Programmation par Objets

Pierre Karpman
https://www-ljk.imag.fr/membres/Pierre.Karpman/

d'après un support de Clément Pernet

L3-MI, UFR IM^{2}AG, Université Grenoble Alpes

Plan du cours

Approche orientée objet

Programme informatique:

- Données
- Traitements (procédures, fonctions, ...)

Approche orientée objet

Programme informatique:

- Données
- Traitements (procédures, fonctions, ...)

Programmation Structurée

Que fait le programme ?

- Accent mis sur l'action: le traitement
- Découpage selon la fonction
- *programmes, sous-programmes, bibliothèques, ...*

Approche orientée objet

Programme informatique:

- Données
- Traitements (procédures, fonctions, ...)

Programmation Structurée

Que fait le programme ?

- Accent mis sur l'action: le traitement
- Découpage selon la fonction
- *programmes, sous-programmes, bibliothèques, ...*

Prog. Orientée Objet

Sur quoi porte le programme?

- Accent mis sur les données
- Traitements regroupés selon la sémantique des données
  $\rightsquigarrow$ *méthodes*
- classes

Approche orientée objet

Programme informatique:

- Données
- Traitements (procédures, fonctions, ...)

Programmation Structurée

Que fait le programme ?

- Accent mis sur l'action: le traitement
- Découpage selon la fonction
- *programmes, sous-programmes, bibliothèques, ...*

Prog. Orientée Objet

Sur quoi porte le programme?

- Accent mis sur les données
- Traitements regroupés selon la sémantique des données
  $\rightsquigarrow$ *méthodes*
- classes

À terme, les données restent et les fonctions sont ré-écrites.

Notion d'objet, l'encapsulation

L'objet est constitué de membres:

- des données (ou champs)
- des méthodes agissant uniquement sur les données de l'objet
- les valeurs des données définissent *l'état* de l'objet

Encapsulation: lorsque les données ne sont pas accessible de l'extérieur, mais seulement via des méthodes

Dans ce cas, on parle d'abstraction de donnée.
$\rightsquigarrow$ Avantage: permet de modifier la représentation de donnée sans casser l'utilisation de l'objet depuis l'extérieur.

Programmation Structurée vs Orientée Objet

Programmation Structurée

Unité logique : le module

- Une zone pour les variables
- Une zone pour les fonctions
- Structuration descendante du programme
- Chaque fonction résout une partie du problème

Programmation Structurée vs Orientée Objet

Programmation Structurée

Unité logique : le module

- Une zone pour les variables
- Une zone pour les fonctions
- Structuration descendante du programme
- Chaque fonction résout une partie du problème

Prog. Orientée Objet

Unité logique : l'objet

- un état :
  attributs (variables) qui stockent des valeurs
- un comportement :
  méthodes qui modifient des états
- une identité :
  distinction entre objets

Notion de classe

L'ensemble des objets qui peuvent avoir les mêmes états et les mêmes comportements est une classe

- Les classes servent de *moules* à la création d'objets : un objet est une instance d'une classe (généralisation de la notion de type)
- Un programme OO est constitué de classes qui permettent de créer des objets
- Les objets communiquent entre eux en invoquant leurs méthodes
  - modifient l'état
  - invoquent d'autres méthodes
  - ...
- L'ensemble des interactions entre objets définit l'algorithme
- Les relations entre classes reflètent la décomposition du programme.

Héritage

Pour définir une classe

- à partir d'une classe existante (classe parent)
- en lui ajoutant des membres (champs ou méthodes)

$\rightsquigarrow$ simplifier, factoriser l'écriture
$\rightsquigarrow$ hiérarchiser les classes, par spécialisation croissante

Polymorphisme

But: *traiter de la même façon des objets de classes différentes*

Exemple: les opérateurs $<, >, ==$ sur les types `int`, `long`, `double`

Solution: Si une classe *B* hérite de *A*, tout objet, instance de *B* peut être vu comme un objet de *A* (il possède au moins tous les membres de *A*).
$\rightsquigarrow$ on peut passer indifféremment un objet de type *A* ou *B* à une fonction attendant un paramètre de type *A*.

Historique

1967: `Simula 67`, premiers concepts d'objets et de classe

1971: `Small Talk`, premier vrai langage objet

80's:

- Objective C
- C++, pas complètement objet
- Common LISP Object System
- Eiffel

90's:

- standardisation de C++
- Java, Python, Ruby
- ...

Java est un langage orienté objet

Un langage orienté objet pur:

- Tout programme est composé de classes et instancie des objets

Mais:

- les types primitifs ne sont pas des classes (entiers, flottants, caractères, booléens).
- certaines méthodes (méthodes de classes, déclarées avec `static`) peuvent être appelées directement, sans nécéssiter l'instanciation d'un objet. Exemple : le programme principal (`main`).

Portabilité

Portabilité: capacité à exécuter le même programme sur plusieurs systèmes et/ou architectures matérielles (en obtenant le même résultat)

Difficulté: design très différents selon les systèmes et archi.

- taille des types de base (`char, int, long`)
- endianness
- encodage des caractères

En général: recompilation sur le système/archi cible

Portabilité

Portabilité: capacité à exécuter le même programme sur plusieurs systèmes et/ou architectures matérielles (en obtenant le même résultat)

Difficulté: design très différents selon les systèmes et archi.

- taille des types de base (`char, int, long`)
- endianness
- encodage des caractères

En général: recompilation sur le système/archi cible

Java: la solution des bytecodes

Intermédiaire entre code compilé et code interprété:

- code compilé entièrement portable
- exécutable par un interpréteur présent sur chaque système/archi cible: la Java Runtime Machine (JVM)
- Les types de base sont exactement définis indépendamment des architectures cibles ($\neq$ C,C++)

Mais aussi: encodage des caractère en unicode.

Plan

Plan

Exemple, la classe `Point`

Définition des Champs

L'état d'un point est représenté par deux coodonnées entières:

```
private int x; // abscisse
private int y; // ordonnee
```

`private` signifie que ces champs ne sont pas accessible depuis l'extérieur de la classe (seulement depuis ses méthodes).
$\rightsquigarrow$ encapsulation (recommandé mais non obligatoire).
En général, les membres privés sont placés à la fin de la classe.

Remarque

Une instance d'une classe a accès aux champs `private` de n'importe quelle autre instance de la même classe

Exemple, la classe `Point`

Définition des Méthodes

Comme pour une fonction: en-tête et bloc de code

```
public void initialise (int abs, int ord){

   x = abs;
   y = ord;
}
```

- `public`: mode d'accès
- `void`: type de retour
- `initialise`: nom de la méthode
- `abs, ord`: arguments
- Affecte les champs `x` et `y` de l'objet de type `Point`

Exemple, la classe `Point`

```
public class Point
{
  public void initialise(int abs, int ord) {
    x = abs; y = ord;
  }

  public void deplace(int dx, int dy) {
    x += dx; y += dy;
  }

  public void affiche() {
    System.out.println("Je suis un point de coordonnees
    + x ", " + y);
    }

  private int x; // abscisse
  private int y; // ordonnee
}
```

Utilisation de la classe `Point`

- La classe Point permet d'instancier des objets de type `Point` et de leur appliquer les méthodes `initialise`, `deplace` et `affiche`.
- Cette utilisation ne peut se faire que depuis une autre méthode (car toutes les instructions sont dans des méthodes).
- A l'intérieur d'une méthode quelconque, une déclaration comme :

  ```
  Point a;
  ```

  est correcte.
- Attention, contrairement aux types primitifs, cette déclaration ne réserve pas d'emplacement pour un objet de type `Point`, mais seulement un emplacement pour une *référence* à un objet de type `Point`.

Utilisation de la classe `Point`

L'emplacement pour l'objet proprement dit est alloué sur une demande explicite du programme à l'aide de l'opérateur `new` :

```
new Point() // attention aux parentheses
```

- crée un emplacement pour un objet de type `Point`
- et fournit sa référence en résultat.

Exemple de construction:

```
Point a;
a = new Point();
```

Utilisation de la classe `Point`

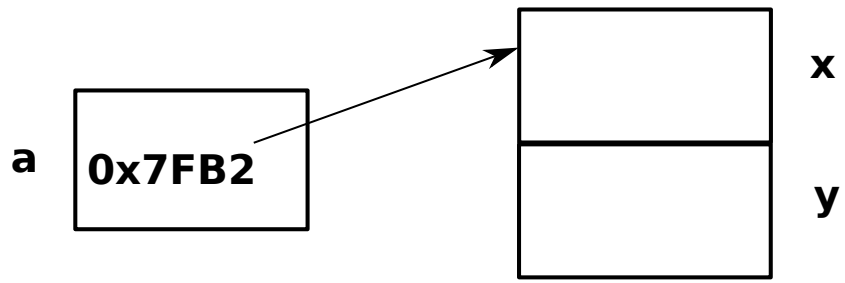
L'emplacement pour l'objet proprement dit est alloué sur une demande explicite du programme à l'aide de l'opérateur `new` :

```
new Point() // attention aux parentheses
```

- crée un emplacement pour un objet de type `Point`
- et fournit sa référence en résultat.

Exemple de construction:

```
Point a;
a = new Point();
```



Utilisation de la classe `Point`

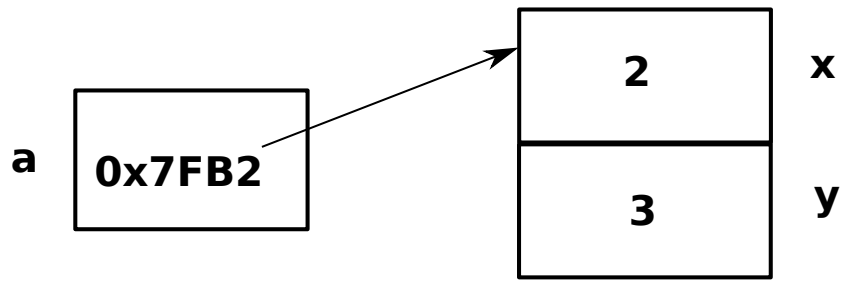
L'emplacement pour l'objet proprement dit est alloué sur une demande explicite du programme à l'aide de l'opérateur `new` :

```
new Point() // attention aux parentheses
```

- crée un emplacement pour un objet de type `Point`
- et fournit sa référence en résultat.

Exemple de construction:

```
Point a;
a = new Point();
```



On peut ensuite utiliser l'objet `a`:

```
a.initialise(2,3);
```

appelle la méthode `initialise` du type `Point` en l'appliquant à l'objet de référence `a`

Utilisation de la classe `Point`

```
public class TestPoint {
  public static void main(String args[]) {
    Point a;
    a = new Point();
    a.initialise(3,5); a.affiche();
    a.deplace(2,0); a.affiche();
    Point b = new Point();
    b.initialise(6,8); b.affiche();
  }
}
```

Mise en œuvre d'un programme comportant plusieurs classes

- Un fichier source par classe (JDK de SUN)

```
javac Point.java
javac TstPoint.java
java TstPoint
```

  Ou bien utilisation d'Eclipse (fichier projet)

Mise en œuvre d'un programme comportant plusieurs classes

- Un fichier source par classe (JDK de SUN)

```
javac Point.java
javac TstPoint.java
java TstPoint
```

  Ou bien utilisation d'Eclipse (fichier projet)
- Plusieurs classes dans un même fichier
  - Une seule classe publique (celle contenant la méthode main)
  - Une classe n'ayant aucun attribut d'accès (inexistant ou public) est accessible à toutes les classes du même paquetage, donc a fortiori à toutes les classes du même fichier

Plan

Notion de constructeur

Définition

Un constructeur est une méthode,

- *sans valeur de retour,*
- *portant le même nom que la classe,*
- *permettant d'automatiser l'initialisation d'un objet*

```
public class Point {
  public Point(int abs, int ord) {
    x = abs; y = ord;
  }
  public void deplace(int dx, int dy) {
    x += dx; y += dy;
  }
  ...
  private int x; // abscisse
  private int y; // ordonnee
}
```

Utilisation des constructeurs

- L'instruction Point a = **new** Point(); ne convient plus.
- Quand une classe dispose d'un constructeur, il n'est pas possible de creer un objet sans l'appeler.

Dans notre exemple, le constructeur a besoin de deux arguments, qui doivent etre fournis lors de la creation :

```
Point a = new Point(1,3);
```

- Mais il est toujours possible d'ajouter à nouveau un constructeur ne prenant aucun argument.

Utilisation des constructeurs

Quelques règles concernant les constructeurs

- Une classe peut ne disposer d'aucun constructeur (mais s'il en existe un, il doit être utilisé)
  T t = **new** T(); // *quand est-ce correct ?*
- Un constructeur ne peut être appelé directement

```
a.Point(8,3); // Erreur
```

- Un constructeur peut appeler un autre constructeur de la même classe
- Un constructeur peut être déclaré privé (il ne peut plus être utilisé de l'extérieur)

Construction et initialisation d'un objet

La création d'un objet par un constructeur entraîne, par ordre chronologique les opérations suivantes:

1. Initialisation par défaut de tous les champs de l'objet
2. Initialisation explicite lors de la déclaration des champs
3. Exécution des instructions du constructeur

Initialisations par défaut:

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
byte, int, short, long	0
flottant	0.f ou 0
Objet	null

Constructeur et initialisation d'un objet

```
class A{
  public A(...){...} // constructeur de A
  private int n = 10;
  private int p;
}
```

L'instruction A a = **new** A(...); entraîne :

- L'initialisation (par défaut) des champs n et p de a à 0
- L'initialisation explicite du champ n à la valeur 10 (valeur figurant dans sa déclaration)
- L'exécution des instructions du constructeur

Constructeur et initialisation d'un objet

```
class A{
  public A(...){...} // constructeur de A
  private int n = 10;
  private int p;
}
```

L'instruction A a = **new** A(...); entraîne :

- L'initialisation (par défaut) des champs n et p de a à 0
- L'initialisation explicite du champ n à la valeur 10 (valeur figurant dans sa déclaration)
- L'exécution des instructions du constructeur

Remarque

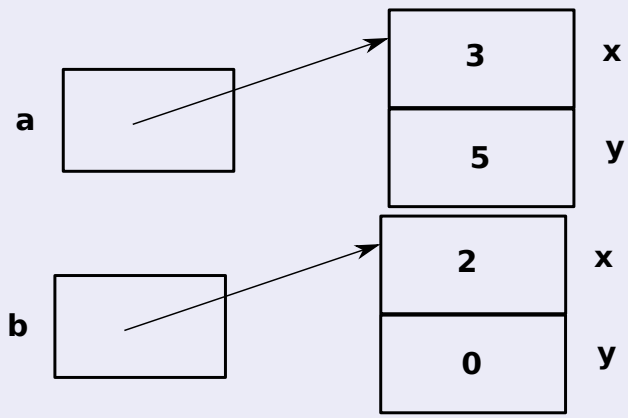
*Les champ avec attribut **final** doivent être initialisés au plus tard dans le constructeur.*

Plan

Affectation et comparaison d'objets

Exemple

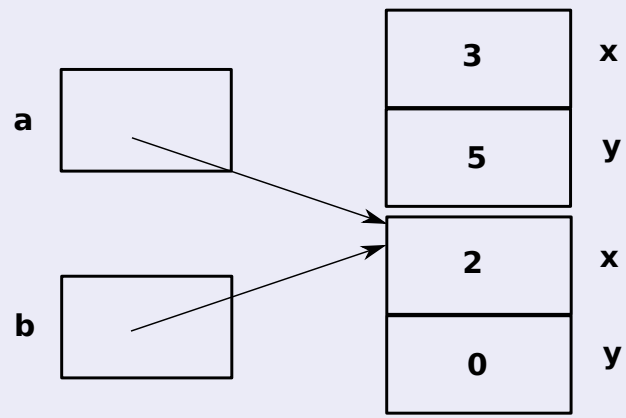
```
Point a, b;
a = new Point (3,5);
b = new Point (2,0);
a = b; // recopie dans a la
       // reference contenue
       // dans b
```



Affectation et comparaison d'objets

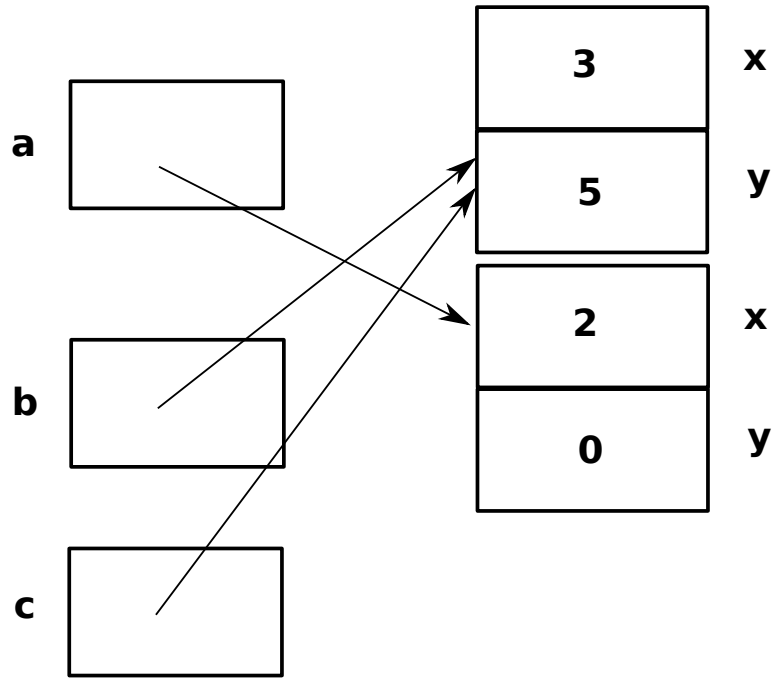
Exemple

```
Point a, b;
a = new Point (3,5);
b = new Point (2,0);
a = b; // recopie dans a la
       // reference contenue
       // dans b
```



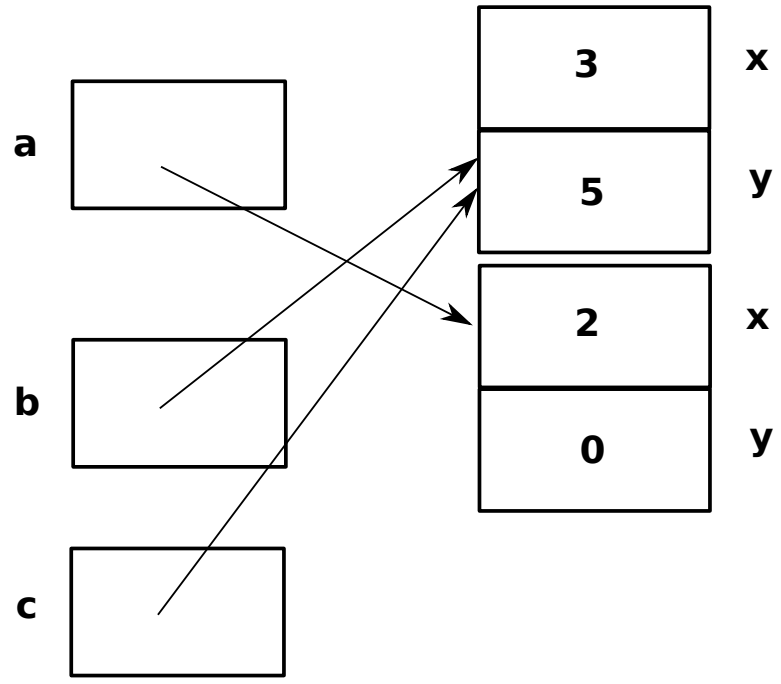
Affectation et comparaison d'objets

```
Point a, b, c;
a = new Point (3,5);
b = new Point (2,0);
c=a;
a=b;
b=c;
```



Affectation et comparaison d'objets

```
Point a, b, c;
a = new Point (3,5);
b = new Point (2,0);
c=a;
a=b;
b=c;
```



- 2 objets de type Point et 3 variables de type Point
- Initialisation de références et références nulles
- Les opérateurs == et != peuvent être utilisés mais s'appliquent aux références.

Affectation et comparaison d'objets

Rappel:

L'affectation a = b modifie les références, pas les objets.

Copie des objets: méthode de clone

```
class Point {
  public Point(int abs, int ord)  { x = abs; y = ord; }
  public Point copie () {
    Point p = new Point();
    p.x = x; p.y = y;
    return p;
  }
  ...
}
Point a = new Point(1,2);
Point b = a.copie(); // b est une copie conforme de a
```

Copie par constructeur

Copie à la construction

```
class Point {
  public Point(int abs, int ord)  { x = abs; y = ord; }
  public Point(Point p)
   {
            x = p.x;
            y = p.y;
   }
  ...
}
Point a = new Point(1,2);
Point b = new Point(a);
```

Remarque

Le second constructeur a accès aux champs `private` d'une instance existante de `Point`

Copies profondes et superficielles

Si certains champs sont des classes :

Copie superficielle: on copie les références des attributs

Copie en profondeur: on clone les attributs

Copie superficielle

```
class Droite {
  public Droite(Point P1,
  Point P2)
  { A = P1;
    B = P2; }
  public Droite copie (){
    Droite d=new Droite();
    d.A = A;
    d.B = B;
    return d;
  }
  private: Point A,B;
}
```

Copie profonde

```
class Droite {
  public Droite(Point P1,
                Point P2)
  { A = P1;
    B = P2; }
  public Droite copie (){
    Droite d=new Droite();
    d.A = A.copie();
    d.B = B.copie();
    return d;
  }
  private: Point A,B;
}
```

Initialisation de référence et référence nulle

Différence entre variables locales et champ:

Variables locales: Initialisation obligatoire avant utilisation

Champs: Si absente, initialisation par défaut

	type primitif	**type classe**
Variable locale	erreur de compilation	erreur de compilation
Champ de classe	0	null

Remarque

Ce mécanisme ne prévient pas toutes les erreurs d'exécution liées à l'utilisation de variable non initialisée.

Plan

Le ramasse-miettes

L'operateur new

- alloue l'emplacement mémoire d'un nouvel objet
- l'initialise (par défaut, explicitement, par constructeur)

Mais pas d'instruction de désallocation

$\rightsquigarrow$ Automatique: le ramasse-miette (garbage collection).

- A tout instant Java connaît le nombre de référence à chaque objet
- Mis à jour à chaque affectation
- Lorsque qu'il n'y a plus de référence
  $\rightsquigarrow$ plus accessible
  $\rightsquigarrow$ désalloue l'espace de l'objet
- Impossible de prédire l'instant où la désallocation aura lieu

Plan

Règles d'écriture

- Méthodes fonctions: une méthode a un type de retour (éventuellement **void**)
- **Arguments muets (ou formels)**: rôle d'une variable locale, initialisés lors de l'appel (ci-dessous: `x`; `y`)
- **Arguments effectifs**: la valeur effective passée lors de l'appel
- Utilisation du **final**: argument en lecture seule

```
class C {
  ...
  double f (final int x, double y){
    y += 1;
    x=2; // Erreur de compilation
    return x+y;
  }
}
```

Conversion de type et ordre d'évaluation

Conversion *type effectif* → *type muet*

selon la hiérarchie des types

byte → short → int → long → float → double
char ↗

- Permet d'utiliser un argument de type `byte` alors qu'un `int` est attendu (mais pas l'inverse! ⇝ ``error: incompatible types: possible lossy conversion from int to byte'')

Ordre d'évaluation des arguments effectifs:

```
int n = 5; double a = 2.5, b = 3.5;
f(n++, n, a = b, a);//valeurs effectives (5,6,3.5,3.5)
int n = 5; double a = 2.5, b = 3.5;
f(n, n++, a, a = b);//valeurs effectives (5,5,2.5,3.5)
```

Plan

Échange d'information avec les méthodes

Deux façon de passer un argument effectif $\rightarrow$ argument muet:

Par valeur: la méthode reçoit une copie (modifs possibles, restent locales)

Par adresse (ou référence): la méthode reçoit l'adresse (référence) de l'argument effectif (les modifs altèrent l'objet effectif).

Échange d'information avec les méthodes

Deux façon de passer un argument effectif $\rightarrow$ argument muet:

Par valeur: la méthode reçoit une copie (modifs possibles, restent locales)

Par adresse (ou référence): la méthode reçoit l'adresse (référence) de l'argument effectif (les modifs altèrent l'objet effectif).

En Java: Par Valeur

Conséquence:

Pour les types primitifs: copie effective

Pour les types objets: copie de la référence

$\rightsquigarrow$ Donc les objet sont passés par référence!

Conséquence pour les types primitifs

Limitation du passage par valeur:

```
class Util {
  public static void Echange(int a, int b)  {
    System.out.println("Debut : " + a + " " + b);
    int c; c = a; a = b; b = c;
    System.out.println("Fin : " + a + " " + b);
  }
}
public class Echange {
  public static void main(String args[])  {
    int n = 10, p = 20;
    System.out.println("Avant appel : " + n + " " + p);
    Util.Echange(n,p);
    System.out.println("Apres appel : " + n + " " + p);
  }
}
```

Conséquence pour les types classe

Copie par valeur de la référence ⇒ copie par référence de l'objet

```
class Point{
  ...
  public void permute (Point a)
  { Point c = new Point (0,0);
    c.x = a.x; c.y = a.y;//copie de a dans c
    a.x = x; a.y = y;//copie du point courant dans a
    x = c.x; y = c.y;//copie de c dans le point courant
  }
  private int x, y;
}
public class TestPoint
{ public static void main (String args[])
  { Point a = new Point(1,2); Point b = new Point(3,4);
    a.affiche(); b.affiche();
    a.permute(b);
    a.affiche(); b.affiche();
  }
}
```

Quand l'argument est de la même classe

L'unité d'encapsulation est la classe

- Une méthode appelée sur un objet *a* est autorisée à accéder aux champs privés d'un autre objet *b* de la même classe.
- Seules les méthodes d'une classe sont autorisées à accéder aux champs privés de cette classe. Ceci vaut pour tous les objets de la classe, pas seulement l'objet courant.

Fonction coincide:

- Syntaxe : a.coincide(b) pour tester si les points a et b coïncident

```
public boolean coincide (Point pt)
{ return ((pt.x == x) && (pt.y == y));
}
```

Cas de la valeur de retour

De la même façon: passage par valeur.

- Type primitif: copie de la valeur
- Type classe: copie de la référence
  $\rightsquigarrow$ passage par référence

Exemple

```
public Point symetrique()
{ Point res = new Point (y,x);
  return res;
}
...
b = a.symetrique();
```

- la référence res est détruite après l'appel à symetrique
- l'objet créé par **new** Point(y,x) continue d'exister (car il est toujours référencé par b)

Autoréférence: le mot-clé **this**

Traduit la notion de *l'objet courant*.
On aurait pu écrire:

```
public Point (int x, int y)
{ this.x = x;
  this.y = y;
}
```

- Permet d'employer des noms d'arguments identiques à des noms de champ
- Permet d'utiliser l'objet dans sa globalité (pour le passer à d'autres méthodes)

Autoréférence: le mot-clé **this**

Traduit la notion de *l'objet courant*.
On aurait pu écrire:

```
public Point (int x, int y)
{ this.x = x;
  this.y = y;
}
```

- Permet d'employer des noms d'arguments identiques à des noms de champ
- Permet d'utiliser l'objet dans sa globalité (pour le passer à d'autres méthodes)
- Permet d'appeler un constructeur depuis un autre constructeur (obligatoirement en 1ère instruction):

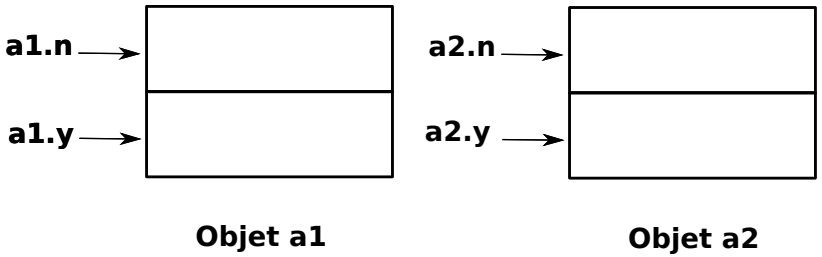
```
public Point(int abs, int ord)
{ x = abs; y = ord; }
public Point()
{ this (0,0);}
```

Champs et méthodes de classe

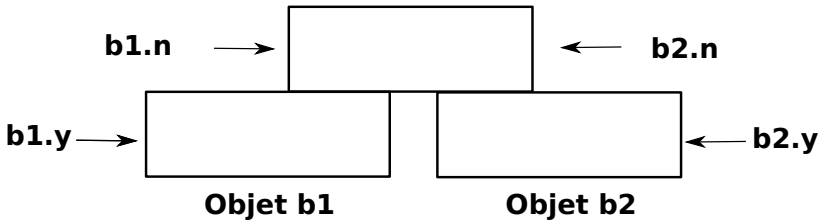
Champ de classe, ou champ statique

Qui n'existe qu'en un seul exemplaire pour la classe, partagé par tous les objets

```
class A
{ int n;
  float y;
}
A a1 = new A();
A a2 = new A();
```




```
class B
{ static int n;
  float y;
}
B b1 = new B();
B b2 = new B();
```



Champs et méthodes de classe

- Un champ static est indépendant des objets qui instancient la classe:

```
B a1 = new B(), a2=new B();
a1.n;
a2.n;
B.n;  // sont tous les 3 equivalents
```

- une méthode peut aussi être déclarée statique.
  - rôle indépendant d'un quelconque objet
  - Attention: ne peut agir sur les champs non static
  - Exemple: la méthode main.

Champs et méthodes de classe

- Un champ static est indépendant des objets qui instancient la classe:

```
B a1 = new B(), a2=new B();
a1.n;
a2.n;
B.n;  // sont tous les 3 equivalents
```

- une méthode peut aussi être déclarée statique.
  - rôle indépendant d'un quelconque objet
  - Attention: ne peut agir sur les champs non static
  - Exemple: la méthode main.

Exercice

Modifier la classe Point pour qu'elle compte le nombre d'instances existantes (accessible par une méthode NbPoint).

Plan

Sur-définition de méthodes

- On parle de *sur-définition* ou *surcharge* lorsqu'un même symbole possède plusieurs significations.
- En Java, plusieurs méthodes peuvent porter le même nom, pourvu que la liste des arguments permette de lever l'ambiguïté pour le compilateur

```
public void deplace(int dx) { x += dx;}
public void deplace(int dx, byte dy) {x += dx; y += dy;}
public void deplace(byte dx, int dy) {x += dx; y += dy;}
...
Point a = ...
int n; byte b;
a.deplace (n);  // OK
a.deplace(n,b);          // OK
a.deplace(b,n);          // OK
a.deplace(b,b);          // erreur: ambiguite
```

Règle pour la sur-définition

Pour un appel donné, le compilateur recherche toutes les méthodes acceptables et choisit la meilleure, si elle existe.

Une méthode est acceptable si:

- elle dispose du nombre d'arguments voulu
- le type de chaque arg. effectif est compatible par affectation avec celui de l'arg. muet correspondant
- elle est accessible

Le choix se déroule ainsi:

- si aucune méthode acceptable: erreur de compilation
- si plusieurs acceptables: élimination des méthodes M' si il existe une méthode M dont les arguments muets sont compatibles en affectation avec les arg muets de M'.
- Après élimination: si il en reste
  - aucune: erreur de compilation.
  - plusieurs: erreur de compilation, ambiguïté

Sur-définition et accessibilité

Que fait le code suivant?

```
class A {
  public void f(float x) {
    System.out.println("-f(float) x = " + x);}
  private void f(int n) {
    System.out.println("-f(int) n = " + n);}
  public void g() {
    System.out.println("--- dans g");
    int n=1; float x=1.5f; f(n); f(x);
  }
}
public class Surdfacc {
  public static void main(String args[]) {
    A a = new A();
    a.g();
    System.out.println("--- dans main");
    int n=2; float x=2.5f; a.f(n); a.f(x);
  }
}
```

Plan

Objets membres et classes internes

Conception de classe: **Contrats** et **Implémentation**

Contrat: *ce que fait la classe*

- en-tête des méthodes publiques (interface)
- comportement de ces méthodes

Implémentation: *comme elle le fait*

- Champ et méthodes privées, corps des méthodes publiques

Encapsulation: Indépendance entre Contrat et Implémentation

- Permet de changer l'implémentation sans que l'utilisateur n'ait à modifier le code utilisant l'interface

Classes internes

Définition imbriquée d'une classe dans une classe

```
class E // classe englobante
{ ...
  class I // interne a la classe E
  { ...
  }
}
```

- Pas nécessairement d'instance de I dans E
- Quel rôle?
  1. Un objet interne est toujours associé à un objet englobant
  2. Un objet interne a accès aux champs et méthodes privés de **l'objet englobant** (et d'aucun autre, y compris les autres objets instanciant la même classe)
  3. Un objet externe a accès aux champs et méthodes privés de **tout objet interne** auquel il a donné naissance.

Distinction avec les **objets membres**: (2) et (3)

Classes internes

Utilisation et construction:

```
class E
{
  public E (int e) { this.e = e; }
  class I
  {
    public I () { this.i = e; }
    private int i;
  }
  private int e;
}
class Test
{ public static void main (String args[])
  {
    E e1 = new E(2);
    E e2 = new E(3);
    E.I i1 = e1.new I();//creation d'objets de type I,
    E.I i2 = e2.new I();//rattache a e1 et e2
  }
}
```

Classe internes et attribut static

Une méthode statique n'est rattachée à aucun objet, donc:

- Une méthode statique d'une classe externe ne peut créer d'objet interne,
- Une classe interne ne peut contenir de membres statiques.

Plan

Plan

Héritage

- Ecrire de nouvelles classes *à partir* de classes existantes
- Ajout de nouveaux membres (méthodes, champs)
- Altération (re-définition) de certains membres sans modifier la classe de base
- Procédé itératif: une classe dérivée peut servir de classe de base pour une nouvelle classe dérivée

Exemple: Création d'une classe de points colorés:

```
class PointColore extends Point
{ public void colorie (byte couleur)
  { this.couleur = couleur; }
  private byte couleur;
}
PointColore pc = new PointColore();
```

Règles de l'héritage

Règle 1: Un objet d'une classe dérivée accède aux membres publics de sa classe de base (comme s'ils étaient définis dans la classe dérivée)

Règle 2: Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base (sinon violation du principe d'encapsulation)

Règle 3: Une méthode d'une classe dérivée a accès aux membres publics de sa classe de base

```
public void affichec() {
  affiche();
  System.out.println(" et ma couleur est :" + couleur);
}
```

Plan

Construction et initialisation des objets dérivés

Règle 4: Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet

Le constructeur de PointColore doit

- initialiser la couleur,
- appeler le constructeur de Point

Règle 5: Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot-clé **super**.

```
public PointColore (int x, int y, byte couleur)
{ super (x, y);//obligatoirement en premiere position
  this.couleur = couleur;
}
```

Construction et initialisation des objets dérivés

Remarques:

- Rappel: **this** permet aussi d'appeler un autre constructeur de la **même** classe. Il doit aussi avoir lieu à la première instruction
  $\rightsquigarrow$ incompatible avec **super**.
- *Une classe peut dériver d'une classe qui dérive elle-même d'une autre classe :* l'appel par **super** concerne le constructeur de la classe du niveau immédiatement supérieur
- *Classe de base et classe dérivée possèdent chacune un constructeur :* le constructeur de la classe dérivée prend en charge l'intégralité de la construction, quitte à s'appuyer sur le constructeur de base
- *La classe de base ne possède aucun constructeur :* on peut appeler le constructeur par défaut dans le constructeur de la classe dérivée (**super**();)
- *La classe dérivée ne possède aucun constructeur :* appel d'un constructeur par défaut de la classe dérivée, qui appellera un constructeur sans argument (soit public, soit par défaut) de la classe de base

Initialisation d'un objet dérivé

```
class B extends A {...}
```

La création d'un objet de B se déroule en 6 étapes:

1. Allocation mémoire pour un objet de type B
2. Initialisation par défaut de tous les champs aux valeurs nulles
3. Initialisation explicite des champs hérités de A
4. Exécution du corps du constructeur de A
5. Initialisation explicite des champs de B
6. Exécution du corps du constructeur de B

Plan

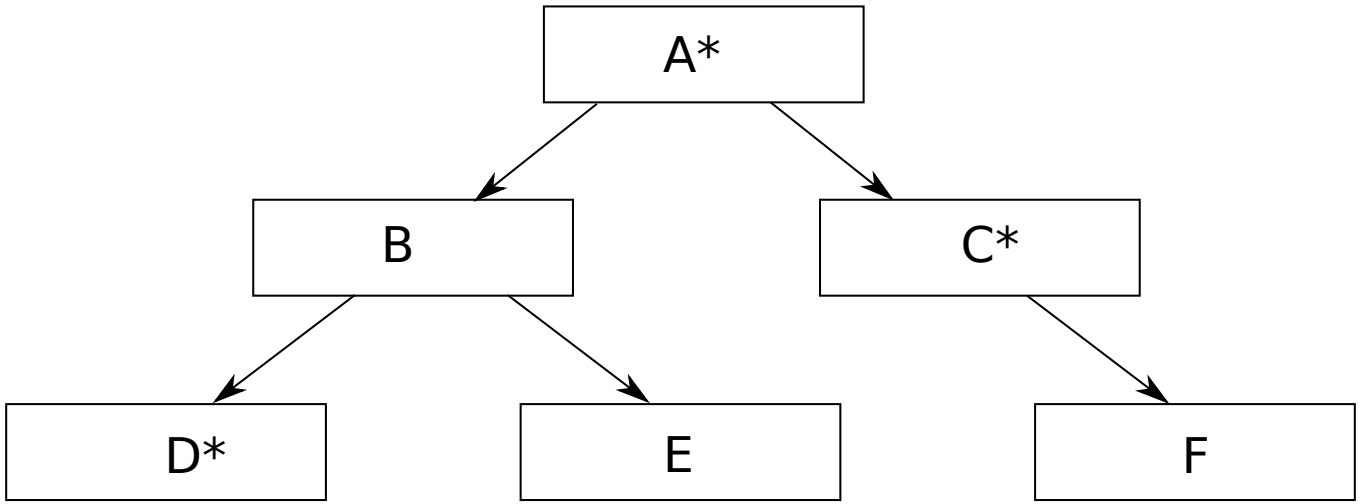
Redéfinition et surdéfinition de membres

- Notion de sur-définition déjà vue au niveau d'une même classe
- S'étend au niveau des méthodes hérités
- Nouveauté: **la re-définition**

```
class PointColore extends Point {
  public void affichec(){
    //affichec joue le meme role que affiche
    affiche();
    System.out.println(" et ma couleur est :"+couleur);
  }
}

class PointColore extends Point {
  public void affiche() {
    // utilisation du meme nom !
    affiche();//ambiguite ici car recursivite!erreur!
    super.affiche();//l'appel est cette fois correct
    System.out.println(" et ma couleur est :"+couleur);
  }
}
```

Redéfinition de méthodes et dérivations successives



X* signifie une défintion ou redéfinition d'une fonction f

- classe A: méthode f de A
- classe B: méthode f de A
- classe C: méthode f de C
- classe D: méthode f de D
- classe E: méthode f de A
- classe F: méthode f de C

Surdéfinition et héritage

Une classe héritée peut surdéfinir une méthode d'une classe de base (ou d'une classe ascendante).

```
class A {
  public void f(int n) { ... }
  ...
}
class B extends A {
  public void f(float x) { ... }
  ...
}
A a; B b;
int n; float x;
...
a.f(n);
a.f(x);
b.f(n);
b.f(x);
```

Surdéfinition et héritage

Une classe héritée peut surdéfinir une méthode d'une classe de base (ou d'une classe ascendante).

```
class A {
  public void f(int n) { ... }
  ...
}
class B extends A {
  public void f(float x) { ... }
  ...
}
A a; B b;
int n; float x;
...
a.f(n);// appelle f(int) de A
a.f(x);// erreur de compilation
b.f(n);// appelle f(int) de A
b.f(x);// appelle f(float) de B
```

Surdéfinition et héritage

Utilisation simultanée de redéfinition et surdéfinition

```
class A {
  public void f(int n) { ... }
  public void f(float x) { ... }
  ...
}
class B extends A {
  public void f(int n) { ... }// redefinition de f(int)
  public void f(double y) { ... }//surdefinition de f
  ...
}
A a; B b;
int n; float x; double y;
...
a.f(n);
a.f(x);
a.f(y);
b.f(n);
b.f(x);
b.f(y);
```

Surdéfinition et héritage

Utilisation simultanée de redéfinition et surdéfinition

```
class A {
  public void f(int n) { ... }
  public void f(float x) { ... }
  ...
}
class B extends A {
  public void f(int n) { ... }// redefinition de f(int)
  public void f(double y) { ... }//surdefinition de f
  ...
}
A a; B b;
int n; float x; double y;
...
a.f(n); // appel de f(int) de A
a.f(x); // appel de f(float) de A
a.f(y); // erreur de compilation
b.f(n); // appel de f(int) de B (redef)
b.f(x); // appel de f(float) de A (surdef)
b.f(y); // appel de f(double) de B (surdef)
```

Contraintes sur la redéfinition

1. Java impose, en cas de redéfinition, l'identité des signatures **et** des types de retour. (pas nécessaire pour la surdéfinition!)

```
class A
{ public int f (int n){...}
}
class B extends A
{ public float f (int n) {...} //erreur
}
```

2. Cas particulier des types de retour co-variant: le type de retour doit être identique ou dérivé de celui de la méthode redéfinie

```
class A
{ public A f(){...}
}
class B extends A
{ public B f(){...}// redefinition ok
}
```

Contraintes sur la redéfinition

3 Les droits d'accès:
Règle 6 : **La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode**

```
class A {
  public void f(int n) {...}
}
class B extends A {
  private void f(int n) {...} // Erreur
}
```

```
class A
{ private void f(int n) {...}
}
class B extends A
{ public void f(int n) {...} // OK
}
```

Règles générales de redéfinition

On a affaire à une redéfinition ssi

1. Même signature, même valeur de retour (ou valeurs covariantes)
2. Droit d'accès de la méthode de la classe dérivée au moins aussi élevé que celui de la classe ascendante,
3. La clause throws de la méthode de la classe dérivée ne mentionne pas d'exceptions non mentionnées dans la clause throws de la méthode de la classe ascendante (cf. plus loin)

Remarques

- Une méthode de classe static ne peut être redéfinie

Plan

Le polymorphisme

Concept puissant en P.O.O., qui complète l'héritage, permettant de manipuler des objets sans connaître exactement leur type.

Exemple

- *Créer un tableau d'objets de type Point et PointColore*
- *Appeler la méthode affiche () sur chaque élément*

```
Point p;
p = new Point(3,5);
p = new PointColore(3,5,(byte) 2);
// p de type Point contient la reference  a un
// objet de type PointColore
```

Le polymorphisme

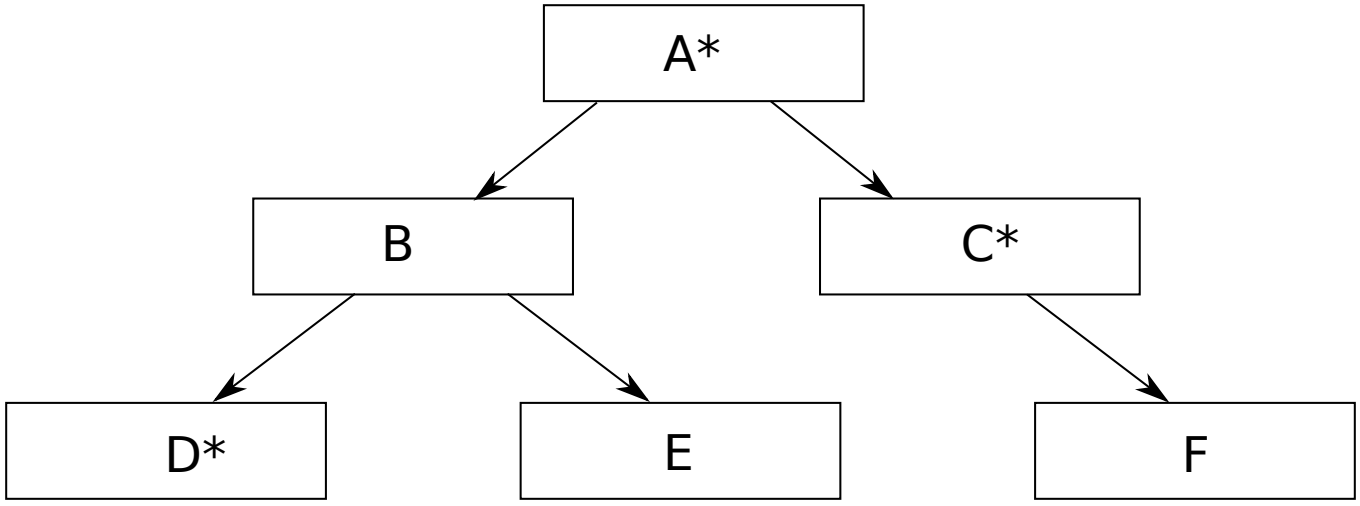
- De manière générale, **on peut affecter à une variable objet une référence à un objet de type dérivé.**
- Conversion implicite d'une référence à un type de classe T en une référence à un type ascendant de T.
- On parle aussi de compatibilité par affectation entre un type classe et un type ascendant.
- **L'appel à une méthode se fonde** non pas sur le type de la variable, mais **sur le type de l'objet référencé** : p.affiche() appelle la méthode affiche() de PointColore
- **Ligature dynamique**: choix de la méthode au moment de l'exécution et non de la compilation

Le polymorphisme

En résumé, le polymorphisme c'est :

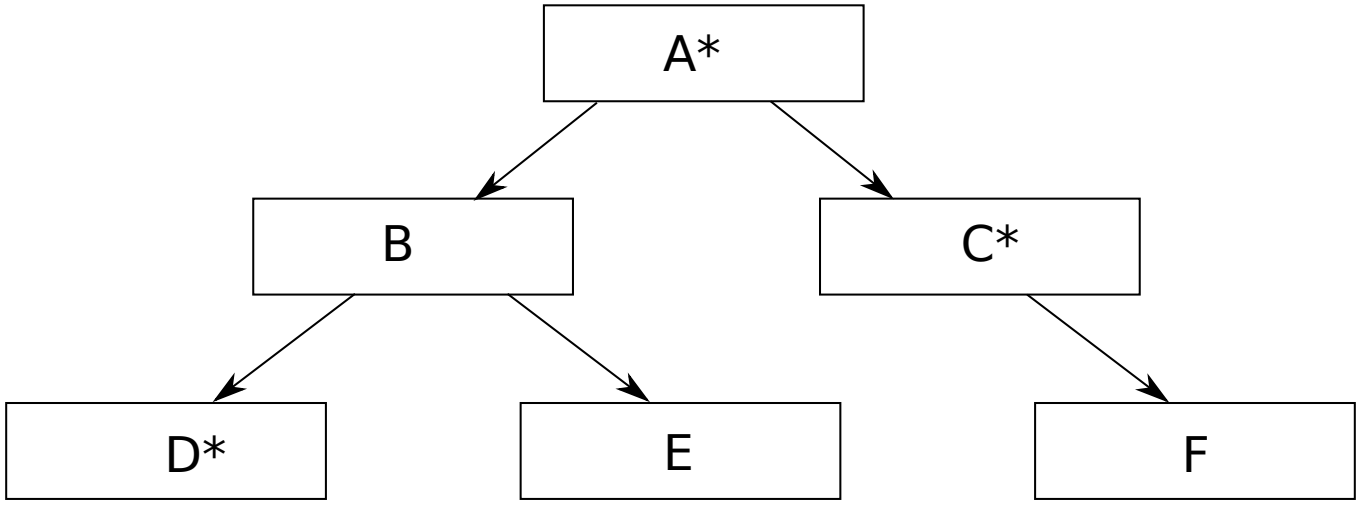
- la compatibilité par affectation entre un type classe et un type ascendant,
- la ligature dynamique des méthodes.

Généralisation à plusieurs classes




```
A a; B b; C c; D d; E e; F f;
// Affectations legales :
a = b; a=c; a=d; a=e; a=f;
b = d; b=e;
c=f;
// Affectations illegales :
b = a; // erreur: A ne descend pas de B
d = c; // erreur: C ne descend pas de D
c = d; // erreur: D ne descend pas de C
```

Généralisation à plusieurs classes




```
A a; a = ...
```

Méthode f appelée selon la nature de l'objet pointé par a:

- a référence un objet de type A : méthode f de A
- a référence un objet de type B : méthode f de A
- a référence un objet de type C : méthode f de C
- a référence un objet de type D : méthode f de D
- a référence un objet de type E : méthode f de A
- a référence un objet de type F : méthode f de C

Exemple d'utilisation du polymorphisme

On veut que la méthode affiche de la classe Point

- Affiche les coordonnées du point
- Affiche la nature du point (normal, coloré...), en appelant une méthode identifie ()

```
Je suis un point
  Mes coordonnees sont 0 2
Je suis un point colore de couleur 3
  Mes coordonnees sont 1 5
```

Exercice

Ecrire les classes Point et PointColore

Exemple d'utilisation du polymorphisme: correction

```
class Point {
  public Point (int x, int y) { this.x = x; this.y = y; }
  public void affiche () {
    identifie();
    System.out.println("Mes coordonnees sont :" + x + " " + y);
  }
  public void identifie() {
    System.out.println("Je suis un point" );
  }
  private int x, y;
}
class Pointcol extends Point {
  public Pointcol (int x, int y, byte couleur) {
    super(x,y);
    this.couleur = couleur;
  }
  public void identifie() {
    System.out.println("Je suis un point colore de couleur "+couleur);
            }
  private byte couleur;
}
```

Exemple d'utilisation du polymorphisme

Exemple

```
class TabPolym
  { public static void main (String[] args)
  { Point [] tabPts = new Point [4];
    tabPts [0] = new Point (0, 2);
    tabPts [1] = new Pointcol (1, 5, (byte) 3);
    tabPts [2] = new Pointcol (2, 8, (byte) 9);
    tabPts [3] = new Point (1, 2);
    for (int i=0; i <  tabPts.length; i++)
        tabPts[i].affiche();
  }
}
```

Valeurs affichées?

Polymorphisme, redéfinition, surdéfinition

Conversion implicite des arguments effectifs:

- Permet de généraliser le fonctionnement des types de base au classes

```
class A {
  public void identite()
{ System.out.println("objet de type A");}
}
class B extends A {...}
// pas de redefinition de identite
class Util {
  static void f(A a)  { a.identite(); }
}
A a = new A(); B b = new B();
Util.f(a); // OK : appel usuel
Util.f(b); // OK : references compatibles
//(-> objet de type A)
```

Polymorphisme, redéfinition, surdéfinition

```
class A {
  public void identite()
  { System.out.println("objet de type A"); }
}
class B extends A {
  public void identite()
  { System.out.println("objet de type B"); }
}
class Util { static void f(A a)  { a.identite(); } }
A a = new A(); B b = new B();
Util.f(b); // (-> objet de type B)
```

- D'abord conversion de b en A.
- L'appel à a.identite() appelle la méthode définie par le type de l'objet réellement référencé.
- **La conversion implicite d'un type dérivé à un type de base n'est donc pas dégradante**

Polymorphisme, redéfinition, surdéfinition

```
class A {...}
class B extends A {...}
class Util {
  static void f(int p, B b) {...}
  static void f(float x, A a) {...}
}
A a = new A(); B b = new B();
int n; float x;
Util.f(n,b); // OK, sans conversion
Util.f(x,a); // OK, sans conversion
Util.f(n,a); // conversion de n en float -> f(float, A)
Util.f(x,b); // conversion de b en A -> f(float, A)
```

Polymorphisme, redéfinition, surdéfinition

```
class A {...}
class B extends A {...}
class Util {
  static void f(int p, A a) {...}
  static void f(float x, B b) {...}
}
A a = new A(); B b = new B();
int n; float x;
Util.f(n,a); // OK, sans conversion
Util.f(x,b); // OK, sans conversion
Util.f(n,b); // erreur de compilation: ambiguite
Util.f(x,a); // erreur de compilation: aucune fonction
// ne convient
```

Polymorphisme: principe de la résolution des types

```
class A {
  public void f (float x) {...}
}
class B extends A {
  public void f (float x)  {...} //redefinition
  public void f (int) {...}//surdefinition (pour A et B)
}
A a = new A(...); B b = new B(...); int n;
a.f(n);// f(float) de A
b.f(n);// f(int) de B
a = b; a.f(n);          // appelle f(float) de B !!!
```

- Au dernier appel, le compilateur cherche la meilleure méthode parmi toutes les méthodes correspondant au type de classe de a (A).
- La signature et le type de retour sont figés
- A l'exécution, le type de l'objet référencé par a (B) est utilisé pour trouver la méthode.

Les règles du polymorphisme

En résumé, le polymorphisme se traduit par :

- La compatibilité par affectation entre un type classe et un type ascendant,
- La ligature dynamique des méthodes.

Compatibilité : Il existe une conversion implicite d'une référence à un objet de classe T en une référence à un objet de classe ascendante de T (affectations et conversion d'arguments effectifs)
Ligature dynamique : Dans un appel `x.f (...)` , le choix de `f` est déterminé :

- *à la compilation* : on détermine dans T ou ses ascendantes la signature de la meilleure méthode f ; signature et valeur de retour sont alors figées,
- *à l'exécution :* on recherche la méthode de signature et de type voulus (avec possibilité de covariance) à partir de la classe correspondant au type effectif de l'objet référencé par x ; si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie (au pire jusqu'à T).

Polymorphisme, exemple

Exemple

```
class A {
  public void f (float x) {...}
  public void g (float x) {...}
}
class B extends A {
  public void f (float x) {...} // redefinition
  public void f (int n) {...} // surdefinition
  public void g (int n) {...}
}
A a = new A(...); B b = new B(...);
int n; float x;
a = b;
a.f(n);          // appelle ?
a.g(x);          // appelle ?
```

Polymorphisme, exemple

Exemple

```
class A {
  public void f (float x) {...}
  public void g (float x) {...}
}
class B extends A {
  public void f (float x) {...} // redefinition
  public void f (int n) {...} // surdefinition
  public void g (int n) {...}
}
A a = new A(...); B b = new B(...);
int n; float x;
a = b;
a.f(n);          // f (float) de B
a.g(x);          // g (float) de A
```

Plan

La super-classe Object

Il existe une super-classe dont dérive implicitement toute classe simple.

- On peut donc utiliser une référence de type objet

```
Point p = new Point();
Object o = p;
o.affiche() // Erreur de compilation
((Point) o).affiche() // Ok
Point p1 = (Point) o;
p1.affiche(); // Ok
```

- On peut toujours utiliser ses méthodes : toString , equals

La super-classe Object

La super-classe Object possède en particulier les deux méthodes: toString et equals.

toString : fournit la chaîne de caractère
<Nom_De_la_Classe>@<Adresse_hexa_de_l_objet>

```
class Point
{ public Point (int abs, int ord) {x=abs;y=ord;}
  private int x,y;
}
class TestToString
{ public static void main (String args[])
  { Point a = new Point(1,2), b = new Point(2,3);
    System.out.println("a = "+a.toString());
    System.out.println("b = "+b.toString());
  }
}
a = Point@fc17aedf
b = Point@fc1baedf
```

La super-classe Object

equals: compare les adresses des deux objets concernés et retourne le booléen correspondant.

```
class Point {
  public boolean equals  (Point p)
  { return((p.x==x) && (p.y==y)); }
}
Point a = new Point(1,2);
Point b = new Point(5,6);
a.equals(b); // Retourne ?
Object o1 = new Point(1,2);
Object o2 = new Point(1,2);
o1.equals(o2); // Retourne ?
((Point)o1).equals(o2); // Retourne ?
((Point)o1).equals((Point)o2); // Retourne ?
```

La super-classe Object

equals: compare les adresses des deux objets concernés et retourne le booléen correspondant.

```
class Point {
  public boolean equals  (Point p)
  { return((p.x==x) && (p.y==y)); }
}
Point a = new Point(1,2);
Point b = new Point(5,6);
a.equals(b); // Retourne false
Object o1 = new Point(1,2);
Object o2 = new Point(1,2);
o1.equals(o2); // Retourne false
((Point)o1).equals(o2); // Retourne false
((Point)o1).equals((Point)o2); // Retourne true
```

Plan

Membres protégés, classes et méthodes finales

Membre protégé :

- Un membre déclaré **protected** est accessible aux classes dérivées de sa classe, ainsi qu'aux membres du même paquetage.

Classes et méthodes finales :

- Une méthode déclarée **final** ne peut pas être redéfinie dans les classes dérivées.
- Une classe déclarée **final** ne peut pas être dérivée.

Plan

Les classes abstraites

Une classe abstraite ne permet pas d'instancier des objets mais sert de base pour créer des classes dérivées.
Formée par :

- méthodes et champs habituels
- méthodes abstraites : seulement la signature et le type de retour.

```
abstract class A {
  public void f() {...}
  public abstract void g (int n) ;
}
class B extends A {
  public void g (int n) {...} // definition de g
  ...
}
A a; // OK
A a = new A(...); // erreur
A a = new B(...); //OK
```

Les classes abstraites : règles

1. Dès qu'une classe contient une méthode abstraite, elle est abstraite
2. Une méthode abstraite doit obligatoirement être publique
3. Les noms des arguments muets doivent figurer dans la signature d'une méthode déclarée abstraite
4. Une classe dérivée d'une classe abstraite peut être déclarée abstraite
5. Si une classe dérivée d'une classe abstraite ne redéfinit pas toutes les méthodes abstraites de sa classe de base, elle est elle-même abstraite

Les classes abstraites : utilisation

Le recours aux classes abstraites facilite la conception orientée objet :
Permet de placer toutes les fonctionnalités qu'on veut faire figurer dans les classes dérivées.

- Soit sous forme d'une implémentation complète de méthodes (non-abstraites) et de champs, lorsqu'ils sont partagés par toutes ses descendantes
- Soit sous forme d'une méthode abstraite, dont l'implémentation varie selon les classes dérivées, mais pour s'assurer qu'elle sera présente dans toute classe dérivée instanciable.

$\rightsquigarrow$ très utile pour le polymorphisme

Plan

Les interfaces

Une interface peut être vue comme une classe abstraite n'implantant aucune méthode et aucun champ (hormis les constantes).

1. Une classe pourra implanter plusieurs interfaces
2. La notion d'interface se supperpose à celle de dérivation (les interfaces peuvent se dériver).
3. On peut utiliser des variables de type interface

```
public interface I1
{ void f (int n);//attributs public, abstract
  void g(float x); // facultatifs
}
```

Définition d'une classe qui implémente une interface:

```
public class A implements I1
{
  void f(int n){...}//implementation necessaire de f
  void g(float x){...}//implementation necessaire de g
}
```

Les interfaces

```
import java.Math;
interface I1 { void identifie(); }
interface I2 { double DistOrig(); }
class Point implements I1 {
  public Point (int x, int y){this.x = x; this.y = y;}
  public void affiche ()
  { identifie();
    System.out.println("Mes coordonnees sont : "+x+" "+y);}
  public void identifie() {
    System.out.println("Je suis un point" ); }
  private int x, y; }
class Pointcol extends Point implements I2
{ public Pointcol (int x, int y, byte couleur) {
    super(x,y); this.couleur = couleur; }
  public void identifie() {
    System.out.println("Je suis un point colore de couleur "+couleur);
    System.out.println("Ma distance a l'origine est : "+DistOrig()); }
  public double DistOrig() {
    return (Math.sqrt (Math.pow((double)this.x,2.0)+
                       Math.pow((double)this.y,2.0)) );
  }
  private byte couleur;
}
...
```

Les interfaces

Une interface peut aussi servir directement de type classe pour des variables (pour le polymorphisme).

```
interface Affichable { void affiche(); }
class Entier implements Affichable {
  public Entier (int n) { valeur = n; }
  public void affiche ()
  {System.out.println("Je suis un entier de valeur:"+valeur);}
  private int valeur;
}
class Flottant implements Affichable {
  public Flottant (float x) { valeur = x; }
  public void affiche ()
  {System.out.println("Je suis un flottant de valeur:"+valeur)
  private float valeur;
}
public class Tabhet {
  public static void main (String args[]) {
    Affichable [] tab = new Affichable[2];
    tab[0] = new Entier(27); tab[1] = new Flottant(1.5f);
    for (int i=0;i<tab.length;i++) { tab[i].affiche(); }
} }
```

Les interfaces

- On ne peut affecter aux éléments du tableau une référence à qqch du type interface, mais on peut affecter une référence à un objet d'une classe qui implémente une interface.
- Une interface contient des en-têtes de méthodes et des **constantes symboliques** (**static final**) accessibles à toutes les classes qui implémentent l'interface.

Les interfaces

- Une classe peut implémenter plusieurs interfaces

```
interface I1{void f(); void g();}
interface I2{void g(int n);}
public class A implements I1, I2
{ void f(){...}
  void g(){...}
  void g(int n){...}
}
```

- Conflit de noms :

```
interface I1{void f();}
interface I2{int f();}
public class A implements I1, I2
{ ... } // Erreur de compilation
```

L'interface Cloneable

- La classe Object possède une méthode clone protégée qui effectue une copie **superficielle** de l'objet.
- L'interface Cloneable n'impose pas de redéfinir la méthode clone, mais mentionne qu'elle sera redéfinie.
- Dans la re-définition de la méthode clone, on peut alors faire une **copie en profondeur**
- Un appel à la méthode clone d'une classe qui n'implémente pas l'interface Cloneable conduit à la levée de l'exception CloneNotSupportedException
- L'entête de clone est

```
Object clone();
```

  $\rightsquigarrow$ utiliser un cast vers le type désiré.

Exemple de clonage

```
class Vide {
  private int dummy = 7;
}
class Entier implements Cloneable{// pour appel sans lever
↪  d'exception
  public Entier(int a) { this.a = a; this.v = new Vide(); }
  public int a;
  public Vide v;

  public Object clone() { throws CloneNotSupportedException
    return super.clone(); // utilisation de la méthode héritée
  }
}
public class Clone {
  public static void main(String args[]) throws
  ↪  CloneNotSupportedException {
    Entier e1, e2;
    e1 = new Entier(8);
    e2 = (Entier)e1.clone();
    System.out.println(e1.toString()); // Entier@4d591d15
    System.out.println(e2.toString()); // Entier@65ae6ba4
    System.out.println(e1.a); // 8
    System.out.println(e2.a); // 8
    System.out.println(e1.v.toString()); // Vide@48cf768c
    System.out.println(e2.v.toString()); // Vide@48cf768c
  }
```

Plan

Les classes enveloppes

Les objets (instances de classes) et les variables (de types primitifs) ne se comportent pas toujours de la même façon :

- l'affectation porte sur l'adresse des objets, et sur la valeur des variables
- hiérarchie d'héritage pour les objets, de type pour les variables
- le polymorphisme ne s'applique qu'aux objets

Les *classes enveloppes* permettent de manipuler des types primitifs comme des objets:

Boolean	Character	Byte	Short	Integer	Long	Float	Double
boolean	char	byte	short	int	long	float	double

Ces classes sont finales et inaltérables.

Les classes enveloppes

- Les classes enveloppes possèdent toutes
  - un constructeur avec le type primitif correspondant
  - un accesseur <type>Value vers la valeur dans le type primitif
- comparaison avec la méthode equals, ou avec ==

```
// possible mais déprécié
Integer nObj = new Integer(12);
Double xObj = new Double(5.25);
// conseillé
Integer nObj2 = Integer.valueOf(12);
Double xObj2 = Double.valueOf(5.25);
// ou même
Integer nObj3 = 12;
Double xObj3 = 5.25;
// Exemple d'utilisation
int n = nObj.intValue();
double x = xObj.doubleValue();
int xn = xObj.intValue();
```

Les Classes anonymes

Java permet de définir ponctuellement une classe sans lui définir de nom.

```
class A { public void affiche()
  { System.out.println("Je suis un A");} }

public class Anonym {
  public static void main(String[] ags) {
    A a;
    a = new A()  {
      public void affiche() {
        System.out.println(" Je suis un anonyme
        ↪  derive de A");
      }
    }          ;
    a.affiche(); // !!! Exploitation du
    ↪  polymorphisme
  }
}
```

Élement de conception des classes

Respect du contrat :

Une classe constitue un contrat défini par les interfaces des méthodes publiques (signature et valeur de retour) et leur sémantique. En principe, ce contrat doit être respecté en cas de dérivation : lorsqu'on surdéfinit une méthode, on s'arrange pour en conserver la sémantique.

Relation entre classes :

L'héritage crée une relation de type **est**. L'utilisation d'objets membres crée une relation de type **a** (un objet membre est un champ qui est un objet).

Interface et héritage :

On dit souvent que Java ne possède pas d'héritage multiple, mais que celui-ci peut être remplacé par l'utilisation d'interfaces. Ce jugement n'est pas entièrement vrai car une interface ne fournit que des en-têtes !

Plan

Plan

Gestion des exceptions

Situations exceptionnelles:

- Erreur de saisie clavier
- Division par 0
- échec d'un algorithme probabiliste
- ...

Idée 1: *Laisser le programmeur les traiter de façon explicite*

- Omissions possibles
- Codes confus et rapidement illisible
- Difficile de traiter le problème autre part que dans la fonction créant le problème
- Paradigme des valeurs de retour (9 ou -1): inapplicable aux constructeurs; utilisation de variable globale peu satisfaisante

Gestion des exceptions

Situations exceptionnelles:

- Erreur de saisie clavier
- Division par 0
- échec d'un algorithme probabiliste
- ...

Idée 2: Traitement par exceptions

- Distinction entre le contexte de détection et le contexte du traitement
- Séparation du traitement des exception du reste du code: meilleure lisibilité

Exemple

```
class ErrCoord extends Exception {}
class Point {
  public Point (int x, int y) throws ErrCoord {
    if ( (x < 0) || (y < 0) ) throw new ErrCoord();
    this.x = x; this.y = y; }
  public void affiche () {
    System.out.println("Coordonnees : " + x " " + y); }
  private int x, int y; }
public class Except1 {
  public static void main (String args[]) {
    try {
      Point a = new Point(1,4); a.affiche();
      a = new Point(-3,5); a.affiche();
    }
    catch(ErrCoord e) {
      System.out.println("Erreur construction");
      System.exit(-1);
    }
  }
}
```

Exceptions: déclenchement

Déclenchement de l'exception: **throw** suivi d'une référence vers une instance d'une classe identifiant l'exception

```
throw new ErrCoord();
```

Classes d'exception: doivent dériver de la super-classe Exception

```
class ErrCoord extends Exception{}
```

Spécification: Les méthodes **doivent** spécifier quelles exceptions sont susceptible d'être levées pendant leur exécution.

```
public Point (int x, int y) throws ErrCoord {...}
```

Exceptions: réception

```
try{
   ...
}
catch (ErrCoord e){
   ...   // bloc catch
}
```

le bloc **try**: définit la zone où la levée d'exception est attendue.

le bloc **catch**: le gestionnaire d'exception qui définit le comportement lorsque une exception est levée. Permet de définir des comportements différents pour une même exception selon le contexte où il est écrit.

Les blocs **try** et **catch** doivent être contigus.

Gestion de plusieurs exceptions

```
// methode de la classe Point
public void deplace (int dx, int dy) throws ErrDepl
{ if ( (x+dx < 0) || (y+dy < 0) ) throw new ErrDepl();
  x = x + dx; y = y + dy;
}
class ErrDepl extends Exception { }

public class Except2 {
  public static void main (String args[]) {
    try
    { Point a = new Point(1,4); a.affiche();
      a.deplace(-3,5); a.affiche();
    }
    catch(ErrCoord e)
    { System.out.println("Erreur construction");
      System.exit(-1);
    }
    catch(ErrDepl e)
    { System.out.println("Erreur deplacement");
      System.exit(-1);
    }
} }
```

Gestion de plusieurs exception

Si une méthode est suceptible de lever plusieurs exceptions, il faut nécessairement toute les traiter:

- Soit par un bloc **catch**
- Soit par une spécification **throws** de la méthodes appelante

Transmission d'information au gestionnaire

1. Par l'objet soumis à **throw**:

```
public point (int x, int y) throws ErrCoord {
  // constructeur de la classe Point
  if ( (x < 0) || (y < 0) ) throw new ErrCoord(x,y);
  this.x = x; y = this.y;
}

class ErrCoord extends Exception {
  ErrCoord(int abs, int ord)
  { this.abs=abs; this.ord=ord; }
  public int abs, ord;
}
...
// bloc catch de la methode main
catch(ErrCoord e) {
  System.out.println("Erreur construction " + e.abs
                     + " " + e.ord);
  System.exit(-1);
}
```

Transmission d'information au gestionnaire

2. Par un message au constructeur dérivé de Exception

```
public point (int x, int y) throws ErrCoord
{ if ( (x < 0) || (y < 0) )
      throw new ErrCoord("Erreur construction avec "
                              + x + " " + y);
   this.x = x; this.y = y;
}
class ErrCoord extends Exception {
  ErrCoord(String mes)
  { super(mes); }
  // appel du constructeur a un argument de type
  // string de Exception
}
...
catch(ErrCoord e) { // bloc catch de la methode main
  // appel de la methode getMessage de Exception
  System.out.println(e.getMessage());
  System.exit(-1);
}
```

Utilisation pour des comportements non exceptionnels

```
while (true)
{
  try
  {
    z = ZnZ.element_aleatoire();
    z.inverse();
    break;
  }
  catch(PasInversible e)
  {
    continue;
  }
}
```

- C'est un comportement attendu de rentrer dans le bloc `catch`.
- Ce n'est plus une "erreur" à traiter

Plan

Propriétés du gestionnaire d'exception

Le bloc **catch** est un bloc de code, pas une méthode

- Accès aux variables du contexte englobant
- **return** fait sortir de la méthode courrante

Mécanisme de gestion des exceptions:

- Après exécution d'un bloc **catch**, en l'absence de **return** ou **exit**, l'exécution se poursuit après le dernier bloc **catch**,
- Recherche du bloc **catch** dans l'ordre où ils apparaissent ; on sélectionne le premier ayant le type exact ou un type parent de l'exception levée.

```
class ErrPoint extends Exception {...}
class ErrCoord extends ErrPoint {...}
class ErrDepl extends ErrPoint {...}
catch (ErrPoint e) {...}
```

Exceptions et héritage

Permet de regrouper certains traitements:

```
try
{ ... }
catch (ErrCoord e) {...} // Exception de coordonnees
catch (ErrPoint e) {...} // Toutes les autres
```

Attention:

```
try
{ ... }
catch (ErrPoint e) {...} // Toutes les autres
catch (ErrCoord e) {...} // Exception de coordonnees
```

génère une erreur de compilation : le deuxième bloc **catch** n'est pas atteignable.

Règles des exceptions

1. Toute méthode qui ne traite pas une exception qui peut être levée durant son exécution doit mentionner son type dans une clause **throws**
2. Re-déclenchement d'une exception: permet d'effectuer un traitement, sans interrompre la chaîne de levée d'exception.

```
try{...}
catch(ErrPoint e)
{  ...
   throw e;
}
```

Mot clé finally

3 Mot clé **finally**

- placé après le dernier gestionnaire **catch**
- code exécuté après les gestionnaires, mais avant la remontée au bloc supérieur (si aucun gestionnaire trouvé)
- utile pour les libérations de ressource en cas d'interruption de l'exécution par une exception non récupérée

```
void f () throws Ex
{
  ...
  try
  { ... // ouvre un fichier
    ... // peut lever Ex
  }
  finally
  { ... // ferme le fichier }

}
```

Plan

Les exceptions standards

Java fournit des classes standards d'exception, dérivées de la super-classe Exception. Elles peuvent être

explicites: comme précédemment: gérées par un bloc **catch** ou spécifiées par une clause **throws**

implicites: ne doivent pas être spécifiées par **throws**, on peut les traiter par **catch** mais on n'est pas obligé.

Les exceptions standards

Exemple: paquetage Java.lang

Explicites:

```
Exception
    ClassNotFoundException
    NoSuchMethodException
    ...
```

Implicites:

```
Exception
    RunTimeException
        ArithmeticException
        IndexOutOfBoundsException
            ArrayIndexOutOfBounds
            StringIndexOutOfBounds
        NegativeArraySizeException,
        NullPointerException,
        ...
```

Les exceptions standards

Exemple: paquetage Java.io

Toutes explicites:

```
Exception
    IOException
        CharConversionException
        EOFException
        FileNotFoundException
        InterruptedIOException
        ObjectStreamException
            InvalidClassException
            InvalidObjectException
            ...
        SyncFailedException
        UnsupportedEncodingException
        UTFDataFormatException
```

Plan

Threads: introduction

Motivation: exécution *simultanée*

- Tirer parti du parallélisme,
- Illusion de la simultanéité d'exécution de plusieurs programmes sur un processeur
- Problèmes d'accès à des ressources lointaines (réseau, accès disque, scanner, rendu graphique...), ou attente de saisie par l'utilisateur $\rightsquigarrow$ bloquant l'exécution

Plan

Classe Thread

- Un thread est décrit par une classe,
- Qui hérite de la super-classe Threads
- Une méthode run: décrit le programme à exécuter
- L'exécution est lancée par la méthode héritée start :
  - S'occupe de la création du thread (au niveau de la machine virtuelle et du système)
  - lance le code de la méthode run
- une méthode statique sleep qui met le thread en attente

Exemple

```
class Ecrit extends Thread {
  public Ecrit (String texte, int nb, long attente)
  { this.texte = texte ;
    this.nb = nb;
    this.attente = attente ; }
  public void run ()
  { try {
      for (int i=0; i<nb; i++) {
        System.out.print (texte) ;
        sleep (attente) ;
    } }
    catch (InterruptedException e) { }
    // impose par sleep
  }
  private String texte ;
  private int nb;
  private long attente ;
}
```

Exemple

```
public class TstThrd {
  public static void main (String args[])
  { Ecrit e1 = new Ecrit ("bonjour ", 5, 5);
    Ecrit e2 = new Ecrit ("bonsoir ", 7, 10);
    Ecrit e3 = new Ecrit ("\n", 3, 15);
    e1.start();
    e2.start(); // lance la methode start
    e3.start(); // de la classe Thread
        }
}
```

Affichage possible:

```
Bonjour Bonsoir Bonsoir Bonjour
Bonjour Bonjour Bonsoir
Bonsoir Bonsoir
Bonjour Bonsoir Bonjour Bonsoir Bonsoir
```

Remarques

- Un programme comporte toujours au moins un thread principal: (méthode main)
- Un appel direct aux méthodes run ne créé aucun thread $\rightsquigarrow$ pas de comportement simultané
- Une méthode start ne peut être appelée qu'une seule foi pour un thread
- sleep est une méthode statique qui met en sommeil le thread courant

L'interface Runnable

Limitation de l'approche *dérivation de Threads*:

- Héritage multiple impossible
- Impossible de faire des threads dans une hiérarchie de classes existante
- Pour l'héritage multiple: les interfaces

L'interface Runnable

- une seule méthode: run
- on passe alors l'objet en argument au constructeur de la classe Thread
- appel de start sur un objet de la classe Thread

Exemple

```
class Ecrit implements Runnable {
  public Ecrit (String texte, int nb, long attente)
  { // meme contenu que precedemment }
  public void run ()
  { try {
      for (int i=0; i<nb; i++) {
        System.out.print (texte) ;
        Thread.sleep (attente) ;
      }
    }
    catch (InterruptedException e) { }
  }
}

Ecrit e1 = new Ecrit ("bonjour ", 5, 5) ;
// cree un objet Thread associe a e1
Thread t1 = new Thread(e1);
t1.start();
```

Exemple

On peut naturellement doter la classe Ecrit d'une méthode start qui

- créé un objet Thread
- lance le thread associé

```
public void start () {
  Thread t = new Thread(this);
  t.start(); }
```

Plan

Interruption d'un thread

Un thread peut interrompre un autre thread:

- en invoquant la méthode interrupt du thread à interrompre
- qui fait positionner un indicateur marquant une demande de terminaison
- libre au thread de lire cet indicateur (méthode interrupted () de la class Thread pour décider de s'interrompre

Exemple

Tread 1

```
t.interrupt();
// positionne un
// indicateur dans t
```

Thread 2 nommé t

```
run
{ ...
  if (interrupted())
  { ...
    return;//fin du thread
  }
```

Exemple

```
class FiboThread extends Thread {
    private long max;
    public FiboThread (long max){this.max=max;}
    private long fibo(long n){
        if (n<=1) return 1;
        else return fibo(n-1)+fibo(n-2); }
    public void run(){
        for (long i = 0; i < max; i++){
            if (interrupted()){
                System.out.println("signal recu");
                return;  }
            System.out.println (fibo (i));
        } } }
public static void main(String[] args){
  FiboThread e1 = new FiboThread(Integer.parseInt(args[0]));
  e1.start();
  System.out.println("Thread launched");
  try{Thread.sleep(4000);}
  catch (InterruptedException e){}
  e1.interrupt();
}
```

Interruption d'un thread

Remarques:

- La terminaison d'un thread reste de la responsabilité du thread lui-même
- La méthode interrupted est statique: elle porte sur le thread courant
- A la place on peut utiliser la méthode non-statique IsInterrupted pour lire l'indicateur
- L'appel à interrupted remet l'indicateur en position initiale
- Les méthodes sleep et wait examinent aussi l'indicateur, et lèvent l'exception InterruptedException si celui-ci est positionné.

Thread démons et arrêt brutal

Deux catégories de threads:

- threads utilisateurs (par défaut)
- threads démons
  - Si il ne reste plus que des thread démons actifs, le programme s'arrête
  - obtenus par setDaemon(**true**) avant l'appel à start
- Tout thread est de la catégorie du thread qui l'a créé

```
public class Test
{ public static void main(String args[])
  { Ecrit e=Ecrit("Bonjour", 5, 4);
    e.setDaemon(true);
    e.start();
  }
}
```

Plan

Coordinations des threads

Threads vs Processus:

Avantage: permet de partager des données

Inconvénient: éviter les situations de blocage (lecture/écriture simultanée)

$\rightsquigarrow$ coordination par

- synchronisation (protection des accès)
- attente et notification (dépendance entre tâches)

Exemple

Itérativement,

- *incrément d'un nombre x et calcul de son carré y*
- *affichage de x et y*

Synchronisation : exemple

```
public class Synchr1
{ public static void main (String args[])
  { Nombres nomb = new Nombres();
    Thread calc = new ThrCalc (nomb);
    Thread aff  = new ThrAff (nomb);
    calc.start() ; aff.start();
    Clavier.lireString();
    calc.interrupt() ; aff.interrupt();
  }
}
class Nombres
{ public synchronized void calcul()
  { n++ ;
    carre = n*n ;
  }
  public synchronized void affiche ()
  { System.out.println (n + " a pour carre " + carre);
  }
  private int n=0, carre;
}
```

Synchronisation : exemple

```
class ThrCalc extends Thread
{ public ThrCalc (Nombres nomb)
  { this.nomb = nomb ; }
  public void run ()
  { try
    { while (!interrupted())
      { nomb.calcul (); sleep (50);
      } }
    catch (InterruptedException e) {} }
  private Nombres nomb;
}
class ThrAff extends Thread
{ public ThrAff (Nombres nomb)
  { this.nomb = nomb; }
   public void run ()
  { try
    { while (!interrupted())
      { nomb.affiche(); sleep (75) ;
      } }
    catch (InterruptedException e) {} }
  private Nombres nomb;
}
```

Synchronisation

Mot clé synchronized : attribut de méthodes d'un objet assurant qu'une seule de ces méthodes pourra être exécutée à la foi.

Remarques :

- On ne synchronise pas l'activité des threads, mais l'accès à l'objet partagé
- Une méthode synchronisée appartient à un objet quelconque, pas nécessairement à un thread

Notion de verrou :

- Un verrou est attribué à la méthode synchronisée appelée sur un objet
- Restitué à la sortie de la méthode
- Tant qu'il n'est pas restitué, aucune méthode synchronisée ne peut le recevoir
- Ne concerne pas les méthodes non synchronisées

Synchronisation

L'instruction synchronized :
Contraintes des méthodes synchronisées:

1. Une méthode synchronisée obtient nécessairement le verrou sur l'objet par lequel elle a été appelée
2. L'objet est verrouillé pour toute la durée de l'exécution de la méthode

Pour plus de souplesse: l'instruction **synchronized**:

- obtient un verrou sur un objet quelconque (passé en argument)
- seulement sur la porté du bloc de code suivant

```
synchronized (objet) { // bloc de code }
```

L'interblocage

Étreinte mortelle:

- Le tread t1 possède un verrou sur l'objet o1 et attend celui de o2
- Le tread t2 possède un verrou sur l'objet o2 et attend celui de o1

Solution?

- Impossible à détecter en Java
- Rôle du programmeur...
- Une stratégie: numéroter les verrous et les demander par ordre croissant uniquement

Attente et notification

Coordination de threads dans le cas de dépendance: *un thread a besoin d'un résultat calculé par un autre pour démarrer son calcul*

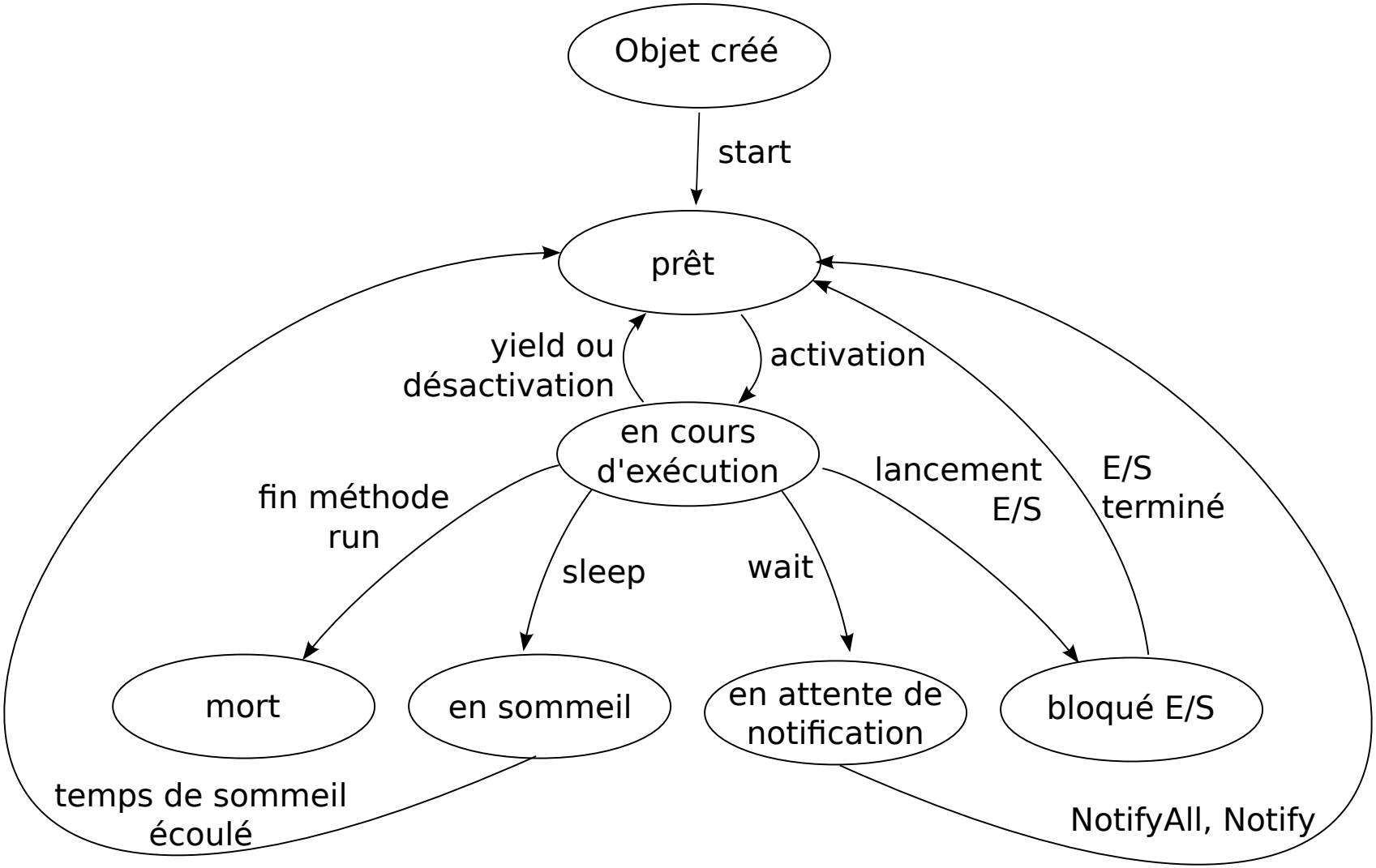
Une méthode synchronisée peut :

- appeler la méthode wait de l'objet dont elle a le verrou:
  - Rend le verrou à l'environnement
  - Met en attente le thread correspondant (plusieurs thread peuvent être en attente sur le même objet)
- appeler la méthode NotifyAll d'un objet pour prévenir tous les threads en attente sur cet objet et leur donner la possibilité de s'exécuter (le mécanisme utilisé et le thread effectivement sélectionné peuvent dépendre de l'environnement)

Exemple

```
public class Synchr1
{ public static void main (String args[])
  { Nombres nomb = new Nombres() ;
    Thread calc = new ThrCalc (nomb) ;
    Thread aff  = new ThrAff (nomb) ;
    calc.start() ; aff.start() ;
    Clavier.lireString() ;
    calc.interrupt() ; aff.interrupt() ;
} }
class Nombres
{ public synchronized void calcul() throws InterruptedException
  { if (!pret) { n++; carre=n*n; pret=true; notifyAll(); }
    else wait(); }
  public synchronized void affiche () throws
  ↪  InterruptedException
  { if (pret)
    { System.out.println (n+" a pour carre "+carre);
      pret=false; notifyAll(); }
    else wait(); }
  private int n=0, carre ;
  private boolean pret = false ;
}
```

États d'un thread



Priorité d'un thread

Priorité d'un thread

- Changement de la priorité par SetPriority
- Comprise entre MinPriority (1) et MaxPriority (10)
- Utilisé par l'environnement d'exécution pour choisir le thread à qui donner la main
- Si un thread plus prioritaire que le thread courant devient prêt, on lui donne la main.
- Peu conseillé d'agir sur les priorité dans des programmes portables (dépend de l'environnement d'exécution)

Plan

Plan

Programmation graphique

Programme à interface console

- l'utilisateur se laisse guider par le programme
- dialogue **séquentiel**
- En mode texte

Programme à interface graphique

- l'utilisateur donne des ordres (clics sur menus, boutons, ...)
- comportement **événementiel** du programme
- aspect graphique des contrôles (boutons, fenêtre, boîtes, menus déroulants, ...)

Une première fenêtre

- Utilisation de l'API Swing (préférable à AWP depuis Java2)
- La classe `JFrame` (constructeur sans argument) permet de créer des objets de type fenêtre (pouvant être retaillées, déplacées, réduites, ...)
- Une fenêtre console est créée en parallèle

```
import javax.swing.* ;
public class Premfen0
{ public static void main (String args[])
  { JFrame fen = new JFrame() ;
    fen.setSize (300, 150) ;
    fen.setTitle ("Ma premiere fenetre") ;
    fen.setVisible (true) ;
  }
}
```

Durée de vie d'une fenêtre

- un thread utilisateur est créé pour la fenêtre
- le thread principal est reservé au main
- a la fin du main, le thread principal meurt, mais le thread utilisateur persiste.

Conséquence sur la terminaison du programme:

- terminaison lorsque le thread utilisateur meurt
- la fermeture de la fenêtre (clic sur la croix, ...) ne tue pas le thread utilisateur
- pour l'instant : taper Ctrl-c sous Unix/Linux ou en fermant la fenêtre console sous Windows.
- plus tard: gestion de l'événement fermeture

Durée de vie d'une fenêtre

```
import javax.swing.* ;
class MaFenetre extends JFrame
{ public MaFenetre ()    // constructeur
  { setTitle ("Ma premiere fenetre") ;
    setSize (300, 150) ;
  }
}
public class Premfen
{ public static void main (String args[])
  { JFrame fen = new MaFenetre() ;
    fen.setVisible (true) ;
    while (true)  // fin sur longueur titre nulle
    { System.out.print ("nouvelle largeur : ") ;
      int larg = Clavier.lireInt() ;
      System.out.print ("nouvelle hauteur : ") ;
      int haut = Clavier.lireInt() ;
      System.out.print ("nouveau titre : (vide pour finir) ");
      String tit = Clavier.lireString() ;
      if (tit.length() == 0) break ;
      fen.setSize (larg, haut) ;
      fen.setTitle(tit) ;
    } } }
```

Gestion d'un clic dans une fenêtre

Un événement: le clic sur la fenêtre

- Objet **source**: la fenêtre
- On associe à la source un objet **écouteur**
- La classe de l'écouteur doit implémenter une **interface** de **catégorie d'événement**: ici, `MouseListener`
- Chaque méthode de l'interface est un événement possible de cette catégorie
- `MouseListener` comporte cinq méthode:

```
class EcouteurSouris implements MouseListener
{ public void mouseClicked (MouseEvent ev) {...}
  public void mousePressed (MouseEvent ev) {...}
  public void mouseReleased(MouseEvent ev) {...}
  public void mouseEntered (MouseEvent ev) {...}
  public void mouseExited (MouseEvent ev) {...}
  ...
}
```

Gestion d'un clic dans une fenêtre

- Pour associer l'écouteur à l'objet source: méthode `AddMouseListener(ecouteur)`.
- ici, l'objet fenêtre sera son propre écouteur.

```
import javax.swing.*;
import java.awt.event.*;//pour MouseEvent et MouseListener
class MaFenetre extends JFrame implements MouseListener
{ public MaFenetre ()// constructeur
  { setTitle ("Gestion de clics");setBounds (10,20,300,200);
    addMouseListener (this);// la fenetre sera son propre
                            //  ecouteur d'evenements souris
  }
  public void mouseClicked(MouseEvent ev)
  { System.out.println ("clic dans fenetre");}
  public void mousePressed (MouseEvent ev) {}
  public void mouseReleased(MouseEvent ev) {}
  public void mouseEntered (MouseEvent ev) {}
  public void mouseExited  (MouseEvent ev) {}
}
public class Clic1
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ; } }
```

Utilisation des informations de l'objet événement

La classe `MouseEvent` fournit des informations sur l'événement déclenché.

```
...
public void mouseClicked(MouseEvent ev)
  // objet de classe MouseEvent est cree par Java
  // et transmis a l'ecouteur voulu
  { int x = ev.getX();
    int y = ev.getY();
    //coordonnees du curseur au moment du clic
    System.out.println ("clic au point de coordonnees "
                        + x + ", " + y ) ;
  }
...
```

Adaptateurs

- Pour éviter d'implémenter tous les événements de l'interface `MouseListener`, l'**adaptateur** `MouseAdapter` est une classe fournissant une implémentation vide de cette interface.
- Il suffit alors de re-définir les méthodes voulues

```
class EcouteurSouris extends MouseAdapter
{
  public void mouseClicked (MouseEvent e) {...}
}
class MaFenetre extends JFrame
{
  MaFenetre(){
    addMouseListener (new EcouteurSouris());
    ...
  }
}
```

Adaptateurs

Si l'on veut que la fenêtre soit son propre gestionnaire d'événements (comme précédemment):

- Problème: `MaFenetre` ne peut hériter de `JFrame` et `MouseAdapter` à la foi
- Résolu en utilisant les classes annonymes:

```
class MaFenetre extends JFrame
{ public MaFenetre()
  { ...
    addMouseListener(new MouseAdapter {
      public mouseClicked (MouseEvent e)
      {...}
    } );
    ...
  }
}
```

La gestion des événements en général

- Un événement déclenché par un objet nommé *Source*, peut être traité par un objet nommé *Écouteur*, préalablement associé à la source.
- A une catégorie **`Xxx`** d'événement donnée, on pourra toujours associer un objet écouteur d'événements **`xxxEvent`** par une méthode **`AddXxxListener`**.
- On pourra alors soit définir toutes les méthodes de l'interface **`XxxListener`**, soit dériver une classe de l'adaptateur **`XxxAdapter`**.
- Objet source et objet écouteur peuvent être identiques, et un même événement peut disposer de plusieurs écouteurs.
- Tous les événements sont pris automatiquement en charge par Java et subissent un traitement par défaut.

Exemple : fin du programme sur fermeture

- L'interface `WindowListener` définit les méthodes à implémenter pour la gestion des événements d'une fenêtre
- La méthode `windowClosing` est appelée à la fermeture d'une fenêtre $\rightsquigarrow$

```
class FermFen extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.out.println("Fermeture...");
    System.exit(0);
  }
}
```

Exemple : suite

```
class Fen extends JFrame
{ public Fen ()     // constructeur
  { setTitle ("Fermable"); setSize (300, 200);
    addWindowListener(new FermFen());
  }
}
public class FenClose {
  public static void main (String args[]) {
    Fen fen = new Fen();
    fen.setVisible(true);
  }
}
```

Exercice: écrire un programme qui crée deux fenêtres et se termine quand elles ont toutes deux été fermées

Plan

Un premier composant : le bouton

Création d'un bouton:

```
JButton monBouton = new JButton ("Cliquez moi");
```

Insertion du bouton au contenu de la fenêtre:

```
Container c = getContentPane();
c.add (monBouton);
```

Changement du gestionnaire de mise en forme de l'objet contenu pour améliorer le positionement du bouton

```
c.setLayout(new FlowLayout());
```

Remarques:

- Par défaut un bouton est visible (contrairement aux fenêtres)
- Requiert les paquetages `awt` et `awt.event`.

Gestionnaire d'événement associé:

- catégorie `Action`, interface `actionListener` qui n'implémente qu'une méthode: `actionPerformed`

Gestion du bouton avec écouteur

```
import javax.swing.*; import java.awt.event.*;
import java.awt.*;
class FenButt extends JFrame implements ActionListener
{ public FenButt ()     // constructeur
  { setTitle ("Premier bouton"); setSize (300, 200);
    monBouton = new JButton ("ESSAI");
    getContentPane().setLayout (new FlowLayout());
    getContentPane().add (monBouton);
    monBouton.addActionListener (this);
  }
  public void actionPerformed(ActionEvent ev)
  { System.out.println("Action sur bouton ESSAI"); }
  private JButton monBouton;
}
public class Bouton {
  public static void main (String args[]) {
    FenButt fen = new FenButt();
    fen.setVisible(true);
  }
}
```

Gestion de plusieurs composants

```
import javax.swing.*; import java.awt.event.*;
import java.awt.*;
class FenButt extends JFrame implements ActionListener
{ public FenButt ()     // constructeur
  { setTitle ("Plusieurs boutons"); setSize (300, 200);
    monBouton1 = new JButton("A");
    monBouton2 = new JButton("B");
    Container contenu = getContentPane();
    contenu.setLayout (new FlowLayout());
    contenu.add (monBouton1); contenu.add(monBouton2);
    monBouton1.addActionListener(this);
    monBouton2.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ev)
  {System.out.println("Action sur un bouton");}
  private JButton monBouton1, monBouton2;
}
public class Bouton {
   public static void main (String args[]) {...}
}
```

Gestion de plusieurs composants

Traitement séparé selon les composants : la méthode `getSource`

- s'applique à tous les événements, pour tous les composants.
- retourne une référence vers l'objet ayant généré l'événement

```
...
public void actionPerformed(ActionEvent ev) {
  if (ev.getSource() == monBouton1)
  { System.out.println "action sur bouton 1");}
  if (ev.getSource() == monBouton2)
  {System.out.println("action sur bouton 2");}
}
...
```

Gestion de plusieurs composants

Alternative: la méthode `getActionCommand`

- Une autre façon d'identifier la source de l'événement
- Ne s'applique qu'aux événements de la catégorie *Action*
- C'est une méthode de la classe `actionEvent`
- Tout événement est caractérisé par une chaîne de caractère (par ex. l'étiquette du bouton)

```
...
public void actionPerformed(ActionEvent ev) {
   String nom = ev.getActionCommand();
   System.out.println("action sur bouton " + nom);
}
...
```

Gestion de plusieurs composants

Quand l'objet *Ecouteur* est différent de l'objet fenêtre: avec autant d'écouteurs que de boutons,

```
class Fen2Boutons extends JFrame
{ public Fen2Boutons ()
  { setTitle ("Avec deux boutons"); setSize (300, 200);
    monBouton1 = new JButton ("Bouton A");
    monBouton2 = new JButton ("Bouton B");
    Container contenu = getContentPane() ;
    contenu.setLayout(new FlowLayout()) ;
    contenu.add(monBouton1); contenu.add(monBouton2);
    EcouteBouton1 ecout1 = new EcouteBouton1();
    EcouteBouton2 ecout2 = new EcouteBouton2();
    monBouton1.addActionListener(ecout1);
    monBouton2.addActionListener(ecout2); }
  private JButton monBouton1, monBouton2 ;}
class EcouteBouton1 implements ActionListener
{ public void actionPerformed (ActionEvent ev)
  { System.out.println ("action sur bouton 1");} }
class EcouteBouton2 implements ActionListener
{ public void actionPerformed (ActionEvent ev)
  { System.out.println ("action sur bouton 2");} }
```

Gestion de plusieurs composants

Avec un seul écouteur pour tous les boutons:

```
class Fen2Boutons extends JFrame
{ public Fen2Boutons ()
  { setTitle ("Avec deux boutons"); setSize (300, 200) ;
    monBouton1 = new JButton ("Bouton A");
    monBouton2 = new JButton ("Bouton B");
    Container contenu = getContentPane() ;
    contenu.setLayout(new FlowLayout()) ;
    contenu.add(monBouton1) ; contenu.add(monBouton2) ;
    EcouteBouton ecout1 = new EcouteBouton(10) ;
    EcouteBouton ecout2 = new EcouteBouton(20) ;
    monBouton1.addActionListener(ecout1);
    monBouton2.addActionListener(ecout2);
  }
private JButton monBouton1, monBouton2 ;
}
class EcouteBouton implements ActionListener
{ public EcouteBouton (int n)  { this.n = n ; }
  public void actionPerformed (ActionEvent ev)
  { System.out.println ("action sur bouton " + n) ;  }
  private int n ;
}
```

Dynamique des composants

A tout moment, on peut:

- créer des composants
- supprimer des composants
- désactiver des composants
- réactiver un composant désactivé

add: ajout suivi de `compo.revalidate()` ou `contenu.validate()` si la fenêtre est déjà affichée.

remove: suppression. `contenu.validate()`

comp.setEnabled(true/false): activation/désactivation.

Plan

Dessins

Permanence graphique

- Rafraîchissement automatique des objets graphiques de la fenêtre (lors de redimensionnements, déplacements, etc)
- Besoin d'effectuer les dessins dans une méthode `paintComponent` qui sera ré-exécutée pour les rafraîchissements
- On ne dessine pas sur la fenêtre directement, mais sur un **panneau**
- Le rafraîchissement peut être forcé par l'invocation de la méthode `repaint` d'un panneau

Exemple de panneau

Le panneau: `panel`

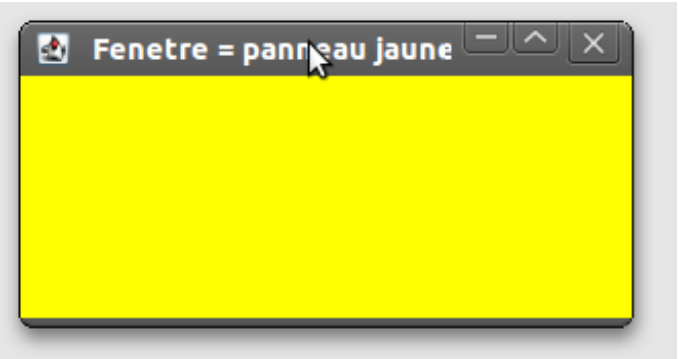
- Composant à la fois contenu (dans la fenêtre) et conteneur (de dessins par ex)

```
class MaFenetre extends JFrame
{ public MaFenetre ()
  // constructeur
  { .....
    panneau = new JPanel ()
    getContentPane().add(panneau) ;
  }
  private JPanel panneau ;
}
```

Exemple

Exemple:

```
import javax.swing.*; import java.awt.*;
class MaFenetre extends JFrame
{ public MaFenetre ()
  { setTitle ("Fenetre = panneau jaune") ;
    setSize (300, 150) ;
    panneau = new JPanel() ;
    panneau.setBackground (Color.yellow) ;
    getContentPane().add(panneau) ;
  }
  private JPanel panneau ;
}
public class Panneau
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
} }
```



Dessin sur un panneau

- Re-définition de la méthode

  ```
  void paintComponent(Graphics g)
  ```

  de l'objet panneau (dérivé de `JPanel`)
- Dans le corps, on doit commencer par invoquer le `paintComponent` du parent (car il y a sur-définition)
- puis les instructions voulues:

  ```
  g.drawLine(15,10,100,50);
  ```

Exemple complet

```
import javax.swing.* ; import java.awt.* ; import java.awt.event.* ;
class MaFenetre extends JFrame implements ActionListener
{ MaFenetre ()
  { setTitle ("Exemple appel repaint") ;
    setSize (300, 200) ;
    Container contenu = getContentPane() ;
    pan = new Panneau() ;
    pan.setBackground (Color.cyan) ;
    contenu.add(pan) ;
    rectangle=new JButton("Rectangle");contenu.add(rectangle,"North");
    rectangle.addActionListener (this) ;
    ovale=new JButton("Ovale"); contenu.add(ovale, "South");
    ovale.addActionListener (this);
}
  public void actionPerformed (ActionEvent ev)
  { if (ev.getSource() == rectangle) pan.setRectangle();
    if (ev.getSource() == ovale) pan.setOvale();
    pan.repaint();//pour forcer la peinture du panneau des maintenant
  }
  private Panneau pan ;
  private JButton rectangle, ovale ;
}
```

Exemple complet

```
class Panneau extends JPanel
{ public void paintComponent(Graphics g)
  { super.paintComponent(g) ;
    if (ovale)
    g.drawOval (80, 20, 120, 60) ;
    if (rectangle) g.drawRect (80, 20, 120, 60) ;
  }
  public void setRectangle() {rectangle = true ; ovale = false ; }
  public void setOvale()
  {rectangle = false ; ovale = true ; }
  private boolean rectangle = false, ovale = false ;
}
public class Repaint
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

Remarques diverses

- Lorsque c'est possible, préférer les modifications des composants eux-mêmes (couleur de fond, texte, ...) plutôt qu'utiliser un panneau.
- Gestion des dimensions:

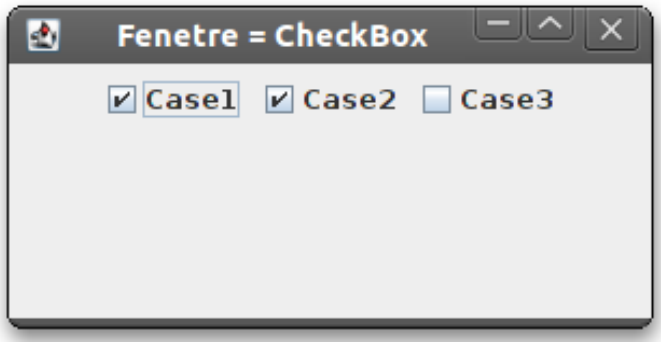
```
Toolkit tk = Toolkit.getDefaultToolkit() ;
Dimension dimEcran = tk.getScreenSize() ;
larg = dimEcran.width ;
haut = dimEcran.height ;
```

-

```
compo.setPreferredSize(new Dimension(100, 100));
```

Plan

Les cases à cocher



- Choix de type Oui/Non
- Classe `JCheckBox`. Constructeur prenant en argument le libellé qui sera affiché à coté de la case:

```
JCheckBox  coche = new JCheckBox("CASE");
getContentPane().add(coche);
```

- État par défaut: non coché. On peut passer l'état booléen en second argument du constructeur

Les cases à cocher

- Chaque action sur une case génère
  - Un événement `Action` dont la chaîne de commande associée est le libellé de la case
  - Un événement `Item`
- Un écouteur de `Action` doit implémenter l'interface `ActionListener`, c'est à dire la méthode `actionPerformed`
- Un écouteur de `Item` doit implémenter l'interface `ItemListener`, c'est à dire la méthode `itemStateChanged`:

```
public void itemStateChanged (ItemEvent ev)
```

Cases à cocher

On peut naturellement aussi récupérer l'état de la case, à tout moment:

- `bool isSelected();`
- `void setSelected(bool);` : dans ce cas, génère un événement `Item` mais pas `Action`

Boutons radio

- Similaire à une case à cocher,
- mais se trouvant dans un groupe où un seul choix est possible en même temps.

```
JRadioButton brouge = new JRadioButton("rouge");
JRadioButton bvert = new JRadioButton("vert", true);
ButtonGroup groupe = new ButtonGroup();
groupe.add(brouge);
groupe.add(bvert);
```

Boutons radio

Chaque action utilisateur sur un bouton radio *r* provoque

- Un événement `Action` et un événement `Item` sur le bouton *r*
- Un événement `Item` sur le bouton qui perd la sélection
- Ces événements doivent être écoutés sur les boutons eux-mêmes, et non le groupe de boutons

L'événement `Item` permet de cerner les changements d'état.

Les étiquettes

Un composant de type `JLabel` permet d'afficher un bloc de text non modifiable par l'utilisateur, mais pouvant évoluer pendant le programme.
Il sert à

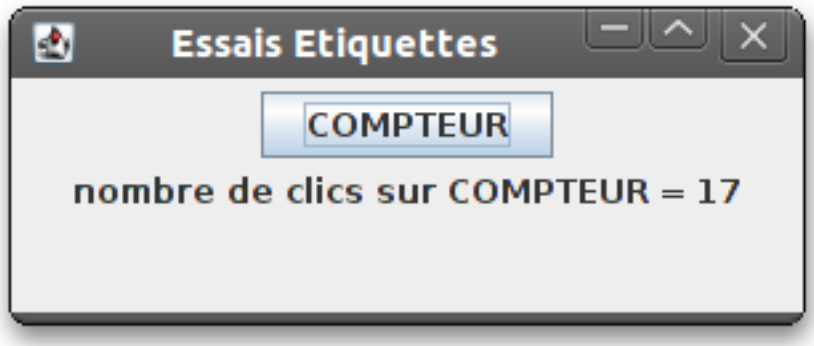
- Affcher une information
- Donner un libellé à un composant qui n'en a pas déjà un

```
JLabel texte = new JLabel("texte initial");
texte.setText("nouveau texte");
```

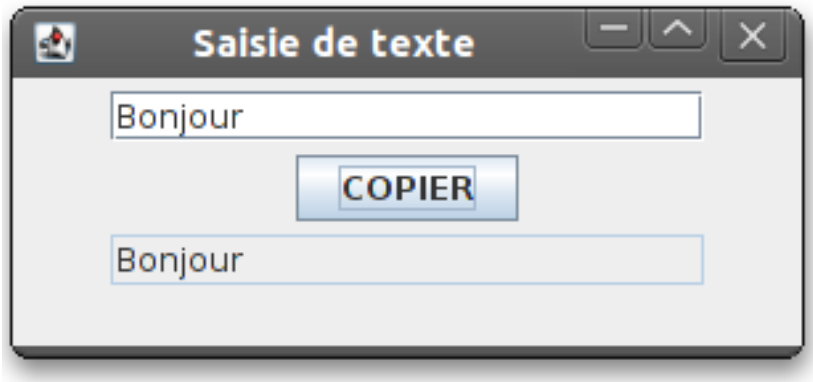
Exercice

Exercice

Écrire un programme avec une fenêtre comportant un bouton et une zone de texte qui affiche une phrase indiquant le nombre de fois que le bouton a été pressé.



Champs de texte



Un champ de texte (classe `JTextField`) est une boîte rectangulaire où l'utilisateur peut entrer ou modifier du texte.

- Son constructeur doit spécifier sa taille en nombre de caractères (taille moyenne, car police proportionnelles).
- On peut (dés)activer la saisie de texte en utilisant `setEditable()`
- On dispose des mêmes méthodes `setText()` et `getText()` qu'un `JLabel`

Champs de texte : événéments

Deux événements peuvent survenir:

- `Action` provoqué par l'appui de la touche Entrée par l'utilisateur
- Événement perte de focus, de la catégorie `Focus`, lorsque l'utilisateur sélectionne un autre composant.

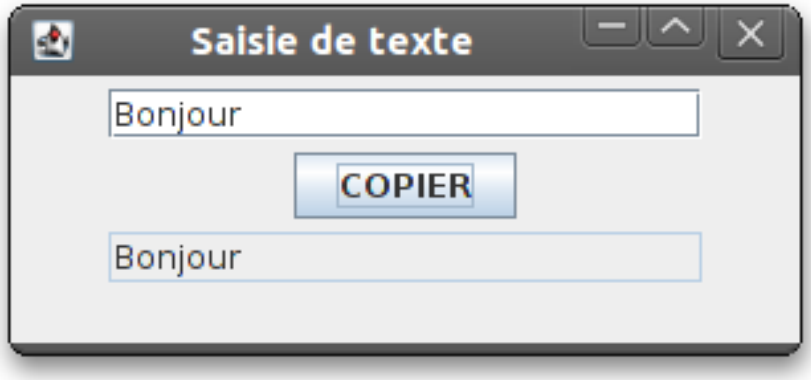
Un `FocusListener` doit implémenter:

- `focusLost(FocusEvent ev)`
- `focusGained(FocusEvent ev)`

Exercice

Exercice

1. Écrire un programme qui saisit une chaîne de caractère dans une boîte de texte et la copie dans une une boîte non éditable lorsque l'on clique sur un bouton **copier**.
2. Modifier le programme pour effacer le texte copié quand le focus est gagné, et copier le texte dès que le focus est perdu ou que la touche entrée est pressée.



Les boîtes combo

- La boîte de liste (`JComboBox`) combinée associe un champ texte (par défaut non éditable – sert à présenter la sélection courante de la liste) et une boîte de liste à sélection simple.
- Tant que le composant n'est pas sélectionné, seul le champ de texte s'affiche. Lorsque l'utilisateur sélectionne le champ de texte, la liste s'affiche. L'utilisateur peut alors sélectionner soit une valeur de la liste, soit un texte de son choix (qui ne sera pas ajouté à la liste).
- La méthode `getSelectedItem` fournit la valeur sélectionnée (fournit un résultat de type `Object` – conversion nécessaire, par ex. en `String`).

Les boîtes combo

Les événements générés par une boîte combo sont de deux types:

- Un événement `Action` provoqué par la sélection d'une valeur dans la liste ou par la validation du champ texte (lorsqu'il est éditable)
- Un événement `Item` à chaque modification de la sélection.
  - Interface `ItemListener`

    ```
    public void itemStateChanged (ItemEvent e)
    ```
  - On obtient en fait toujours deux événements : suppression d'une sélection, nouvelle sélection

Les boîtes combo

Évolution dynamique de la liste d'une boîte combo :

- La méthode `addItem` permet d'ajouter une nouvelle valeur à la fin de la liste
- La méthode `insertItemAt` permet d'insérer une nouvelle valeur à un rang donné de la liste
- La méthode `removeItem` permet de supprimer une valeur existante

Plan

Les applets

En Java, deux sortes de programmes événementiels:

- Les applications graphiques $\rightsquigarrow$ programme indépendant
- Les applets $\rightsquigarrow$ programme téléchargé depuis un serveur distant, par l'intermédiaire d'une page `html`. Éxécuté au sein du navigateur.

Les notions de la programmation graphiques s'étendent aux applets, à quelques exceptions près.

- Une applet est constituée d'une classe dérivée de `JApplet`, qui est un conteneur de plus haut niveau (comme `JFrame`).
- Au lancement d'une applet, on dispose obligatoirement d'une fenêtre graphique, dont les dimensions initiales sont définies par des commandes dans le code HTML appelant.

Les applets

Fichier HTML minimal de lancement d'une applet:

```
<HTML>
    <BODY>
        <APPLET
            CODE = "toto.class"
            WIDTH = 350
            HEIGHT = 100
        >
        </APPLET>
    </BODY>
</HTML>
```

Le fichier de byte-code de l'applet `toto.class` se trouve:

- Dans le répertoire courant si l'applet est exécutée depuis un visualisateur d'applet,
- Dans le répertoire correspondant à l'adresse URL où se trouve la page html, si on l'exécute depuis un navigateur.

Les applets

La méthode `init`

- La méthode init est exécutée automatiquement lors du lancement d'une applet (pas de `main`)
- Usage: la méthode `init` est utilisée pour:
  - construction des composants
  - association des écouteurs
  - association des composants au conteneur `ContentPane`
- on ne redéfinit pas le constructeur (usage)

Exemple d'applet

```
import java.awt.* ; import java.awt.event.* ;
import javax.swing.event.* ; import javax.swing.* ;
public class App extends JApplet implements ActionListener
{ public void init ()
  { pan = new JPanel (); panCom = new JPanel() ;
    Container contenu = getContentPane() ;
    contenu.add(pan) ; contenu.add(panCom, "South") ;
    rouge = new JButton ("rouge") ;
    jaune = new JButton ("jaune") ;
    rouge.addActionListener(this) ;
    jaune.addActionListener(this) ;
    panCom.add(rouge) ; panCom.add(jaune) ;
  }
public void actionPerformed (ActionEvent e)
  { if (e.getSource()==rouge) pan.setBackground(Color.red);
    if (e.getSource()==jaune) pan.setBackground(Color.yellow);
  }
  private JPanel pan, panCom ;
  private JButton rouge, jaune ;
}
```

Différents stades de vie d'une applet

Méthode `init`: exécutée lors du lancement de l'applet

Méthode `destroy` exécutée quand l'applet se termine:

- lorsque l'utilisateur quitte le navigateur
- en fermant la fenêtre correspondante (visualisateur d'applet)

$\rightsquigarrow$ en général inutile de redéfinir `destroy`

Méthode `stop`: exécutée chaque fois que l'applet n'est plus visible, et avant `destroy`

Méthode `start`: exécutée chaque fois que l'applet redevient visible et avant `init`

Exemple

```
import javax.swing.* ;
public class Etats extends JApplet
{ public Etats()
  { System.out.println ("Construction");}
  public void init()
  { System.out.println ("Appel init");}
  public void start()
  { System.out.println ("Appel start");}
  public void stop()
  { System.out.println ("Appel stop");}
  public void destroy()
  { System.out.println ("Appel destroy");}
}
```

Transmission d'information à une applet

Par le biais de commandes dans le fichier HTML.

```
<PARAM NAME="mois" VALUE="avril">
```

- Noms et valeurs sont toujours des chaînes de caractères (la casse est sans effet).
- Ces commandes doivent figurer à l'intérieur des balises `<APPLET>` et `</APPLET>`,
- Les valeurs de ces informations sont récupérées dans la méthode init à l'aide la méthode `getParameter` (de la classe `JApplet`).

Exemple

```
<HTML>
   <BODY>
      <APPLET CODE="Infos.class" WIDTH=250 HEIGHT=100>
         <PARAM NAME="mois" VALUE="février">
         <PARAM NAME="annee" VALUE="2019">
      </APPLET>
   </BODY>
</HTML>
```

```
import javax.swing.* ;
public class Infos extends JApplet
{ public void init ()
  { String nomMois = getParameter ("mois") ;
    String nomAnnee = getParameter ("annee") ;
    int annee, anneeSuiv ;
    System.out.println ("Mois = " + nomMois) ;
    System.out.println ("Annee = " + nomAnnee) ;
    annee = Integer.parseInt(nomAnnee) ;
    anneeSuiv = annee+1 ;
    System.out.println ("Annee suivante = " + anneeSuiv)
  }
}
```

Restrictions imposées aux applets

Au départ, restrictions pour assurer la sécurité de la machine d'exécution. En particulier, la machine virtuelle interdisait aux applets :

- d'accéder aux fichiers locaux
- De lancer un programme exécutable local
- D'obtenir des informations relatives au système local

$\rightsquigarrow$ Toute tentative de ce type provoquait une exception `SecurityException`.

Avec la généralisation des applets, ces restrictions sont apparues comme trop sévères. Désormais:

- notion de gestionnaire de sécurité permet à un environnement de définir les opérations qu'il autorise pour les applets.
- notion d'applet certifiée pour accorder des permissions supplémentaires à une applet pour laquelle on dispose d'une garantie d'origine déterminée.

Plan

Programmation Générique

Programme générique:

Lorsque qu'un même code peut être utilisé avec des objets de types variables

Exemple : algorithmes de tri d'objets (entiers, chaînes de caractères, ...)

Deux types de généricité:

- lorsque au sein d'une même instance, plusieurs types différents peuvent être manipulés sous une même dénomination $\rightsquigarrow$ héritage et polymorphisme
- lorsque le type est inconnu au moment de l'écriture, mais est fixé de façon unique lors de l'instanciation de la classe ou l'appel de la méthode. $\rightsquigarrow$ **paramètres de type**

On parle alors de classes ou de méthodes génériques.

Plan

Notion de classe générique

Exemple: Une classe représentant des couples d'objets (de même type)

```
class Couple<T>
{ private T x, y ;  // les deux elements du couple
  public Couple (T premier, T second)
  { x = premier ; y = second ; }
  public void affiche () // x et y convertis
   ↪  automatiquement par toString
  { System.out.println("premiere valeur:"+ x +
                       "-deuxieme valeur:"+y);}
  T getPremier () {return x;}
  T getDeuxieme () {return y;}
}
```

Utilisation de la classe générique

- Lors de la déclaration d'un objet de type `Couple`, on doit spécifier le type effectif correspondant à `T`:

```
Couple<Integer> ci;
Couple<Point> cp;
```

  Seule contrainte: `T` doit être une classe: `Couple<int>` serait rejeté.

- On spécifie aussi le type lors de l'appel au constructeur:

```
ci = new Couple<Integer>(i1, i2);
```

- Mais pas pour l'appel `ci.affiche()`

Plusieurs paramètres de type

```
class Couple<T, U>
{ private T x ;
  private U y ;
  public Couple (T premier, U second)
  { x = premier ; y = second ; }
  public T getPremier () { return x ; }
  public U getSecond () { return y ; }
  public void affiche ()
  { System.out.println ("premiere valeur: "+x+
                        " - deuxieme valeur: "+y);
public class CoupleM
{ public static void main (String args[])
  { Integer oi1 = 3 ; Double od1 = 2.5 ;
    Couple<Integer,Double> ch1
          = new Couple<Integer,Double>(oi1,od1);
    ch1.affiche() ;
    Integer oi2 = 4 ;
    Couple<Integer,Integer> ch2
          = new Couple<Integer,Integer>(oi1,oi2);
    ch2.affiche() ;
  }
}
```

Plan

Compilation de classes génériques

Principe: *Remplacer les types génériques par le super-type* `Object` *(principe de l'effacement)*

- Dans une classe générique: substitution `<T>` $\rightarrow$ `Object`.
- Dans l'utilisation d'une classe générique:
  `ci.getPremier()` retourne un `Object` et non un `Integer`
  $\rightsquigarrow$ conversion du type `Object` vers le type `Integer`

```
(Integer) ci.getPremier();
```

Limitation des classes génériques

Le principe de l'effacement impose de fortes limitations:

1. On ne peut pas instancier un objet de type paramétré:

```
class Ex<T> {
  T x;
  void f () {
    x = new T(); // Interdit
    T[] t =  new T[100]; // Interdit
  }
}
```

2. On ne peut pas instancier de tableau d'éléments d'un type générique (même quand le type est spécialisé).

```
Couple <Double> [] tcd = new Couple <Double> [5];
```

3. Seul le type brut est connu lors de l'exécution:

```
Couple<String> cs = new Couple<String>(...);
Couple<Double> cd = new Couple<Double>(...);
cs instanceOf Couple // true
cd instanceOf Couple // true
```

Limitation des classes génériques

4. Une classe générique ne peut être dérivée de `Throwable` $\rightsquigarrow$ ni de `Exception` ni de `Error`
5. Un champ statique est commun à toutes les instances d'une classe générique, quelque soit le paramètre de type:

   - Un champ statique ne peut être d'un type paramétré

```
class<T> Couple {
  static T x; // Erreur de compilation
  ...
}
```

Plan

Méthodes génériques

Comme pour les classes, les méthodes peuvent être paramétrées par un/des types.

```
public class MethGen1
{ static <T> T hasard (T [] valeurs)
  { int n = valeurs.length ;
    if (n == 0) return null ;
    int i = (int) (n * Math.random() ) ;
    return valeurs[i] ;
  }
  public static void main(String args[])
  { Integer[] tabi = { 1, 5, 8, 4, 9} ;
    System.out.println ("hasard sur tabi = "
                        + hasard (tabi) ) ;
    String [] tabs = { "bonjour", "salut", "hello"} ;
    System.out.println ("hasard sur tabs = "
                        + hasard (tabs) ) ;
  }
}
```

Méthodes génériques

Une méthode générique

- peut être une méthode d'une classe de type fixé:

```
class A{
  public <T> T f(T t){...}
  ...
}
```

- ou une méthode d'une classe générique

```
class<T> A{
  public <U> U f(U u){...}
  ...
}
```

Méthode générique à deux arguments

Une méthode avec **un** paramètre de type sur deux arguments

```
public class MethGen2
{ static <T> T hasard (T e1, T e2)
  { double x = Math.random() ;
    if (x <0.5) return e1 ;
    else return e2 ;
  }
}
```

peut être utilisée avec

- le même type sur les deux arguments

  ```
  Integer n1=1; Integer n2=2; Double x=2.5;
  hasard(n1,n2);
  ```

- ou **deux** types différents

  ```
  hasard(n1, x);
  ```

Forçage du type paramétré

Pour imposer au compilateur une condition sur le type paramétré de la méthode générique:

```
MaClasse.<typeForce> methode (x);
```

Par exemple

```
Integer n = 2; Double x = 2.5;
MethGen2.<Double> hasard (n,x);//Erreur de compilation
MethGen2.<Number> hasard (n,x);//OK
```

Plan

Limitations sur les paramètres de type

Lors de la définition d'une classe générique, on peut imposer au paramètre de type certaines conditions

- d'être dérivé d'une certaine classe

  ```
  class Couple <T extends Number> {...}
  ```

  $\rightsquigarrow$ signale au compilateur d'effectuer l'effacement avec le type `Number` et non `Object`.

  $\rightsquigarrow$ `d=hasard(...);` sera traduit par

  ```
  d = (Number) hasard(...);
  ```

  Mais on ne pourra toujours pas instancier d'objet de type variable `T`!

Limitations sur les paramètres de types

- d'implémenter une certaine interface:

```
class Couple <T extends Comparable> {...}
```

  $\rightsquigarrow$ Attention: toujours le mot clé **extends**

- d'implémenter plusieurs interfaces:

```
class Couple <T extends Comparable & Cloneable> {...
```

  le type utilisé pour l'effacement est le premier fourni (ici `Comparable`)

- de dériver d'une classe et implémenter une ou plusieurs interfaces

```
class Couple < T extends Number &
                         Comparable &
                         Cloneable > {...}
```

Plan

Héritage et programmation générique

Dérivation d'une classe générique:

- la classe dérivée conserve les paramètres de type de la classe parent:

```
class D<T> extends C<T> {...}
```

- la classe dérivée conserve les paramètres de type de la classe parent en en ajoutant d'autres:

```
class D<T,U> extends C<T> {...}
```

- la classe introduit des limitations sur les paramètres de type de la classe parent :

```
class D<T extends Number> extends C<T> {...}
```

Mais on ne peut pas faire

- `class D extends C<T>{...}`
- ni

```
class D<T> extends C<T extends Number>{...}
```

Héritage et programmation générique

Si `T2` dérive de `T1`, alors `C<T2>` ne dérive pas de `C<T1>`

```
class Couple1<T>
{ private T x, y ;
  public Couple1 (T premier, T second)
  { x = premier ; y = second ; }
  public T getPremier () { return x ; }
  public void setPremier (T premier)
  { x = premier ; }
  public void affiche ()
  { System.out.println ("premiere valeur : "+x+
                        " - deuxieme valeur : "+y);}
}
```

Exemple posant problème:

```
Object o1,o2;
Integer i1,i2;
Couple1 <Object> co = new Couple1 <Object>(o1,o2);
Couple1 <Integer> ci = new Couple1 <Integer>(i1,i2);
co=ci; // Interdit
co.setPremier(o1);
```

Les jokers

Problème : Si `T2` dérive de `T1`, alors `C<T2>` ne dérive pas de `C<T1>`.

- En lecture seule, on peut effectivement considérer tout `C<T2>` comme un `C<T1>`
- Le problème a lieu lorsque l'on modifie les valeurs de l'objet

Les Jokers: `C<?>`

permet de profiter de l'héritage mais *en lecture* seulement: tout appel à une fonction prenant en argument un objet du type paramétré `T` est interdit.

```
Couple<Integer> ci;
Couple<Double> cd;
cd = ci; // Interdit: erreur de compilation
Couple<?> cq = ci; // autorise
cq.setPremier(...); // Interdit:erreur de compil.
```

Plan

Collections et Algorithmes

Extension et harmonisation des bibliothèques de classes utilitaires (`java.util`).

Les collections

- **dénominateur commun de plusieurs structures de données :** listes chaînées, vecteurs dynamiques, ensembles, queues, ...
- **fonctions communes :** génériques, itérateurs
- **algorithmes *généraux*** agissant sur ces structures: recherche, max, min, tri,...

Plan

Concepts Généraux

Les collections implémentent l'interface `Collection<E>`.

Généricité: les éléments de la collection sont de type `E` ou dérivé

Ordre des éléments:

- Certaines collections n'ont pas d'ordre: ensembles
- D'autres ont un ordre de stockage naturel (et arbitraire): vecteur, liste chaînée, queue
- Parfois, besoin d'ordonner les éléments suivant un ordre spécifique (calcul de max, tri, ...):

Comparer des éléments

L'interface `Comparable`

Impose de fournir la méthode `compareTo`

```
public int compareTo (E obj);
```

Retourne un entier :

- **négatif** si l'objet courant est **inférieur** à obj
- **nul** si les objets sont **égaux**
- **positif** si l'objet courant est **supérieur** à obj

Comparer des éléments

L'objet comparateur

Lorsque

- on veut comparer des objets dont la classe n'implémente pas l'interface `Comparable`.
- on veut disposer de plusieurs relations d'ordre sur les éléments d'une même classe

C'est un objet d'une classe implémentant `Comparator<E>` et fournissant la méthode:

```
public int compare (E obj1, E obj2);
```

qui retourne un entier

- **négatif** si `obj1` est **inférieur** à `obj2`
- **nul** si `obj1` est **égal** à `obj2`
- **positif** si `obj1` est **supérieur** à `obj2`

Test d'égalité

Essentiel de pouvoir tester si deux éléments sont égaux

- Structure d'ensemble (un même élément n'apparaît qu'une fois)
- méthode `remove` pour enlever tous les éléments ayant une valeur donnée

La méthode `equals` de la classe `Object`

- Pour les types `String` et `File`: comparaison du contenu
- Pour les autres objets: comparaison de la référence

$\rightsquigarrow$ À redéfinir si besoin

Remarque: s'assurer de la compatibilité avec la méthode `compareTo`

Les itérateurs

- Objets permettant le parcours itératif d'une collection.
- Peuvent être mono- ou bi-directionnels

L'interface `Iterator`

Chaque collection founit une méthode `iterator`

- retourne un objet d'une classe implémentant l'interface `Iterator<E>`
- indique une position courante dans la collection
- `next()` retourne l'élément courant et *avance* dans la collection
- `hasNext()` indique si on est en fin de collection

Canvas d'utilisation des itérateurs

Appliquer une fonction sur chaque élément de la collection:

```
Iterator<E> iter = c.iterator () ;
while ( iter.hasNext() )
{ E o = iter.next () ;
  // utilisation de o
}
// Ou encore plus simplement:
for (E o : c)
{ utilisation de o
}
```

Méthode `remove` d'un itérateur:

```
Iterator<E> iter = c.iterator() ;
while (c.iter.hasNext())
{ E o = iter.next() ;
  if (condition) iter.remove() ;
}
```

Attention: seulement après un `next()` pour que l'itérateur pointe sur l'objet suivant.

Les itérateurs bi-directionnels

Certaines collections (listes chaînées, vecteurs dynamiques) permettent le parcours dans les deux sens: méthode `listIterator` retournant un objet d'une classe implémentant `ListIterator<E>`:

- `next, hasNext` mais aussi `previous, hasPrevious`
- `set, add`

```
ListIterator <E> iter ;
/*position courante en fin de liste*/
iter = l.listIterator (l.size());
while (iter.hasPrevious())
{ E o = iter.previous () ;
  ...//utilisation de l'objet courant o
}
it = c.listIterator() ;
it.next();/* premier element = element courant */
it.next();/* deuxieme element = element courant */
it.add (elem);//ajoute elem a la position courante, cad
              //entre le premier et le deuxieme element
```

Opérations commune des collections

Construction :

```
C<E> c = new C<E>();//Constructeur vide
C<E> c2 = new C<E>(c);//Constructeur  de copie
```

Iterateur :

`iterator`, `listIterator` (parfois)

Autres:

`add`, `addAll`, `removeAll`, `size`, `contains`, `toString`...

Structuration des collections

Architecture d'interfaces :

```
Collection
  List//implementee par LinkedList,ArrayList et Vector
  Set //implementee par HashSet
    SortedSet //implementee par TreeSet
      NavigableSet // implementee par TreeSet
  Queue // implementee par LinkedList, PriorityQueue
    Deque // implementee par ArrayDeque, LinkedList
```

Plan

Les listes chaînées: `LinkedList`

Chaque élément est lié à son successeur et son prédécesseur.

- itérateur bi-directionnel `ListIterator`
- insertion, suppression en $\mathcal{O}(1)$
- recherche et ième élément en $\mathcal{O}(n)$

Méthodes spécifiques:

- `getFirst, getLast`
- `add`: ajout d'un élément à la position courante ($\neq$ `add` de `Collection`)
- `removeFirst, removeLast`, et
  `public bool remove(element)`

Exercice

1. Écrire la méthode `affiche` qui affiche chaque élément de la liste espacés d'un blanc.
2. Que fait le code suivant ?

```
import java.util.* ;
public class Liste1
{ public static void main (String args[])
  { LinkedList<String> l = new LinkedList<String>() ;
    System.out.print ("Liste en A : ") ; affiche (l) ;
    l.add ("a") ; l.add ("b") ;
    System.out.print ("Liste en B : ") ; affiche (l) ;
    ListIterator<String> it = l.listIterator() ;
    it.next() ;
    it.add ("c") ; it.add ("b") ;
    System.out.print ("Liste en C : ") ; affiche (l) ;
    it = l.listIterator() ;
    it.next() ;
    it.add ("b") ; it.add ("d") ;
    // et on ajoute deux elements
    System.out.print ("Liste en D : ") ; affiche (l) ;
    ...
```

Exercice

```
    it = l.listIterator (l.size()) ;
    while (it.hasPrevious())
    { String ch = it.previous() ;
      if (ch.equals ("b"))
      { it.remove() ; // et on le supprime
        break ;
      }
    }
    System.out.print ("Liste en E : ") ; affiche (l) ;
    it = l.listIterator() ;
    it.next() ; it.next() ;
    it.set ("x") ;
    System.out.print ("Liste en F : ") ; affiche (l) ;
}
```

Plan

Les vecteurs dynamiques: `ArrayList`

- Implémentent l'interface `List`
- Accès direct aux éléments: lecture/écriture en $\mathcal{O}(1)$:
  `get(i), set(i,elem)`
- Plus souple que les tableaux classiques: taille variable
- Additions/suppressions en $\mathcal{O}(n)$:
  `add(i,elem), remove(i)`
- Itération, comme partout:

```
public<E> static void  affiche (ArrayList<E> (v)
{ for (E e : v)
        System.out.print(e + " ");
  System.out.println();
}
```

Les vecteurs dynamiques

Gestion de l'emplacement mémoire:

- Un `ArrayList` dispose d'une capacité (espace mémoire réservé, utilisable)
- Les ajouts d'éléments se font en utilisant cet espace
- Lorsqu'il n'y a plus d'espace libre, allocation d'un espace deux fois plus grand, et copie

Plan

Les Ensembles

Deux représentations:

Les HashSet: technique de hachage de l'élément pour obtenir un *résumé* unique. Test d'appartenance en $\mathcal{O}(1)$

Les TreeSet: utilise une relation d'ordre sur les éléments pour construire un arbre binaire de recherche. Test d'appartenance en $\mathcal{O}(\log n)$

Rappel : Les types `String` et `File` possèdent déjà une relation d'ordre (`compareTo`)

Exemple:

```
HashSet<E> e1 = new HashSet<E> ();
TreeSet<E> e2 = new TreeSet<E> (e1);
E elem;
e2.add(e);
```

Les ensembles

Méthode a.remove(x): en $\mathcal{O}(1)$ pour les `Hashset` ou $\mathcal{O}(\log n)$ pour les `TreeSet`.

Méthode a.addAll(c): ajoute tous les éléments de la collection c (union)

Méthode a.removeAll(c): supprime tous les éléments présents dans la collection c (complémentaire de c dans a)

Méthode a.retainAll(c): supprime tous les éléments autres que ceux de c (intersection de a et c)

Exemple

```
import java.util.* ;
public class Ens2
{ public static void main (String args[])
  { String phrase = "je me figure ce zouave qui joue" ;
    String voy = "aeiouy" ;
    HashSet <String> lettres = new HashSet <String>() ;
    for (int i=0 ; i<phrase.length() ; i++)
    lettres.add (phrase.substring(i, i+1)) ;
    System.out.println ("lettres presentes : " + lettres) ;
    HashSet <String> voyelles = new HashSet<String>() ;
    for (int i=0 ; i<voy.length() ; i++)
    voyelles.add (voy.substring (i, i+1)) ;
    lettres.removeAll (voyelles) ;
    System.out.println ("Consonnes presentes: " + lettres) ;
  }
}
```

Les Hashset

- En général il faut redéfinir les méthodes `equals` et `hashCode`
- Pour les `String`, `File` et les types enveloppes, c'est déjà fait

Principe des tables de hachage

- On fixe un entier N, le nombre de seaux
- La collection est un tableau de N listes chaînées, appelées les seaux
- Chaque élément est stocké dans le seau $j = \mathit{hashCode}(e) \mod N$
- Pour la recherche: on calcule $j = \mathit{hashCode}(e) \mod N$, et on n'appelle la méthode `equals` que sur les éléments d'un même seau.
- Le ratio entre le nombre d'éléments de la collection et N est le facteur de charge (généralement autour de 0.75).
- Java augmente N dès que le facteur de charge dépasse 0.75.

Exemple

```
import java.util.* ;
public class EnsPt1
{ public static void main (String args[])
{ Point p1 = new Point (1, 3), p2 = new Point (2, 2) ;
  Point p3 = new Point (4, 5), p4 = new Point (1, 8) ;
  Point p[] = {p1, p2, p1, p3, p4, p3} ;
  HashSet<Point> ens = new HashSet<Point> () ;
  for (Point px : p)
  { System.out.print ("le point ") ;
    px.affiche() ;
    boolean ajoute = ens.add (px) ;
    if (ajoute) System.out.println (" a ete ajoute") ;
    else System.out.println ("est deja present") ;
    System.out.print ("ensemble = " ) ; affiche(ens) ;
  }
}
public static void affiche (HashSet<Point> ens)
{ Iterator<Point> iter = ens.iterator();
  while (iter.hasNext())
  { Point p = iter.next(); p.affiche(); }
  System.out.println () ;
 } }
```

Exemple

```
class Point
{ Point (int x, int y) { this.x = x ; this.y = y ; }
  public int hashCode () { return x+y ; }
  public boolean equals (Object pp)
  { Point p = (Point) pp ;
        return ((this.x == p.x) && (this.y == p.y)) ;
  }
  public void affiche ()
  { System.out.print ("["+x+" "+y+"]");}
  private int x, y ;
}
```

Les TreeSet

Collection implémentée par un arbre binaire de recherche: besoin de compararer les élément. La classe des éléments doit :

- Implémenter `Comparable`
- Fournir une méthode `compareTo`

```
class Point implements Comparable
{ Point (int x, int y) { this.x = x ; this.y = y ; }
  public int compareTo (Object pp)
  { Point p = (Point) pp ;
     if (this.x < p.x) return -1 ;
     else if (this.x > p.x) return 1 ;
     else if (this.y < p.y) return -1 ;
     else if (this.y > p.y) return 1 ;
     else return 0 ;
}
```

Plan

Les Queues

Structure FIFO (First In First Out).

L'interface `Queue`

- Ajout d'un élément: `offer`. Comme `add`, mais sans exception quand la queue est pleine (retourne **`false`**)
- Prélèvement du premier élément: `poll` (enlève cet élément de la queue)
- Lecture du premier élément: `peek` (conserve cet élément dans la queue)

Deux collections implémentent cette interface:

LinkedList : déjà vue

PriorityQueue : les élément doivent posséder une méthode `compareTo`. Ils sortent de la queue par ordre croissant selon cette relation.

Les queues à double entrée: `Deque`

L'interface `Deque`

Permet

- d'ajouter
- d'examiner
- de supprimer

un élément aux deux extrémités de la file d'attente.

	Exception	Valeur Spéciale
Ajout	`addFirst, addLast`	`offerFirst, offerLast`
Examen	`getFirst, getLast`	`peekFirst, peekLast`
Suppression	`removeFirst, removeLast`	`pollFirst, pollLast`

Implémenté par la classe `ArrayDeque`.

Plan

Algorithmes

Méthodes statiques de la classe `Collections`:

- `max, min`
- `sort` : tris en $\mathcal{O}(n \log n)$
- `shuffle` : mélange aléatoire

```
Collections.sort(l); // si l possede un compareTo
Collections.sort(l, new Comparator(){...} // sinon
```

Plan

Introspection (11)

Introspection

Introspection:

Capacité d'apprendre des informations sur un objet instanciant une classe a priori inconnue

Exemple : `p instanceof Point` vaut `True` ssi `p` est une instance de la classe `Point` ou d'une classe dérivée.

Plus généralement, on souhaite pouvoir:

- Accéder au nom d'une classe inconnue
- À la liste de ses constructeurs, attributs, méthodes
- À leurs droits d'accès
- Créer des instances de la classe
- Appeler et lire les méthodes et attributs
- Etc.

La super-classe `Class<T>`

Toute classe `C` possède un attribut **`static class`** et une méthode `getClass()` renvoyant une instance de la super-classe générique `Class<C>` représentant `C`:

```
Point p = new Point();
Class<?> c = p.getClass(); // utilisation générique
Class<Point> c2 = p.getClass(); // on peut
↪  spécifier complètement le type de Class
Class<? extends Point> c3 = Point.class; // ou
↪  partiellement
```

Utilisations: tests et affichage

Permet de tester “exactement” la classe d’un objet ou d’afficher son nom:

```
class Point {}
class Point2 extends Point {}
Point p = new Point();
Point p2 = new Point2();
System.out.println(p.getClass() == p2.getClass());
 ↪  // false
System.out.println(p.getClass() == Point.class); //
 ↪  true
System.out.println(p.getClass()); // Class Point
System.out.println(p.getClass().getName()); //
 ↪  Point
```

Création d'instances

Le code suivant permet de créer une instance d'une classe a priori inconnue

```
Point p = new Point();
class<?> c = p.getClass();
try
{
  Object p2 =
   ↪  c.getDeclaredConstructor.newInstance();
  System.out.println(p2.getClass() == Point.class)
   ↪  // true;
}
catch(....) {...}
```

Problèmes:

- Le constructeur sans arguments peut être inexistant ou inaccessible
- On peut souhaiter appeler un constructeur avec arguments

$\rightsquigarrow$ Il faut être en mesure de lister tous les constructeurs disponibles

Accès aux constructeurs déclarés

Un objet `Class` est doté d'une méthode `Constructor<?>[] getDeclaredConstructors()` qui retourne la liste des constructeurs déclarés, ou `getConstructors()` pour tous les constructeurs disponibles

Un objet `Constructor<T>` dispose entre autres des méthodes `Class<?>[] getParameterTypes()` et `T newInstance(Object... initargs)` où:

- `T` est le type instancié de la classe générique `Constructor<T>`
- `initargs` est un tableau d'`Object`s contenant les valeurs des arguments éventuels

Exemple

```
class Point
{
        public Point() { }

        public Point(int a) {
                this.a = a; this.b = a; }

        public Point(int a, int b) {
                this.a = a; this.b = b; }

        public int a = 0;
        public int b = 0;
}
```

Exemple, suite (en trichant un peu)

```
public static void main(String args[])
{
Point p = new Point();
Class<?> c = p.getClass();

Constructor<?>[] constrs = c.getDeclaredConstructors();
for (int i = 0; i < constrs.length; i++)
{
        System.out.println(constrs[i]);
        Class<?>[] params = constrs[i].getParameterTypes();
        for (int j = 0; j < params.length; j++)
        {
                System.out.print(params[j] + " ");
        }
        System.out.println("\n=======");
}

Constructor<?> constr = constrs[2];
Object[] cargs = new Object[2];
cargs[0] = Integer.valueOf(1);
cargs[1] = Integer.valueOf(2);
Point p2 = (Point)constr.newInstance(cargs);
System.out.println(p2.a + " ; " + p2.b);
}
```

Accès aux méthodes

L'accès aux méthodes déclarées d'une classe se fait comme pour les constructeurs: utilisation de
`Method[] getMethods()` ou
`Method[] getDeclaredMethods()`

`Method` possède des méthodes:

- `getParameterTypes` et ses variantes
- `Class<?> getReturnType()`
- `Object invoke(Object obj, Object... args)` qui permet d'appeler la méthode depuis un object `obj`
- Etc.

Accès aux attributs

On peut accéder aux attributs d'une classe de façon similaire, grâce à `Field[] getFields()` et ses variantes. Un objet `Field` dispose des méthodes:

- `Class<?> getType()`
- `Object get(Object obj)` retourne la valeur du champ de l'objet `obj`, en tant qu'`Object` générique
- `double getDouble(Object obj)` variante spécialisée, retournant un `double`
- `void set(Object obj, Object value)`
- `void setDouble(Object obj, double d)`
- Etc.

Violation d'encapsulation

La méthode `getFields()` retourne les champs *accessibles* ; la méthode `getDeclaredFields()` retourne les champs déclarés, y compris les champs **`private`**.

On ne peut normalement pas accéder à un champ **`private`**, mais on peut changer son accessibilité pour les méthodes précédentes (ça ne le rend cependant pas "**`public`**") !

Exemple

```
class ConstZero
{
        int getZero() {return a;}
        private int a = 0;
}
public static void main(String args[])
{
        ConstZero zero = new ConstZero();
        Class<?> cz = zero.getClass();
        Field[] f = cz.getDeclaredFields();
        System.out.println(f[0]);
        System.out.println(zero.getZero());
        f[0].setAccessible(true);
        int val = 0;
        try
        {
                f[0].setInt(zero, 1);
                val = f[0].getInt(zero);
        }
        catch(IllegalAccessException e)
        {
                System.out.println("I'm afraid I can't do
                ↪  that");
        }
        System.out.println(val + " / " + zero.getZero());
}
```