
NFP121, Cnam/Paris

Cours 1

jean-michel Douin, douin au cnam point fr
version : 27 Septembre 2021

Notes de cours

Contenu

- Structure des applications objets avec JAVA
- Aspects impératifs du langage, tests unitaires
- Interfaces, classes et polymorphisme
- Introduction aux patrons de conception (*Design Pattern*)
 - Les fondamentaux Délégation, classe abstraite, interface,
 - Les patrons *Adaptateur* et *Procuration*
- Programmation événementielle
 - Les patrons *Observateur/Observé*, *Stratégie*, *MVC*
- Généricité et Collections
 - Les patrons *Patron de méthode*, *Itérateur*, *Fabrique*
- Les patrons réunis
 - Composite*, *Interpréteur* et *Visiteur*
 - Memento*, *Commande*, ...
- Introspection et réflexivité en Java
- Le patron *Décorateur* et les Entrées/Sorties
 - Invariant, pré-post assertions comme décorations
- Sérialisation XML/JSON et JAVA
- Injection de dépendances et inversion de contrôle
 - Le patron *Fabrique*, *ServiceLocator*
- Les patrons de conception et un outil d'injection de dépendances
 - Vers une séparation de la configuration de l'utilisation

□

Fil conducteur : **Design Pattern**

Prérequis

Connaissances de niveau Bac+2 en informatique.

Objectifs pédagogiques

Maîtriser les concepts avancés de la programmation objet. Apprendre à identifier et utiliser les patrons de conception (*design pattern*) adaptés à une situation de conception. Patrons utilisables quelque soit le langage de programmation.

Compétences

Maîtrise des concepts objets et de plusieurs patrons de conception (Design pattern)

Plan de l'unité, NFP121 Programmation avancée

- L3 en Licence d'Informatique générale
- Des Cours , des Exercices dirigés, des devoirs hebdomadaires
- Cours 2h + 0h30 à 1h Présentation devoirs
- ED 2h centrés sur la réponse aux devoirs et compléments

- Un agenda pour les devoirs
- Un forum d'entraides
- JNEWS un outil d'aide à la réponse attendue
 - Le site <http://jfod.cnam.fr>

Horaires et lieux

- **Horaires**

- 18h15 tous les mardis, à distance via Teams et en présentiel
- 18h30 tous les vendredis, à distance via Teams et en présentiel

énoncés : <http://lmi92.cnam.fr/nfp121/supports/podcast/>

- <http://emploi-du-temps.cnam.fr/emploidutemps2>
 - Une application Android/iPhone

Sommaire du cours 1

- **Introduction**

- Présentation des concepts de l'orienté Objet,
- Java un langage OO : Les objectifs des concepteurs,

- **Outil de développement**

- BlueJ dernière version (jdk11 est inclus)
 - <http://www.bluej.org>
 - <http://jfod.cnam.fr>

Principale bibliographie utilisée pour NFP121

- [Grand00]
 - Patterns in Java le volume 1
- [head First]
 - Head first : <http://www.oreilly.com/catalog/hfdesignpat/#top>
- [DP05]
 - L'extension « Design Pattern » de BlueJ : <http://www.patterncoder.org/>
- [Liskov]
 - Program Development in Java, Abstraction, Specification, and Object-Oriented Design, B.Liskov avec J. Guttag Addison Wesley 2000. ISBN 0-201-65768-6
- [divers]
 - Certains diagrammes UML : <http://www.dofactory.com/net/design-patterns>
 - informations générales <http://www.edlin.org/cs/patterns.html>

Concepts de l'orienté objet

. Un historique ...

- Classe et objet (instance d'une classe)
- État d'un objet et données d'instance
- Comportement d'un objet et méthodes
 - liaison dynamique

- Héritage
- Polymorphisme
 - d'inclusion

Un historique

- **Algorithm + Data Structures = Program**
 - Titre d'un ouvrage de N.Wirth (Pascal, Modula-2, Oberon)

A + d = P langage de type pascal, *années 70*

A + D = P langage modulaire, Ada, modula-2, *années 80*

a + D = P langage Orienté Objet *années 90, simula67...*

$$A + d = P$$

- **surface** (triangle t) =
- **surface** (carré c) =
- **surface** (polygone_régulier p) =
-
- **perimetre** (triangle t) =
- **perimetre** (carré c) =
- **perimetre** (polygone_régulier p) =
-
- *usage : import de la **librairie** de calcul puis*

`carré unCarré; // une variable de type carré`

`y = surface (unCarré)`

$$A + D = P$$

- type carré = structure
- longueurDuCote
- fin_structure;
- >>>-----<<<<

- surface (carré c) =
- perimetre (carré c) =
- (carré c) =

- usage : import du **module carré** puis

```
carré unCarré;     // une variable de type carré
```

```
y = surface ( unCarré)
```

$$a + D = P$$

- classe **Carré** =
- longueurDuCote ...
- **surface** () =
- **perimetre** () =
- () =
- fin_classe;
- *usage : import de la **classe carré** puis*

– **carré unCarré;** *// une instance de la classe Carré*

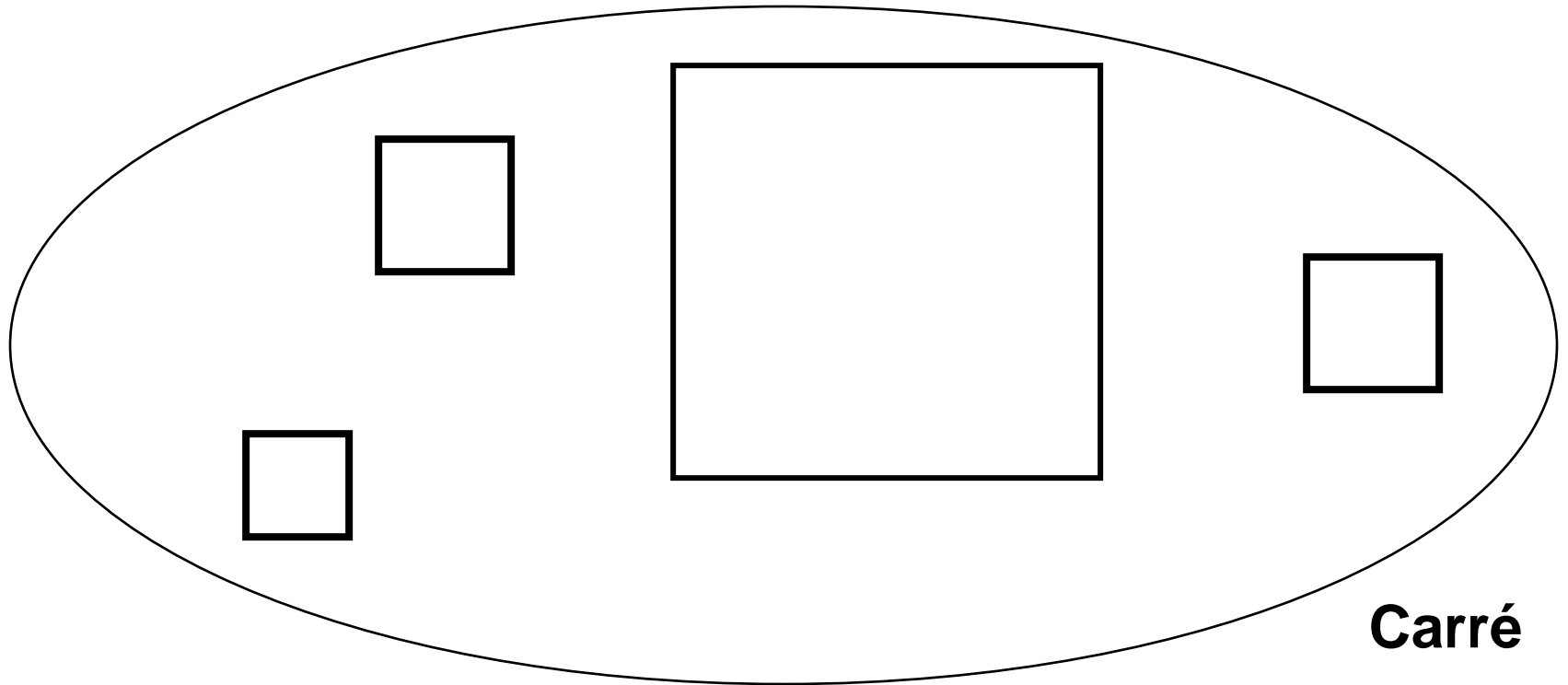
- **y = unCarré.surface ()**



Classe et objet (instance d'une classe)

```
class Carré{
```

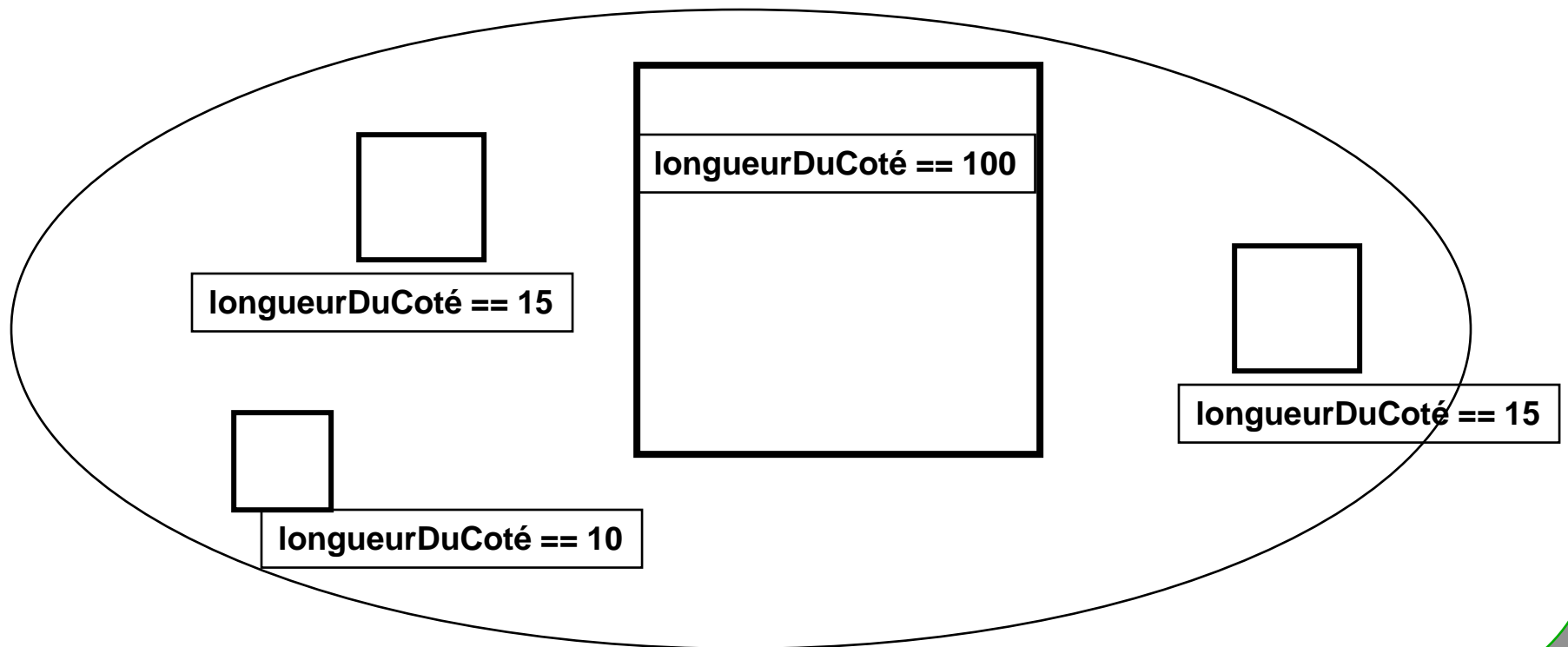
```
}
```



Carré

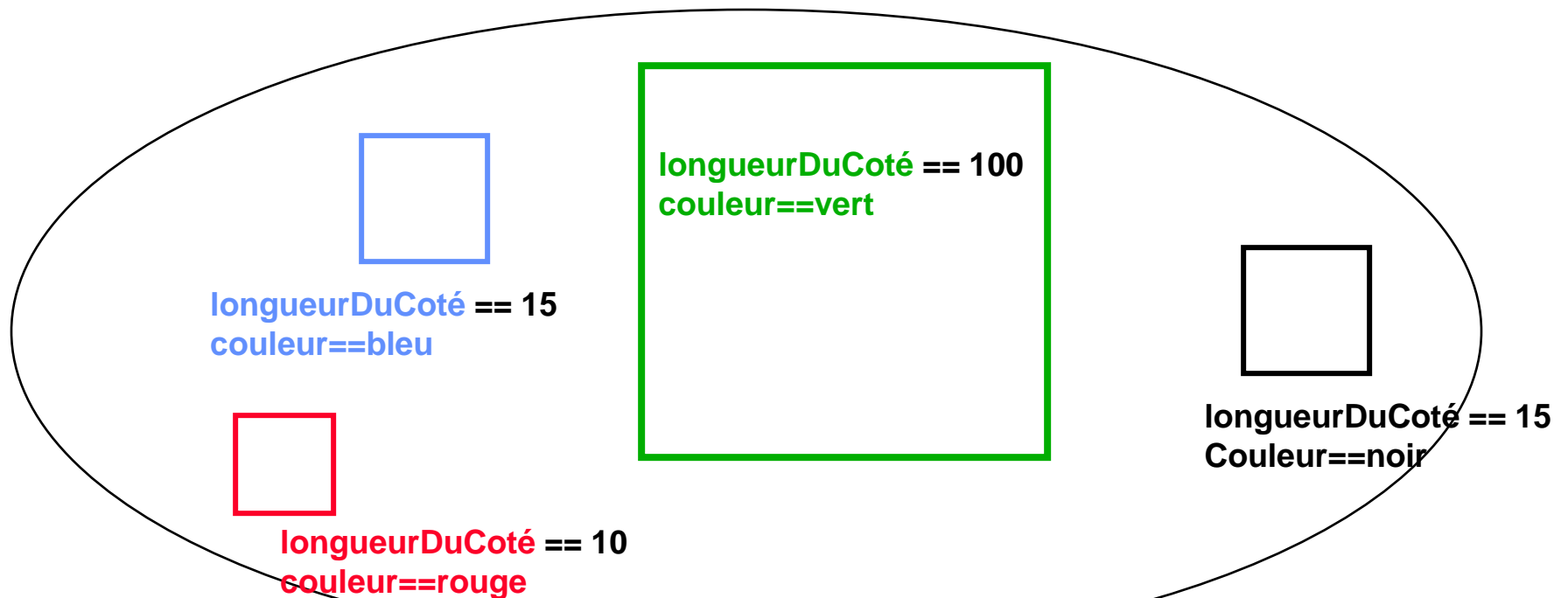
État d'un objet et données d'instance

```
class Carré{  
    int longueurDuCoté;  
}
```



État d'un objet et données d'instance

```
class Carré{  
    int longueurDuCoté;  
    Couleur couleur;  
}
```



Classe et Encapsulation

```
class Carré{  
    private int longueurDuCoté;  
    private Couleur couleur;  
  
}
```

- **contrat avec le client**
 - interface publique, une documentation, le nom des méthodes
 - **implémentation privée**
 - Les données d'instances sont privées

Classe et Encapsulation

```
class Carré{
    private int longueurDuCoté;
    private Couleur couleur;

    /* par convention d'écriture
       get pour les accesseurs, set pour les mutateurs */
    public Couleur getColor(){ .....;}
    public void setColor(Couleur couleur){ .....;}

    /* accesseur et mutateur */
    /* getter and setter */

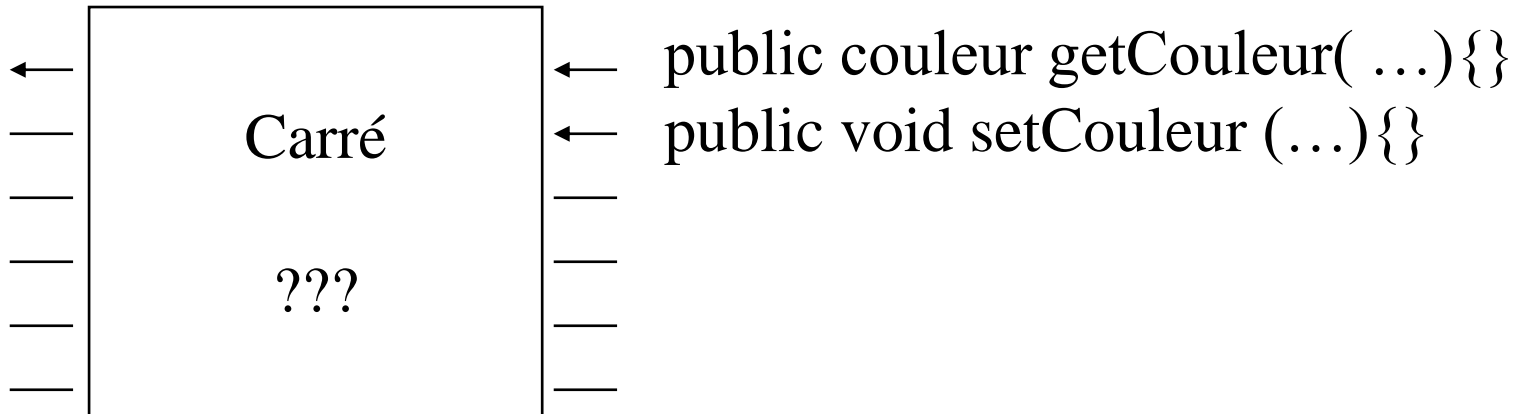
}
```

Classe et Encapsulation

```
class Carré{
```

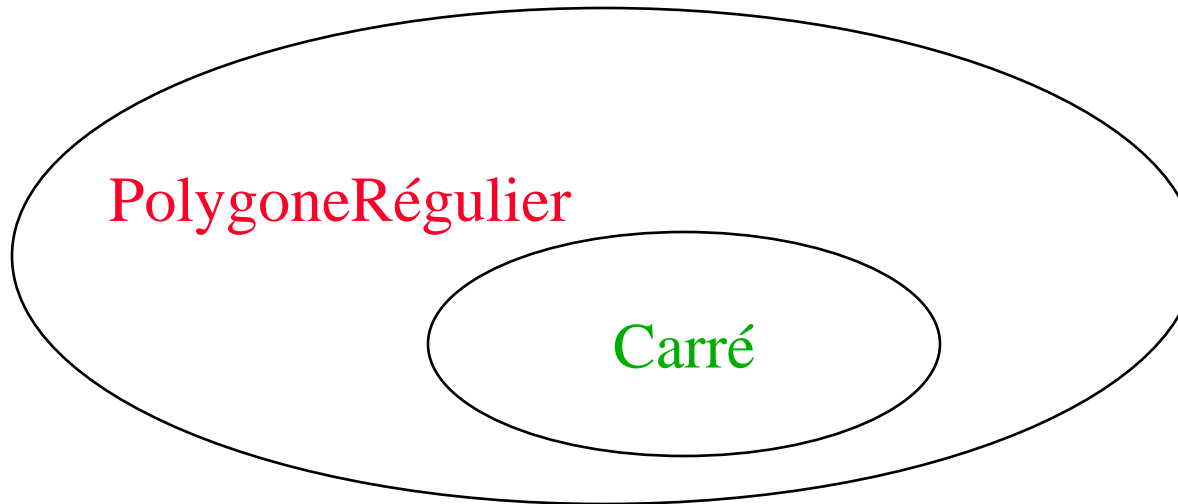
```
private
```

```
// publiques  
public couleur getCouleur( ...){  
public void setCouleur (...){  
  
}
```



Héritage

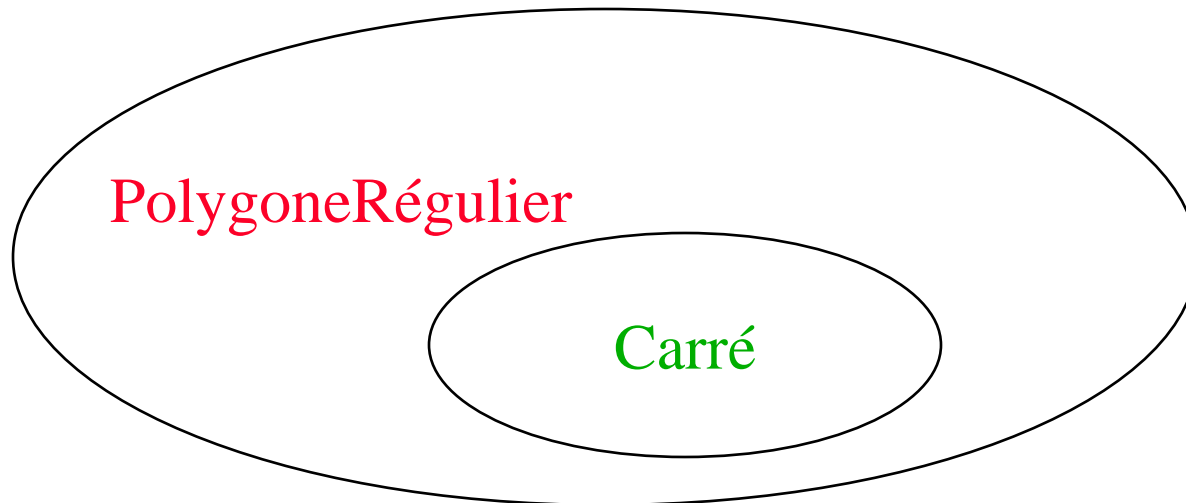
- **Les carrés sont des polygones réguliers**
 - On le savait déjà ...
 - Les carrés héritent des propriétés des polygones réguliers



- un polygone régulier pourrait être un carré

Héritage et classification

- Définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante
 - ajout de nouvelles fonctions
 - ajout de nouvelles données
 - redéfinition de certaines propriétés héritées (masquage)
- Une approche de la classification en langage naturel
- Les carrés **sont** des polygones réguliers (*ce serait l'idéal...*)



La classe PolygoneRegulier

```
class PolygoneRégulier{
```

```
private
```

```
/* accesseurs et mutateurs */
```

```
/* getter and setter */
```

```
/* opérations */
```

```
public int surface(){ ....}
```

```
}
```

La classe Carré est vide...

```
class Carré extends PolygoneRégulier{
```

```
    // vide ...
```

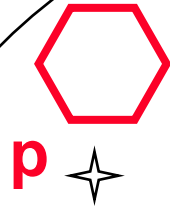
```
}
```

Carré c ... int s = c.surface();

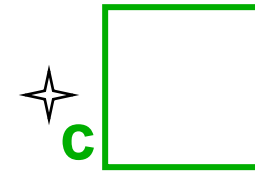
← surface est héritée

Héritage, Les Carrés sont des polygones réguliers

PolygoneRégulier
surface();



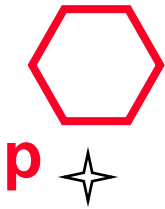
Carré



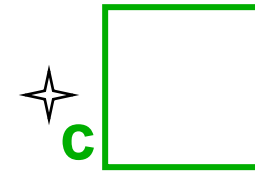
- **PolygoneRégulier p ... int s1 = p.surface();**
- **Carré c ... int s = c.surface();** ← **surface est héritée**

Héritage et redéfinition (masquage)

PolygoneRégulier
surface();



Carré
surface();



- **PolygoneRégulier p ... int s1 = p.surface();**
- **Carré c ... int s = c.surface();**

La classe Carré, le retour

```
class Carré extends PolygoneRégulier{
```



```
public int surface(){ ....}
```

```
}
```

```
Carré c ... int s = c.surface();
```

← surface est redéfinie

Comportement d'un objet et méthodes

PolygoneRégulier
surface();

Carré
surface();



• **PolygoneRégulier p = c;**

• **p.surface();** **???** **surface();** ou **surface();** **???**

Liaison dynamique

- **PolygoneRégulier p = c;**
- **p.surface();** **???** **surface();** ou **surface();** **???**

- Hypothèses

- 1) La classe Carré hérite de la classe PolygoneRégulier
- 2) La méthode surface est redéfinie dans la classe Carré

- **PolygoneRégulier p = c;**
 - **p doit se comporter comme un Carré**

PolygoneRégulier p1 = ... un pentagone ;

- **p = p1** Possible **???** **p.surface() ???**

Démonstration

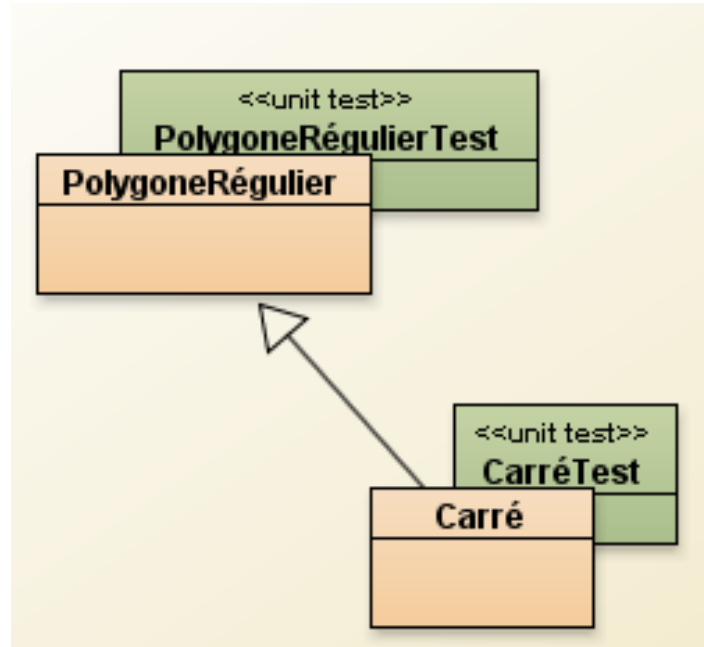
- **BlueJ**

- <http://www.bluej.org/>



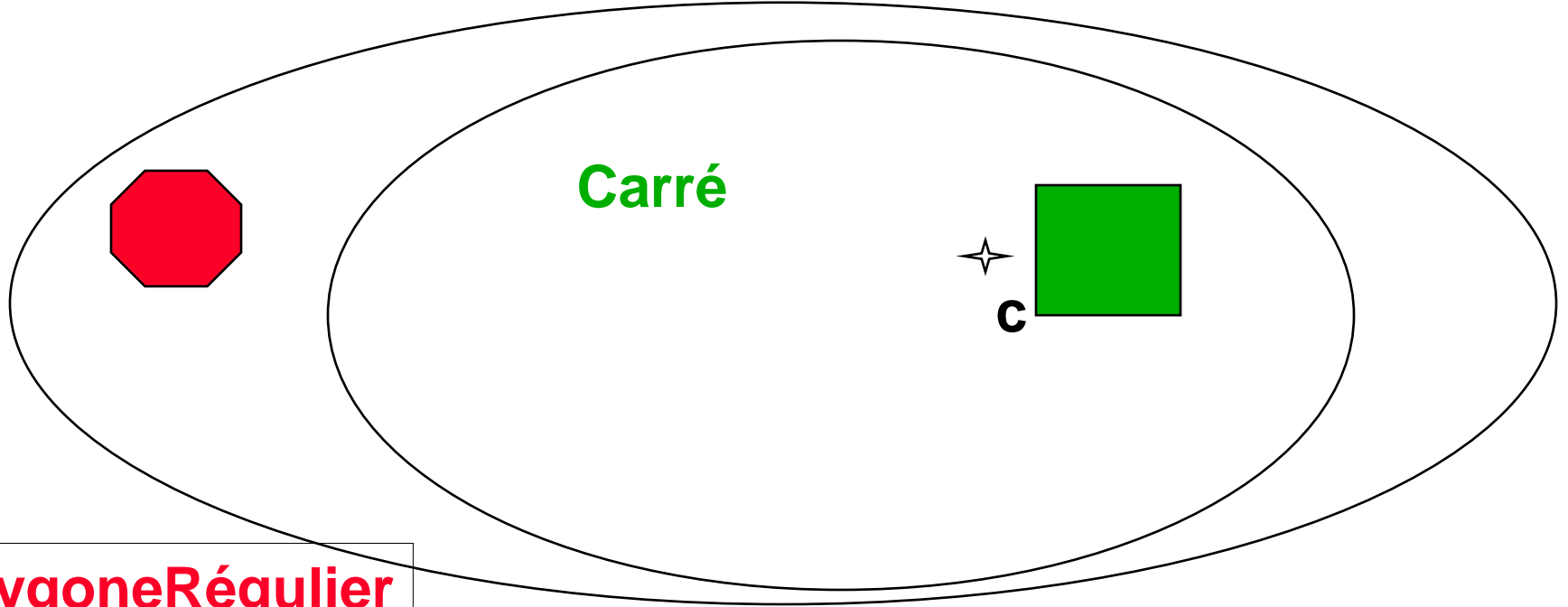
- **Outil avec tests unitaires intégrés**

Démonstration, discussion



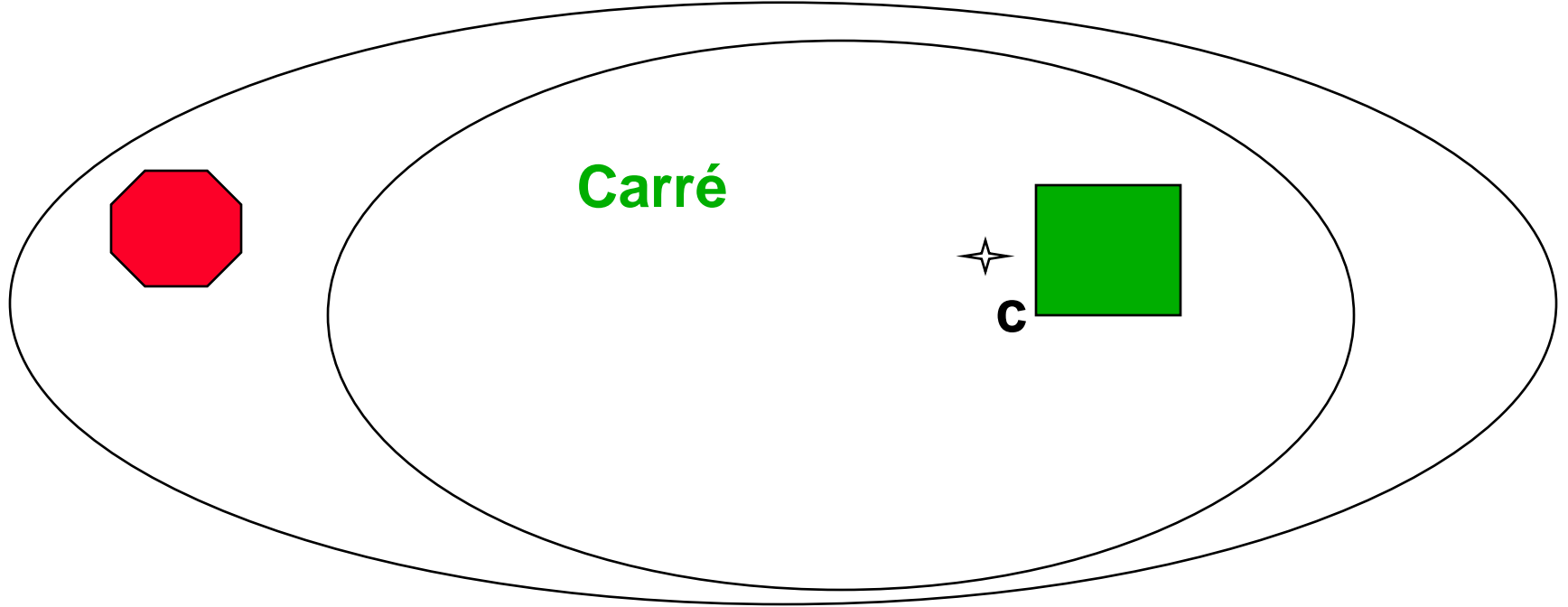
- Tests unitaires en vert

Les carrés sont des polygones, attention



- **Un extrait du cahier des charges**
 - Les carrés sont verts
 - Les polygones réguliers sont rouges

Les carrés sont des polygones, attention



- ? Couleur d'un PolygoneRegulier de 4 côtés ?

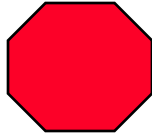
- Discussion :

- Serait-ce une incohérence du cahier des charges ?
- Ou bien un polygone régulier de 4 côtés ne devrait-il pas muter en carré ?
- ...

La plupart des langages à Objets, y compris Java...

PolygoneRégulier

couleur();



Carré

couleur();



c



- Alors en java
- Un PolygoneRégulier de 4 côtés est rouge



- Un Carré est vert
- ???



La suite

- **Discussion**
- **Polymorphisme de plus près**
- **Liaison dynamique**

Polymorphisme : définitions

- **Polymorphisme ad'hoc**
 - **Surcharge(overloading)**,
 - plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est **résolu statiquement** dès la compilation
- **Polymorphisme d'inclusion**
 - **Redéfinition, masquage (overriding)**,
 - est fondé sur la relation d'ordre partiel entre les types, relation induite par l'héritage. si le type B est inférieur selon cette relation au type A alors on peut passer un objet de type B à une méthode qui attend un paramètre de type A, le choix de la méthode est **résolu dynamiquement** en fonction du type de l'objet receveur
- **Polymorphisme paramétrique** ou généricité,
 - consiste à définir un modèle de procédure, ensuite incarné ou instancié avec différents types, ce choix est **résolu statiquement**
 - extrait de M Baudouin-Lafon. La Programmation Orientée Objet. ed. Armand Colin

Polymorphisme ad'hoc

- $3 + 2$ $3.0 + 2.5$ `"bon" + "jour"`
- `out.print(5);` `out.print(5.5);` `out.print("bonjour");`

– le choix de la méthode est **résolu statiquement** à la compilation

– `print(int)` `print(double)` `print(String)`

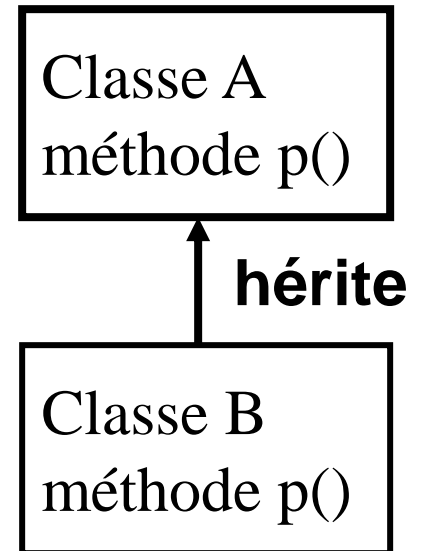
Polymorphisme d'inclusion

- **A** **a = new A();** **a.p();**
- **B** **b = new B();** **b.p();**
- **a = new B();** **a.p();**

```
void m(A a){  
    a.p();  
}
```

```
m(new B());  
m(new A());
```

- B hérite de A, B est inférieur selon cette relation au type A
- *le choix de la méthode est **résolu dynamiquement** en fonction du type de l'objet receveur*



Polymorphisme paramétrique

- Une liste homogène

- **class** Liste<T>{
void add(T t) ...
void remove(T t) ...
...
}

```
Liste<Integer> li = new Liste<Integer>();  
li.add(new Integer(4));
```

```
Liste<A> la = new Liste<A>();  
la.add(new A());  
la.add(new B()); // B hérite de A
```

- incarné ou instancié avec différents types, ce choix est **résolu statiquement**

Affectation polymorphe

- **Création d'instances**

- Carre c1 = new Carre(100);
- Carre c2 = new Carre(10);
- PolygoneRegulier p1 = new PolygoneRegulier(4,100);

- **Affectation**

- c1 = c2; *// synonymie : c1 est un autre nom pour c2*
 // c1 et c2 désignent le même carré de longueur 10

- **Affectation polymorphe**

- p1 = c1;

- **Affectation et changement de classe**

- c1 = (Carre) p1; *// Hum, Hum ... Hasardeux, levée d'exception possible*

- *if (p1 instanceof Carre) c1 = (Carre)p1; // mieux, beaucoup mieux ...*

Liaison dynamique

- Sélection de la méthode en fonction de l'objet receveur
- **déclaré** / **constaté** à l'exécution

- **PolygoneRegulier p1 = new PolygoneRegulier(5,100);**
*// p1 **déclarée** PolygoneRegulier*
- **Carre c1 = new Carre(100);**

- **int s = p1.surface();** *// p1 **constatée** PolygoneRegulier*

- **p1 = c1;** *// affectation polymorphe*

- **s = p1.surface();** *// p1 **constatée** Carre*

- Note importante : la recherche de la méthode s'effectue uniquement dans l'ensemble des méthodes masquées associé à la classe dérivée
 - Rappel : Dans une classe dérivée, la méthode est masquée seulement si elle possède **exactement** la même signature

Selon Liskov cf. Sidebar2.4, page 27

- The **apparent type** of a variable is the type understood by the compiler from information available in declarations. The **actual type** of an Object is its real type -> the type it receives when it is created.

Ces notes de cours utilisent

- *type déclaré pour **apparent type** et*
- *type constaté pour **actual type***

Java un langage à Objets

- **Sommaire**
 - Généralités
 - Démonstration
 - Une classe
 - Tests unitaires

Java : les objectifs

- **« Simple »**
 - syntaxe " C «
- **« sûr »**
 - pas de pointeurs, vérification du code à l'exécution et des accès réseau et/ou fichiers
- **Orienté Objet**
 - (et seulement !), pas de variables ni de fonctions globales, types primitifs et objet
- **Robuste**
 - ramasse miettes, fortement typé, gestion des exceptions
- **Indépendant d'une architecture**
 - Portabilité assurée par la présence d'un interpréteur de bytecode sur chaque machine
- **Environnement riche**
 - Classes pour l'accès Internet
 - classes standard complètes
 - fonctions graphiques évoluées
- **Technologie « Transversale »**

Simple : syntaxe apparentée C,C++

```
public class Num{  
  
    public static int max( int x, int y){  
        int max = y;  
        if(x > y){  
            max = x;  
        }  
        return max;  
    }  
}
```

Fichier Num.java

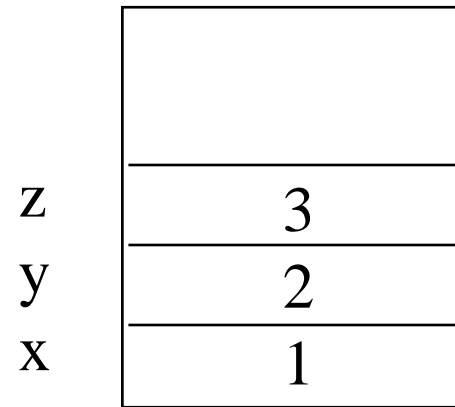
Note : C# apparenté Java

Sûr par l'absence de pointeurs (accessibles au programmeur)

- Deux types : primitif ou **Object** (et tous ses dérivés)

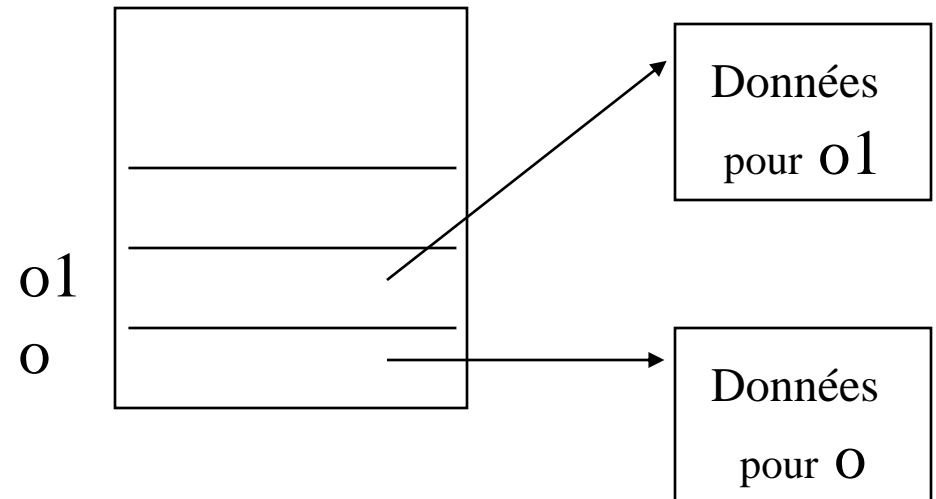
- **primitif** :

- int x = 1;
- int y = 2;
- int z = 3;



- **Object**

- Object o = new Object();
- Object o1 = new Object();



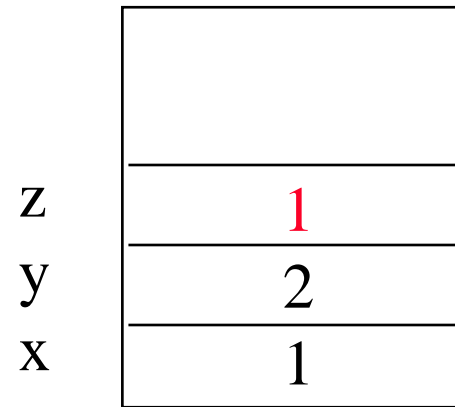
Sûr par l'absence de pointeurs, attention à la sémantique du =

- Deux types : primitif ou **Object** (et tous ses dérivés)

- primitif :

- int x = 1;
- int y = 2;
- int z = 3;

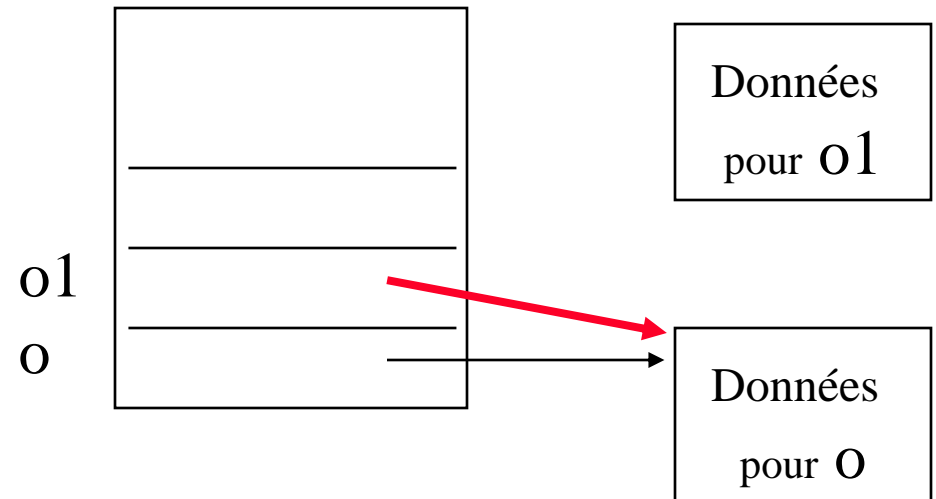
- **z = x;**



- **Object**

- Object o = new Object();
- Object o1 = new Object();

- **o1 = o;**



Robuste

- **Ramasse miettes ou gestionnaire de la mémoire**
 - Contrairement à l'allocation des objets, leur dé-allocation n'est pas à la charge du programmeur
 - Ces dé-allocations interviennent selon la stratégie du gestionnaire
- **Fortement typé**
 - Pas d'erreur à l'exécution due à une erreur de type
 - Mais un changement de type *hasardeux* est toujours possible...
- **Généricité**
 - Vérification statique, à la compilation, du bon « typage »
- **Exceptions**
 - Mécanisme de traitements des erreurs,
 - Une application ne devrait pas s'arrêter à la suite d'une erreur,
 - (ou toutes les erreurs possibles devraient être prises en compte ...)

Portable

Le source Java
Num.java

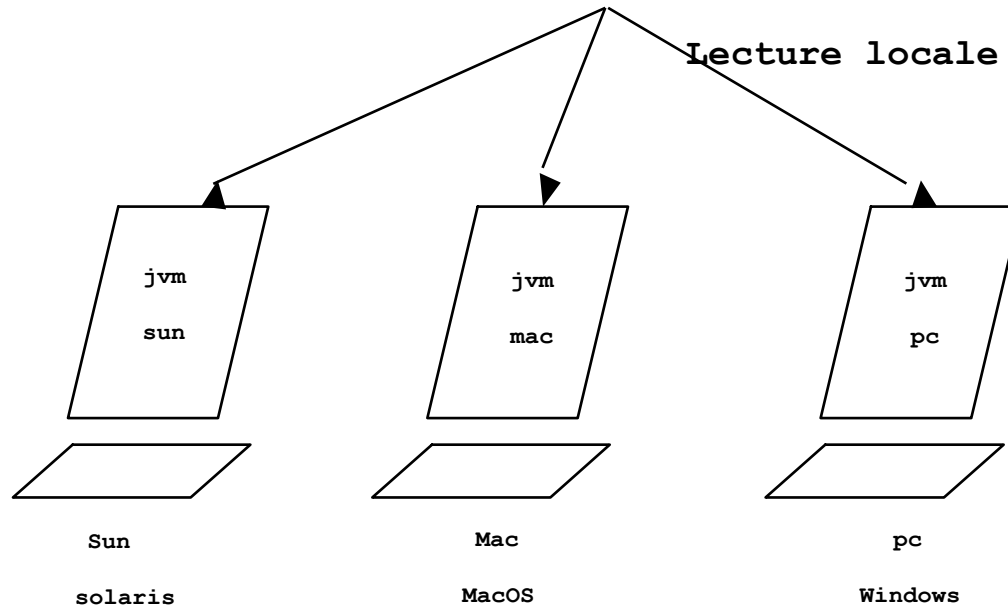
```
public class Num {  
...  
}
```

1) **compilation**

Le fichier compilé
Num.class

```
1100 1010 1111 1110 1011 1010 1011 1110  
0000 0011 0001 1101 .....
```

Lecture locale ou distante du fichier

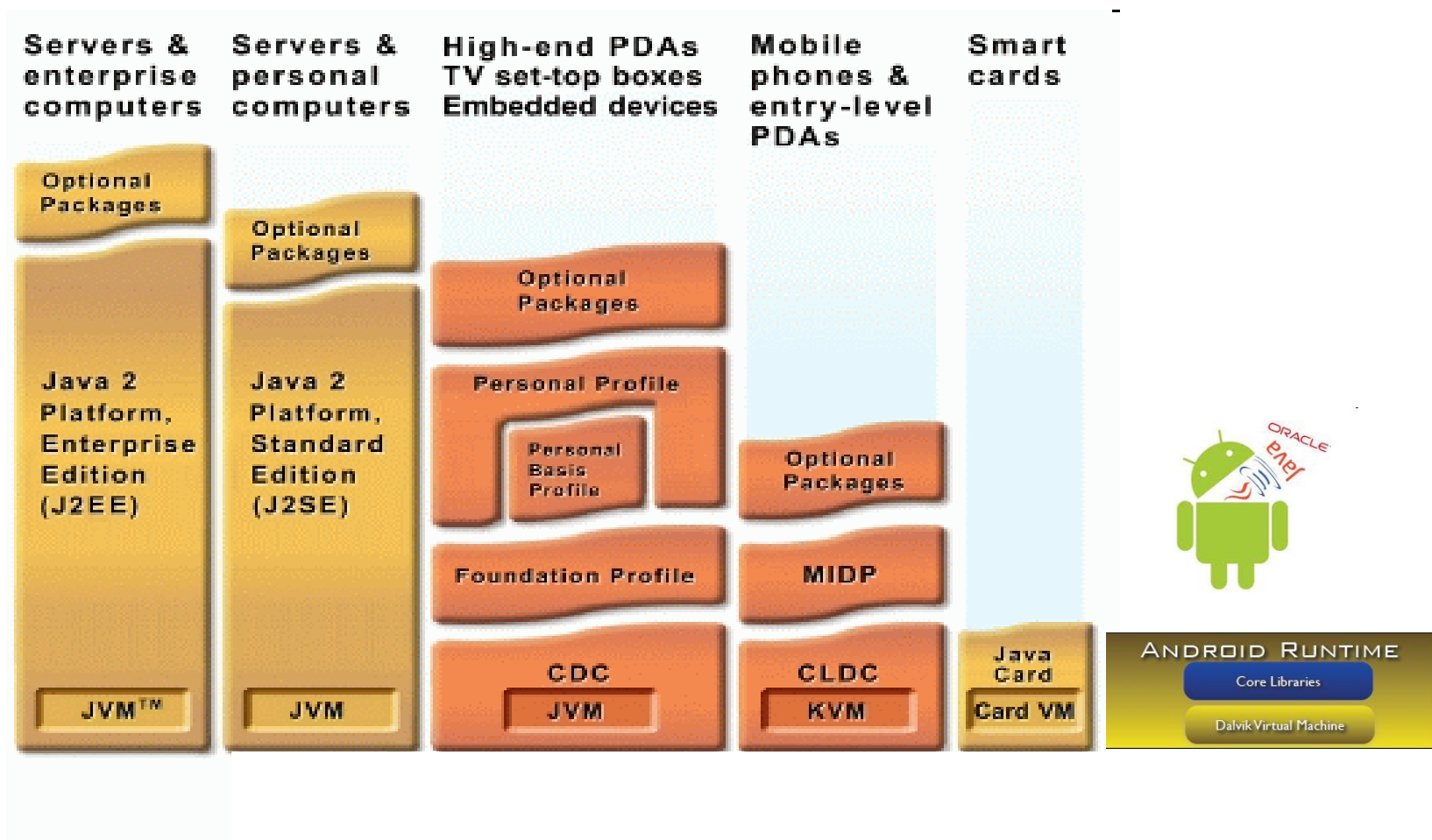


2) **interprétation**
/ exécution

Environnement (très) riche

- **java.applet**
 - **java.awt**
 - **java.beans**
 - **java.io**
 - **java.lang**
 - **java.math**
 - **java.net**
 - **java.rmi**
 - **java.security**
 - **java.sql**
 - **java.text**
 - **java.util**
 - **javax.accessibility**
 - **javax.swing**
 - **org.omg.CORBA**
 - **org.omg.CosNaming**
-
- **Liste des principaux paquetages de la plate-forme JDK 1.2 soit environ 1500 classes !!! Et bien d'autres A.P.I. JSDK, JINI, ...**
 - **le JDK1.3/1850 classes,**
 - **Le JDK1.5 ou j2SE5.0 3260 classes**
 - **Le J2SE 1.6, 1.8**

Technologie « Transversale »



Source : <http://java.sun.com/javame/technology/index.jsp>

Pause ... enfin presque

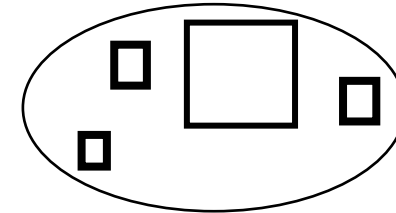
- **Quelques « rappels » de Syntaxe**
 - En direct

- **Démonstration**
 - BlueJ

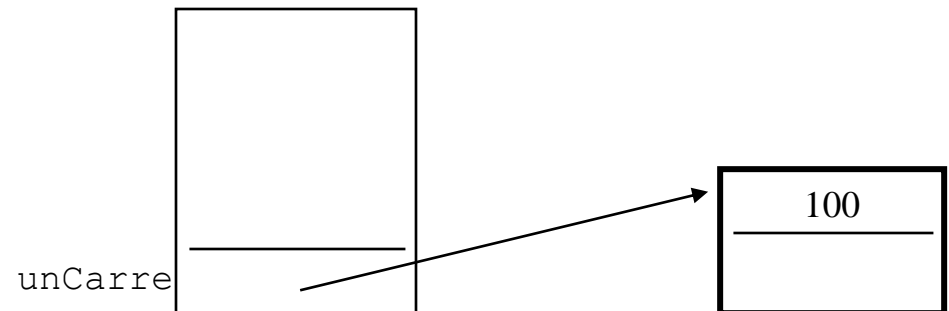
- **Présentation de JNEWS**
 - Un outil d'aide à la réponse attendue développé au Cnam

Un exemple en Java : la classe Carre

```
public class Carre {  
    private int longueurDuCote;  
  
    public void initialiser(int longueur){  
        longueurDuCote = longueur;  
    }  
  
    public int surface(){  
        return longueurDuCote * longueurDuCote;  
    }  
  
    public int perimetre(){  
        return 4*longueurDuCote;  
    }  
}
```

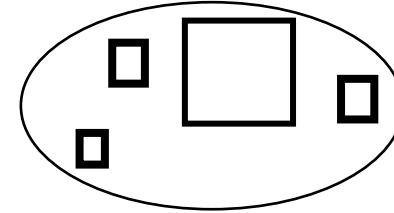


```
// un usage de cette classe  
Carre unCarre = new Carre();  
unCarre.initialiser(100);  
int y = unCarre.surface();
```



la classe Carre avec un constructeur

```
public class Carre {  
    private int longueurDuCote;  
  
    public Carre (int longueur){  
        longueurDuCote = longueur;  
    }  
  
    public int surface(){  
        return longueurDuCote * longueurDuCote;  
    }  
  
    public int perimetre(){  
        return 4*longueurDuCote;  
    }  
}
```

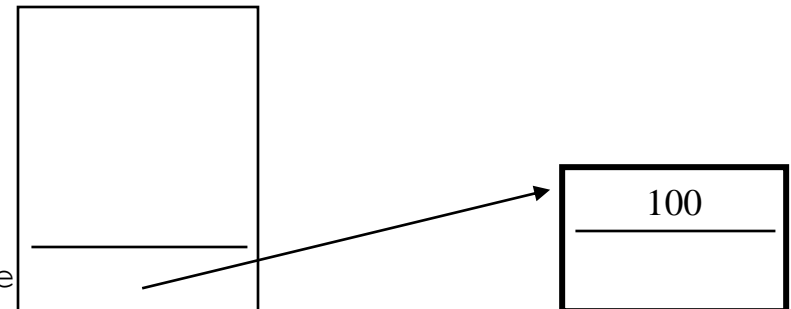


// un usage de cette classe

```
Carre unCarre = new Carre(100);
```

```
int y = unCarre.surface();
```

unCarre



La classe PolygoneRégulier

```
public class PolygoneRegulier {
    private int nombreDeCotes;
    private int longueurDuCote;

    public PolygoneRegulier(int nombreDeCotes, int longueurDuCote) {
        this.nombreDeCotes = nombreDeCotes;
        this.longueurDuCote = longueurDuCote;
    }

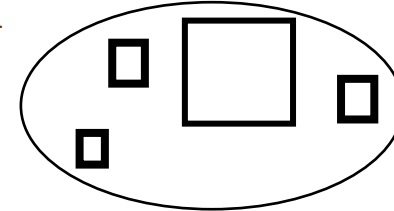
    public int perimetre() {
        return nombreDeCotes * longueurDuCote;
    }

    public int surface() {
        return (int) (1 / 4.0 * nombreDeCotes * Math.pow(longueurDuCote, 2.0)
            * cotg(Math.PI / nombreDeCotes));
    }

    public int longueurDuCote(){ return longueurDuCote; }
    private static double cotg(double x) {
        return Math.cos(x) / Math.sin(x);
    }
}
```

la classe Carre extends PolygoneRégulier

```
public class Carre extends PolygoneRegulier{
```



```
    public Carre (int longueur){
```

```
        super(4, longueur);
```

```
    }
```

```
    // redéfinition ... @Override
```

```
    public int surface() {
```

```
        return longueurDuCote() * longueurDuCote();
```

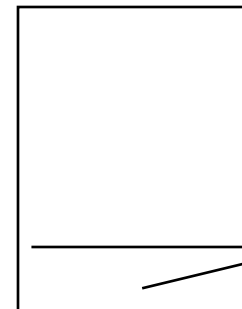
```
    }
```

```
}
```

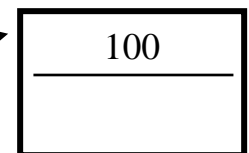
```
// un usage de cette classe
```

```
Carre unCarre = new Carre(100);
```

```
int y = unCarre.surface();
```



unCarre



- **Hiérarchie**
- Java supports *type hierarchy*, in which one type can be the **supertype** of other types, which are its **subtypes**. A subtype's objects have all the methods defined by the supertype.
- **Object la racine de toute classe**
- All objects type are subtypes of **Object**, which is the top of the type hierarchy. **Object** defines a number of methods, including `equals` and `toString`. Every object is guaranteed to have these methods.

Démonstration

- **Outil Bluej**

- La classe Carré
- Instances inspectées

- **Tests Unitaires**

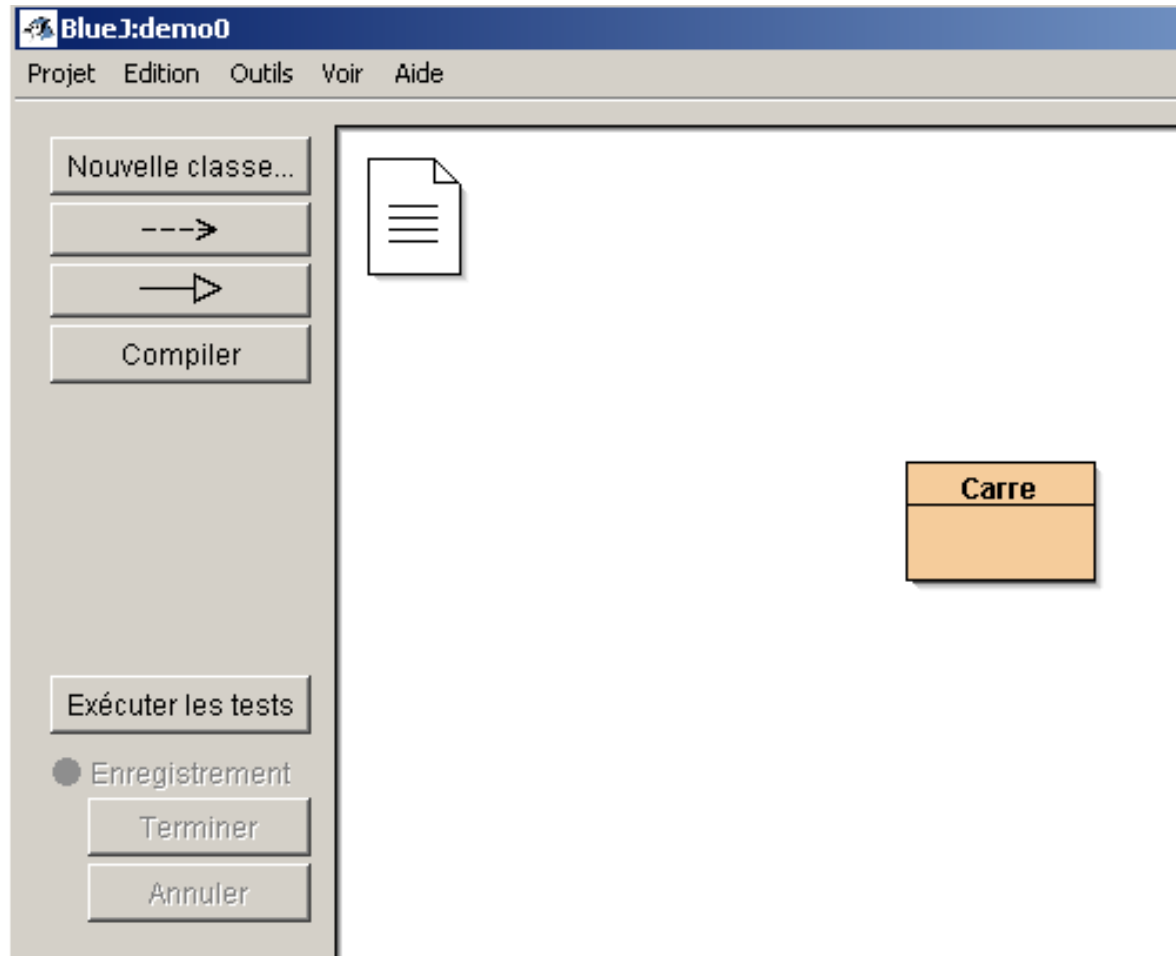
- La Classe CarréTest

- Ou la mise en place d'assertions pertinentes
- En assurant ainsi des tests de non régression,
 - « vérifier » que le système ne se dégrade pas à chaque modification...
- Augmenter *le taux de confiance* envers le code de cette classe ...
 - Très informel, et comment obtenir des tests pertinents ?

- **Tests Unitaires « référents » : Outil JNEWS**

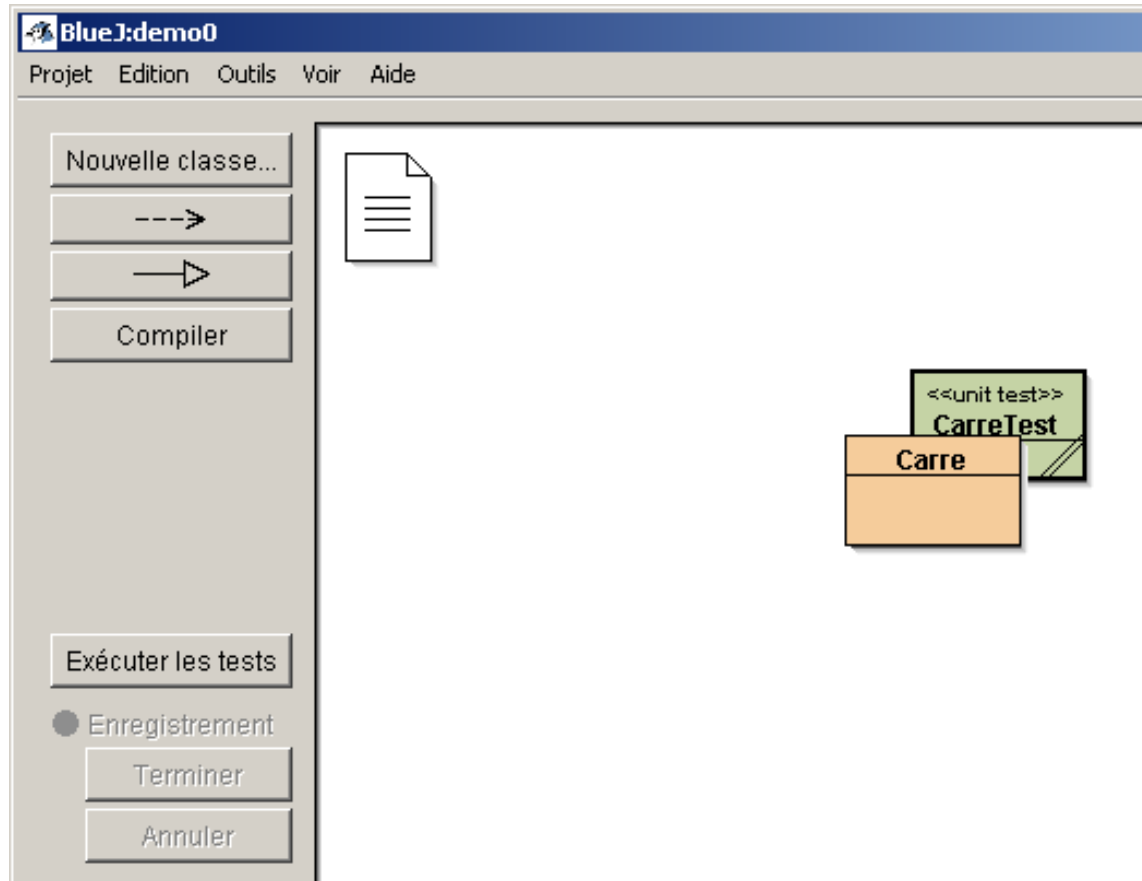
- Java New Evaluation Web System

Demo : Bluej



- **Instances et leur inspection**

Demo : Bluej + tests unitaires



- **Test unitaires depuis BlueJ ou en source**

Tests unitaires : outil *junit* intégré

- www.junit.org
- <http://junit.sourceforge.net/javadoc/junit/framework/Assert.html>

- **Un exemple :**

```
public class CarreTest extends junit.framework.TestCase{

    public void testDuPerimetre(){
        Carre c = new Carre();
        c.initialiser(10);
        assertEquals(" périmètre incorrect ???" ,40, c.perimetre());
    }
}
```

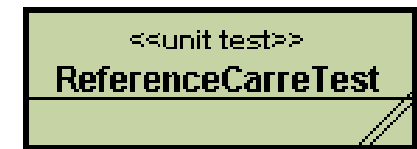
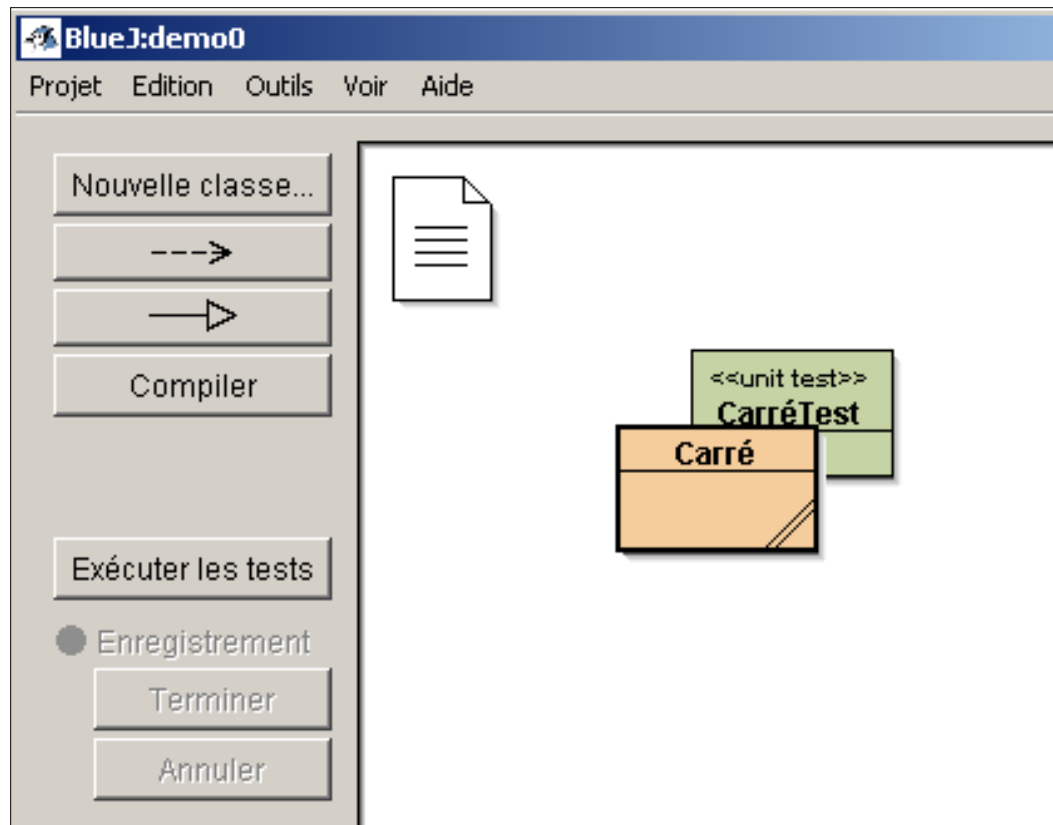
Assertions

```
assertEquals(attendu, effectif);  
assertEquals(" un commentaire ???", attendu, effectif);  
  
assertSame(" un commentaire ???", attendu, effectif);  
assertTrue(" un commentaire ???", expression booléenne);  
assertFalse(" un commentaire ???", expression booléenne);  
assertNotNull(" un commentaire ???", une référence);  
...
```

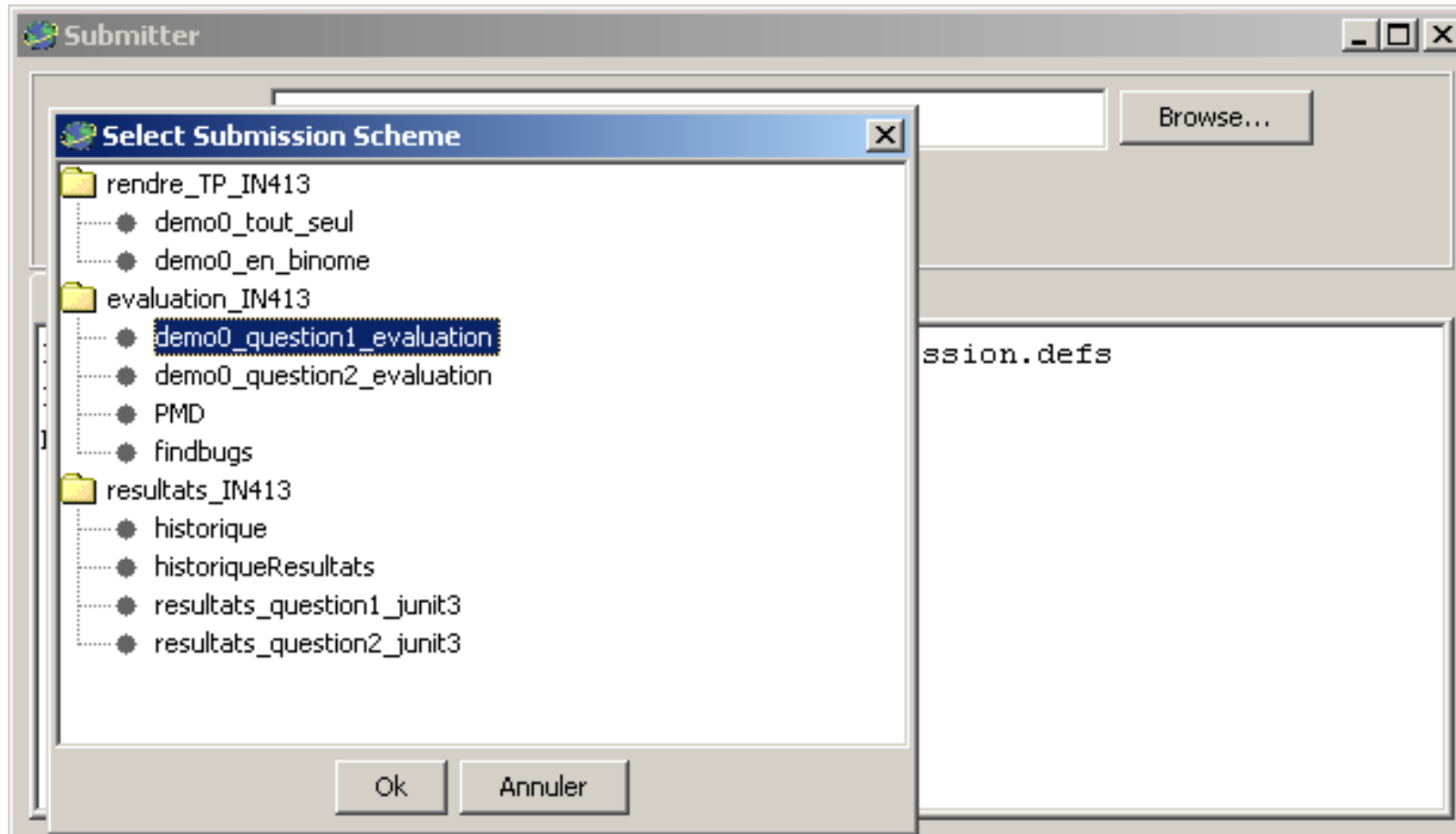
Le *commentaire* est affiché lorsque l'assertion échoue

JNEWS contient des Tests unitaires distants

- **Tests unitaires distants et référents** *(ce qui est attendu...)*



JNEWS : outil *Submitter*... intégré



- **Un clic suffit**

Démonstration : JNEWS une réponse : Bravo!!

Submitter

Scheme Browse...

Submit Annuler

Status Log

Loading H:\IN413_IN4A21\IN413_0708\demo0\submission.defs
Loading Done

Submission Result

M/Mme/Melle **test** (IN413, demo0q1) le 03-09-07 à 14:47

Résultat : **0 échec** aux **2 tests¹** référents

Bravo !!

(0/6)²

site : CEP. Au demo0q1, à ce jour : **1 succès / 1 utilisateur.**

*JNEWS_Cnam, pour Java New Evaluation Web Services est un service d'aide au développement attendu.
Vos travaux sont évalués par l'exécution de classes de tests référents sur le serveur <http://jfofod.cnam.fr/jnews/>, ces classes de tests utilisent ici junit3, tout comme votre environnement bluej.
Le nombre d'échecs aux tests référents représente le résultat de l'évaluation et un historique vous rappelle les derniers résultats obtenus.
Un historique détaillé est également accessible depuis BlueJ, (menu Outils>Item submit puis Browse et JNEWS_IN413/historique)
JNEWS_Cnam est bien entendu perfectible, n'hésitez pas à nous envoyer vos critiques, souhaits d'améliorations, mesures supplémentaires, etc...*

Bonnes soumissions avec JNEWS!

[JNEWS_Cnam](#), Java New Evaluation Web Services du Cnam.

¹ Un test comporte plusieurs assertions, une assertion fausse induit un échec.
² Soit le (nombre d'essais antérieur / nombre maximal de tentatives).

+++ notez qu'il est préférable de fermer cette fenêtre par la case [X] en haut à droite plutôt que d'utiliser le bouton Ok...+++

Ok

JNEWS au Cnam

- Une aide à la réponse attendue
 - +
 - Outils en ligne comme PMD , findbugs , checkstyle,...
 - +
 - Remise planifiée d'un rapport de TP et des sources correctement documentées
-
- <http://jfod.cnam.fr/jnews/>

Conclusion

Annexe

- c.f. http://jfod.cnam.fr/progAvancee/sources_ED_Cours/