réseau, pas d'installations, pas de problème d'architecture.
- En fait tout ceci n'est qu'une liste de propriétés. Il faudrait introduire une définition d'un système/langage distribué... Mais une telle discution est hors des objectifs de cette présentation.



Slide 8

- On dispose de règles seémantiques rendant plus strictes la génération d'applications (nombre important d'erreurs filtrées par le compilateur).
- La machine virtuelle garantit de propritétés à l'exécution (*run time checks*) (accès dans les tableaux, cohérence des modules,...)
- On évite ainsi de nombreuses situations pathologiques et dangereuses : initialisations, indexes de tableau, ...

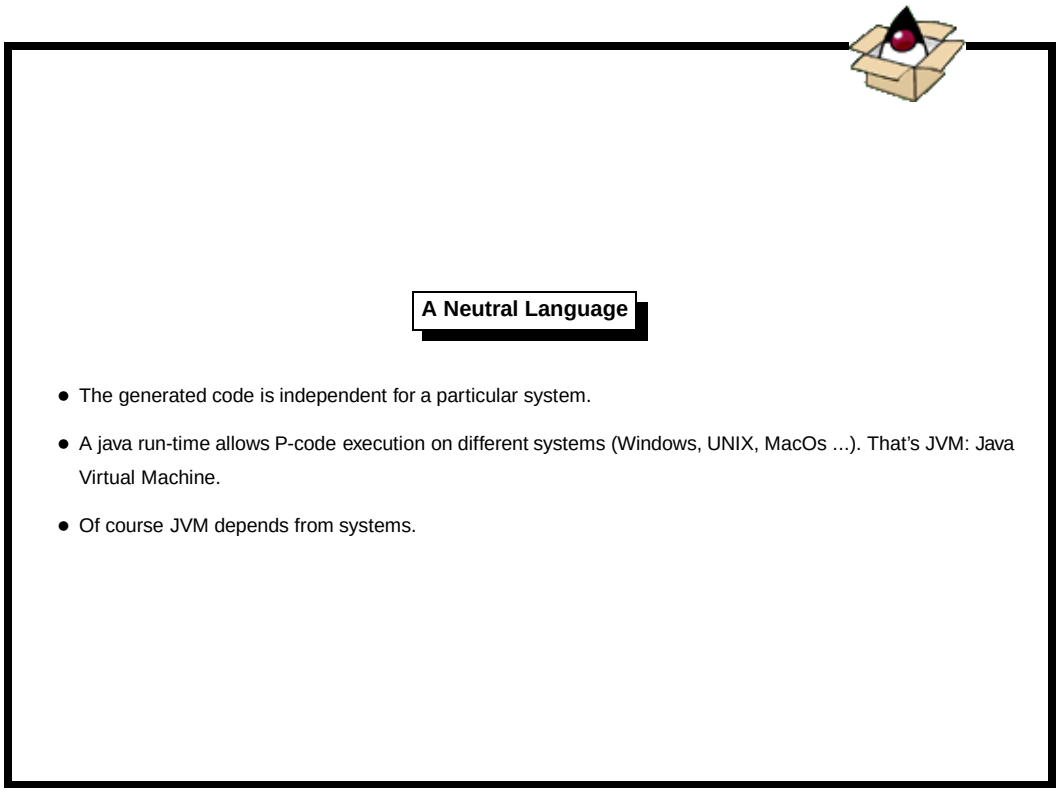
C'est ici une notion de "sureté de fonctionnement" (*safety*). Il faut se protéger contre les disfonctionnements système, logiciel, réseau...

A Safe Language

- Some code can be downloaded through the net, code and data can migrate from one JVM to another ...
- $\Rightarrow$ Problems of security :“virus”, “Trojan horse”.
- Authentication based on public/private key.
- The applet concept (applications are under control of a security manager that prevents almost all system calls)
- Pointer manipulation is impossible (Big hole for virus).
- Class loader and Security Manager...

Slide 9

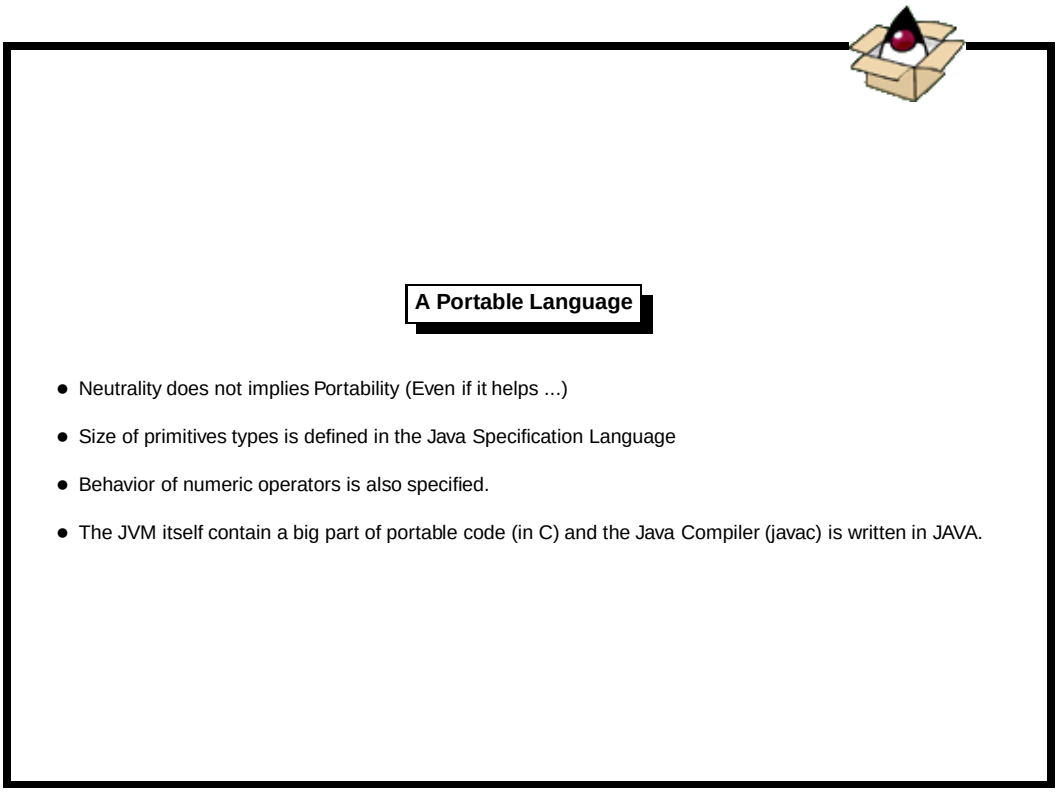
C’est ici une notion de sécurité contre les “malversations” (*security*).



Slide 10

On peut remarquer que ceci facilite la conception d'application distribuée.

On peut aussi noter que quelque part SUN a réussi (?) un petit exploit : il a proposé un langage et un système "neutre" qui devient petit à petit un "standard de fait".



Slide 11

La définition des types primitifs (entier, virgule flottante) est beaucoup plus précise qu'en C. Les différents réels (float, double) et leurs rapports avec les autres types (entiers) sont toujours définis dabs le cadre de la norme IEEE 754.

An Interpreted Language ?

- IMHO: From the point of view of the programmer, that's compiled (Byte code is generated, Type Control is checked...)
- But of course, the JVM interprets the Java Byte Code...
- Native java compiler exists !
- JVM can load at run-time byte-compiled class definition and continue execution...

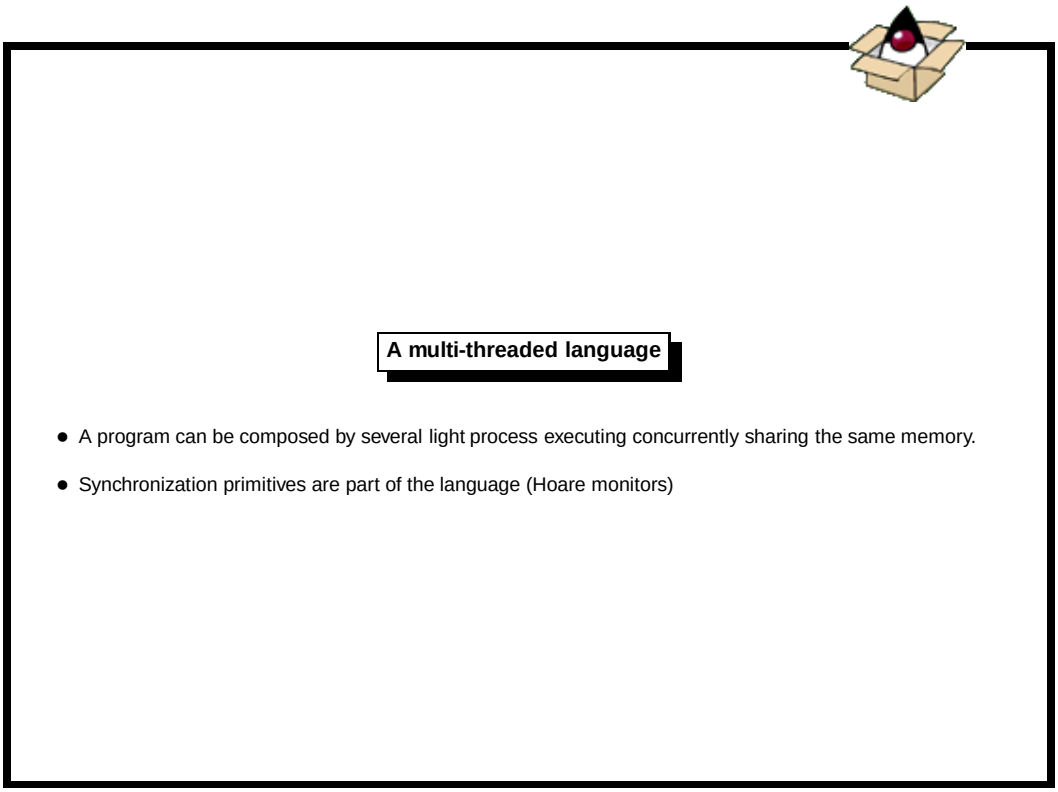
Slide 12

A High Performance language

- Parallelism with threads ...
- The P-Code can be translated in native code on the fly by the JVM itself...
- That's JIT : Just in Time Compiler
- However, even with JIT extension, the JVM is slower than native application, waiting for the sun hotspot technology..., native compiler ...

Slide 13

Attention : le terme “langage haute performance” a une connotation “calcul haute performance” où fleurissent des Cray, des sgi, des machines multiprocesseurs flirtant avec le terraflops. Or manisfestement java ne se positionne pas dans ce domaine, le terme “haute performance” est donc très discutable. Cette discution est hors du sujet de cet exposé.



Slide 14

- Sur ce point java a entre autre comme précurseur ADA qui intègre dans le langage (syntaxe et sémantique) les concepts de tâches et de rendez-vous.
- Smalltalk gère aussi la concurrence ...

A dynamic language

- Classes exists as objects at run-time (unlike with c++).
- There are no meta-classes like in smalltalk, you cannot create new classes at run-time...
- But you can ask to your class object what fields and methods are defined and compose dynamically a message call.
- that's a dynamic language, not reflexive.
- But you can generate code at run-time, call the compiler and load the resulting byte-code in JVM (as in C/C++)...

Slide 15

Conclusion

Java is a language:

- **Simple, Object Oriented, Distributed, Interpreted, Robust, Safe, Neutral, Portable, High Performance, Multi-Thread, Dynamic.**
- IMHO: Allows to develop more easily modern applications integrating:
  - Graphical Interface
  - Database Access
  - Network Access
  - Distributed application
- And it's portable !! Famous "Write once, run everywhere !"

Slide 16

Personnellement, je ne connais aucune toolkit permettant à elle seule de développer une telle panoplie d'application, et en plus c'est portable, quelque part cela force mon admiration ....

Contents

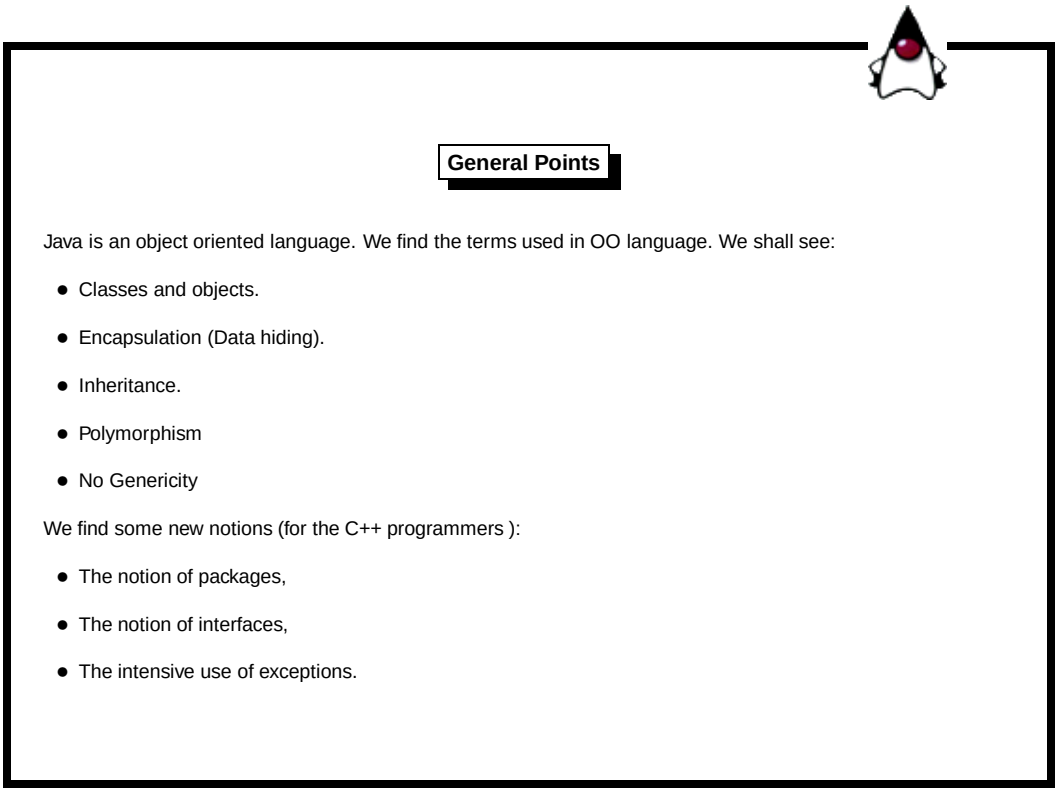
- Introduction
- **Language: classes, inheritance, polymorphism, Packages...**
- Threads
- Advanced Window Toolkit (AWT)
- Applets
- Perspective-Conclusions

Slide 17

2 The Java Programming Language

2.1 Overview

Bon, nous n'avons pas la prétention d'expliquer ce qu'est une classe ou un langage, de classes, à objet... Voir : [**?**]. Mais, on en parlera un peu quand même pour que la présentation soit autre chose qu'une litanie de mots clefs et de définitions opérationelles.



Slide 18

General Points

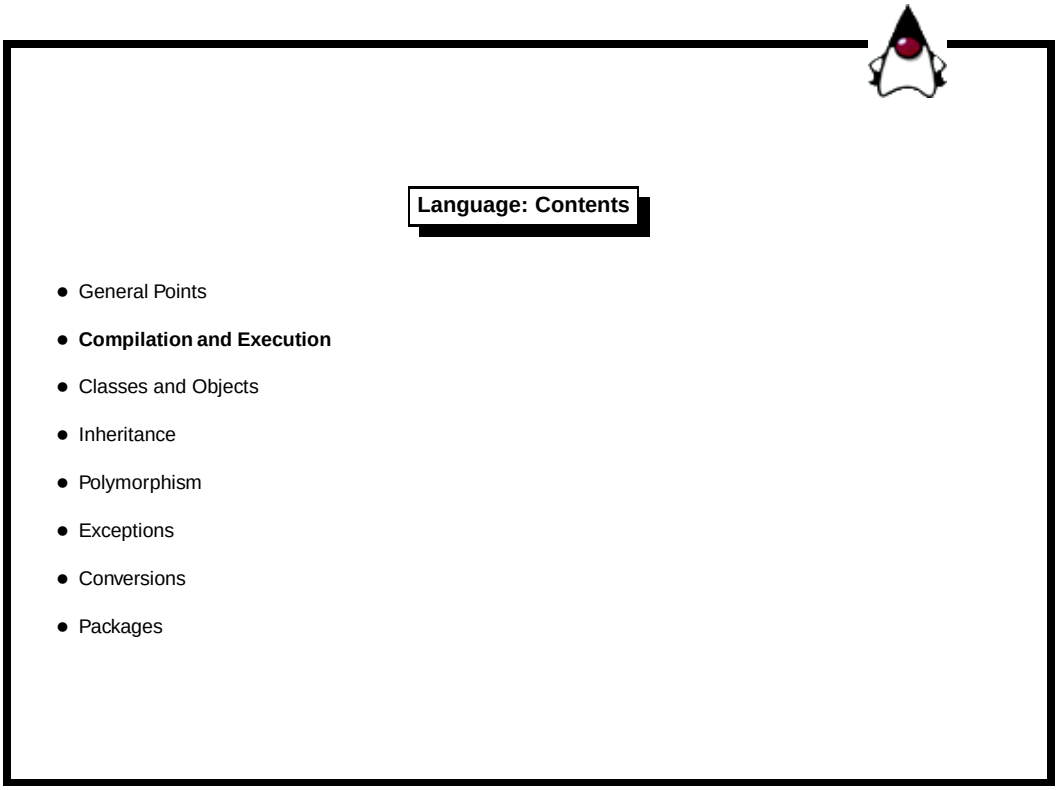
- No more pointers, just references
- No more destructors (like C++), a garbage collector
- Primitives types are still not objects (like C++, unlike Smalltalk).
- Arrays are objects with index bound checking at run-time
- String are objects (not an array of char) !
- The beginning of reflexivity (`class Class`): a dynamic language..
- Syntax like C/C++.

Slide 19

General Points

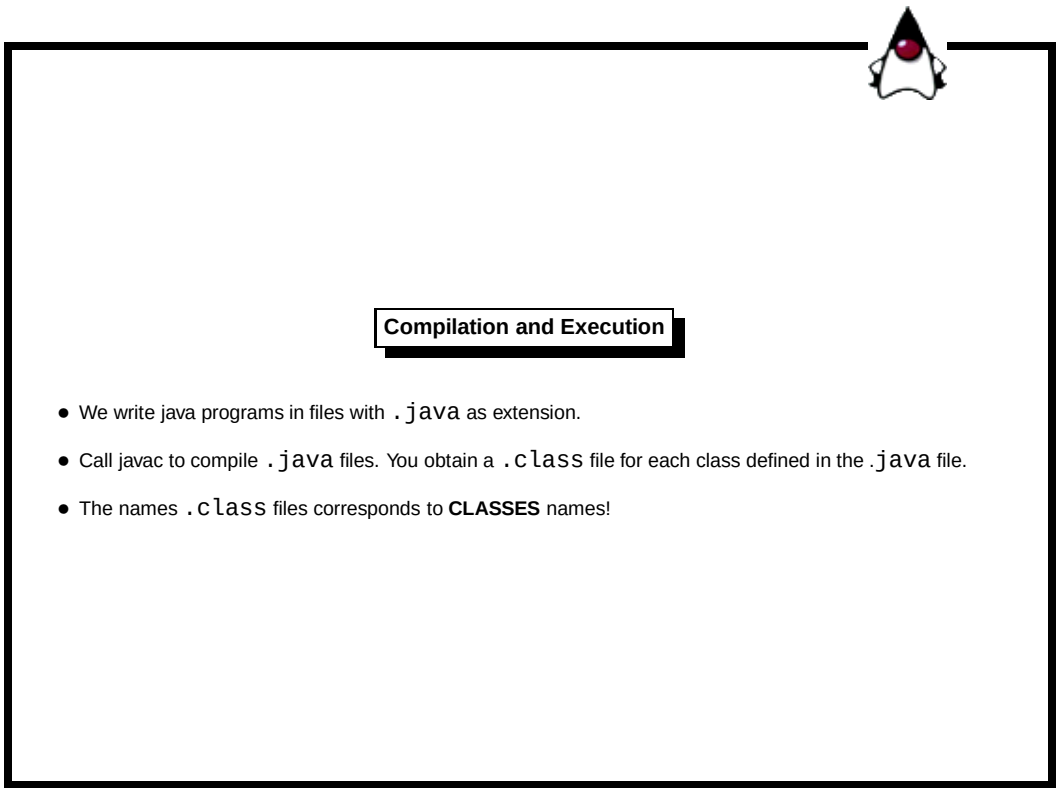
- Threads : support for concurrent programming
- So many “basic” libraries or extension :
  - graphical interfaces : AWT, Swing
  - Data Bases : JDBC
  - Distributed Object : RMI, CORBA
  - Support for Native Implementation : Natives
  - 2D/3D, Multimedia, Embedded, Personal...

Slide 20



Slide 21

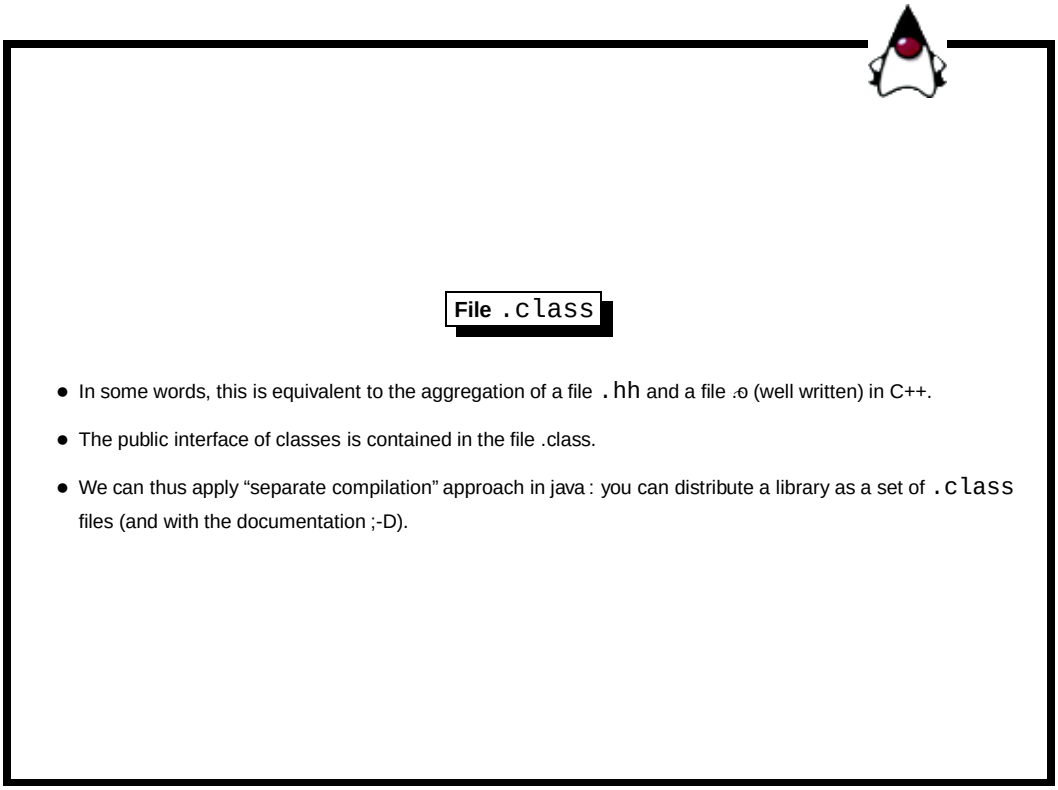
2.2 Compilation and Execution



Slide 22

- Donc il n'y a plus de pré-processeur. Il n'a plus de .hh et de .cc. Tout (interface implémentation) est conditionné dans un seul fichier .java. Ceci est dû à la nature des fichiers `.class` générés lors de la compilation (voir plus loin).
- Une règle brutale de programmation java (avec le jdk) peut être : "une classe ou interface par fichier .java". Ceci peut être utile entre autre pour les classes d'exception (le compilateur `javac` du `jdk` génére parfois des erreurs un peu obscur[1]
- Nous ne sommes pas concerné par la longueur du suffixe `.java` puisque le jdk n'est pas supporté pour MSDOS. Il est d'ailleurs ironique de voir, malgré les beaux discours de présentation, comment les problèmes de portage apparaissent vite dès que l'on passe à l'implantation du support.

[1]Tout comme la *force* java a un coté obscur.



Slide 23

Bon, ça a l'air de rien, mais c'est important cette histoire de compilation séparée. Typiquement, un vendeur de librairie peut difficilement diffuser une librairie si il doit en donner les sources. En java, bon nombre de librairies sont diffusées comme un packet de fichier .class.

Alors où est le problème ? eh bien, il existe d'autres approches de compilation comme en SmallEiffel, où le compilateur suit le flot de controle pour compiler. Il est alors capable de générer un code trés efficace et compact (le code généré ne contient que le code vivant). Le revers de la médaille est qu'il est obligatoire d'avoir les sources... ce qui n'est pas toujours compatible avec des contraintes industrielles.

Virtual Machine

- When you call the Java Virtual Machine (JVM), You have to give as a parameter, the **N**AME OF A CLASS (and not the name of a file).
- The corresponding **.**class must be found in the list of directories contained in the CLASSPATH environment variable.
- When this .class is found, it's loaded in the JVM and the execution starts be calling the following method:

  public static void main(String[] args)

Slide 24

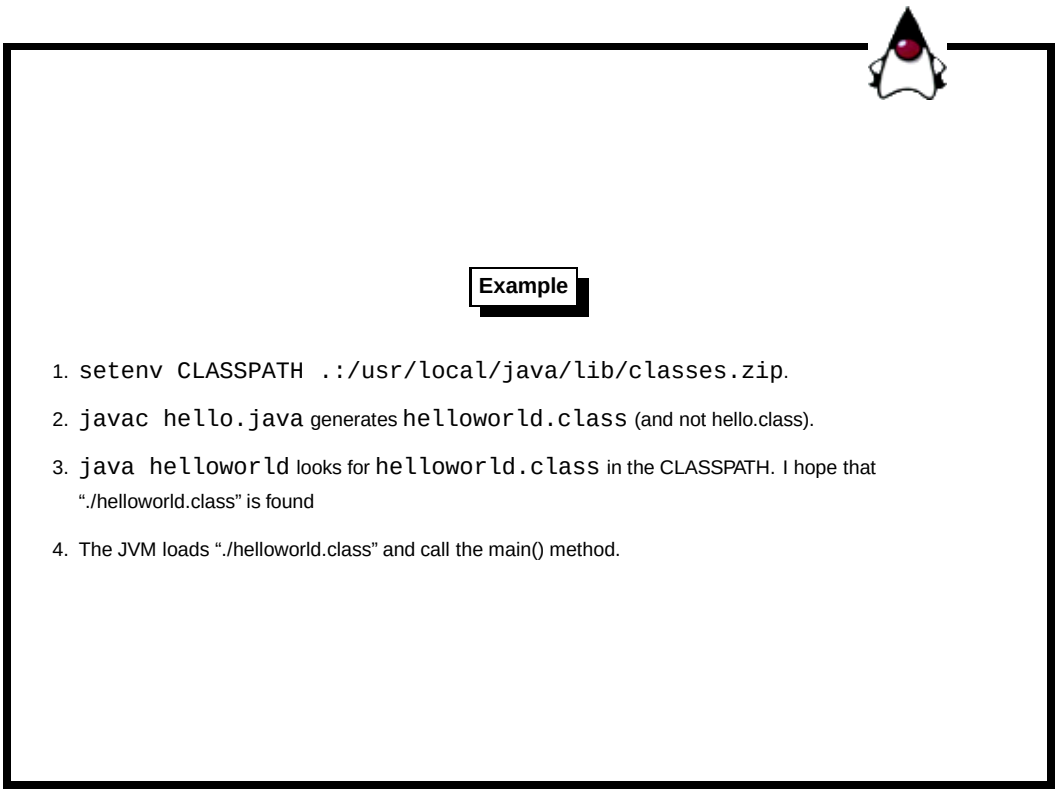
Example

examples/hello.java

```
class helloworld {
    public static void main(String[] args) {
      System.out.println("hello world");
   }
}
```

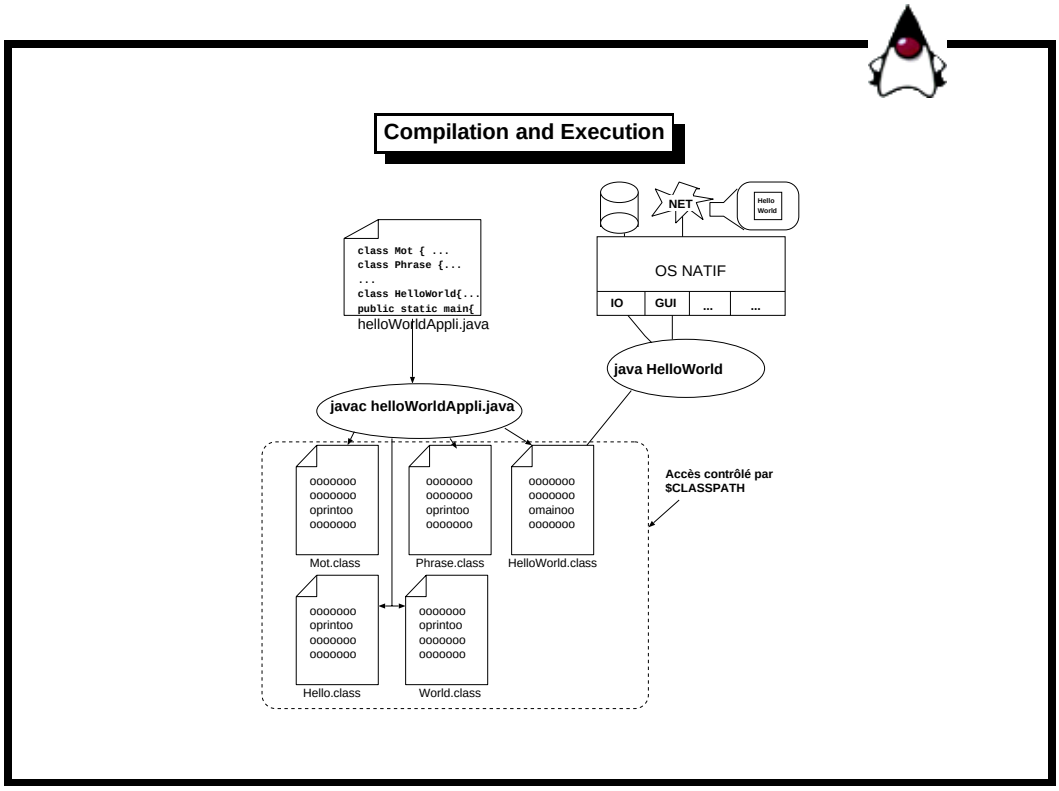
Slide 25

On sauve ça dans hello.java



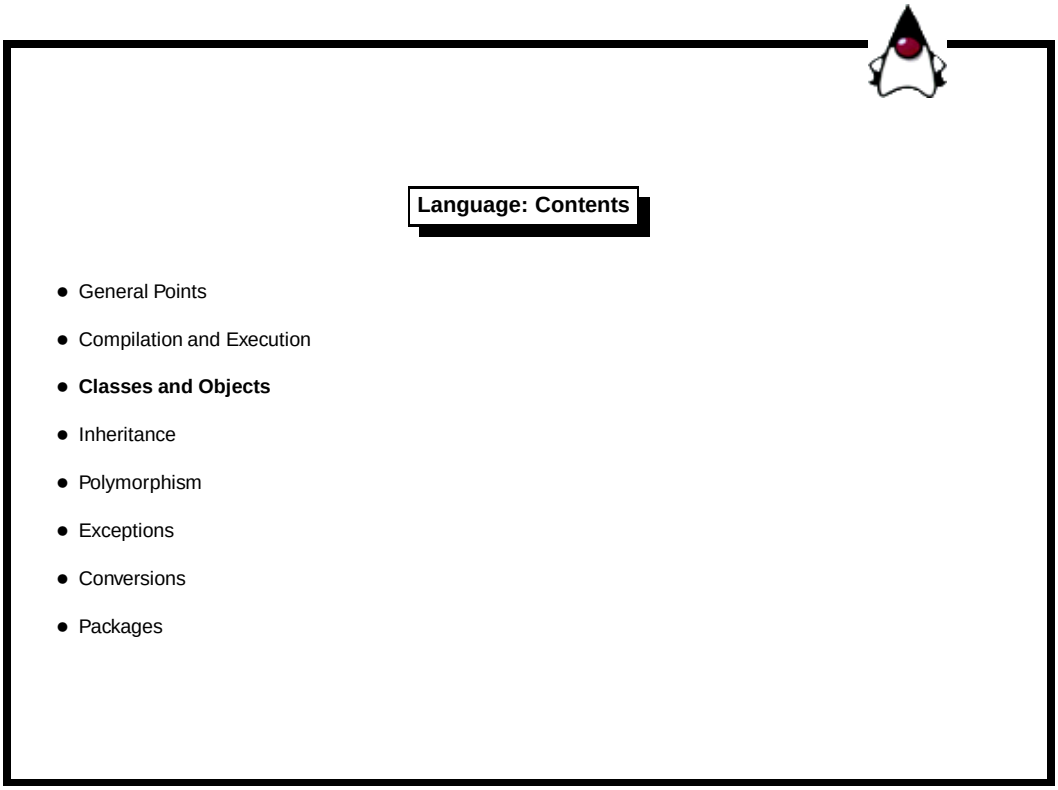
Slide 26

La variable d'environnement (OS support de la jvm) `$CLASSPATH` est utilisé par le chargeur de classe de la machine virtuelle java pour retrouver les `.class` à charger. On peut déjà s'avancer et noter que le "class loader" est aussi une classe qui peut être spécilisée par sous classement. Quand la machine virtuelle est intégrée dans un browser, la recherche des `.class` se fait à travers le réseau. De manière générale, l'édition de lien se fait qclasse par classe (chargement de la classe) et de manière **tardive**. On est à l'opposé de la génération d'exécutable qui contient **tout** le code éventuellement nécessaire, où **tous** les appels sont liés à du code (compilation statique).



Slide 27

Attention, un fichier .java peut contenir plusieurs définition de classes. Pour un .java on peut donc avoir plusieurs .class générés (pas de paniques) !



Slide 28

2.3 Classes and Objects

2.3.1 Classes and Objects

Classes&Objects

Everything is not object:

- The primitives types: byte, short, int, long, float, double, char, boolean are not objects.
- Thus, the operations +,-,*,/ ... are just language terminals.
- It is not possible to override these operations as in C++.

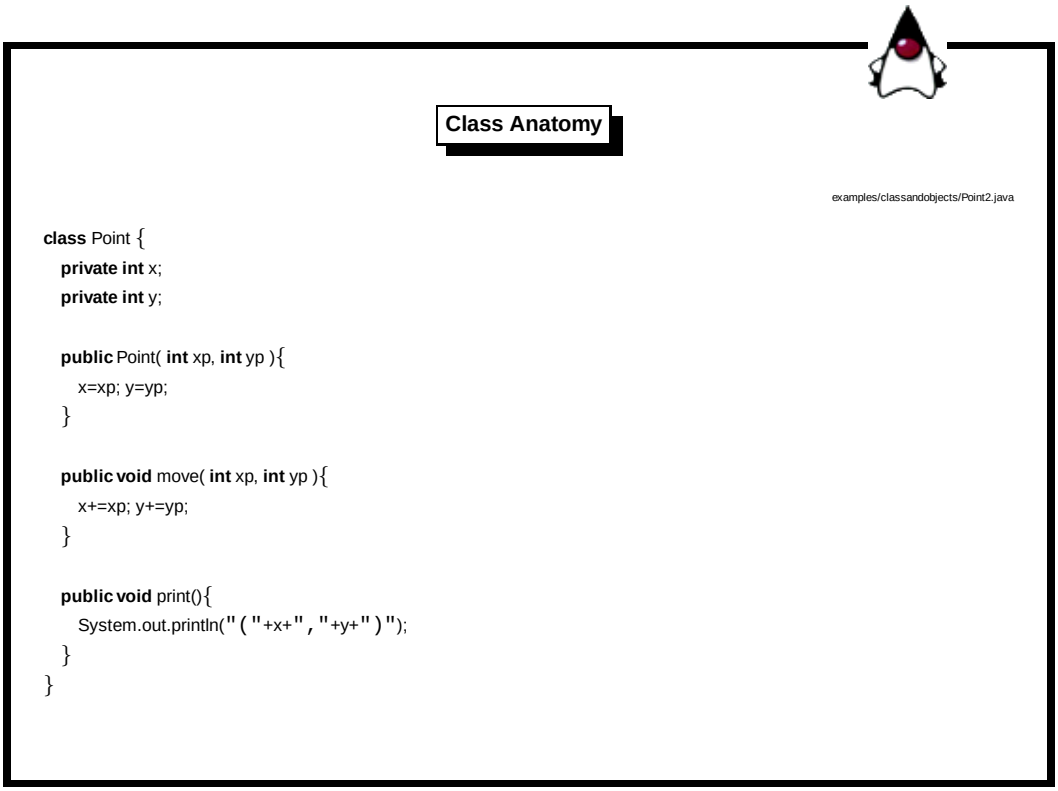
Slide 29

Classes&Objects

- The classes exist as objects at run-time. Calling the method getClass() on any instance return an object of class `Class`
- More precisely, the JVM creates an object of the class Class for each loaded class.
- By this way, the description of the classes is available at run-time, and it's possible to generates method invocation.
- But it's not possible to create new classes (as in smalltalk language) : Java is dynamic but not reflexive.
- Applications : debuggers, interpretor, inspectors, classes browser

Slide 30

A priori, les applications peuvent découvrir les classes qui existent dans la JVM à l'exécution. Par exemple la méthode de classee (statique) `Class.forName("A")` va renvoyer l'instance de la classe `Class` qui représente la classe `A` à l'exécution, si elle existe.



Class Anatomy

examples/classandobjects/Point2.java

```
class Point {
  private int x;
  private int y;

  public Point( int xp, int yp ){
    x=xp; y=yp;
  }

  public void move( int xp, int yp ){
    x+=xp; y+=yp;
  }

  public void print(){
    System.out.println("("+x+","+y+")");
  }
}
```

Slide 31

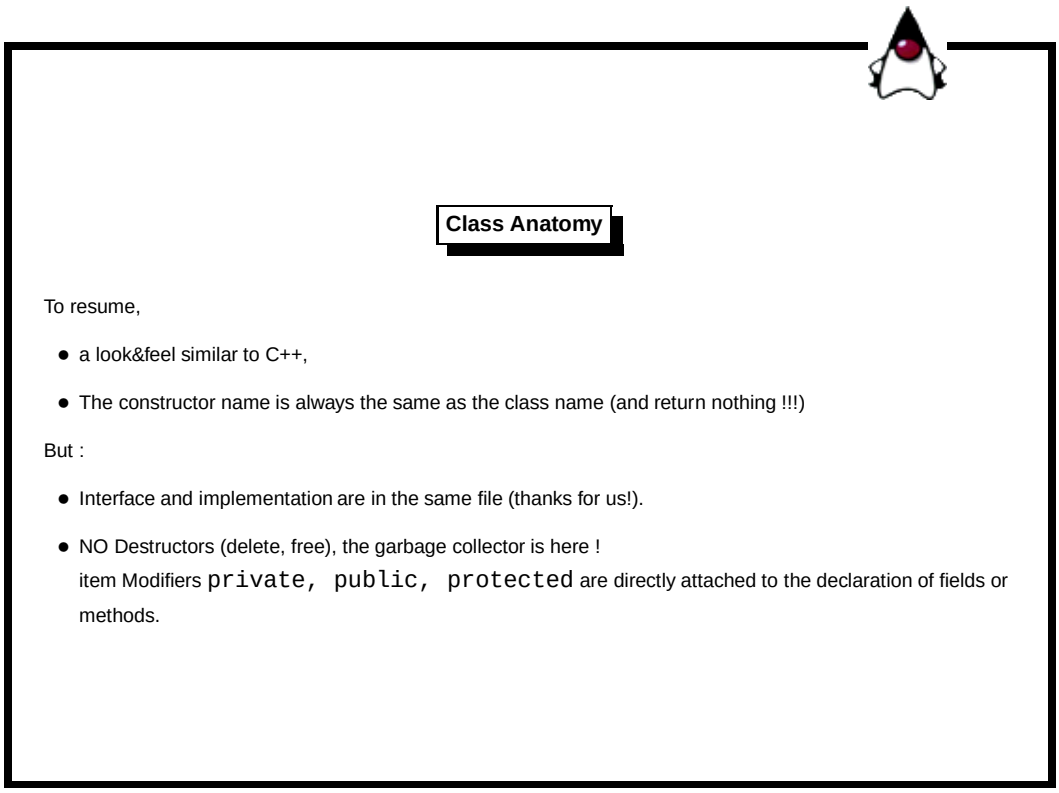
De loin tout ceci ressemble à du C++.
Il y a tout de même pas mal de différences :

1. Contrairement à C++ il n'y a pas possibilité de séparer l'interface d'une classe de son implémentation. (e.g : `.hh` et `.C`). Ceci n'est pas réellement nécessaire puisque c'est la version "compilée" d'une classe (`.class`) qui fournit les informations d'interface nécessaire à l'utilisation d'une classe lors de la compilation. (en C++ : il faut les modules objets compilés, `. o` ET les interfaces `.hh`. En Eiffel : on ne peut pas. En Ada : module objet compilé ET package avec uniquement l'interface).

2. Les constructeurs ont la même tête. On verra que la construction d'une instance elle est un peu différente. Ils portent donc le même nom que la classe. Java définit dans tous les cas un constructeur par défaut.

3. PAS de destructeur : usage d'un garbage collector et uniquement de variables "références à une instance" les rendent inutiles. Seule une la méthode `finalize` subsiste. Elle ne doit pas être confondue avec un destructeur. C'est une méthode optionelle qui peut être déclenchée par le GC quand il se débarrasse pour de bon d'une instance.

4. ;-¿ Il n'y a pas besoin de mettre un ";" après l'accolade finale de la définition d'une classe. C'est quand même permis pour supporter les programmeurs C++ indécrottables.



Slide 32

Le fait d'avoir une ramasse-miette a un impact énorme sur la qualité des programmes produits (même si c'est difficile à mesure). Avec cette manière de gérer la mémoire, il est impossible d'avoir des corruptions de mémoire ou des fuites de mémoire ! Cela coupe court à tous nombres de problèmes qui pourrissent la vie des programmeurs C++.

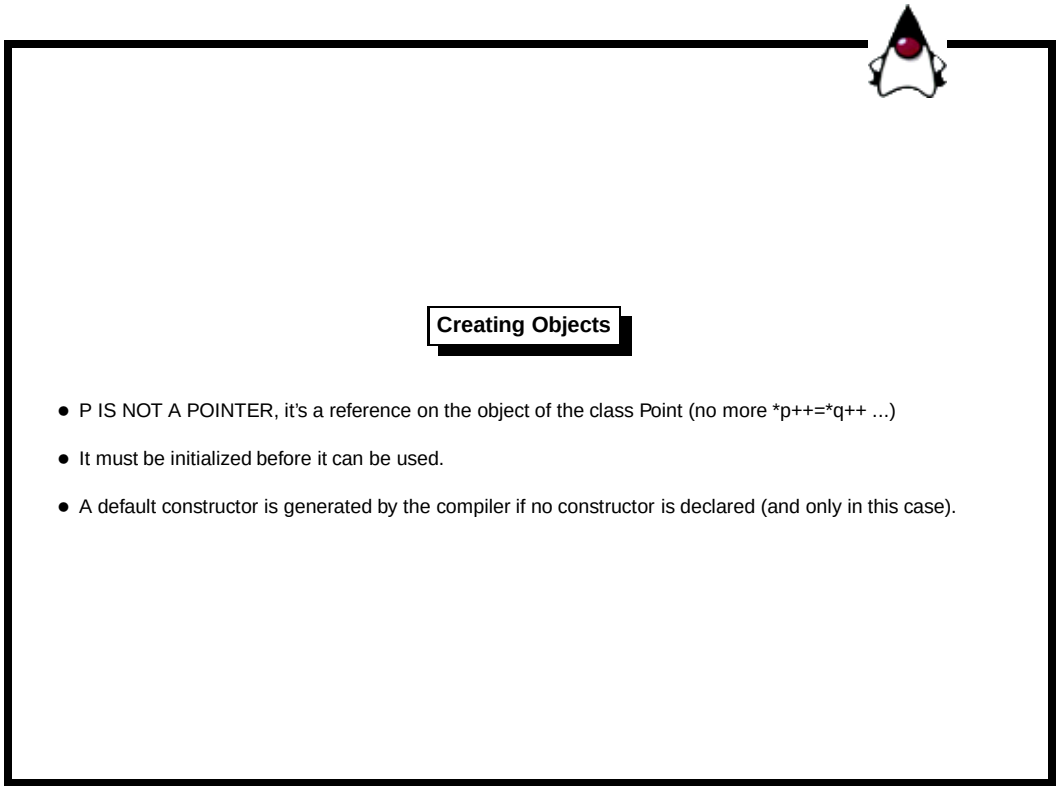
Instanciation

examples/SimplePointDemo.java

```
class SimplePoint {
   private int x,int y;
   public SimplePoint( int pX, int pY ){x=pX;y=pY;}
   public void move( int pDx, int pDy ){x+=pDx; y+=pDy;}
   public int getX(){return x;}
   public int getY(){return y;}
   public void print(){System.out.println("("+getX()+","+getY()+")");}
}
public class SimplePointDemo {
   public static void main(String[] args) {
      SimplePoint p = new SimplePoint(2,3);
      p.print();
      SimplePoint q;
      q.print();                                                    // error
      q=new SimplePoint();                                          // error
      q = new SimplePoint(3,4);
      q.print();
   }
}
```

Slide 33

First error: Not initialized Second error: Constructor does not exist.



Slide 34

Un pointeur c'est un entier représentant une addresse en mémoire. On donc faire des opérations arithmétiques dessus (*i++ ...). Une référence c'est un lien sur UN OBJET TYPÉ. Ce lien n'est pas altérable. Une référence n'a que deux état : non initialisée ou initialisée sur un objet typé. La différence a l'air subtile mais elle est énorme dans les faits !

Le constructeur par défaut, sans paramètre est une simple facilité offerte au programmeur. Il n'est pas impératif pour Java de disposer d'un constructeur sans paramètre. Il est nécessaire (entre autre) en C++ pour initialiser les tableaux d'objets. En java il n'y que des tableaux de références (voir §2.3.4), où le programmeur initialise explicitement ces références (new, autres références).

2.3.2 Instantiation and Constructors

Instantiation and Constructors

- A class may have several constructors
- Using of overloading ...
- A method is identified by its name, the number and the type of its parameters (the return type makes no differences)
- If no constructors are defined, the compiler generates a default constructors (as in C++).
- Otherwise, if a constructor is defined, the default constructor is not generated (unlike with C++).

Slide 35

Instanciation and Constructors

examples/classandobjects/constructors.java

```
class Point {
   private int x;
   private int y;
   public Point( int xp, int yp ){
      x=xp; y=yp;
   }
   public Point(Point p) {
      x=p.x;y=p.y;
   }
   // ...
}
class Main {
   public static void main(String arg[]) {
      Point p=new Point(5,6);
      Point q=new Point(q);
      Point r=new Point();                                    // error
   }
}
```

Slide 36

Current Reference

- The `this` reference allows an instance method to reference the instance itself.

examples/classandobjects/this.java

```
class Point {
  private int x;
  private int y;
  // main constructor
  public Point( int x, int y ){
    this.x=x; this.y=y;
  }
  // other constructors redirecting construction to main constructor
  // using "this" syntax
  public Point() {this(0,0);}
  public Point(Point p) { this(p.x(),p.y());}

  // this is unusefull, but syntax is right ...
  public int x(){ return this.x; }
  public int y(){ return this.y; }
  public void print(){ System.out.println("("+this.x()+","+this.y()+")"); }
}
```

Slide 37

C'est plus important que ca en a l'air. Par exemple, bien noter comment Point() appelle Point(int xp, int yp). Ça devient encore plus important lorsque l'héritage et la liaison dynamique sont de la partie. On y reviendra.

2.3.3 Classes and Encapsulation

Classes and Encapsulation

- In Java, encapsulation is based on classes (like c++) and not on objects (like Smalltalk or eiffel)
- Two objects of the same class can access their private fields ...
- A private field or method of an object is visible by all others objects of the same class
- A public field or method of an object is visible by all others objects.
- *a* priori: The state of an object should be defined as private and only method composing the public interface should be declared as public.

Slide 38

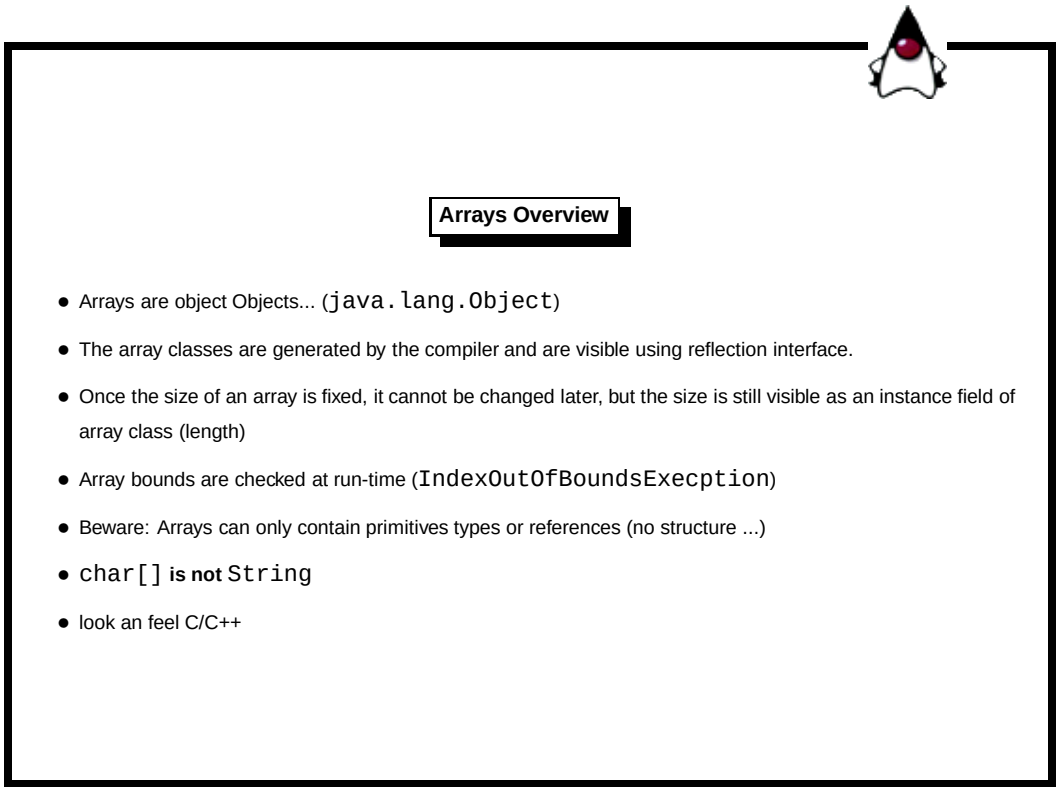
Classes and encapsulation

examples/classandobjects/encap.java

```
class Point {
  private int x;
  private int y;

  public Point( int xp, int yp ){
    x=xp; y=yp;
  }
  public int x() { return x;}
  public int y() { return y;}
  public Point(Point p) {
    x=p.x;                                  // ok but not
    y=p.y;                                    // very nice

    x=p.x();                                  // better ...
    y=p.y();                                          //
  }
  // ...
}
```

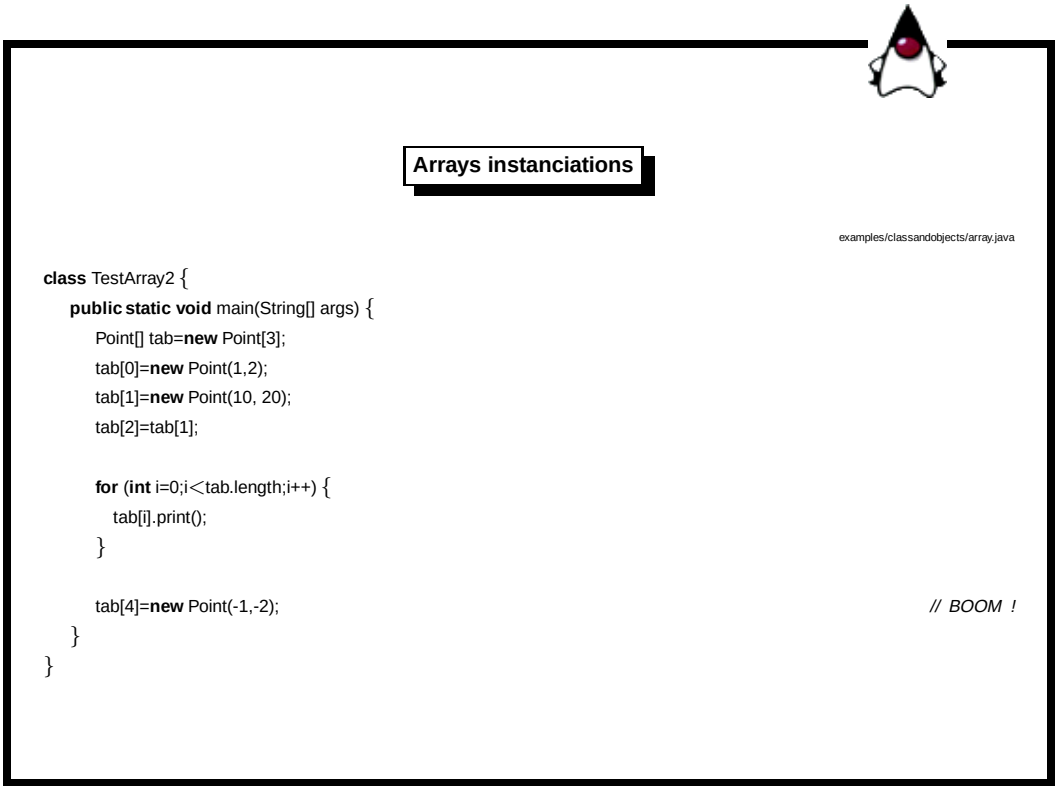
Slide 39

2.3.4 Arrays



Slide 40

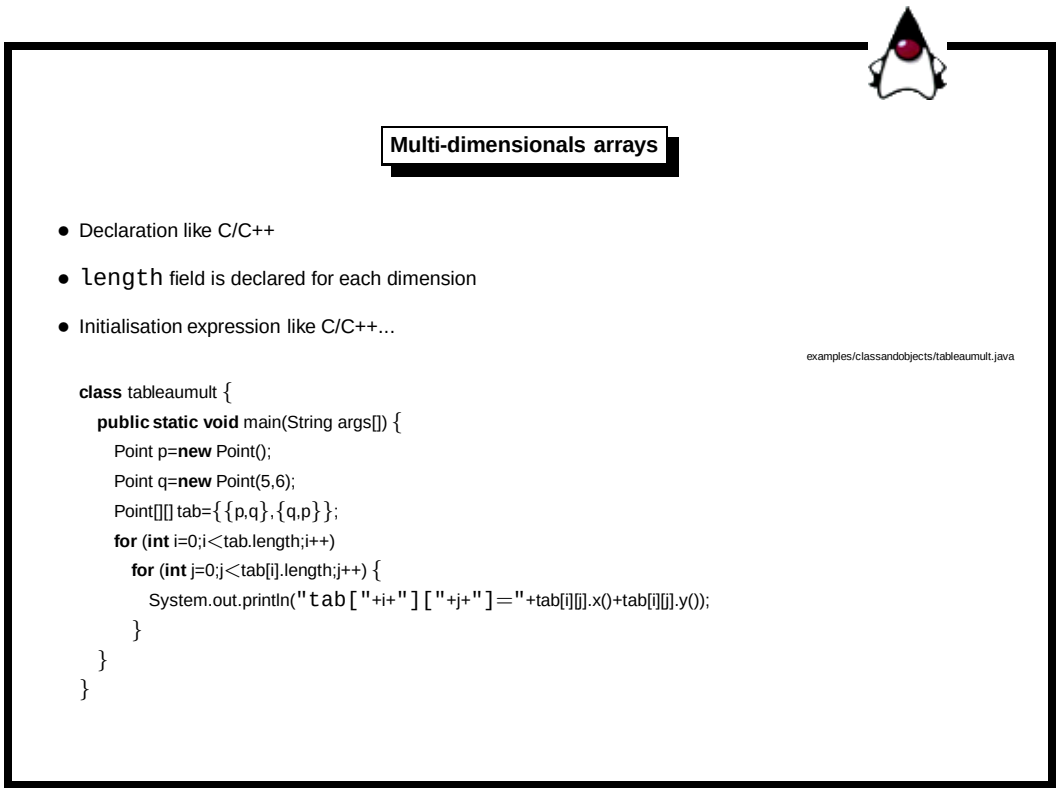
Dans ce cas une classe tableau [Point est générée par le compilateur (et est sous-classe de object)



Slide 41

Dans l'ordre :

1. On déclare une référence (non initialisée) sur un objet de type tableau de points,
2. On initialise unTab : on crée trois références sur des objets de type points. Ces références sont non-initialisées.
3. On initialise les références ...



Slide 42

- La remarque à propos des tableaux de caractères et des chaînes ne sera pas répétée ! Et puis il n'y a plus d'histoire de "null terminated" ou tout autre infection du genre.
- Il faut préciser que des problèmes de conversions subtils peuvent se produire à cause des tableaux. La classe effectivement associée à un objet tableau est celle donnée au `new` du tableau. Une fois créé un tableau peut contenir au mieux des instances de la classe effective ou de sous classe de celle-ci. Par contre il est légal affecter une variable tableau (une référence d'objet...) par une autre du moment où le deuxième utilise une sous classe du premier. Ceci peut engendrer des problèmes non vérifiables par le compilateur (voir slides **??**, **??**) Ces problèmes sont traités à l'**exécution** comme l'indique [JG96][200] :

> l'affectation d'un élément d'un tableau dont le type est : `A[]`, où `A` est une référence de type est vérifiée à l'exécution pour s'assurer que la valeur affectée peut bien l'être vers le type réel d'un élément du tableau.

2.3.5 Garbage Collectors vs Destructors

Garbage Collector Vs Destructors

- No destructors in JAVA.
- A garbage collector takes care to find allocated objects which are no longer referred and deallocates the memory.
- The garbage collector takes care also of memory fragmentation problems.

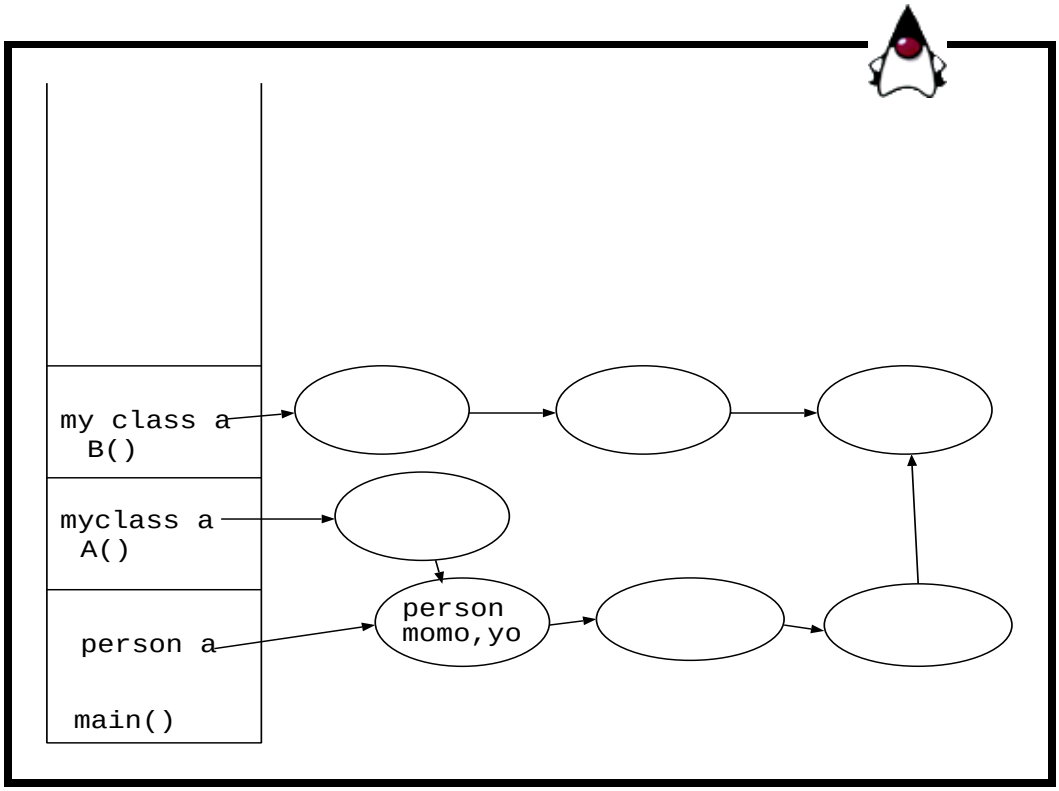
Slide 43

GC Mark&Sweep: Go through the references

- Start with the root and mark all the reachables objects.
- When all the graphs are visited, unmarked objects are destroyed.

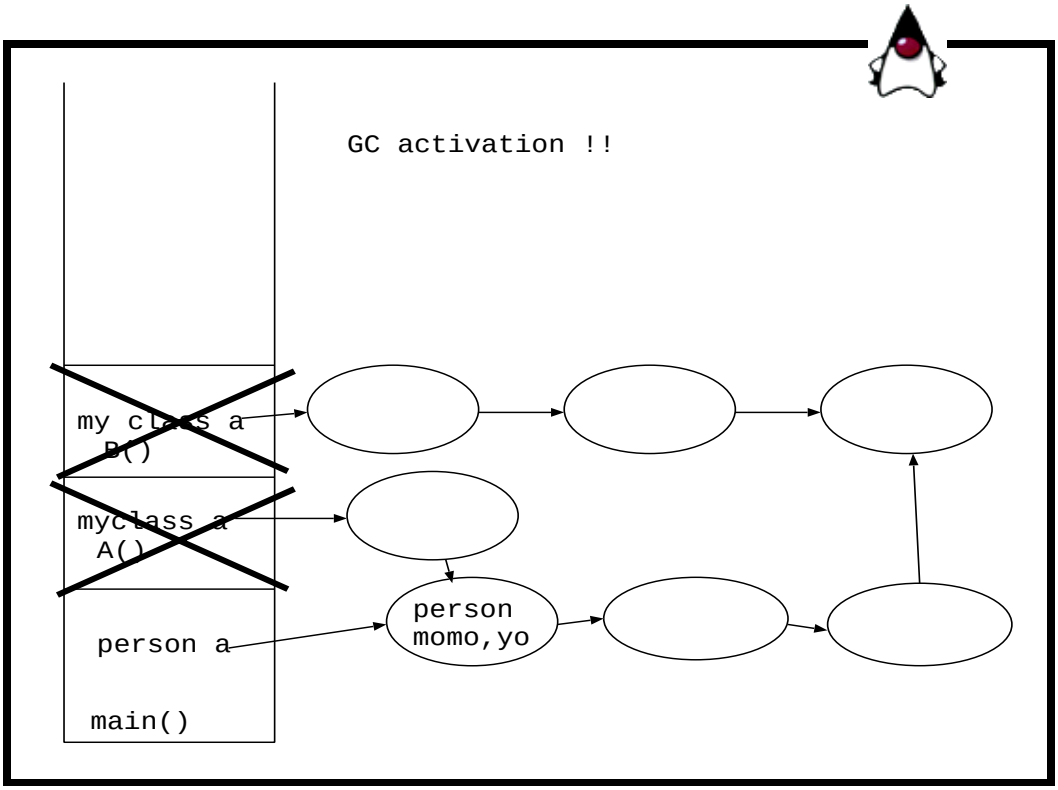
Slide 44

Les racines se trouvent en fait dans la pile d'appel des méthodes !



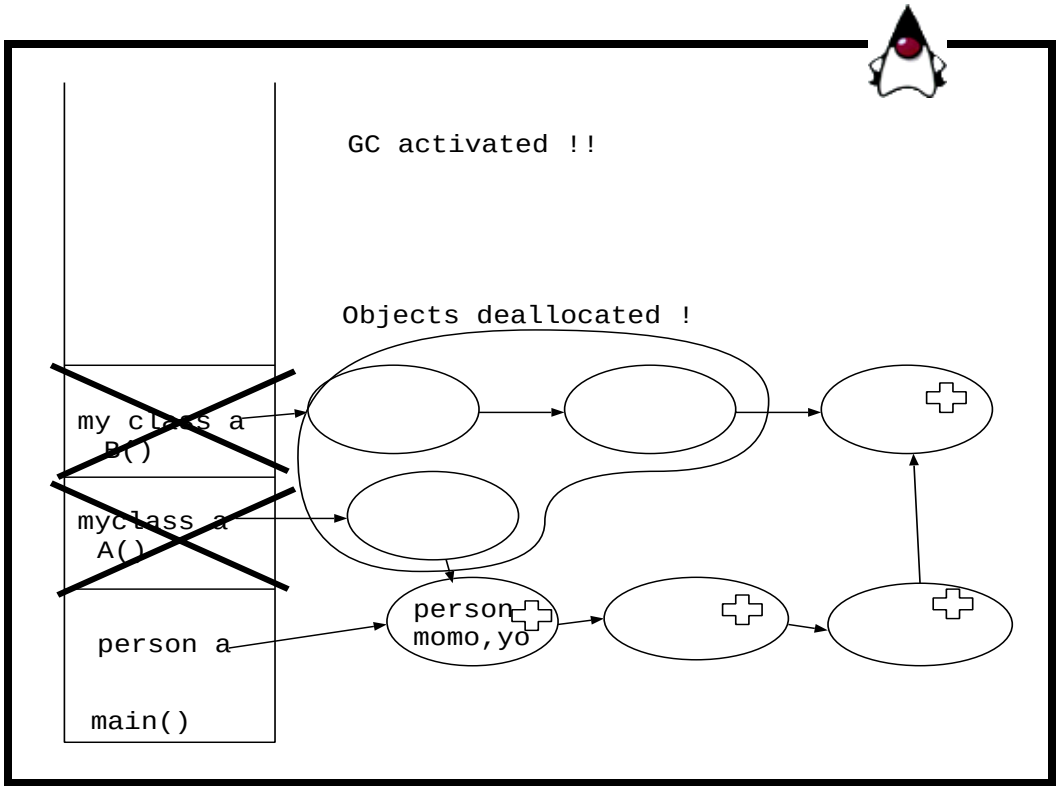
Slide 45

Supposons qu'aprés trois appel de méthodes, on ait développé le graphe d'instance ci-dessus.



Slide 46

Fin du deuxième et troisième appel, à ce moment le GC se déclanche (pour plein de mauvaises raisons). Le GC peut être un thread asynchrone dévlanché soit à intervalle régulier (mais c'est nul) soit quand la mémoire dépasse un certain pourcentage de saturation (c'est mieux) soit directement appelée par le programme (faut avoir de bonnes raisons pour faire ça ...)



Slide 47

Le mark&sweep se déclanche. La seule racine c'est "Person a" dans main. Les objets atteignables sont marqué d'une croix (mark). Les objets non-atteignale sont libéré à la deuxième passe (sweep).

GC Mark&Sweep: Compacting

- Several strategies:
  1. An allocation table and we try to fill up the gaps after the deallocation..
  2. All the living objects during the “mark” phase are copied in a new memory space (2x bigger in memory but very speed). The original memory space is available for the next execution of the garbage collector.

Slide 48

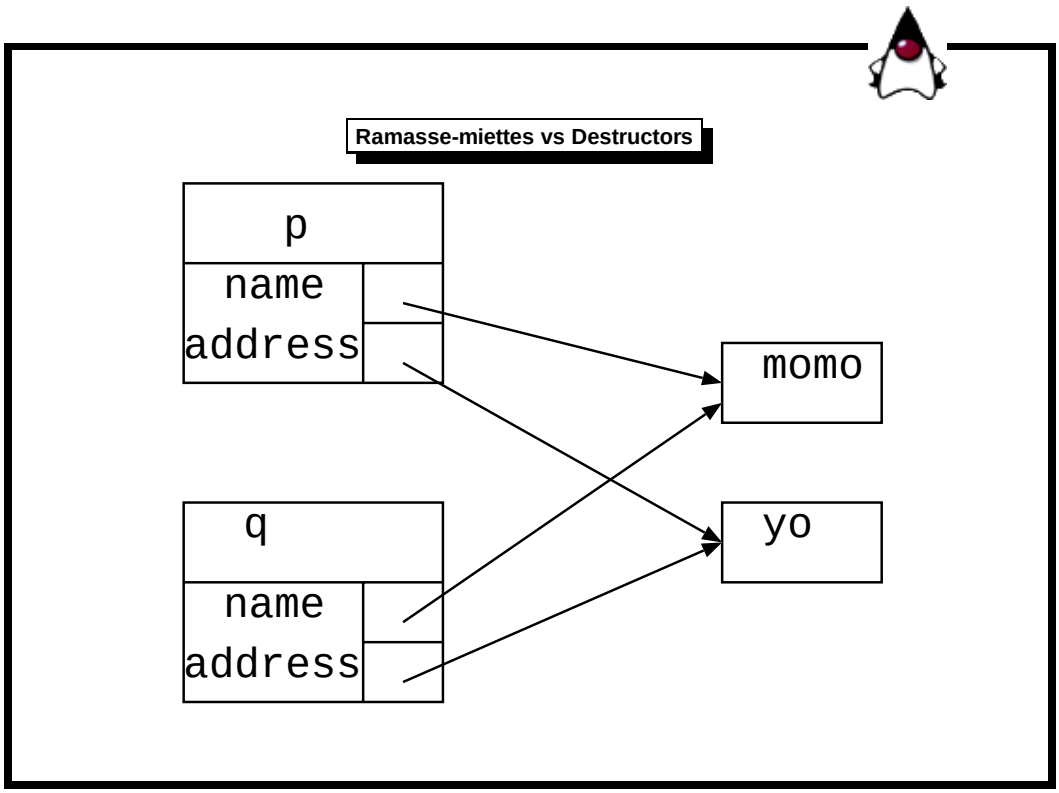
Garbage Collectors Vs Destructors

examples/PersonDemo.java

```
class Person {
  String name, address;
  public Person(String namep, String addressp) {
    name=namep;
    address=addressp;
  }
  public void print() {
    System.out.println(name+", "+address);
  }
}
class PersonDemo {
  public static void main(String[] args) {
    String s1=new String("momo");
    String s2=new String("yo");
    Person p=new Person(s1,s2);
    Person q=new Person(s1,s2);
    p.print();
    q.print();
  }
}
```

Slide 49

Ce programme produit un core-dump en C++ !! Pour les gens qui font du C++

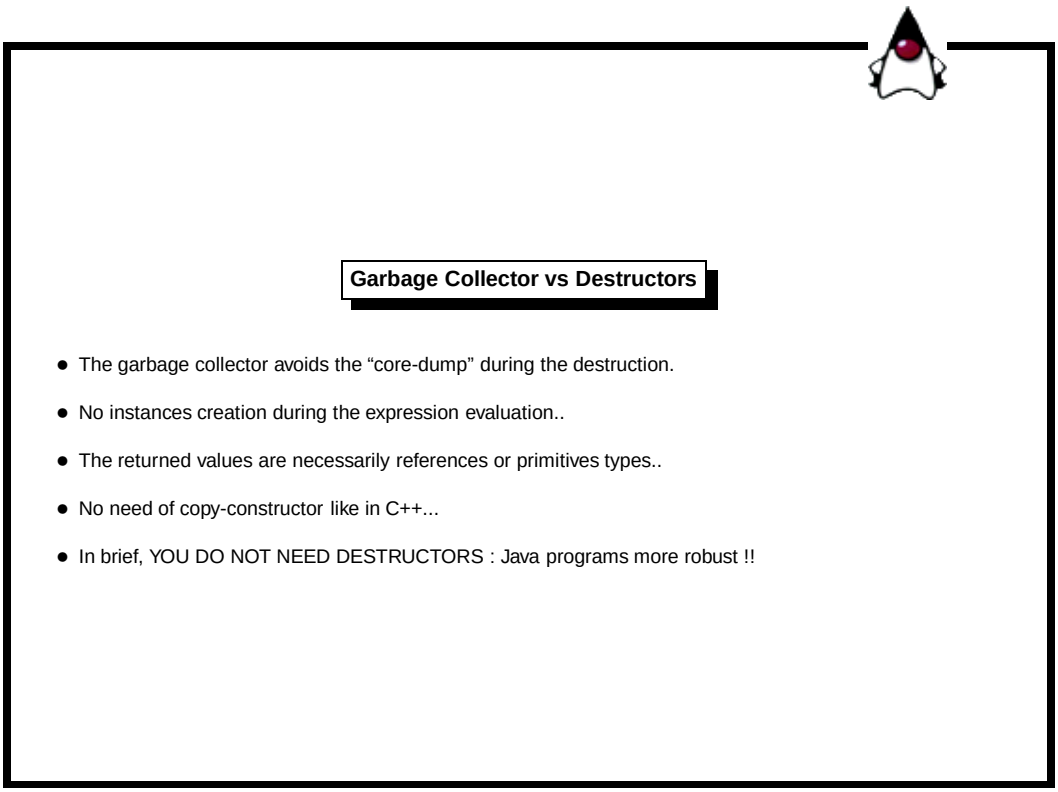


Slide 50

Le dessin précédent est un cas typique où il faut faire attention en C++. Si le copy constructor par défaut est appelé (recopie bit à bit de l'état d'un objet) on peut facilement aboutir à une situation similaire à celle du dessin. Or il y a toutes les chance que le destructeur ait été codé pour détruire les chaines implantant nam et adresse. Si par exemple q est un objet transitoire créé par C++, le destructeur va être appelé pour q de manière transparente. Après cela les accès au nom et adresse de p seront incorrects.

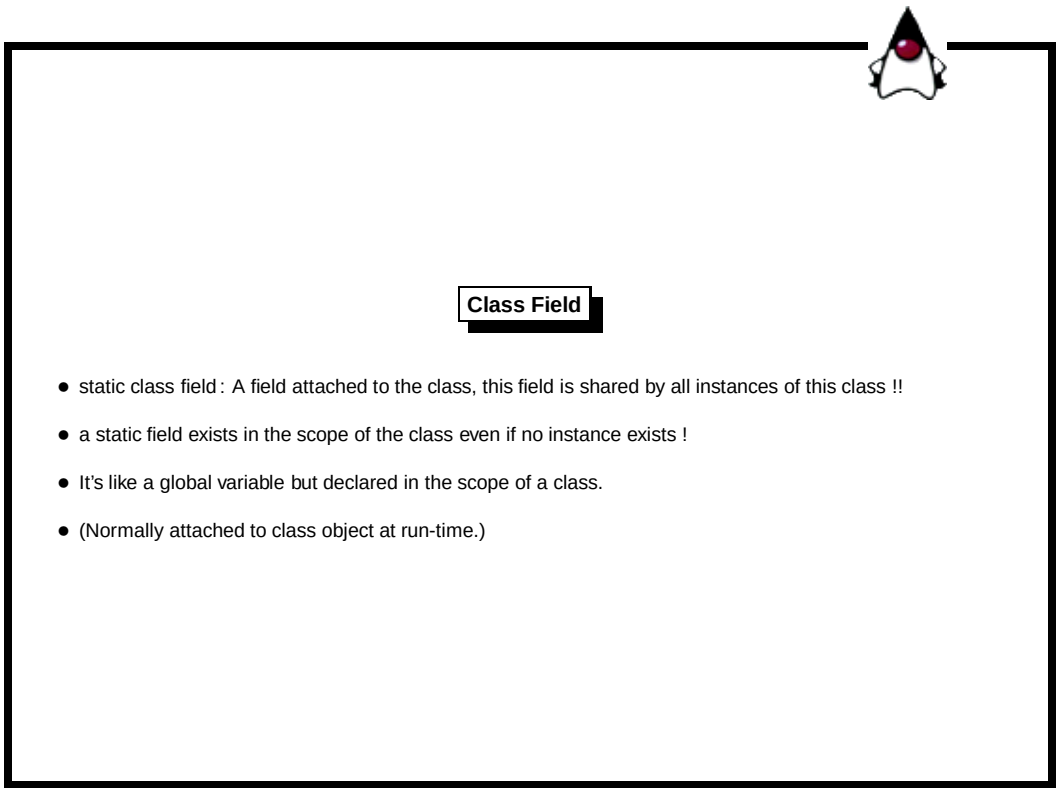
En java il n'y a pas à se préoccuper de cela. Les chaines seront éventuellement ramassées par le GC quand plus aucun objet ne les référencera.

Il est possible de se poser la question pour une classe donnée de ce que veut dire "faire une copie d'une instance". La classe `Object` définissant l'objet de base Java possède d'ailleurs une méthode `clone`.



Slide 51

2.3.6 Class Fields, Class Methods



Slide 52

C'est donc finalement un variable globale mais définie dans le contexte d'une classe.

examples/classandobjects/static.java

```
class Point {
  public static int nbpoint=0;                                    // that's a counter
  private int x,y;
  public Point(int xp, int yp) {
    nbpoint++;                                                    // increment the point counter
    x=xp; y=yp;
  }
  public int nbpoint() {
    return nbpoint;
  }
}
class SimpleStaticDemo {
  public static void main(String[] args) {
    // direct call to static variable
    System.out.println(Point.nbpoint());

    Point p=new Point(5,6);
    // calling static method and field with an object
    System.out.println(p.nbpoint);
  }
}
```

Slide 53

Class Methods

- `static` method can be called without a reference to an object. The receiver is the name of the class
- That's nearly a function but declared in the scope of a class (and data hiding rules are checked!)
- Beware: A class method cannot access an instance field, "this" makes no sense in the body of a class method !

Slide 54

examples/classandobjects/staticmethod.java

```
class Point {
  private static int nbpoint=0;                          // that's a counter (private now)
  private int x,y;
  public Point(int xp, int yp) {
    nbpoint++;                                           // increment the point counter
    x=xp;
    y=yp;
  }
  public static int nbpoint() {
    System.out.println(x);                               // error ! object variable !
    return nbpoint;
  }
}
class StaticMethod {
  public static void main(String[] args) {
    System.out.println(Point.nbpoint());

    Point p=new Point(5,6);
    System.out.println(p.nbpoint());                     // ok but dangerous
  }
}
```

Slide 55

Class Method, Class Fields

- `main()`: inevitably a class method
- `System.out`: a class field for STDOUT (PrintStream) initialized when loading the "System" class.
- The System is automatically loaded when the JVM start.

examples/hello.java

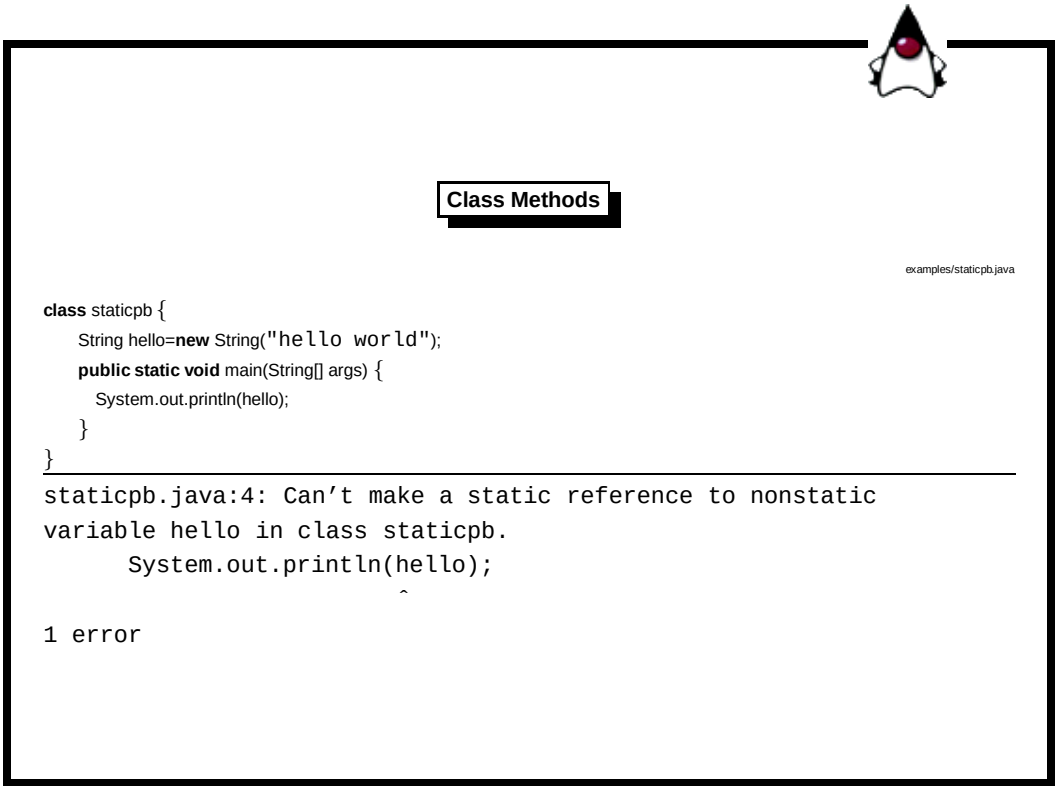
```
class helloworld {
    public static void main(String[] args) {
      System.out.println("hello world");
    }
}
```

Slide 56

Le plus bel exemples sur les variables de classes et méthode de classes !

Hé oui ! forcément pour commence un programme, il n'y a pas d'instances, on appelle donc une méthode de classe ...

"System.out" oh la belle variable de classe que voila ! `public static final PrintStream out` dans java.lang.system



Slide 57

Bon, c'est pas la peine de référencer des variables d'instances qui ne sont pas instanciées. Et c'est pas la peine non plus de mettre des static partout pour faire en sorte que ça compile !! Le mieux : la classe contenant "main" n'est pas une classe de l'application, elle est juste la pour contruire les instances des classes applicatives.

La bonne méthode :

1. Écrire ses classe applicatives,
2. Écrire une nouvelle classe contenant le main après !

Constants

- The `final` modifier specifies that a field is constant.
- This field have to be initialized at declaration-time and cannot be changed later ...

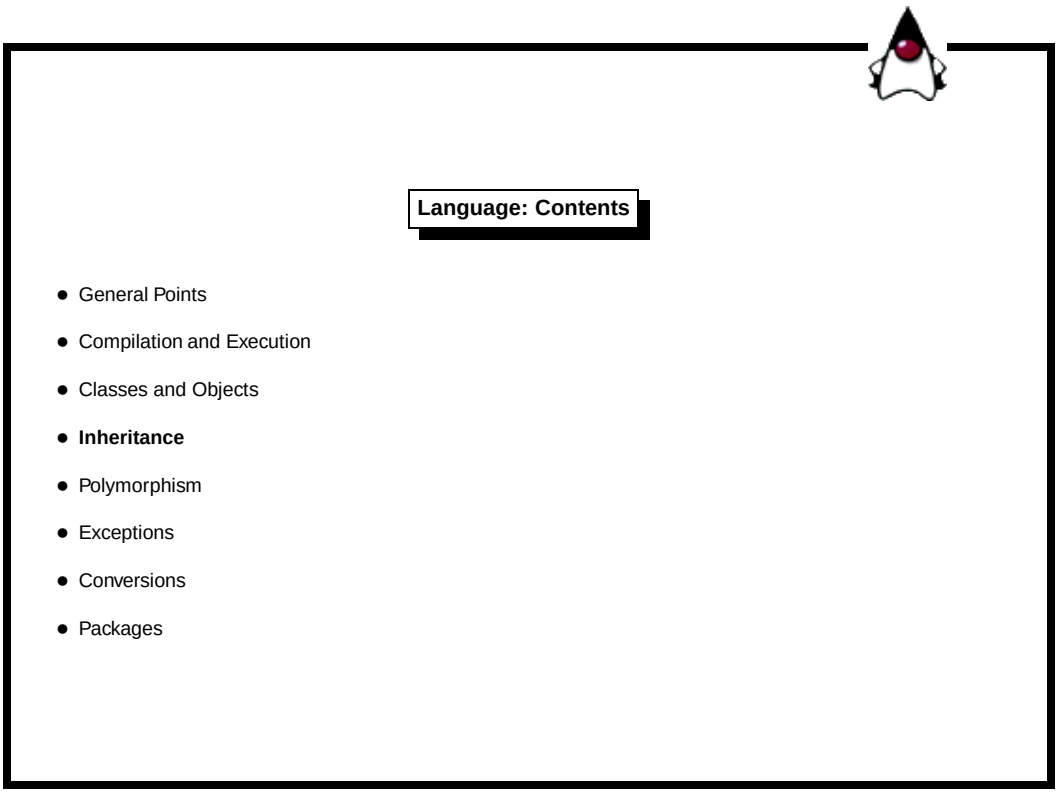
Slide 58

examples/classandobjects/final.java

```
class Point {
   private int x;
   private int y;
   // main constructor
   public Point( int x, int y ){
      this.x=x;
      this.y=y;
   }

   // final fields : have to be initialized
   // cannot be changed later.
   final static Point origin = new Point(0,0);

}
```

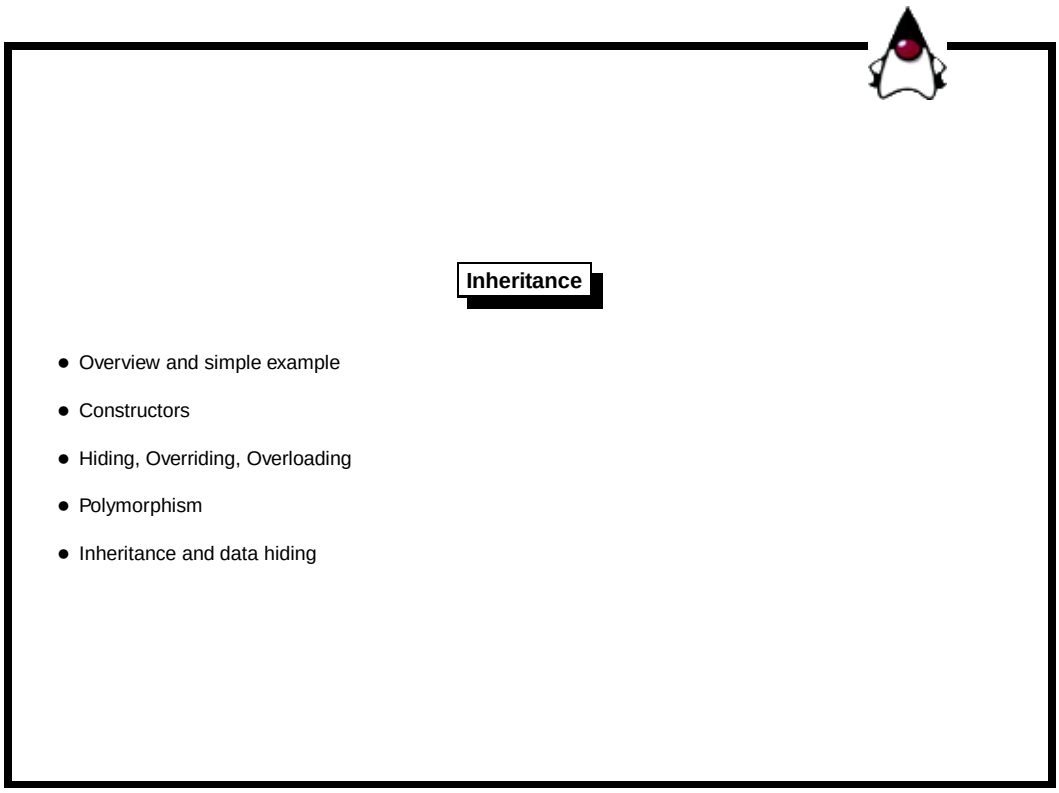
Slide 59

attention sur cet exemple : le contenu de l'objet origin peut être changé mais la référence sur l'objet ne peut plus l'être (encore une histoire à la const ;-D).



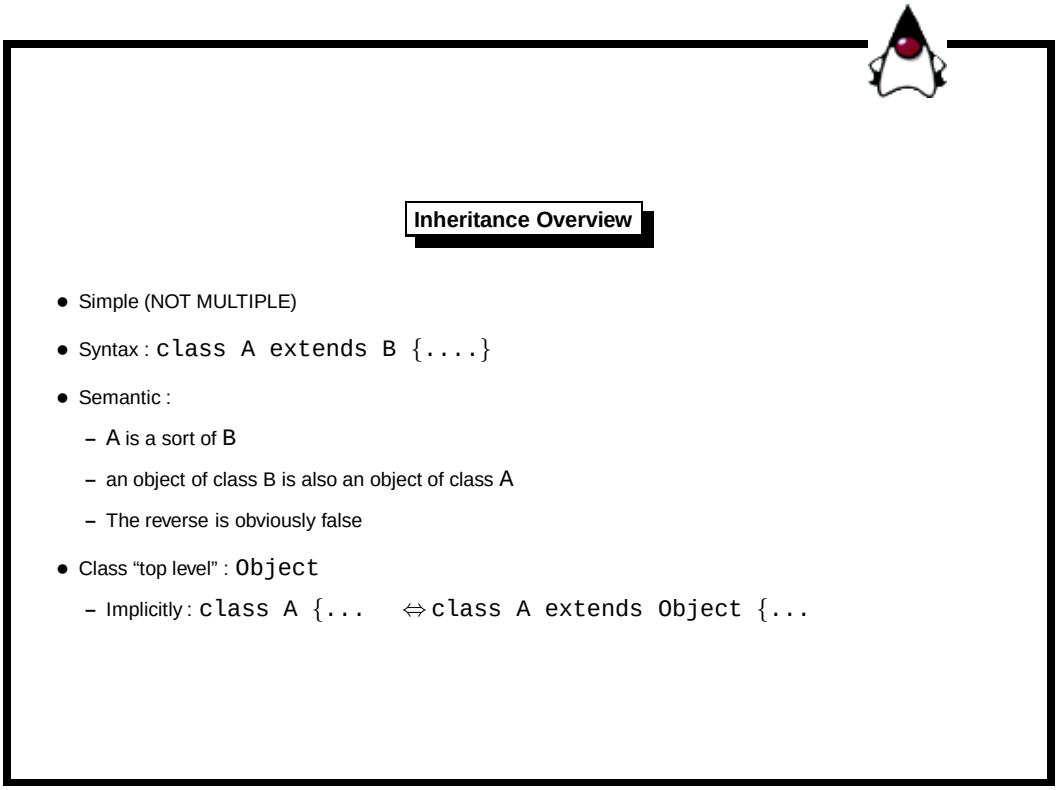
Slide 60

2.4 Inheritance



Slide 61

2.4.1 Overview and Simple Example



Slide 62

- **PAS** de trucs globaux et obscurs à la C++ sur les héritages : `protected` qui modifient globalement les accès aux members; `virtual` pour résoudre les problèmes d'héritages multiples et de NON liaison dynamique.

Example

examples/heritage/Client.java

```
class Person {
  String name, address;
  public Person(String namep, String addressp) {
    name=namep; address=addressp; }
  public void print() { System.out.println(name+", "+address); }
}
class Client extends Person {
  int amount=0;
  public Client(String namep, String addressp,int amountp) {
    super(namep,addressp);
    amount=amountp;
  }
  public void print() { System.out.println(name+", "+address+", "+amount); }
}
class ClientDemo {
  public static void main(String[] args) {
    Client c=new Client("momo","yo",100); c.print();
  }
}
```

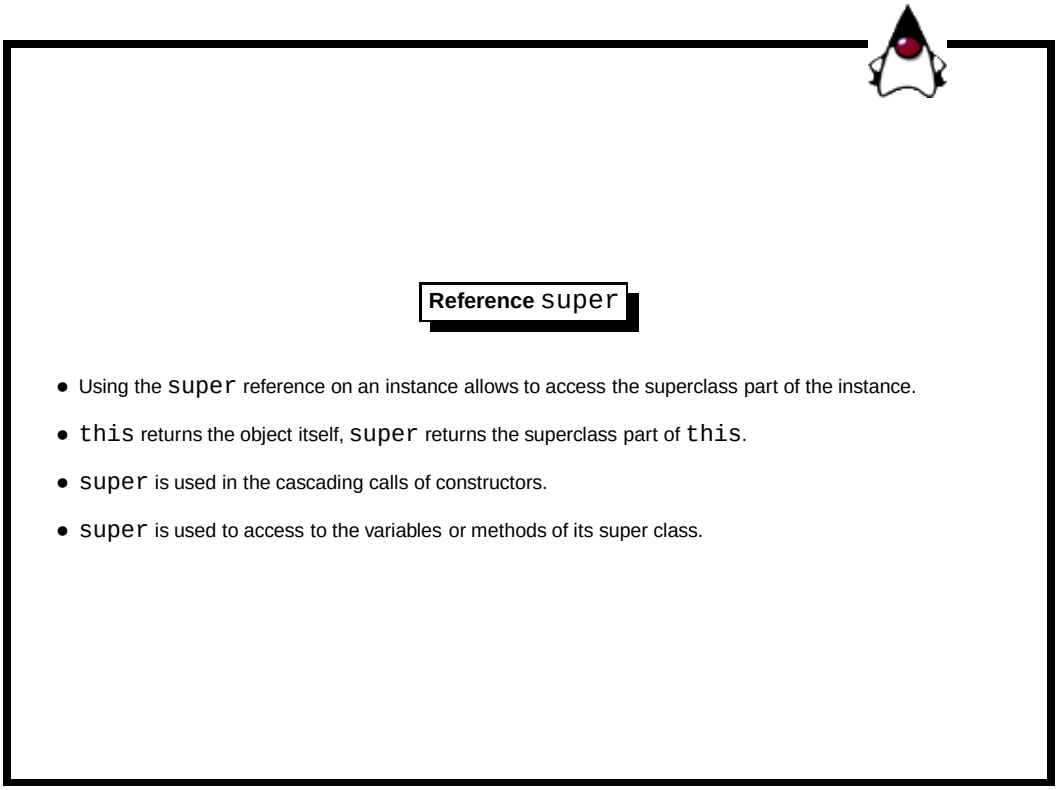
Slide 63

Beaucoup de choses à dire sur cet exemple, d'aboirt sur la structure :

1. Un petit coup d'oeil sur la syntaxe "extends".
2. un gros coup d'oeil sur le constructeur de client et sur la syntaxe super ! (on y vient juste après)
3. Enfin bien remarquer l'overriding sur print.

Prendre son temps et bien dérouler l'exécution de ce programme en partant de main().

2.4.2 Inheritance and Constructors



Slide 64

- Bien entendu la classe `Object` ne dispose pas de la référence `super`.

Example of reference by `super`

examples/TestChpFinal.java

```
class A { final int c=10; }
class TestChpFinal extends A {
  int c= super.c+10;
  void print(){
    System.out.println(c);
    System.out.println(super.c); }
  public static void main( String[] args ){
      TestChpFinal b=new TestChpFinal();
      b.print(); }
}
```

Slide 65

Résultat de l'exemple :

```
20
10
```

Inheritance and Constructors

- The initial state of an object is defined by calling the constructor. In the case of a construction by inheritance, it's necessary to have **discipline** of calls
- **TOP-DOWN Construction** : The super-constructor have to called before executing the body of the current constructor.

Slide 66

Inheritance and Constructors

In a constructor of a class A :

- The call of `super(...)` is used in the first line and invokes a constructor of the super class.
- The initialisation of the instance variables introduced by A;
- And the execution of the rest of the code of the considered constructor.
- By default, if a call of `this(..)` or `super(...)` (is in the first line of a constructor), the compiler introduces implicitly a call to a `super()`

Slide 67

Constructors

examples/heritage/Client2.java

```
class Person {
  String name, address;
  public Person(String namep, String addressp) {
    name=namep; address=addressp; }
  public void print() { System.out.println(name+", "+address); }
}

class Client extends Person {
  int amount=0;
  public Client(String namep, String addressp,int amountp) {
    amount=amountp;
  }
  public void print() { System.out.println(name+", "+address+", "+amount); }
}

class ClientDemo {
  public static void main(String[] args) {
    Client c=new Client("momo","yo",100); c.print();
  }
}
```

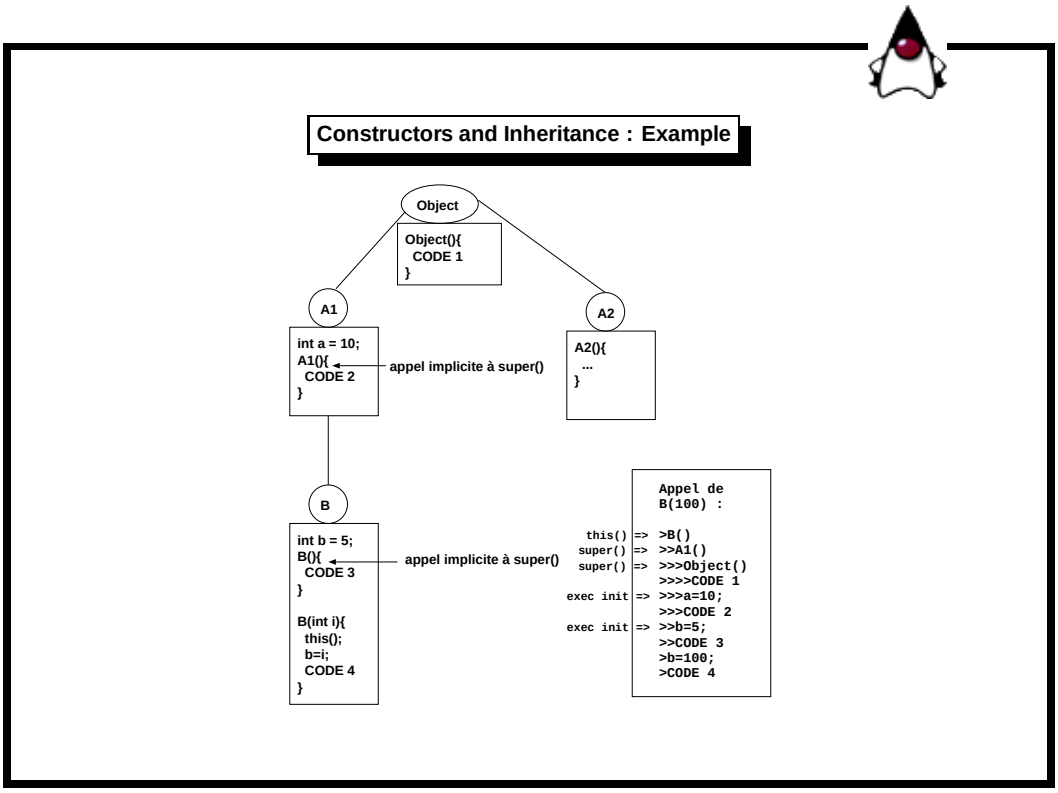
Slide 68

Constructors

```
Client.java:14: No constructor matching Person() found in class Person.
  public Client(String namep, String addressp,int amountp) {
         ^
1 error
```

Slide 69

- Bien entendu la class `object` ne dispose pas de l'appel `super(...)`.
- Ne pas confondre l'appel `super(...)` avec la référence `super`.



Slide 70

2.4.3 Hiding, Overriding, Overloading...

Hiding, Overriding, Overloading

Hiding : If a class defines a **c**lass method, this method hides all methods with the same signature in super-classes.

Overriding : If a class defines an **i**nstance method, this method overrides all methods of same signature in super-classes

Overloading : If two methods of the same class have the same name but with different signatures, there is overloading

Slide 71

examples/heritage/Client3.java

```
class Person {
  static int nbpers=0;
  String name, address;
  public Person(String namep, String addressp) { name=namep; address=addressp; nbpers++; }
  public void print() { System.out.println(name+", "+address+", "+this.getCard()); }
  static int getCard() { return nbpers; }}
class Client extends Person {
  static int nbclient=0;
  int amount=0;
  public Client(String namep, String addressp) { this(namep,addressp,0); }
  public Client(String namep, String addressp,int amountp) { super(namep,addressp); amount=amountp; nbclient++; }
  public void print() { System.out.println(name+", "+address+", "+amount+", "+this.getCard()); }
  static int getCard() { return nbclient; }}
class Client3Demo {
  public static void main(String[] args) {
    Person p=new Person("toto","titi"); p.print();
    Person c=new Client("momo","yo",100);
    System.out.println(c.getCard()); c.print();
    Client d=new Client("placid","muzo"); d.print();
    System.out.println(Person.getCard());} }
```

Slide 72

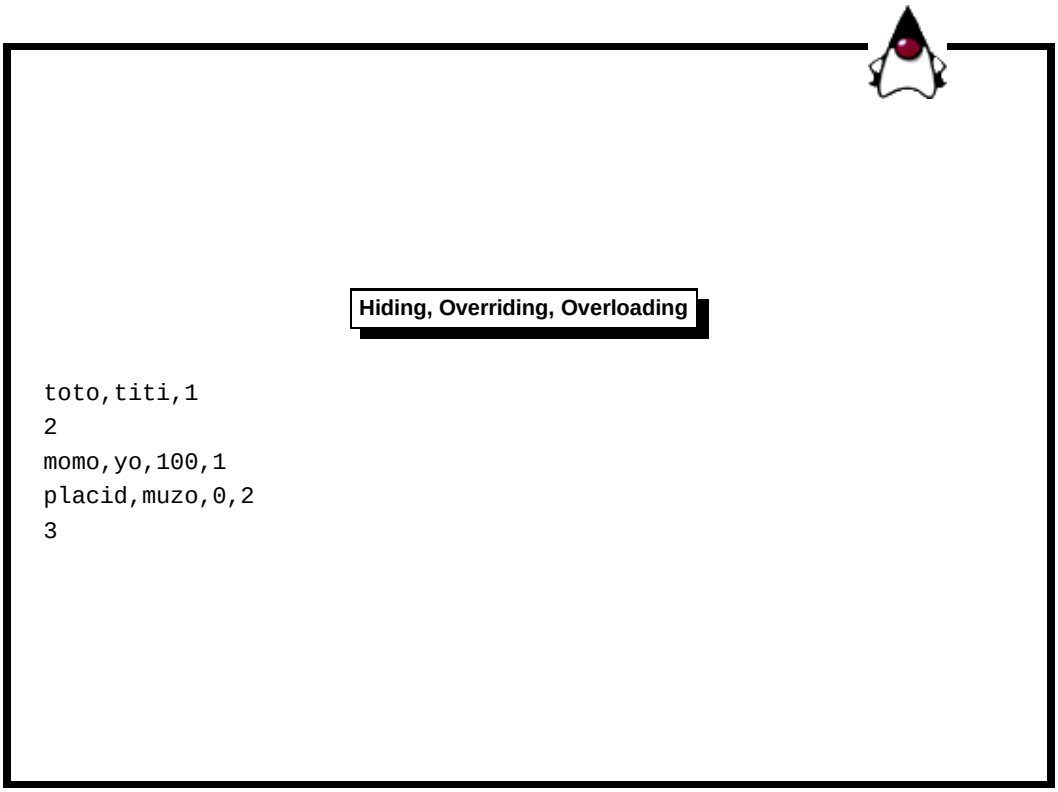
Il y a dans cet exemple :

Hiding sur getCard()

overriding sur print()

overloading sur les constructeurs de Client.

Qu'est ce qu'imprime le programme ?



Slide 73

- Beware ! there is no dynamic binding for class methods
- Note the difference between c.getCard() static binding, and c.print(), dynamic binding ...
- Note that ((Person)d).print() changes nothing, the dynamic type is *ad vitam eternam* of class Client.

Dynamic binding/Static Binding

- Dynamic type is used for an instance method call
- Static type is used for class method call
- Beware :

```
Person c=new Client("momo","yo",100);
c.getCard();
```

Person::getCard if getCard is static, Client::getCard() otherwise !!

Slide 74

Dynamic Binding and Constructors

- Dynamic binding is always used, even during construction (unlike C++)
- ⇔ cascading calls of constructors does not reduce dynamic bindings...
- The dynamic type of `this` in a constructor is the type requested in the new syntax and not the class of the constructor.
- ⇔ possible “observation” of instance fields not yet initialized
- that’s not true in C++ (well known trap !)

Slide 75

Dynamic Binding and Constructors, Ex

examples/TestLDetCons.java

```
class A {
    private String yo="yo";                                          // init
        String getString(){ return yo;}
        A(){ System.out.println(yo);
        System.out.println(getString());                  // <<== Dynamic Binding
        }}
class B extends A {
    private String momo="momo";                                      // init
        String getString(){ return momo;}
        B(){ super();
        System.out.println(momo);
        System.out.println(getString());
        }}
class TestLDetCons {
    public static void main( String[] args ){
        B b = new B();
    }}
```

- **Execution ? ...**

Slide 76

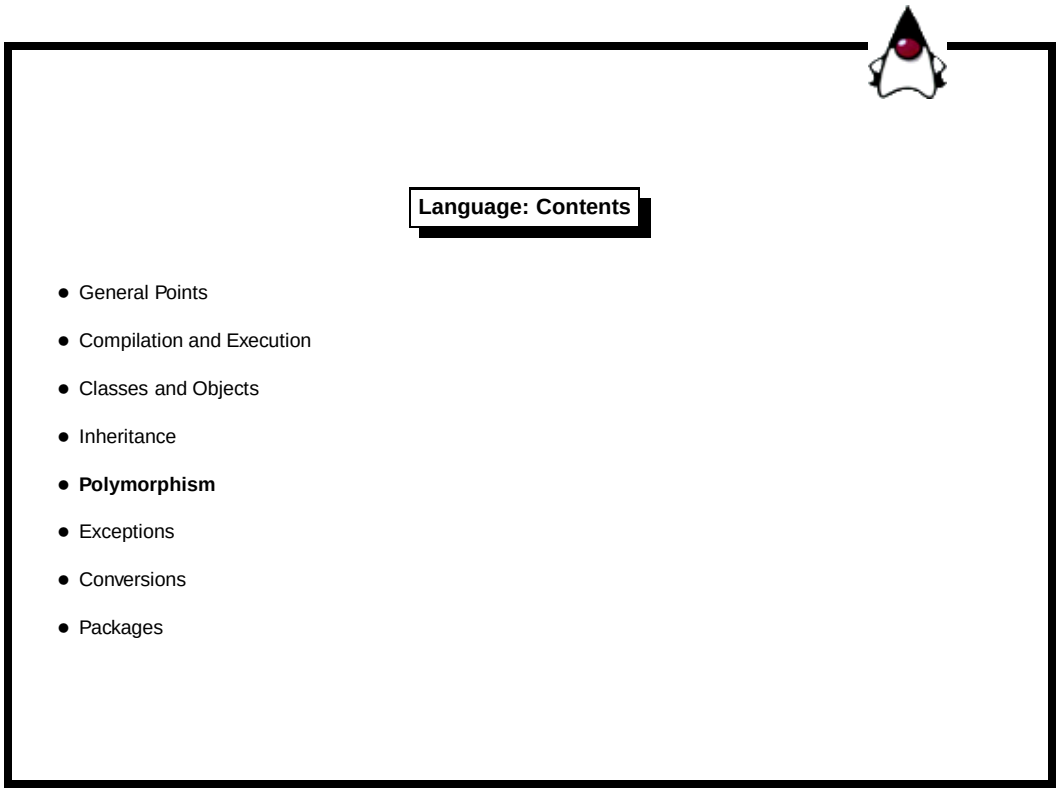
Dynamic Binding and Constructors, Ex

```
yo
null
momo
momo
```

- We build an instance of B ⇒ the dynamic type of `this` in constructors is B.
- The call to `super` is done **before** initialization (=`"momo"`)
- Dynamic binding for `getString`

Slide 77

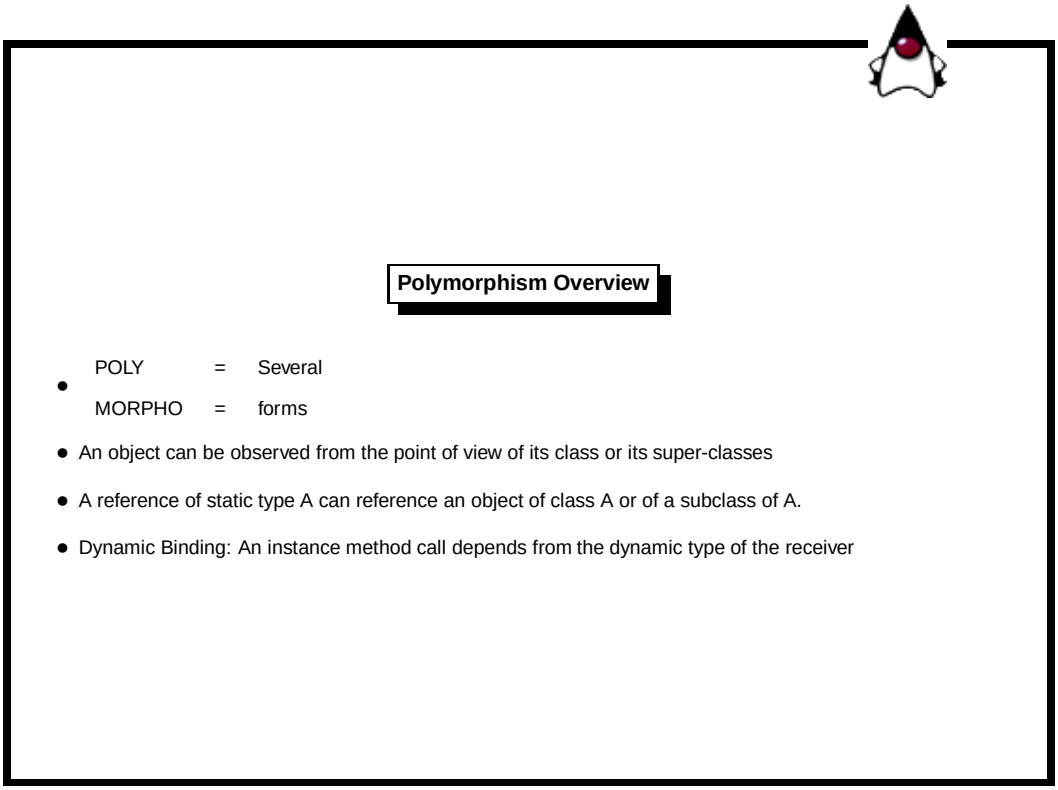
- L'exemple construit une instance de `B`. Donc lors de l'exécution des constructeurs (de `A`, de `B`) la classe effective de l'instance courante, `this` sera `B`.
- Si on se reporte au transparent 70 page 70, on sait que les expressions d'initialisations (commentaires `//int`) des variables d'instances ajoutées par une classe sont exécutées par un constructeur de cette classe **après** l'appel à `super()` et avant le reste du code du constructeur.
- On rappelle aussi que par défaut le compilateur fait un appel à `super()` tout au début d'un constructeur. Dans l'exemple c'est le cas pour le constructeur `A()`.
- On voit qu'il est possible par liaison dynamique (ici pour la méthode `getString` (commentaire `LD`) sur une instance de classe effective à l'exécution : `B`) pour un constructeur d'une super classe d'accèder à des variables d'instance introduites par une sous classe avant l'évaluation de leur expression d'initialisation.



Slide 78

2.5 Polymorphism

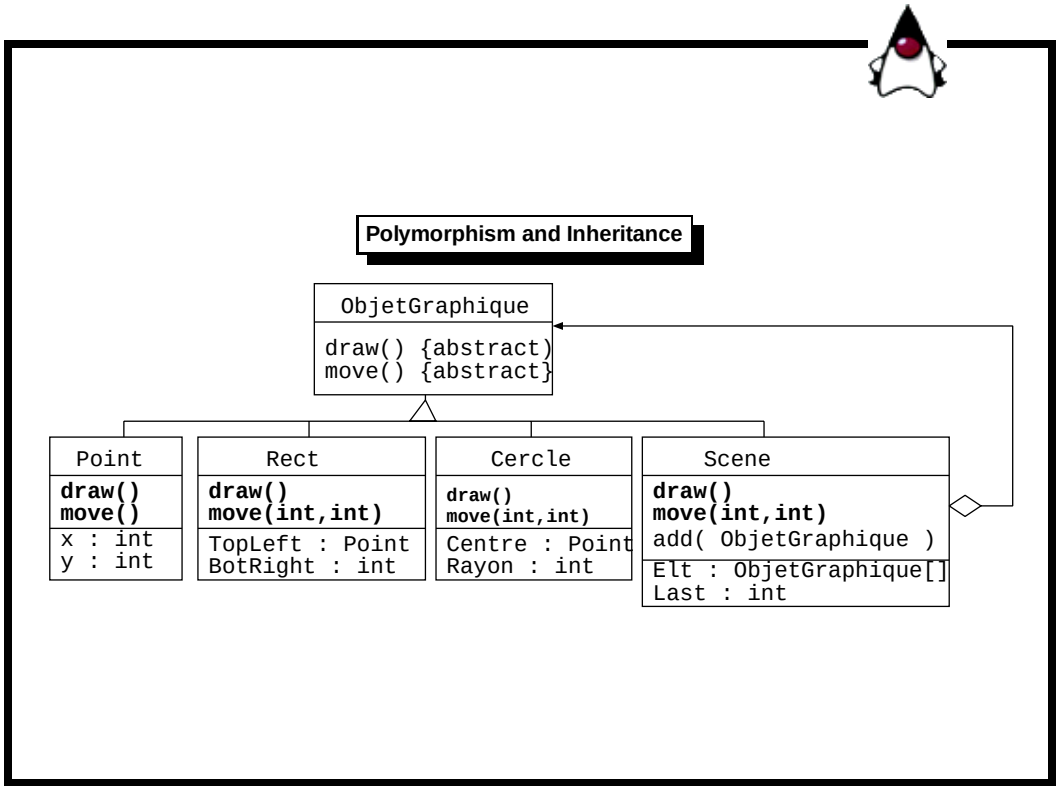
2.5.1 Overview



Slide 79

- Une référence d'objet peut bien entendu aussi être un élément de tableau.

2.5.2 Polymorphism, inheritance and abstract classes



Slide 80

Présentons directement cette notion de polymorphisme d'héritage par un exemple plus que classique.

On définit des `ObjetGraphique`s que l'on peut tracer et déplacer. `Scene` groupe des `ObjetGraphique`s et les manipule en temps que tels. Par contre pour une instance donnée c'est bien les `draw`, le `move` de `Cercle` ou `Rect` ou `Scene` que l'on veut atteindre selon la nature effective de l'instance (méthodes en **gras**).

La figure est *OMT-like*, mais l'analyse serait erronée : certains attributs (`Elts, Centre, ...` sont en fait l'implantation de **relations**.

examples/polymorphisme/ObjetGraphique.java

```
public abstract class ObjetGraphique {
    public abstract void draw();
    public abstract void move( int dx, int dy );
}
```

examples/polymorphisme/Scene.java

```
class Scene extends ObjetGraphique {
  private int Last;
  private ObjetGraphique[] Elts;
  public Scene( int MaxOG ) { Elts= new ObjetGraphique[MaxOG]; Last= -1; }
  public void draw(){
    for( int i=0; i≤ Last; i++){
      Elts[i].draw();                                          // Dynamic binding
    }
  }
  public void move ( int dx, int dy ){
    for( int i=0; i≤Last; i++) { Elts[i].move(dx,dy);}   // Dynamic binding
  }
  public void add( ObjetGraphique unOG ){
    if (Last==Elts.length) { }   // Exception
    else{ Last++; Elts[Last]=unOG; } }}
```

Slide 81

examples/polymorphisme/TestOG.java

```
class TestOG {
    public static void main( String[] args ){
        Scene s1 = new Scene(10);
        Scene s2 = new Scene(10);

        s1.add( new Cercle( new Point(10, 10), 30 ));
        s1.add( new Rect( new Point(30, 30), new Point(100, 100)));
        s1.draw();
        s1.move(1, 1);
        s1.draw();

        s2.add(s1);                                          // sub-class of ObjetGraphique
        s2.add( new Rect( new Point(-10, -10), new Point(-5, -5)));
        s2.draw();
        s1.move(1,1);
        s2.draw();
    }
}
```

Slide 82

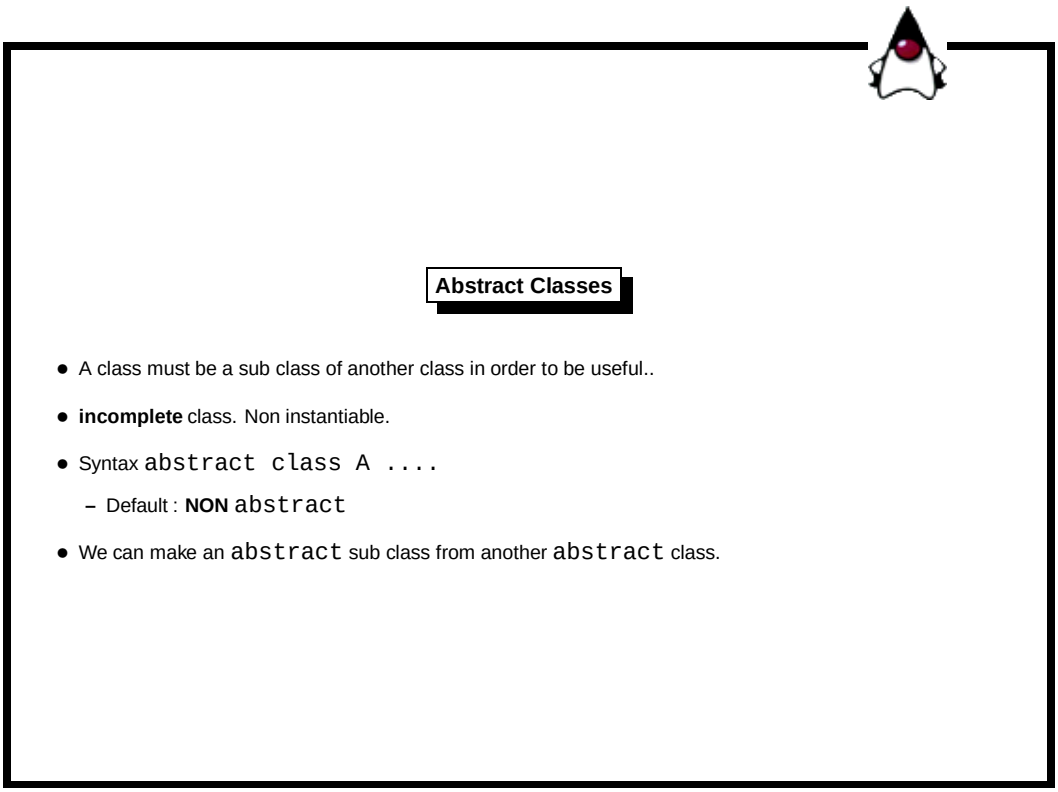
Voici le résultat de l'exécution:

```
[ Scene ------------
[ Cercle : Centre= (10,10)R=30 ]
[ Rect : TL= (30,30) , BR= (100,100) ]
] --------------
[ Scene ------------
[ Cercle : Centre= (11,11)R=30 ]
[ Rect : TL= (31,31) , BR= (101,101) ]
] --------------
[ Scene ------------
[ Scene ------------
[ Cercle : Centre= (11,11)R=30 ]
[ Rect : TL= (31,31) , BR= (101,101) ]
] --------------
[ Rect : TL= (-10,-10) , BR= (-5,-5) ]
] --------------
[ Scene ------------
[ Scene ------------
[ Cercle : Centre= (12,12)R=30 ]
[ Rect : TL= (32,32) , BR= (102,102) ]
] --------------
[ Rect : TL= (-10,-10) , BR= (-5,-5) ]
```

On voit donc bien que bien qu'un objet `Scene` manipule des objets qu'il considère être des `ObjetGraphique` ; mais que les `Cercle` restent des cercles, les `Rect` des rectangles...

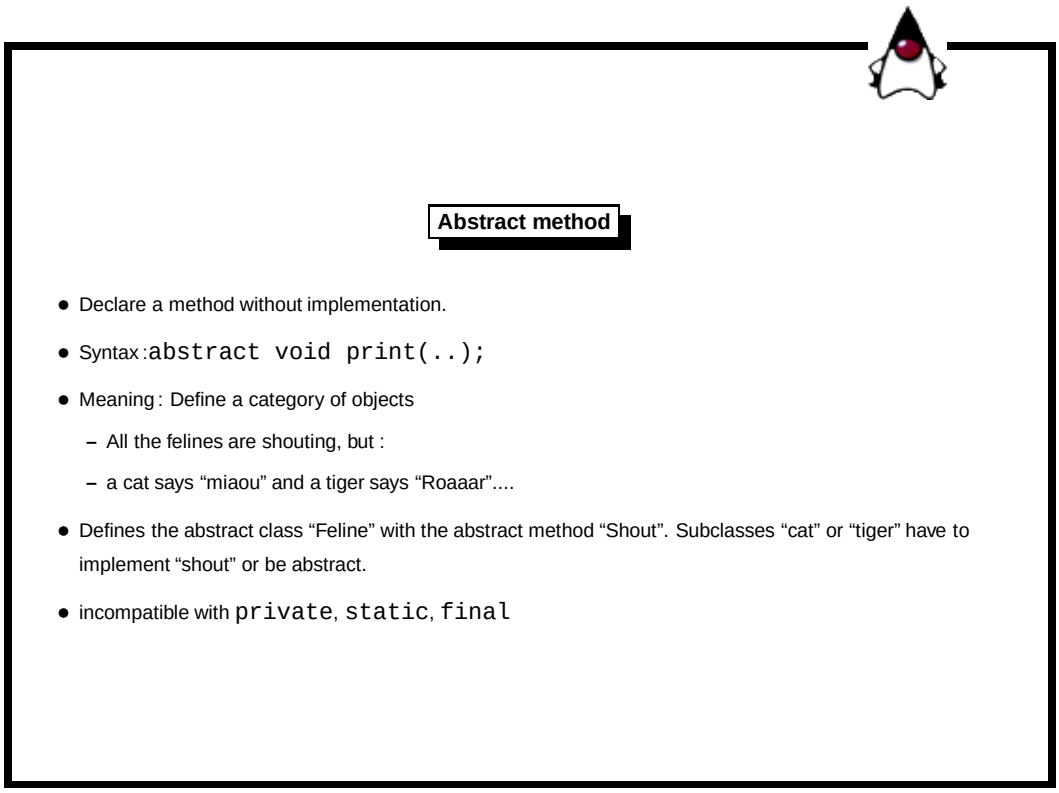
On pourrait mettre ici un joli dessin des objets en représentation héxagonale. Ca serait d'ailleurs un bon moyen de monter que polymorphisme et encapsulation sont très complémentaires.

2.5.3 Abstract and Final Classes



Slide 83

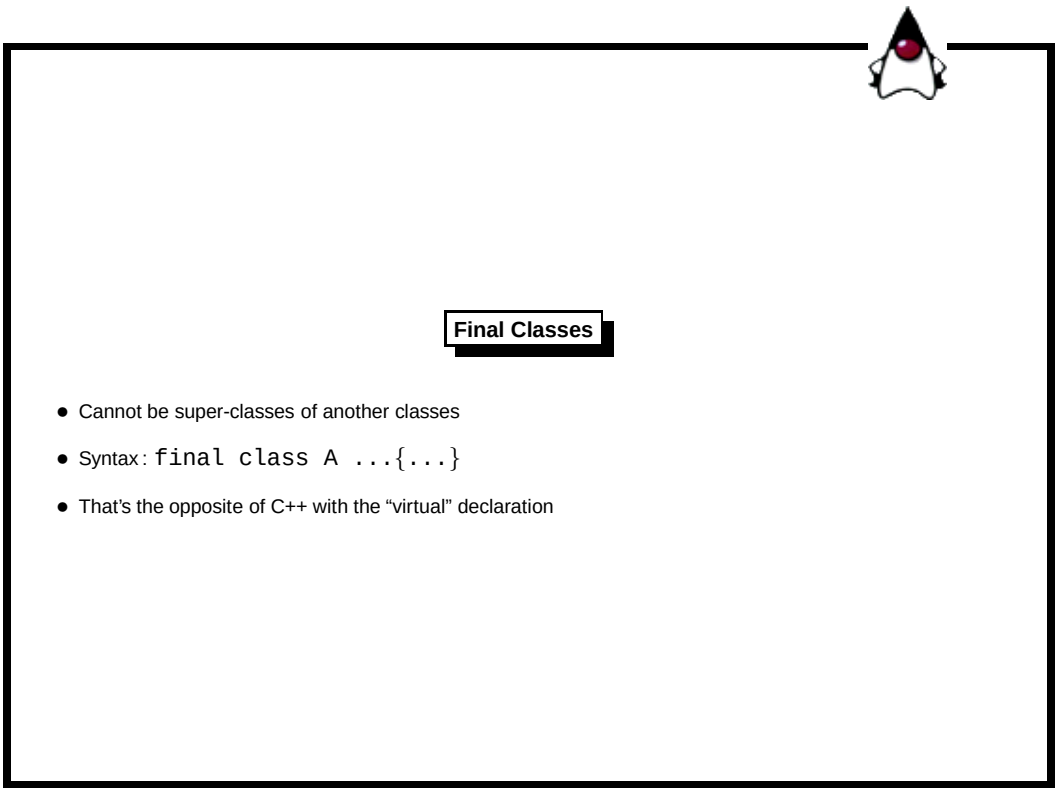
- Si on tente un `new` sur une classe `abstract` le compilateur doit générer une erreur.
- Une classe `abstract` est proche des *pure virtual classes* de C++ : celles qui comportent une méthode `virtual ... =0;`.
- Si une classe comporte une **méthode** `abstract` elle doit comporter le *modifier* `abstract`. Bref on déclare une classe comme `abstract` lorsque qu'elle n'est qu'une ébauche de classe, elle devra en tout état de cause être complétée par **sous classement** pour que des variables de référence d'instance de cette classe puisse être affectées (en créant des instances de sous classes non `abstract`).



Slide 84

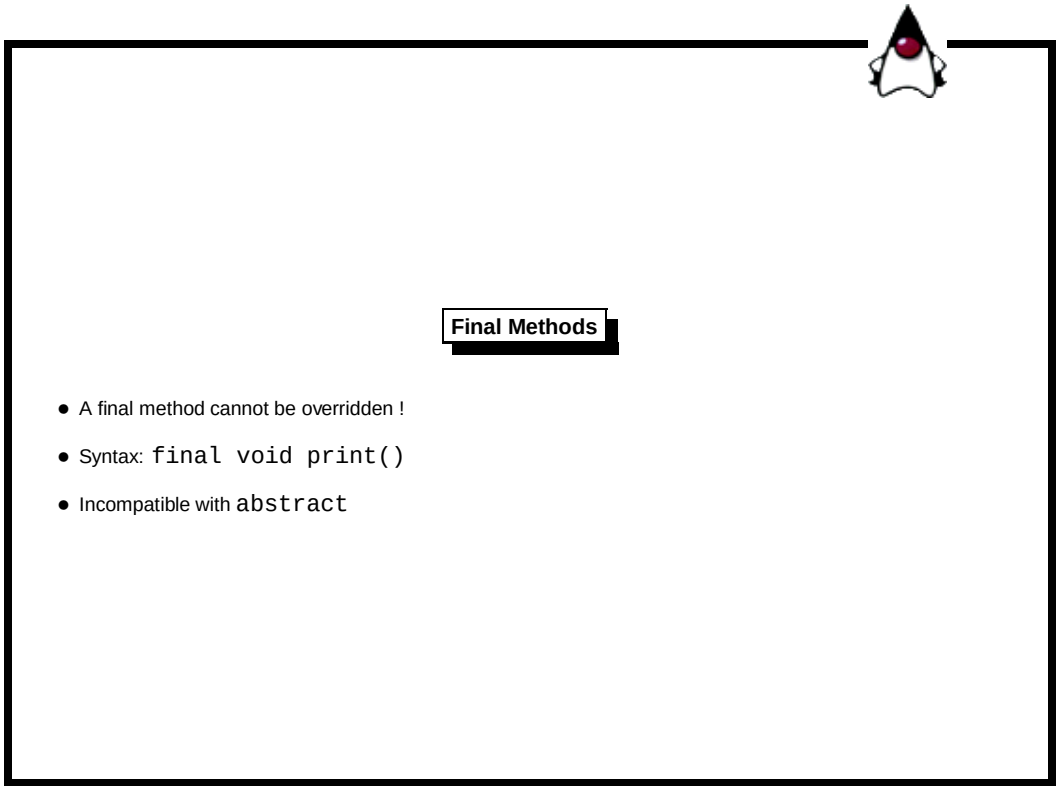
- Pour expliciter autrement l'effet d'une déclaration de méthode `abstract` on peut reprendre [JG96][page 158] : si une méthode m est déclarée comme `abstract` dans une classe `abstract` A il faut que toute sous classe **non** C `abstract` de A puisse trouver une classe B telle :
  - B est une super classe de C (possiblement C elle même),
  - B est une sous classe de A,
  - B fournit une déclaration de m qui n'est pas `abstract`, donc avec un corps qui l'implémente. Cette déclaration doit pouvoir être accessible par C (voir les *modifiers* d'accés des méthodes).
- Il est donc stupide de déclarer une méthode comme `abstract` et :
  - `private` : si il y a pas d'héritage elle ne peut pas être vue par une sous classe ! Ceci devrait être assuré par le compilateur. **Ce n'est pas le cas dans le jdk 1.1b2**.
  - `static` : un appel `static` n'utilise pas de liaison dynamique, une classe doit donc obligatoirement fournir un corps pour une telle méthode. Ceci devrait être assuré par le compilateur.
  - `final` : une méthode `final`. Dans l'explication précédente la classe B ne peut donc pas être trouvée. Ceci devrait être assuré par le compilateur. **Ce n'est pas le cas dans le jdk 1.1b2**
- C'est le `virtual ... =0;` de C++. Il n'y a pas l'équivalent en java de `virtual m() { code...}` puisque qu'une méthode java d'instance s'accède forcement par liaison dynamique. En java “tout est `virtual`. `abstract` indique réellement que la déclaration n'est qu'une simple **spécification abstraite**.

- C'est une erreur de compilation de mettre une méthode `abstract` dans une classe qui n'y est pas. L'inverse ne l'est pas : le concept de classe abstraite est **indépendant** de celui de méthode abstraite.
- Les restrictions `final`/`static` sont de la même veine qu'en C++ (rappel : C++ : liaison dynamique = `virtual` et `virtual` + `static` est incorrect).



Slide 85

- Une classe `final` ne peut pas être l'ancêtre de nouvelle classe. On assure dont par cette déclaration qu'une classe restera une **feuille** de l'arbre d'héritage.



Slide 86

- `final` : On peut noter d'une méthode `private` est **implicitement** `final` (il n'est donc pas recommendé de déclarer une méthode `private final`. Tout ceci n'a rien à voir avec les méthodes `const` de C++, qui n'ont pas d'équivalent en Java).
- À ne pas confondre avec le `final` appliqué sur les champs (support des constantes, accès en "lecture seulement").

examples/heritage/AbstractClient.java

```
abstract class Person {
  private String name, address;
  public Person(String namep, String addressp) { name=namep; address=addressp; }
  abstract void print();}

class Client extends Person {
  private int amount=0;
  public Client(String namep, String addressp,int amountp) {
    super(namep,addressp); amount=amountp;
  }
  public void print() { System.out.println(name+", "+address+", "+amount); }}

final class Employee extends Person {
  private int salary;
  public Employee(String namep,String addressp,int salaryp) { super(namep,addressp); salary=salaryp; }}

class chief extends Employee {}

class AbstractClientDemo {
  public static void main(String[] args) {
    Person p=new Person("Dicentime","bougredane");
    Client c=new Client("momo","yo",100); }}
```

Slide 87

La liste des erreurs dans l'exemples ci-dessus :

1. `AbstractClient.java:21: class Employee must be declared abstract and not final. It does not define void print() from class Person.`
2. `//AbstractClient.java:29: Can't subclass final classes: class Employee`
3. `AbstractClient.java:35: class Person is an abstract class. It can't be instantiated.`

Bien noter toutes les erreurs sur l'exemple ci-dessus. Surtout le conflit sur employee (final vs abstract).

2.5.4 Polymorphism and Interfaces

Interfaces

- An interface is a restricted form of an abstract class, for example :

examples/interface/Drawable.java

```
interface Drawable {
  int draw_area_top = 10;
  int draw_area_left = 10;
  int draw_area_bottom = 100;
  int draw_area_right = 200;

  void drawme(int x, int y);
}
```

⇔

examples/interface/Adraw.java

```
abstract class Drawable {
  public static final int draw_area_top = 10;
  public static final int draw_area_left = 10;
  public static final int draw_area_bottom = 100;
  public static final int draw_area_right = 200;

  public abstract void drawme(int x, int y);
}
```

- It cannot be instantiated
- All methods are public and abstract
- All fields are final and static
- **A class can inherit from only one class, however...**
- **it can “implements” several interfaces.**

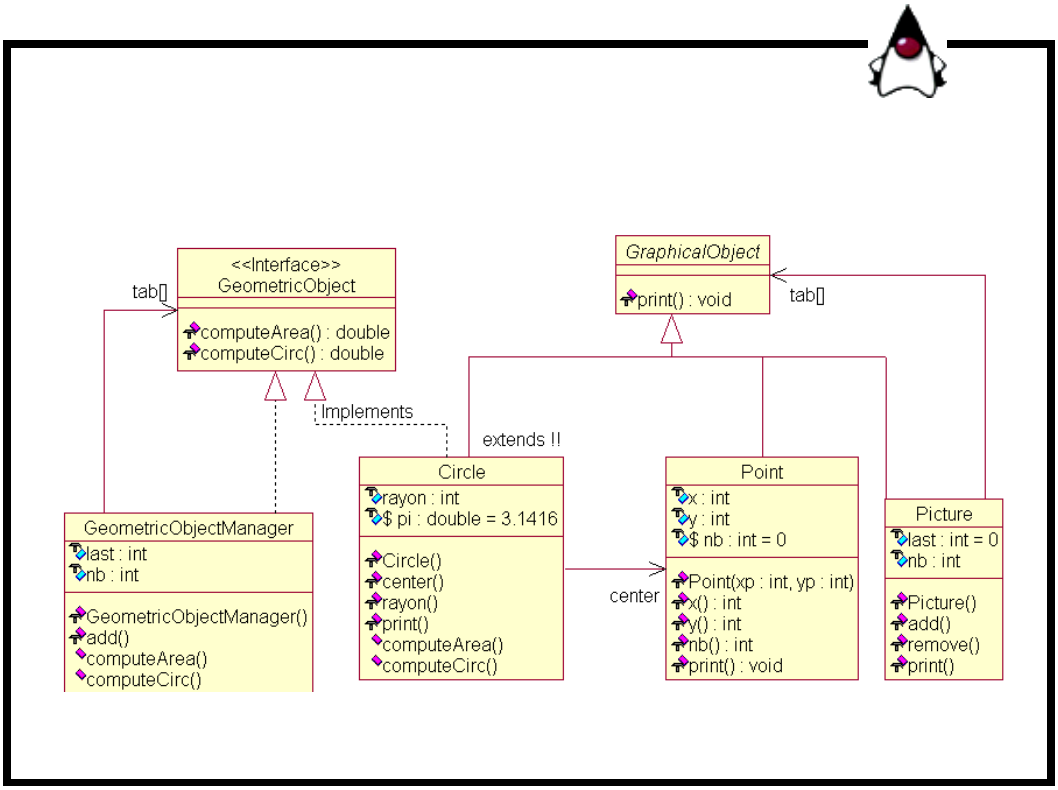
Slide 88

Une interface est une restriction d’une classe abstraite :

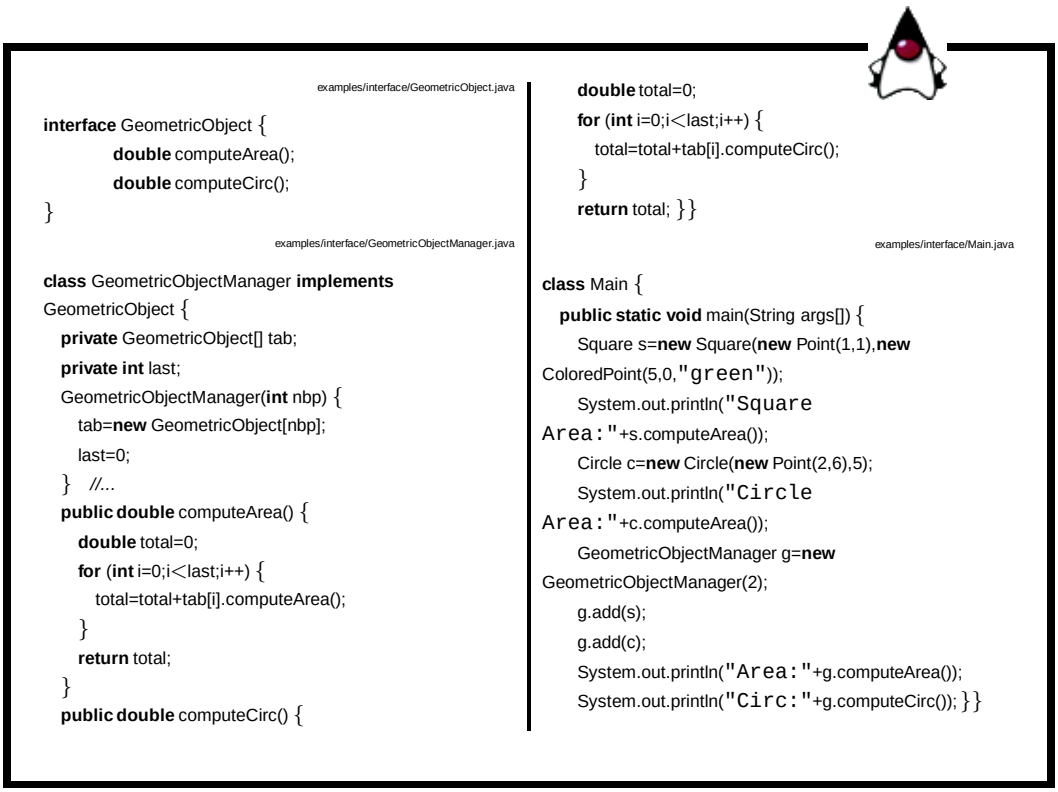
- Pas de constructeurs !
- Pas de variables membres !
- Pas de méthodes non abstraites !

Conséquences (pour ceux qui connaissent C++) :

- Pas de problèmes de “virtual base classes”
- Pas de problèmes de convention d’appel des constructeurs



Slide 89



examples/interface/GeometricObject.java

```
interface GeometricObject {
        double computeArea();
        double computeCirc();
}
```

examples/interface/GeometricObjectManager.java

```
class GeometricObjectManager implements
GeometricObject {
  private GeometricObject[] tab;
  private int last;
  GeometricObjectManager(int nbp) {
    tab=new GeometricObject[nbp];
    last=0;
  }  //...
  public double computeArea() {
    double total=0;
    for (int i=0;i<last;i++) {
      total=total+tab[i].computeArea();
    }
    return total;
  }
  public double computeCirc() {
    double total=0;
    for (int i=0;i<last;i++) {
      total=total+tab[i].computeCirc();
    }
    return total; }}
```

examples/interface/Main.java

```
class Main {
  public static void main(String args[]) {
    Square s=new Square(new Point(1,1),new
ColoredPoint(5,0,"green"));
    System.out.println("Square
Area:"+s.computeArea());
    Circle c=new Circle(new Point(2,6),5);
    System.out.println("Circle
Area:"+c.computeArea());
    GeometricObjectManager g=new
GeometricObjectManager(2);
    g.add(s);
    g.add(c);
    System.out.println("Area:"+g.computeArea());
    System.out.println("Circ:"+g.computeCirc()); }}
```

Slide 90

2.5.5 Interfaces vs Multiple Inheritance

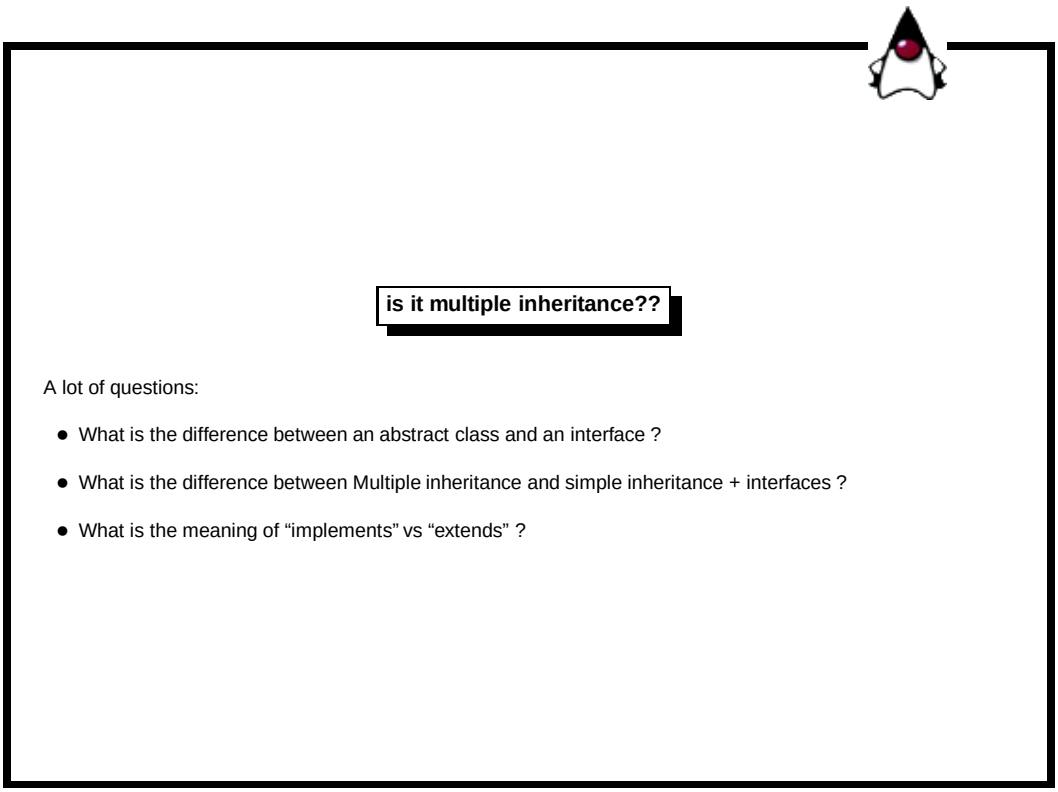
Interface inheritance

- An interface can inherit (yes ! extends !)from one or several other interfaces !
- Multiple inheritance for interfaces !

examples/interface/Multidraw.java

```
interface Draw_constants {
int draw_area_top = 10;
int draw_area_left = 10;
int draw_area_bottom = 100;
int draw_area_right = 200;
}
interface Draw_methods {
void drawme(int x, int y);
}
interface Drawable extends Draw_constants,Draw_Methods {
}
```

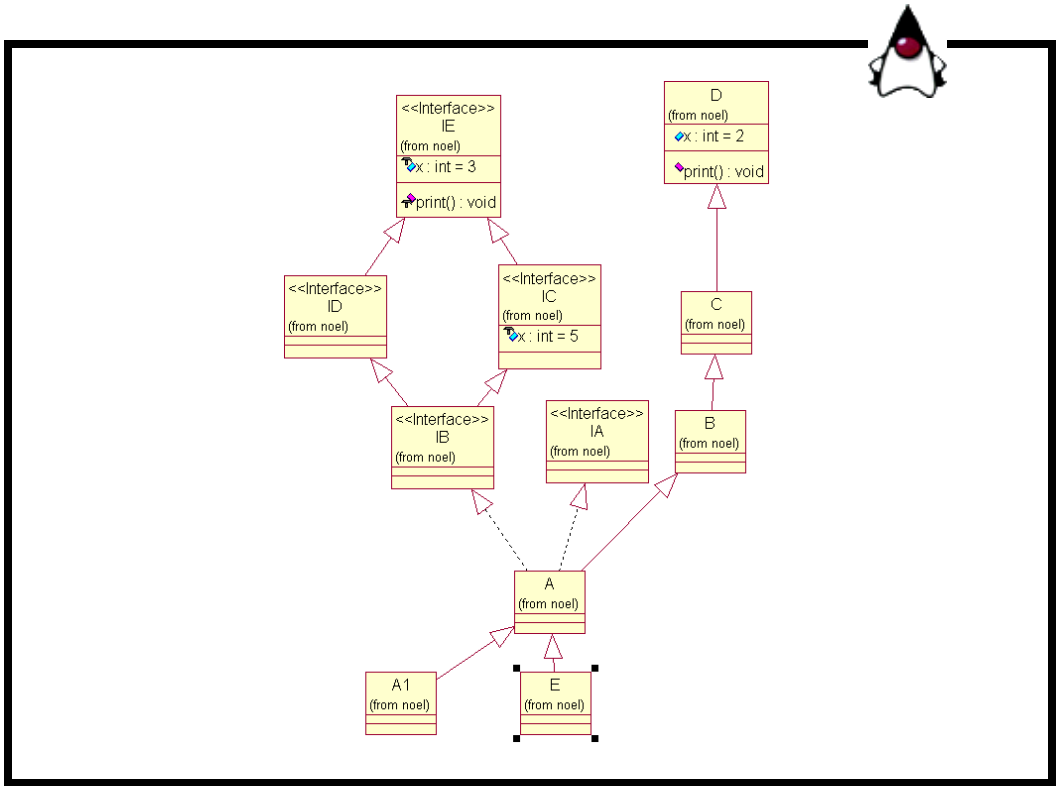
Slide 91



Slide 92

C'est encore un quizz !

- Entre une classe abstraite et une interface ? de manière opérationnelle, la différence est minime, tous les champs d'une interface sont public/static et les méthodes abstract/final. Ces restrictions ne s'applique pas aux classe abstraite. Je pense que les interfaces sont plus là pour caractériser une capacité spéciale d'une classe (Drawable, Serializable, remote ...) alors qu'une classe abtraite représente les caractéristiques intrinséque d'une famille de classes. La différence est donc plus conceptuelle qu'opérationnelle.
- (MOAAAA) Je pense que l'héritage multiple permet de représenter plus de choses. Par exemple, les hotels, les restaurants et les hotels-restaurants se modèlisent trés bien avec de l'héritage multiple et très mal si on doit avoir recours à des interfaces. Maintenant l'héritage multiple apporte pas mal de complexité avec lui (voir les pb de virtual base class en C++) qui ne peuvent exister avec ce systèmes d'interfaces + héritage simple (pas de constructeurs, pas de variables membres dans les interfaces).
- “Implements” vs “extends” ? Extends signifie “est une sorte de” alors qu'à mon avis “implements” signifie “est capable de répondre à”. L'interface est alors une sorte de spécification de protocole (entre moulte guillemets ...) dans le sens où il assure à l'utilisateur de l'interface que les implémentations pourront bien répondre aux méthodes déclarée dans l'interface.
- À quoi ça sert ? À pouvoir écrire du code juste en connaissant une interface et non l'implémentation. Du code générique en quelque sorte. Les exemples sont nombreux dans la lib java. Je pense que le plus beau, c'est les interfaces de sérialisation. Toutes les classes implémentant “Serializable” sont capable de devenir persistantes si elles sont atteignable à partir d'une instance que l'on veut persistante (voir doc “Serialization” !!! dans la doc du JDK).



Slide 93

Bien remarquer les relations "implements" et "extends".

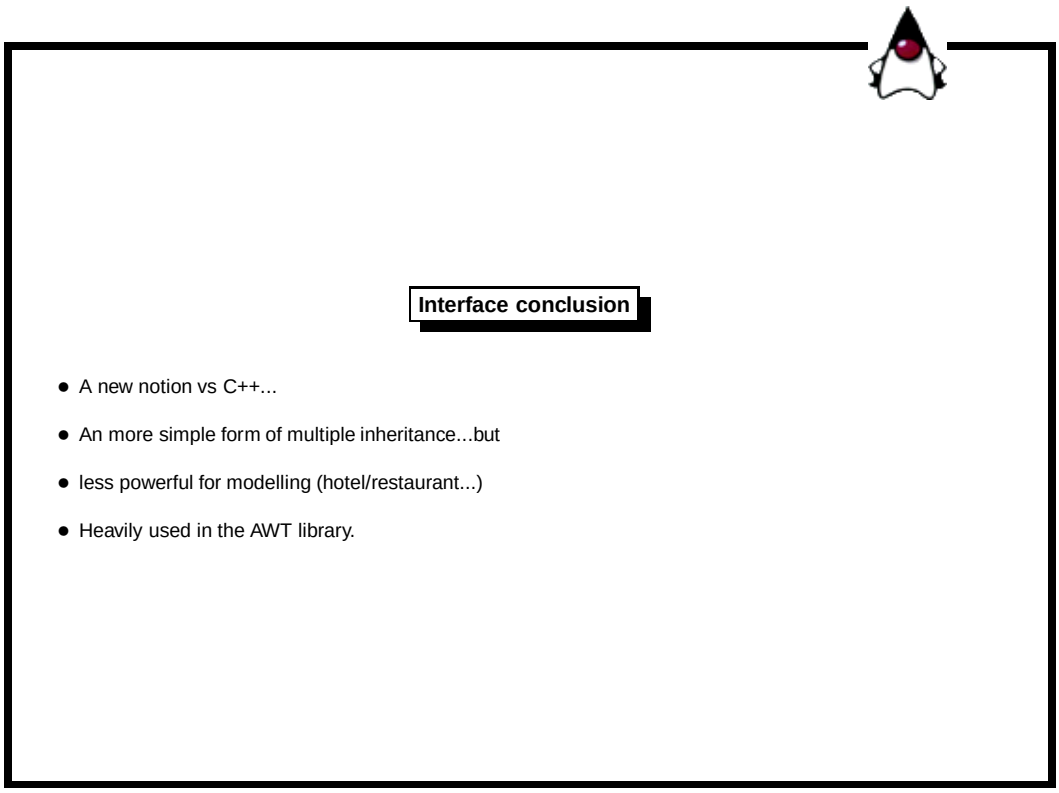
examples/interface/noel.java

```
class A1 extends A {}
class A extends B implements IB, IA {}
class B extends C {}
class C extends D {}
class D {
  public int x=2;
  public void print() { System.out.println(x);};}
class E extends A{}
interface IA {}
interface IB extends IC, ID{}
interface IC extends IE { int x=5;}
interface ID extends IE {}
interface IE{
  int x=3;
  void print();
}
class Main {
    public static void main(String args[]) {
      E e=new E(); e.print();
//      System.out.println(e.x); // error
      System.out.println(IE.x);
      System.out.println(((B)e).x); }}
```

Slide 94

En résumé :

- Héritage simple lors de la définition d'une hiérarchie de classes
- Héritage multiple lors de la définition d'une hiérarchie d'interfaces
- Une classe peut implémenter plus d'une interface : explicitement en mettant plusieurs noms d'interface dans une clause `implements`, ou par héritage des interfaces que déclarent implémenter ses super classes.



Slide 95

- Diminuer le couplage entre classes favorise la réutilisation : si on modifie l'une, les modifications dans les classes utilisatrices doivent être minimes.

2.6 Inheritance and Encapsulation

2.6.1 Inheritance and Private

Inheritance and Private

examples/heritage/Client4.java

```
class Person {
  private String name, address;
  public Person(String namep, String addressp) {
    name=namep; address=addressp;
  }
  public void print() { System.out.println(name+","+address); }
}

class Client extends Person {
  private int amount=0;
  public Client(String namep, String addressp,int amountp) {
    super(namep,addressp); amount=amountp; }
  public void print() { System.out.println(name+","+address+","+amount); }
}

class ClientDemo {
  public static void main(String[] args) {
    Client c=new Client("momo","yo",100); c.print();
  }
}
```

Slide 96

Les champs déclarés en private ne reste accessible que de la classe. i.e. ils ne sont pas visible pour les sous-classes.

```
Client4.java:19: Variable name in class Person not accessible from class Client.
    System.out.println(name+","+address+","+amount);
                       ^
Client4.java:19: Variable address in class Person not accessible from class Client.
    System.out.println(name+","+address+","+amount);
                               ^
2 errors
```

2.6.2 Encapsulation and Overriding

Ensuring the contract

- Overriding of a method ⇒
  - Method Access have not to be reduced... however
  - It can be enlarged : A private method can be overridden as a public method
- General rule : you have to ensure the contract you have passed with your superclass.

Slide 97

Ensuring the contract... Reduction

examples/heritage/Client5.java

```
class Person {
  private String name, address;
  public Person(String namep, String addressp) {
    name=namep; address=addressp; }
  public void print() { System.out.println(name+", "+address); }
}

class Client extends Person {
  private int amount=0;
  public Client(String namep, String addressp,int amountp) {
    super(namep,addressp); amount=amountp; }
  private void print() { System.out.println(name+", "+address+", "+amount); }
}

class ClientDemo {
  public static void main(String[] args) {
    Client c=new Client("momo","yo",100); c.print(); }
}
```

Slide 98

Ensuring the Contract

```
Client5.java:18: Methods can't be overridden to be more private.
 Method void print() is public in class Person.
  private void print() {
               ^
1 error
```

Slide 99

Ensuring the contract.. enlarging

examples/heritage/Client6.java

```
class Person {
  private String name, address;
  public Person(String namep, String addressp) {
    name=namep; address=addressp; }
  private void print() { System.out.println(name+","+address); }
}

class Client extends Person {
  private int amount=0;
  public Client(String namep, String addressp,int amountp) {
    super(namep,addressp); amount=amountp; }
  public void print() { System.out.println(amount); }
}

class Client6Demo {
  public static void main(String[] args) {
    Client c=new Client("momo","yo",100); c.print();
  }
}
```

Slide 100

Bien noter sur cet exemple le passage des variables membre de Person en protected et surtout private void print() redéfinit en public void print() dans Client. Bien sur, ce code compile.

2.6.3 Protected Modifier

`protected` Modifier

- Allow field or method access to the class and all subclasses
- Code that reference a protected field or method of class “C” have to be defined within the class C or subclasses of C.

Slide 101

Protected

examples/heritage/protected.java

```
class Person {
   private String name
   protected String address;
   public Person(String namep, String addressp) {
      name=namep; address=addressp; }
   private void print() { System.out.println(name+","+address); }
}
class Client extends Person {
   private int amount=0;
   public Client(String namep, String addressp,int amountp) {
      super(namep,addressp); amount=amountp; }
   public void print() {
      System.out.println(name);                          //error
      System.out.println(address);                       //ok protected
      system.out.println(amount);                        // ok it's mine
}}
class ProtectedDemo {
   public static void main(String[] args) {
      Client c=new Client("momo","yo",100); c.print(); } }
```

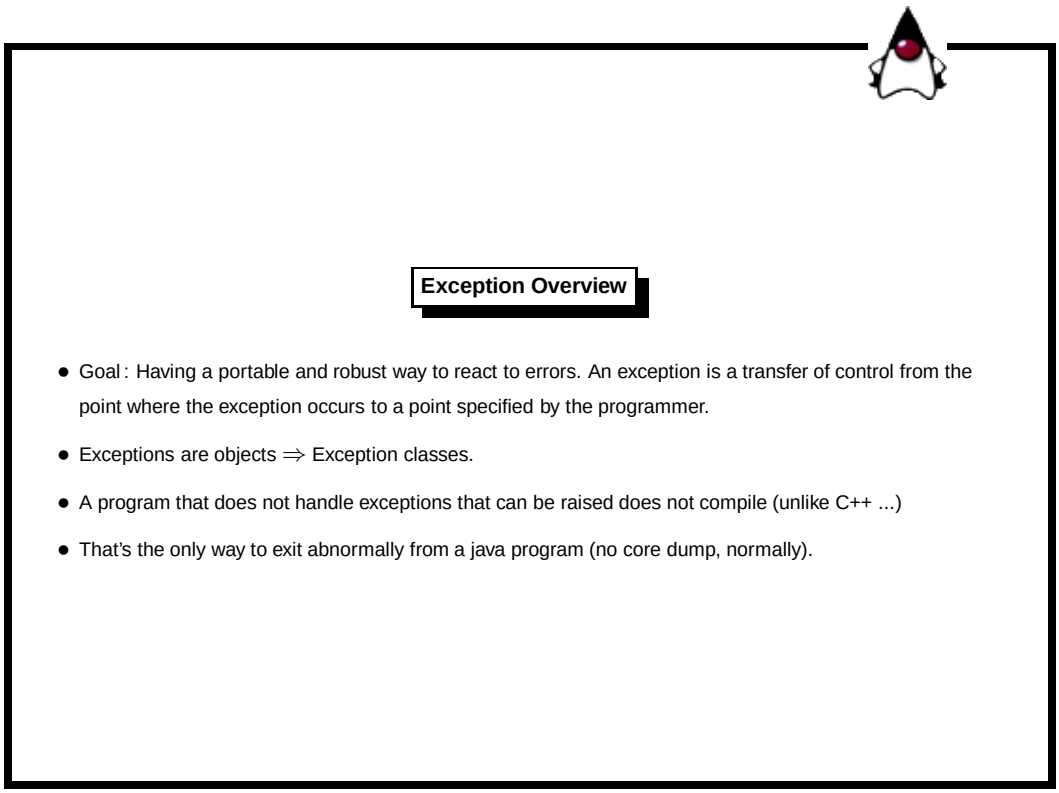
Slide 102

Language: Contents

- General Points
- Compilation and Execution
- Classes and Objects
- Inheritance
- Polymorphism
- **Exceptions**
- Conversions
- Packages

Slide 103

2.7 Exceptions



Slide 104

2.7.1 Anatomy and Life Cycle

Exceptions Objects

- Exceptions are objects !

examples/exception/myexcep.java

```
class MyException extends Exception {
  MyException(String why) {
    super(why);
  }
}
```

- The class “Exception” is the root of checked exceptions i.e. exceptions that are checked at compile-time.
- The exception class has a sister “Error” for unrecoverable errors (out of memory, no class found ...)

Slide 105

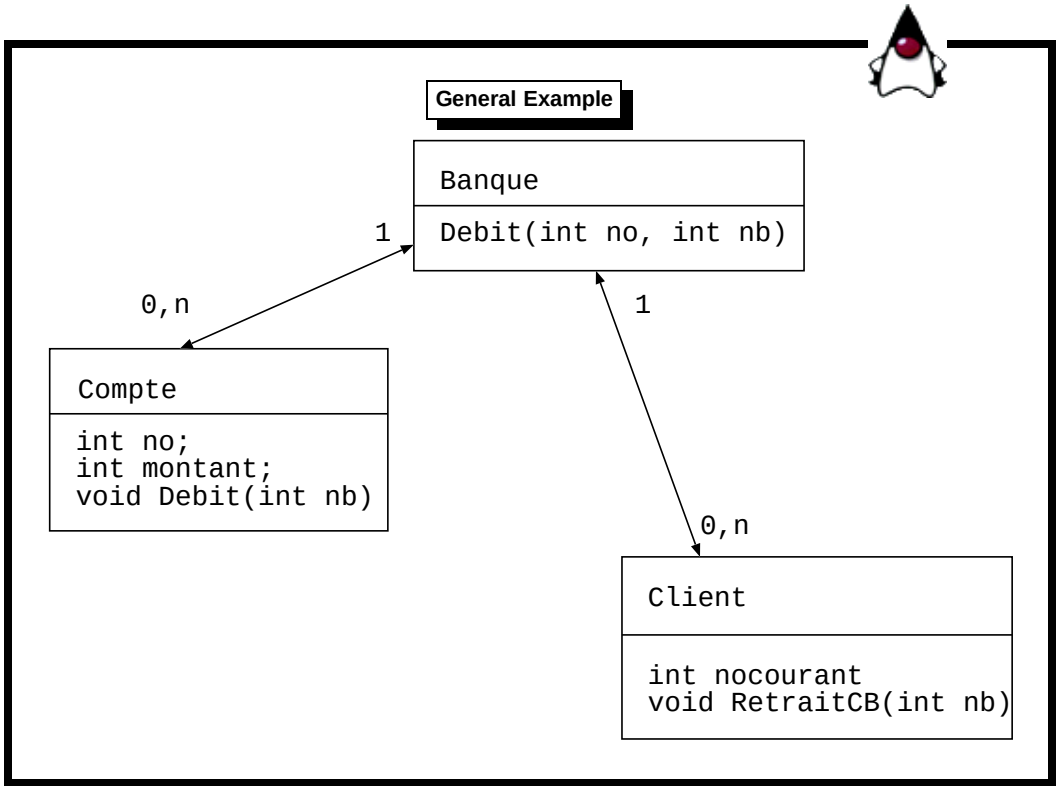
Exception Life Cycle

Throwed : A method throws an exception : it creates an exception object, stop the method execution and return the control to calling method

Propagated : When calling a method, an exception occurs, control come back with an exception object, I don’t manage this kind of exception, I stop my execution and return control to the calling method

Catched : A method return an exception : that was an expected scenario, I manage it and stop the execution propagation.

Slide 106

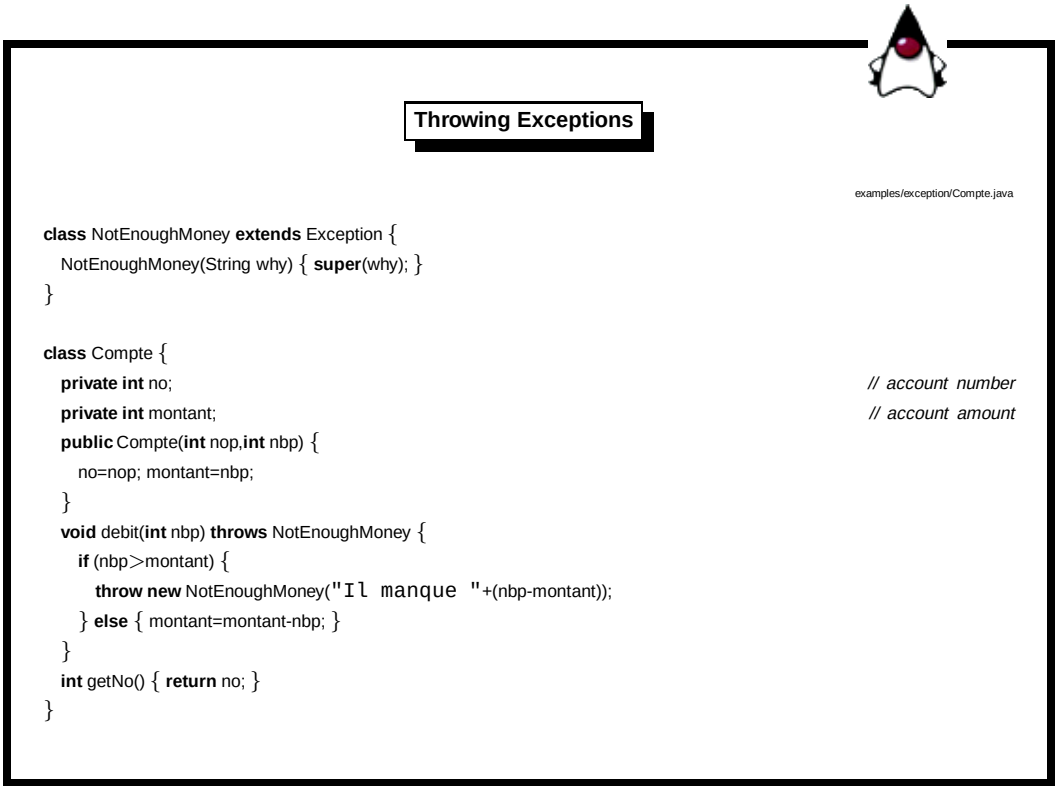


Slide 107

Bon l'exemple est le suivant :

- Le client fait un retrai CB
- appel de retraitCB → appel de Branque.debit() → appel de compte.debit()
- pas assez d'argent sur le compte
- emission d'une exception dans compte,
- propagation dans Banque,
- interception dans Client.

2.7.2 Throwing



Slide 108

Bien remarquer la syntaxe “throws” dans la signature de Debit, et l’appel “throw new NotEnoughMoney” dans le corps de Debit.

- La clause throws déclare avec quelles exceptions on peut sortir de la méthode. Si une exception se produit et qu’elle n’est pas déclarée dans la clause throws, alors une “ERROR” est levée ! La clause throws peut déclarer une liste d’exception.
- Bien noter que “throw” lance l’objet exception dans la remontée de la chaine d’appel. Il faut bien entendu créer cet objet exception de manière tout à fait classique avant de lancer la propagation de l’exception.

La clause `throw expression`, [JG96][287], où expression doit se résoudre à une référence d’instance d’objet affectable à `Throwable`, lève l’exception indiquée, prend le contrôle (*abrupt completion* de ce qui s’exécutait), parcourt la pile et :

2.7.3 Propagation

Exception Propagation ...

examples/exception/Banque.java

```
class Banque {
  private Compte[] cpts=new Compte[20];
  private static int nbcompte=0;

  void register(Compte cp) {
    cpts[cp.getNo()]=cp;
  }
  void Debit(int nop, int amountp) throws NotEnoughMoney {
    cpts[nop].debit(amountp);
  }
  int getLast() {
    nbcompte++;
    return nbcompte;
  }
}
```

Slide 109

- L'opération Banque.Debit appelle Compte.Debit. Or Compte.Debit peut se terminer avec une exception NotEnoughMonney. Si c'est le cas, alors Banque.Debit peut aussi se terminer avec NotEnoughMoney et la remontée de la chaine d'appel des méthodes continue.
- Bien noter que si NotEnoughMoney ne figure pas dans la clause "throws" de Banque.Debit, alors il y a une erreur de compilation !!

2.7.4 Catching

Catching Exception

examples/exception/Client.java

```
class Client {
  private int no;
  private Banque banque;
  Client(Banque bp,int amountp) {
    banque=bp;
    no=banque.getLast();
    banque.register(new Compte(no,amountp));
  }
  void RetraitCB(int amountp) {
    try {
      banque.Debit(no,amountp);
    }
    catch (NotEnoughMoney e) { e.printStackTrace(); }
    catch (Exception e) { e.printStackTrace(); }
    finally { System.out.println("Au revoir !"); }
  }
}
```

Slide 110

Exception : Operational Model

- `try`, No exception ⇒ Execution of the try/catch and the `finally` block.
- `try`, An exception `e` of dynamic type `E` is raised :
  - `e` is an instance of a class defined in a `catch` clause : The corresponding block is executed and then the “finally block”.
- **Beware :** An exception throwed in a catch block can be hidden by another exception throwed in the finally block.

Slide 111

[JG96][291]. Le problème est de savoir pour une clause *try/catch/finally* quelles peuvent être les complétions possibles et pourquoi... En bref, c'est plutôt "logique". Le `finally` gagne en tout état de cause.

2.7.5 Execution

Execution

examples/exception/main.java

```
class main {
  public static void main(String argv[]) {
    Banque banque=new Banque();
    Client c=new Client(banque,100);
    c.RetraitCB(200);
    for (int i=0;i<40;i++) {
      new Client(banque,100);
    }
  }
}
```

```
NotEnoughMoney: Il manque 100
        at Compte.debit(banque.java:16)
        at Banque.Debit(banque.java:34)
        at Client.RetraitCB(banque.java:52)
        at main.main(banque.java:68)
Au revoir !
java.lang.ArrayIndexOutOfBoundsException: 20
        at Banque.register(banque.java:31)
        at Client.<init>(banque.java:48)
        at main.main(banque.java:70)
```

Slide 112

Bien remarquer la différence entre les deux exceptions !

- La première est notre exception ! Tout s'est passé comme nous l'avions prévu.
- la seconde est une "run-time" exception. Le compilateur ne nous a pas prévenu de cette eventualité et ne nous pas forcé à en tenir compte.

C'est la différence entre les "checked" exception, généralement celles que nous déclarons et les "run-time" exception. Le compilateur n'exige pas que les run-time exception soit propagée ou interceptée (voir java.lang.RuntimeException dans la doc du JDK).

2.7.6 Obfuscated Examples

Exception : Obfuscated Examples

examples/ExceptionFumet.java

```
class Exception1 extends Exception {
    Exception1() { super(); }
    Exception1( String s) { super(s); }}
class Exception2 extends Exception {                                    //...
}
class ExceptionFumet {
        public static void main(String[] args) throws Exception2 {
            try {
// Plein de code d'où peuvent surgir plein
// de checked exceptions
                throw new Exception1("Hop !!");
            } catch (Exception1 e) {                          // PARCOUR dans l'ORDRE
                throw new Exception2();                        //OK:specifiee dans profil
            } catch (Exception reste) {                                         //...
            } finally {                                                  // OPTIONEL
                // qui ne doit pas lever de checked exceptions
                // autre que Exception2
            }}}
```

Slide 113

Exception : Obfuscated Examples

examples/TestException.java

```
class Exception1 extends Exception {}
class Exception2 extends Exception {Exception2(String s){super(s);}}
class TestException {
    public static void main(String[] args) throws Exception2 {
        try {
            Class c=Class.forName("Exception2");                 // ClassNotFound possible
            throw new Exception1();
        } catch (Exception1 e) { System.out.println("Exception 1 catchée");
            throw new Exception("UN");                                // masquée par finally
        } catch (Exception reste) {
            System.out.println("Autre exception :  " + reste);             //filtre ClassNotFound
        } finally { System.out.println("Cio!");
        throw new Exception2("DEUX");                              // remontée niveau thread
        }}}
```

Slide 114

Exception : Obfuscated Examples

```
Exception 1 catchée
Cio!
Exception2: DEUX
at TestException.main(TestException.java:13)
```

- two first lines on stdout : `System.out.println`
- two last lines on sdtderr : Default, JVM

Slide 115

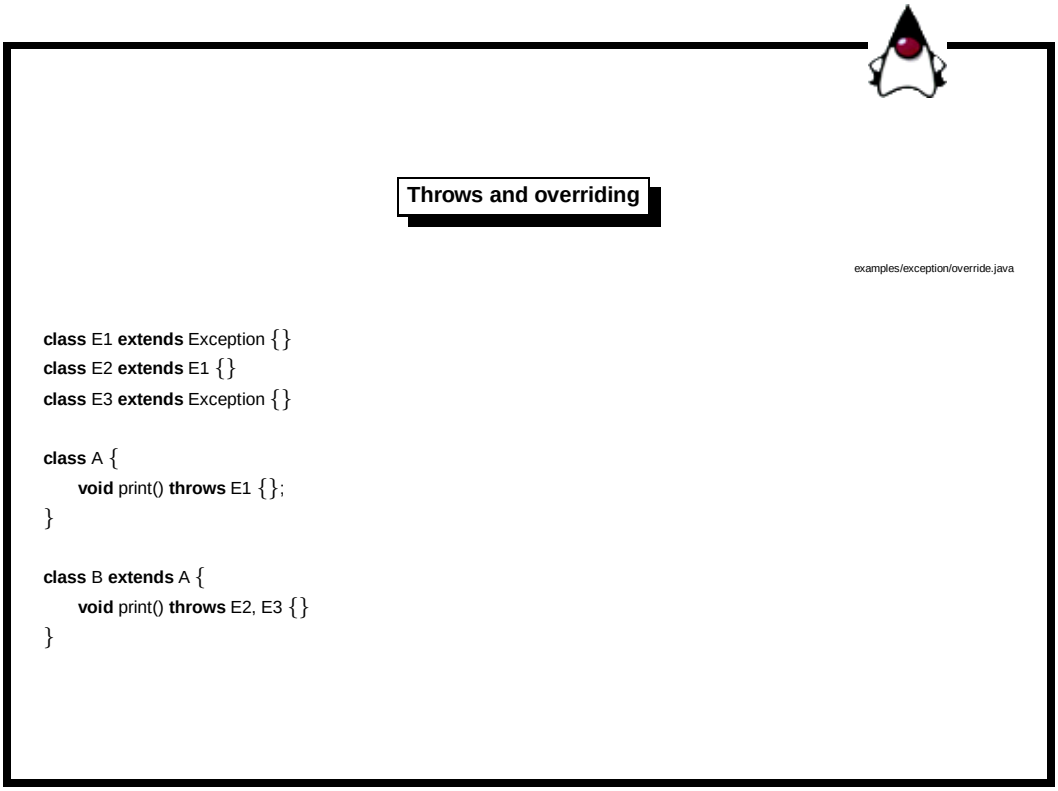
`throws` and Inheritance

- **Problem :** Throws and overriding...
- **Idée :** Ensuring the contract ...
- Client code that use the overriden method have to work in the same way for the overriding method
- The throws clause of the overriding method must contain only classes subclass of the overriden method throws clause.

Slide 116

- La restriction n'implique pas que la méthode de la sous classe spécifie moins d'exceptions. Au contraire elles peuvent être plus nombreuses car plus spécifiques. La méthode précise ainsi "exactement" les cas critiques. La restriction assure qu'un code utilisateur de la super classe pourra globalement les récupérer.

Met-on un exemple sur les pbs de "compatibilité" clause throws en cas de surcharge ou overriding ? Un pourrait simplement mettre un schema : une classe at une sous classe avec une methode m override qui specifie des exceptions situee dans 2 arbres d'heritage separes.



Slide 117

Result :

```
override.java:11: Invalid exception class E3 in throws clause. The exception must be a subclass of
    void print() throws E2, E3 {}
         ^
1 error
```

Exception : Conclusion

- splitting normal code and error management code
- Reduction of the code size vs `==-1` UNIX style error management...
- it has a price :
  - All checked exception have to be catched !
  - bad using of the mechanism....

Slide 118

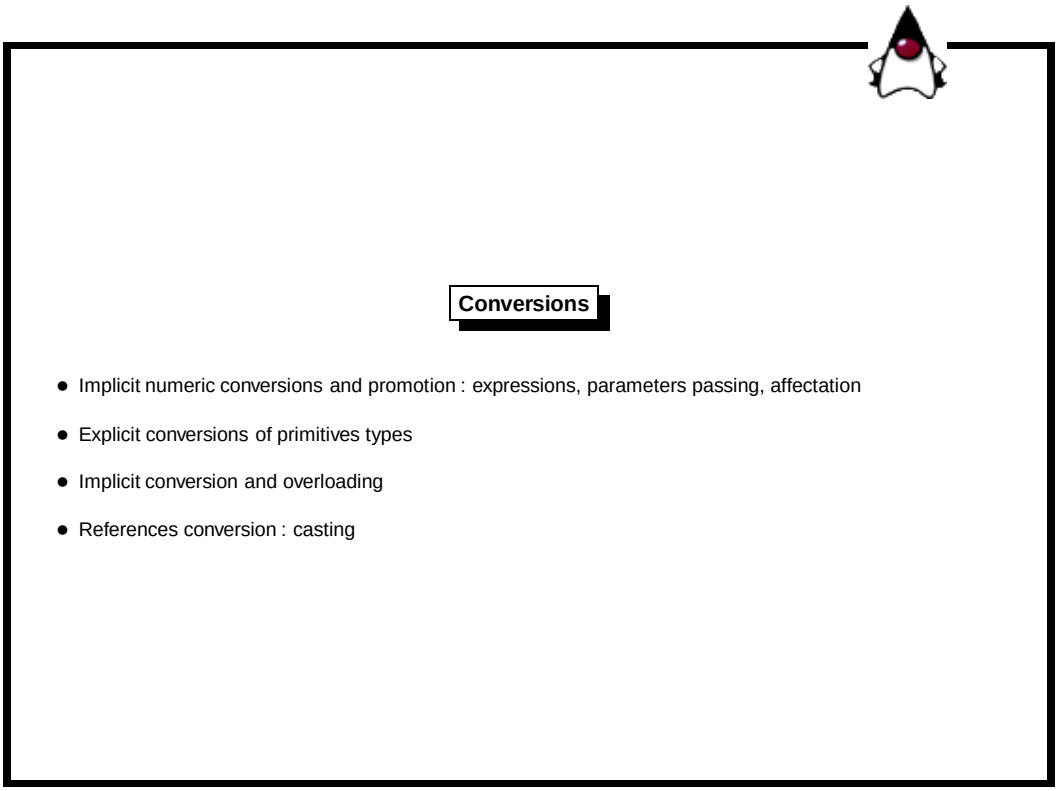
Language: Contents

- General Points
- Compilation and Execution
- Classes and Objects
- Inheritance
- Polymorphism
- Exceptions
- **Conversions**
- Packages

Slide 119

2.8 Conversions

Le lecteur est invité à consulter [JG96][chapitre 5] si il désire connaitre la description de toutes les possibilités de conversions de Java. On va donner ici seulement les cas importants et les principes généraux régissant les conversions.



Slide 120

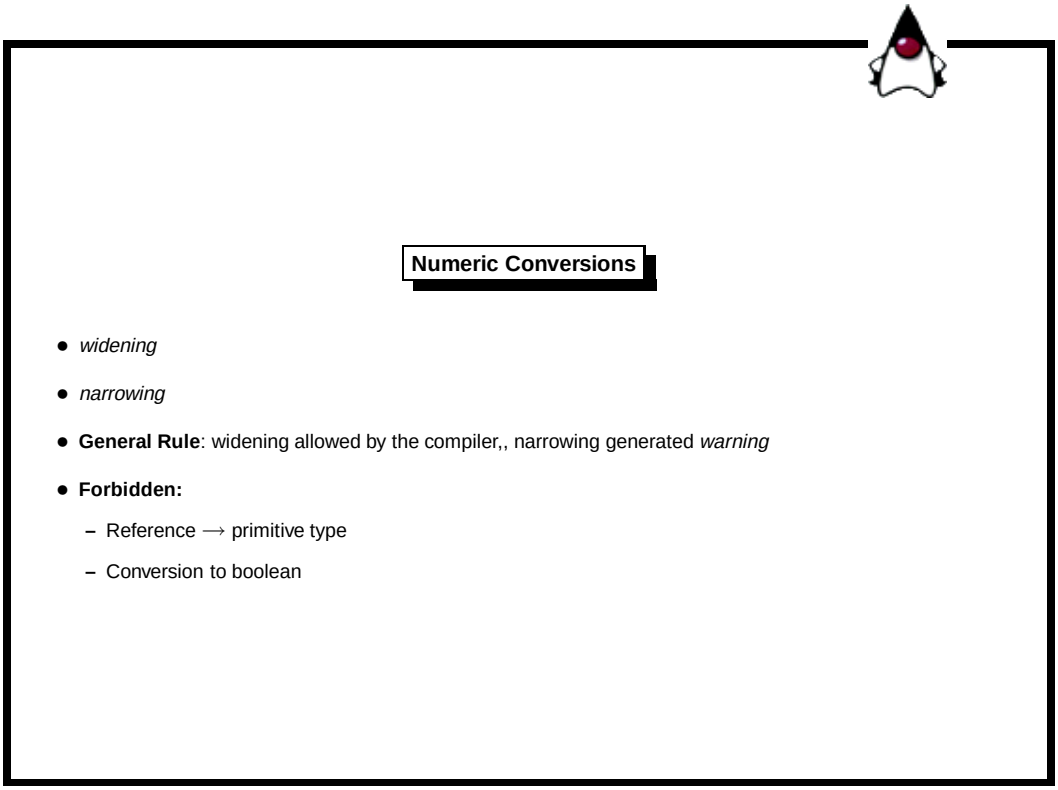
2.8.1 Primitives types and conversion/promotions

Implicit Conversion: When ?

- Affectation (implicit conversion)
- Method call
- Expression evaluations
- `String` conversion : Special meaning of “+” for string ⟶ System.out.println(“toto”+1), method toString();
- Cast (like C/C++)

Slide 121

Mettre un exemple pour le coup du toString() ...



Slide 122

- L'interdiction des casts "sauvages" entre types primitifs et références est un point important pour assurer une sureté de fonctionnement du code, et une certaine sécurité contre les virus.

Numeric Conversions

- Java have a table of prices to convert a primitive type to another.
- < 10 : *Widening*
- otherwise *narrowing*

Slide 123

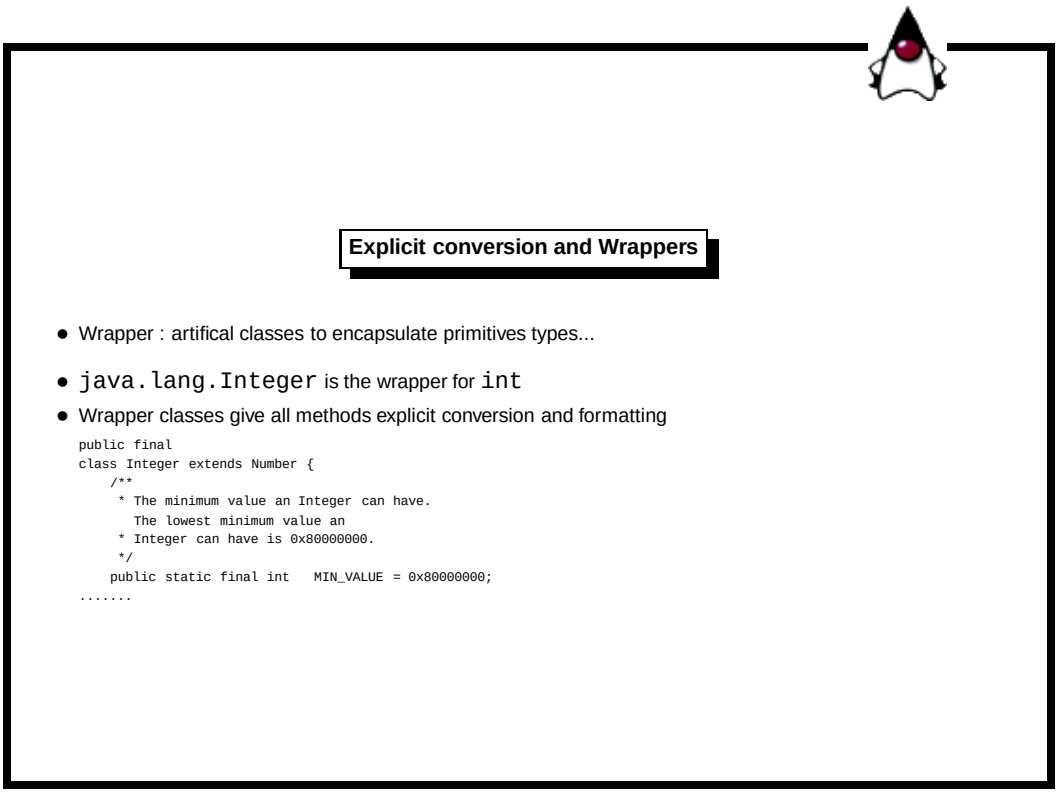
Numeric Conversions

from: / to:	byte	short	char	int	long	float	double
byte	0	1	2	3	4	6	7
short	10	0	10	1	2	4	5
char	11	10	0	1	2	4	5
int	12	11	11	0	1	4	5
long	12	11	11	10	0	6	5
float	15	14	13	12	11	0	1
double	16	15	14	13	12	10	0

Slide 124

- Le tableau précédent vient de : [Anu96].
- Plus une valeur du tableau est petite moins la conversion est coûteuse, d'où la diagonale à 0 ; la première ligne croisante ; ou la dernière ligne très chère et décroissante : `double` est un des type les plus grand et complexe à manipuler.
- Au dessus d'un coût de 10 (inclus), il y a perte de précision. On ne perd rien pour un `byte`, mais on perd toujours quelque chose pour un `double`.
- Les modes opératoires pour les arrondis, sont indiqués dans la spécification du langage.

2.8.2 Explicit Conversions and Wrappers



Slide 125

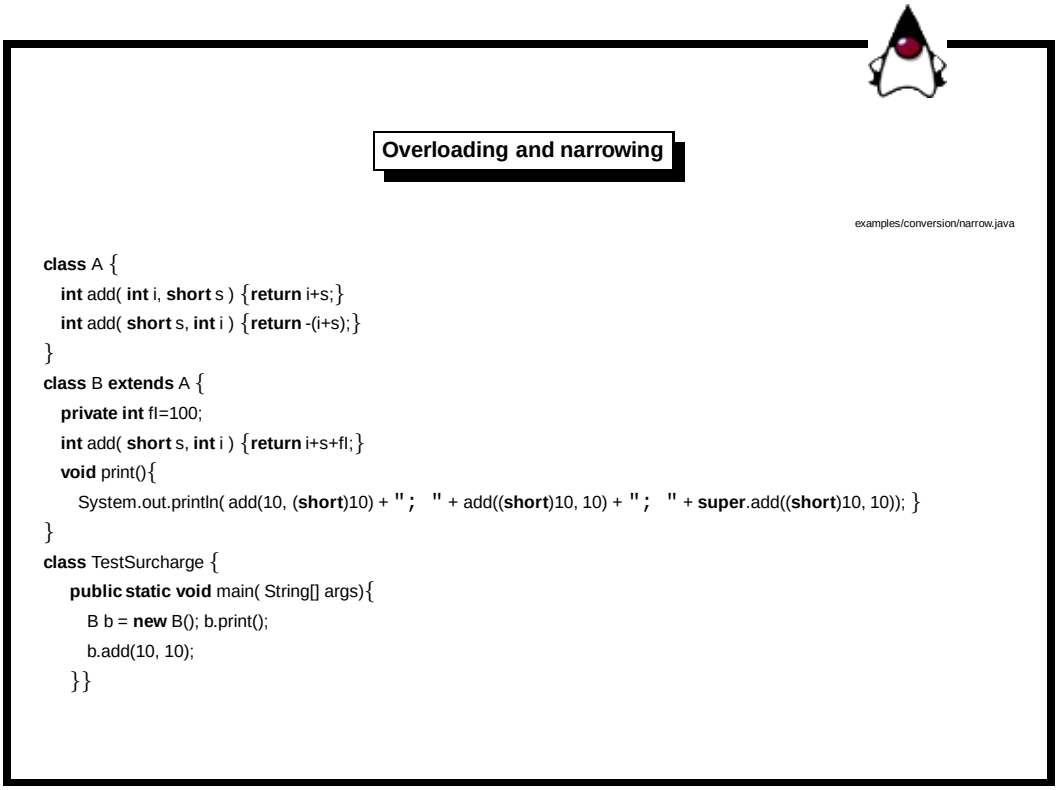
Scanf ?

examples/Scanf.java

```
import java.io.*;
import java.util.*;
class Scanf {
 public static void main( String[] args ) {
 try {
  BufferedReader StdIn=new BufferedReader(new InputStreamReader(System.in));
  String line=StdIn.readLine();
  StringTokenizer tokens= new StringTokenizer(line);
 while (tokens.hasMoreTokens()){
  String s=tokens.nextToken();
  int i =Integer.parseInt(s);
  System.out.println("On lit :"+ i);
  }
 }catch(IOException e){ }
  catch(NumberFormatException e){ System.err.println("On veut un entier"); }
 }
}
```

Slide 126

2.8.3 Implicit Conversion and Overloading



Slide 127

Pas évident : n.add(10,10) génère une erreur

```
No method matching add(int,int) found in class B.
```

En effet, aucun narrowing n'est permis pour résoudre le polymorphisme paramétrique.

2.8.4 Conversion of references and inheritance

Widening of references

- Sub classes ⟶ super class
- Class S to interface I if `S implements I`
- Sub Interface to super interface

Slide 128

Examples

examples/conversion/conversion.java

```
class Person {
  protected String name, address;
  public Person(String namep, String addressp) {
    name=namep;
    address=addressp;
  }
  private void print() {
    System.out.println(name+","+address);
  }
}

class Client extends Person {
  private int amount=0;
  public Client(String namep, String addressp,int amountp)
{
    super(namep,addressp);
    amount=amountp;
  }
  public void print() {
    System.out.println(name+","+address+","+amount);
  }
}

class Client6Demo {
  public static void main(String[] args) {
    Client c=new Client("Jacques","celere",100);
    Person p= new Person("jean","braye");
    Person p1=c;
    p1=(Person)p1;                                  // 1
    p1=(Client)p1;                                  // 2
    Client c1=p;                                    // 3
    Client c1=(Client)p;                            // 4
  }
}
```

Slide 129

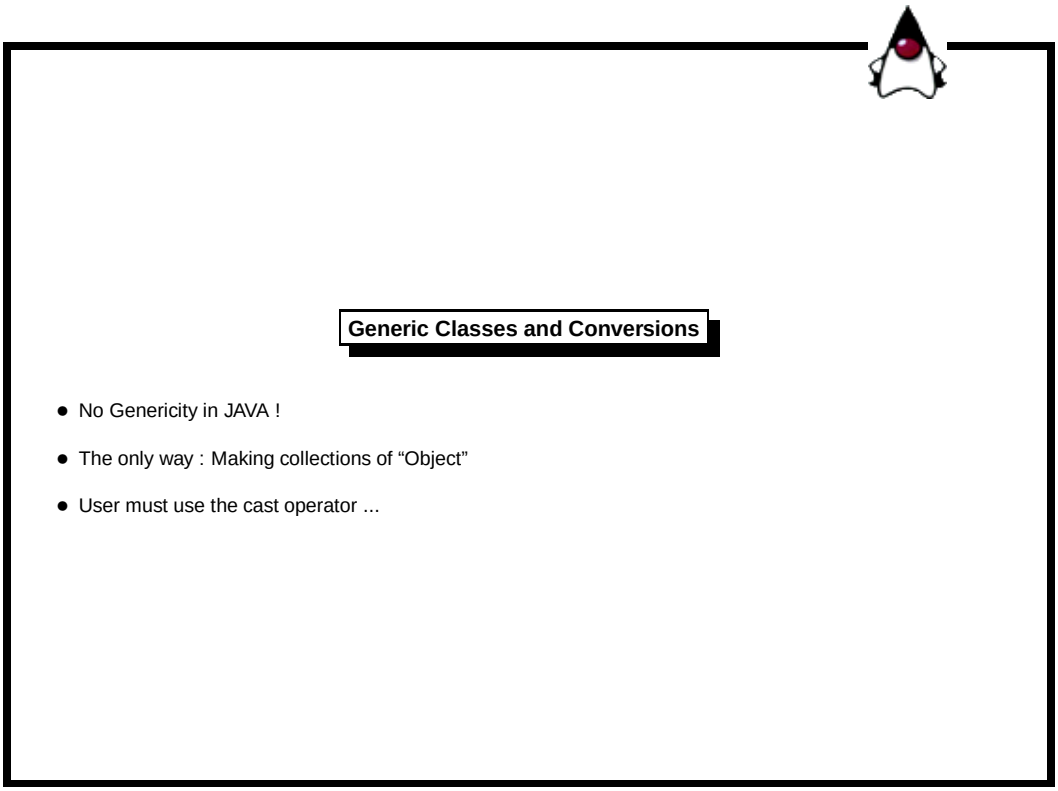
1. p1 est une référence sur une Person. Autorisé
2. l'objet pointé est bien un Client. Autorisé. Au passage bien remarquer qu'une fois qu'un objet est créé comme étant d'une certaine classe, il reste ad vitam eternam de cette classe !! (type safe)
3. error Compil !!

```
conversion.java:28: Incompatible type for declaration.
Explicit cast needed to convert Person to Client.
```

4. Erreur run-time:

```
java.lang.ClassCastException: Person
        at Client6Demo.main(conversion.java:33)
```

2.8.5 Generic classes and Conversions



Slide 130

examples/polymorphisme/stack.java

```
class Point {
  private int x, y;
  public Point( int x, int y ){ this.x=x; this.y=y;}
  public void move( int xp, int yp ){x+=xp; y+=yp;}
  public void print(){System.out.println("("+x+","+y+")");}}
class stack {
  private Object[] tab;
  private int top;
  public stack(int size) { tab=new Object[size]; top=0;}
  public void push(Object elt) { tab[top]=elt; top++; }
  public Object pop() { top--; return tab[top]; }}
class Main {
  public static void main(String args[]) {
    stack s=new stack(5);
    s.push(new Point(1,2));
    Object elt=s.pop();
    elt.print();                                                    // error
    Class classname=elt.getClass();
    System.out.println(classname);
    ((classname.toString())elt).print();                            //error
    ((Point)elt).print();
}}
```

Slide 131

Language: Contents

- General Points
- Compilation and Execution
- Classes and Objects
- Inheritance
- Polymorphism
- Exceptions
- Conversions
- **Packages**

Slide 132

2.9 Packages

Package: Overview

Basically, a package is group of class !

- The package provides naming spaces : a class “A” defined within a package “P” is called “P.A” et non “A”.
- It allows to hide some classes : A class can be declared public or not.
- It defines a new modifier for fields and methods to give a greater access between classes of the same package.

Slide 133

Package: First Example

examples/packages/Main.java

```
package MyPackage;
public class Main {
    public static void main(String args[]) {
      System.out.println("hello world");
    }
}
```

- if I work in `~molli`...
- This class must be defined inside a file `~molli/MyPackage/Main.java`
- The CLASSPATH must be set to : `.:~molli/:/usr/jdk/lib/classes.zip`
- `java MyPackage.Main` start execution... looking for `MyPackage/Main.class`
- `./MyPackage/Main.class` does not exist ...
- `molli/MyPackage/Main.class` exists → loading of Main.class and execution of main()...

Slide 134

2.9.1 Package and Naming

Encapsulation : `package` : naming

A package provides a naming space, like a UNIX directory.

- A class “A” defined in a class “Q” is not named “A” but “Q.A” ! the class “A” does not exists !
- A package can have sub-packages, in the same way that a directory can have sub-directories.
- Avoids name conflict (Symbol multiply defined ;-D)
- A package naming convention can use URL like : `COM.Apple.quicktime.v3`

Slide 135

L’exemple bateau vient du C++ ! nombre de librairies déclarent une classe String pour plein de mauvaises raisons. Si vous prenez deux librairies de ce type vous ne pourrez compiler sans avoir le “String : multiply defined”. Il faut absolument disposer du source des librairies pour pouvoir solutionner ce problème. Évidemment, le problème survit, puisqu’à chaque nouvelle version de la librairie il faudra propager vos changements du au conflit de nom. Bien sur, les package évite ce problème, à condition de générer des noms de package unique.

Import

examples/packages/vector.java

```
import java.util.Vector;

class Point {
  private int x,y;
  Point(int x, int y) {
    this.x=x;this.y=y;
  }
}

class Main {
  public static void main(String args[]) {
    java.util.Vector v2=new java.util.Vector();                    //ok without import
    v2.addElement(new Point(3,4));

    Vector v1=new Vector();                                        // ok with import
    v1.addElement(new Point(1,2));

  }
}
```

Slide 136

Naming : referencing classes defined in packages

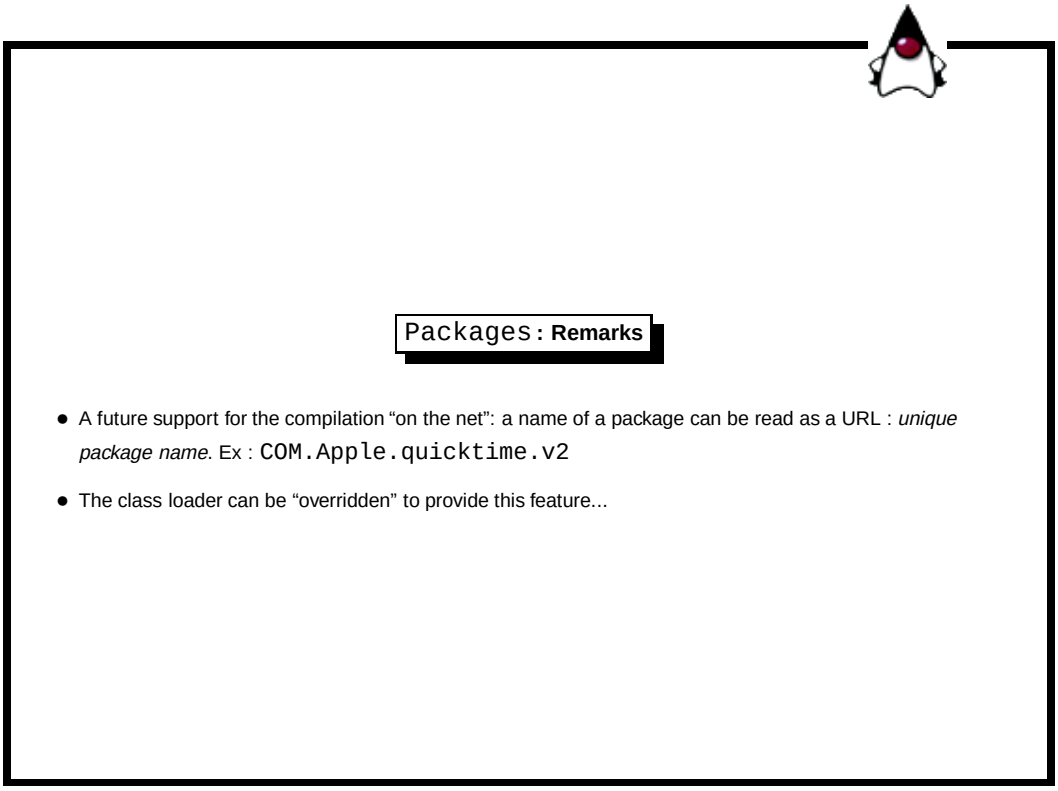
To reference a class defined in a package !

- Reference public classes with its entire name : COM.Apple.quicktime.v3.Main
- `import Package-name.Class-name` : Allows to use “Class-name” in the code without using “Package-name.Class-name”. It’s a writing convenience .
- `import Package-name.*` allows to use all classes defined as public in Package-name without specifying the entire name in the current java file.
- Beware ! No similarities with #include !
- if java or javac looks for a class A.B.C.D.ECLASSE, it has to find a file ECLASSE.class in a directory A/B/C/D. The CLASSPATH has to contain a path to the “A” directory.

Slide 137

- `import` correspond un peu au `WITH` ADA.
- Plusieurs clauses `import` peuvent être utilisées en début de fichier : le compilateur doit s'assurer alors qu'il n'y aura pas d'ambiguité entre les noms de classes directement utilisables

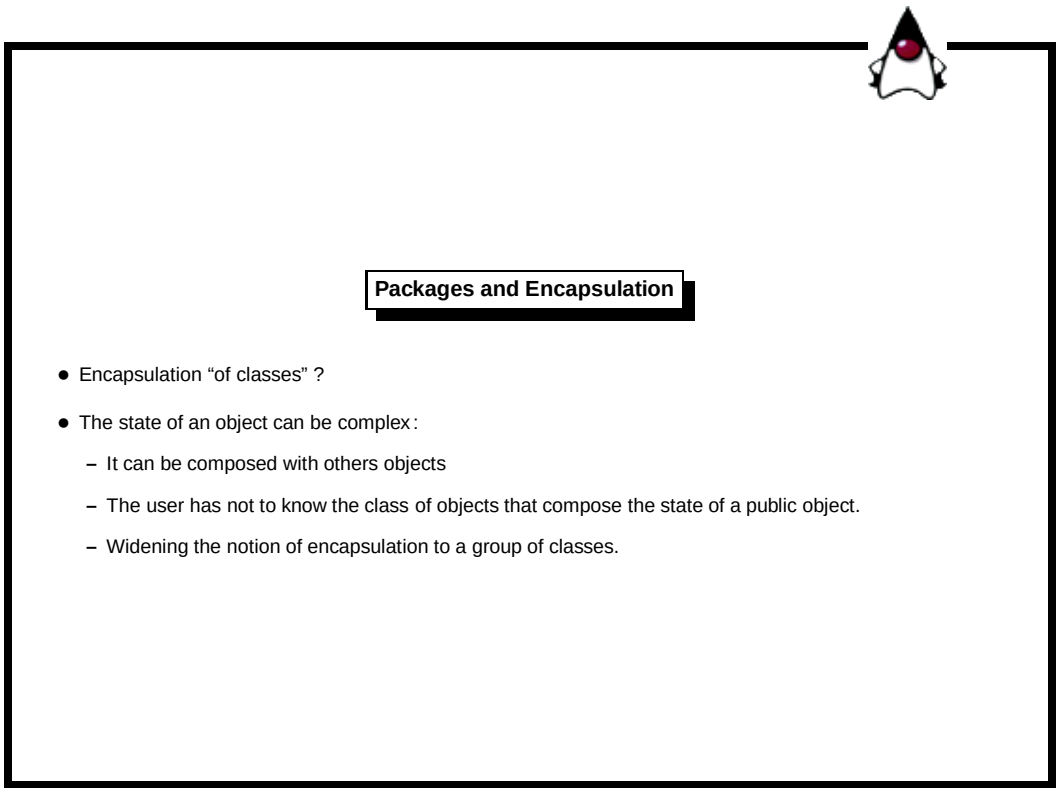
Si vous avez des erreurs du type "class not found" à la compilation ou à l'exécution pensez bien à cet algorithme et regardez votre CLASSPATH !!!



Slide 138

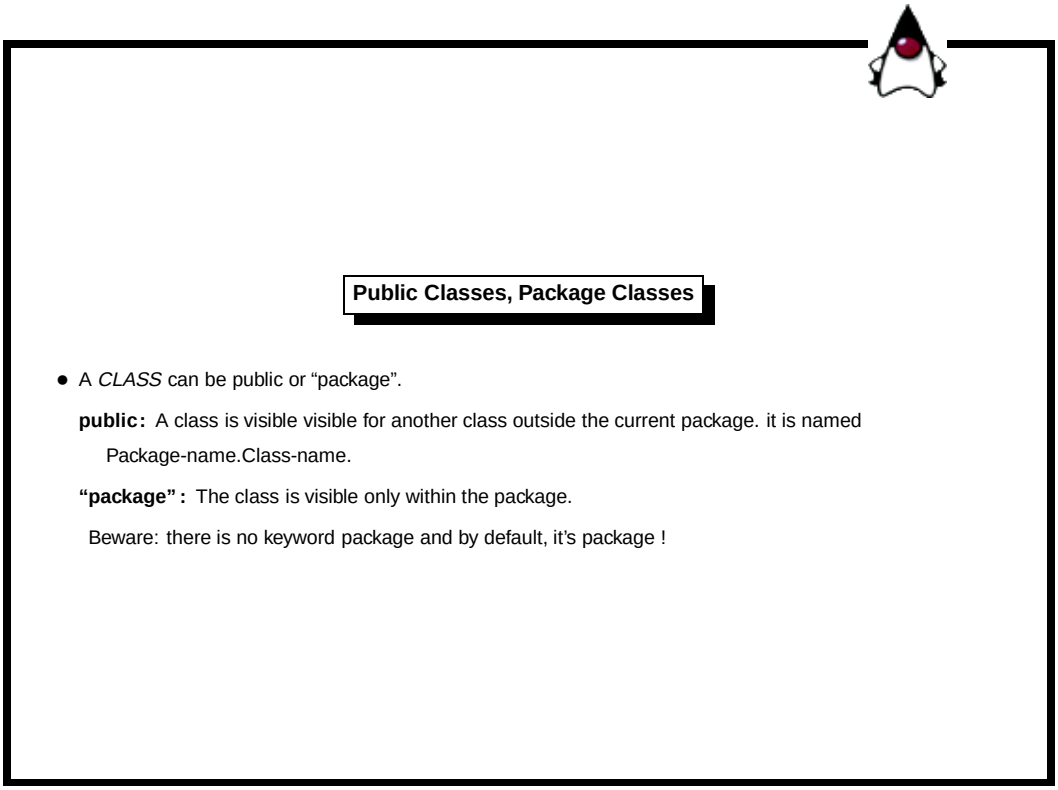
- Pour les règles et usages sur les noms de packages se reporter à [JG96][pages 113-126]. Il faut surtout noter qu'un nom de package doit commencer par une **minuscule**.
- En résumé une *compilation unit* que l'on fournit au compilateur javac est formée un fichier `.java` contenant le source d'un certain nombre de classes d'un même package (une seule clause `package`, en en utilisant d'autres (zéro ou plusieurs clause `import`).
  - Des imports "automatiques" sont faits pour les packages prédéfinis dans java (⇔ `import java.*`).
  - En cas d'abscence de clause `package`, la *compilation unit* se rapporte au "package par défaut", de nom vide.
- Il faut donc noter que la notion de package n'est pas uniquement un mécanisme d'encapsulation, c'est aussi une facilité de base pour organiser un tant soit peu le code.

2.9.2 Public classes, Package classes



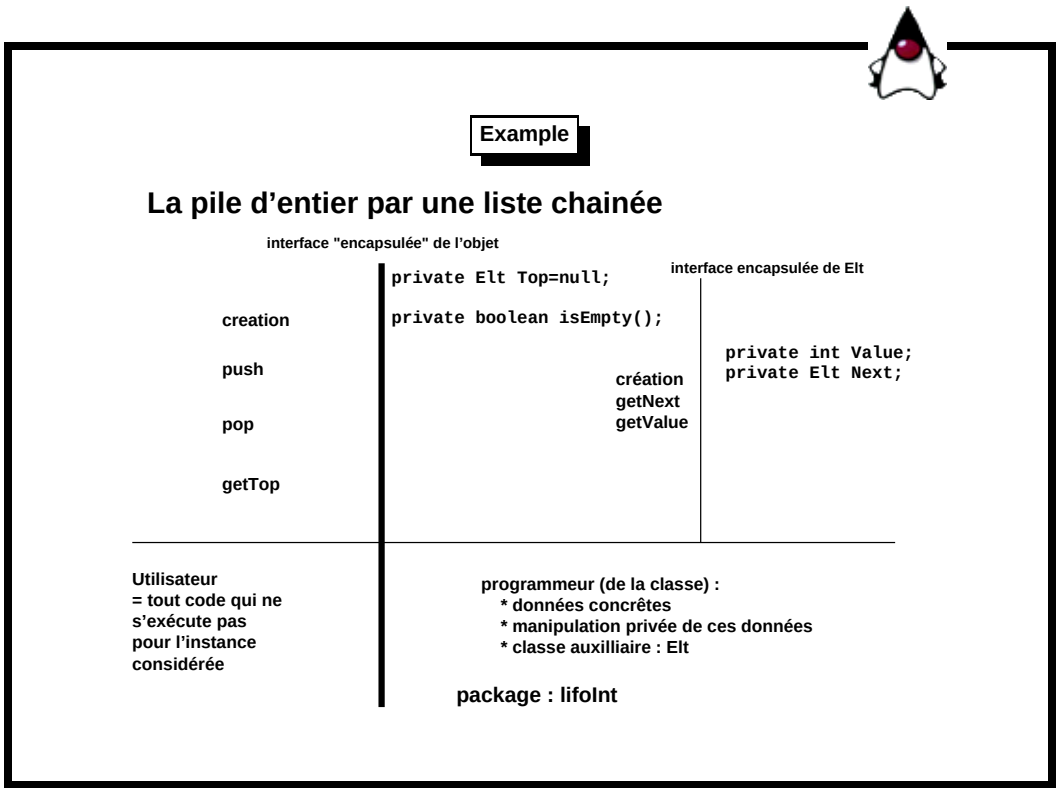
Slide 139

Attention, cette pseudo encapsulation de “classe” reste purement syntaxique. Les packages ne sont pas des objets.



Slide 140

Finallement, le mécanisme correspondant en c++, c'est les friends. Mais en Java ça me parait plus simple même si c'est pas aussi souple.



Slide 141

- Bon et bien voilà c'est pas compliqué ! On implante une pile à l'aide d'une liste chainée dont on accède qu'à la tête.
- L'état d'une instance de `LifoInt` utilise des instances de `Elt` une classe dont l'utilisateur de `LifoInt` ne doit en aucun cas avoir accès. On conditionne donc la classe `LifoInt` et la classe `Elt` dans un même package `lifoInt`. `LifoInt` y est `public`, `Elt` reste uniquement avec l'accès package.
- Le source java suit...

A stack : the source code

File : `examples/lifoInt/LifoInt.java`

examples/lifoInt/LifoInt.java

```
package lifoInt;                                    // implémentation par une liste une chainée d'Elt

public class LifoInt {
    private Elt Top=null;
    private boolean isEmpty(){ return Top==null; }
    public void pop(){
        if (isEmpty()){ }   //exception
        else{ Top=Top.getNext();}
    }
    public void push( int pValeur ){
        Top = new Elt(Top, pValeur);
    }
    public int getTop(){
        return (isEmpty() ? 0 : Top.getValue());
    }
}
```

Slide 142

LifoInt est la classe publique du package lifoInt.

File: `examples/lifoInt/Elt.java`

examples/lifoInt/Elt.java

```
package lifoInt;
class Elt {
        private int Value;
        private Elt Next;
        Elt( Elt pNext, int pValue ){
            Next=pNext; Value=pValue;}
        Elt getNext(){ return Next; };
        int getValue(){ return Value; }}
```

Slide 143

Elt est la classe “package” du package lifoInt. Dans cet exemple, seul LifoInt peut voir Elt.

Il faut noter beaucoup de chose sur ces exemples :

- Un package peut etre multi-fichiers (déclaration package en tete de chaque fichier.
- Une classe peut être publique cela signifie que seule cette classe est visible pour du code utilisant ce package. Dans cet exemple seul LifoInt peut voir Elt.

2.9.3 Public, Private, Protected and Package

public, private, package

- 2 protections :
  1. The class: Methods or Fields public/private/protected, public/private, protected.
  2. the package: a group of classes friends.
- Determine visibility rules for :
  1. an instance of a class vs an instance of the same class,
  2. an instance of a class vs an instance of the another class but in the same package,
  3. an instance of a class vs an instance of the another class but in another package.

Slide 144

Public, Private, Package

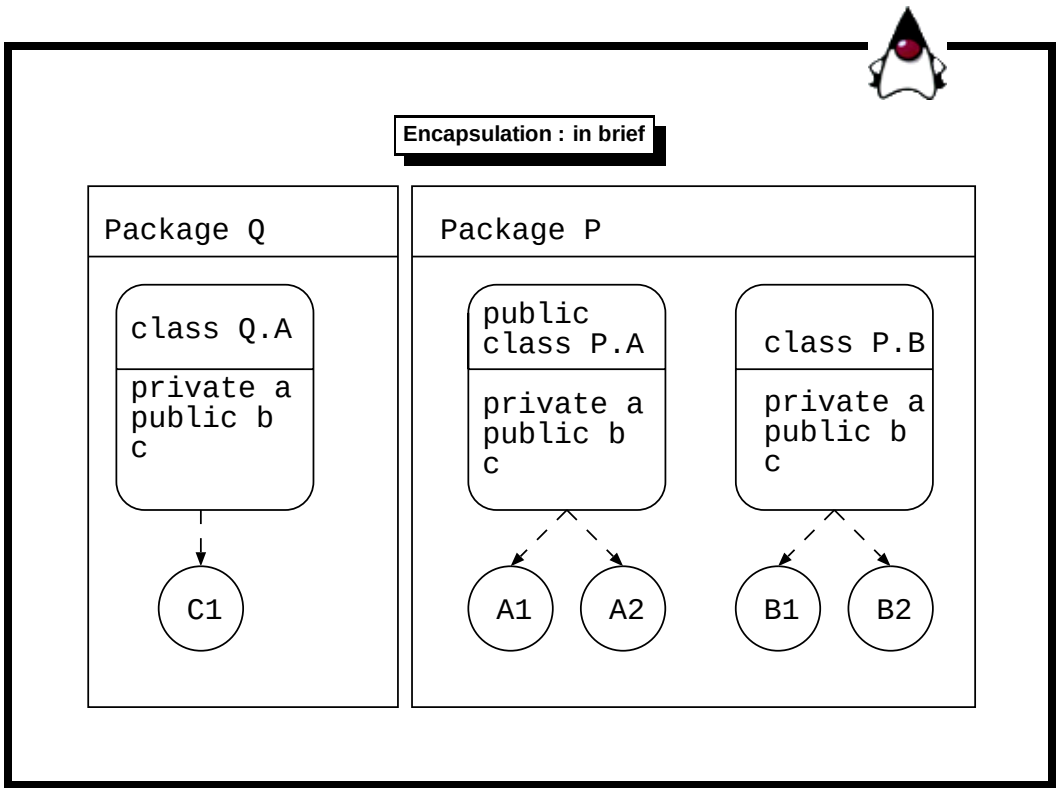
- A field or a method can be private, public or package.

  private : a private field or method is visible by the other instances of the same class (encapsulation is based on the classes).

  package : a field or method of an instance package is visible by all the other instance of the other classes in the same package.

  public : a public field or method is visible by all the instances of the other classes.

Slide 145

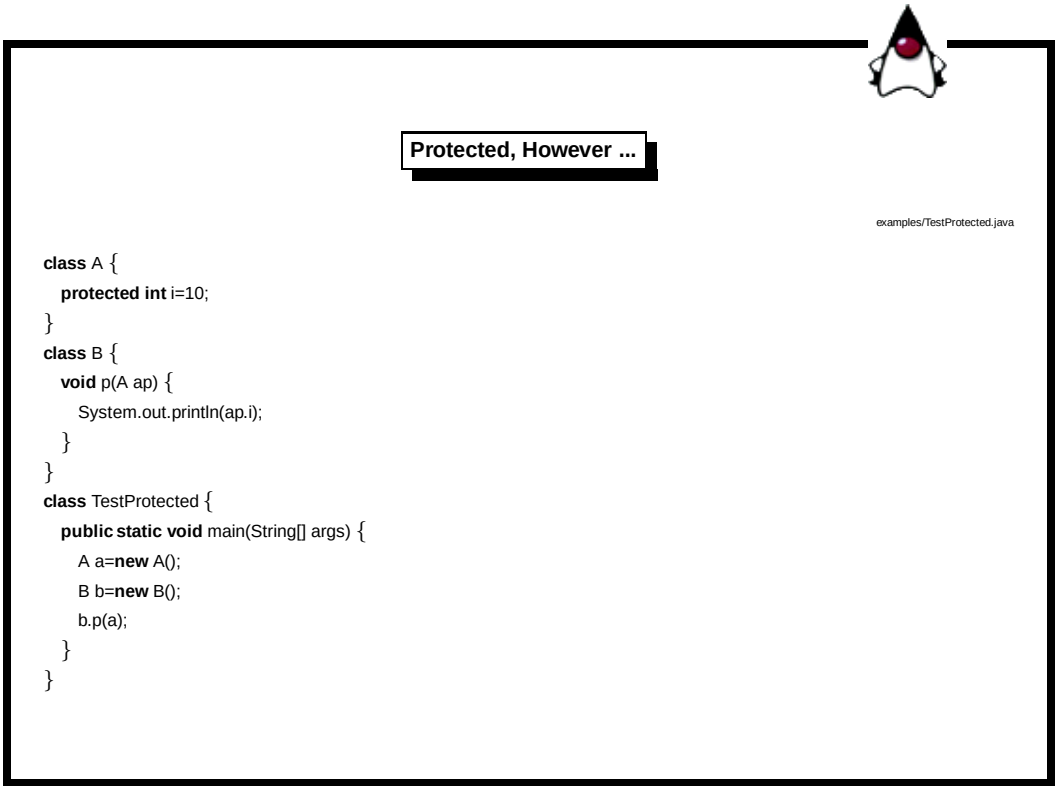


Slide 146

Quizz:

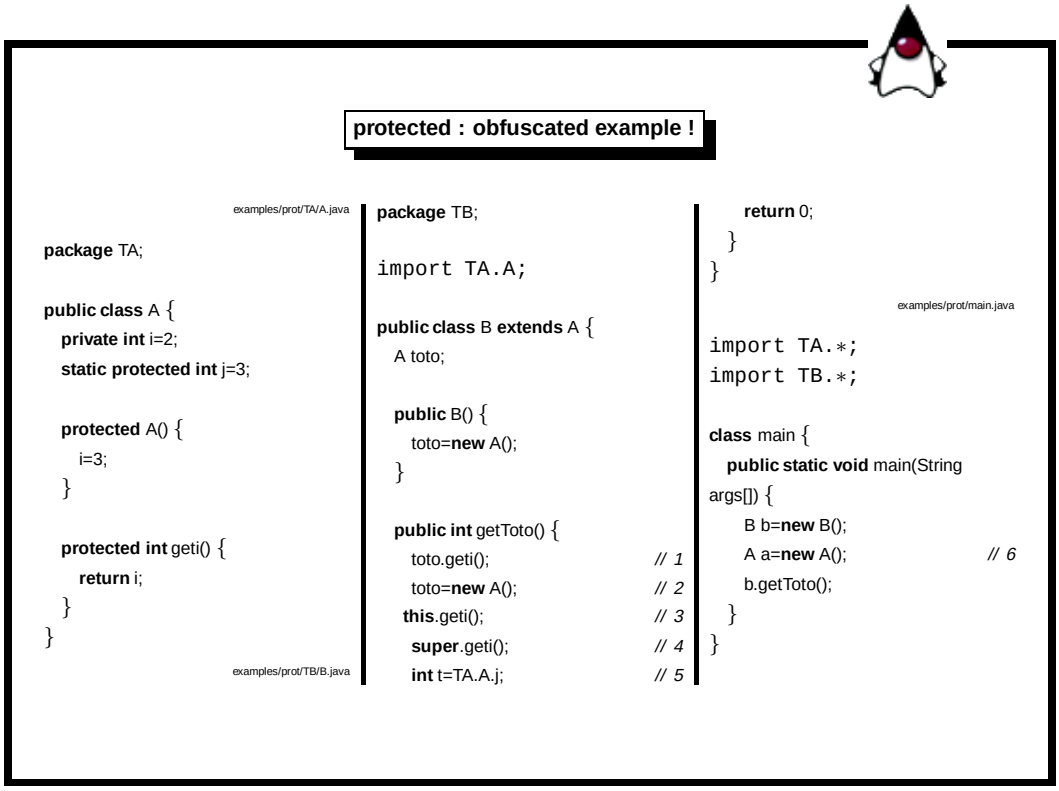
- Si dans une méthode de A1, j'ai une référence sur A2. Vois-je A2.a ? A2.b ? A2.c ?
- Si dans une méthode de A1, j'ai une référence sur B1. Je vois B1.a ? B2.a ? B2.c ?
- Si dans une méthode de A1, j'ai une référence sur sur C1. Je vois C1.a ? C1.b ? C1.c ?
- Si dans une méthode de C1, j'ai une référence sur B1, je vois B1.a ? B1.b ? B1.c ? (gnark, gnark ...)

2.9.4 Package, Encapsulation and Inheritance



Slide 147

Ben oui, protected ne signifie rien pour une classe amie ! (du même package). Pour voir un effet quelconque à protected, il faut faire des packages.



Slide 148

Remember the law: "A protected member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object."

1. `TB/B.java:13: Can't access protected method geti in class TA.A. TA.A is not a subclass of the current class. toto.geti();`
2. `ok ?? but A protected constructor can be accessed by a class instance creation expression only from within the package in wich it is defined.`
3. ok
4. ok
5. ok
6. `main.java:7: No constructor matching A() found in class TA.A. A a=new A();`

Contents

- Introduction
- Language: classes, inheritance, polymorphism, Packages...
- **Threads**
- Advanced Window Toolkit (AWT)
- Applets
- Conclusion

Slide 149

3 Threads

3.1 Overview

Overview

- Thread = A “lightweight” process: a single sequential flow of control within a program.
- → Multi-tasking independent of a plate-forme. Ex: java machine on windows3.1.
- Integrated part of the language : the synchronization primitives are key words of the language : `Synchronized`.
- Beware : All the threads executing on a virtual machine shared the same memory space :
  - Advantage : communication inter-process is easy and very efficient (no overhead).
  - Inconvenient ?

Slide 150

Quand même !! Avoir un système multi-tâche de manière indépendante d'une architecture est un point assez important. Bien sûr, ça existait déjà en ADA ou en Smalltalk mais pas en C ou en C++ ...

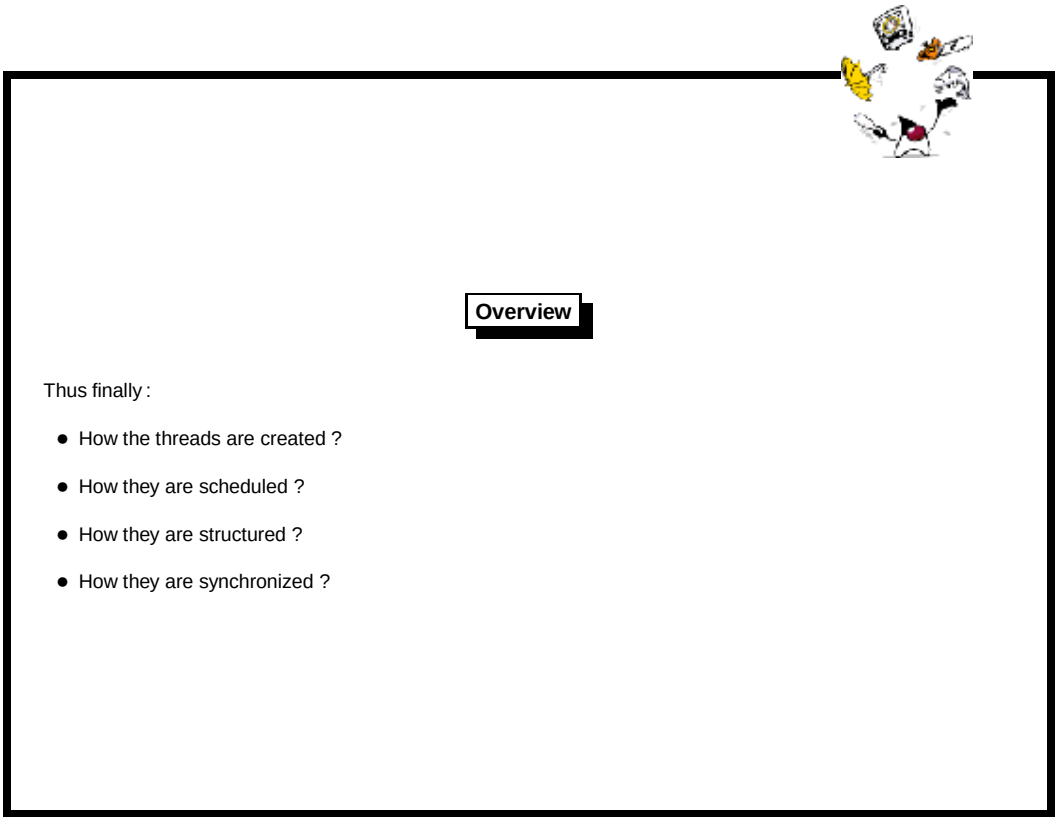
Les threads java sont en fait des *lightweight threads* dans le mesure où tous les threads partagent le même espace mémoire. C'est une philosophie très éloignée d'un système comme UNIX où tous les processus ont un espace mémoire privé et protégé des autres espaces des autres processus. Pour avoir un espace partagé il faut créer explicitement des mémoires partagées. En java, finalement tout fonctionne comme si la mémoire de la machine virtuelle est une grosse mémoire partagée entre tous les threads.

Inconvénient : Il faut bien se rendre compte qu'*a priori* en java il n'y a pas de manipulation de mémoire (pas de memcopy ou de manip de pointeurs) ! Il n'est donc pas possible de corrompre la mémoire soit de manière intentionnelle soit par des débordements de tableaux intempestifs ou des manipulation de pointeurs. Si un thread est planté seul les threads communiquant avec lui seront génés (comme ce serait le cas sous unix). Sinon il n'est pas possible qu'un thread même mal écrit viennent corrompre la représentation mémoire d'un autre thread !

Alors quelle est l'avantage fondamental entre la gestion de processus à la UNIX est la gestion de threads à la JAVA ?

En Unix les processus sont beaucoup plus lourd, mais la gestion du contrôle d'accès entre processus semble plus riche en UNIX : notion de propriétaire d'un processus, groupe de processus...

À revoir ...



Slide 151

Cette présentation est une extraction/concentration du tutorial (pour le plan), avec un peu du java source book [Anu96] et toujours la spécification du langage.

3.2 Example

Threads

- Threads are objects ! Thus, there is a class Thread
- To create a thread, we have to :
  1. Subclass the class Thread.
  2. Override the method Run() with its own code.
  3. Instantiate this class.

Slide 152

Example

examples/TwoThreadsTest.java

```
class TwoThreadsTest {
    public static void main (String[] args) {
      new SimpleThread("Jamaica").start();
      new SimpleThread("Fiji").start();
    }
}
class SimpleThread extends Thread {
    public SimpleThread(String str) {
      super(str);
    }
    public void run() {
      for (int i = 0; i < 10; i++) {
        System.out.println(i + " " + getName());
        try {
          sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {}
      }
      System.out.println("DONE! " + getName());
    }
}
```

Slide 153

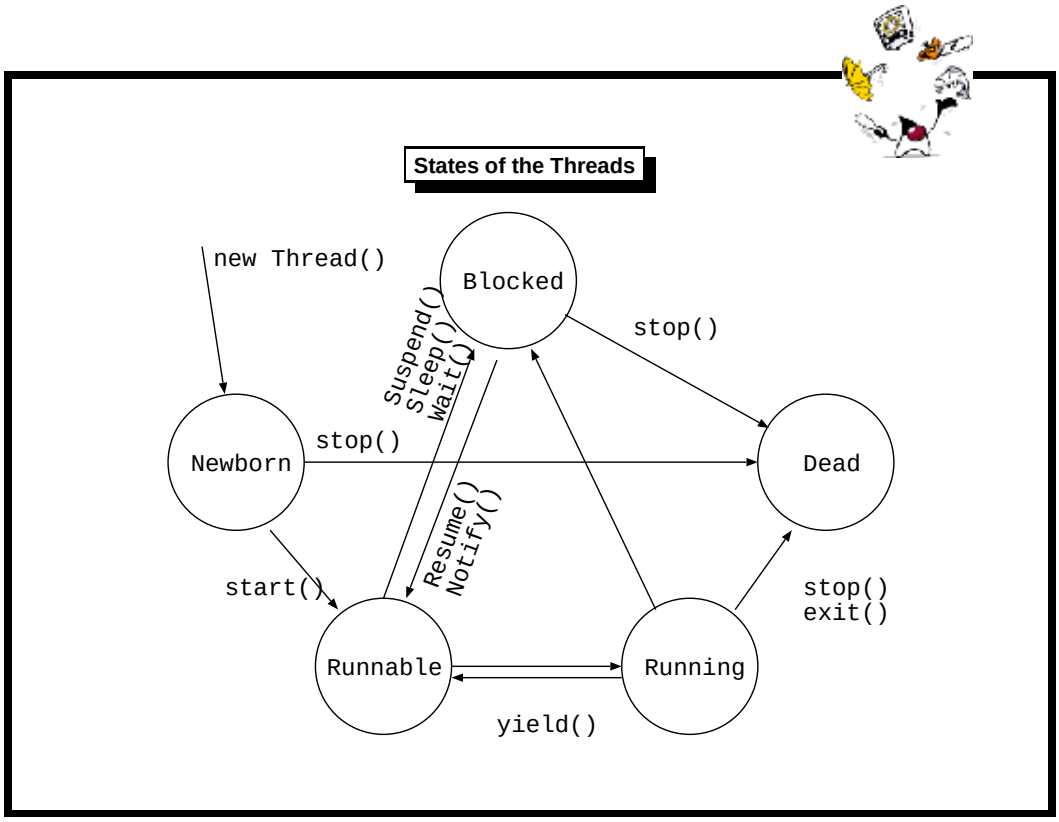
- Il faut bien noter que la méthode `start` retourne immédiatement. Le thread concerné est lancé, le code qui a fait le `start` continue et sera éventuellement préempté par la suite.

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Fiji
2 Jamaica
3 Jamaica
4 Jamaica
3 Fiji
4 Fiji
5 Jamaica
5 Fiji
6 Jamaica
7 Jamaica
6 Fiji
8 Jamaica
9 Jamaica
7 Fiji
DONE! Jamaica
8 Fiji
9 Fiji
DONE! Fiji
```

Slide 154

3.3 Threads Attributes

3.3.1 The States of the Threads



Slide 155

3.3.2 Priorities

Priorities

- To ensure the sharing of a CPU among several threads →
- Scheduling based on fixed priorities (and not floating !).
- At a given instant, if several threads are in the state “runnable”, the thread of higher priority pass in the state running.
- If several threads have the same priority, then the Scheduler behaves differently according to the host system :
  - For system supporting time-slicing: time slicing
  - For other: Just a sequence

Slide 156

En Unix, aux dernières nouvelles, les processus ont des priorités flottantes. Moins un processus est executé, plus sa priorité monte, une fois qu'il est passé dans le processeur, sa priorité diminue. Ce n'est pas le cas en JAVA. Les threads ont une priorité fixe déterminée par la priorité du thread qui leur a donné naissance. Bien sur, cette priorité peut être changée par le corps du thread (ou de l'extérieur : `setPriority` est `public`) lui-même mais en aucun pas par l'ordonnanceur. C'est tout de même assez primitif... La encore, on peut ressentir le compromis constant avec les différents systèmes pouvant acceuillir le système JAVA. Comme toutes les machines hôtes ne savent pas faire du temps partagé, on se replie sur des solutions plus primitives mais plus générales.

On peut aussi voir la priorité des threads comme une sorte de *nice* UNIX. La priorité UNIX étant alors vue que comme une technique pour assurer un temps partagé à peu près équitable.

Remarque : La priorité la plus élevée est un entier le plus grand possible dans la limite de `java.lang.Thread.MAX_PRIORITY`. C'est l'inverse de la numérotation UNIX.

Priorities

A thread execute until :

- A thread of a higher priority becomes in the state “runnable”
- It returns the control : yield(), sleep() . . .
- On the systems which support time slicing, its time quota is expired.

At its creation a thread has the same priority as its parent.

Slide 157

Priorities

- The system is preemptive, in the context that :
- If at a given instant, a thread t_0 becomes in the state “runnable” with higher priority than the thread t_1 which executes ...
- Then t_1 becomes in the state runnable and t_0 passes in the state “running”.
- Simply, there is no pre-emption among threads with the same priority. However, an implementation on a machine could allow time slicing...
- According to the JVM, we have time slicing or not ... Thus you don’t have to make hypothesis when you code!

Slide 158

- Attention : si il n'y a pas de temps partagé pour les threads, leur intéret dans ce cas pour gérer des entrées-sorties bloquantes semble limité. Or ceci est un des usages de base des threads : le monitoring d'un stream sur un serveur, et éviter des `fork`, ou de gérer explicitement une "top level loop".
- Quand même bizarre...

Priorities : Example

examples/RaceTest.java

```
class RaceTest {
  final static int NUMRUNNERS = 2;
  public static void main(String[] args) {
    SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i] = new SelfishRunner(i); runners[i].setPriority(2); }
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i].start(); }
  }
}
class SelfishRunner extends Thread {
  public int tick = 1;
  public int num;
  SelfishRunner(int num) { this.num = num; }
  public void run() {
    while (tick < 400000) { tick++; if ((tick % 50000) == 0) { System.out.println("Thread #" + num + ", tick = "
+ tick); }}
  }
}
```

Slide 159

Priorities : on Solaris

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

Slide 160

OUARRGGG !! pas de temps partagé sur Solaris !!!!!

Remarque : Les threads (même POSIX !) sont disponibles sur Solaris. On vit une époque formidable. `man thr_create`.

Priorities : On Windows95

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 50000
Thread #0, tick = 150000
Thread #1, tick = 100000
Thread #0, tick = 200000
Thread #1, tick = 150000
Thread #0, tick = 250000
Thread #1, tick = 200000
Thread #0, tick = 300000
Thread #1, tick = 250000
Thread #0, tick = 350000
Thread #1, tick = 300000
Thread #0, tick = 400000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

Slide 161

Sighh, ça marche mieux sur windows95 (Nous suis déçu ... 8-<).
Petite Annonce : Vends Sun Ultra Sparc 1 140 sous solaris 2.51. Pas cher.

Priorities : yield()

examples/RaceTestSun.java

```
class RaceTestSun {
  final static int NUMRUNNERS = 2;
  public static void main(String[] args) {
    SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i] = new SelfishRunner(i); runners[i].setPriority(2); }
    for (int i = 0; i < NUMRUNNERS; i++) { runners[i].start(); }
  }
}
class SelfishRunner extends Thread {
  public int tick = 1;
  public int num;
  SelfishRunner(int num) { this.num = num; }
  public void run() {
    while (tick < 400000) { tick++;
    if ((tick % 50000) == 0) {
        System.out.println("Thread #" + num + ", tick = " + tick);
        yield(); }  // next thread of same priority !!
    } } }
```

Slide 162

Priorities: yield()

- Give the control explicitely with `yield()`
- Beware! A thread which call yield() can give the control only to a thread which has the SAME PRIORITY !!

Slide 163

Priorities: solaris

```
Thread #0, tick = 50000
Thread #1, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #0, tick = 150000
Thread #1, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 200000
Thread #0, tick = 250000
Thread #1, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #0, tick = 350000
Thread #1, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000
```

Slide 164

Jouons avec les priorités maintenant ...

Priorities

examples/RaceTest2.java

```
class RaceTest2 {
  public static void main(String[] args) {
    SelfishRunner runner1 = new SelfishRunner(1);
    SelfishRunner2 runner2 = new SelfishRunner2(2);
    runner1.setPriority(2);
    runner2.setPriority(2);
    runner1.start();
    runner2.start();
  }}
class SelfishRunner extends Thread {
  public int tick = 1;
  public int num;
  SelfishRunner(int num) { this.num = num; }
    public void run() {
    while (tick < 400000) {
      tick++;
      if ((tick % 50000) == 0) {
System.out.println("Thread #" + num + ", tick =
" + tick); }
    }
  }}
class SelfishRunner2 extends Thread {
  public int tick = 1;
  public int num;
  SelfishRunner2(int num) {
    this.num = num;
  }
  public void run() {
    while (tick < 400000) {
      tick++;
      if ((tick % 50000) == 0) {
        System.out.println("Thread #" + num + ",
tick = " + tick);
      }
        if (tick==200000) {
        this.setPriority(4);
        System.out.println("Thread #" + num +",
priority boosted !");
      }
    }
  }}
```

Slide 165

Le même qu'avant, on change les priorités en cours de route. J'ai plutôt essayé sous windows95 ... Puisqu'on n'utilise pas le `yield` dans l'exemple.

ça donne ça :

Priorities

```
Thread #1, tick = 50000
Thread #2, tick = 50000
Thread #1, tick = 100000
Thread #2, tick = 100000
Thread #1, tick = 150000
Thread #2, tick = 150000
Thread #1, tick = 200000
Thread #2, tick = 200000
Thread #2, priority boosted !
Thread #2, tick = 250000
Thread #2, tick = 300000
Thread #2, tick = 350000
Thread #1, tick = 250000 // ??
Thread #2, tick = 400000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

Slide 166

- Les ?? marquent une exécution qui semble incorrecte puisque le thread numéro 2 posséde à ce moment une plus grande priorité que le thread numéro 1. Mais il est bien dit (et nous l'avons déjà dit) dans [JG96][page 589] qu'un schedule exact n'est pas garanti.

3.3.3 Thread daemons

Daemons

- A thread can be declared as “Daemon”.
- It is a service thread for another thread !
- A thread daemon exists only if there is another thread non-daemons
- Thus if only the daemons threads are active, the java machine stops.
- Example : The threads “image fetcher” for HotJava.

Slide 167

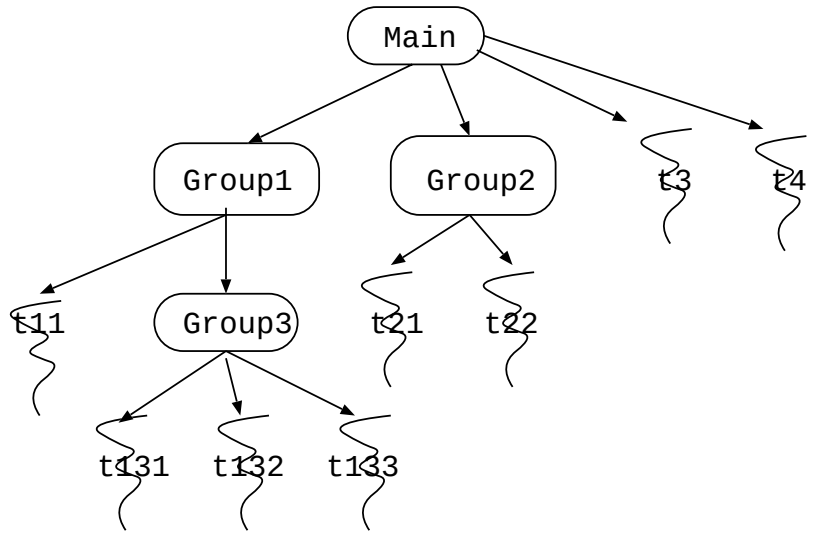
Il faudrait un exemple précis pour motiver l’idée

3.3.4 Groups of Threads

Group of Threads

- A java thread is necessarily a member of a group !
- A group of threads has 0-n threads or group of threads
- There is thus a hierarchical organisation of the group of threads ...

Slide 168



Slide 169

Why ?

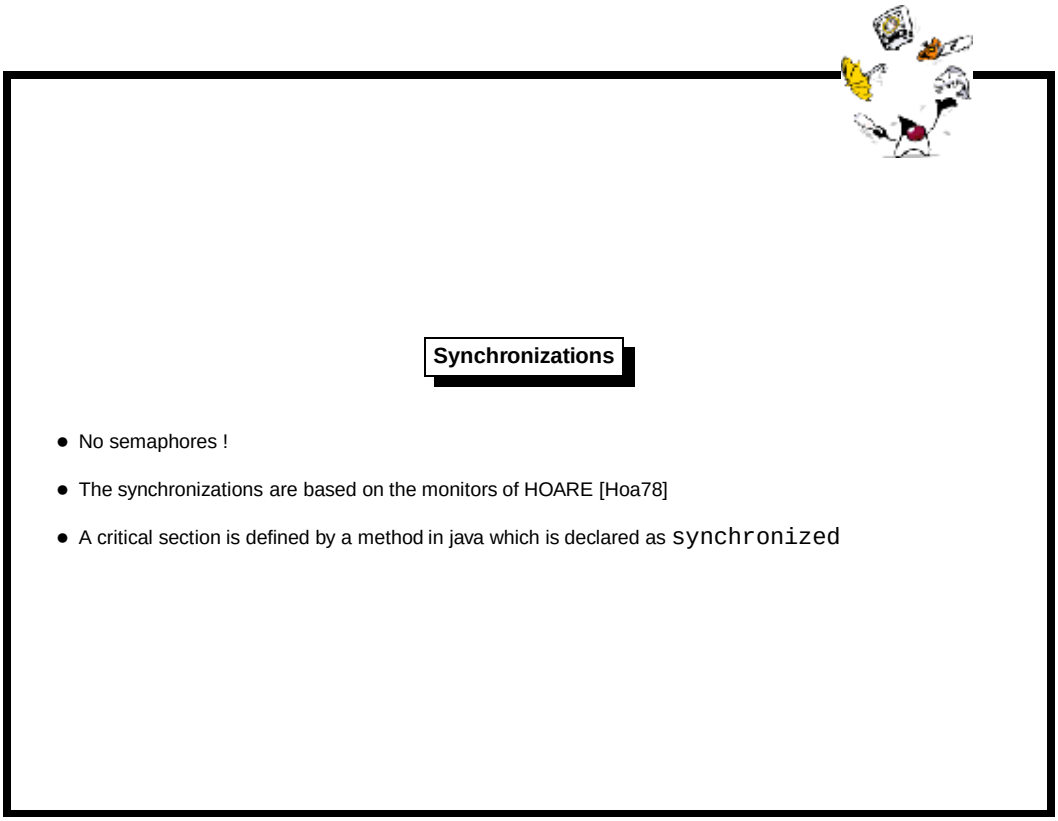
Security :

- A thread of a group can not create a thread in another group without the permission of the security manager.
- Each group has a priority maximal limited which can not exceeded ⇒ produces scales of priorities.
- The groups are named ⟶ space names for the thread (to facilitate the debugging ...).
- Handling a set of threads (`stop, suspend, notifyall` ...)

Slide 170

3.4 Multi-Threaded Programs

3.4.1 Synchronization



Slide 171

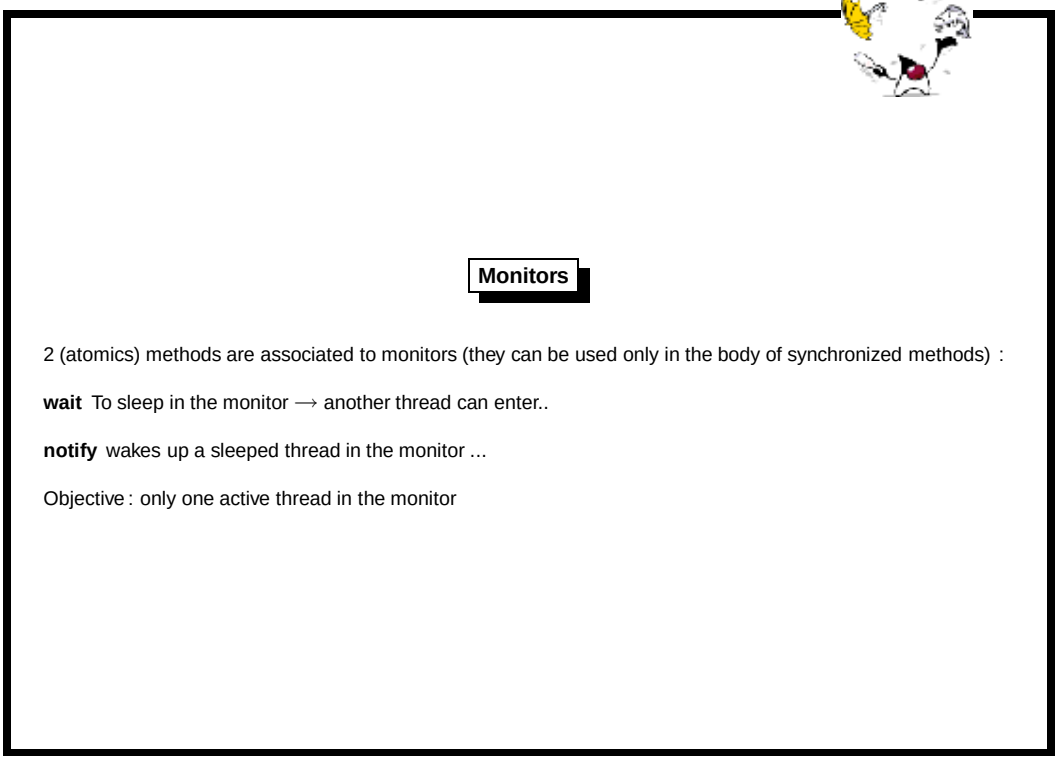
Visiblement, il est possible de déclarer des sections de code comme `synchronized` mais c'est fortement déconseillé par les concepteurs du langages.

Monitors

- A monitor is generally associated to a data and functions like a lock on this data.
- When a thread has the monitor of a data, the others threads are locked. They can't neither read nor modify this data.
- In java, an unique monitor is associated to each object which has at least a "Synchronized" method.
- If a Class method is declared as "Synchronized" then a monitor is associated to the instance of the class `Class` which represents the class of the static method considered in the JVM.

Slide 172

Vu la description de `wait` dans [JG96][589], il semble que les moniteurs de java ne sont pas tout à fait des moniteurs de HOARE, comme présentés dans les transparents qui suivent.



Slide 173

Il faut noter qu'en fait l'implantation des moniteurs nécessite deux autres primitives : `entrer`, et `sortir` car la protection d'une zone de code nécessite l'exécution d'un prologue et d'un épilogue **atomique**. En java il semble que c'est l'appel et le retour d'une méthode `synchronized` qui implante ce prologue et cet épilogue pour le code du corps de la méthode.

Pour ceux qui aurait des trous de mémoire (comme nous) voilà un peu de culture sur les moniteurs :

3.4.2 CIS 307: Implementing Hoare's Monitors

From : `http://www.cis.temple.edu /~ingargio /cis307 /readings /monitor.html`

Monitors are treated very nicely in Tanenbaum section 2.2.7 and section 2.2.9. Here I show the implementation of monitors using semaphores in the case that the Signal command can be applied anytime within the monitor, not just, as in Tanenbaum, when exiting monitor calls. That is, while Tanenbaum shows the Brinch Hansen implementation for monitors, here we examine the Hoare implementation. This latter implementation is less restrictive and less efficient. It is interesting as an example of complex concurrent program and it is useful if you want to use monitors in your programs. In fact, though monitors were intended for use as language constructs when programming in languages such as Concurrent Pascal, or Concurrent Euclid, monitors, as an idea, can be used very conveniently in our concurrent programs writteng in C and using Unix system services.

Most of the code we present appears in Silberschatz et al: Operating Systems Concepts. We want to show how to implement a monitor using semaphores.

- Global variables

  mutex : Semaphore initialized to 1; It is used to control the number of processes allowed in the monitor.

  next : Semaphore initialized to 0; it is used as a waiting queue by processes that are in the monitor and wants to allow other processes to run in the monitor.

next_count : integer variable initialized to 0; It counts the number of processes sleeping in the next semaphore. It is always equal to the number of processes executing monitor operations, minus 1. [This is a complex way of saying that only one process at one time can be executing within a monitor operation, this process is called the active process.]

- Implementation of Condition variables as instances of the abstract data type, or class, Condition [1]:

```
type condition is record
 count   : integer initialized to 0;
 queue   : semaphore initialized to 0;
end;
procedure wait(x : in out condition) is
begin
 x.count := x.count + 1;
  if next_count > 0 then V(x.queue)
                    else V(mutex);
   P(x.queue);
   x.count := x.count - 1;
 end;
procedure signal(x : in out condition) is
begin
 if x.count > 0 then begin
  next_count := next_count + 1;
  V(x.queue);
  P(next);
  next_count := next_count - 1;
 end;
end;
```

Prologue Code executed when starting execution of a monitor's operation:

```
P(mutex);
```

Epilogue Code executed when ending execution of a monitor's operation:

```
if next_count > 0 then V(next) else V(mutex);
```

Notes

- A Monitor is not a process, not an active entity. It is just an abstract data type, or class, whose code can be executed by only one process at a time.
- Let's notice that next_count is only modified within Condition operations. Thus if we do not use condition variables, the prologue code and epilogue code just make sure that only one process is executing within the monitor.
- When condition variables are used within monitor procedures, they are used only to allow processes that do not want to continue at this point, to get out of the way to allow other processes to run. These processes will sleep in the semaphore turn and next_count keeps track of their number.
- The wait operation on conditions always puts to sleep the process that executes it. But before going to sleep, the current process either wakes up a process waiting in the next semaphore or, if next was empty, opens the mutex semaphore to allow some

[1] Il y a une erreur dans une des primitives, à vous de voir où elle est ...

other process to enter from the outside. Notice that the wait operation puts to sleep the active process and gives precedence to the processes that are sleeping withing the monitor over the processes that are trying to enter the monitor.

- The signal operation on condition variables is null if there is no process currently waiting on that condition.
- The code for the Wait and Signal operations of Conditions, and for the Epilogue code, need not be protected as critical regions because only one process is executing in the monitor at a time.
- If a process A executes the signal operation on a condition variable, and if a process B is the highest priority process waiting on this condition, then A moves to Ready state and B to Running state. That is, B has priority over A. We could have implemented the signal operation so that A has priority on B until A exits the monitor.

Within the monitor (intermediate) scheduling takes place. We are with three (semaphore) queues:

- The mutex queue, where processes waiting to enter the monitor wait
- The next queue, where processes that are within the monitor and ready, wait to become the active process, and
- The condition queue(s), where a blocked process waits to be signaled.

In the implementation we have shown processes that are coming out of the condition queue take precedence over the active process and over the next queue; in turn processes in the next queue take precedence over processes on the mutex queue. These decisions are based on some FIFO ideas, but alternative decisions could be made.

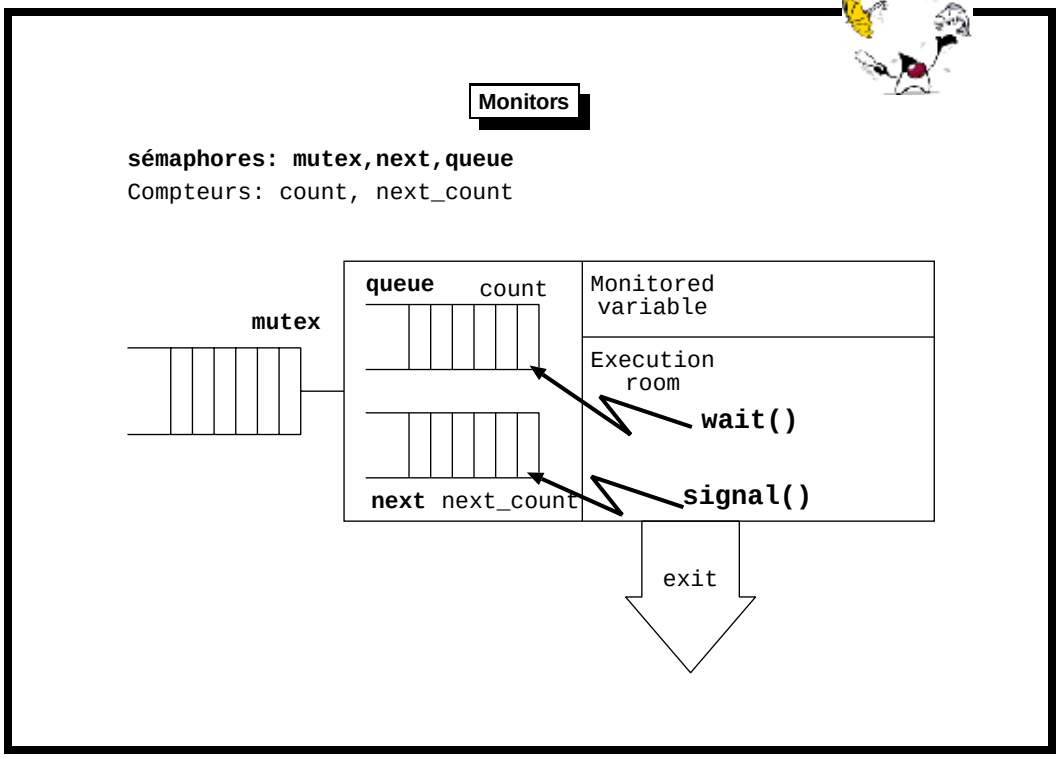
ingargiola.cis.temple.edu

Monitors

```
type condition is record
 count   : integer initialized to 0;
 queue   : semaphore initialized to 0;
end;
procedure wait (x : in out condition) is
begin
 x.count := x.count + 1;
  if next_count > 0 then V(x.queue)
                    else V(mutex);
   P(x.queue);
   x.count := x.count - 1;
 end;
procedure signal (x : in out condition) is
begin
 if x.count > 0 then begin
  next_count := next_count + 1;
  V(x.queue);
  P(next);
  next_count := next_count - 1;
 end;
end;
```

Slide 174

Et où elle est l'erreur ici ?? c'est pas V(x.queue) dans Wait mais V(r[illegible]) bien sur ...



Slide 175

Monitors : the producer

```
class Producer extends Thread {
  private CubbyHole cubbyhole;
  private int number;
  public Producer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
  }
  public void run() {
    for (int i = 0; i < 10; i++) {
      cubbyhole.put(i);
      System.out.println("Producer #" + this.number + " put:  " + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) {
      }}}}
```

Slide 176

The Consumer

```
class Consumer extends Thread {
private CubbyHole cubbyhole;
private int number;

public Consumer(CubbyHole c, int number) {
cubbyhole = c;
this.number = number;
}

public void run() {
int value = 0;
for (int i = 0; i < 10; i++) {
value = cubbyhole.get();
System.out.println("Consumer #" + this.number + " got:  " + value);
}}}
```

Slide 177

The object buffer

```
class CubbyHole {
private int contents;
private boolean available = false;
public synchronized int get() {
while (available == false) {
try { wait();} catch (InterruptedException e) {}}
available = false;
notify();
return contents;}
public synchronized void put(int value) {
while (available == true) {
try { wait();} catch (InterruptedException e) {}}
contents = value;available = true;
notify();
}}
```

Slide 178

Le programme principal

```
class ProdConsTest {
public static void main(String[] args) {
CubbyHole c = new CubbyHole();
Producer p1 = new Producer(c, 1);
Consumer c1 = new Consumer(c, 1);

p1.start();
c1.start();
}
}
```

Slide 179

Un producteur consommateur (buffer à une place) en JAVA [1].

Monitors

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

Slide 180

[1] Issu du tutorial de java

Monitors

examples/Reentrant.java

```
class Reentrant {
  public synchronized void a() {
    b();                                          // reentering monitors
    System.out.println("here I am, in a()");
  }
  public synchronized void b() {
    System.out.println("here I am, in b()");
  }
}
```

- The monitors of java are “reentrant”..
- $\longrightarrow$ no deadlock problems with itself ...

Slide 181

Exception of Thread

- Synchronous: diverse consequence of the actions of the thread
  - nothing can be executed after the occurrence of a `throw` (*exceptions are precise*)
- Asynchronous:
  - Invocation of `stop` from another thread,
  - Internal error JVM (ex : pb GC)
  - $\Rightarrow$ a delay is authorised to stop the *thread* in a stable state
- The methods `synchronized`are supported..
- .... ?

Slide 182

Quand une exception est prise en compte (remontée de la pile) il faut être sûr de relâcher les verrous liées aux méthodes `synchronized` qui "pendent" dans la pile (pour le thread actif).

3.4.3 Deadlock, famines ...

Multi-Threads Programs

JAVA does not take in consideration :

- Neither the deadlocks
- Nor the famine
- It is the programmer responsibility to use programming models (producer/consumer etc...) to avoid these problems.

Slide 183

Contents

- Introduction
- Language: classes, inheritance, polymorphism, Packages...
- Threads
- **Advanced Window Toolkit (AWT)**
- Applets
- Conclusion

Slide 184

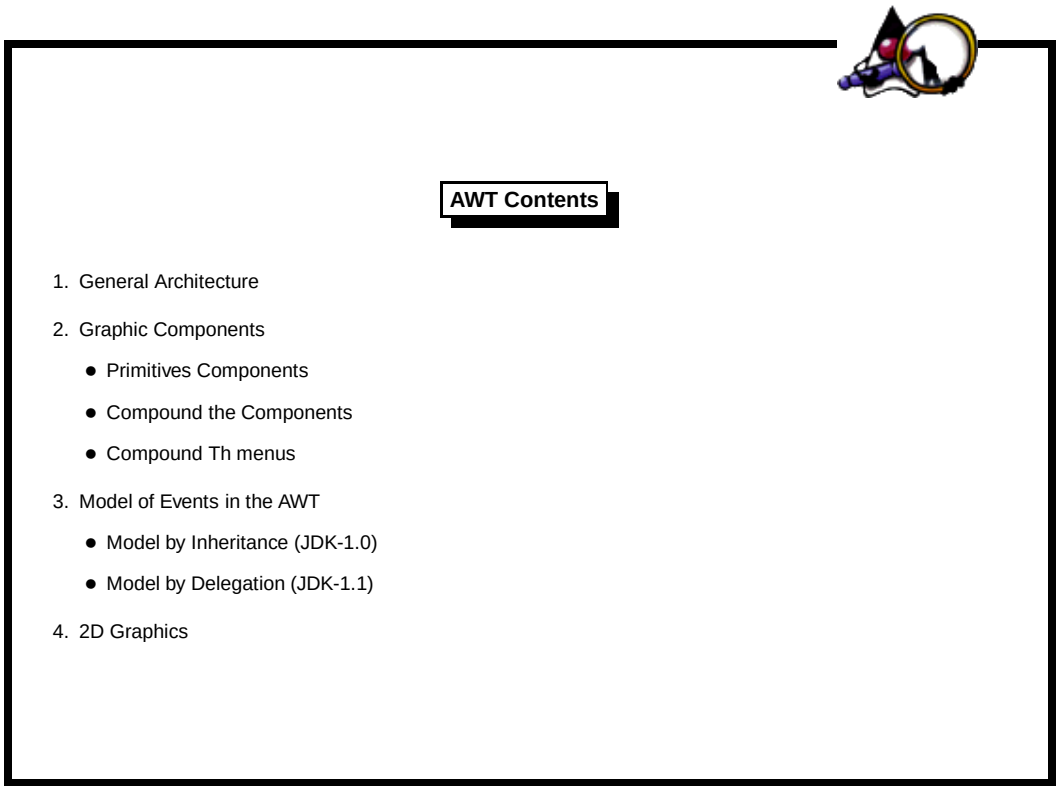
4 Advanced Window Toolkit

Advanced Window Toolkit

The AWT is a package of classes. It allows the construction of sophisticated graphical interfaces. The initial objectives are:

- Having independent graphical system of a particular plate-forme.
- Having an API which can be simply programmed (no new concepts!).
- No new look and feel ! $\longrightarrow$ the same buttons as Motifs under Unix, as Windows95 under NT, as Mac under MacOS

Slide 185

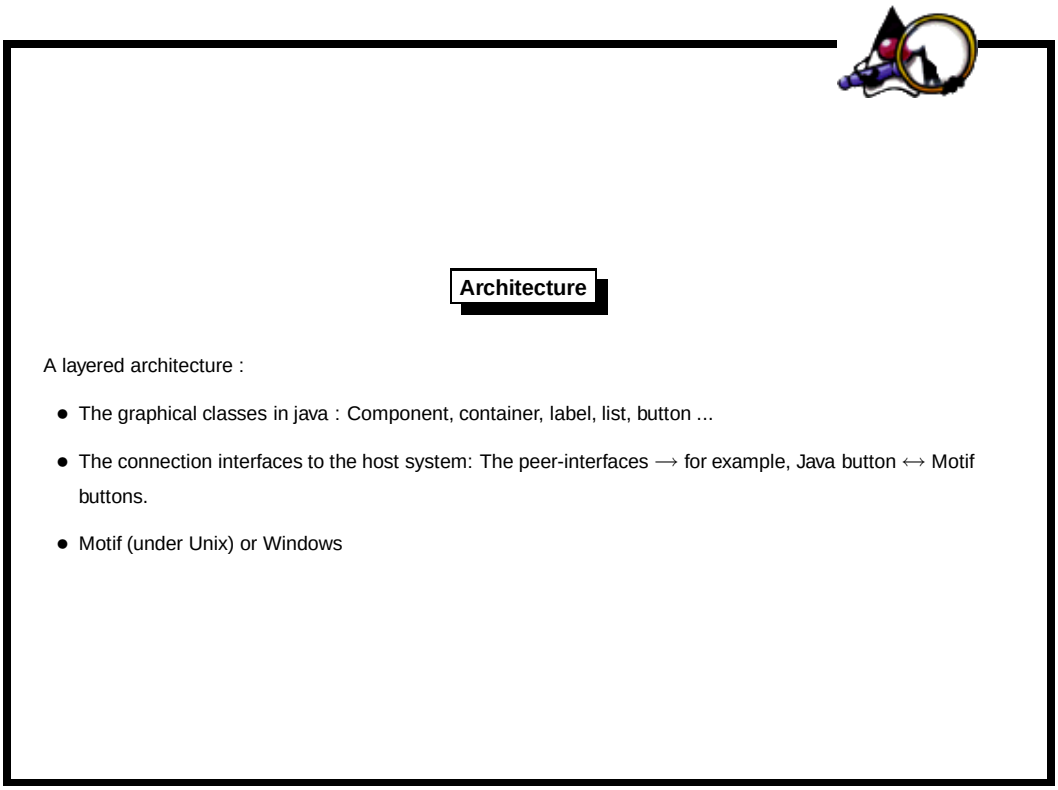


Slide 186

Le dernier JDK propose un modèle d'événements par délégation mais comme tout ce qui existe jusqu'à maintenant utilise l'ancien modèle par d'événements par héritage, nous l'aborderons aussi (et puis en plus c'est intéressant alors ...).

Pour faire ces slides j'ai longuement parcouru, le java tutorial qui utilise malheureusement encore le modèle d'événements par héritage, la java API (indispensable), j'ai été voir les sources (dans l'installation, il faut penser à décompacter src.zip), j'avais la "java source book" [Anu96] à coté de moi et les releases note du JDK-1.1 (en html).

4.1 General Architecture

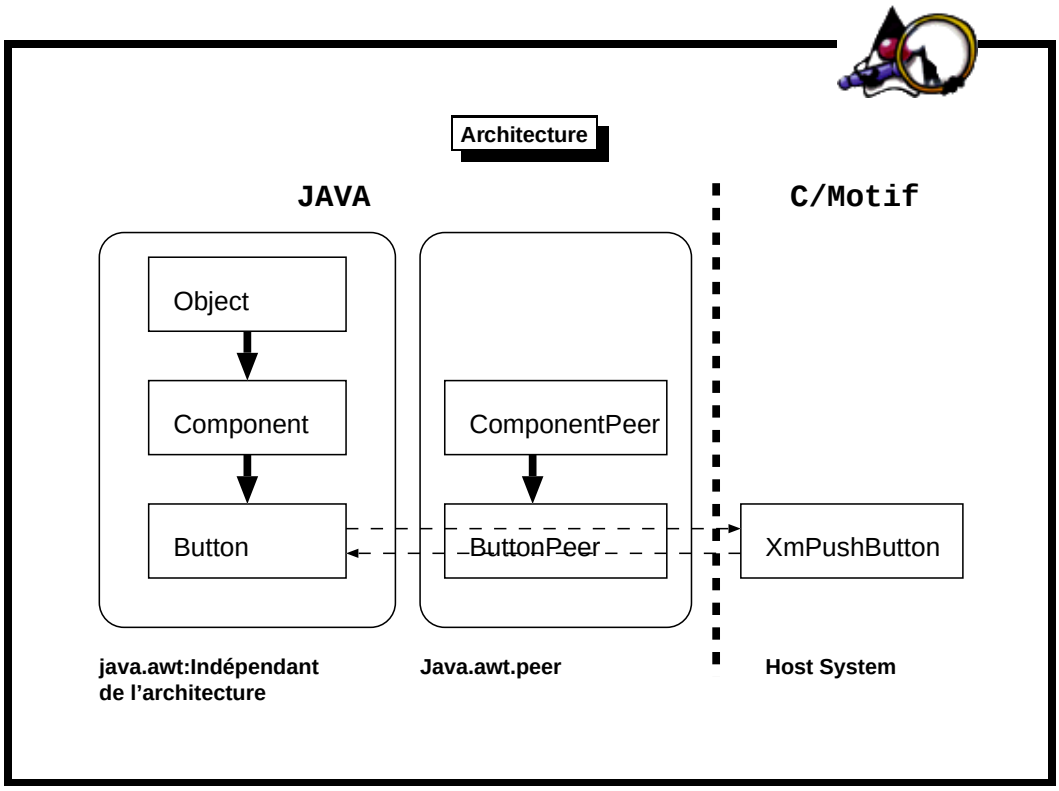


Slide 187

Donc un choix de départ très particulier. En smalltalk, le système graphique est construit complétement en smalltalk. Quelles sont les motivations d'un tel choix ?

- respecter à la lettre les différent look&feel sur chaque plateforme ?
- Essayer de réutiliser au maximum l'exsitant pour construire une interface graphique rapidement ?
- Pourquoi n'ont-ils pas pris l'option de tout reconstruire comme en smalltalk et de réimplanter les look&feel ? trop long ?

J'avoue ne pas avoir de réponses ... la vérité est-elle ailleurs ?



Slide 188

Le programmeur java ne voit que les classes de java.awt. Son code est donc indépendant d'une architecture donnée. Les "peer" ne sont en fait que des interfaces qui se comportent donc comme une spécification des services minimum qui doivent exister dans la toolkit du système hôte.

Ces interfaces doivent bien être implémentée quelque part (mais je ne sais pas où exactement) et sont délivrées par le fournisseur du JDK.

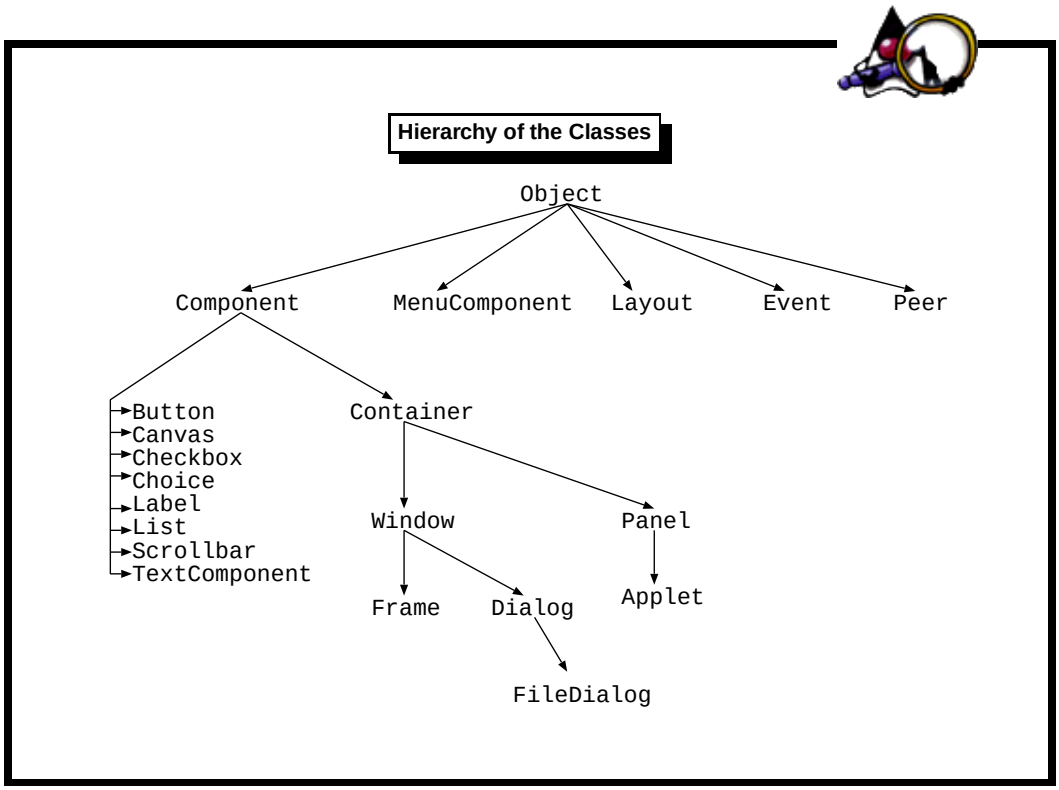
Architecture

- Concretely, for each java button, there is a corresponding Motif button (under Unix) instantiated in the JVM.
- In the JVM, at the reception of an event, the callback of the Motif button generates the creation of a Java event which is considered by the AWT.
- Beware, there is no graphical primitive in the JVM ! The mechanism of native methods is used to implement the peer-interfaces.

Slide 189

Rien n'est très clairement expliqué à ce niveau. On ne trouve rien dans les docs de sun. Toujours est-il que j'ai pu lire des choses non-officielles sur ces aspects sur le web.

4.2 Graphical Components



Slide 190

Les classes graphiques sont bien entendu sous-classes de `object`. On voit déjà bien des choses sur cette arborescence de classes. Il y a des :

- des composants graphiques : avec une différenciation niveau Component/menuComponent.
- des "layout" : des algorithmes de placements.
- des événements.
- des peer (voir architecture).

AWT Organisation

Component : is an abstract object which has a position, a size, able to receive events and that can be drawed.

- Primitive graphical objects: label, button ...
- Container: in some way, “directories” of graphical components.

MenuComponent : The menus are managed in a very special way.

Layout : Algorithms for placing (not attached to container)

Event : The delicate point of the AWT ...

Peer : connexion classes with the graphical toolkit of the host system.

Slide 191

Au premier niveau de la hiérarchie, on trouve les concepts de bases de l'AWT:

Component : Un objet abstrait ayant une position, ue taille, pouvant être dessiné à l'écran et pouvant recevoir des événements. C'est l'équivalent des widgets en Motif.

On peut différencier les objet graphiques primitifs (bouton, label), des “container” qui permettent de composer des objets primitifs ou d'autres “container”. On peut faire l'analogie avec des fichiers qui sont des éléments de stockage primitifs et les repertoires qui peuvent contenir soit des fichiers soit d'autres répertoires. Les “containers” sont donc bien des répertoires d'objets grahiques.

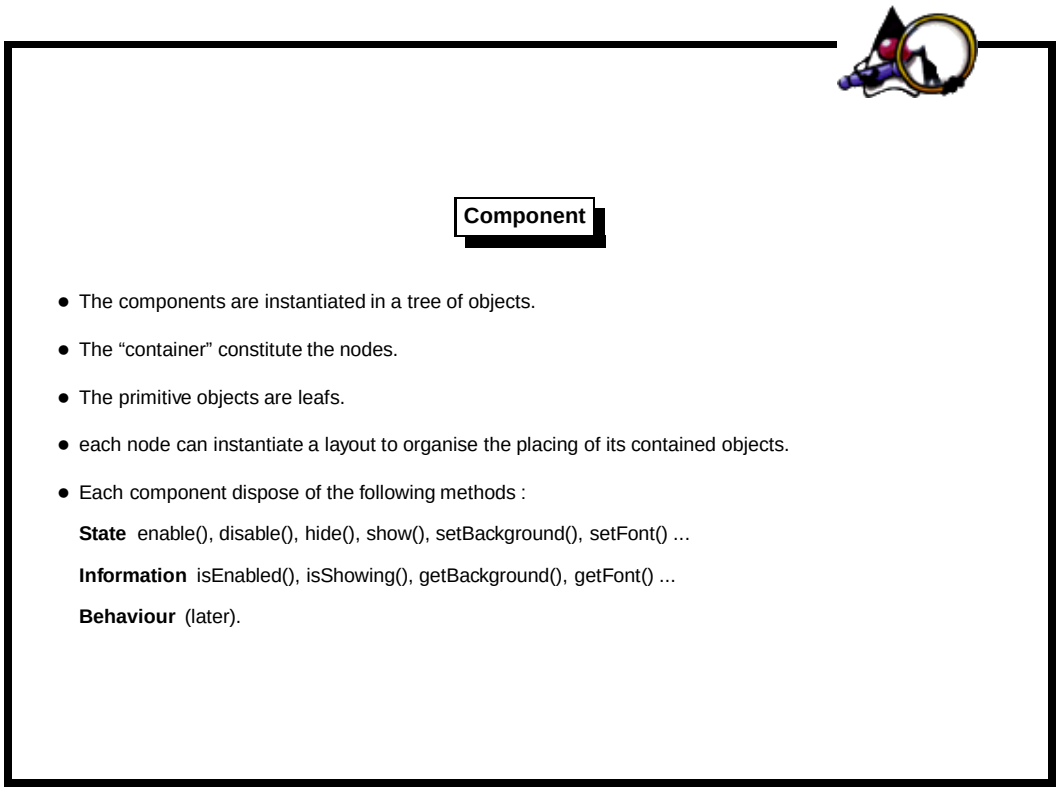
MenuComponent : Toute l'artillerie pour faire des menus (menubar, menuitem, menu, Checkboxmenu). Cette différenciation est due à l'hétérogénéité des systèmes hôtes où les menus sont traités de manière tellement disparates qu'il n'était pas possible d'avoir une seule hiérarchie de composants (comme en Motif).

Layout : Dans une toolkit graphique, les placements des objets grahiques sont calculés et non précisés par des coordonnées absolues. Les “layout” précisent les contraintes de placement des objets graphiques contenu dans un “container”. À la différence de Motif, L'AWT fait la distinction entre “container” et “layout” qui sont définis séparément. De ce fait, il est possible d'affecter dynamiquement un “layout” à un “container” à l'exécution.

Event : Cette classe représente les événements. Le modèle d'évènement de l'AWT est assez différent de Motif et change à chaque release du JDK (ouarg !). Nous reviendrons ultérieurement sur ces modèles qui constitue le point le plus délicat de l'AWT.

Peer Les classes de connexions aux toolkits graphiques des systèmes hôtes. Pour chaque classe de “component” ou “menucomponent” il existe, une classe “componentpeer” ou “menucomponentpeer”.

Nous allons maintenant passer en revue ces différents éléments.



Slide 192

Il faut bien noter qu'un nœud ou une feuille de l'arbre ne peut en aucun cas être effectivement un `Component`. C'est une classe abstraite ! Il ne faut pas confondre l'arbre des objets graphiques à l'exécution et la hiérarchie des classes d'objets de l'awt. C'est comme en Motif (Xt) : classes de widgets et relations entre children à l'exécution.

Attention : pour le jdk 1.1 on dispose de `setEnabled(boolean)`, `setVisible(boolean)`. Le nomage des méthodes a été homogénéïsé. Voir`http://www.javasoft.com/ products/JDK/ 1.1/docs/guide/ awt/index.html` pour plus de précision. Il y a même un script de conversion.

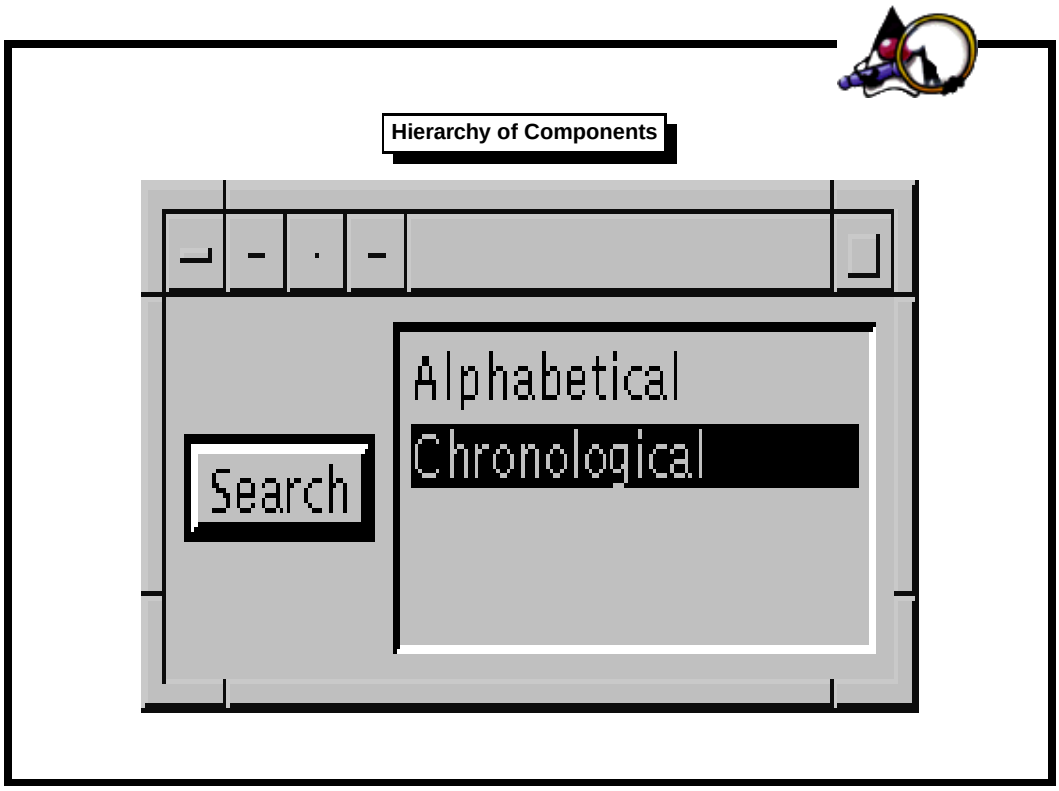
Hierarchy of Components

examples/treeex.java

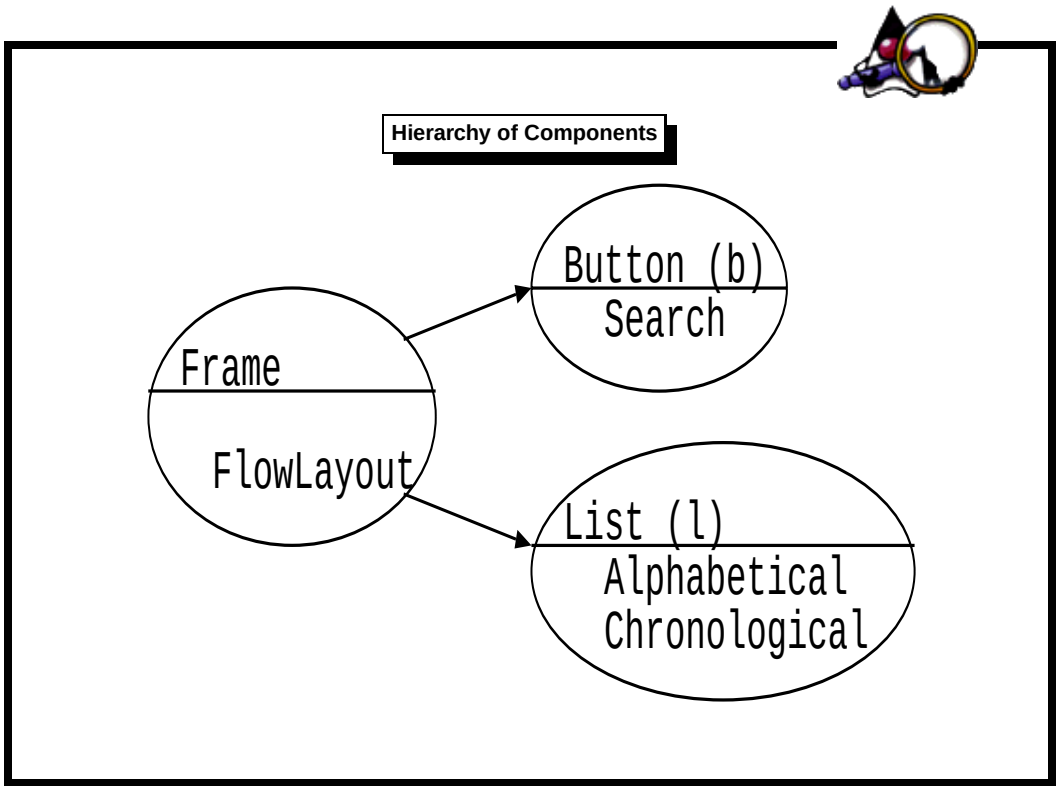
```
import java.awt.*;

public class treeex {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new FlowLayout());
    Button b;
    f.add(b = new Button("Search"));
    List l;
    f.add(l = new List());
    l.add("Alphabetical");
    l.add("Chronological");
    f.pack();
    f.show();                                          // 1.1 : setVisible(true);
  }
}
```

Slide 193

Bon ben, un Fraaammme et une bouton et une Liste dessus. Juste remarquer que ça reste assez simple voir presque agréable à lire (surtout quand on fait du motif de base).



Slide 194



Slide 195

4.2.1 Basic Components

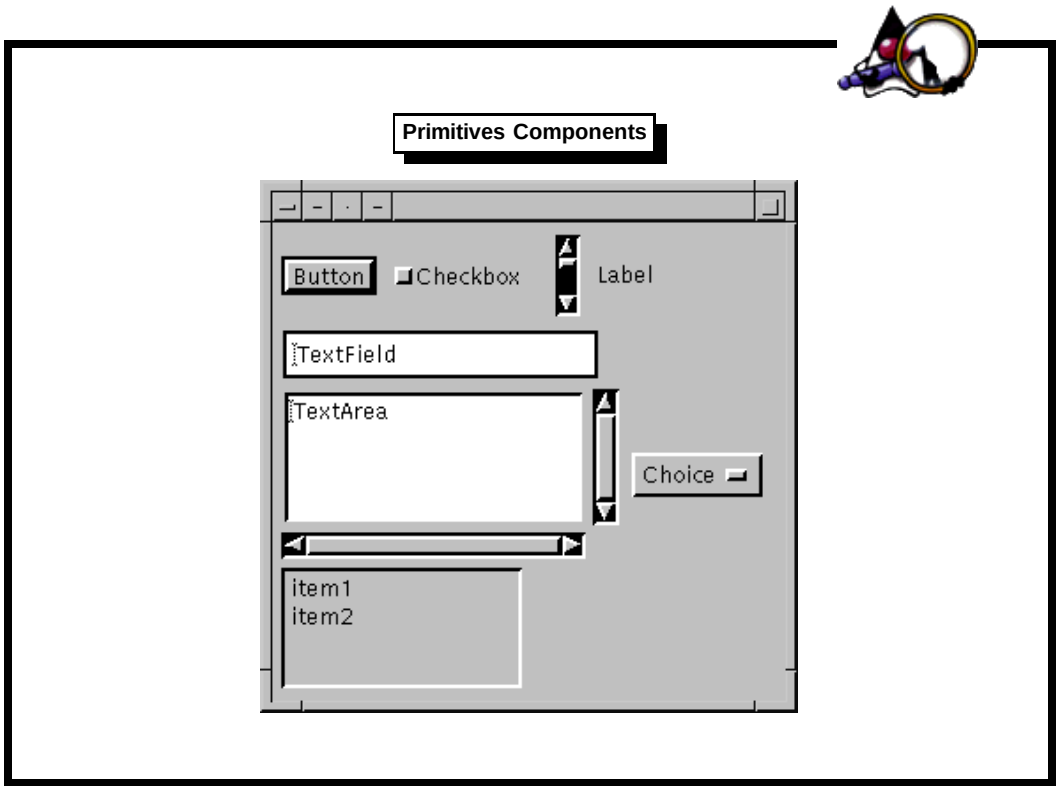
Basic Graphical Classes

- Button, canvas, check-box, choice, label, list, scrollbar, text-component
- "Basic" : They can't contain other graphical components
- In contrast to Motif, the "Components" have not their own resources .
- It is possible to declare properties. These properties are managed by the user.

Slide 196

Les "properties" (java.util) sont en fait basés sur des dictionnaires associant une valeur à une clef. Ces dictionnaires peuvent être sauvegardés dans un fichier. C'est au programmeur de gérer ces dictionnaires. Les "properties" peuvent être utilisées pour gérer des attributs spécifiques à un programme java. Elles peuvent être utilisées pour gérer des ressources comme en Motif mais sans les facilités de gestion des ressources de Motif.

Il semble que le jdk1.1 fournissent un système de ressources, utilisant les properties. Ya meme des utilitaires "a la resedit mac" sous windows.



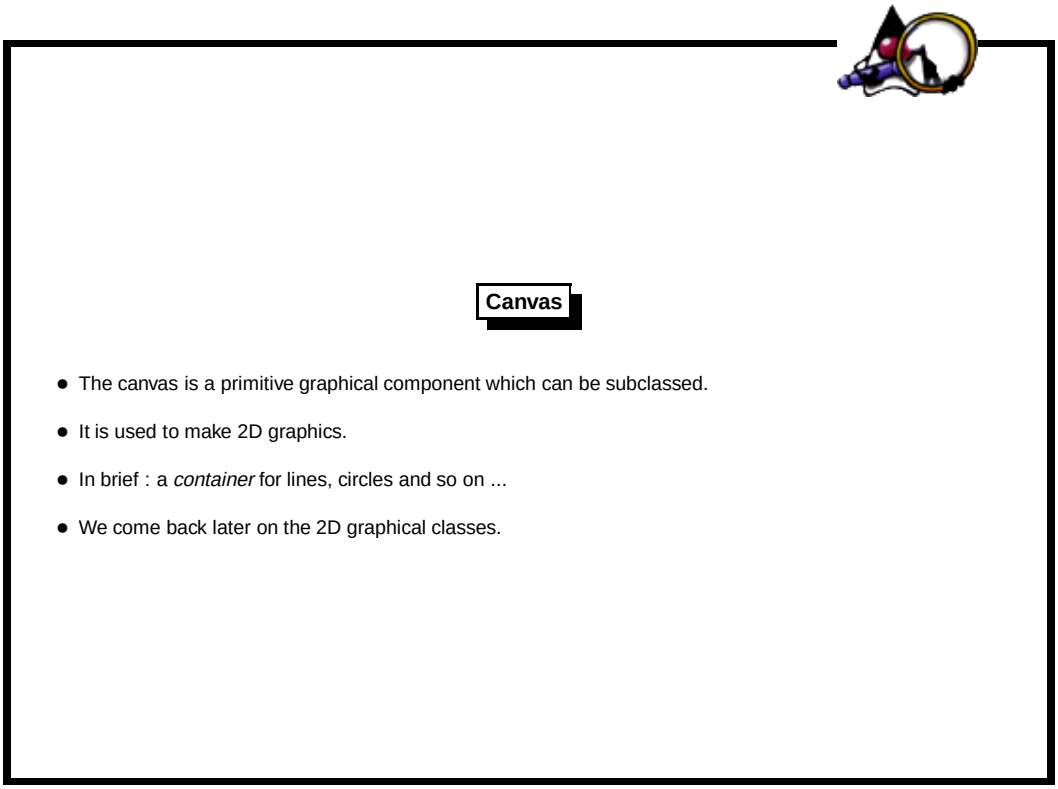
Slide 197

Pour ceux qui veulent jeter un œil sur le source, tout cela reste très simple :

examples/Demo.java

```
import java.awt.*;
public class Demo {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new FlowLayout(FlowLayout.LEFT));
    Button b;
    f.add(b = new Button("Button"));
    Checkbox cbox;
    f.add(cbox= new Checkbox("Checkbox"));
    Scrollbar sb;
    f.add(sb= new Scrollbar());
    Label lab;
    f.add(lab= new Label("Label"));
    TextField tf;
    f.add(tf=new TextField("TextField",20));
    TextArea ta;
    f.add(ta= new TextArea("TextArea",5,20));
    Choice co;
    f.add(co=new Choice());
    co.addItem("Choice");
    List l;
    f.add(l = new List());
    l.add("item1");
    l.add("item2");
    f.pack();
    f.setSize(300,300);
    f.show();
```

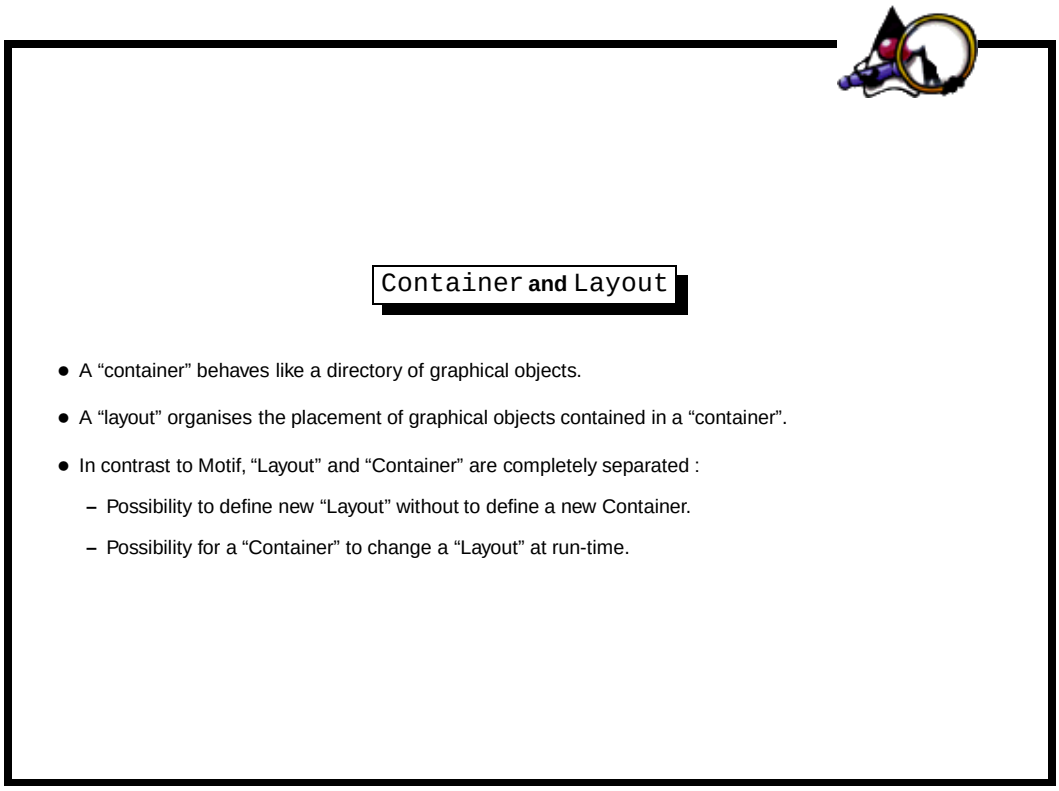
```
  }
}
```



Slide 198

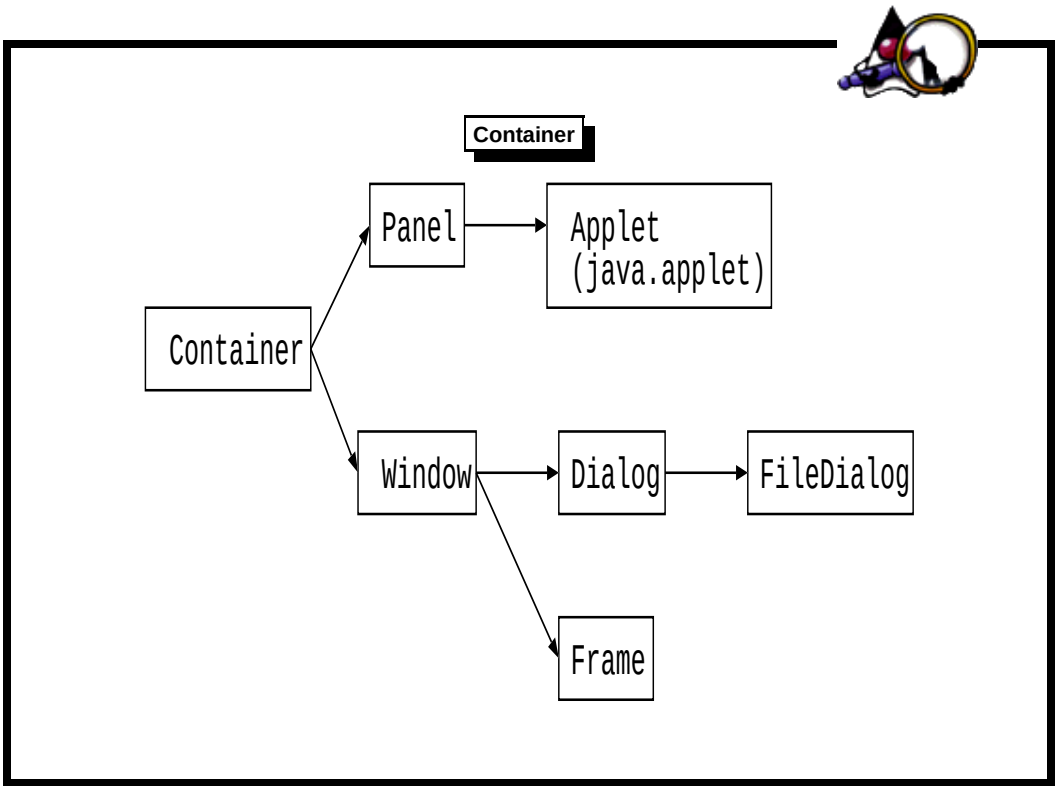
Le jkd1.1 fournit des possibilités de tracés 3D.

4.2.2 Compound the Components

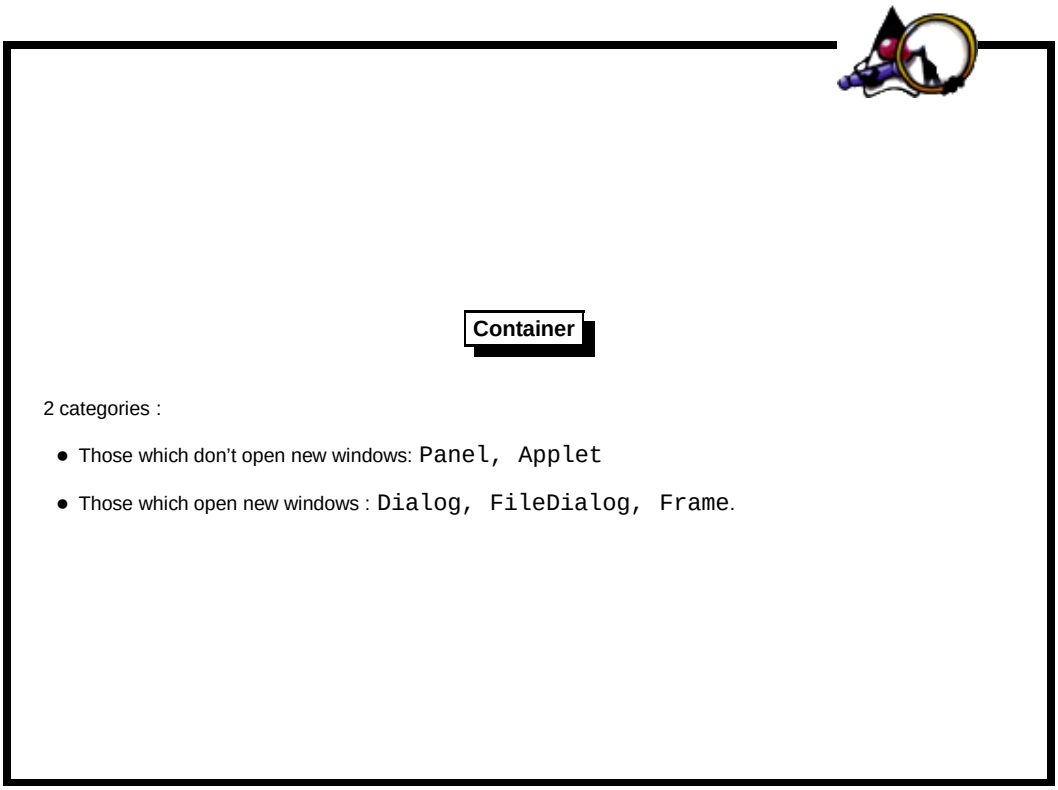


Slide 199

Bien sur, un `Container` peut contenir d'autres `Container`. Cette dissociation `Container/Layout` est tout de même bien agréable par rapport à Motif.



Slide 200



Slide 201

On reviendra lourdement sur les applets plus tard.

Container(2)

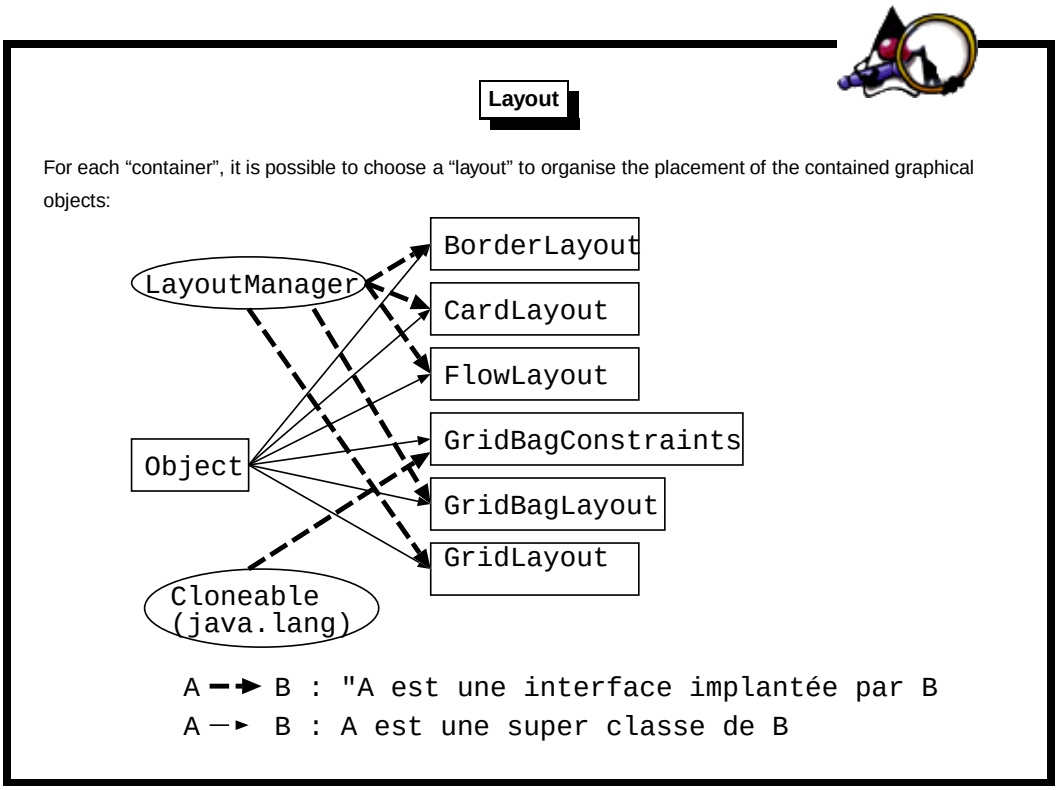
- Each graphical application instantiates at least one “frame”.
- A “dialog” window is designed to be a temporary window.
- A “dialog container” has always a “frame container” as a parent “container”.
- A dialog window is not visible at its creation, it must be made visible explicitly by calling the method show().

Slide 202

Pourquoi cette différenciation “frame” versus “dialog” ? On a déjà vu un frame. On peut jeter un oeil sur la façon de faire un dialogue.

examples/SimpleDialogDemo.java

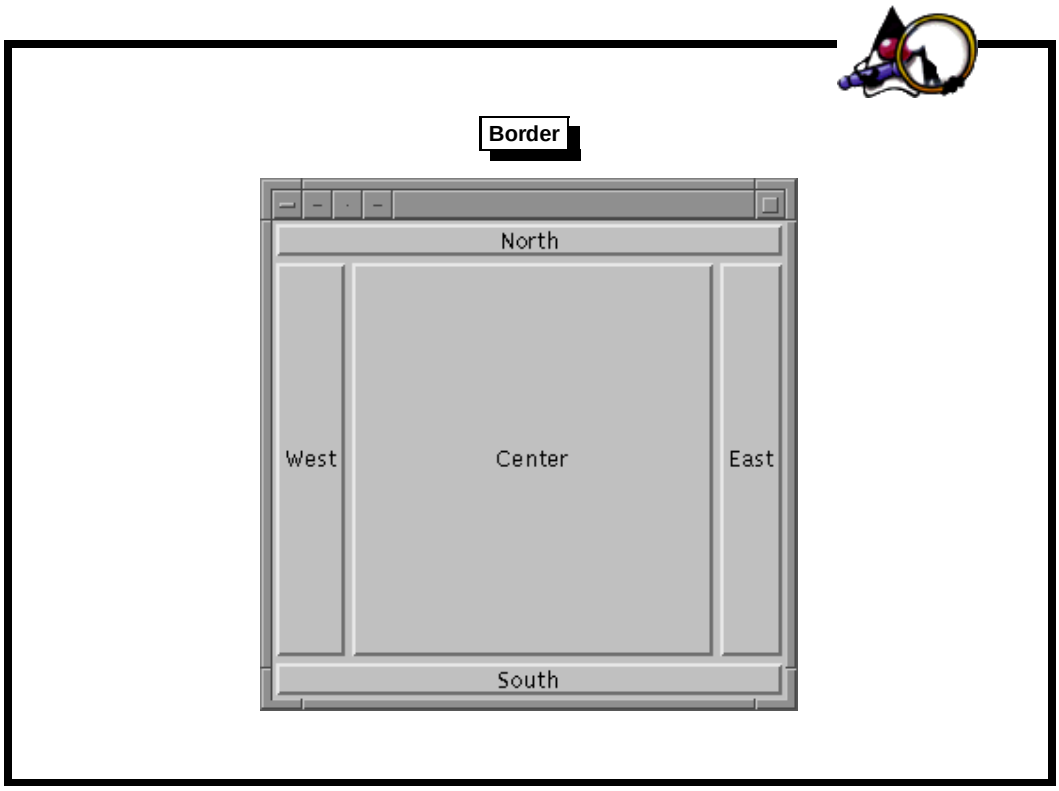
```
import java.awt.*;

public class SimpleDialogDemo {
  static public void main(String args[]) {
    Frame f = new Frame();
    Dialog d = new FileDialog(f,"Yo!");

    f.resize(300,300);
    f.show();
    d.show();
  }
}
```



Slide 203

Pour les applet, le frame “père” est la fenêtre du browser. Il semble que par défaut un dialogue ait un comportement modal.

Passons maintenant en revue les différents layouts (vu qu'il n'y en a pas 36 ...).



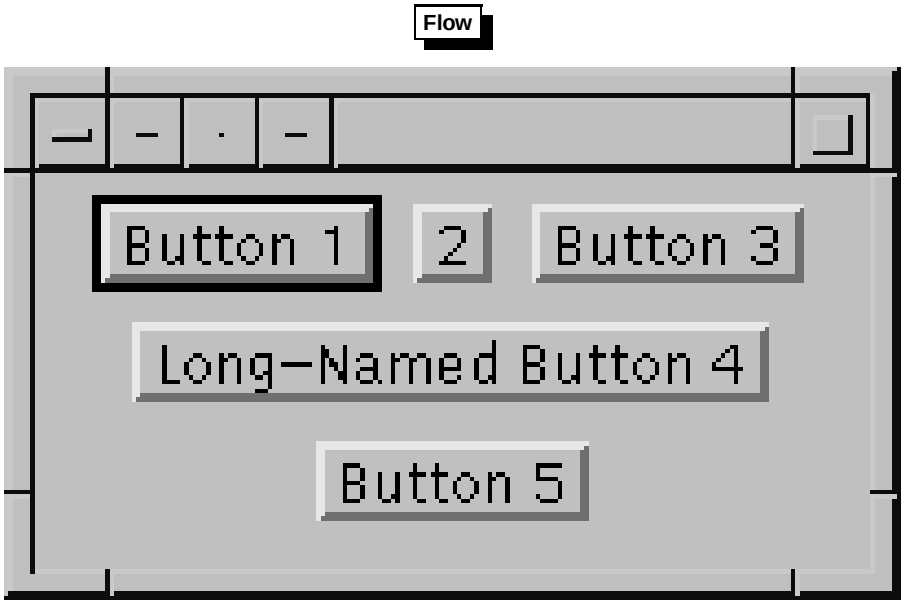
Slide 204

A `BorderLayout` divides a container into 5 zones as shown in the previous slide. It is possible to instantiate a component (which can be a container) in each zone. Hereafter the code of the example:

examples/Border.java

```
import java.awt.*;

public class Border {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new BorderLayout());

    f.add("North", new Button("North"));
    f.add("South", new Button("South"));
    f.add("East", new Button("East"));
    f.add("West", new Button("West"));
    f.add("Center", new Button("Center"));

    f.pack();
    f.setSize(300,300);
    f.show();
  }
}
```

Slide 205



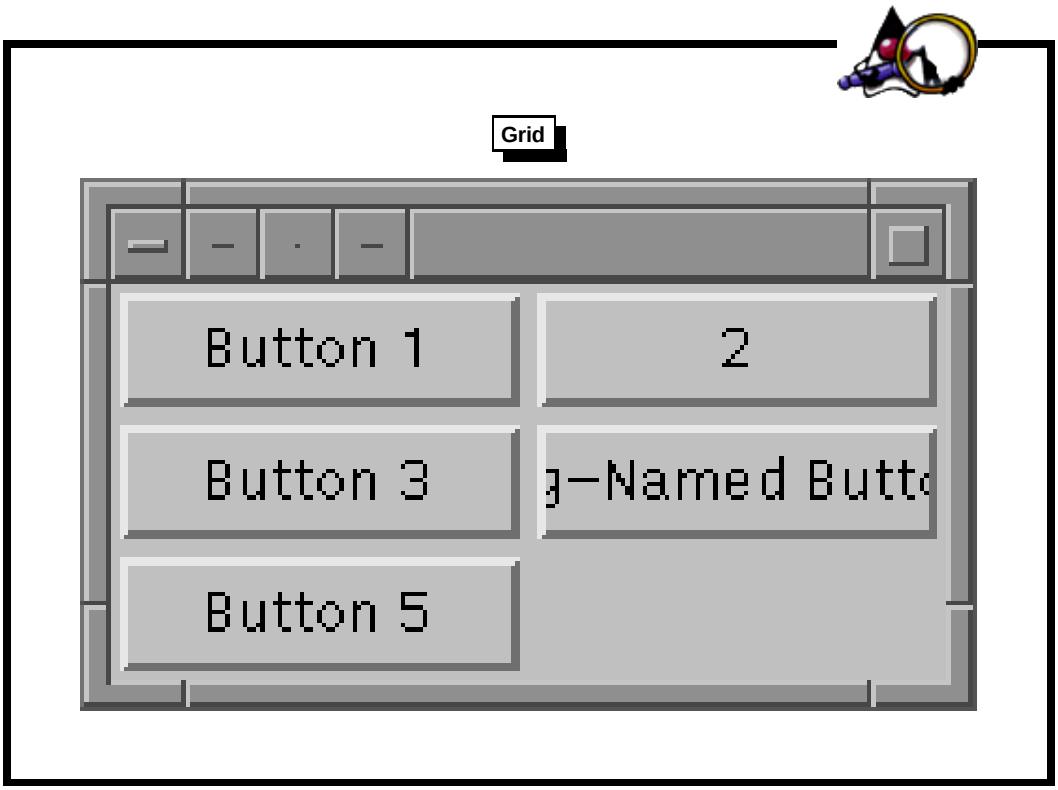
Slide 206

Un flow layout remplit place les composants d'un conteneur les uns à coté des autres en partant de la gauche. Si la place vient à manquer, il passe à la ligne. Le nombre de lignes dépends du nombre d'éléments, leur taille, la taille du container pour lequel le layout travaille. Voila le code :

examples/Flow.java

```
import java.awt.*;

public class Flow {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new FlowLayout());

    f.add(new Button("Button 1"));
    f.add(new Button("2"));
    f.add(new Button("Button 3"));
    f.add(new Button("Long-Named Button 4"));
    f.add(new Button("Button 5"));


    f.pack();
    f.setSize(200,120);
    f.show();
  }
}
```

Slide 207



Slide 208

Un `GridLayout` place les composants d'un conteneur selon un matrice de cellules de **tailles égales**. Le nombre de lignes et de colonnes est fixé lors de l'appel au constructeur de cet algorithme de placement. Voulez-vous le code ?

À la création un nombre de lignes, resp. colonnes à zéro indique que le `GridLayout` doit déterminer ce nombre de lignes, resp. colonnes en fonction du nombre d'éléments gérés et du nombre de colonnes, resp. lignes demandé.

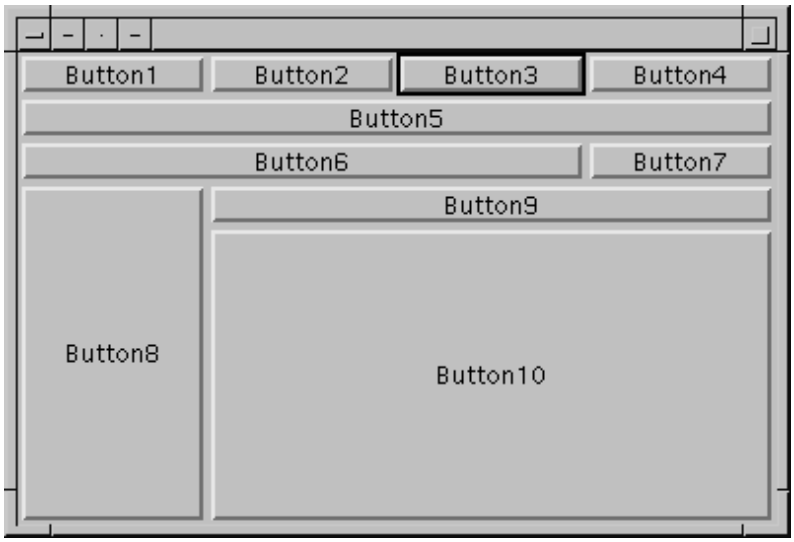
Le source de l'exemple suit :

examples/Grid.java

```
import java.awt.*;
public class Grid {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new GridLayout(0,2));                    // 0 means "any" number of lines
    f.add(new Button("Button 1"));
    f.add(new Button("2"));
    f.add(new Button("Button 3"));
    f.add(new Button("Long-Named Button 4"));
    f.add(new Button("Button 5"));
    f.pack();
    f.setSize(200,120);
    f.show();
  }
}
```

Slide 209

GridBag



Slide 210

Le `GridBagLayout` est l'algorithme de placement le plus sophistiqué de l'AWT. Il permet de placer les composants sur une matrice selon des contraintes qui leurs sont associées. Il est proche des *Forms* Motif. Bien utilisé, il fournit un moyen efficace pour composer des fenêtres ayant un comportement élégant devant les changements de tailles (d'elle même, de ses éléments).

Les constraintes de placement relatives à un composant (voir plusieurs) sont spécifiées dans un objet de classe `GridBagConstraints`. Un objet `GridBagConstraints` doit donc être instancié et ses variables d'instances déterminées avant d'être associé à un composant d'un conteneur contrôlé par un `GridBagLayout`. Cette association se fait via la méthode `setConstraints(Component, GridBagConstraints)` de la classe `GridBagLayout`. Il faut noter qu'un `GridBagLayout` conserve une **copie** de ces contraintes. L'utilisateur peut très réutiliser et modifier un `GridBagConstraints` pour une autre association. Les variables d'instance d'une contrainte permettent de spécifier :

gridwidth, gridheight Le nombre de cellules horizontales et verticales que le composant utilise. `GridBagConstraint.REMAINDER` est une constante permettant de spécifier que le composant est le dernier de sa ligne ou de sa colonne. C'est une manière de déterminer le nombre de ligne ou de colonnes. `GridBagConstraints.RELATIVE` spécifie que le composant précède le dernier de sa ligne ou de sa colonne.

fill Spécifie comment le composant remplit sa cellule. (HORIZONTAL, VERTIVAL, BOTH)

weightx weighty distribue l'espace disponible du container entre les cellules (important pour le resize). 0 tout reste collé au centre du conteneur. 1 les composants prennent tout l'espace disponible.

Il y en a encore qq'une mais bon, à vous d'aller voir l'API ...

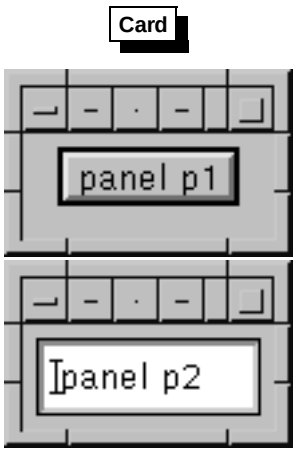
Par défaut, les composants s'aligne de la gauche vers la droite sur une ligne. la contrainte REMAINDER détermine le retour à la ligne et donc le nombre de colonne.


```
import java.awt.*;
public class GridBag extends Frame {
  protected void makebutton(String name, GridBagConstraints c) {
    Button button = new Button(name);
    ((GridBagLayout)this.getLayout()).setConstraints(button, c);
    this.add(button);
  }
  public GridBag() {
    setLayout(new GridBagLayout());
    GridBagConstraints c = new GridBagConstraints();
    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    makebutton("Button1", c);
    makebutton("Button2", c);
    makebutton("Button3", c);
    c.gridwidth = GridBagConstraints.REMAINDER;                    //end of row
    makebutton("Button4", c);
    c.weightx = 0.0;                                               //reset to the default
    makebutton("Button5", c);                                      //another row
    c.gridwidth = 1;                                               //reset to the default
    //      c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
    makebutton("Button6", c);
```

Slide 211

```
    c.gridwidth = GridBagConstraints.REMAINDER;                    //end of row
    makebutton("Button7", c);
    c.gridwidth = 1;                                               //reset to the default
    c.gridheight = 2;
    c.weighty = 1.0;
    makebutton("Button8", c);
    c.weighty = 0.0;                                               //reset to the default
    c.gridwidth = GridBagConstraints.REMAINDER;                    //end of row
    c.gridheight = 1;                                              //reset to the default
    makebutton("Button9", c);
    makebutton("Button10", c);
  }
  static public void main(String args[]) {
    GridBag f = new GridBag();
    f.pack();
    f.show();
  }
}
```

Slide 212



Slide 213

Le “Card Layout” est assez particulier. Il permet d’avoir plusieurs feuilles de composants (des panels dans notre cas) et de les intervertir dans la fenêtre principale. Il vaudrait mieux parler de remplacement que de placement mais bon ...

C’est largement utilisé pour faire des écrans de choix d’option avec des onglets. Allez pouf du code pour les sceptiques :

/Card.java

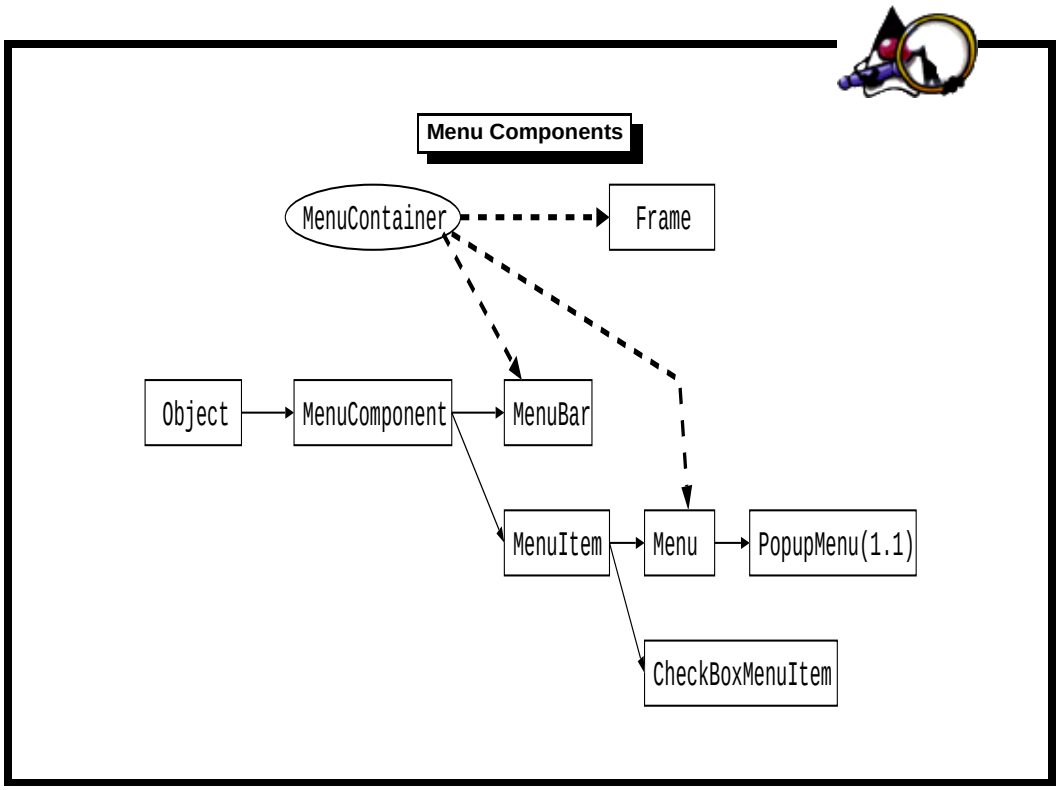
```
import java.awt.*;

public class Card extends Frame {
  static public void main(String args[]) {
    Card f = new Card();
    f.setLayout(new CardLayout());
    Panel p1 =new Panel();
    p1.add(new Button("panel p1"));
    Panel p2 =new Panel();
    p2.add(new TextField("panel p2"));
    f.add("panel1",p1);
    f.add("panel2",p2);
    f.setSize(100,100);
    f.pack();
    f.show();
  }
  public boolean keyDown(Event evt, int key) {
    ((CardLayout)this.getLayout()).show(this,"panel"+(char)key);
    return true;
  }
}
```

Slide 214

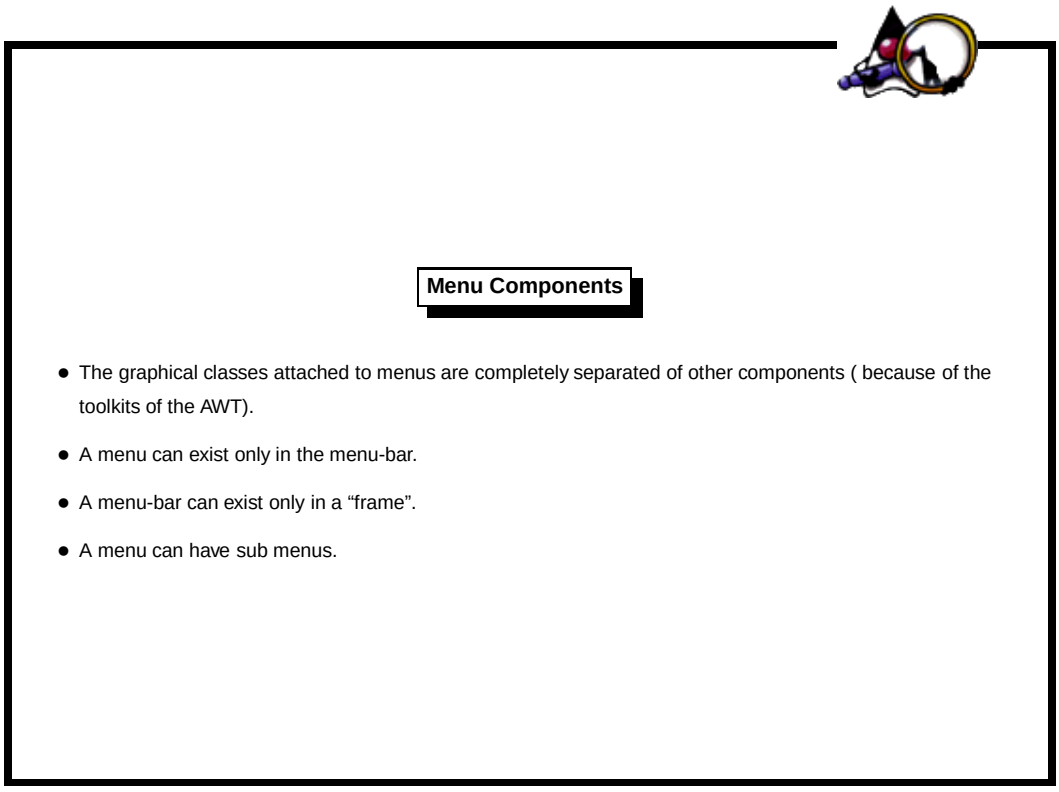
Dans cet exemple l’appui de la touche “1” met en place dans la fenêtre principale le panel p1 (donc physiquement on voit un bouton). Appuyer sur la touche “2”, affiche le champ de saisie.

La méthode `show` rendant visible un élément d’un conteneur géré par un `CardLayout` est propre à ce layout, d’où le cast dans la méthode `keyDown`.

4.2.3 Managing Menus



Slide 215



Slide 216

Dans la figure les flèches en pointillé sont des liens d'implémentation d'interface, celles en trait plein des liens d'héritage.

Quand même, il faut remarquer que le design de l'AWT est fonction des capacités des toolkits des systèmes hôtes. On peut (encore) noter l'approche très pragmatique des concepteurs. On retrouve une démarche de factorisation des éléments que l'on peut trouver sur toutes les architectures.

Ceci explique un peu cela : dans la version 1.0, vraiment un noyau d'élements graphiques avec un modèle d'événements très primitif, à chaque release, intégration de nouveaux éléments ...

Bon sinon, trève de plaisanteries, pourquoi "Frame" implémente l'interface MenuContainer ??

Et les Choices alors ... PM: RAF.

Menus

examples/SimpleMenuDemo.java

```
import java.awt.*;
import java.awt.Menu;
public class SimpleMenuDemo {
  static public void main(String args[]) {
    Frame f = new Frame();
    MenuBar mb=new MenuBar();
    f.setMenuBar(mb);
    Menu m1 = new Menu("File",true);
    m1.add(new MenuItem("Open ..."));
    m1.addSeparator();
    m1.add(new CheckboxMenuItem("Yo"));
    m1.add(new MenuItem("Quit"));
    mb.add(m1);
    Menu m2 = new Menu("Help",false);
    m2.add(new MenuItem("Help ..."));
    mb.add(m2);
    mb.setHelpMenu(m2);
    f.pack();
    f.setSize(300,300);
    f.show(); }}
```

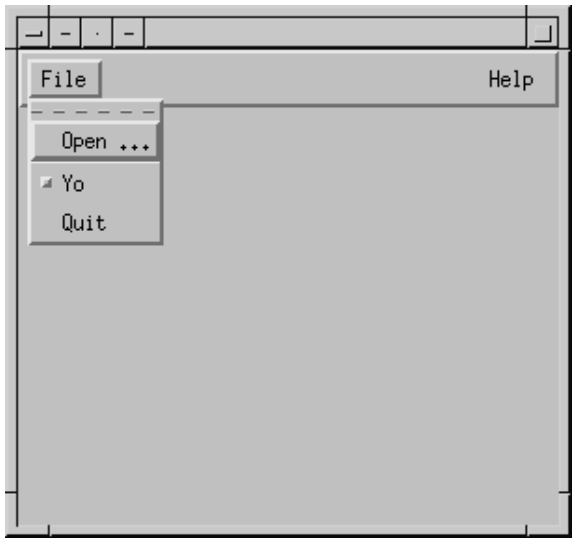
Slide 217

Bon, pour ceux qui se sont essayés à définir des menus en Motif de base, c'est quand même un vrai bonheur. Ça devient presque agréable de mettre en place une barre de menus.

M'enfin cette simplicité de mise en place peut être rapprochée des choix assez restrictifs qui ont été fait au départ.

- Et comment je fais un popup menu avec ça ?
- Est-ce paramétrable via des ressources ?
- Comment je mets des bitmaps dans ces menus ?

A Menu



Slide 218

Menus

- Until the version 1.1 only the pull-down menus are supported ...
- Since the jdk 1.1
  - Keyboard Shortcuts + popup-menu.
  - The popup-menus are new class ...

Slide 219

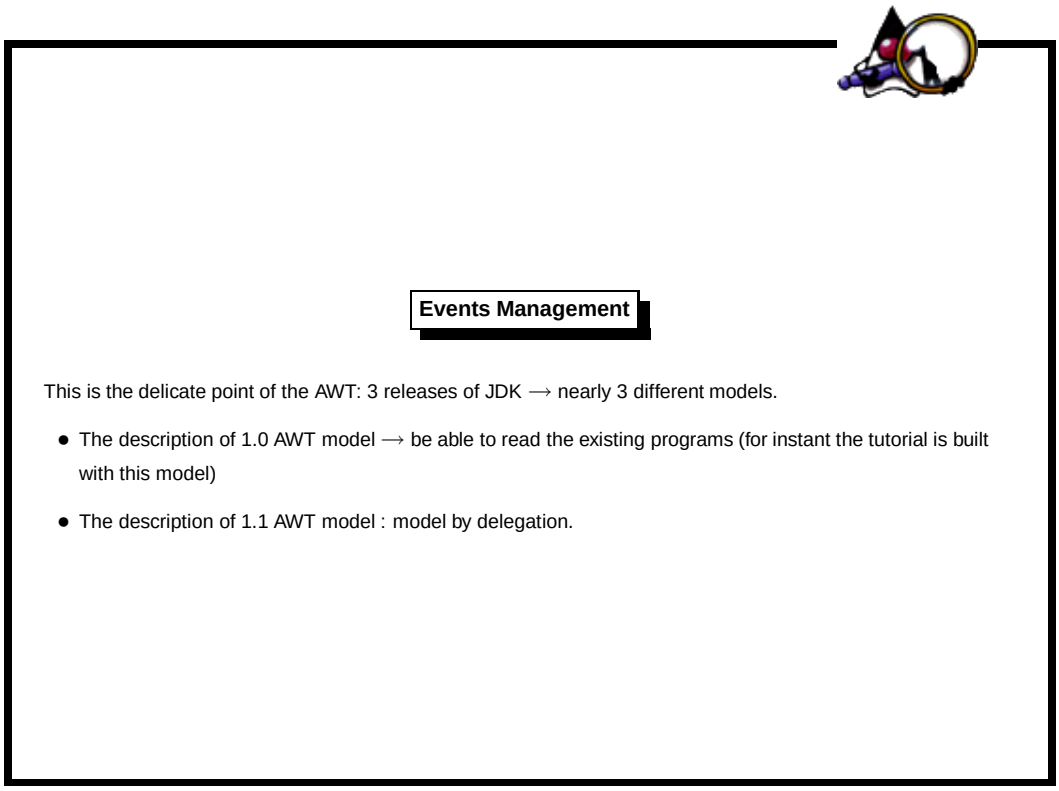
Il peut y avoir quelques contreverses sur la notion de *pop up menu*... Un popup est un menu qui apparait **là où est le pointeur** de la souris quand un bouton de celle-ci est pressé. Il ne faut pas les confondre avec des *pull-down* menus dispersés de ci, de là dans une fenêtre : les `Choices` en java qui sont des `Component` utilisable partout (dans un `Container` géré par un `Layout`).

Oui, mais bon, nous (PM+LA) n'avons pas réussi à faire fonctionner les raccourcis claviers. Par contre, pour les popup-menus voila un exemple QUI MARCHE avec le JDK1.1 (pas comme dans la doc) :

examples/PopupDemo.java

```
import java.awt.*;
import java.awt.Menu;
import java.awt.event.*;

class Mypopup extends MouseAdapter {
  PopupDemo f;
  PopupMenu m1;
  public Mypopup(PopupDemo PopupDemop) {
    f=PopupDemop;
    m1 = new PopupMenu("File");
    m1.add(new MenuItem("Open ..."));
    m1.addSeparator();
    m1.add(new CheckboxMenuItem("Yo"));
    m1.add(new MenuItem("Quit"));
    f.add(m1);
  }
  public void mousePressed(MouseEvent e) {
    m1.show(f,e.getX(),e.getY());
  }
}
public class PopupDemo extends Frame {
  public PopupDemo() {
    Mypopup p=new Mypopup(this);
    this.addMouseListener(p);
    setSize(200,200);
    show();
  }
  static public void main(String args[]) {
    PopupDemo f = new PopupDemo();
  }
}
```

4.3 Events Management



Slide 220

Nous avons décider de présenter les deux modèles : le premier pour pouvoir comprendre ce qui s'est fait, le deuxième pour utiliser la version actuelle du jdk.

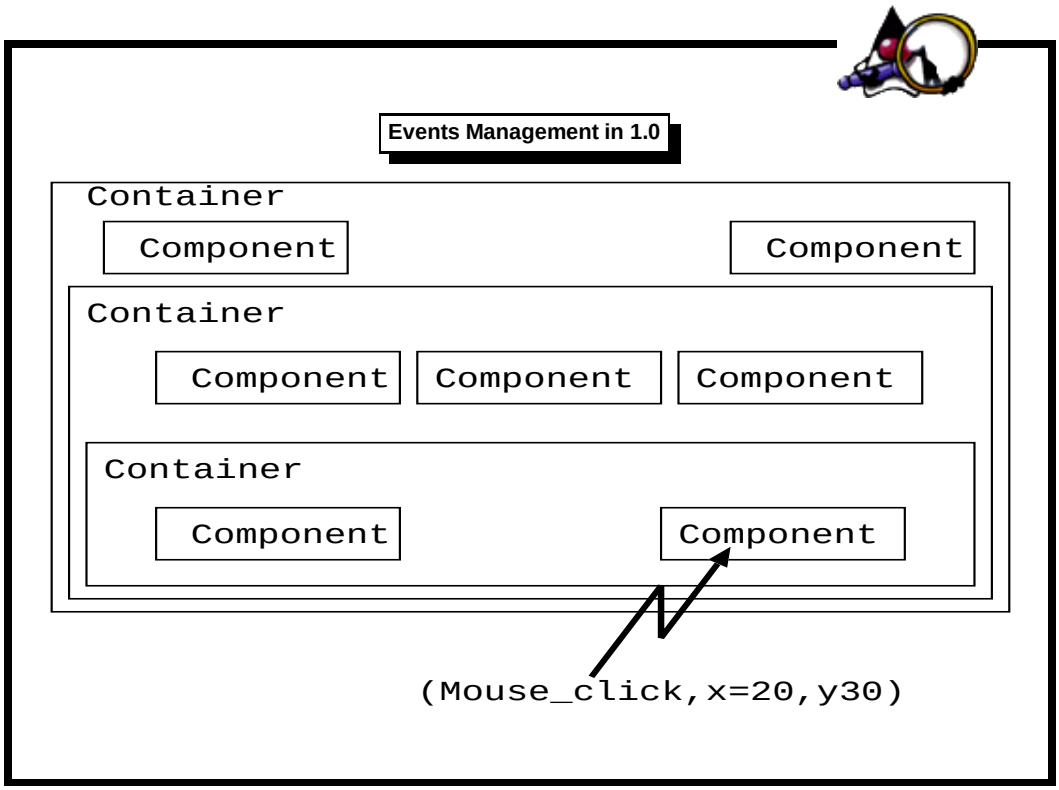
4.3.1 Events Management in 1.0

Events Management in (1.0)

- The events are detected by the host system and then are translated into java events (independently of the architecture).
- There is thus an unique class “event” instantiated at the arriving of the events from the host system.
- A field of the object event (id) is used to identify the type of the event : `MOUSE_DRAG`, `WINDOW_MOVED`, `KEY_PRESS` ...
- Other fields store annexes informations, for example, (x,y) for a mouse event, key (code touch) for the event `KEY_PRESS`...
- A field is reserved to store an arbitrary object but filled by the event transmitter.

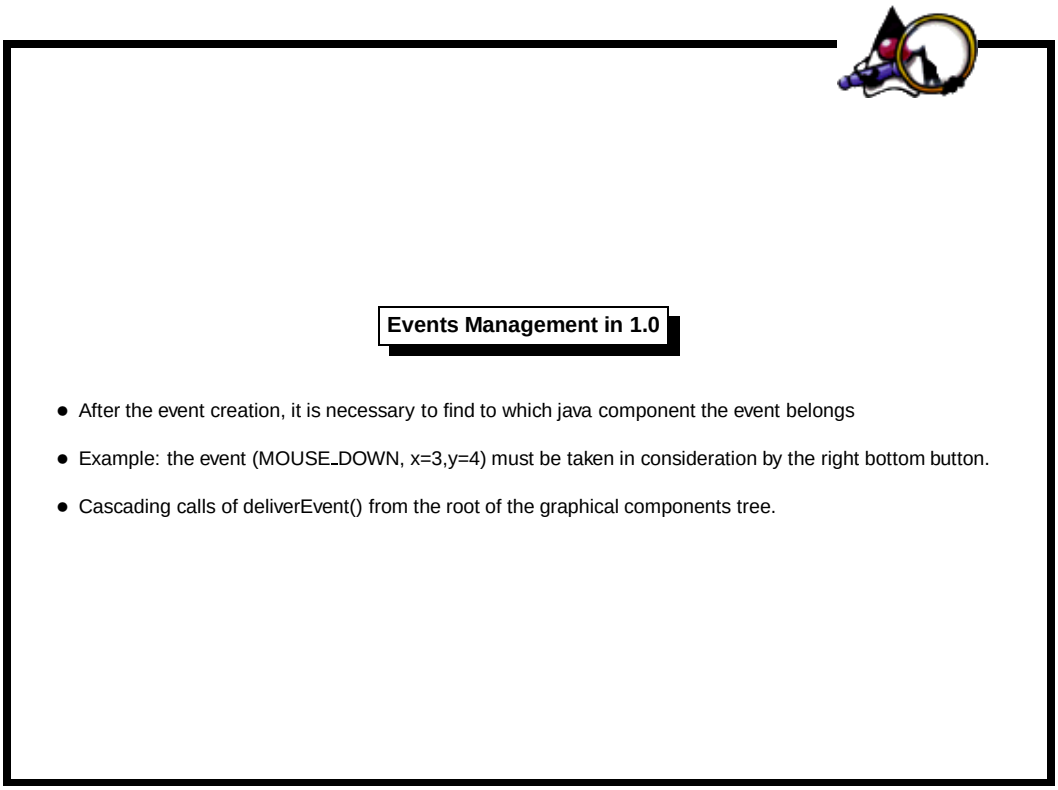
Slide 221

Sur ce dernier point, un bouton va mettre dans cet objet arbitraire le texte qu'il contient, de même pour un champ de saisie. Un bouton d'état va quand à lui remplir cet objet avec un booléen indiquant son état courant ...



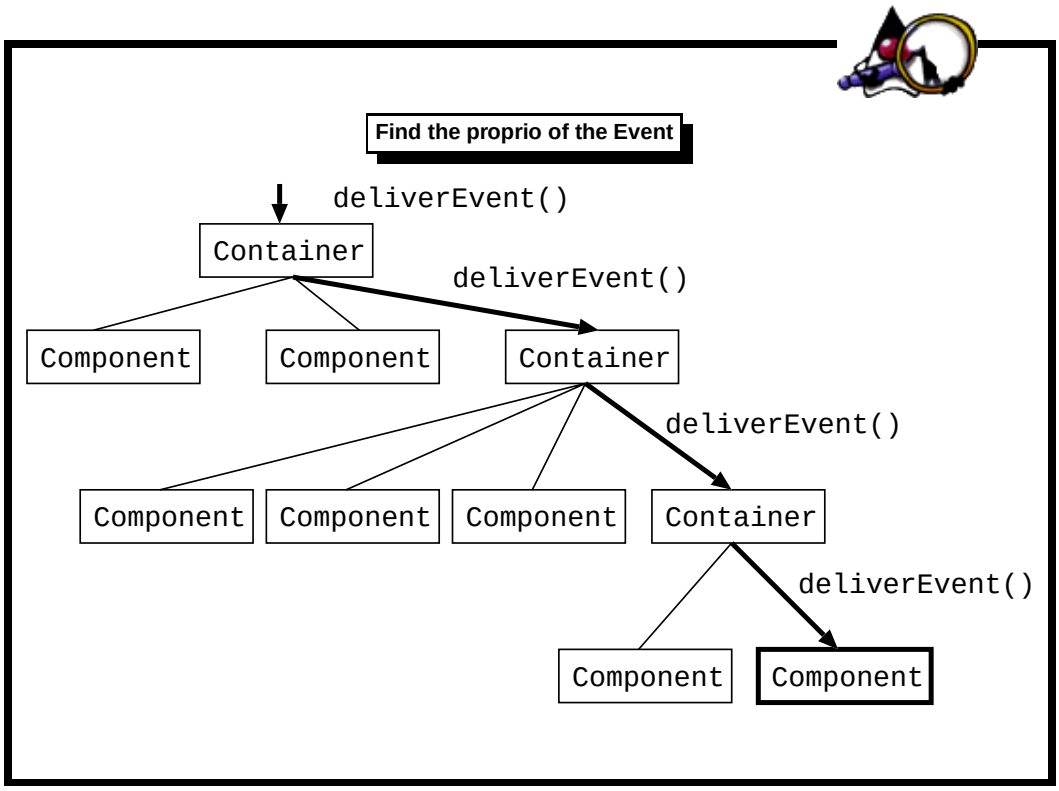
Slide 222

Nous sommes bien d'accord. Si le "component" en question est représenté par un `XmPushButton` sur la toolkit Motif hôte, alors le callback associé au `XmPushButton` va appeler (depuis C), la méthode de création d'un évènement java et mettre à jour l'objet ainsi créé avec les renseignements associés (x,y). Tout ceci est la tâche des *perr objects*.

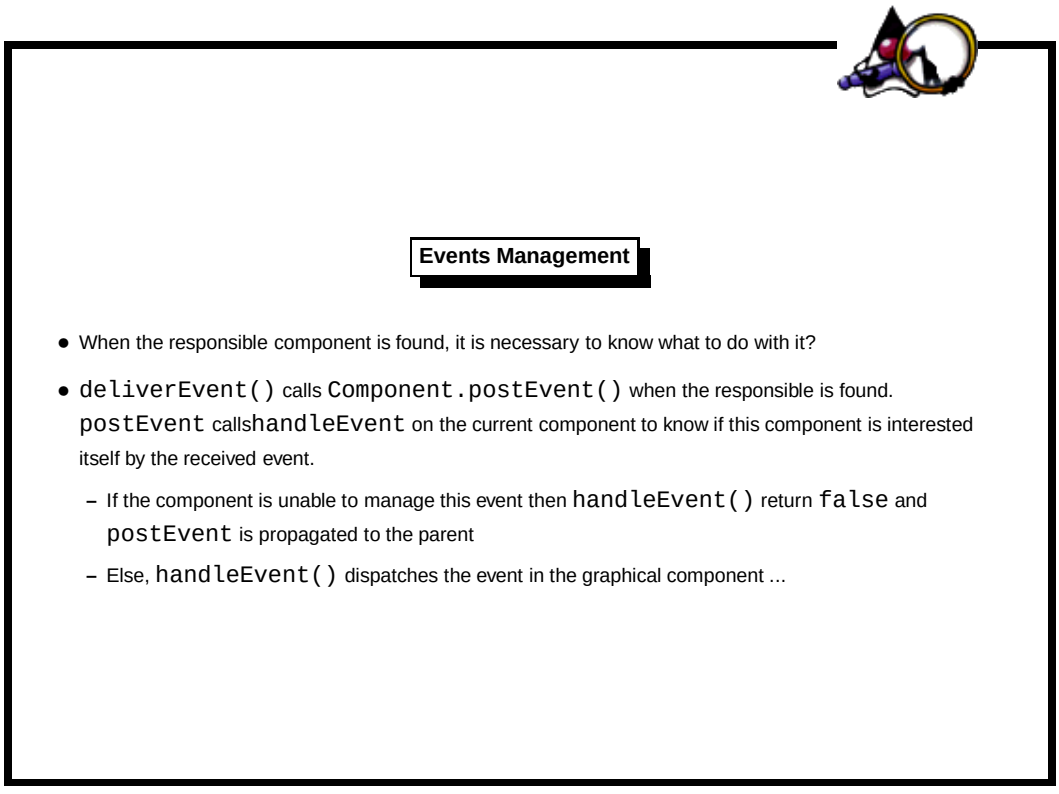


Slide 223

`deliverEvent()` est une méthode définie dans `Component` et dans `Container` (celle de `Container` *override* celle de `Component`). La différence entre ces deux méthodes est que celle de de `Container` effectue les translations nécessaires sur (x,y) avant de transmettre aux sous-composant.

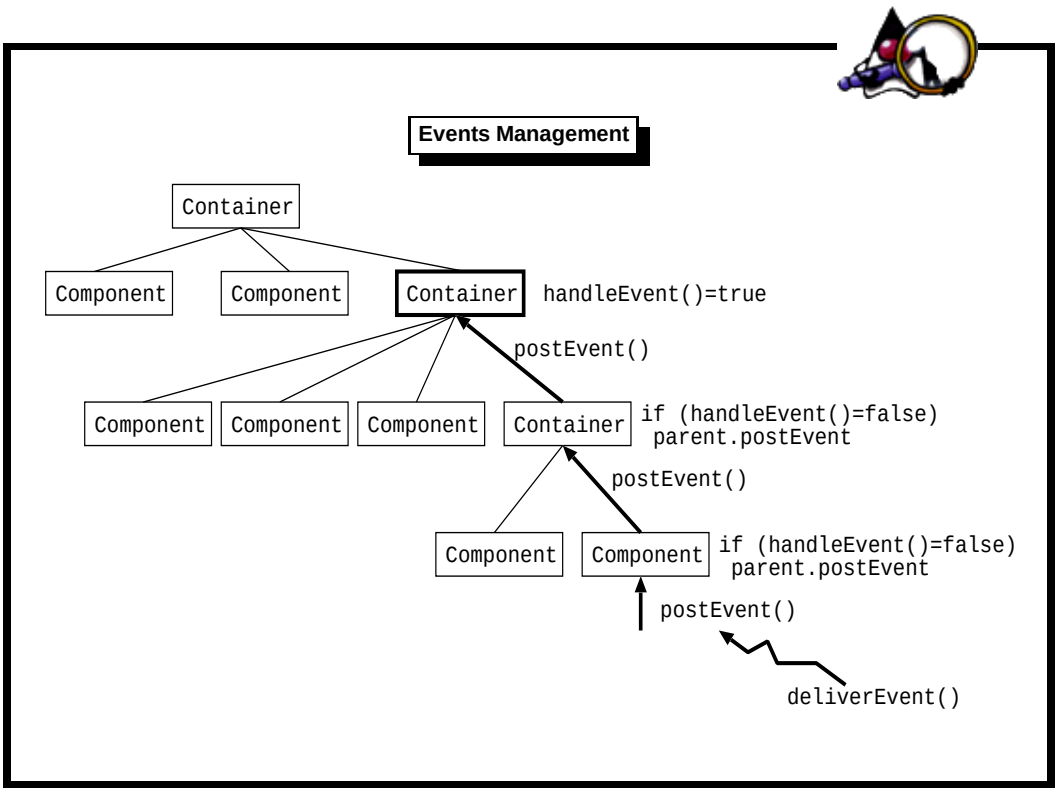


Slide 224

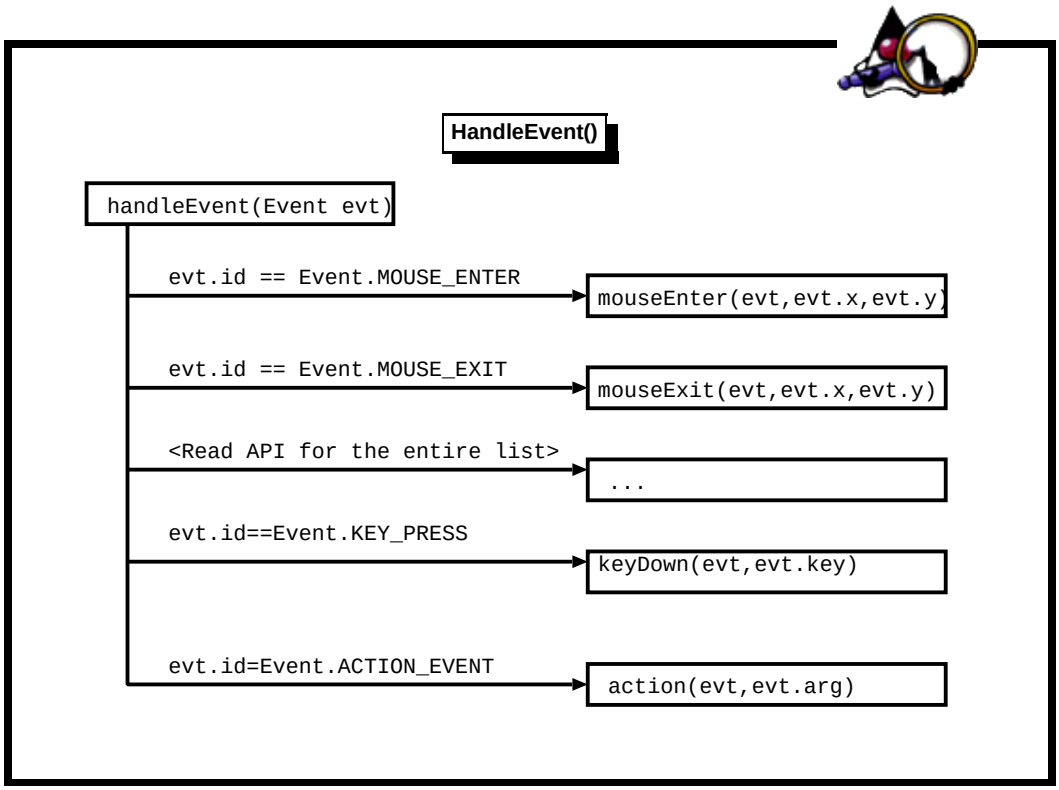


Slide 225

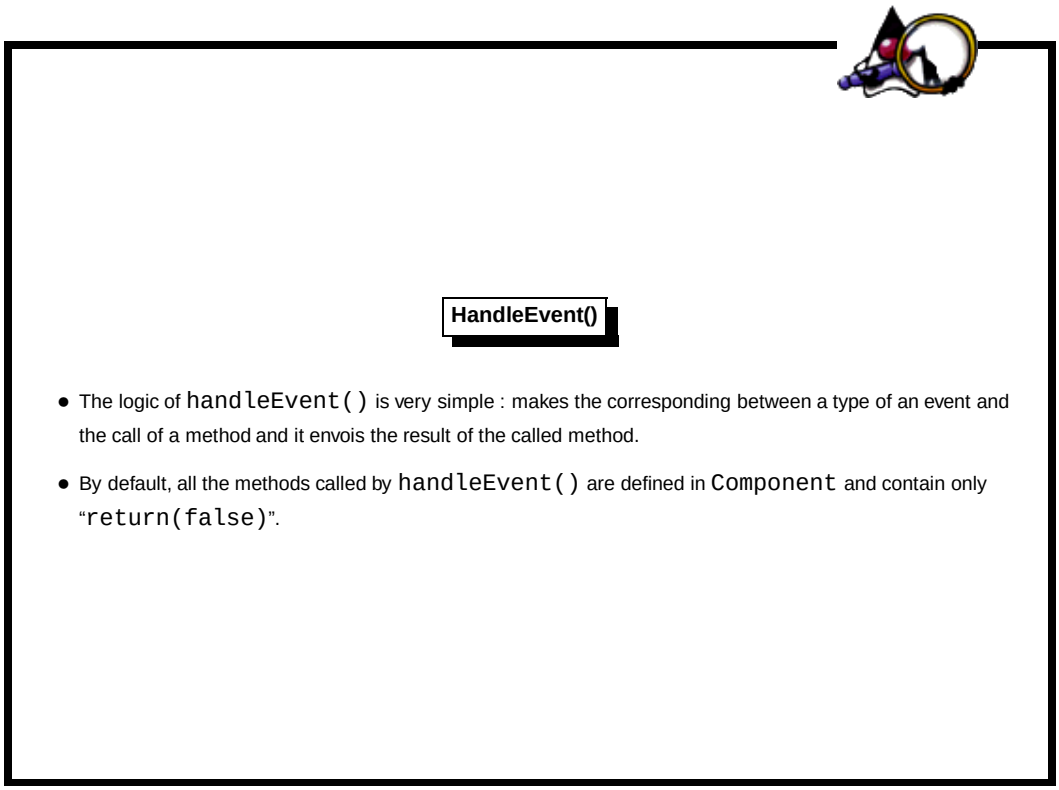
Le champ `event.target` est mis à jour avec une référence sur l'objet qui prend en charge l'événement. Attention! `MenuComponent` définie sa propre méthode `postEvent()`.



Slide 226



Slide 227



Slide 228

Dans les bouquins, ils disent que handleEvent() dispatche l'évènement au sein de l'objet. Et je le prouve voila un fragment de la classe Component (JDK-1.0.2) :

```
public boolean mouseEnter(Event evt, int x, int y) {
 return false;
}
public boolean handleEvent(Event evt) {
        switch (evt.id) {
          case Event.MOUSE_ENTER:
            return mouseEnter(evt, evt.x, evt.y);
          case Event.MOUSE_EXIT:
            return mouseExit(evt, evt.x, evt.y);
          case Event.MOUSE_MOVE:
            return mouseMove(evt, evt.x, evt.y);
          case Event.MOUSE_DOWN:
            return mouseDown(evt, evt.x, evt.y);
          case Event.MOUSE_DRAG:
            return mouseDrag(evt, evt.x, evt.y);
          case Event.MOUSE_UP:
            return mouseUp(evt, evt.x, evt.y);
          case Event.KEY_PRESS:
          case Event.KEY_ACTION:
            return keyDown(evt, evt.key);
          case Event.KEY_RELEASE:
          case Event.KEY_ACTION_RELEASE:
            return keyUp(evt, evt.key);
          case Event.ACTION_EVENT:
            return action(evt, evt.arg);
          case Event.GOT_FOCUS:
            return gotFocus(evt, evt.arg);
          case Event.LOST_FOCUS:
            return lostFocus(evt, evt.arg);
        }
        return false;
    }
```

Action Events

- The events attached to the activation of a graphical component of “high level” (button ,check-box) generate a particulate type of events : The “*Action Event*”.
- The method called by the `handleEvent()` has the following profile:

  public boolean action(Event, Object arg)

Slide 229

Action Methods

Component	Class of arg !	Contain of (or)arg !
Checkbox	Boolean	Stateof the checkbox
Choice	String	String choosed in the “Choice”
List	String	String choosed in the list
TextField	String	String type in the textfield

Slide 230

Oui, d'accord, ok ok, mais comment ça s'utilise au bout du compte ??

But how we do ??

- The first method: subclass the graphical components and *override* a or several of the following methods :

  mouseEnter(Event evt,int x,int y);

  mouseExit(Event evt,int x,int y);

  mouseMove(Event evt,int x,int y);

  mouseDown(Event evt,int x,int y);

  mouseDrag(Event evt,int x,int y);

  mouseUp(Event evt,int x,int y);

  keyDown(Event evt,int key);

  keyUp(Event evt,int evt.key);

  action(Event evt,Object arg);

  gotFocus(Event evt, Object arg);

  lostFocus(Event evt,Object arg);

Slide 231

First Method: SubClassing...

examples/EvDemo.java

```
import java.awt.*;
class MyButton extends Button {
  String Yo;
  public MyButton(String stringp) {
    super(stringp);
    Yo=stringp;
  }
  public boolean action(Event e, Object argp) {
    System.out.println(Yo);
    return(true);                    // to handleEvent() !!
  }
  public boolean gotFocus(Event e,Object argp) {
    System.out.println("Mouse trapped !");
    return(true);
  }
}
class MyField extends TextField {
  public MyField(int size) {
    super(size);
  }
  public boolean action(Event e,Object argp) {
    System.out.println(argp);
    return(true);                    // to handleEvent();
  }
}
public class EvDemo {
  static public void main(String args[]) {
    Frame f = new Frame();
    f.setLayout(new FlowLayout(FlowLayout.LEFT));
    f.add(new MyButton("Yo!"));
    f.add(new MyField(20));
    f.resize(200,200);
    f.show();
  }
}
```

Slide 232

The Second Method

- The events which are not consumed by the basic components are passed into the “containers” ⇒
  - they are intercepted to the level of the “container”
  - Do the treatment for all the sub-components..

Slide 233

Central Treatment

examples/EvDemo2.java

```
import java.awt.*;


class MyFrame extends Frame {
  public MyFrame() {
    super();
    setLayout(new FlowLayout(FlowLayout.LEFT));
    add(new Button("Yo!"));
    add(new TextField(20));

    resize(200,200);
    show();
  }

  public boolean gotFocus(Event e,Object argp) {
    if (e.target instanceof Button) {
      System.out.println("Mouse Trapped");
    }
    return(true);
  }

  public boolean action(Event e,Object argp) {
    if (e.target instanceof Button) {
      System.out.println("Yo!");
      return true;
    }
    if (e.target instanceof TextField) {
      System.out.println(argp);
      return true;
    }
    System.out.println("ohoh ?");
    return true;
  }
}

public class EvDemo2 {
  static public void main(String args[]) {
    MyFrame f = new MyFrame();
  }
}
```

Slide 234

Problems

- Sub-Classing generated so many classes for a complex graphical interface.
- No separation between application code et UI code
- Events management is very error prone (hendleEvent return true or false ..)
- Not very efficient PostEvent/HandleEvent
- Data passed with events are very poor: String

Slide 235

Il faudrait expliquer le coup de la chaine en parametre du action event.

4.3.2 The Method by Delegation (1.1)

The Method by Delegation

Objectives :

- Resolve the previous problems.
- Reorganisation to support more complex UI.

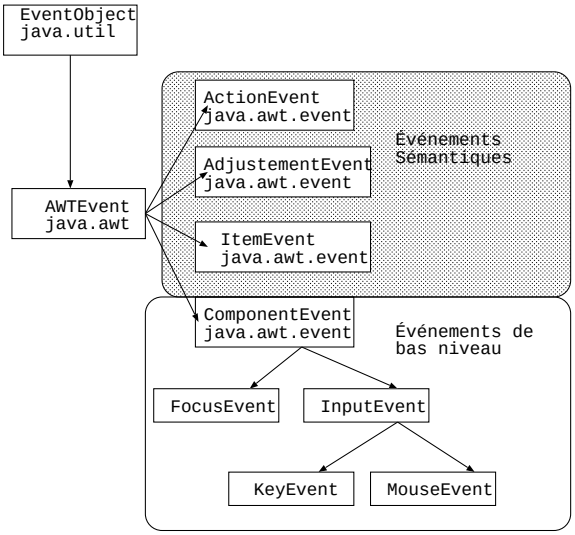
Conception objectives:

- Simpler and easier to understand (thank for us !)
- A clear separation between application and UI code (yes more..)
- More robust events management (compile-time checking) (*parental agreement*)
- Support for interface builders (run time discovery) (*censured*)

Slide 236

Model by Delegation

No more a class Event, but a hierarchy of classes !



Slide 237

Delegation Model

Low level events :

- `ComponentEvent` : *resized, moved ...*
- `ContainerEvent` : *added, removed*
- `FocusEvent` : *focus gained, focus lost*
- `KeyEvent` : *key-pressed, key-released*
- `MouseEvent` : *mouse-down, mouse-move ...*
- `WindowEvent` : *activated, closed ...*

Slide 238

Model by Delegation

The semantic of the events :

- `ActionEvent` : do a command
- `AdjustementEvent` : a value has been adjusted
- `ItemEvent` : the state of an item is modified
- `TextEvent` : the state of the text catch zone (`TextComponent, TextArea`) is modified..

Slide 239

- Les `ContainerEvent` reflètent l'entrée ou la sortie d'un élément graphique dans un `Container` donné.
- Pour les `ItemEvent` la notion d'item est très large : elle correspond à tout élément sélectionable au sein d'un élément graphique groupant des items. Citons les groupes de `CheckBox`, les listes... En fait
- `ActionEvent` correspond donc à un événement de haut niveau "demande d'action". Plusieurs composant graphiques peuvent en générer : les `Button`, mais aussi les listes (return, double click), le choix d'un item dans un pull-down menu, ou un pop-up (Une action dans un `Choice` est vue comme une **sélection** d'un item) ...
- Les différentes possibilités seront détaillées sur le transparent 244, composant graphique par composant graphique.

LA : Bon il y aussi le InputEvent. Un peu spé celui-la ...

The Events Management

- The user can eventually adds his owns events type in this model.
- an events queue managed by a thread ensures the distribution of the events among the objects (asynchronous delivery of events...).
- All the generated events are placed in this queue.

`java.awt.EventQueue` :

```
public static EventQueue getEventQueue()
public synchronized void postEvent(AWTEvent e)
public synchronized AWTEvent getNextEvent()
public synchronized AWTEvent peekEvent()
public synchronized AWTEvent peekEvent(int eventID)
```

Slide 240

Pas de panique on n'aura pas à gérer la file d'événements. Il suffit juste de savoir qu'elle existe.

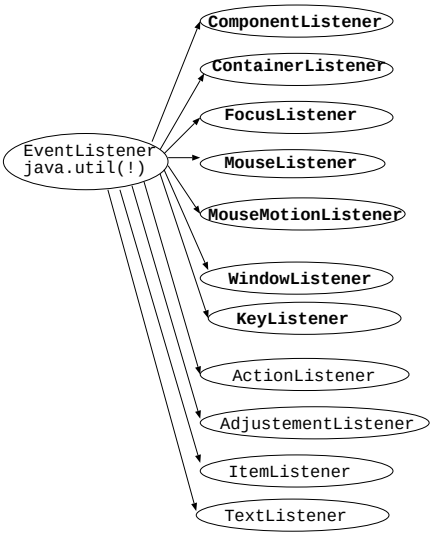
Cette file d'événements est donc bien un objet prédéfini et géré par l'AWT.

Model by Delegation : Idea

- An event is propagated from a SOURCE (Event Source) into one or many destinations (*Event Listeners*).
- A destination (*listener*) is an object which implements one or many interfaces of the hierarchy of the interface `EventListener`. A priori, this is an object of the application.
- A source is an object capable to do `postEvent()`. Thus normally, this is an object of the graphical interface.
- The graphical objects design explicitely the objects to which the events are propagated which are generated via new methods of the class `Component` and sub-classes...

Slide 241

Listeners



Slide 242

Tout cela n'est qu'interfaces ! Les flèches du schéma précédent vont de la super interface vers une sous interface. Les interfaces relatives à des événements de bas niveau sont en gras. Il faut remarquer que `EventListener` lui même ne fait pas partie de `java.awt` mais de `java.util` !! Toutes ces interfaces relatives à des événements AWT sont dans le package : `java.awt.event`.

Donc en fait les éléments de l'application qui sont à l'écoute d'événements se produisant dans l'interface graphique doivent implémenter une ou plusieurs de ces interfaces.

Listeners

- A `Listener` does not consume events (read-only for the listener)..
- A class can decide to implement one or many of these listeners. it is necessary to implement all the methods defined in these interfaces. Example :

```
public interface KeyListener extends EventListener {
        public void keyTyped(KeyEvent e);
        public void keyPressed(KeyEvent e);
        public void keyReleased(KeyEvent e);
}
```

Slide 243

Attention : les événements passés en paramètres des méthodes d'une interface *listener* sont des sous classes de `java.awt.AWTEvent` (elle même sous classe de `java.util.EventObject`) et **non pas** de `java.awt.Event` comme l'étaient les paramètres des méthodes utilisées dans le premier modèle d'événement (`deliverEvent, posEvent...`).

Dans le modèle par délégation on dispose de la méthode `getSource()` de `java.awt.AWTEvent` pour connaître l'élément graphique qui a généré l'événement. Dans l'ancien on utilisait le champ `target` de `java.awt.Event`. Le nom était d'ailleurs plutôt ambigu, mais logique puisse que l'objet graphique en question était bien la cible atteinte lors de la phase de `deliverEvent()`.

Vérifier toutes ces histoires de classes d'événements et de target avec Pascal.

Sources

A low level destination :

Component	addComponentListener(ComponentListener l) addFocusListener(FocusListener l) addKeyListener(KeyListener l) addMouseMotionListener(MouseMotionListener l)
Container	addContainerListener(ContainerListener l)
Dialog	addWindowListener(WindowListener l)
Frame	addWindowListener(WindowListener l)

Slide 244

Add destinations for semantic events :

Button	addActionListener(ActionListener l)
Choice	addItemListener(ItemListener l)
Checkbox	addItemListener(ItemListener l)
CheckboxMenuItem	addItemListener(ItemListener l)
List	addActionListener(ActionListener l) addItemListener(ItemListener l)
MenuItem	addActionListener(ActionListener l)
Scrollbar	addAdjustmentListener(AdjustmentListener l)
TextComponent	addTextListener(TextListener l)
TextField	addActionListener(ActionListener l)

Slide 245

- Dans les deux tableaux précédents, les composants graphiques de la première colonne disposent des méthodes de la deuxième colonne pour indiquer qu'ils vont propager leurs événements vers l'élément de l'application passé en paramètre.
- Bien entendu une sous classe d'une classe de la colonne de gauche dispose des même ajouts de *listener* que sa super classe. Par exemple `TextArea`, et `TextField` disposent chacune de `addTextListener`.
- Les éléments succeptibles de générer des `ItemEvent` semblent devoir implémenter l'interface `ItemSelectable`. Par exemple `Choice`, `CheckboxMenuItem`, `List` `MenuItem` implémentent cette interface.

Model by Delegation

examples/DelDemo.java

```
import java.awt.*;
import java.awt.event.*;
class SaySomething implements ActionListener,
FocusListener {
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() instanceof Button) {
      System.out.println("That's Button !");
    } // à la place de e.target ...
    System.out.println(e.paramString());
  }
  public void focusGained(FocusEvent e) {
    System.out.println("Hu ?");
  }
  public void focusLost(FocusEvent e) {
    System.out.println("ciao !");
  }
}

class DelDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.setLayout(new FlowLayout(FlowLayout.LEFT));

    SaySomething ss=new SaySomething();

    Button yo=new Button("Yo !");
    yo.addActionListener(ss);

    TextField tf= new TextField(20);
    tf.addActionListener(ss);
    tf.addFocusListener(ss);

    f.add(yo);
    f.add(tf);
    f.resize(200,200);
    f.show();
  }
}
```

Slide 246

Sur cet exemple, on voit bien la séparation entre UI et application. La classe SaySomething représente l'application alors que la classe DelDemo représente la partie UI.

Le bouton et le champ de saisie déclare l'objet `ss` comme destination des événements qu'ils génèrent. Ceci est cohérent (*compile-check*) dans la mesure où `SaySomething` implémente bien les interfaces `ActionListener, FocusListener`.

The classes *Adapter*

One of the problems:

- A class implements an interface *listener* must a priori implement all the methods defined in the interface ⇒ a little painful l sometime ...
- The **classes** *adapter* proposes an implantation by default of all the defined methods of the low level *listeners*.
- The user defines thus its application class as a sub-class of the class *adapter* and surcharges only the method which is interested for him.

Slide 247

Attention les classes *adapters* n'existent que pour les événements de bas niveau (voir le transparent 238).

The Classes *Adapters*

examples/adapter.java

```
import java.awt.*;
import java.awt.event.*;
class JugeDeTouche extends KeyAdapter {
  // pas la peine d'implémenter keyPressed ou keyReleased -> fournit par KeyAdapter !
  public void keyTyped(KeyEvent e) { System.out.println(e.getKeyChar()); }
}
public class adapter {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.setLayout(new FlowLayout(FlowLayout.LEFT));
    JugeDeTouche jdt=new JugeDeTouche();
    f.addKeyListener(jdt);
    f.resize(200,200);
    f.show();
  }
}
```

Slide 248

New Components : Events Management

- A new graphical component have to define what events it wants to handle.

  protected final void enableEvents(long eventsToEnable)

- then it have to define what it does with events by overriding:

  protected void processEvent(AWTEvent)

  protected void process*EventClass*(EventClass)

Slide 249

New Components

examples/PeventDemo.java

```
import java.awt.*;
import java.awt.event.*;
class SpeechButton extends Button {
  String speech;
  public SpeechButton(String speechp) {
    super(speechp);
    enableEvents(AWTEvent.FOCUS_EVENT_MASK);
    speech =speechp;
  }
  protected void processFocusEvent(FocusEvent e) {
    switch (e.getId()) {
    case FocusEvent.FOCUS_GAINED:
      System.out.println(speech);
      break;
    case FocusEvent.FOCUS_LOST:
      System.out.println("ciao");
    }
    super.processFocusEvent(e);
  }
}


public class PeventDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.setLayout(new FlowLayout(FlowLayout.LEFT));

    SpeechButton yo=new SpeechButton("Yo!");

    f.add(yo);
    f.resize(200,200);
    f.show();
  }
}
```

Slide 250

Advantages of the new model

- Separation between Application/UI
- No use to subclass “components”
- Only listened event are propagated (no more postEvent/HandleEvent cascading calls) ... (Filtering)

Slide 251

LA : Pascal tu pourrais donner un exemple justifiant le dernier point. PM: Oui, je pourrai, tu as raison LA... ;-D

4.4 2D Graphics

2D Graphics

- In brief: A graphical component is an abstract object with a position, a size, able to receive events and THAT CAN BE DRAWED !
- Any component have a `paint()` method and an `update()` method
- There is a screen updater thread for each JVM. This thread travels across components tree, detects what components have to be repainted and call the paint() method of the corresponding component.

Slide 252

2D Graphics

If you want to draw, you have to :

- Extends a Graphical Component (For example Canvas which is dedicated to this job)
- Overrides the `paint(Graphics g)` method
- Every graphical component is associated with an object of class `Graphics`. The graphics class have methods for drawing images, texts, rectangle, lines ...

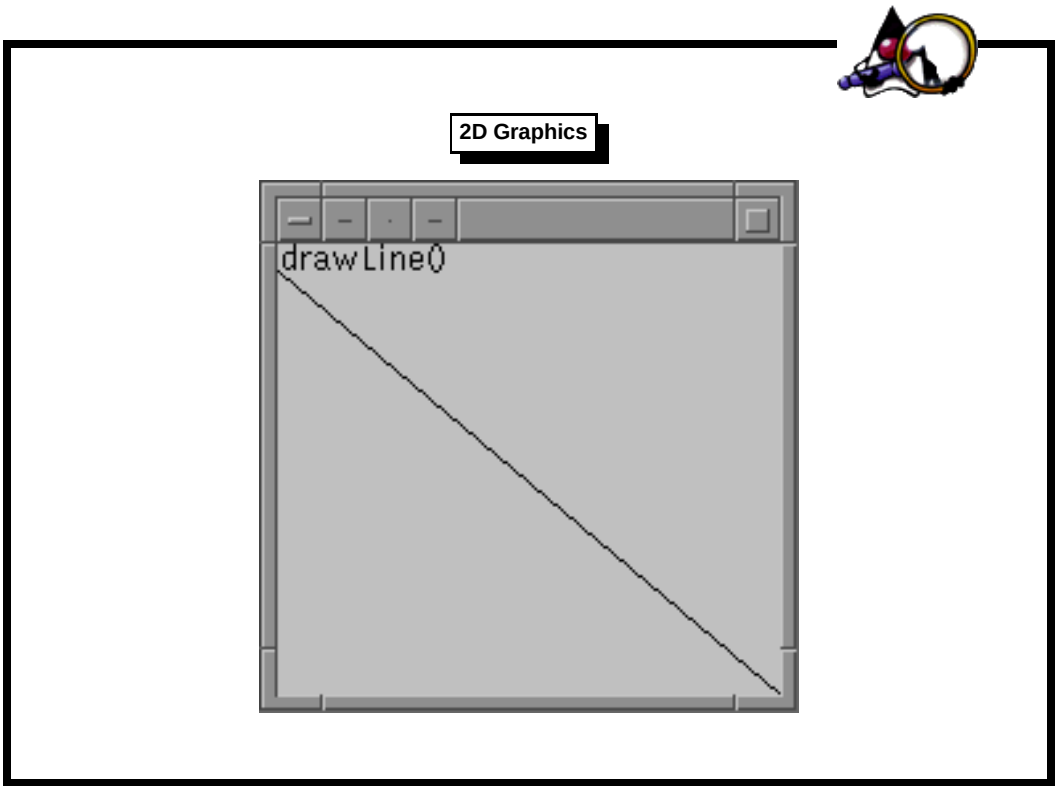
Slide 253

2D Graphics

examples/SimpleGraphicDemo.java

```
import java.awt.*;

class MyCanvas extends Canvas {
  public void paint(Graphics g) {
    Dimension d = size();
    g.drawLine(0, 10, d.width, d.height);                    // x1, y1, x2, y2
    g.drawString("drawLine()", 0, 10);
  }
}


public class SimpleGraphicDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.add(new MyCanvas());
    f.resize(200,200);
    f.show();
  }
}
```

Slide 254



Slide 255

- Ceci est Juste une petite démo. Il faut noter que dans l'exemple nous avons surchargé `Canvas` mais ceci aurait pu parfaitement marcher avec juste un `Frame` ou un `Panel`. Tous les `component` ont une méthode `paint()` avec une `Graphics` associé !
- Avez -vous bien compris ? Si oui répondez à la question : "Que se passe-t'il lorsqu'on redimensionne la fenêtre ??". Justifiez votre réponse `;->`.
- Attention : de subtiles intéractions avec les `Layout` (entre autre le `GridBagLayout`) font que parfois la méthode `paint()` d'un `Container` n'est pas appelée lors d'un rafraichissement de l'ensemble des composants graphiques. Un exemple suit.

examples/BugPaint.java

```
/* Exemple montrant les interferences entre layout, container, et
   paint ! appeller l'appli avec un parametre pour avoir le cas
   "bugge". Ce n'est pour etre pas un bug, mais simplement que c'est
   ainsi et pas autrement... */
import java.awt.*;

class BugPaint extends Frame {
      static public void main(String[] args){
         // calling the appli with any parameter is entering buggy case
         BugPaint f=new BugPaint(args.length≠0);
         f.setSize(100, 200);
         f.setVisible(true);
      }
      BugPaint(boolean pShowBug){
         setLayout(new FlowLayout());
         add(new Button("Pouet"));
         BPanel p1= new BPanel();
```

```
            add(p1);
            BPanel p2 = (pShowBug) ? p1 :new BPanel();
            p2.setLayout(new GridBagLayout());
            if (p2≠p1){
                p1.add(p2);
            }
            GridBagConstraints c=new GridBagConstraints();
            c.gridwidth=GridBagConstraints.REMAINDER;
            c.fill=GridBagConstraints.HORIZONTAL;
            c.ipadx=10;
            for (int i=0; i<3; i++){
                Button b = new Button("Bouton "+i);
                p2.add(b);
                ((GridBagLayout)p2.getLayout()).setConstraints(b, c);
            }
        }
}
class BPanel extends Panel {                                    // Bordered Panel
        BPanel() {}
    public void paint(Graphics g){
        Color lColor = g.getColor();
        Dimension d=size();
        g.setColor(Color.black);
//          g.drawRect(0,0, d.width-1, d.height-1);
        g.draw3DRect(0,0, d.width-1, d.height-1,false);
        g.setColor(lColor);
        System.out.println("painting "+this);
    }
}
```

LA : ya pas de “pen size” a la quickdraw ? Il faudrait aussi reprendre les expli de graphics : Coordinates are infinitely thin and lie between the pixels of the output device. Operations which draw the outline of a figure operate by traversing along the infinitely thin path with a pixel-sized pen that hangs down and to the right of the anchor point on the path. Operations which fill a figure operate by filling the interior of the infinitely thin path. Operations which render horizontal text render the ascending portion of the characters entirely above the baseline coordinate. Ceci explique d’ailleurs les - 1 dans le paint de l’exemple précédent.

Fonts

- Of course, we can change fonts within a graphics...
- You have to instanciate a `Font` object and associate it to your graphics object.

Slide 256

Fonts

examples/SimpleFontDemo.java

```
import java.awt.*;

class MyCanvas extends Canvas {
  Font myfont;
  public MyCanvas() { myfont=new Font("TimesRoman",Font.PLAIN,24); }
  public void paint(Graphics g) {
    Dimension d = size();
    g.setFont(myfont);
    g.drawString("Jacob de la Font", 0, d.height);
  }
}
public class SimpleFontDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.add(new MyCanvas());
    f.resize(200,200);
    f.show();
  }
}
```

Slide 257

Manisfestement l'AWT n'aime pas les `\n` dans les `drawString()`...

Colours

- Similar to fonts:
- Create a `Colour` object and set the `Graphics` object with the Colour object
- **public** Color(int r,int g,int b)
- The fundamentals colours are defined as constants of the class `Color` ex: `Color.yellow`.

Slide 258

Rappel : les "constantes" sont des variables `final static` d'une classe (ou d'une interface). Dans le cas d'une référence d'objet cela assure que cette référence donnera toujours l'accès au même objet. Mais en aucun cas cela assure que l'objet en question est constant : il suffit de changer son état via la référence `final` ou une autre (appel de méthode, accès à une variable d'instance). Pour les `Color` il n'y a pas de problème : les méthodes ne peuvent pas changer l'état d'une instance (sauf les constructeurs évidemment).

Colours

examples/SimpleColorDemo.java

```
import java.awt.*;
class MyCanvas extends Canvas {
  Color mycolor;
  public MyCanvas() { mycolor=new Color(12,100, 200); }
  public void paint(Graphics g) {
    Dimension d = size();
    g.setColor(mycolor);
    g.drawString("Jacob de la Font", 0, d.height);
  }
}
public class SimpleColorDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    f.add(new MyCanvas());
    f.resize(200,200);
    f.show();
  }
}
```

Slide 259

Ça donne un espèce de bleu (beurk, beurk...)

And the rest ...

- We can't see all :
- Lot of functionalities in the "Graphics" class:
- clipping, translation, scaling, flipping ...
- Take a look to the API `Graphics`...
- Focus on pictures now ...

Slide 260

Pictures

- It seem's quite simple :
- Just create a picture image, and obtain an object of class `Image`
- Just call `Graphic.drawImage()` to draw it.

Slide 261

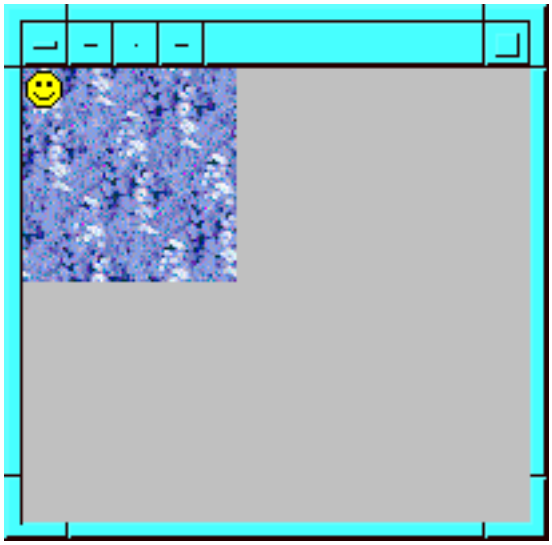
Picture

examples/SimpleImageDemo.java

```
import java.awt.*;
import java.net.*;

class MyCanvas extends Canvas {
  Image myimage;
  Image myimage2;
  public MyCanvas() throws MalformedURLException {
    Toolkit toolkit=Toolkit.getDefaultToolkit();
    myimage=toolkit.getImage(new
URL("http://www.loria.fr/smile.gif"));
    myimage2=toolkit.getImage("clouds.gif");
  }
    public void paint(Graphics g) {
    Dimension d = size();
    g.drawImage(myimage2,0,0,this);
    g.drawImage(myimage,0,0,this);
  }
}

public class SimpleImageDemo {
  static public void main(String args[]) {
    Frame f= new Frame();
    try {
      f.add(new MyCanvas());
    } catch (MalformedURLException e) {
      System.out.println("yo?");
      return;
    }
    f.resize(200,200);
    f.show();
  }
}
```

Slide 262

Images



Slide 263

Images

- The Image constructor just builds an empty image...
- To load a gif picture and creates the corresponding Image object, you have to use the toolkit....
- The toolkit is an encapsulation of the host toolkit which provides
  - peer-object instantiation
  - fonts
  - colours and colour model
  - Screens characteristics
  - Image file loader...

Slide 264

Images

- Hypothesis : Loding a picture may be slow
- If loading is synchronous, programs is blocked...
- Idea : Separates the Image Objects from the pixels that composes the Image. Goal ? Asynchronous loading of images...
- An Image Object is not a pixel array but a pipe between a pixel source and a pixel consumer.

Slide 265

c'est bô ce que je viens de dire là ? c'est pas de moi :-(voir [Anu96][page 315, Working with Images].

Bon certain ramène des belles phrase, et moi bien je ramène ma science : l'idée de "pouvoir commencer à travailler avant que (tous) les pixels arrivent" est très pertinente. En effet certaines techniques de compressions (transformées en cosinus discrêtes) permettent de considérer (envoyer) en premier les informations **les plus pertinentes** et ceci sur l'ensemble d'une l'image. Une image peut alors être `complétement` (au sens de sa surface) affichée en plusieurs fois, avec à chaque fois une qualité accrue. Ceci est d'ailleurs appliquable à la vidéo : on pourra sélectionner une qualité pour un flux vidéo donné.

Images

- The `getImage()` call returns immediately an object of the class "Image": a pixel pipe.
- On one side there is the pixel consumer : just a class that implements the ImageObserver Class
- On the other side, the pixel producer : just a class that implements the ImageProducer interface.

Slide 266

LA : Hum on pourrait mettre un CHJOLI dessin, arch !

ImageObserver

- A class that implements `ImageObserver` has to implement the update method :

  public abstract boolean imageUpdate(Image img,
  int infoflags,
  int x,
  int y,
  int width,
  int height)

- This method is called asynchronously when pixels are avaliable in the pipe (Image object).
- Fortunately, “Component” implements “ImageObserver”, so you don’t have to write the code of this method, it’s inherited.

Slide 267

Images : the process

```
class MyCanvas extends Canvas {
  Image myimage2;
  public MyCanvas() throws MalformedURLException {
    Toolkit toolkit=Toolkit.getDefaultToolkit();
    myimage2=toolkit.getImage("clouds.gif");
  }
  public void paint(Graphics g) {
   g.drawImage(myimage2,0,0,this);
  }
}
```

Slide 268

- Quand l'image est créée avec un `getImage()` : il y a juste instanciation d'un objet `Image`.
- Quand `drawImage()` est appelée, alors l'objet `Image` cherche à donner des pixels à `drawImage`, comme il n'en dispose pas, il crée un nouveau thread pour aller chercher les pixels qu'on lui demande. Pendant ce temps là, `drawImage` rend la main.
- Au fur et à mesure que le thread va apporter des pixels, `imageUpdate()` est appelé. Les pixels sont placés dans l'objet `Graphics` mais pas encore visualisés. `ImageUpdate()` termine son exécution par un appel à `repaint()`, ce qui a pour effet de marquer le composant graphique comme étant à repeindre.
- Lorsque le thread d'affichage vient à s'occuper de ce composant, il voit qu'il est à repeindre et appelle la méthode `update()` du composant qui finit par appeler la méthode `paint()`. Le `drawImage()` s'exécute à nouveau avec les nouvelles données disponibles.
- Cette valse infernale dure tant que tous les pixels en latence ne sont pas arrivés.
- Lorsque tous les pixels sont là, `drawImage` renvoie vrai. L'image est alors complétement visualisé.
- Abracadabra fouchtra !

4.5 Conclusion

Conclusion

- AWT allows to build complex graphical interfaces (Especially with the 1.1 version of the JDK).
- Its design is quite original due to pragmatical choices...
- It receives continuously updates (quite frustrating), (the swing evolution arrives ...)
- However, Building a graphical interface is quite simple (Especially vs Motif in C) and works on most platforms (Browsers included).

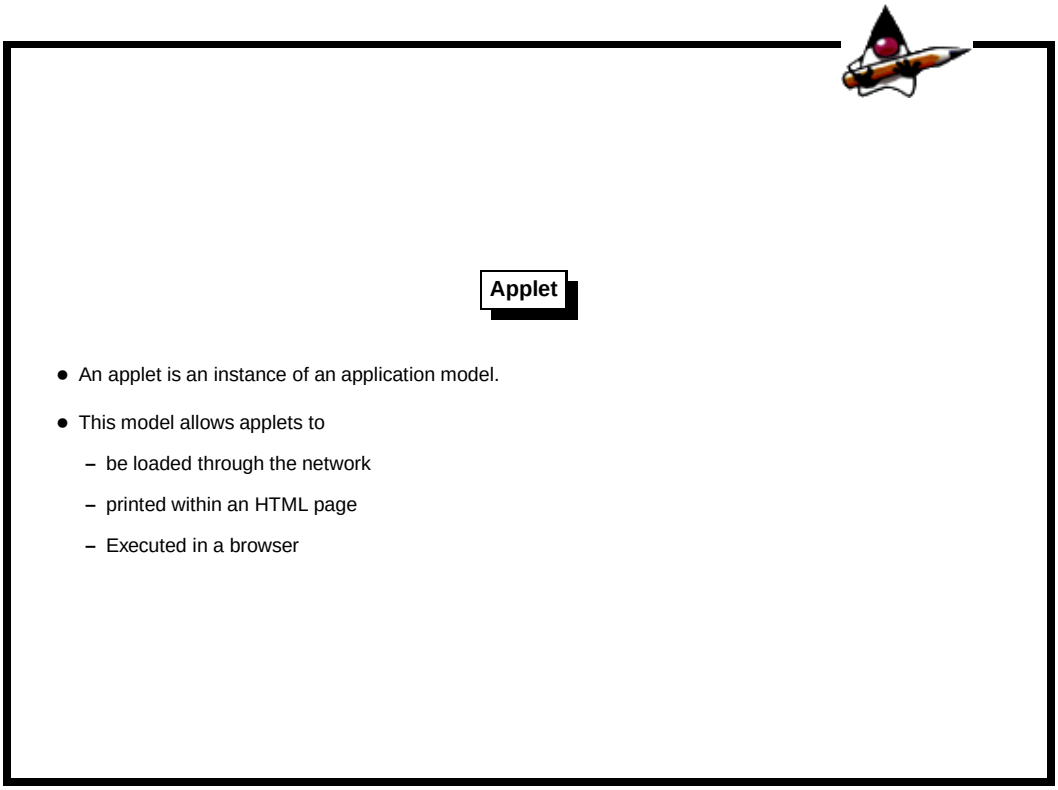
Slide 269

Contents

- Introduction
- Language: classes, inheritance, polymorphism, Packages...
- Threads
- Advanced Window Toolkit (AWT)
- **Applets**
- Conclusion

Slide 270

5 Applet



Slide 271

HTML : Hyper Text Markup Language. Le langage utilisé pour décrire les pages hypertextes diffusées via des serveurs web (protocole http). Voir `http://www.w3.org/` le serveur du consortium w3, on y trouve la spécification de HTML 3.2 (en autre).

Plan

- First Part : What is an applet ?
- Second Part : Applet and security.

Slide 272

Applet

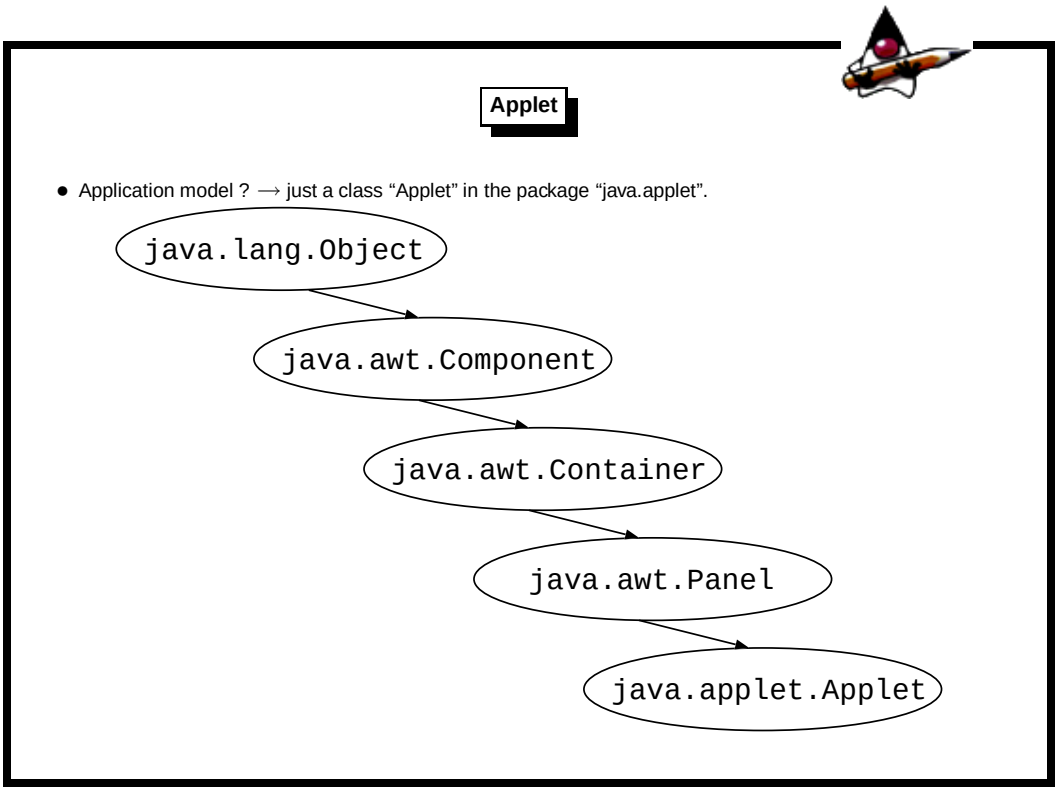
An applet (IMHO) is a restricted form of application :

- always a graphical application...
- No system calls available to ensure security
- Life cycle of an applet within a browser is different from a traditional life cycle of a real application.

Slide 273

Que se passe-t-il quand on arrive sur une page WEB qui contient une applet ? Que se passe-t-il quand on quite cette page ?

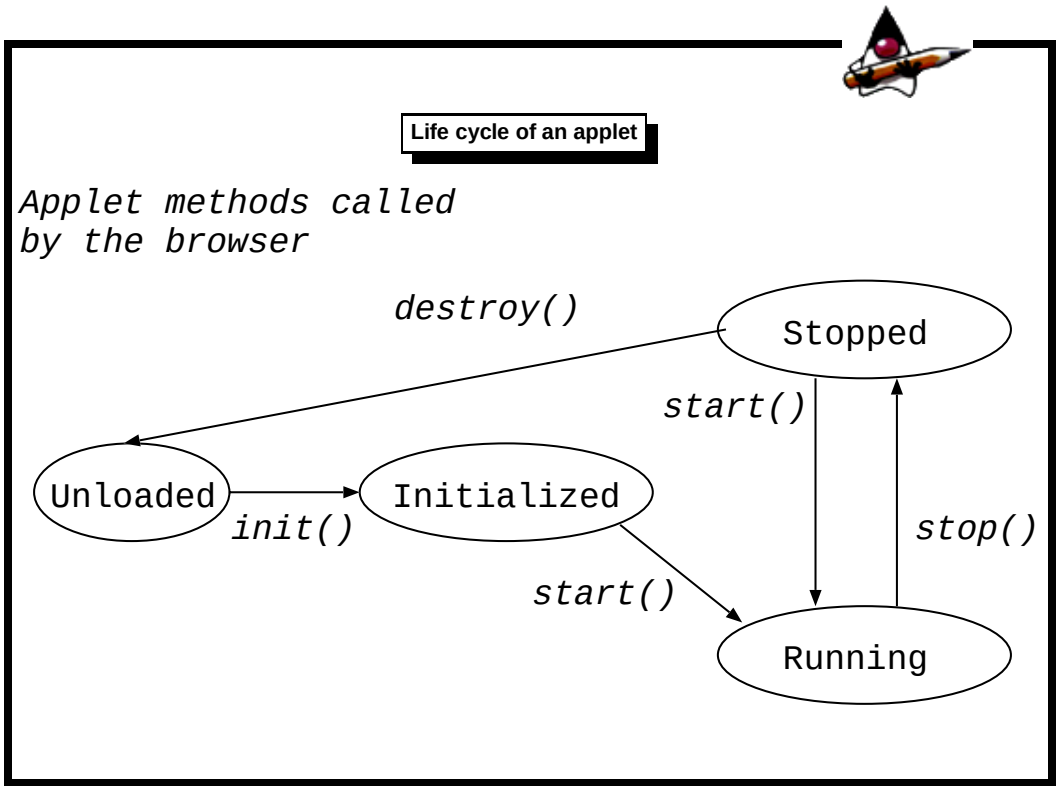
5.1 Applet et AWT



Slide 274

Donc en fait une Applet est un Panel, donc un objet graphique ...

5.2 Life cycle of an applet within a browser



Slide 275

init() est appelée par le browser la première fois que la page WEB hébergant l'applet est chargée.

start() est appelée après init() et chaque que la page WEB hébergeant l'applet est chargée.

stop() est appelée quand on quitte la page hébergeant l'applet avec le browser ou si on iconifie le browser.

destroy() est appelée quand on quitte le browser, ou quand on recharge la page (shift + refresh sous netscape !!). Si l'applet est active, elle est d'abord stoppée.

On voit donc que les points d'entrée nécessaires à l'exécution d'une applet sont un peu plus sophistiqués que le bête `public static main(...)` des applications java.

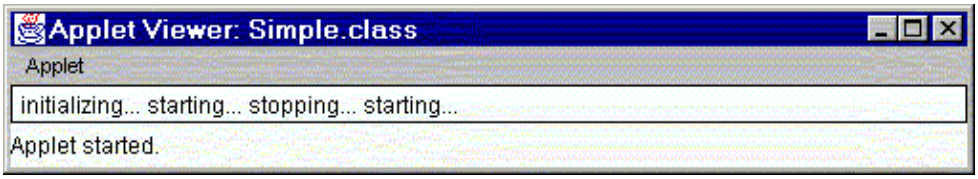
Example

examples/Simple.java

```
import java.applet.Applet;
import java.awt.Graphics;

public class Simple extends Applet {
  StringBuffer buffer;
  public void init() {
    buffer = new StringBuffer();
    addItem("initializing...  ");
  }
  public void start() {
    addItem("starting...  ");
  }
  public void stop() {
    addItem("stopping...  ");
  }
  public void destroy() {
    addItem("preparing for unloading...");
  }
  void addItem(String newWord) {
    System.out.println(newWord);
    buffer.append(newWord);
    repaint();
  }
  public void paint(Graphics g) {
    //Draw a Rectangle around the applet's display area.
    g.drawRect(0, 0, size().width - 1, size().height - 1);
    //Draw the current string inside the rectangle.
    g.drawString(buffer.toString(), 5, 5, 15);
  }
}
```

Slide 276

Appletviewer



Slide 277

5.3 Applets and HTML

Slide 278

Pour "appeler" une applet dans une page HTML, il suffit d'utiliser le TAG `<APPLET>`. Le paramètre `code` permet de dire quelle classe il faut charger en premier, puis initialiser, puis démarer... Bien entendu la classe ainsi référencée doit être une sous classe de `java.applet.Applet`.

Les paramètres `width` et `height` de ce TAG permettent de préciser la taille du *panel* support de l'applet dans le browser.

Pour l'utilisation en général d'un TAG dans une page HTML veuillez vous reporter au standard HTML (voir le site indiqué précédemment).

Parameters passing

```
<APPLET CODE=AppletButton.class
 CODEBASE=example
 WIDTH=350
 HEIGHT=60>
 <PARAM NAME=windowClass
        VALUE=BorderWindow>
 <PARAM NAME=windowTitle
        VALUE="BorderLayout">
 <PARAM NAME=buttonText
        VALUE="Click here">
</APPLET>
```

Slide 279

Les valeurs de paramètres sont forcément des `Strings` !

Parameters passing

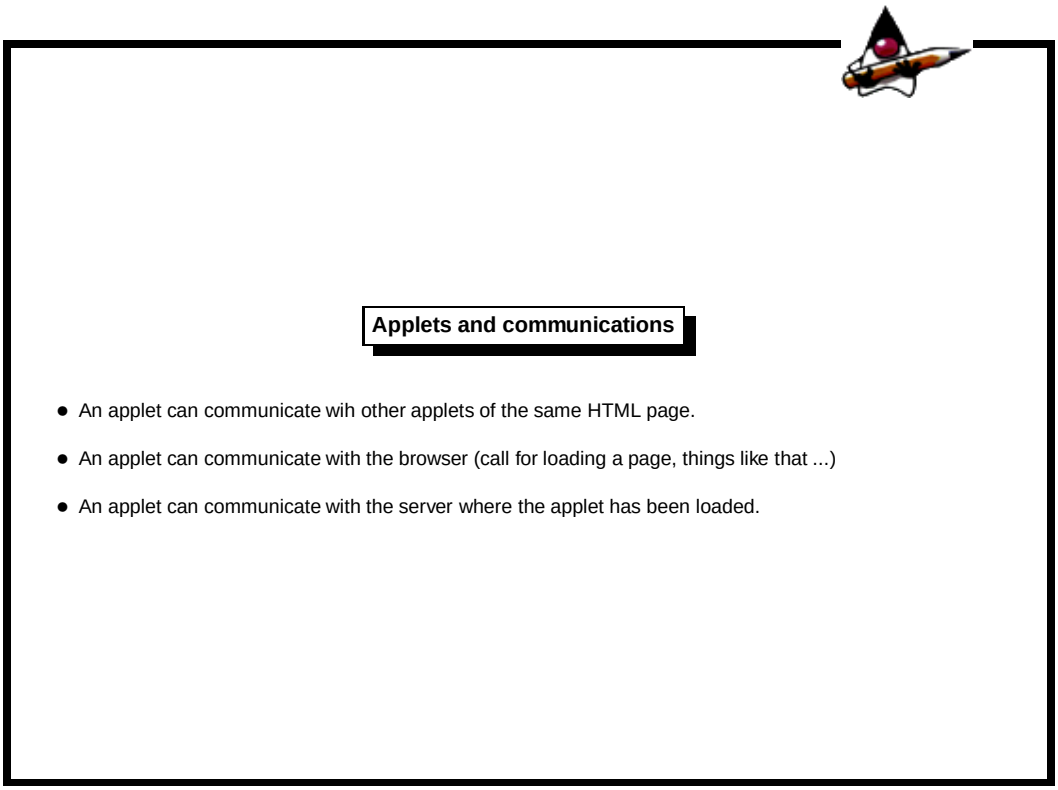
examples/AppletButton.java

```
import java.applet.Applet;
public class AppletButton extends Applet {
        String windowClass;
        String buttonText;
        String windowTitle;
    public void init() {
        windowClass = getParameter("WINDOWCLASS");
        if (windowClass == null) {
     windowClass = "TestWindow";
        }
        buttonText = getParameter("BUTTONTEXT");
        if (buttonText == null) {
            buttonText = "Click here to bring up
a " + windowClass;
        }
        windowTitle = getParameter("WINDOWTITLE");
        if (windowTitle == null) {
            windowTitle = windowClass;
        }
        String windowWidthString =
getParameter("WINDOWWIDTH");
        if (windowWidthString ≠ null) {
            try {
                int requestedWidth =
Integer.parseInt(windowWidthString);
            } catch (NumberFormatException e) {
                //Use default width.
    }}}}
```

Slide 280

A noter qu'en **redéfinissant** la méthode `getParameterInfo()`, il est possible de renseigner le browser sur les paramètres utilisés par une applet.

5.4 Applets and Communications



Slide 281

Aller voir le java tutorial où tout cela est très bien expliqué.

Restrictions

- No Native methods
- No access to local file system
- No network connections except with the original server
- No process
- No access to system properties
- Applet winfows are different from others windows ...

Slide 282

5.4.1 Conclusions

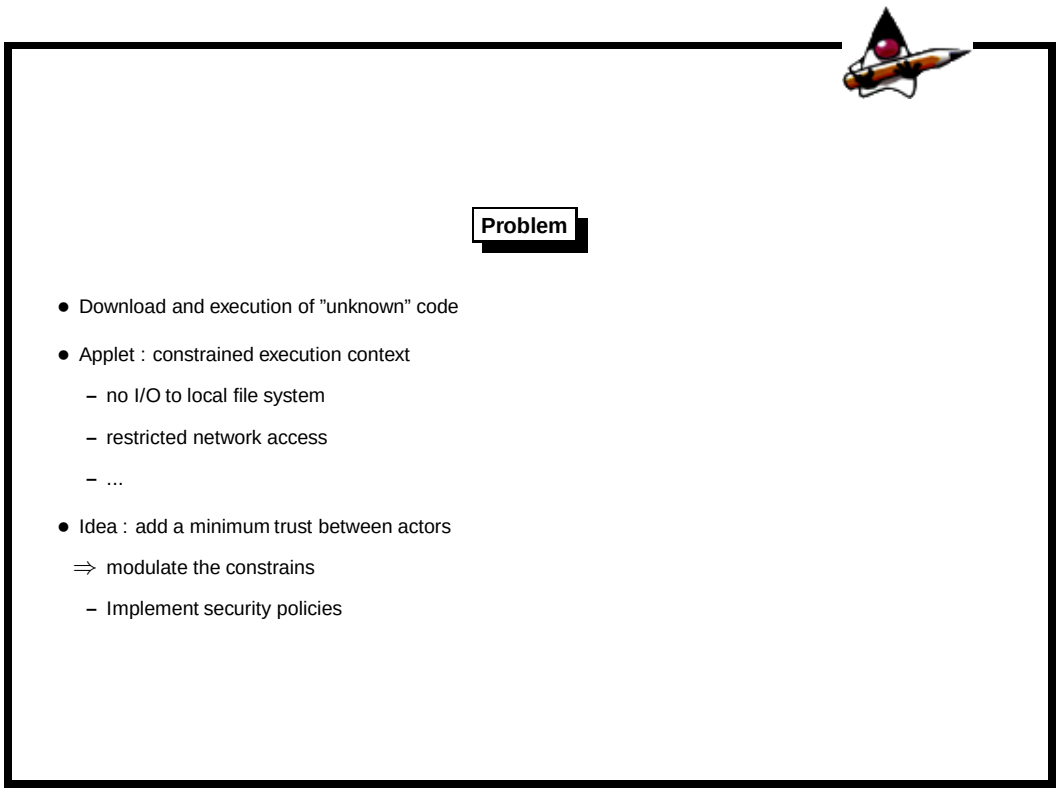
Conclusions

- An applet is restricted form of applications providing its execution within a browser.
- Provides a large scale diffusion... but it's APPLET, it's not APPLICATIONS !
- Others new mechanisms allows large scale diffusion of programs and data (Castanet, Channels ...)

Slide 283

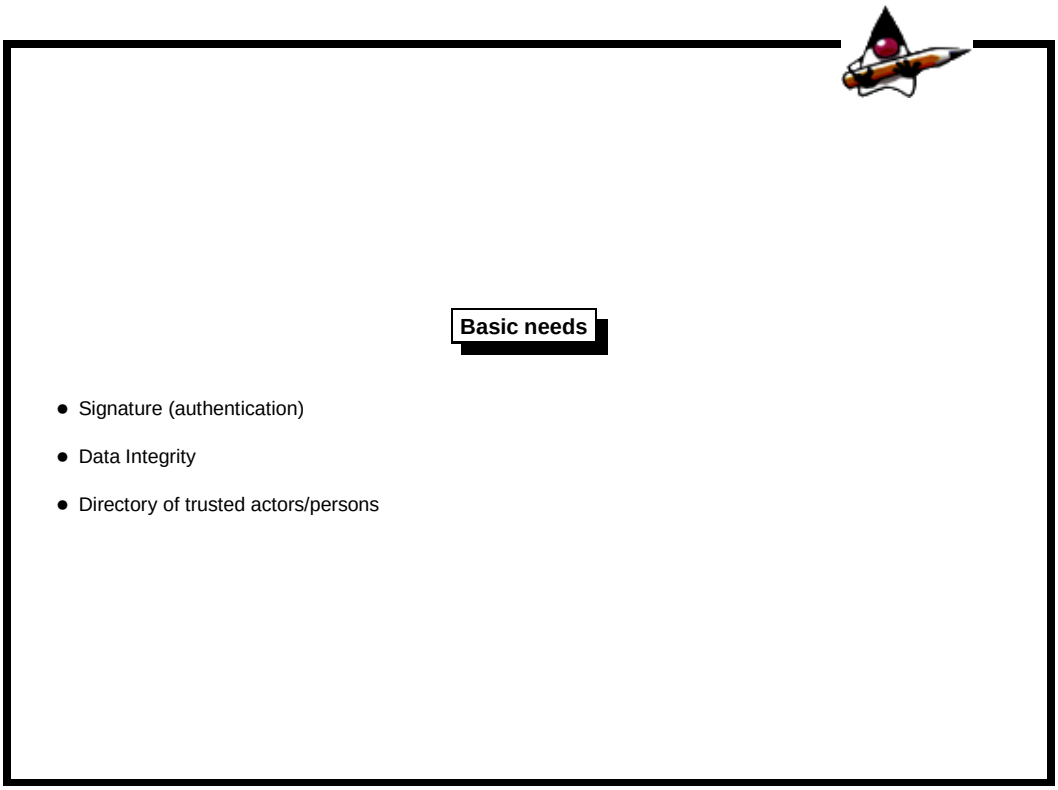
5.5 Code distribution and digital signature

5.5.1 Applet and security



Slide 284

On conseille aux enfants de ne pas accepter de bonbons des inconnus, on peut lui dire ça car on l'estime capable de reconnaitre les amis des pervers pépéres. Il faut donc arriver à déployer des techniques permettant de reconnaitre ses amis.



Slide 285

- En fait on désire l'usage simultané d'une signature (on identifie qui nous envoie quelque chose) et d'un dispositif assurant l'intégrité (le contenu ne peut pas être modifié). On est proche d'un système de sceau (*seal*) fermant un document (type parchemin moyennageux).
- Le répertoire est nécessaire : il faut être capable de mettre un nom, un niveau de confiance sur une signature... Les anciens avaient des grimoires où étaient reproduits armoiries, sceaux des seigneurs connus.
- Il va se poser des problèmes de gestion et de confiance en ce répertoire.

5.5.2 Digital Signature and public keys

Crypto-systems using public keys

- A $user$ is provided with a pair of private/public keys: K^{user}_{Priv}, K^{uti}_{Pub}
- Any of those keys can cypher a data, and the other can decipher it (**asymmetric** crypto-system):
  1. $K^{user}_{Priv}[K^{user}_{Pub}[D]] = D$
  2. $K^{user}_{Pub}[K^{user}_{Priv}[D]] = D$
- A user does not need to communicate its private key to anybody. **No share secret**
- The receiver just needs the sender's public key
- Ex: RSA, DSA, PGP
- CPU consuming

Slide 286

DSA Digital Signature Algorithm, défini dans NIST's Digital Signature Standard document, fip-186.

RSA Rivest, Shamir and Adleman public key system. RSA Encryption défini dans RSA Laboratory Technical Note PKCS#1.

PGP Pretty Good Privacy

DES Data Encryption System. Le système de cryptographie à clef **secrête** le plus commun. Ici les clefs de codage et de décodage sont les **même**. Il y a **partage du secret** entre émetteur et récepteur (en général entre groupe d'utilisateurs communicant ensemble). Il est impossible de prouver à une tierce persone qui a généré quoi. Ce système permet aux membre du groupe de se protéger de l'extérieure. Ce genre de crypto-système est dit **symétrique**.

L'idée des algorithmes à clef publique est d'utiliser des fonctions de codage dont l'algorithme inverse est connu, mais aussi connu pour être lourd à mettre en œuvre (complexité élevée). Seuls les possesseurs de certains éléments (une moitié de paire) peuvent en pratique faire l'inverse. La pluspart des systèmes à clef publique sont bâtis sur le fait que la décomposition en facteurs premiers d'un nombre est supposée être un problème non polynômial (exponantiel). À partir d'un "grand nombre" (512 bits) cela devient réellement impossible à l'heure actuelle (mais bon vu à la vitesse où les cadences de processeurs sont multipliées par deux ...).

Public keys bits length is a current hot topic.

Mettre des refs sur PGP. Situer tout ça face à kerberos. OTP and other bullshit.

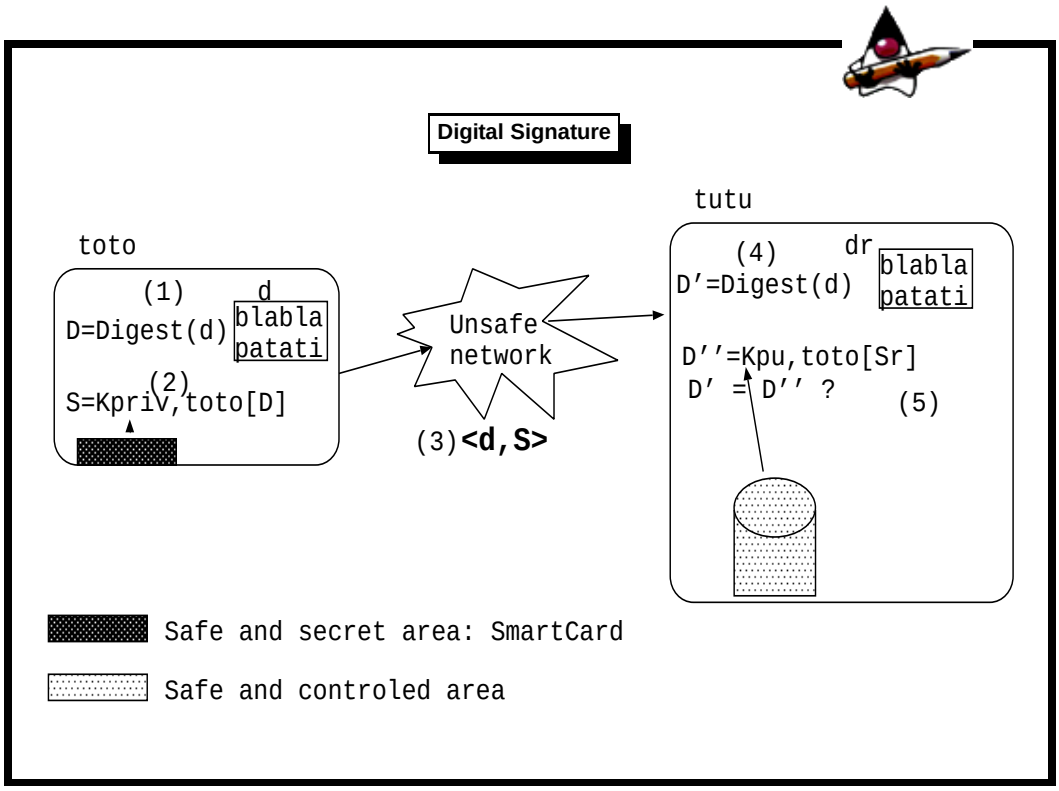
Message digest algorithms

- Goal : Get an about ten byte length digest from any input data (message digest, data footprint)
- Expected properties:
  1. Surjection **all possible data** $\longrightarrow$ **all possible digests**.
  2. Correlate a change on the data to a change on a digest is impossible $\Rightarrow$ It is impossible to "compensate" (counterbalance) a modification on the data to keep a known value for the digest
- Ex : SHA-1, MD5, MD2
  - SHA : *Security Hash Algorithm* from NIST
  - MD : *Message Digest* from RSA

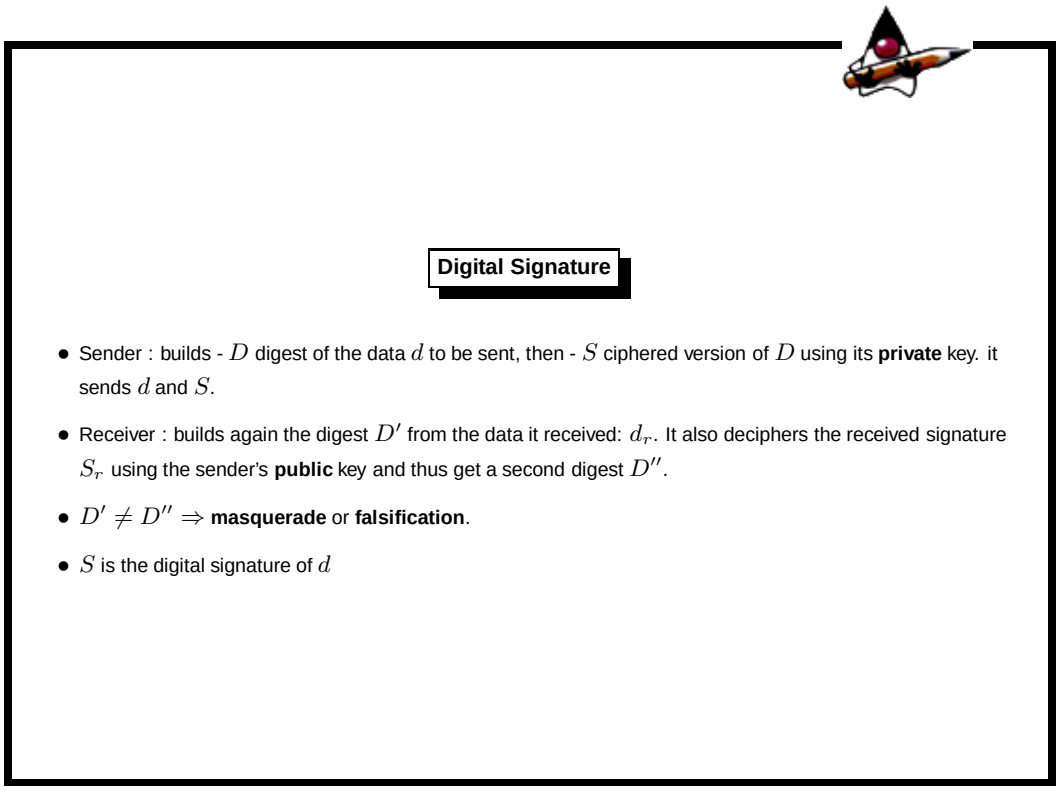
Slide 287

Un contre exemple typique d'un algorithme de résumé de sécurité sont les *check sum*. Ce sont souvent une addition en ligne en complément à 1, modulo 8 ou 16. Rien de plus facile à compenser !

Les propriétés des algorithmes de résumé (leur "sécurité") correspondent à des études mathématiques/statistiques poussées.



Slide 288



Slide 289

- Le transparent précédent montre comment on peut transmettre une donnée visible par tous, mais telle que :
  - Tout lecteur peut s'assurer que c'est bien l'émetteur prétendu qui a écrit le message (pas de **mascarade**),
  - Que personne n'a **falsifié** le message.
- Le système présenté n'empêche pas en temps que tel les possibilités de *replay*. En effet rien n'empêche une tierce personne de de réémettre le couple $< D, S >$. Ceci peut être une facilité utile : distribution de code à partir de mirroir. Pour éviter cela il faut ajouter les *time stamps* ou des numéros de série dans le corps du message (numéro d'ordre d'une transaction bancaire...).
- Un point critique est : l'endroit où est stocké la clef privée d'un utilisateur. De nombreux systèmes les mettent sur les cartes à puce ou autres *security tokens*. Certains vont même jusqu'à y implanter les algorithmes de tirage de clefs et de cryptographie par la clef privée de façon à qui n'y ait pas besoin de devoir rendre celle-ci visible à l'extérieur de la carte.
- Un deuxième point critique, mais moins immédiat est l'accès et la distribution des clefs publiques. Le récepteur a besoin d'un répertoire maintenant des associations $< nom\ d'un\ signataire, clef\ publique >$. Or une clef publique est forcément créée en même temps que la clef privée associé : tirage d'une paire de clefs. Ceci est fait en général par le signataire. Le problème est : il faut pouvoir diffuser les enregistrements $< nom\ d'un\ signataire, clef\ publique >$ d'une façon à ce qu'une personne les recevant ait **confiance**. En effet si aucune mesure n'est prise rien n'empêche qu'une personne malveillante, disons X, ne génére une paire de clefs K', puis ne diffuse un "faux" enregistrement $< toto, K'_{pub} >$. X pourra alors générer des messages signés via K'_{priv}, et les personnes "contaminées" par les enregistrements croiront que c'est toto qui les contacte. On a donc tout perdu : X peut très bien émettre un ordre de virement du compte de toto vers une banque suisse, où il dispose d'un compte anonyme... La diffusion des enregistrements sans être confidentielle, doit être sûre. Elle peut être fait de grès à grès entre chaque utilisteur sans passer par le réseau "non sûr" (*out of band way*). Cela peut être très lourd si on veut déployer un système avec de nombreux utilisateurs : difficulté de la diffusion initiale, lenteur des mises à jour ou des invalidations en cas d'incidents. Une technique existente : la mise en place d'autorités de certification et la diffusion de certificats.

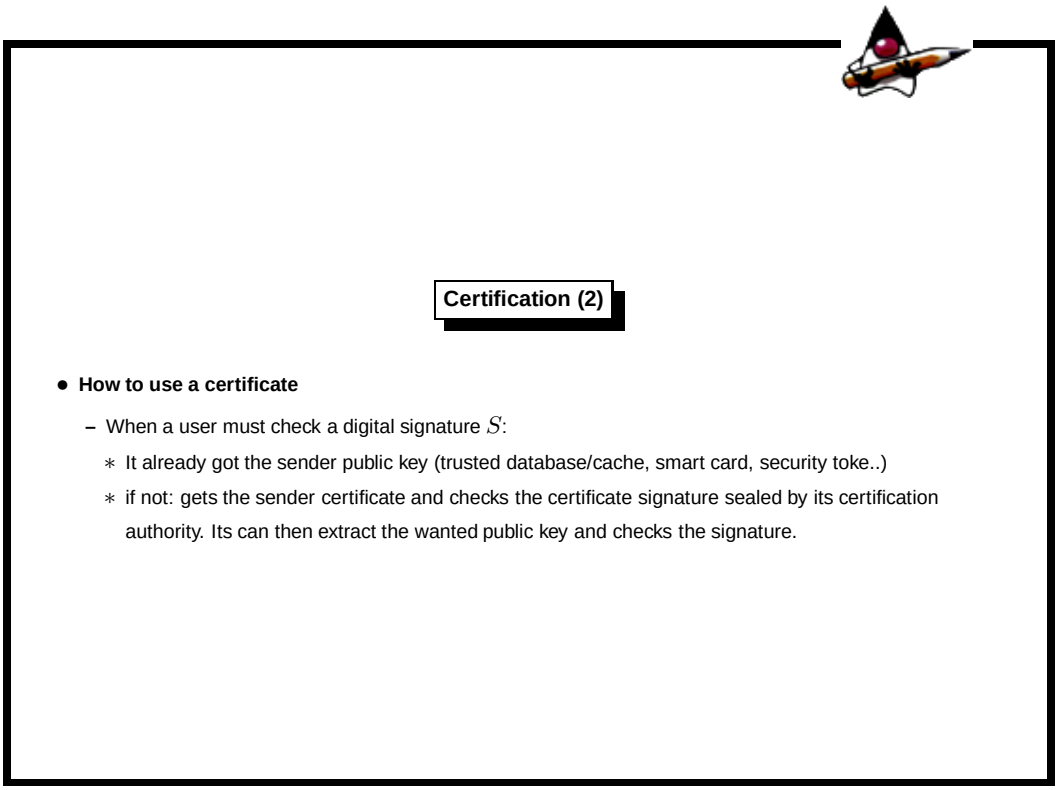
5.5.3 Large scale deployment of public keys: certification

Certification

- **Goal** : Distribute in a trusted way records like: $< name, public\ key >$.
- **How ? - make certificate**
  - Creation of a special signer: a "certification authority" C, featured with a key pair: K^C_{priv} and K^C_{pub}.
  - When a new user $user$ has to be able to issue some digital signatures:
    1. Key pair determination: K^{user}_{priv} and K^{user}_{pub}
    2. *out of band* delivery by C of K^C_{pub} to $user$ who stores it in a trusted local place
    3. *out of band* delivery by $user$ of K^{user}_{pub} to C
    4. **Certificate** generation for $user$: C **digitally signs** a $< user, K^{uti}_{user} >$ record using its private key (K^C_{priv})

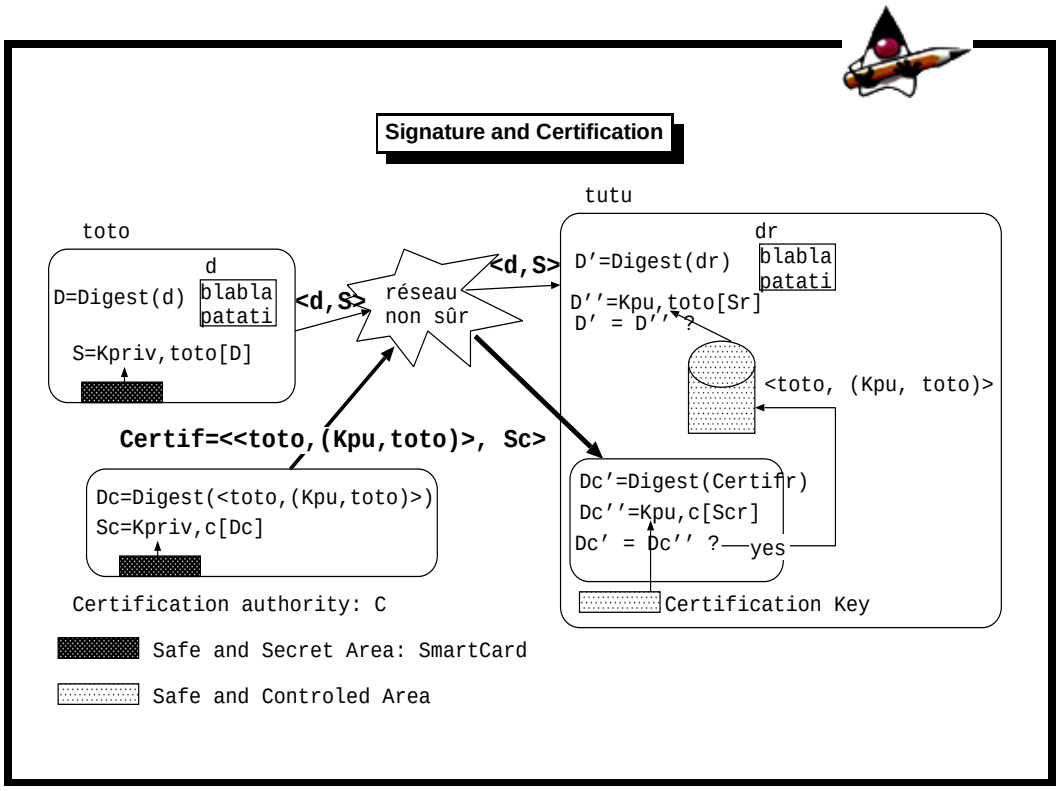
Slide 290

Indeed the signed record to build a certificat features more information that a simple name: compagny, creation date, validity...

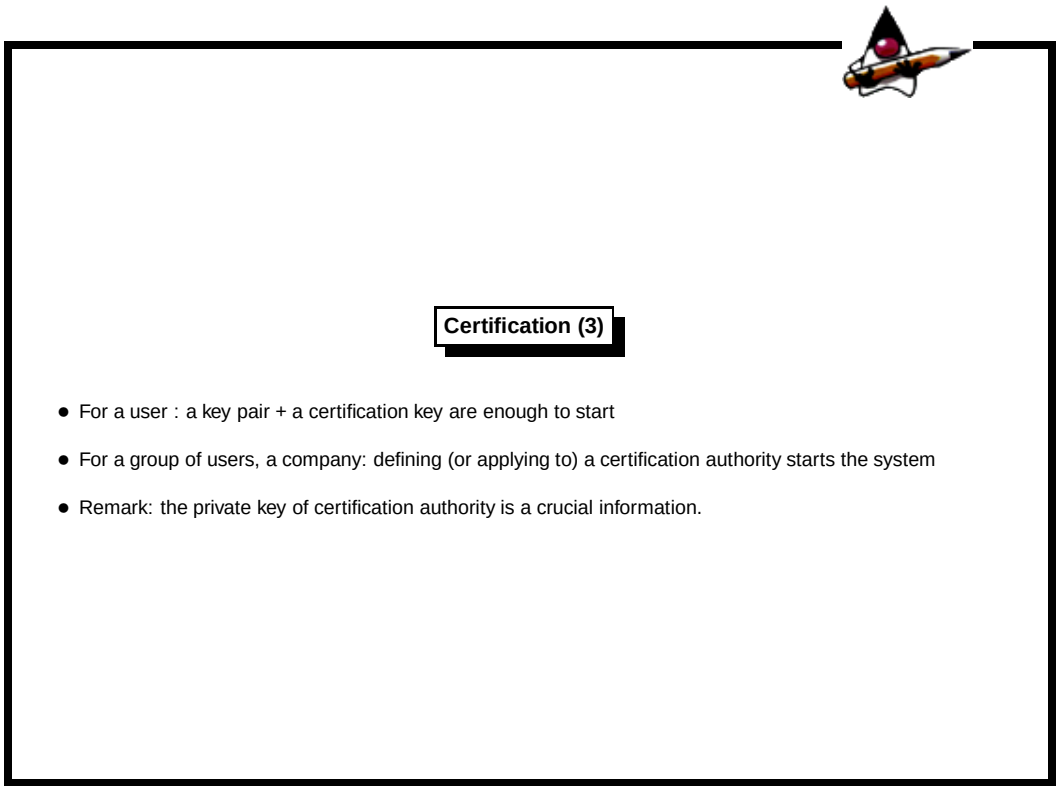


Slide 291

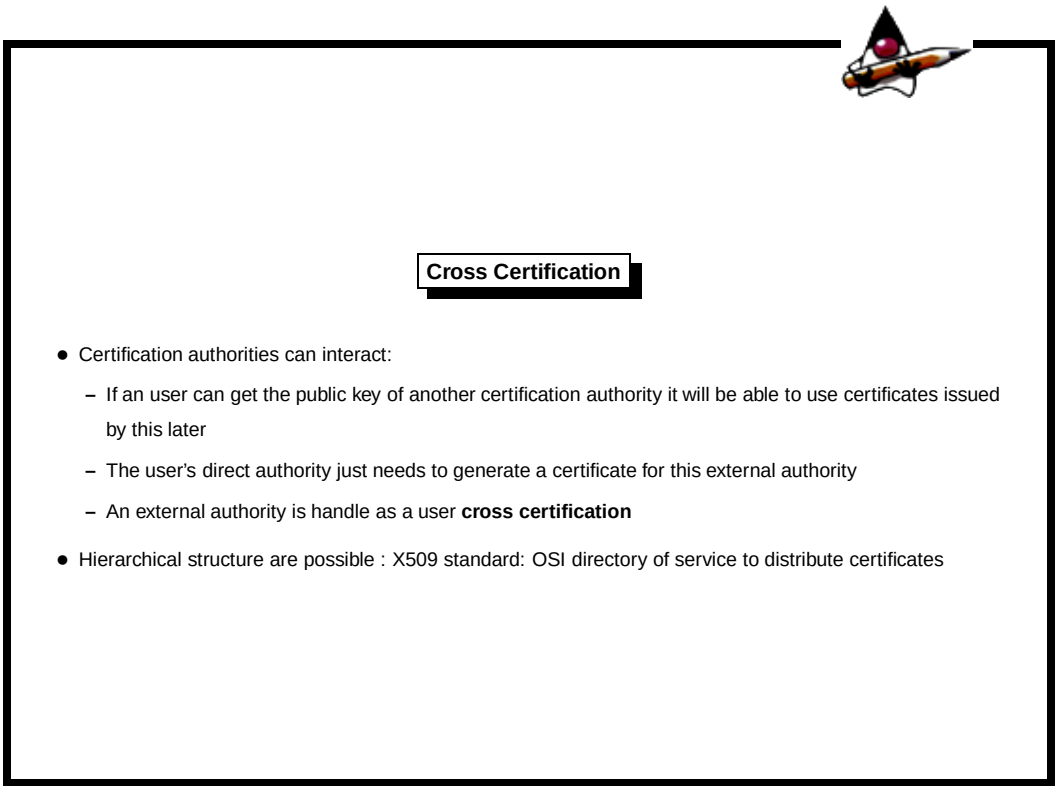
Typically: Verign is an certification authority. its public key is distributed "out of band" together with your netscape/ie (question: how can you trust the binary code of your browser 8-¡). Lots of commercial web servers pay to get a certificat signed by Verign and so can immediatly world wide sprend it. This certificat is use to open SSL connection to the server (https): the browser window with a lock in it.



Slide 292



Slide 293



Slide 294

Les autorités de certification **gérent** les certificats : date de péremptions, invalidations si un utilisateur n'a plus confiance en sa clef privée...

Il est donc clair qu'un certificat contient plus qu'un nom et une clef publique. De plus, vu qu'ils sont prévus pour être échangés ⇒ une standardisation de leur format est critique (X509, RFCs de la série PEM (*Privacy Ehanced Mail*)).

Certificate Example

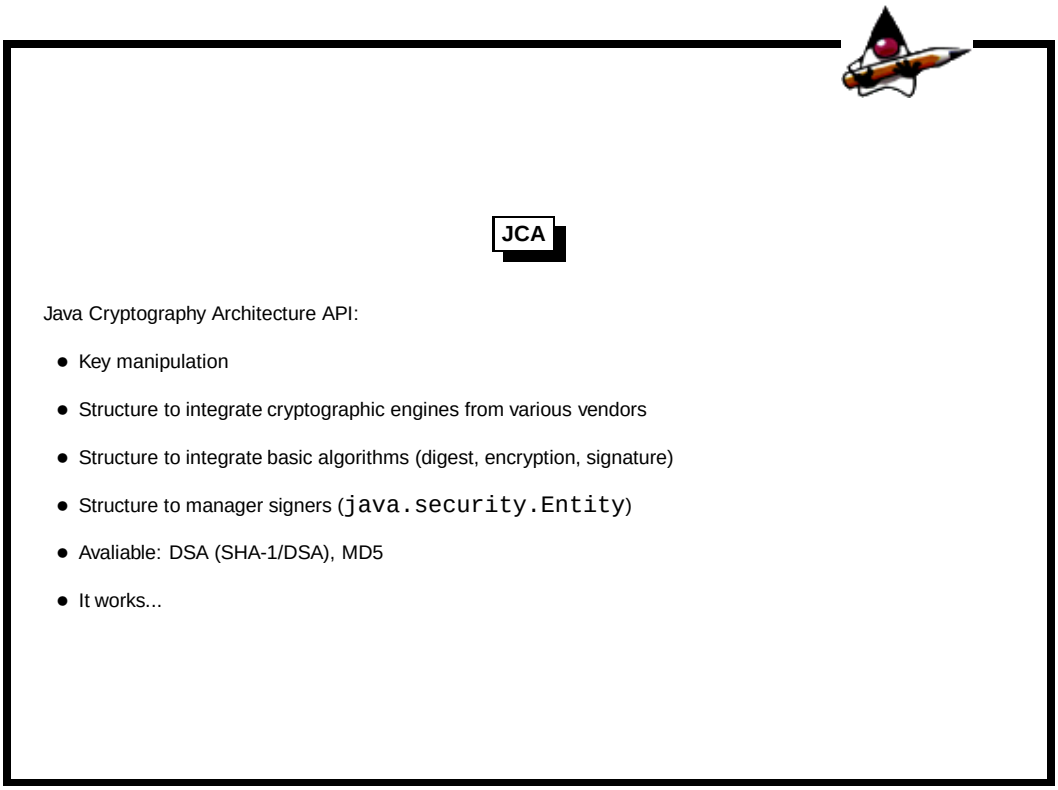
```
% javakey -dc toto.x509
[ X.509v1 certificate,
  Subject is CN=Arthur Toto, OU=derriere L'eglise,
  O=couillonade Inc., C=France
  Key:  Sun DSA Public Key
parameters:
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae
01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d1427
1b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4
y: 486fa6ec689096bc130acf9d7d01b2b800409e3d42d4d3c41494705955c7
dd7a4f7bbd0f4bd0118a1a8ab59b7db52f0293adab4f13f380b2278407fd72cf51c0
  Validity <Fri Jan 31 16:00:00 PST 1997> until
  <Sat Jan 31 16:00:00 PST 1998>
  Issuer is CN=Test autorite certification, OU=kra,
  O=Kramoll's Inc., C=France
  Issuer signature used [SHA1withDSA]
  Serial number =     03e9                 ]
```

Slide 295

Ceci est la version "lisible" d'un certificat affichée par `javakey` (voir ci après). Le format d'échange est une structure ASN1.

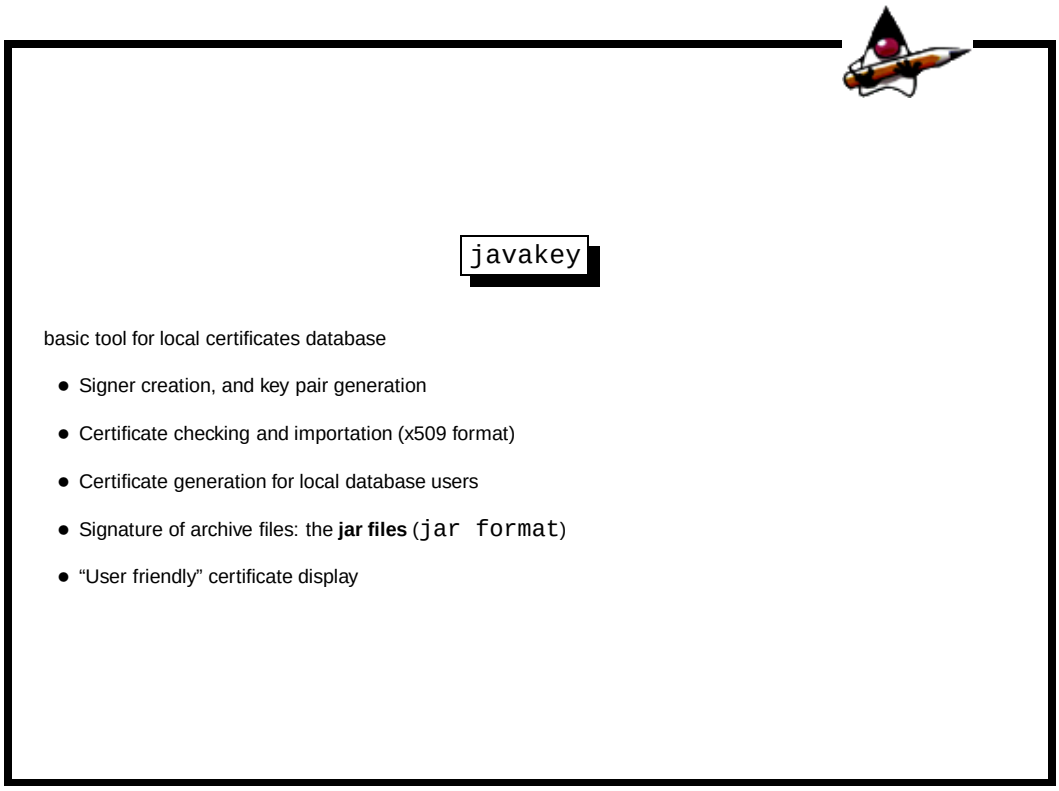
Le champs `y` *doit* être la signature.

5.5.4 jdk 1.1 implementation



Slide 296

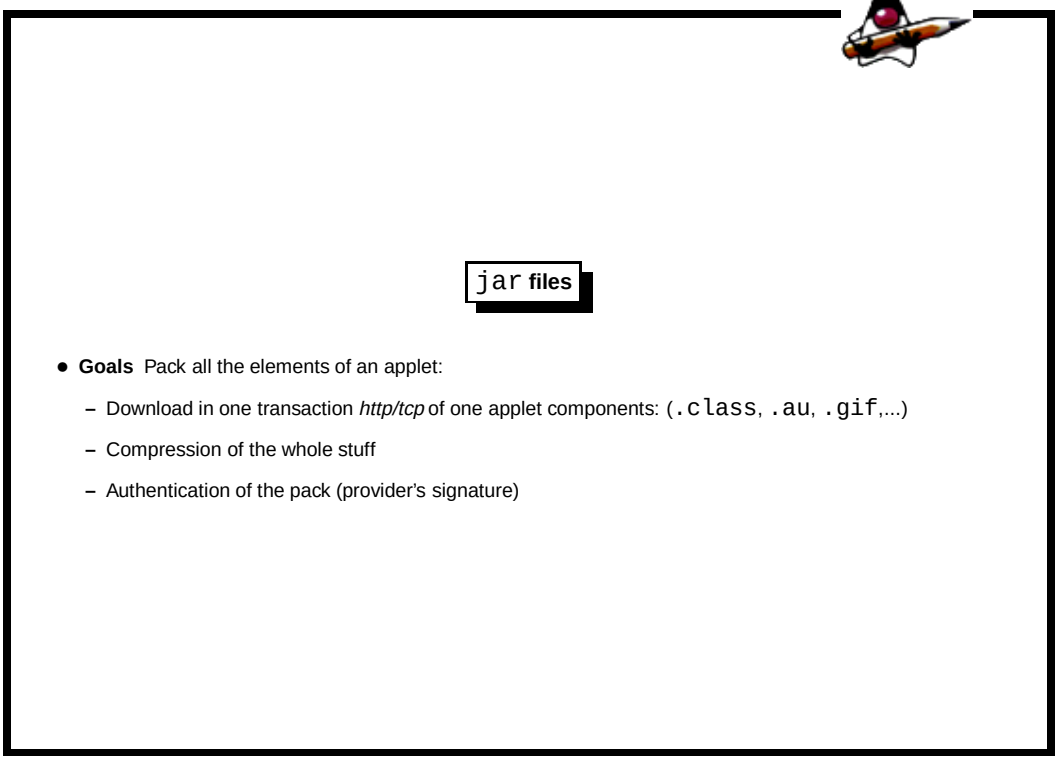
- Il existe un package : `JCE` (*Java Cryptography Extension*) fournissant le DES et autres facilités, mais il n'est pas actuellement exportable à l'extérieur des états unis, vu la législation locale. De manière générale l'usage de telles techniques est réglementé dans la pluspart des pays. Aux états unis, c'est l'inénarable NSA (*National Secret Agency* à coté de laquelle la CIA et le FBI sont des rigolos) qui impose ses vues.
- Il y a encore quelque problème de design de la JCA dans le jdk1.1b3, mais le jdk1.1 les fixe.
- Tout ceci n'existe pas pour le jdk1.02.



Slide 297

- Une telle base peut servir à un utilisateur final : il s'y crée, génére ses clefs, exporte sa clefs publique vers l'autorité de certification (via un *self signed certificat*), importe des clefs de certification (via un *self signed certificat*), importe des certificats. Dans ce cas l'exportation ou l'importation doit être reliés à un échange "sûr".
- Elle peut aussi servir à la certification : création de l'autorité et de ses clefs ; importation des clefs publics des membres (via des *self signed certificats*, *out of band*) et générationi de nouveaux certificats signés cette fois-ci par la clef privée de certification.
- Un reproche ? Un fichier Unix, ou Window n'est pas forcément un "bon endroit" pour une clef privée... Des éléments de réponses à ce genre de questions est disponible dans `http: //www.javasoft.com:80/ security/policy.html`, les *java key policy recommendations*.

5.5.5 The `jar` file: Java ARchives



Slide 298

L'idée est d'augmenter un peu les performances lors du chargement d'une applet. L'idée de paquer l'ensemble à la tar/compress semble antinomique avec l'édition de lien tardive mise en œuvre dans java. Mais dans l'avenir il y aura certainement possibilité de charger des fichiers jar supplémentaires en cours d'exécution...

On peut imaginer mettre uniquement dans un fichier `jar` signé tout le code demandant des "priviléges" à l'exécution, et les ressources (images, sons) absolument nécessaires au démarage d'une applet.

Tout ceci est au début d'un déploiment complet...

Commands

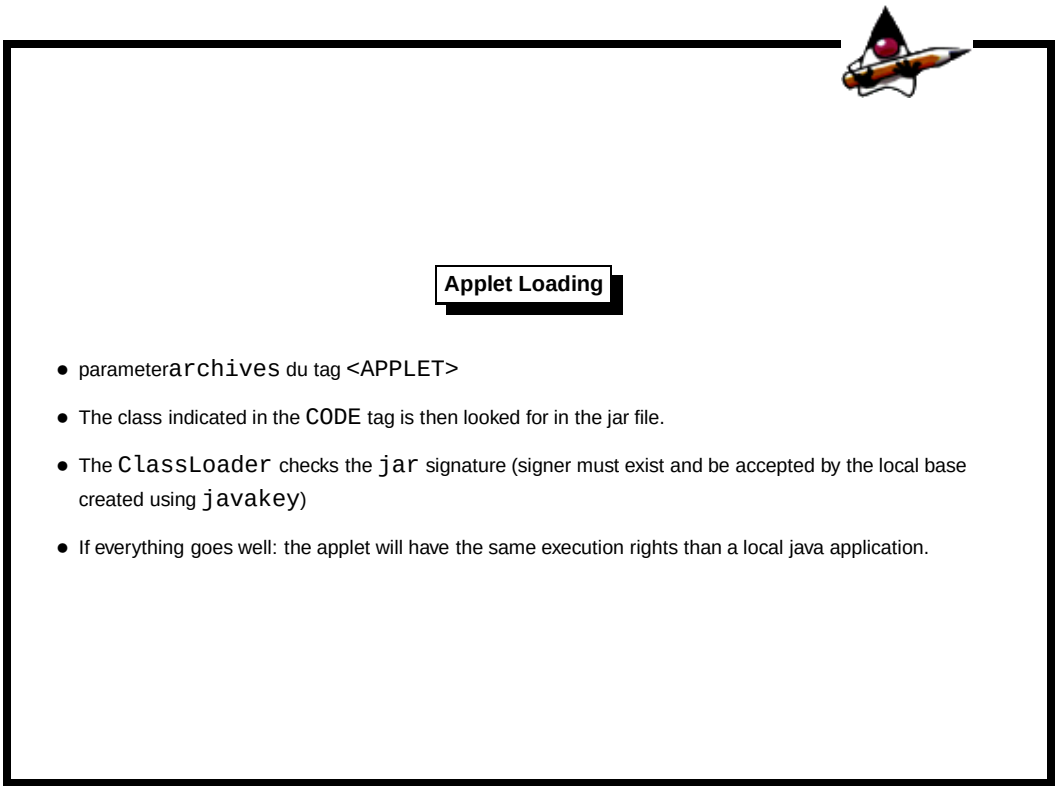
- `jar`:
  - Creates an archive file (ZLIB format)
  - Options *à la* tar
  - Adds a directory: `META-INF` for the "meta informations": `MANIFEST.MF` file which includes the archive files digests, signature...
- `javakey` :
  - Generates a signature for a `jar` file (option `-gs`)
  - Adds a `.SG` file `META-INF` directory

Slide 299

Un jar fichier est donc :

- un fichier d'archive zippé,
- contenant un répertoire spécial `META-INF` où on trouve des fichiers :
- `MANIFEST.MF` indiquant quels éléments de l'archive ont été signés, et comment ils l'ont été.
- Le méta répertoire contient aussi les signatures proprement dites : fichiers `.SG`.

La spécification du contenu des fichiers `MANIFEST` et de signature est disponible à : `http://www.javasoft.com:80/ products/jdk/1.1/docs /guide/jar/ manifest.html`. Il faut noter que finalement le système de méta file et fichier signature proposé peut s'appliquer à tout type d'archive du moment où il fournit un système de nomage hiérarchique (un bon vieux système de fichiers UNIX, ou NT doit pouvoir faire l'affaire).



Slide 300

Bien entendu actuellement aucun browser html du commerce ne supporte ces extentions du tag `<APPLET>`. Le dernier hotjava / jdk1.1 est *sensé* l'implanter...

Extrait de : `http://www.javasoft.com/ products/jdk/1.1/docs/ guide/jar/jarGuide.html`

The APPLET tag

Changing the APPLET tag in your HTML page to accomodate a JAR file is simple. The JAR file on the server is identified by the ARCHIVES parameter, where the directory location of the jar file should be relative to the location of the html page:

```
<applet code=Animator.class
  archives="jars/animator.jar"
  width=460 height=160>
  <param name=foo value="bar">
</applet>
```

Note that the familiar `CODE=myApplet.class` parameter must still be present. The `CODE` parameter, as always, identifies the name of the applet where execution begins. However, the class file for the applet and all of its helper classes are loaded from the JAR file.

Once the archive file is identified, it is downloaded and separated into its components. During the execution of the applet, when a new class, image or audio clip is requested by the applet, it is searched for first in the archives associated with the applet. If the file is not found amongst the archives that were downloaded, it is searched for on the applet's server, relative to the `CODEBASE` (i.e., it is searched for as in JDK1.0.2).

The archives tag may specify multiple JAR files. Each JAR file must be separated by "`+`" (addition signs). Each file is downloaded in turn:

```
<applet code=Animator.class
  archives="classes.jar +  images.jar +   sounds.jar"
  width=460 height=160>
  <param name=foo value="bar">
```

```
</applet>
```

There can be any amount of white space between entries within the archives parameter. In addition, the archives tag itself is case-insensitive; it can be lower-case, upper-case, or any combination of lower- and upper-case letters, such as ArCHiVes.

A traduire .... Tag Applet et jar
Biblio securité
Détail de javakey ??

5.5.6 Brief Bibliography

FIPS 186, Digital Signature Standard (DSS), specifies a Digital Signature Algorithm appropriate for applications requiring a digital rather than a written signature.

FIPS 185, Escrowed Encryption Standard (EES), specifies a voluntary technology available for protecting telephone communications (e.g., voice, fax, modem).

FIPS 180, Secure Hash Standard (SHS), specifies a Secure Hash Algorithm (SHA) for use with the Digital Signature Standard. Additionally, for applications not requiring a digital signature, the SHA is to be used whenever a secure hash algorithm is required for federal applications.

FIPS 46-2, Data Encryption Standard (DES), provides the technical specifications for the DES.

FIPS 113, Computer Data Authentication, specifies a Data Authentication Algorithm, based upon the DES, which may be used to detect unauthorized modifications to data, both intentional and accidental. The Message Authentication Code as specified in ANSI X9.9 is computed in the same manner as the Data Authentication Code as specified in this standard.

FIPS 140-1, Security Requirements for Cryptographic Modules, establishes the physical and logical security requirements for the design and manufacture of modules implementing NIST-approved cryptographic algorithms.

NIST Special Publication 800-2, Public Key Cryptography, by James Nechvatal, presents a survey of the state-of-the-art of public key cryptography circa 1988-1990.

FIPS 171, Key Management Using ANSI X9.17, adopts ANSI X9.17 and specifies a particular selection of options for the automated distribution of keying material by the federal government using the protocols of ANSI X9.17.

JavaTM Cryptography Architecture API Specification & Reference. Benjamin Renaud - JavaSoft Security Group, Mary Dageforde. `http://www.javasoft.com/ products/jdk/1.1/docs/ guide/security/ CryptoSpec.html`

Information on FIPS 186 is available from:

Computer Systems Laboratory Room B64, Technology Building National Institute of Standards and Technology Gaithersburg, MD 20899-0001 Telephone: (301) 975-2816 Fax: (301) 948-1784 E-mail: dward@enh.nist.gov

Other FIPS and NIST Special Publications are for sale by:

National Technical Information Service U.S. Department of Commerce 5285 Port Royal Road Springfield, VA 22161 Telephone: (703) 487-4650

Copies of "American National Standard for Financial Institution Key Management (Wholesale)," ANSI X9.17, can be purchased from: Washington Publishing Company, P.O. Box 203, Chardon, OH 44024-0203, telephone (800) 334-4912.

Contents

- Introduction
- Language: classes, inheritance, polymorphism, Packages...
- Threads
- Advanced Window Toolkit (AWT)
- Applets
- **Conclusion**

Slide 301

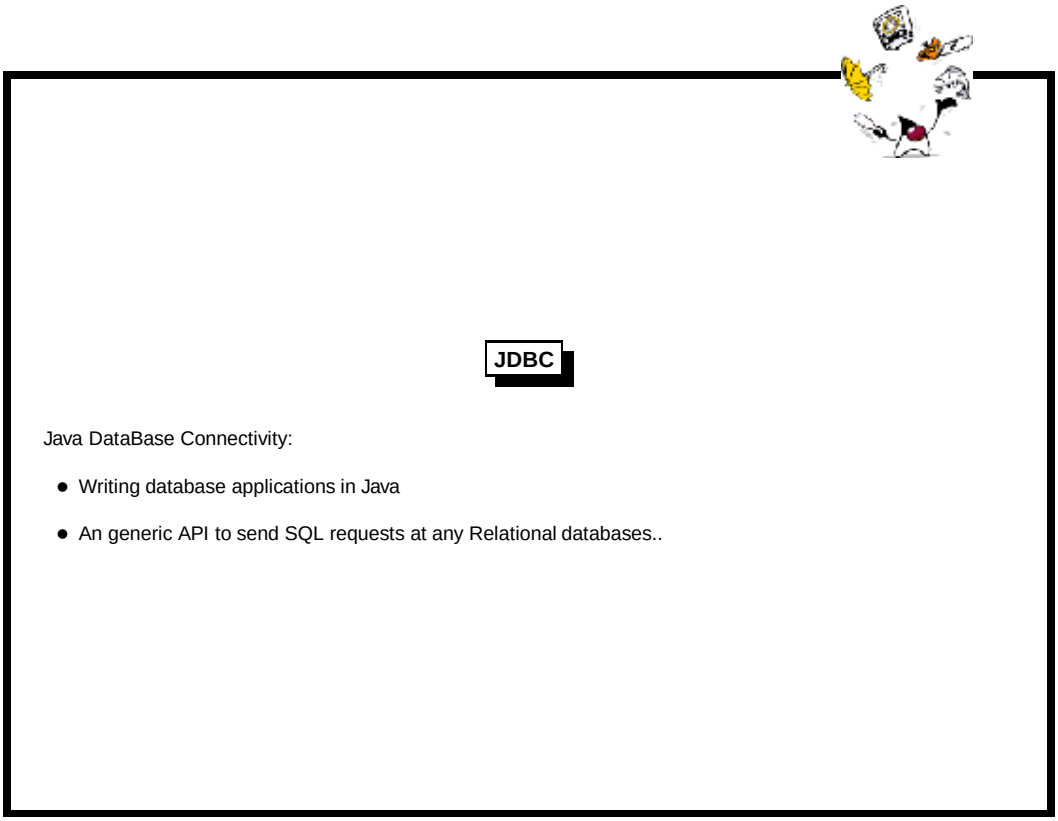
6 Conclusion

Extensions

- JavaOS, PicoJava ...
- Extensions ...
  - JDBC, RMI, JavaBeans, JavaIDL ..
  - JINI, JavaMedia, Java3d...
  - JavaServer, Java SmartCard, Embedded Java, Personal Java ...

Slide 302

6.1 Two words on JDBC



Slide 303

6.2 Two words on RMI

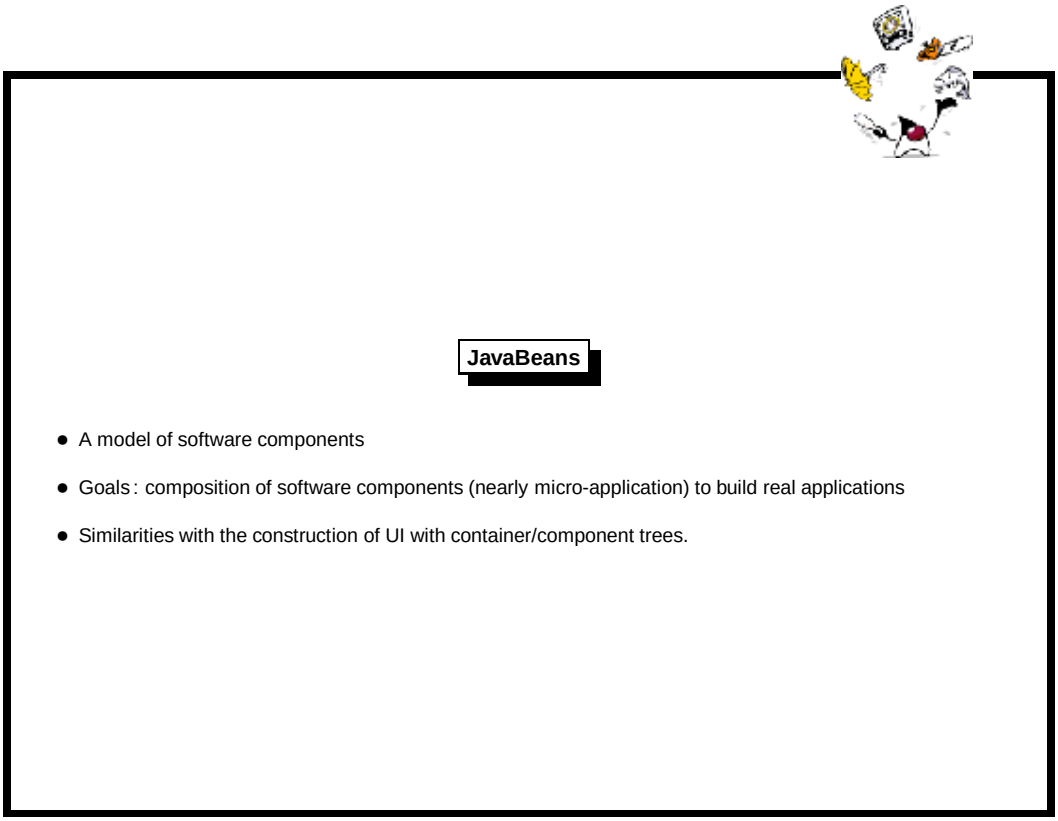
RMI

Remote Method Invocation:

- The Java version of RPC...
- Built as a language extension
- A program running on a JVM can obtain a reference to another object on another JVM.
- Once you have a reference to an object you can call methods as if the remote object is local.
- A distributed garbage collector extends the traditional garbage collector for working with RMI.

Slide 304

6.3 Three words on JavaBeans



Slide 305

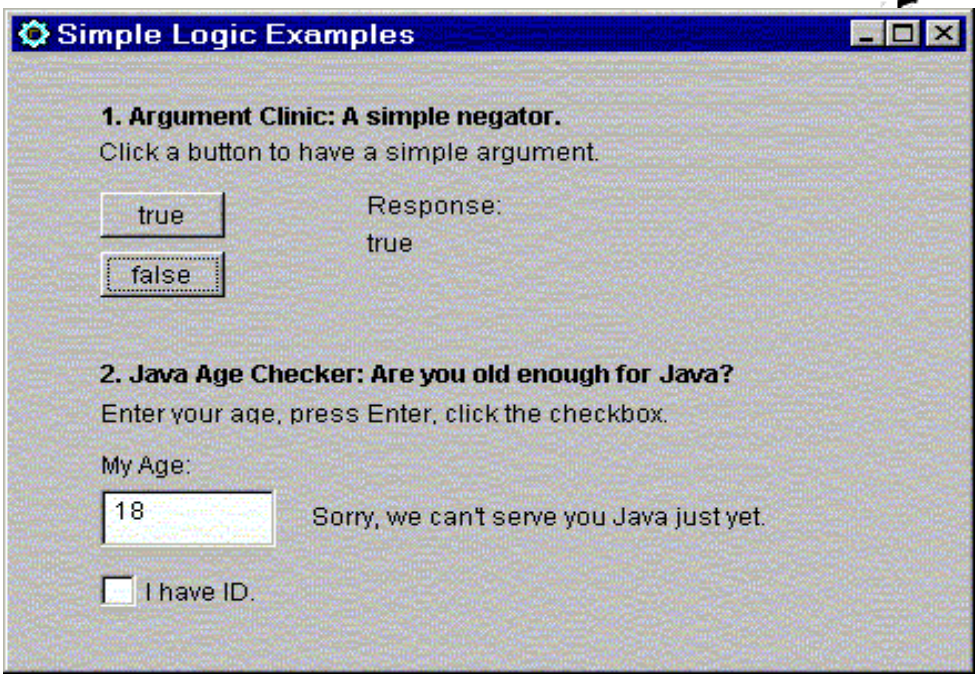
Je conseille fortement la lecture de JavaBeans white paper qui décrit assez ce qu'est un beans ...

JavaBeans

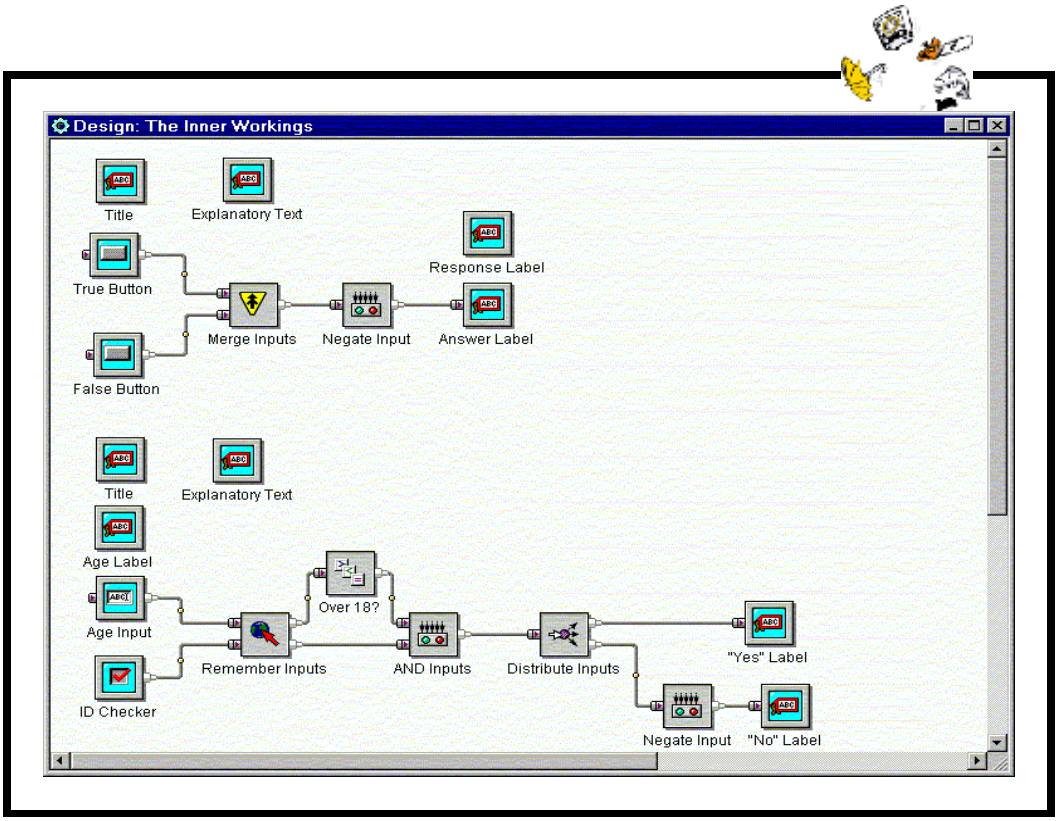
A bean is a class that have to implement the Bean Interface. The Bean Interface provides :

- Interface declaration and run-time discovery of the interfaces of the others components on the same container
- Attributes
- Events management
- Persistency
- Support for application builder !
- Packaging

Slide 306



Slide 307



Slide 308

6.4 Overall Conclusion

Overall Conclusion

- The application field is very big. A priori, we can find a JVM in almost anything:
  - A mobile phone
  - A network equipment
  - A network computer
  - A database Server
  - A Smart Card
  - A washing machine ? A coffee machine ?

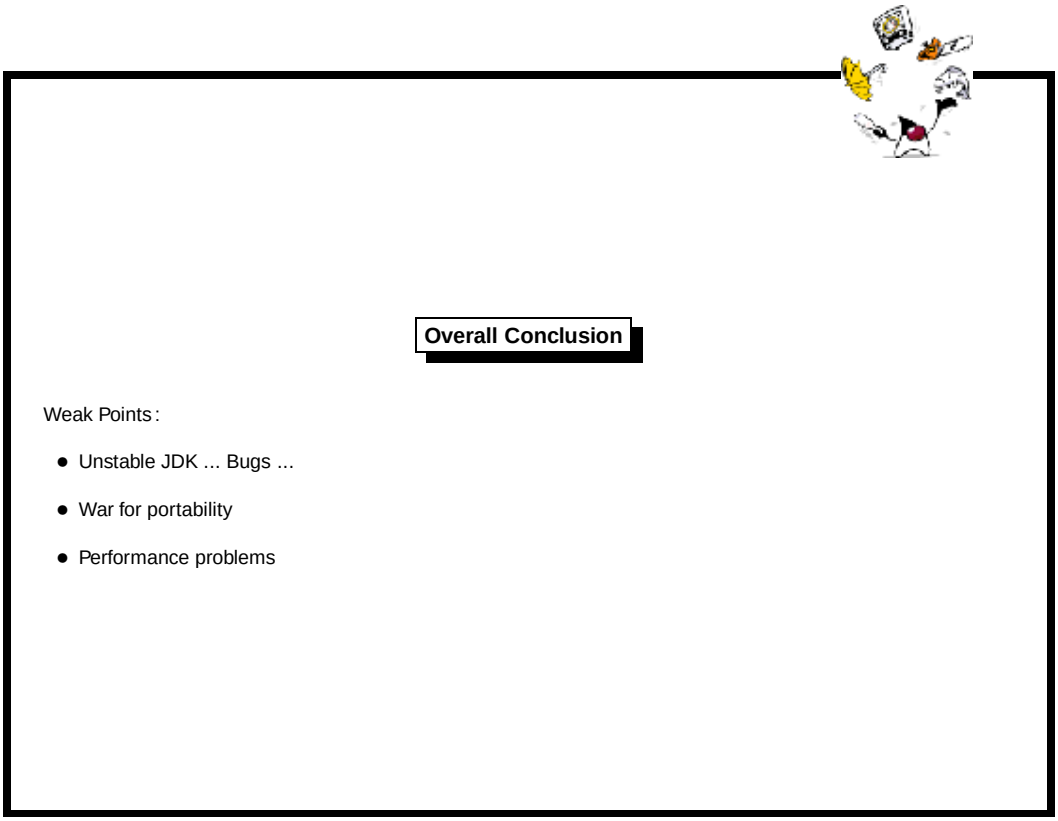
Slide 309

Overall Conclusion

Strong points :

- “Write once, run everywhere”
- Huge libraries to build complex applications (AWT, RMI, JDBC, Beans ...)
- Possibilities of deployment on a wide area network with security ...

Slide 310



Slide 311

la guerre des géants n'y est pas étrangère à ces problèmes ... Difficulté pour les développeurs de sunsoft et javasoft de répondre à la pression des utilisateurs et des concurrents.

References

[Anu96] Ed Anuff. *JAVA Source Book*. Wiley Computer Publishing, 1996.

[Arn96] Ken Arnold et James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[Fla96] David Flanagan. *Java in Nutshell*. O'Reilly, 1996.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, august 1978.

[Jam95] Gosling James et McGilton Henry. The java language environnement. a white paper. ftp://www.javasoft.com/docs/whietepaper.ps, 1995.

[JG96] Bill Joy James Gosling et Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Mic95] Sun Microsystem. The java(tm) language: an overview. ftp://ftp.javasoft.com/docs/java-overview.ps, 1994-1995.