IHM

Rappels et Compléments Java

Frédéric MOAL
Université d'Orléans

Master M1 MIAGE 2008/2009

Sommaire

- Introduction
  - Nouveautés du JDK 5/6
- Rappels :
  - Rappels divers sur les classes
  - Héritage
  - Classes abstraites
  - Interfaces
- Compléments :
  - Package
  - Assertions
  - Tests unitaires

Ressources

- **Le site officiel de Sun** http://java.sun.com
- **Documentation SUN de l'API Java :**
  http://java.sun.com/apis.html (java –version, 1.4 / 5 / 6...)
- **Le tutorial SUN sur java :**
  http://java.sun.com/docs/books/tutorial/
- **Livres (vérifiez JDK 5/6 !)**
  « Java in Nutshell », O'Reilly
  « Java - la synthèse », 4ème édition, Clavel…
  « Penser en Java », v3,
  http://bruce-eckel.developpez.com/livres/java/traduction/tij3/
- **Cours en ligne:**
  Richard GRIN, Université de Nice
  http://deptinfo.unice.fr/~grin/messupports/index.html
- **Indispensable !**
  http://www.developpez.com, l'onglet java
  Jean Michel DOUDOUX - "Développons en Java" (1300 pages)
  http://www.jmdoudoux.fr/

Introduction

3 éditions de Java

- Java SE : Java 2 Standard Edition ; JDK = *J2SE Development Kit*, aussi appelé SDK (*Software Development Kit*) pour certaines versions 1.0, 1.1, 1.2 (Java2), 1.3, 1.4, 5, 6, 7
- Java EE : Enterprise Edition, pour serveur avec servlet, JSP, EJB, WS…
- Java ME : Micro Edition, version allégée pour écrire des programmes embarqués (cartes à puce/Java card, téléphones portables,…)

- [JRE] : Java Runtime Environnement

JDK 5.0+

Plusieurs “grandes” nouveautés dans Java 5

- Types énumérés
- StringBuilder
- Boxing/Unboxing
- For each
- Nombre variable d’arguments
- Import static
- printf
- Annotations
- Généricité

Types énumérés

- définir un nouveau type en énumérant toutes ses valeurs possibles (par convention, les valeurs sont en majuscules)
- Plus sûrs que d'utiliser des entiers pour coder les différentes valeurs du type (vérifications à la compilation)
- Utilisés comme tous les autres types

Énumération « interne »

- On peut définir une énumération à l'intérieur d'une classe (elle est implicitement **static**) :

```
public class Carte {
  public enum Couleur
   {TREFLE, CARREAU, COEUR, PIQUE};
  private Couleur couleur;
  . . .
  this.couleur = Couleur.PIQUE;
```

- Depuis une autre classe :

```
carte.setCouleur(Carte.Couleur.TREFLE);
```

Énumération « externe »

- les types énumérés : classes qui héritent de la classe **java.lang.Enum**
- Comme les classes, peuvent être définis indépendamment d'une classe
- Le fichier qui contient une énumération publique doit avoir le nom de l'énumération avec le suffixe « .java »

```
public enum CouleurCarte {
   TREFLE, CARREAU, COEUR, PIQUE;
}
public class Carte {
   private CouleurCarte couleur;
   ...
}
```

Classe **Enum**

- Classe mère des énumérations

- **Enum** implémente **Comparable** et **Serializable** et contient (entre autres) les méthodes publiques :
  - **int ordinal()** retourne le numéro de la valeur (en commençant à 0)
  - **String name()**
  - **static valueOf(String)**

Les valeurs

- **toString()** retourne le nom de la valeur sous forme de **String** ; par exemple, "TREFLE“
- **static valueOf(String)** renvoie la valeur de l’énumération correspondant à la **String**
- En plus, toutes les énumérations ont une méthode **static values()** qui retourne un tableau contenant les valeurs de l’énumération le type du tableau est l’énumération

Utilisable avec == et switch


```
CouleurFeux couleurFeux;
...
if(couleurFeux == CouleurFeux.VERT))
...
switch(couleurFeux) {
   case CouleurFeux.ROUGE:
     stop();
     break;
   case CouleurFeux.VERT:
     passer();
     break;
   default: ralentir();
}
```

Enum

on peut ajouter des méthodes et constructeurs dans un type énuméré :

```
public enum Couleur {
  TREFLE("Trèfle"), CARREAU("Carreau"),
  COEUR("Coeur"), PIQUE("Pique");
  private String couleur;
  Couleur(String couleur) {
   this.couleur = couleur;
  }
  public String toString() {
   return couleur;
  }
}
```

StringBuilder

- **StringBuilder** a été introduite par le JDK 5.0
- Mêmes fonctionnalités et noms de méthodes que **StringBuffer** mais ne peut être utilisé que par un seul *thread* (pas de protection contre les problèmes liés aux accès multiples)
- **StringBuilder** fournit de meilleures performances que **StringBuffer**

Boxing/unboxing

- Le « *boxing* » (mise en boîte) automatise le passage entre les types primitifs et les classes qui les enveloppent
- L'opération inverse s'appelle « *unboxing* »
- Le code v1.4 :

```
liste.add(new Integer(89));
int i = liste.get(n).intValue();
```

peut maintenant s'écrire en v5.0 :

```
liste.add(89);
int i = liste.get(n);
```

For each

- La syntaxe est plus simple/lisible qu'une boucle *for* ordinaire
- Attention, on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)
- Encore plus utile pour parcours des « collections »

```
String[] noms = new String[50];
...
// Lire « pour chaque nom dans (:) noms »
for (String nom : noms) {
 System.out.println(nom);
}
```

Nombre variable d'arguments

- Dernier paramètre dont le type a le suffixe « **...** » **(String..., int...)**
- Un seul paramètre de ce type est autorisé par méthode
- Le compilateur traduit ce type spécial par un type tableau :

  m(int p1, String... params)

  → **m(int p1, String[ ] params)**
- Utilisation de **params** comme un tableau (**for**, affectation, etc.)

Nombre variable d'arguments

- Exemple :

```
public static int max(int... valeurs) {
  if (valeurs.length == 0) throws new
  IllegalArgumentException("Au moins 1
  valeur requise");
  int max = Integer.MIN_VALUE;
  for (int i : valeurs) {
   if (i > max) max = i;
  }
  return max;
}
```

Importer des constantes **static**

- on peut importer des constantes ou méthodes statiques d'une classe ou d'une interface
- Code allégé, par exemple pour l'utilisation des fonctions mathématiques de la classe **java.lang.Math**

```
import static java.lang.Math.*;
import java.util.*;
public class Machin {
…
x = max(sqrt(abs(y), y+2));
```

- On peut importer une seule constante ou méthode :

```
import static java.lang.Math.PI;
x = 2 * PI;
```

Le printf de C

- Avec l'actuelle méthode *println()* :

```
double a = 5.6d ;
double b = 2d ;
double mult = a * b ;
System.out.println(a + " mult by " + b
 + " equals " + mult);
```

- Avec le nouveau printf sorties formatées :

```
System.out.printf("%3.2lf mult by
 %3.2lf equals %3.2lf\n", a, b ,
 mult);
```

Annotations

- Possibilité d'annoté le code source (méta programmation)
- équivalent des « attributs » que propose le framework .NET de Microsoft

- Exemple : méthodes qui redéfinissent une méthode d'une classe ancêtre

```
@override
public Dimension getPreferredSize() {
```

Généricité

- permet de paramétrer une classe avec un ou plusieurs types de données
- eg le type des éléments d'un **Arraylist** :

  ArrayList<E>

  E est un paramètre formel de type
- Peut remplacé par un argument de type concret pour typer des expressions ou créer des objets :

```
ArrayList<Integer> l = new
        ArrayList<Integer>();
```

Généricité : exemple


```
List<Employe> employes =
  new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
. . .
for(int i=0;i<employes.size();i++) {
  System.out.println(
  employes.get(i).getNom());
}
```

Rappels divers sur les classes

Types d'autorisation d'accès

- **private** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **public** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès

this

“ this ”

- Le code d’une méthode d’instance désigne
  – l’instance qui a reçu le message (l’instance courante), par le mot-clé **this**
  – donc, les membres de l’instance courante en les préfixant par « **this.** »

- Lorsqu’il n’y a pas d'ambiguïté, **this** est optionnel pour désigner un membre de l’instance courante

Granularité de la protection

- En Java, la protection des attributs se fait classe par classe, et pas objet par objet
- Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés

Opérateur **instanceof**

- La syntaxe est :

  objet **instanceof** *nomClasse*

  Exemple : **if (x instanceof Livre)**
- Le résultat est un booléen :
  - **true** si l'objet est une instance de la classe
  - **false** sinon
- On verra des compléments sur **instanceof** dans la

  partie sur l'héritage

Variable d'état final

- Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances
- Une variable d'instance **final** peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs
- Une variable de classe **final** peut ne pas être initialisée à sa déclaration mais elle doit alors recevoir sa valeur dans un bloc d'initialisation **static**

Variables de classe

- Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur **static**)
- Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

Exemple de variable de classe


```
public class Employe {
  private String nom, prenom;
  private double salaire;
  private static int nbEmployes = 0;
  // Constructeur
  public Employe(String n, String p) {
        nom = n;
        prenom = p;
        nbEmployes++;
  }
  . . .
}
```

Méthodes de classe

- Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe
- Une méthode de classe peut être considérée comme un message envoyé à une classe

Exemple :

```
public static int nbEmployes() {
  return nbEmployes;
}
```

Désigner une méthode de classe

- Pour désigner une méthode **static** depuis une autre classe, on la préfixe par le nom de la classe :

```
int n = Employe.nbEmploye();
```

- On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est **static**) :

```
int n = e1.nbEmploye();
```

Méthodes de classe

- Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (**this**)
- Il serait, par exemple, interdit d'écrire

```
static double tripleSalaire() {
    return salaire * 3;
}
```

- Une méthode de classe ne peut avoir la même signature qu'une méthode d'instance

Question

La méthode **main()** est nécessairement **static**

- Pourquoi ?

Question

La méthode **main()** est nécessairement **static**

- Pourquoi ?

→ La méthode **main()** est exécutée au début du programme.

Aucune instance n'est donc déjà créée lorsque la méthode **main()** commence son exécution. Ça ne peut donc pas être une méthode d'instance.

Blocs d'initialisation **static**

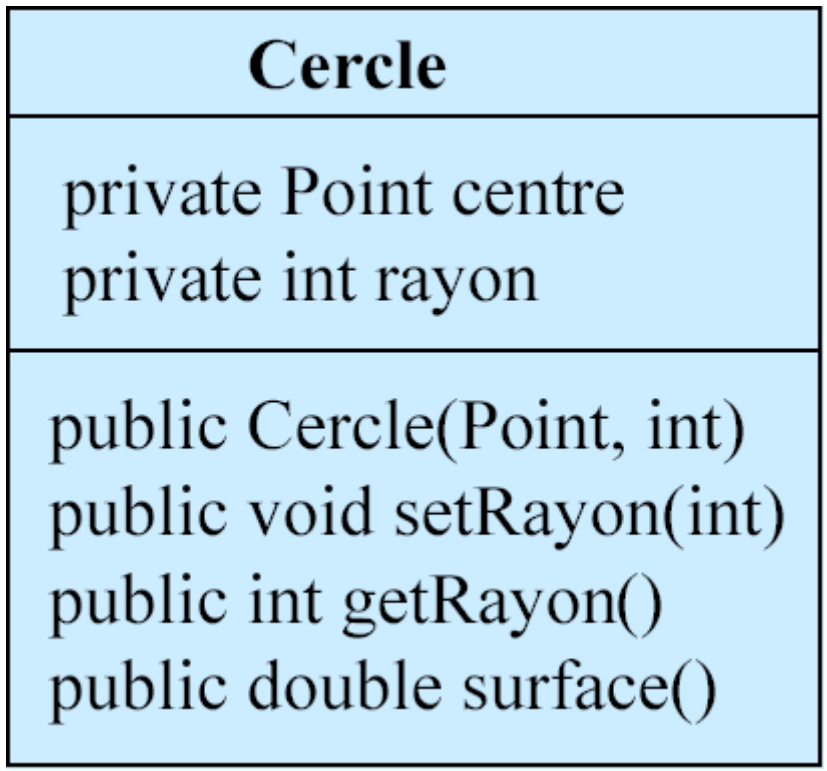
- Les blocs **static** permettent d'initialiser les variables **static** trop complexes à initialiser dans leur déclaration :

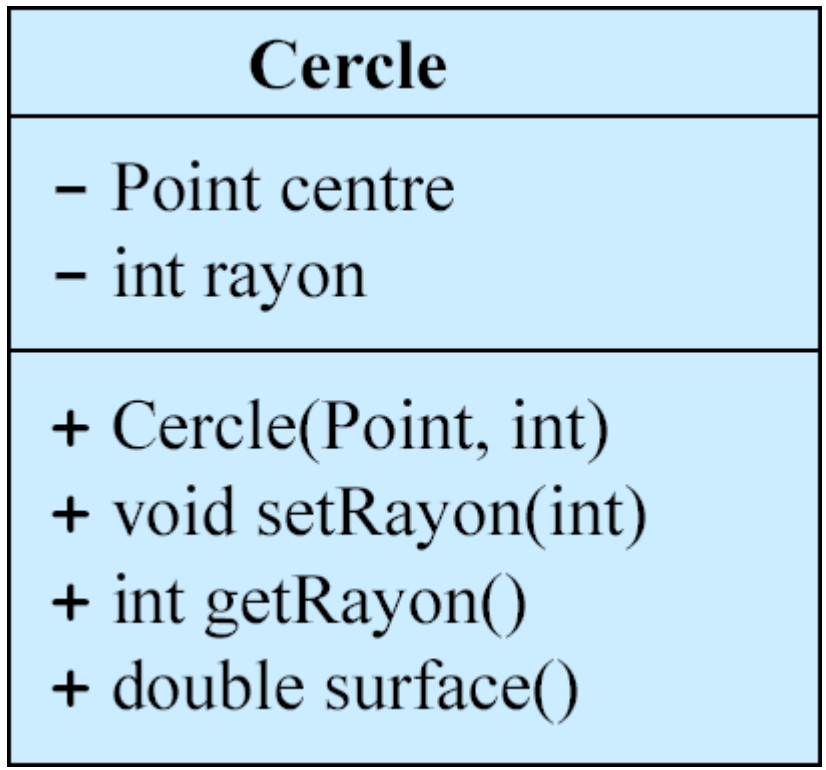
```
class UneClasse {
  private static int[] tab = new int[25];
  static {
          for (int i = 0; i < 25; i++) {
               tab[i] = -1;
          }
  }
  . . .
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

Représentation en UML

Représentation graphique d'une classe en notation UML (*Unified Modeling Language*)





(**-** : private, **#** : protected, **+** : public, **$** (ou souligné) : static)

Déclaration et création des tableaux

- Tableaux considérés comme des objets

  – les variables de type tableau contiennent des références aux tableaux

  – les tableaux sont créés par l'opérateur **new**

  – ils ont une variable d'instance : **final int length**

  – ils héritent des méthodes d'instance de **Object**

- Déclaration : la taille n'est pas fixée

  int[] tabEntiers;

- Création : on doit donner la taille (fixée)

  tabEntiers = new int[5];

  Chaque élément du tableau reçoit la valeur par défaut du type de base du tableau

Tableaux à plusieurs dimensions

- Déclaration

```
int[][] notes;
```

Il faut donner au moins les premières dimensions

- Création

notes = new int[30][3]; // 3 notes, 30 étudiants

notes = new int[30][ ]; // nbre de notes variable

- Déclaration, création et initialisation et affectation

```
int[][] notes = {       {10, 11, 9} // 3 notes
                                {15, 8} // 2 notes
```

Chaînes de caractères

- 2 classes + 1:
  - **String** pour les chaînes constantes
  - **StringBuffer** pour les chaînes variables
  - JDK5 : **StringBuilder = StringBuffer** en plus rapide et pas *thread safe*
- Cf API

Egalité de **Strings**

- La méthode **equals()** teste si 2 instances de **String** contiennent la même valeur :

```
String s1, s2;
s1 = "Bonjour ";
s2 = "les amis";
if ((s1 + s2).equals("Bonjour les amis"))
                System.out.println("Egales");
```

- Le test d'égalité « **==** » teste si les 2 objets ont la même adresse en mémoire ; il ne doit pas être utilisé pour comparer 2 chaînes, même s'il peut convenir dans des cas particuliers
- **equalsIgnoreCase()** ignore la casse des lettres

En-tête d'une méthode

- [accessibilité] [**static**] type-ret nom([liste-param])

public
protected
private

void
int
Cercle
...

public double salaire()

static int nbEmployes()

public void setSalaire(double unSalaire)

private int calculPrime(int typePrime, double salaire)

public int m(int i, int j, int k)

Le type retourné peut être le nom d'une classe

Passage des paramètres

- Le passage se fait par valeur
- Mais pour les objets, on passe une référence par valeur
- Donc,

  – si la méthode modifie l'objet référencé, l'objet sera modifié en dehors de la méthode,

  – si la méthode change la référence, ceci n'aura pas d'incidence en dehors de la méthode

Paramètres **final**

- **final** indique qu'un paramètre ne pourra être modifié dans la méthode
- Si le paramètre est d'un type primitif, la valeur du paramètre ne pourra être modifiée :

  int m(final int x)
- Attention ! si le paramètre n'est pas d'un type primitif, la référence à l'objet ne pourra être modifiée mais l'objet lui-même pourra l'être

  int m(final Employe e1)

  Le salaire de l'employé **e1** pourra être modifié

Recherche des classes par la JVM

Chemin de recherche des classes

- Les outils *java* et *javac* recherchent toujours d'abord dans les fichiers système :

  – fichiers **rt.jar** et **i18n.jar** dans le répertoire **jre/lib** où java a été installé,

  – fichiers .jar ou .zip dans le sous-répertoire **jre/lib/ext**

- Ils regardent ensuite dans le *classpath* ; on peut indiquer le *classpath* par la variable **CLASSPATH** ou par l'option **-classpath**

Chemin de recherche des classes

Classpath

- Le *classpath* contient par défaut le seul répertoire courant (il est égal à « **.** »)
- Si on donne une valeur au *classpath*, on doit indiquer le répertoire courant si on veut qu'il y soit
- Dans le *classpath* on indique des endroits où trouver les classes et interfaces :
  - des répertoires (les classes sont recherchées sous ces répertoires selon les noms des paquetages)
  - des fichiers **.jar** ou **.zip**

Exemples de Classpath

- Sous Unix, le séparateur est « **:** » :

  .:~/java/mespackages:~/mesutil/cl.jar

- Sous Windows, le séparateur est « **;** » :

  .;c:\java\mespackages;c:\mesutil\cl.jar

Héritage

Surclassement (upcasting)

- ***Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A***

```
class B extends A {
…
}
```

- tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.

```
A a = new B(…);
```

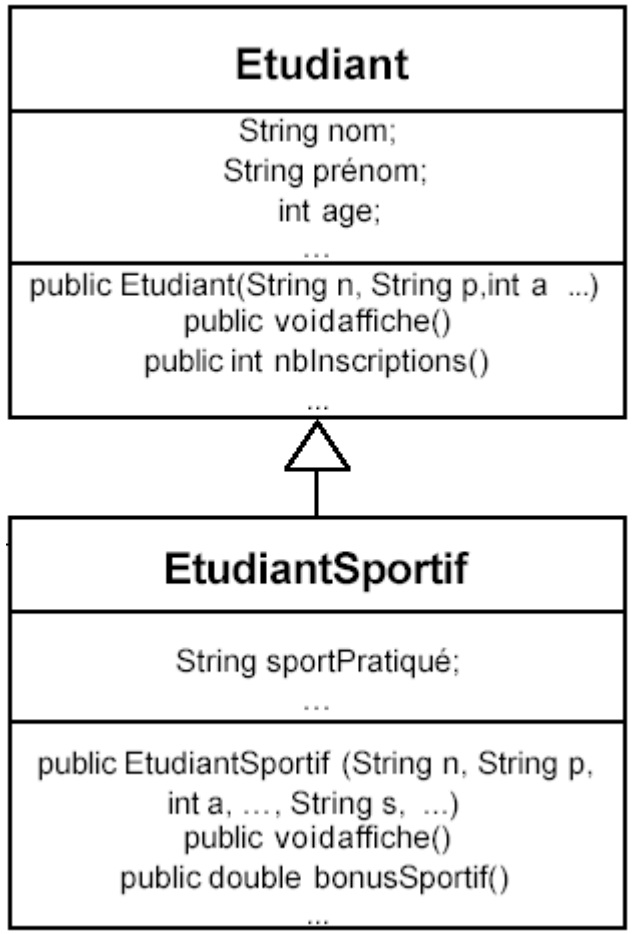
Surclassement (upcasting)

- Lorsqu'un objet est "sur-classé" il est vu comme un objet du type de la référence utilisée pour le désigner
- *Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence*

Surclassement (upcasting)


```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT",
   “fred”,25,..,"Badminton",..);
Etudiant e;

e = es; // upcasting
e.affiche();
es.affiche();
e.nbInscriptions();
es.nbInscriptions();
es.bonusSportif();
e.bonusSportif();
```

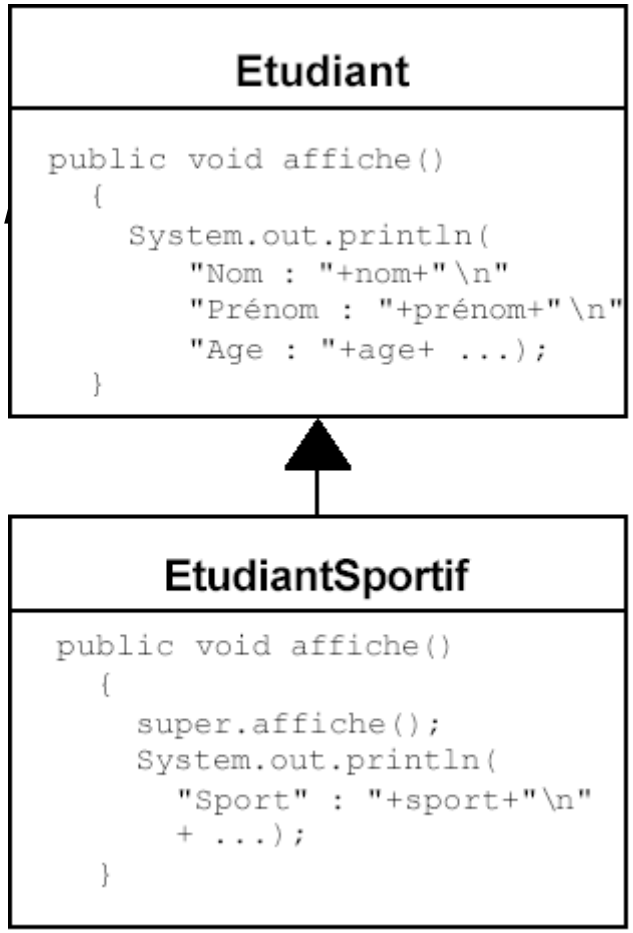


Surclassement et lien dynamique


```
EtudiantSportif es;
es = new EtudiantSportif("DUPONT",
   “fred”,25,..,"Badminton",..);
Etudiant e;

e = es; // upcasting
e.affiche();
```

Quelle méthode est appellée ?



Surclassement et lien dynamique

- **Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée**
- Les messages sont résolus **à l'exécution**
- Ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamic binding, latebinding ou run-time binding)

Surclassement et lien dynamique

- **A la compilation** :seules des **vérifications statiques** qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées

```
public class A {
   public void m1() {
   }
}

public class B extends A {
   public void m1() {
   }
   public void m2() {
   }
}
```

```
A obj = new B();
obj.m1();
obj.m2();
```

Lien dynamique et surcharge

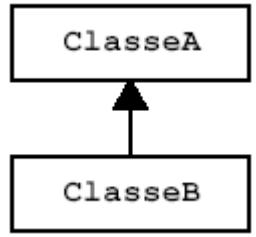
- Le choix de la méthode à exécuter est effectué **statiquement à la compilation** en fonction du type des paramètres (signature)
- La sélection du code à exécuter est effectué **dynamiquement à l'exécution** en fonction du type effectif du récepteur du message

Polymorphisme

- En programmation Objet, on appelle polymorphisme
  - *le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe*
  - *le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.*

Polymorphisme et surcharge


```
public class ClasseC {
  public static void methodeX(ClasseA a){
  }
  public static void methodeX(ClasseB b){
  }
}
```




```
ClasseA refA = new ClasseA();
ClasseC.methodX(refA);
ClasseB refB = new ClasseB();
ClasseC.methodX(refB);
refA = refB; // upCasting
ClasseC.methodX(refA);
```

Polymorphisme et surcharge


```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o
 }
 ...
}
```

```
public class Point {
 private double x;
 private double y;
 public boolean equals(Point pt) {
  return this.x == pt.x && this.y == pt.y;
 }
}
```

Polymorphisme et surcharge


```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);    ??
Object o = p2;
p1.equals(o);     ??
o.equals(p1);     ??
```

- **Le choix de la méthode à exécuter est effectué statiquement à la compilation en fonction du type déclaré de l ’objet récepteur du message et du type déclaré du (des) paramètre(s)**

Casting

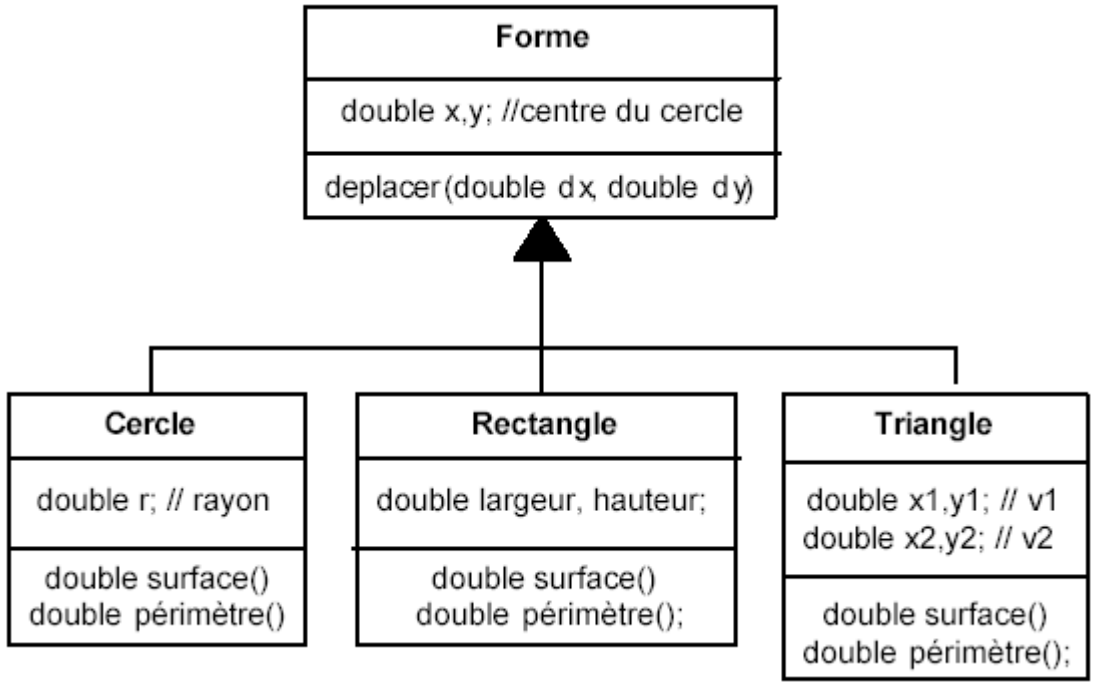
- Le downcasting (ou transtypage) permet de «forcer un type» à la compilation
- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de ***obj*** soit «compatible» avec le type ***ClasseA***

  *Compatible : la même classe ou n'importe qu'elle sous classe de **ClasseA** (**obj instanceof ClasseA**)*
- Si la promesse n'est pas tenue une erreur d'exécution se produit

  ClassCastException est levée et arrêt de l'exécution

Classes abstraites

Classes abstraites



Classes abstraites


```
public class ListeDeFormes {
   public static final int NB_MAX = 30;
   private Forme[] tabForme = new Forme[NB_MAX];
   private int nbFormes = 0;
public void ajouter(Forme f) {
   if (nbFormes < NB_MAX)
   tabForme[nbFormes++] =f
}
public void toutDeplacer(double dx,double dy) {
   for (int i=0; i < NbFormes; i++)
   tabFrome[i].deplace(dx,dy);
}
public double périmetreTotal(){
   double pt = 0.0;
   for (int i=0; i < NbFormes++; i++)
      pt += tabForme[i].périmetre();
   return pt;
}
```

Classes abstraites

- Utilisation d'une classe abstraite
- **Utilité** :
  - *définir des concepts incomplets qui devront être implémentés dans les sous classes*
  - *factoriser le code*


```
public abstract class Forme {
  protected double x,y;
  public void deplacer(double dx, double dy) {
    x += dx; y += dy;
  }
  public abstract double périmètre() ;
  public abstract double surface();
}
```

Classes abstraites

- Une ***classe abstraite*** est une classe non instanciable
- Une ***opération abstraite*** est une opération n'admettant pas d'implémentation
- Une classe pour laquelle au moins une opération abstraite est déclarée est une classe abstraite
- Les opérations abstraites sont particulièrement utiles pour mettre en œuvre le polymorphisme.

  l'utilisation du nom d'une classe abstraite comme type pour une (des) référence(s) est toujours possible (souvent souhaitable !!)

Classes abstraites

- Pour être utile, une classe abstraite doit admettre des classes descendantes ***concrètes***.
- Toute classe **concrète** sous-classe d'une classe abstraite doit “concrétiser” toutes les opérations abstraites de cette dernière.

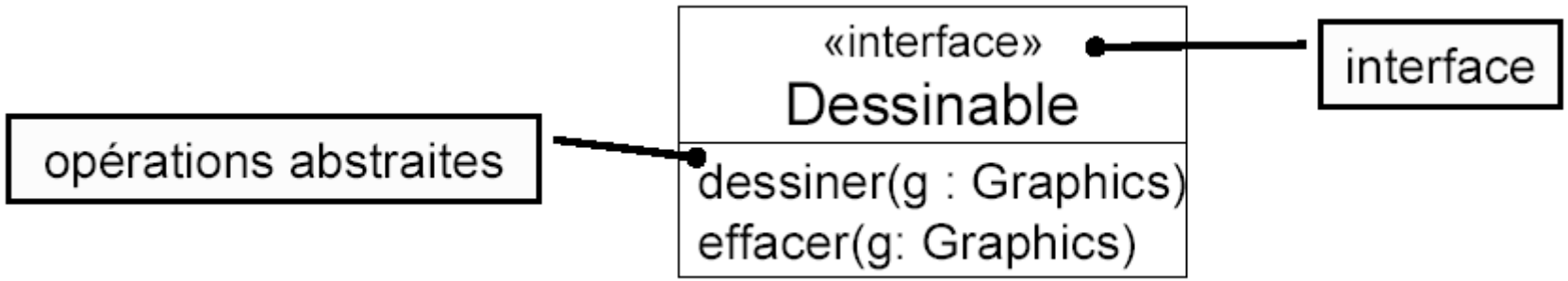
Interfaces

Interfaces

- Une *interface* est une collection d'opérations utilisée pour spécifier un service offert par une classe.
- Une interface peut être vue comme une classe abstraite sans attributs et dont toutes les opérations sont abstraites.

Interfaces


```
import java.awt.*;
public interface Dessinable {
public void dessiner(Graphics g);
void effacer(Graphics g);
}
```



Interfaces

- Toutes les méthodes sont abstraites
- Elles sont implicitement publiques
- Possibilité de définir des attributs à condition qu'il s'agisse d 'attributs de type primitif
- Ces attributs sont implicitement déclarés comme **static final**

Interfaces

- Une interface est destinée à être *“réalisée”* (*implémentée*) par d’autres classes (celles-ci en héritent toutes les descriptions et concrétisent les opérations abstraites)
- *Les classes réalisantes **s'engagent** à fournir le service spécifié par l'interface*

Interfaces

- De la même manière qu'une classe étend sa super-classe elle peut de manière **optionnelle** implémenter une ou plusieurs interfaces

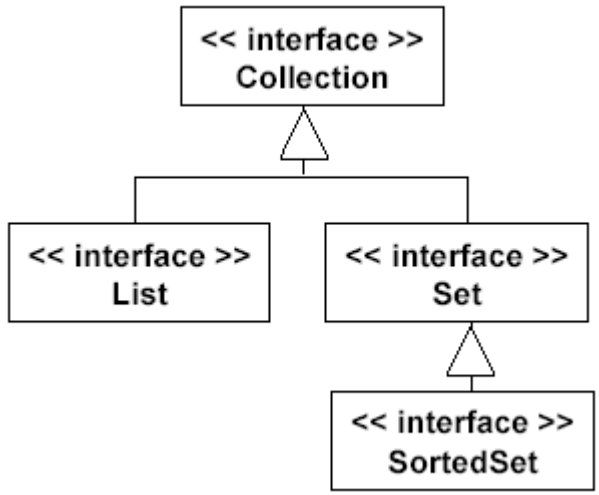
dans la définition de la classe, après la clause **extends *nomSuperClasse***, *faire apparaître explicitement le mot clé* **implements** *suivi du nom de l'interface implémentée*

Interfaces

- Une interface peut être utilisée comme un type
- *A des variables (références) dont le type est une interface il est possible d'affecter des instances de toute classe implémentant l'interface, ou toute sous-classe d'une telle classe.*

Interfaces

- De la même manière qu'une classe peut avoir des sous-classes, une interface peut avoir des "sous-interfaces"
- Une sous interface
  - *hérite de toutes les méthodes abstraites et des constantes de sa "superinterface"*
  - *peut définir de nouvelles con[illegible] abstraites*



Interfaces

- Les interfaces permettent de s ’affranchir d’éventuelles contraintes d’héritage.
  - *Lorsqu’on examine une classe implémentant une ou plusieurs interfaces, on est sûr que le code d’implémentation est dans le corps de la classe. Excellente localisation du code (défaut de l’héritage multiple, sauf si on hérite de classes purement abstraites).*
- Permet une **grande évolutivité** du modèle objet

Interfaces


```
interface EstType {
    void afficherType();
}
```

```
interface Parle {
        void disBonjour();
}
```

```
class Personne implements EstType, Parle {
    public void afficherType() {
    System.out.println("Je suis une personne ");
    }
    public void disBonjour() {
    System.out.println("Bonjour !");
    }
}
class Voiture implements EstType {
    public void afficherType() {
    System.out.println("Je suis une voiture "); }
}
```

Compléments

Classes et interfaces internes

- Types de classes internes

Depuis la version 1.1, Java permet de définir des classes à l'intérieur d'une classe

- Il y a 2 types de classes internes :
  - classes définies à l'extérieur de toute méthode (au même niveau que les méthodes et les variables d'instance ou de classe)
  - classes définies à l'intérieur d'une méthode

Classes internes non incluses dans une méthode

- Le code de ces classes est défini à l'intérieur d'une autre classe, appelée classe englobante, au même niveau que les autres membres :

```
public classe A {
        private int x;
        class B {
        . . .
        }
        public String m() { . . . }
        . . .
}
```

Code de la classe interne

Classes internes non incluses dans une méthode

- Modificateurs
  - Une telle classe peut avoir les mêmes degrés d'accessibilité que les membres d'une classe :

    private, *package*, **protected**, **public**
  - Elle peut aussi être **abstract** ou **final**
- Nommer une classe interne

  La classe englobante (**ClasseE**) fournit un espace de noms pour une classe interne (**Classe1**) :

  son nom est de la forme « **ClasseE.Classe1** »

Classes internes non incluses dans une méthode

- Importer des classes internes
  - On peut importer une classe interne :

    import ClasseE.Classe1;
  - On peut aussi importer toutes les classes internes d'une classe :

    import ClasseE.*;
- Restriction sur les noms des classes internes

  Une classe interne ne peut avoir le même nom qu'une classe englobante (quel que soit le niveau d'imbrication)

Classes internes non incluses dans une méthode

2 types de classes internes définies à l'extérieur d'une méthode :

- Classes **static** (*nested class*) : leurs instances ne sont pas liées à une instance de la classe englobante

- Classes non **static** (*inner class* en anglais) : une instance d'une telle classe est liée à une instance de la classe englobante

Classes internes **static**

- Rôle

  Une classe interne **static** joue à peu près le même rôle que les classes non internes : indique que la classe interne n'a de sens qu'en relation avec la classe externe

- Visibilité
  - Une classe interne **static** a accès à toutes les variables **static** de la classe englobante, même les variables **static private**
  - Elle n'a pas accès aux variables non **static**
  - La classe englobante a accès à tous les membres de la classe interne, qu'ils soient

Classes internes **static**

- Création d'une instance d'une classe interne **static :**

A l'extérieur de la classe englobante, on peut créer une instance de la classe interne par :

ClasseE.Classel x = new ClasseE.Classel(…);

Classes internes **static :** exemple

- On veut récupérer les valeurs minimale et maximale d'un tableau, variable d'instance d'une classe
- Pour cela on écrit une méthode qui renvoie une paire de nombres (pour éviter un double parcours du tableau)
- On crée une classe interne **static** **Extrema** pour contenir cette paire de nombres

Classes internes **static** : exemple

- Classe **TableauInt**

  Classe qui enveloppe un tableau

  Elle définit une classe interne **Extrema** dont les instance contiendront la plus petite et la plus grande valeur du tableau

Classes internes **static** : exemple


```
public class TableauInt {
  private int[] valeurs;
  . . .
  public static class Extrema {
          private int x, y;
          private Extrema(int x, int y) {
               this.x = x;
               this.y = y;
          }
          public int getX() { return x; }
          public int getY() { return y; }
  }
```

Classes internes **static** : exemple


```
public Extrema getMinMax() {
  if (nbElements == 0) return null;
  int min = valeurs[0];
  int max = min;
  for (int i = 1; i < nbElements; i++) {
        if (valeurs[i] < min)
              min = valeurs[i];
        if (valeurs[i] > max)
              max = valeurs[i];
  }
  return new Extrema(min, max);
}
```

Classes internes **static** : exemple

Utilisation de **getMinMax()**


```
// Dans une autre classe que TableauInt
TableauInt t;
. . .
TableauInt.Extrema extrema =
  t.getMinMax();
System.out.println("Min = " +
  extrema.getX());
System.out.println("Max = " +
  extrema.getY());
```

Classes internes non **static**

- Une instance d'une classe interne non **static** ne peut exister que « à l'intérieur » d'une instance de la classe englobante (appelée **`ClasseE`** pour la suite)
- Le code de la classe interne peut désigner cette instance de la classe englobante par **`ClasseE.this`**
- Les classes internes non **static** ne peuvent avoir de variables **static**

Classes internes non **static** : visibilité

Une classe interne non **static** partage tous les membres (même privés) avec la classe dans laquelle elle est définie :

- la classe interne a accès à tous les membres de la classe englobante
- la classe englobante a accès à tous les membres de la classe interne

Classes internes non **static** : nommage

Une classe interne non **static** peut accéder à tout membre (variable ou méthode) ou constructeur de la classe dans laquelle elle est définie

Si le membre n'est pas caché, elle peut le nommer simplement par son nom

Si le membre est une méthode, l'appel s'applique évidemment au *this* englobant

Classes internes non **static** : nommage

- Soit **`ClasseI`** une classe interne non static de **`ClasseE`**
- Soit **`instanceClasseE`** est une instance de **`ClasseE`**
- On crée une instance de **`ClasseI`** interne à **`instanceClasseE`** dans le code d'une autre classe par :

```
instanceClasseE. new ClasseI(...)
```

Classes internes non **static** : exemple


```
public class Tableau {
  private Object[] valeurs;
  private int nbElements;
  public Tableau(int n) {
          valeurs = new Object[n];
  }
  public Iterator elements() {
          return new IterTableau();
  }
  private class IterTableau implements
  Iterator {
  . . .
  }
}
```

Classes internes non **static** : exemple


```
private class IterTableau implements Iterator
  {
  private int indiceCourant = 0;
  public boolean hasNext() {
          return indiceCourant < nbElements;
  }
  public Object next() {
          return valeurs[indiceCourant++];
  }
  public void remove() {
          throw new
  UnsupportedOperationException();
  }
}
```

Accès direct aux attributs privés de la classe englobante

Classes internes non **static** : exemple

Utilisation de **Tableau** :

```
Tableau t1 = new Tableau(10),
           t2 = new Tableau(5);
// Ajout de quelques éléments dans t1 et t2
. . .
// Affichage des éléments de t1 et t2
Iterator it1 = t1.elements(), it2 =
  t2.elements();
while (it1.hasNext()) {
  System.out.println(it1.next());
}
while (it2.hasNext()) {
  System.out.println(it2.next());
}
```

Classes internes non **static** : avantages

Dans l'exemple précédent, le fait que la classe interne **IterTableau** puisse accéder aux variables privées de la classe **Tableau** permet d'encapsuler complètement la structure de données de **Tableau**

Pour faire la même chose avec une classe externe, il aurait fallu ajouter une méthode **get(int i)** à **Tableau**, pour obtenir le ième élément d'une instance de **Tableau** à partir de la classe **IterTableau**

Classe interne **static** ou non ?

- Le critère de choix est le suivant :

– ne choisir non **static** que si la classe interne doit accéder à une variable d'instance de la classe englobante,

– sinon, choisir **static**

Héritage des types internes

Les types internes (classes ou interfaces) s'héritent comme les autres membres

Une sous-classe peut ainsi utiliser une classe interne d'une classe mère si elle est **protected**, par exemple, pour définir une sous-classe de cette classe

Classe interne abstraite

- Une classe peut avoir une interface interne ou une classe interne abstraite
- On ne doit pas pour autant la déclarer abstraite (bien qu'il soit difficile de voir un exemple où on pourrait créer une instance d'une telle classe)
- Ses classes filles devront fournir la classe non abstraite (classe fille de la classe abstraite ou classe qui implémente l'interface) qui permettra le bon fonctionnement de la classe

Classes internes locales

- Classes internes locales, définies à l'intérieur d'une méthode
- 2 types de telles classes (font partie des *inner classes* en anglais) :
  - classes avec un nom
  - classes anonymes

Les classes anonymes sont utilisées plus souvent que les classes avec nom

Classes internes anonymes

On peut utiliser des instances de classes internes à une méthode, sans nom, dites anonymes, sous-classe d'un classe mère ***ClasseMère***

L'instance est créée par :

```
new ClasseMère(listeParamètres) {
// Définition de la classe (variables,
// méthodes)
. . .
}
```

où ***listeParamètres*** doit correspondre à la signature d'un des constructeurs de ***ClasseMère***

Classes internes anonymes : instanciation

Ces classes anonymes n'ont pas de constructeur puisqu'elles n'ont pas de nom

En fait, la création de l'instance de la classe anonyme fait un appel au constructeur de la classe mère dont la signature correspond à ***listeParamètres*** :

```
Cercle c = new Cercle(p, r) {
  public void dessineToi(Graphics g) {
     …
  }
};
```

Classes internes anonymes : instanciation

ClasseMère peut être remplacé par un nom d'interface qu'implémentera la classe anonyme ;

dans ce cas la liste des paramètres doit être vide (appel du constructeur de **Object**) :

```
new Interface() {
  // Définition de la classe
  . . .
}
```

Par exemple :

```
new Iterator() { . . . }
```

Classes internes anonymes : exemple


```
public class Tableau {
   . . .
   public Iterator elements() {
       return
             new Iterator() {
                   //Déf.classe qui implémente Iterator
                   private int indiceCourant = 0;
                   public boolean hasNext() {
                     return indiceCourant < nbElements;
                   }
                   . . .
             }; // Fin de la classe anonyme
   } // Fin de la méthode elements()
}
```

Classes internes anonymes : avantages

Avantages

- Si le code de la classe anonyme est court, l'utilisation d'une classe anonyme améliore la lisibilité car le code de la classe est proche de l'endroit où il est utilisé
- Cela évite aussi d'avoir à utiliser un nouveau nom de classe

Inconvénients

- Si le code est long, la classe anonyme va au contraire nuire à la lisibilité
- Si on veut créer plusieurs instances de la classe, on ne peut utiliser une classe anonyme

Classes internes anonymes : utilisation

- Une classe interne anonyme sert à redéfinir (resp. implémenter) « à la volée » une ou plusieurs méthodes de la classe mère (resp. d'une interface)
- Remarque : ça ne sert à rien d'ajouter de nouvelles méthodes publiques car on ne pourra les appeler de l'extérieur de la classe (vous voyez pourquoi ?)

Classes internes anonymes : utilisation

Contexte d'exécution

Les classes internes locales ont accès :

– aux variables d'instance et de classe de la classe englobante (même privées)

– aux paramètres **final** et aux variables locales **final** de la méthode

Classes internes anonymes : utilisation

Interfaces internes

- Les interfaces internes sont implicitement **static**
- Une interface peut être interne à une classe (rare) ou à une autre interface (par exemple **java.util.Map.Entry**)

Paquetages

Définition d'un paquetage

- Les classes Java sont regroupées en paquetages (*packages* en anglais)
- Ils correspondent aux « bibliothèques » des autres langages comme le langage C, Fortran, Ada, etc...
- Les paquetages offrent un niveau de modularité supplémentaire pour
  - réunir des classes suivant un centre d'intérêt commun
  - la protection des attributs et des méthodes
- Le langage Java est fourni avec un grand nombre de paquetages

Principaux paquetages du SDK

Paquetages

- **java.lang** : classes de base de Java
- **java.util** : utilitaires
- **java.io** : entrées-sorties
- **java.awt** : interface graphique
- **javax.swing** : interface graphique avancée
- **java.applet** : applets
- **java.net** : réseau
- **java.rmi** : distribution des objets

Nommer une classe

- Le nom complet d'une classe (*qualified name* dans la spécification du langage Java) est le nom de la classe préfixé par le nom du paquetage :

  java.util.ArrayList
- Une classe du même paquetage peut être désignée par son nom « terminal » (les classes du paquetage **java.util** peuvent désigner la classe ci-dessus par "**ArrayList**")
- Les classes des autres paquetages doivent être désignées par leur nom complet

Importer une classe d'un paquetage

- Pour pouvoir désigner une classe d'un autre paquetage par son nom, il est possible « d'importer » la classe par l'instruction **import** :

```
import java.util.ArrayList;
public class Classe {
ArrayList liste = new ArrayList();
```

- Classes du paquetage **java.lang** sont automatiquement importées
- Import de toutes les classes d'un paquetage :

```
import java.util.*;
```

- Import des membres statiques d'un paquetage :

```
import static java.lang.Math.*;
```

Lever une ambiguïté

- On aura une erreur à la compilation si
  - 2 paquetages ont une classe qui a le même nom
  - ces 2 paquetages sont importés en entier
- Exemple (2 classes **List**) :

```
import java.awt.*;
import java.util.*;
```

- Pour lever l'ambiguïté, on devra donner le nom complet de la classe. Par exemple,

```
java.util.List l = getListe();
```

Ajout d'une classe dans un paquetage

- **package** *nom-paquetage*;

  doit être la première instruction exécutable du fichier source définissant la classe (avant même les instructions **import**)

- Par défaut, une classe appartient au paquetage par défaut qui n'a pas de nom, et auquel appartiennent toutes les classes situées dans le même répertoire (et qui ne sont pas d'un paquetage particulier)

Sous-paquetage

- Un paquetage peut contenir des sous-paquetages
- Par exemple, le paquetage **java.awt** contient le paquetage **java.awt.event**
- L'importation des classes d'un paquetage n'importe pas les sous-paquetages ; on écrira par exemple :

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

Placement d'un paquetage

- Les fichiers **.class** doivent se situer dans un répertoire placé sous un des répertoires indiqués dans le *classpath*

- Le nom relatif du répertoire doit correspondre au nom du paquetage ;

  ex :les classes du paquetage **fr.miage.orleans.liste** devront se trouver dans un (sous-)répertoire : **fr/miage/orleans/liste**

Classe dans un paquetage

- Si la définition de la classe commence par

  public class

  la classe est accessible de partout

- Sinon, la classe n'est accessible que depuis les classes du même paquetage

Compiler les classes d'un paquetage

javac -d *répertoire-package Classe.java*

- L'option « **-d** » permet d'indiquer le répertoire où sera rangé le fichier compilé
- Si on compile avec l'option « **-d** » un fichier qui comporte l'instruction « **package nom1.nom2**», le fichier **.class** est rangé dans le répertoire *répertoire-package***/nom1/nom2**
- Si on compile sans l'option « **-d** », le fichier **.class** est rangé dans le même répertoire que le fichier **.java** (quel que soit le paquetage auquel appartient la classe)

Exécuter une classe d'un paquetage

- On lance l'exécution de la méthode **main()** d'une classe en donnant son nom complet (préfixé par le nom de son paquetage)
- Par exemple, si la classe **C** appartient au paquetage **p1.p2** :

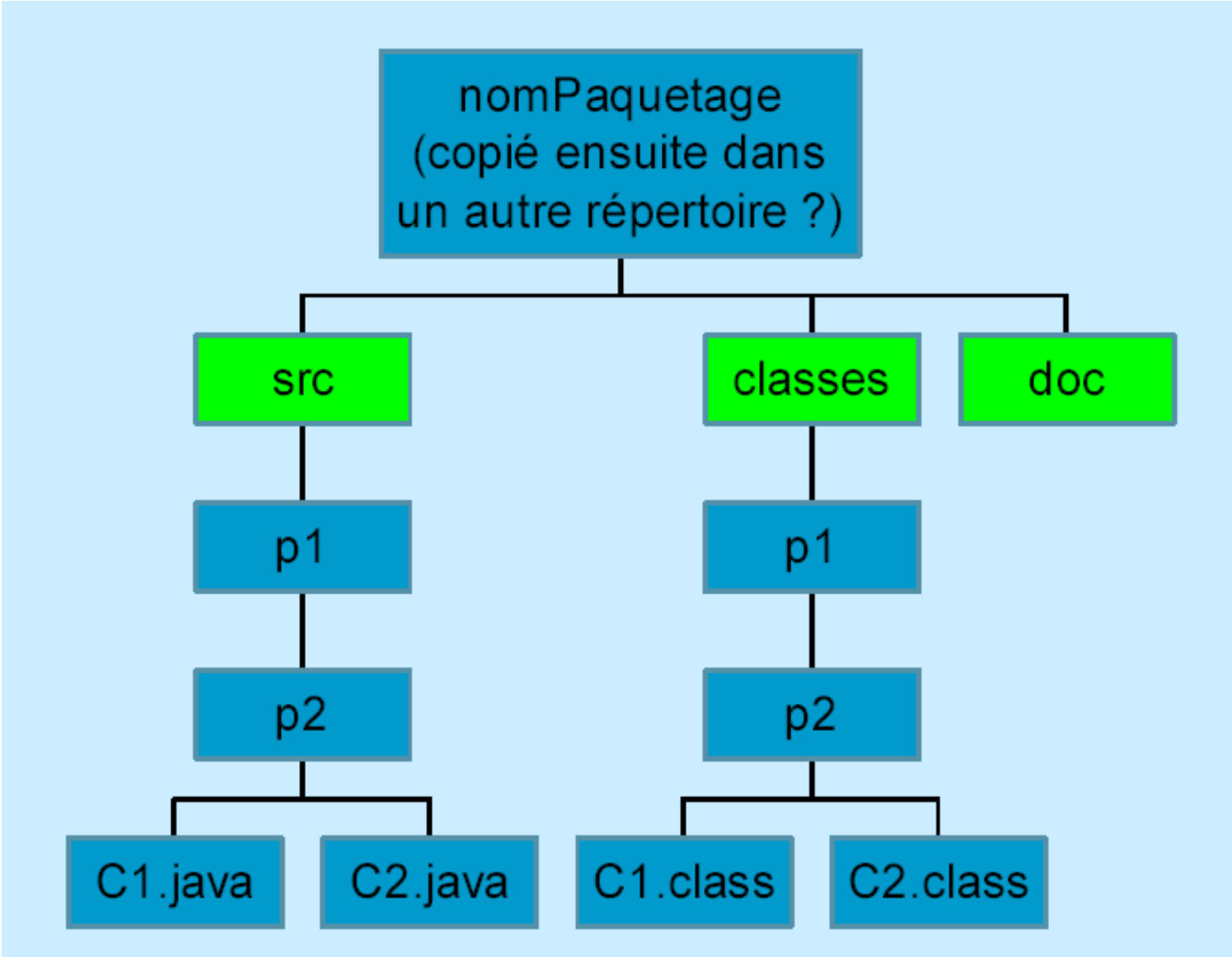
  java p1.p2.C
- Le fichier **C.class** devra se situer dans un sous-répertoire **p1/p2** d'un des répertoires du *classpath* (option **-classpath** ou variable **CLASSPATH**), en général .

Utilisation pratique des paquetages

- Les premières tentatives de développement avec paquetages conduisent à de grosses difficultés pratiques pour compiler/déployer les classes
- Ces difficultés peuvent être évitées
  - en respectant quelques principes simples pour le placement des fichiers sources et classes (exemple dans les transparents suivants)
  - en utilisant correctement les options
    - **–classpath** et **-d**

Placements préconisés

Paquetages



Commandes à lancer

- Si on se place dans le répertoire racine,
  - pour compiler (*en une seule ligne*) :

```
javac -d classes -classpath classes
src/p1/p2/*.java
```

  - pour exécuter :

```
java -classpath classes p1.p2.C1
```

  - pour générer la documentation :

```
javadoc -d doc -sourcepath src p1.p2
```

- On peut ajouter d'autres répertoires ou fichier .jar dans le *classpath*

```
java -classpath classes:lib/mysql.jar:.
p1.p2.C1
```

Classes inter-dépendantes

- Si 2 classes sont inter-dépendantes, il faut les indiquer toutes les deux dans la ligne de commande java de compilation :

```
javac … A.java B.java
```

ou encore

```
javac … A.java B.java
```

Option **-sourcepath**

- javac peut ne pas savoir où sont les fichiers source de fichiers classe qu'il rencontre (en particulier quand un autre paquetage est utilisé)
- Si on veut recompiler ces classes si elles ne sont pas à jour, il faut utiliser l'option

  –**sourcepath** qui indique un répertoire racine pour les fichiers source (l'endroit exact où se trouvent les fichier **.java** doit refléter le nom du paquetage)
- Cela permet aussi de compiler 2 paquetages en même temps

Option **-sourcepath**

- javac recherche les classes dont il a besoin dans le classpath
- S'il ne trouve pas le bon fichier .class, mais s'il trouve le fichier .java correspondant, il le compilera pour obtenir le .class cherché
- S'il trouve les 2 (.class et .java), il recompilera la classe si le .java est plus récent que le .class
- Si l'option –sourcepath a été donnée, il ne recherche que les .class dans le classpath

Compilation de plusieurs paquetages

- Situation : compiler une classe **C**, et toutes les classes dont cette classe dépend (certaines dans des paquetages pas encore compilés)
- « **javac C.java** » ne retrouvera pas les fichiers class des paquetages qui ne sont pas déjà compilés
- On doit indiquer où se trouvent les fichiers source de ces classes par l'option **–sourcepath** par exemple,

  javac –sourcepath src –classpath classes

  –d classes C.java

Fichiers « *makefile* »

- Pour le développement d'applications complexes, il vaut mieux s'appuyer sur un utilitaire de type *make*
- Les développeurs Java ont développé l'utilitaire Ant, très évolué et spécialement adapté à Java
- Autre solution : Maven (v 2) : + le déploiement

Assertions

Assert

- Instruction **assert**
  - *permet de tester des suppositions sur le comportement du code Java*
  - *contient une expression booléenne supposée être toujours vraie*
- A l'exécution
  - *par défaut les assertions ne sont pas activées, instruction sans effet*
  - *possibilité de les activer pour débogage et test*
    - *Évaluation de l'expression booléenne*
    - *Si vraie, assert est sans effet*
    - *Si fausse une erreur est lancée.*

Assert

- Intérêt
  - *Moyen rapide et efficace de détecter les erreurs de programmation*
  - *Permet d'augmenter la confiance dans le code produit*
  - *Facilite la maintenance du code en documentant le fonctionnement interne du programme*
  - *Pas de system.out.println / printf !!!*

Syntaxe

- **Dans sa forme la plus simple une assertion s'écrit :**

  ```
  assert expression1;
  ```

  où

  expression1 *est une expression booléenne*

- A l'exécution :
  - ***expression1*** *est évaluée,*
  - *si elle est fausse une erreur de type AssertionError est lancée*
  - *cet objet AssertionError ne contient pas de message*

Syntaxe

- **Dans sa forme la plus élaborée une assertion s'écrit :**

  `assert` *`expression1 : expression2 ;`*

  où

  expression1 *est une expression booléenne*

  expression2 *est une expression quelconque (qui à une valeur, donc pas void)*

- A l'exécution :
  - ***expression1*** *est évaluée,*
  - *si elle est fausse une erreur de type AssertionError est lancée*
  - *la valeur de expression2 (transformée en chaîne de caractères) est utilisée comme message dans l'objet AssertionError*

Compilation

- Par défaut les assertions ne sont pas testées à l'exécution
- Avec le flag –enableassertions (-ea) pour les activer, -disableassertions (-da) pour désactiver
- La granuralité est spécifiée par les arguments du flag :
  - Aucun
    Active ou non les assertions dans toutes les classes sauf les classes systeme
  - *packageName*...
    Active ou non les assertions dans le package et ses sous-packages.
  - ...
    Active ou non les assertions dans le package par défaut dans le répertoire en cours.
  - *className*
    Active ou non les assertions dans la classe

```
java -ea:fr.test.fruitbat... -da:fr.test.fruitbat.Brickbat
   BatTutor
```

Utilisation

- Pour documenter toutes les suppositions faites quand vous programmez

```
if (x == 0) {
...
else { // x vaut 1
...
}
```

```
if (x == 0) {
...
else {
assert x == 1 : x;
...
}
```

- ***si le code est modifié ultérieurement et que x prend une valeur autre que 0 ou 1*** **possibilités de bugs n'apparaissant pas immédiatement et difficiles à localiser.**
- **lancement d'une AssertionError le bug apparaît immédiatement et est immédiatement loca**lisé

Utilisation

- Pour tester une précondition d'une méthode privée

```
private static Object[] subArray(Object[]a, int x, int y) {
  // précondition x doit être <= à y
  assert x <= y : "subArray: x > y";
  ...
}
```

- Ne pas utiliser assert pour tester les préconditions d'une méthode publique
  - **Le programmeur de la méthode ne peut garantir à l'avance que la méthode sera toujours appelée correctement**
  - **Les arguments doivent être testés explicitement**

Utilisation

- Pour tester un invariant de classe

Écriture d'une classe représentant une liste d'objets, autorisant l'insertion et la suppression d'objets mais conservant toujours la liste triée.


```
public void insert(Object o) {
// effectue l'insertion
...
assert isSorted(); // l'invariant de classe
}
```

Méthode testant si la liste est triée ou non

Utilisation

- **Ce qu'il ne faut pas faire**
  - *écrire des assertions utilisant des expressions avec effet de bord*

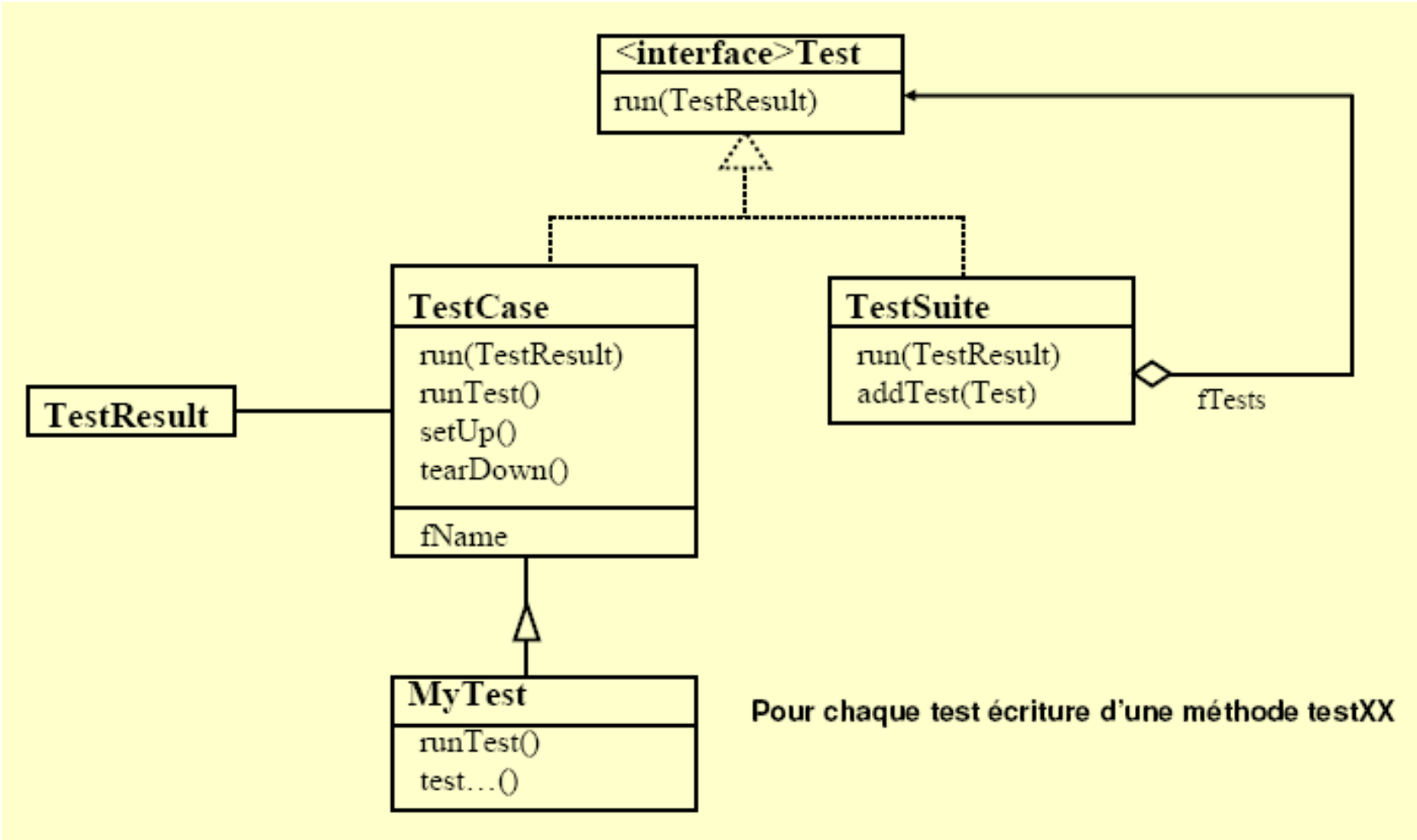
  ! le comportement du programme pourra être différent selon que les assertions sont activées ou non
- *essayer « d'attraper » (« catcher ») une* ***AssertionError***
  - *une des suppositions faites par le programmeur a été violée*
  - *le code est utilisée en dehors des cas prévus, on ne peut attendre qu'il fonctionne correctement*
  - *il n'y a pas de moyen plausible de récupérer ce type d'erreur*

Tests unitaires

Pourquoi JUnit ?

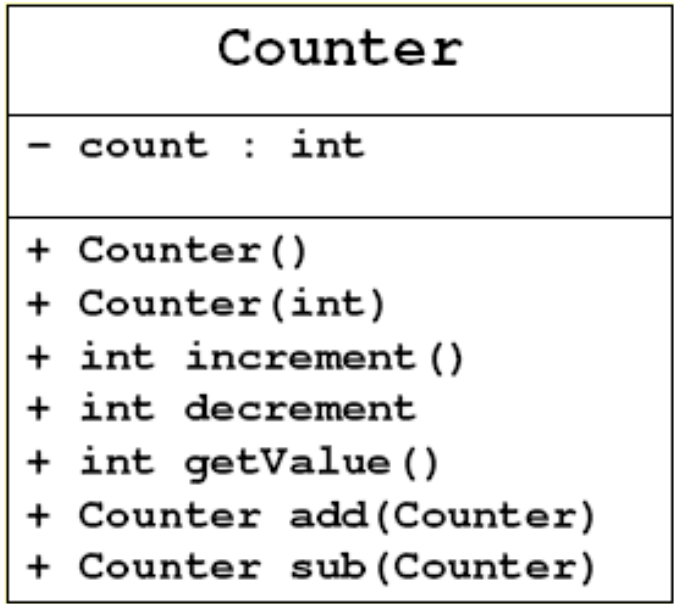
- Traces (instruction **print**) dans les programmes
- Utilisation d'un debugger (possibilité de redéfinir les expressions de test) sans avoir à recompiler les programmes
- Sévère limitations
  - *nécessitent un jugement humain*
  - *problèmes de composition :*
    - *Une seule expression exécutable à la fois avec un debugger*
    - *Si une trace contient de nombreux print, perte de lisibilité*
- JUnit : un framework open source (www.junit.org) pour le test unitaire de programmes Java qui permet d'automatiser les tests

Classes de JUnit



Ecriture d'une classe de test

- Classe à tester :



Ecriture d'une classe de test

- Classe de test :

```
public class TestCounter extends TestCase {
    //…
    public void testSimpleAdd() {

        Counter c1 = new Counter(10);
        Counter c2 = new Counter(12);

        Counter c3 = c1.add(c2);

        assertTrue(c3.getValue() ==
                   c1.getValue() + c2.getValue());
    }
}
```

- **Création des objets qui vont interagir lors du test**
- **Code qui agit sur les objets impliqués dans le test**
- **Vérification que le résultat obtenu correspond bien au résultat attendu**

Ecriture d'une classe de test

`static void assertTrue(boolean test)`

méthode JUnit : vérifie que test == true *et dans le cas contraire lance une exception (en fait une Error) de type* **`AssertionFailedError`**

L'exécuteur de tests JUnit attrape ces objets Errors et indique les tests qui ont échoué

Les méthodes `assert`

- static void assertTrue(String ***message***, boolean ***test***)
  - Le *message* optionnel est inclus dans l'Error

- static void assertFalse(boolean ***test***)
- static void assertFalse(String ***message***, boolean ***test***)
  - *vérifie que* test == true

- assertEquals(***expected***, ***actual***)
- assertEquals(String ***message***, ***expected***, ***actual***)
  - *méthode largement surchargée:* arg1 *et* arg2 *doivent être tout deux des objets* ou bien *du même type primitif*
  - *Pour les objets, utilise la méthode equals (*public boolean equals(Object o) *sinon utilise* ==

Les méthodes `assert`

- assertSame(Object ***expected***, Object ***actual***)
- assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - *Vérifie que* expected *et* actual *référencent le même objet (==)*

- assertNotSame(Object ***expected***, Object ***actual***)
- assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - *Vérifie que* expected *et* actual *ne référencent pas le même objet (==)*

- assertNull(Object ***object***)
- assertNull(String ***message***, Object ***object***)
  - *Vérifie que* objet *est* null

- assertNotNull(Object ***object***)
- assertNotNull(String ***message***, Object ***object***)
  - *Vérifie que* objet *n'est pas* null

Les méthodes assert

- fail()
- fail(String ***message***)
  - *Provoque l'échec du test et lance une* AssertionFailedError
  - *Utile lorsque les autres méthodes assert ne correspondent pas exactement à vos besoins ou pour tester que certaines exceptions sont bien lancées*

```
try {
  // appel d'une méthode devant lancer une Exception
  fail("Did not throw an ExpectedException");
}
catch (ExpectedException e) { }
```

Ecriture d'une classe de test

```
public class TestCounter extends TestCase {
    //…
    public void testAdd() {

        Counter c1 = new Counter(10);
        Counter c2 = new Counter(12);
        Counter c3 = c1.add(c2);
        assertTrue(c3.getValue() ==
                   c1.getValue() + c2.getValue());
    }



    public void testSub() {

        Counter c1 = new Counter(10);
        Counter c2 = new Counter(12);
        Counter c3 = c1.sub(c2);
        assertTrue(c3.getValue() ==
                   c1.getValue() - c2.getValue());
    }
}
```

```
private Counter c2;
private Counter c1;
```

```
public void setUp() {
    super.setUp();
    c1 = new Counter(10);
    c2 = new Counter(12);
}
```

```
public void tearDown() {
    super.tearDown();
    ...
}
```

Exécution d'un test

- **Pour exécuter un test**

```
TestResult result = (new
   CounterTest("testIncrement")).run();
```

- **Pour exécuter une suite de tests**

```
TestSuite suite = new TestSuite();
suite.addTest(new CounterTest("testIncrement"));
suite.addTest(new CounterTest("testDecrement"));
TestResult result = suite.run();
```

- *En utilisant l'introspection*

```
TestSuite suite = new TestSuite(CounterTest.class);
TestResult result = suite.run();
```

Résultats d'un test

- JUnit propose des outils pour exécuter suite de tests et afficher les résultats
  - junit.textui.TestRunner *: affichage textuel*
  - junit.swingui.TestRunner *: affichage graphique*
- Ajouter à la classe de test une methode statique **`suite`** pour rendre une suite accessible à un "exécuteur de test" (TestRunner) :

```
public static Test suite() {
TestSuite suite = new TestSuite();
suite.addTest(new CounterTest("testIncrement"));
suite.addTest(new CounterTest("testDecrement"));
return suite;
}
```

- Autre solution (introsepection)

```
junit.textui.TestRunner.run(CounterTest.class);
```

- De nombreux IDE intègrent JUnit : NetBeans, JBuilder, Eclipse

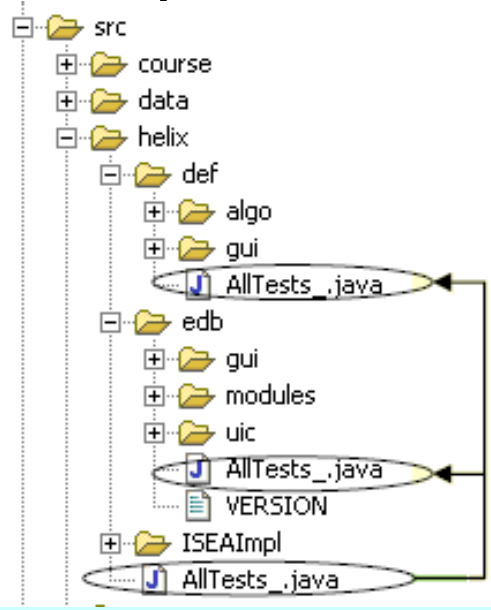
Jouer tous les tests d'un package

- **TestSuite** contient des objets **Test**
  - *Des* **TestCase** *mais aussi des* **TestSuite**
- Une classe **AllTest**
  - *Teste toutes les suite de tests du package*

```
public class AllTests {
      public static Test suite() {
      TestSuite suite =
            new TestSuite("Test for default package");
      suite.addTest(new TestSuite(TokenizerTest.class));
      suite.addTest(new TestSuite(NodeTest.class));
      suite.addTest(new TestSuite(ParserTest.class));
      suite.addTest(new TestSuite(TreeTest.class));
      return suite;
      }
}
```

Jouer tous les tests de sous-package

- Dans chaque package définir une suite qui inclus les suites des sous packages




```
public class AllTests_{
      public static Test suite() {
            TestSuite suite =
new TestSuite("All helix.* and subpackages tests");
            suite.addTest(helix.edb.AllTests_.suite());
            suite.addTest(helix.def.AllTests_.suite());
      return suite;
      }
}
```

Test Tips

- Code a little, test a little, code a little, test a little . . .
- Run your tests as often as possible, at least as often as you run the compiler ☺
- Begin by writing tests for the areas of the code that you're the most worried about . . .write tests that have the highest possible return on your testing investment
- When you need to add new functionality to the system, write the tests first
- If you find yourself debugging using System.out.println(), write a test case instead
- When a bug is reported, write a test case to expose the bug
- Don't deliver code that doesn't pass all the tests

Principes

- **Integration continue** – Pendant le développement, le programme ***marche toujours*** – peut être qu'il ne fait pas tout ce qui est requis mais ce qu'il fait, il le fait bien.
- TDD Test Driven Development :pratique qui se rattache à l'eXtreme Programming

"Any program feature without an automated test simply doesn't exist." **from Extreme Programming Explained, Kent Beck**

Tests : Références

- www.junit.org
- www.extremeprogramming.org
- www.developpez.com
- http://cruisecontrol.sourceforge.net/