Héritage

Obtenir réutilisation et sous-typage avec élégance

- L'**héritage** est un mécanisme pour définir une nouvelle classe [1] B à partir d'une classe existante A : B récupère les caractéristiques [2] de A et en ajoute de nouvelles.
- Ce mécanisme permet la réutilisation de code.
- L'héritage implique le sous-typage : les instances de la nouvelle classe **sont** [3] ainsi des instances (indirectes) de la classe héritée avec quelque chose en plus.

1. Ou bien une nouvelle interface à partir d'une interface existante.
2. concrètement : les membres
3. Par opposition au mécanisme de composition : dans ce cas, on remplacerait « sont » par « contiennent ».

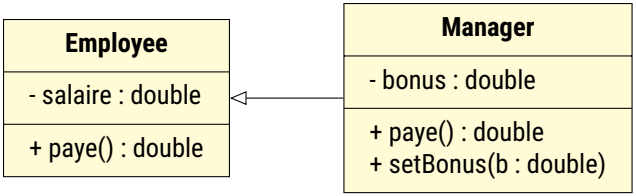
Héritage – syntaxe…

… par l'exemple


```java
class Employee {
    private double salaire;
    public Employee(double s) { salaire = s; }
    public double paye() { return salaire; }
}

class Manager extends Employee { // <------ c'est là que ça se passe !
    private double bonus;

    public Manager(double s, double b) {
        super(s); // <------ appel du constructeur parent
        bonus = b;
    }

    public void setBonus(double b) { bonus = b; }

    @Override
    public double paye() { // <------ redéfinition !
        return super.paye() + bonus;
    }
}
```

Héritage

Notation UML



Notez la flêche : trait plein et tête en triangle côté superclasse, pour signifier ”hérite de”.

La méthode redéfinie (`paye`) apparaît à nouveau dans la sous-classe.

Héritage

Folklore et « piliers » de la POO, avertissement

- D'après le folklore : **« piliers » de la POO** = encapsulation, polymorphisme et héritage.
- **Mais ce dernier principe ne définit pas du tout la POO !** [1]
- Héritage : mécanisme de réutilisation de code très pratique, mais non fondamental.
- ∃ LOO sans héritage : les premières versions de Smalltalk ; le langage Go. La POO moderne incite aussi à préférer la composition à l'héritage. [2]

Avertissement : l'héritage, mal utilisé, est souvent source de rigidité ou de fragilité [3]. Il faudra, suivant les cas, lui préférer l'implémentation d'interface ou la composition.

Faiblesses de l'héritage et alternatives possibles seront discutées à la fin de ce chapitre.

---

1. Rappel : POO = programmation faisant communiquer des objets.
2. Ce qui n'empêche que l'héritage soit évidemment au programme de ce cours.
3. EJ3 19 : « *Design and document for inheritance or else prohibit it* »

Extension vs. implémentation

En POO, en théorie :

- implémentation d'interface $\leftrightarrow$ spécification d'un supertype
- héritage/extension $\leftrightarrow$ récupération des membres hérités (= facilité syntaxique)

En Java, en pratique, distinction moins claire, car :

- implémentation et héritage impliquent tous deux le sous-typage
- quand on « implémente » on hérite des implémentations par défaut (`default`).

Les différences qui subsistent :

- extension de classe : la seule façon d'hériter de la description concrète d'un objet (attributs hérités + usage du constructeur `super()`);
- implémentation d'interface : seule façon d'avoir plusieurs supertypes directs.

Héritage

De classe ou d'interface ?

En Java, la notion d'héritage concerne à la fois les classes et les interfaces.

L'héritage n'a pas la même structure dans les 2 cas :

- Une classe peut hériter directement d'une (et seulement une) classe (« **héritage simple** »).
  Par ailleurs, toutes les classes héritent de la classe `Object`.
- Une interface peut hériter directement d'une ou de plusieurs interfaces.
  Elle peut aussi n'hériter d'aucune interface (pas d'ancêtre commun).

Remarque : avec l'héritage, on reste dans une même catégorie, classe ou interface, par opposition à la relation d'implémentation.

Héritage de classe et d'interface, implémentation et sous-typage

Comparatif des 4 relations

Pour résumer, il est possible de comparer 4 relations différentes :

	héritage de classe	héritage d'interface	implémentation d'interface	sous-typage de types référence
mot-clé	`extends`	`extends`	`implements`	(tout ça)
parent	classe	interface	interface	type
enfant	classe	interface	classe	type
nb. parents	1 [1]	≥ 0	≥ 0	≥ 1 [2]
graphe	arbre	DAG	DAG, hauteur 1	DAG
racine(s)	classe `Object`	multiples	multiples	type `Object`

1. 0 pour classe `Object`
2. 0 pour type `Object`

Héritage : vocabulaire

Héritage direct ou extension

- Une classe [1] A peut « **étendre** »/« hériter directement de »/« être dérivée de/être une sous-classe directe d'une autre classe B. Nous noterons $A \prec B$.
  (Par opposition, B est appelée **superclasse directe** ou classe mère de A : $B \succ A$.)
- Alors, tout se passe comme si les membres visibles de B étaient aussi définis dans A. On dit qu'ils sont **hérités** Conséquences :
  1. toute instance de A peut être utilisée comme [2] instance de B ;
  2. donc une expression de type A peut être substituée à une expression de type B.

  Le système de types en tient compte : $A \prec B \implies A <: B$ (A sous-type de B).
- Dans le code de A, le mot-clé `super` est synonyme de B. Il sert à accéder aux membres de B, même masqués par une définition dans A (ex : `super.f();`).

1. Pour l'héritage d'interfaces : remplacer partout « classe » par interface.
2. Parce qu'on peut demander à l'instance de A les mêmes opérations qu'à une instance de B (critère bien plus faible que le principe de substitution de Liskov !).

Héritage : vocabulaire

Héritage généralisé

Héritage (sous-entendu : « généralisé ») :

- Une classe A **hérite de**/est une **sous-classe** d'une autre classe B s'il existe une séquence d'extensions de la forme : $A \prec A1 \prec \cdots \prec An \prec B$ [1].
  Notation : $A \sqsubseteq B$ (remarques : $A \prec B \implies A \sqsubseteq B$, de plus $A \sqsubseteq A$).
- Par opposition, B est appelée superclasse (ou ancêtre) de A. On notera $B \sqsupseteq A$.
- Héritage implique [2] sous-typage : $A \sqsubseteq B \implies A <: B$.
- Pourtant, une classe n'hérite pas de tous les membres visibles de tous ses ancêtres, car certains ont pu être **masqués** par un ancêtre plus proche. [3]
- ~~`super.super`~~ n'existe pas ! Une classe est isolée de ses ancêtres indirects.

1. L'héritage généralisé est la fermeture transitive de la relation d'héritage direct.
2. Héritage $\Rightarrow_{\text{déf.}}$ chaîne d'héritages directs $\Rightarrow_{\prec \subset <:}$ chaîne de sous-typage $\Rightarrow_{\text{transitivité de } <:}$ sous-typage.
3. C'est pourquoi je distingue héritage direct et généralisé. **Attention :** une instance d'une classe contient, physiquement, tous les attributs d'instance définis dans ses superclasses, même masqués ou non visibles.

Héritage : la racine

La classe `Object`

La classe `Object` est :

- superclasse de toutes les classes ;
- superclasse directe de toutes les classes sans clause **`extends`** (dans ce cas, « **`extends`** `Object` » est implicite) ;
- racine de l'arbre d'héritage des classes [1].

Et le type `Object` qu'elle définit est :

- supertype de tous les types références (y compris interfaces) ;
- supertype direct des classes sans clause ni **`extends`** ni **`implements`** et des interfaces sans clause **`extends`** ;
- unique source du graphe de sous-typage des types références.

1. Ce graphe a un degré d'incidence de 1 (héritage simple) et une source unique, c'est donc un arbre. Notez que le graphe d'héritage des interfaces n'est pas un arbre mais un DAG (héritage multiple) à plusieurs sources et que le graphe de sous-typage des types références est un DAG à source unique.

Héritage : la racine

La classe `Object` : ses méthodes (1)

`Object` possède les méthodes suivantes :

- **`boolean`** `equals(Object other)` : teste l'égalité de **`this`** et `other`
- `String toString()` : retourne la représentation en `String` de l'objet [1]
- **`int`** `hashCode()` : retourne le « hash code » de l'objet [2]
- `Class<?> getClass()` : retourne l'objet-classe de l'objet.
- **`protected`** `Object clone()` : retourne un « clone » [3] de l'objet si celui-ci est `Cloneable`, sinon quitte sur exception `CloneNotSupportedException`.
- **`protected void`** `finalize()` : appelée lors de la destruction de l'objet.
- et puis `wait`, `notify` et `notifyAll` que nous verrons plus tard (cf. *threads*).

1. Utilisée notamment par `println` et dans les conversions implicites vers `String` (opérateur « + »).
2. Entier calculé de façon déterministe depuis les champs d'un objet, satisfaisant, par contrat, `a.equals(b)` $\Longrightarrow$ `a.hashCode()== b.hashCode()`.
3. Attention : le rapport entre `clone` et `Cloneable` est plus compliqué qu'il en a l'air, cf. EJ3 Item 13.

Héritage : la racine

La classe `Object` : ses méthodes (2)

Conséquences :

- Grâce au sous-typage, tous les types référence ont ces méthodes, on peut donc les appeler sur toute expression de type référence.
- Grâce à l'héritage, tous les objets disposent d'une implémentation de ces méthodes...
  ... mais leur implémentation faite dans `Object` est souvent peu utile :
  - `equals` : teste l'identité (égalité des adresses, comme ==) ;
  - `toString` : retourne une chaîne composée du nom de la classe et du `hashCode`.

  $\rightarrow$ toute classe devrait redéfinir `equals` (et donc [1] `hashCode`) et `toString` (cf. EJ3 Items 10, 11, 12).

1. Rappel du transparent précédent : si `a.equals(b)` alors il faut `a.hashCode()== b.hashCode()`.

Héritage : ajout, redéfinition, masquage

Le panorama... (1)

Dans une sous-classe :

- On **hérite** des membres visibles [1] de la superclasse directe [2].
  Visible = **`public`**, **`protected`**, voire *package-private*, si superclasse dans même *package*.
- On peut **masquer** (*to hide*) n'importe quel membre hérité :
  - méthodes : par une définition de même signature
    dans ce cas, le type de retour doit être identique [3], sinon erreur de syntaxe !
  - autres membres : par une définition de même nom
- Les autres membres de la sous-classe sont dits **ajoutés**.

...

1. Il faut en fait visibles et non-**`private`**. En effet : **`private`** est parfois visible (cf. classes imbriquées).
2. que ceux-ci y aient été directement définis, ou bien qu'elle les aie elle-même hérités
3. En fait, si le type de retour est un type référence, on peut retourner un sous-type. Par ailleurs il y a des subtilités dans le cas des types paramétrés, cf généricité.

Héritage : ajout, redéfinition, masquage

Le panorama... (2)

...

- Une méthode d'instance (non statique) masquée est dite **redéfinie** [1] (*overridden*).

  Dans le cas d'une redéfinition, il est interdit de :

  - redéfinir une méthode `final`,
  - réduire la visibilité (*e.g.* redéfinir une méthode `public` par une méthode `private`),
  - ajouter une clause `throws` ou bien d'ajouter une exception dans la clause `throws` héritée (cf. cours sur les exceptions).

  La notion de redéfinition est importante en POO (cf. liaison dynamique).

1. Mon parti pris : redéfinition = cas particulier du masquage. D'autres sources restreignent, au contraire, la définition de « masquage » aux cas où il n'y a pas de redéfinition (« masquage simple »).
La JLS dit que les méthodes d'instance sont redéfines et jamais qu'elles sont masquées... mais ne dit pas non plus que le terme est inapproprié.

Héritage : ajout, redéfinition, masquage

Remarques diverses (1)

- L'accès à toute définition visible (masquée ou pas) de la surperclasse est toujours possible via le mot-clé **`super`**. Par ex. : **`super`**`.toString( )`.
- Les définitions masquées ne sont pas effacées. En particulier, un attribut masqué contient une valeur indépendante de la valeur de l'attribut qui le masque.

```
class A { int x; }
class B extends A { int x; } // le x de A est masqué par celui-ci
...
B b = new B(); // <- mais cet objet contient bien deux int
```

- De même, les définitions non visibles des superclasses restent « portées » par les instances de la sous-classe, même si elles ne sont pas accessibles directement.

```
class A { private int x; } // x privé, pas hérité par classe B
class B extends A { int y; }
...
B b = new B(); // <- mais cet objet contient aussi deux int
```

Héritage : ajout, redéfinition, masquage

Remarques diverses (2)

- Les membres non hérités ne peuvent pas être masqués ou redéfinis, mais rien n'empêche de définir à nouveau un membre de même nom (= ajout).

```
class A { private int x; }
class B extends A { int x; } // autorisé !
// et tant qu'on y est :
B b = new B(); // là  encore, cet objet contient deux int !
```

Héritage : ajout, redéfinition, masquage

Exemple


```
class GrandParent {
    protected static int a, b; // visibilité protected, assure que l'héritage se fait bien
    protected static void g() {}
    protected void f() {}
}

class Parent extends GrandParent {
    protected static int a; // masque le a hérité de GrandParent (tjs accessible via super.a)
    // masque g() hérité de GrandParent (tjs appelable via super.g()).
    protected static void g() {}
    // redéfinit f() hérité de GrandParent (tjs appelable via super.f()).
    @Override protected void f() {}
}

class Enfant extends Parent { @Override protected void f() {} }
```

- La classe `Enfant` hérite `a`, `g` et `f` de `Parent` et `b` de `GrandParent` via `Parent`.
- `a` et `g` de `GrandParent` masqués mais accessibles via préfixe `GrandParent.`.
- `f` de `Parent` héritée mais redéfinie dans `Enfant`. Appel de la version de `Parent` avec `super.f()`.
- `f` de `GrandParent` masquée par celle de `Parent` mais peut être appelée sur un récepteur de classe `GrandParent`. **Remarque :** ~~`super.super`~~ n'existe pas.

Héritage : ajout, redéfinition, masquage

Cas tordu

Un ajout simple dans un contexte peut provoquer un masquage dans un autre :

```java
package bbb;
public class B extends aaa.A {
    public void f() { System.out.println("B"); }    // avec @Override, ça ne compilerait pas.
    // En effet, bbb.B ne voit pas la f de aaa.A. C'est donc un ajout de nouvelle méthode !
}
```

```java
package aaa;
public class A {
    void f() { System.out.println("A"); } // méthode package—private, invisible dans bbb
    public static void main(String[] args) {
        bbb.B b = new bbb.B();
        b.f(); // contexte : récepteur de type B, ici f de bbb.B masque f de aaa.A
        ((A) b).f(); // contexte : récepteur de type A, f de aaa.A ni masquée ni redéfinie
    }
}
```

Ici, une méthode d'instance en masque une autre sans la redéfinir.

$\rightarrow$ Contradiction apparente avec ce qui avait été dit.

En réalité masquage implique redéfinition seulement s'il y a masquage dans le contexte où est définie la méthode masquante.

Liaisons statique et dynamique

Non, mais sérieusement, pourquoi distinguer la redéfinition du masquage simple ?

À la compilation : dans tous les cas, chaque occurrence de nom de méthode est traduite comme référence vers une méthode existant dans le contexte d'appel.

À l'exécution :

- Autres membres que méthodes d'instance : la méthode trouvée à la compilation sera effectivement appelée.
  → Mécanisme de **liaison statique** (ou précoce).
- Méthodes d'instance [1] : une méthode **redéfinissant** la méthode trouvée à la compilation sera recherchée, depuis le contexte de la classe de l'objet récepteur.
  → Mécanisme de **liaison dynamique** (ou tardive).
  Le résultat de cette recherche peut être différent à chaque exécution.
  Ce mécanisme permet au polymorphisme par sous-typage de fonctionner.

1. Sauf méthodes privées et sauf appel avec préfixe "`super.`" → liaison statique.

Liaisons statique et dynamique

Exemple


```java
class A {
   public A() {
     f();
     g();
   }
   static void f() {
     System.out.println("A::f");
   }
   void g() {
     System.out.println("A::g");
   }
}
```

```java
class B extends A {
   public B() {
     super();
   }
   static void f() { // masquage simple
     System.out.println("B::f");
   }
   @Override
   void g() { // redéfinition
     System.out.println("B::g");
   }
}
```

Si on fait `new B();`, alors on verra s'afficher

```
A::f
B::g
```

Liaison statique

Comment le compilateur traduit-il les noms de membres apparaissant dans un code source ?

Principe de la liaison statique : dès la compilation, on décide quelle définition sera effectivement utilisée pour l'exécution (c.-à-d. **toutes** les exécutions).

Pour l'explication, nous distinguons cependant :

- d'abord le cas simple (tout membre sauf méthode)
- ensuite le cas moins simple des méthodes (possible surcharge)

Attention : seules les méthodes d'instance peuvent être liées dynamiquement (et le sont habituellement [1]) ; pour tous les autres membres elle est toujours statique.

1. Les méthodes d'instance peuvent parfois être sujets à une liaison uniquement statique : méthodes `private` ; ainsi que toute méthode lorsqu'elle est appelée avec préfixe `super.`.

Liaison statique

Principe : cas sans surcharge (membres non-méthode)

Pour trouver la bonne définition d'un membre (non méthode) de nom `m`, à un point donné du programme.

- Si *m* membre statique, soit *C* le contexte [1] d'appel de *m*. Sinon, soit *C* la classe de l'objet sur lequel on appelle *m*.
- On cherche dans le corps de *C* une définition visible et compatible (même catégorie de membre, même "staticité", même type ou sous-type...), puis dans les types parents de C (superclasse et interfaces implémentées), puis les parents des parents (et ainsi de suite).
  On utilise la première définition qui convient.

Pour les méthodes : même principe, mais on garde toutes les méthodes de signature compatible avec l'appel, puis on applique la résolution de la surcharge. Ce qui donne...

1. le plus souvent classe ou interface, en toute généralité une définition de type

Liaison statique

Principe : cas des méthodes

Quelle définition de `f` utiliser, quand on appelle `f(x1, x2, ...)` [1] ?

1. `C` := contexte de l'appel de la méthode (classe ou interface).
2. Soit M_f := { méthodes de nom « `f` » dans `C`, compatibles avec `(x1, x2, ...)`}.
3. Pour tout supertype direct `S` de `C`, M_f += { méthodes de nom « `f` » dans `S`, compatibles avec `x1, x2, ...`, non masquées [2] par autre méthode dans M_f}.
4. On répète (3) avec les supertypes des supertypes, et ainsi de suite. [3].
5. On résout la surcharge parmi M_f (i.e. : on prend la signature la plus spécifique).

Le compilateur ajoute au code-octet l'instruction **invokestatic** [4] avec pour paramètre une référence vers la méthode trouvée.

1. Avec `f` habituellement statique, mais pas toujours cf. précédemment.
2. À cause de la surcharge il peut exister des méthodes de même nom non masquées
3. Jusqu'aux racines du graphe de sous-typage.
4. Pour les méthodes statiques. Pour les méthodes d'instance **`private`** ou **`super`**, c'est **invokespecial**.

Liaison dynamique

Principe

Lors de l'appel `x.f(y)`, quelle définition de `f` choisir ?

→ Principe de la liaison dynamique :

1. à la compilation : la même recherche que pour la liaison statique est exécutée (recherche depuis le **type statique** `S` de `x`), mais le compilateur ajoute au code-octet l'instruction **invokevirtual** (si `S` est une classe) ou **invokeinterface** (si `S` est une interface) au lieu de **invokestatic**.
2. à l'exécution : quand la JVM lit **invokevirtual** ou **invokeinterface**, une **redéfinition** de la méthode trouvée en (1) est recherchée dans la classe `C` de l'objet référencé (= **type dynamique** de `x` [1]), puis récursivement dans ses superclasses successives, puis dans les interfaces implémentées (méthode **`default`**) [2] [3] .

1. Le type dynamique de `y` n'est jamais pris en compte (java est *single dispatch*).
2. En fait, seuls les supertypes de `C` sous-types de `S` peuvent contenir des redéfinitions.
3. Comme on se limite aux redéfinitions valides, les éventuelles surcharges ajoutées dans `C` par rapport à `S`, sont ignorées à cette étape. Cf. exemples.

Liaison dynamique

Table virtuelle (implémentation optimisée de la liaison dynamique)

En pratique, pour chaque classe C, la JVM établit une fois pour toutes une **table virtuelle**, associant à chaque méthode d'instance (nom et signature), un pointeur [1] vers le code qui doit être exécuté quand un appel est effectué sur une instance directe de C.

Ainsi, à chaque appel, la liaison dynamique se fait **en temps constant**.

Le calcul de cette table prend en compte les méthodes héritées, redéfinies et ajoutées :

1. la table virtuelle de C est initialisée comme copie de celle de sa superclasse ;
2. y sont ajoutées des entrées pour les méthodes déclarées dans les interfaces implémentées par C ; [2]
3. les redéfinitions de C écrasent les entrées correspondantes déjà existantes ; [3]
4. les ajouts de C sont ajoutés à la fin de la table.

1. Pointeur `null` si la méthode est abstraite.
2. Contenant `null` ou bien pointeur vers le code de la méthode `default`, le cas échéant.
3. Elles existent forcément, sinon ce ne sont pas des redéfinitions !

Liaison dynamique

Cas simple avec redéfinition

- Classe dérivée : certaines méthodes peuvent être redéfinies

```
class A { void f() { System.out.println("classe A"); } }
class B extends A { void f() { System.out.println("classe B"); } }
public class Test {
  public static void main(String args[]) {
    B b = new B();
    b.f(); // <-- affiche "classe B"
  }
}
```

- mais aussi...

```
public class Test {
  public static void main(String args[]) {
    A b = new B(); // <-- maintenant variable b de type A
    b.f(); // <-- affiche "classe B" quand-même
  }
}
```

Liaison dynamique

Cas compliqué, question ?

Imaginons le cas suivant, avec redéfinition et surcharge :

```
class Y1 {}

class Y2 extends Y1 {}

class X1 { void f(Y1 y) { System.out.print("X1 et Y1 ; "); } }

class X2 extends X1 {
    void f(Y1 y) { System.out.print("X2 et Y1 ; "); }
    void f(Y2 y) { System.out.print("X2 et Y2 ; "); }
}

class X3 extends X2 { void f(Y2 y) { System.out.print("X3 et Y2 ; "); } }

public class Liaisons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicites ci-dessous (servent-ils vraiment à rien ?)
        ((X1) x).f((Y1) y);      ((X1) x).f(y);
        ((X2) x).f((Y1) y);      ((X2) x).f(y);
        x.f((Y1) y);             x.f(y);
    }
}
```

Qu'est-ce qui s'affiche ?

Liaison dynamique

Cas compliqué, réponse !

```
class Y1 {}
class Y2 extends Y1 {}
class X1 { void f(Y1 y) { System.out.print("X1 et Y1 ; "); } }
class X2 extends X1 {
    void f(Y1 y) { System.out.print("X2 et Y1 ; "); }
    void f(Y2 y) { System.out.print("X2 et Y2 ; "); }
}
class X3 extends X2 { void f(Y2 y) { System.out.print("X3 et Y2 ; "); } }

public class Liaisons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicites ci-dessous (servent-ils vraiment à rien ?)
        ((X1) x).f((Y1) y);     ((X1) x).f(y);     ((X2) x).f((Y1) y);
        ((X2) x).f(y);          x.f((Y1) y);       x.f(y);
    }
}
```

Affiche : `X2 et Y1 ; X2 et Y1 ; X2 et Y1 ; X3 et Y2 ; X2 et Y1 ; X3 et Y2 ;`

- Pour les instructions commençant par `((X1)x).` : la phase statique cherche les signatures dans `X1` → les surcharges prenant `Y2` sont ignorées à l'exécution.
- Les instructions commentçant par `((X2)x).` se comportent comme celles commençant par `x.` : les mêmes signatures sont connues dans `X2` et `X3`.

Liaison dynamique

Retour sur redéfinition et surcharge

Attention aux “redéfinitions ratées” : ça peut compiler mais...

- si on se trompe dans le type ou le nombre de paramètres, ce n’est pas une redéfinition [1], mais un ajout de méthode surchargée. Erreur typique :

```
public class Object { // la ``vraie'', c.-à -d. java.lang.Object
    ...
    public boolean equals(Object obj) { return this == obj; }
    ...
}

class C /* sous-entendu : extends Object */ {
    public boolean equals (C obj) { return ....; } // <- c'est une surcharge, pas
        une redéfinition !
}
```

- **Recommandé :** placer l’annotation `@Override` devant une définition de méthode pour demander au compilateur de générer une erreur si ce n’est pas une redéfinition.

  Exemple : `@Override public boolean equals(Object obj){ ... }`

1. même pas un masquage

Méthodes et classes finales

On peut déclarer une méthode avec modificateur **`final`** [1]. Exemple :

```
class Employee {
    private String name;
    . . .
    public final String getName() { return name; }
    . . .
}
```

$\Longrightarrow$ ici, **`final`** empêche une sous-classe de `Employee` de redéfinir `getName()`. [2]

Aussi possible :

```
final class Employee {  . . . }
```

$\Longrightarrow$ ici, **`final`** interdit d'étendre la classe `Employee`

1. **Attention :** une variable peut aussi être déclarée avec le mot-clé **`final`**. Sa signification est alors différente : il interdit juste toute nouvelle affectation de la variable après son initialisation.
2. Ainsi, pour résumer, on a le droit de redéfinir les méthodes héritées non **`static`** et non **`final`**.

Méthodes et classes abstraites

- **Méthode abstraite** : méthode déclarée sans être définie.
  Pour déclarer une méthode comme abstraite, faire précéder sa déclaration du mot-clé **`abstract`**, et ne pas écrire son corps (reste la signature suivie de « **;** »).
- **Classe abstraite** : classe déclarée comme **non directement instanciable**.
  Elle se déclare en faisant précéder sa déclaration du modificateur **`abstract`** :

```
abstract class A {
    int f(int x) { return 0; }
    abstract int g(int x);   // <- oh, une méthode abstraite !
}
```

- **Le lien entre les 2 :** une méthode abstraite ne peut être pas déclarée dans un type directement instanciable → seulement dans interfaces et classes abstraites.
  Interprétation : tout objet instancié doit connaître une implémentation pour chacune de ses méthodes.
- Une méthode abstraite a vocation à être redéfinie dans une sous-classe.
  Conséquence : ~~**`abstract static`**~~, ~~**`abstract final`**~~ et ~~**`abstract private`**~~ sont des non-sens !

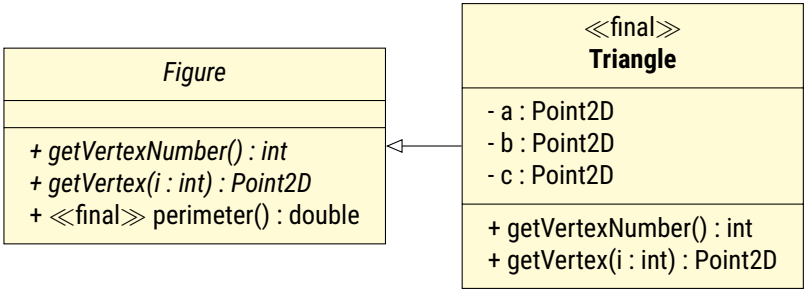
Méthodes et classes abstraites

Exemple


```
abstract class Figure {
    Point2D centre; String nom; // autres attributs éventuellement
    public abstract int getVertexNumber();
    public abstract Point2D getVertex(int i);
    public final double perimeter() {
        double peri = 0;
        Point2D courant = getVertex(0);
        for (int i=1; i < getVertexNumber(); i++) {
            Point2D suivant = getVertex(i);
            peri += courant.distance(suivant);
            courant = suivant;
        }
        return peri + courant.distance(getVertex(0));
    }
}

final class Triangle extends Figure {
    private Point2D a, b, c;
    @Override public int getVertexNumber() {
        return 3;
    }
    @Override public Point2D getVertex(int i) {
        switch(i) {
            case 0: return a;
            case 1: return b;
            case 2: return c;
            default: throw new NoSuchElementException();
        }
    }
}
```

Méthodes et classes finales ou abstraites

Notation UML



Remarquez l'italique pour les méthodes et classes abstraites. En revanche, `final` n'a pas de typographie particulière [1].

1. `final` n'est pas un concept de la spécification d'UML, mais heureusement, UML autorise à ajouter des informations supplémentaires en tant que « stéréotypes », écrits entre doubles chevrons.

abstract et final

le bon usage pour les classes (1)

- **`abstract`** et **`final`** contraignent la façon dont une classe s'utilise.
- Pourquoi contraindre ? → pour empêcher une utilisation incorrecte non prévue (cf. exemples ci-après).

Plus précisément :

- **`final`**, en figeant les méthodes (une ou toutes) d'une classe, permet d'assurer des propriétés qui resteront vraies pour toutes les instances de la classe.
- **`abstract`** (appliqué à une classe [1]) empêche qu'une classe qui ne représenterait qu'une implémentation incomplète ne soit instanciée directement.

Dans les deux cas, on interdit la possibilité d'instances absurdes (respectivement incohérents ou incomplets) de la classe marquée.

1. **`abstract`**, appliqué à une méthode, n'est une contrainte que dans la mesure où cela force à marquer aussi **`abstract`** la classe la contenant.

abstract et final

le bon usage pour les classes (2)

Constat : une classe non finale correspond à une implémentation complétable.

Idéologie : si c'est complétable c'est que c'est donc probablement incomplet. [1]

Si cela est vrai, alors une classe ni finale ni abstraite est louche ! [2]. Comme ~~`abstract final`~~ est exclus d'office, toute classe devrait alors être soit (juste) `abstract` soit (juste) `final`.

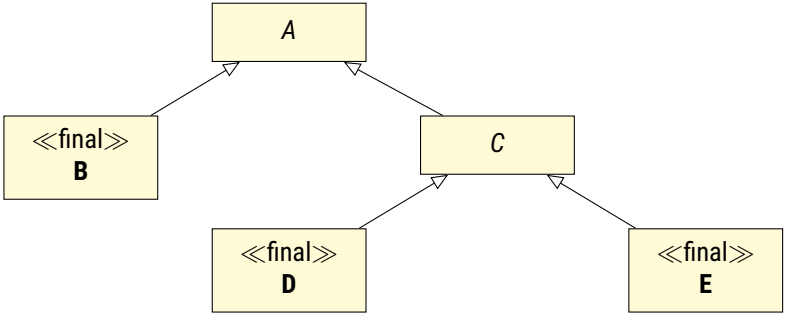
	pas `abstract`	`abstract`
pas `final`	louche	OK
`final`	OK	interdit

1. Ce n'est pas toujours vrai : certaines classes proposent un comportement par défaut tout à fait valable, tout en laissant la porte ouverte à des modifications (cf. composants Swing).
2. On parle de *code smell*. Cela dit, c'est « louche », mais pas absurde, cf. remarque précédente.

abstract et final

le bon usage pour les classes : illustration en UML

En UML, une bonne structure d'héritage donnerait des diagrammes comme celui-ci :



abstract et final

le bon usage pour les classes : exemple

Exemple (à ne pas faire !) :

```
class Personne {
    public String getNom() { return null; } // mauvaise implémentation par défaut
}

class PersonneImpl extends Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}
```

Mieux :

```
abstract class Personne {
    public abstract String getNom();
}

final class PersonneImpl extends Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}
```

abstract et final

le bon usage pour les méthodes : contre-exemple

À ne pas faire non plus :

```
class Personne {
    private String prenom, nom;
    public String getPrenom() { return prenom; } // il faudrait final
    public String getNom() { return nom; } // là aussi
    public String getNomComplet() {
        return getPrenom() + "␣" + getNom(); // appel à méthodes redéfinissables —> danger !!!
    }
}
```

Sans **`final`**, `Personne` est une **classe de base fragile**. Quelqu'un pourrait écrire :

```
class Personne2 extends Personne {
    @Override public String getPrenom() { return getNomComplet().split("␣")[0] }
    @Override public String getNom() { return getNomComplet().split("␣")[1] }
}
```

... puis exécuter **`new`** `Personne2(...).getNom()`, qui appelle `getNomComplet()`, qui appelle `getPrenom()` et `getNom()`, qui appellent `getNomComplet()` qui appelle...

Récursion non bornée ! → `StackOverflowError`.

abstract et final

le bon usage pour les méthodes

Quand on programme une classe extensible :

- Si possible, éviter tout appel, depuis une autre méthode de la classe [1], de méthode redéfinissable (= non `final` = « ouverte »).
- À défaut le signaler dans la documentation.
- Objectif : éviter des erreurs bêtes dans les futures extensions.
  Par exemple : appels mutuellement récursifs non voulus.
- La documentation devra donner une spécification des méthodes redéfinissables assurant de conserver un comportement globalement correct.

1. Cela vaut aussi pour les appels de méthodes depuis une méthode `default` dans une interface.

abstract et final

L'héritage casse-t-il l'encapsulation ?

On entend souvent dire « L'héritage casse l'encapsulation. » (mais c'est exagéré).

Signification : pour qu'une classe soit étendue correctement, documenter ses membres `public` ne suffit pas ; certains points d'implémentation [1] doivent aussi l'être.

→ Cela contredit l'idée que l'implémentation d'une classe devrait être une « boîte noire ».

À défaut de pouvoir faire cet effort de documentation pour une classe, il est plus raisonnable d'interdire d'hériter de celle-ci (→ `final class`).

EJ3, Item 19 : « *Design and document for inheritance or else prohibit it* »

1. À commencer par l'évidence : les membres `protected`. Mais même cela ne suffit pas.

abstract et final

le bon usage : résumé

Une stratégie simple et extrême :

- Déclarer **`final`** toute classe destinée à être instanciée.
  ⇔ feuilles de l'arbre d'héritage.
- Déclarer **`abstract`** toute classe destinée à être étendue [1].
  ⇔ nœuds internes de l'arbre d'héritage.
- Dans ce dernier cas, déclarer en **`private`** ou **`final`** tous les membres qui peuvent l'être, afin d'empêcher que les extensions cassent les contrats déjà implémentés.
- Écrire la spécification de toute méthode redéfinissable (telle que, si elle est respectée, les contrats soient alors aussi respectés).

1. Voire, si la classe n'a pas d'attribut d'instance, déclarer plutôt une interface !

Application : les types scellés

Définition et intérêt

- **Type scellé** : type dont l'ensemble des sous-types est fixé à la compilation de celui-ci.
- **Utilité :**
  - Il devient possible de prouver que les contrats du type sont bien implémentés pour toutes les instances présentes et futures : il suffit de le prouver pour un ensemble de cas fini et déjà connu.
  - Les fameuses listes de `else if (... instanceof ...){ ... }` deviennent plus acceptables car l'exhaustivité est garantie. Cela est utile quand la liaison dynamique ne peut pas être utilisée [1].
- **Exemples :** une classe `final`, n'ayant pas de sous-type du tout, est clairement scellée. De même, toute `enum` (étant par définition un type fini).

1. Notamment :
   - besoin d'écrire une méthode dont les comportements varient en fonction des types de plusieurs paramètres (impossible : la liaison dynamique est *single disptach*) ;
   - besoin d'ajouter un comportement à un type fourni par un tiers (impossible d'y ajouter une méthode).

Application : les types scellés

Les types scellés de Java

Plus souple : classes à constructeurs tous privés. Une telle classe est instanciable et extensible seulement depuis l'intérieur de son corps ou de celui de son type englobant.

Théorème : [1] en Java [2], les types scellés sont exactement ceux définis par une classe `final` ou à constructeurs privés telle que ses sous-classes directes sont aussi scellées.

Preuve :

- $\Leftarrow$ Par récurrence, toute l'arborescence de sous-types est imbriquée dans un même type englobant, donc définie dans le même fichier `.java`. Il n'est donc pas possible d'ajouter un sous type sans modifier le fichier et le recompiler.
- $\Rightarrow$ Réciproquement, si une classe n'est pas `final` et a un constructeur non privé, on peut alors l'étendre depuis un autre fichier.

1. Correct si on considère que les classes privées et locales sont à constructeurs privés.
2. Dans d'autres langages (Scala, Kotlin), un mot-clé **sealed** permet de déclarer un type scellé sans bricoler avec les constructeurs. Ce mot-clé est maintenant en *preview* dans Java 15, cf. JEP 360.

Application : les types scellés

Exemple typique : type algébrique


```java
public abstract class BoolExpr { // classe scellée (et abstraite !)
    public abstract boolean eval();
    private BoolExpr() {}

    public static final class Var extends BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; } // super() est accessible car Var est imbriquée
        @Override public boolean eval() { return value; }
    }

    public static final class Not extends BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; } // même remarque
        @Override public boolean eval() { return !subexpr.eval(); }
    }

    public static final class And extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; } // même remarque
        @Override public boolean eval() { return subexpr1.eval() && subexpr2.eval(); }
    }

    public static final class Or extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; } // même remarque
        @Override public boolean eval() { return subexpr1.eval() || subexpr2.eval(); }
    }
}
```

Application : les types scellés

Exemple typique : type algébrique, version non objet


```
public abstract class BoolExpr {
    public static boolean eval(BoolExpr expr) { // en vrai, la version précédente était mieux
        if (expr instanceof Var) return ((Var) expr).value;
        else if (expr instanceof Not) return !eval(((Not) expr).subexpr);
        else if (expr instanceof And) return eval(((And) expr).subexpr1) && eval(((And) expr).subexpr2);
        else if (expr instanceof Or) return eval(((Or) expr).subexpr1) || eval(((Or) expr).subexpr2);
        else { assert false : "Cannot␣happen :␣the␣pattern␣matching␣is␣exhaustive!"; return false; }
    }
    private BoolExpr() {}

    public static final class Var extends BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not extends BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }

    public static final class Or extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }
```

Application : les types scellés

Exemple typique : type algébrique, version Java 15 preview


```java
public sealed interface BoolExpr { // sealed, pas besoin de constructeur privé ! (et interface autorisée)
    static boolean eval(BoolExpr expr) {
        if (expr instanceof Var varExpr) return varExpr.value; // on profite du nouveau pattern matching
        else if (expr instanceof Not notExpr) return !eval(notExpr.subexpr);
        else if (expr instanceof And andExpr) return eval(andExpr.subexpr1) && eval(andExpr.subexpr2);
        else if (expr instanceof Or orExpr) return eval(orExpr.subexpr1) || eval(orExpr.subexpr2);
        else { assert false : "Cannot happen : the pattern matching is exhaustive!"; return false; }
    }

    public static final class Var implements BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not implements BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
              subexpr2; }
    }

    public static final class Or implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
              subexpr2; }
    }
}
```

Application : les types scellés

Exemple typique : type algébrique, avec pattern-matching switch (futur)


```java
public sealed interface BoolExpr {
    static boolean eval(BoolExpr expr) {
        return switch(expr) {
            case Var varExpr -> varExpr.value;
            case Not notExpr -> !eval(notExpr.subexpr);
            case And andExpr -> eval(andExpr.subexpr1) && eval(andExpr.subexpr2);
            case Or orExpr -> eval(orExpr.subexpr1) || eval(orExpr.subexpr2);
        } // liste exhaustive, pas besoin de cas default !
    }

    public static final class Var implements BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not implements BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }

    public static final class Or implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }
```

Types finis

Pour quoi faire ?

Un **type fini** est un type ayant un ensemble fini d'instances, toutes définies statiquement dès l'écriture du type, sans possibilité d'en créer de nouvelles lors de l'exécution. [1]

Certaines variables ont, en effet, une valeur qui doit rester dans un ensemble fini, prédéfini :

- les 7 jours de la semaine
- les 4 points cardinaux
- les 3 (ou 4 ou plus) états de la matière
- les *n* états d'un automate fini (dans protocole ou processus industriel, par exemple)
- les 3 mousquetaires, les 7 nains, les 9 nazgûls...

→ Situation intéressante car, théoriquement, nombre fini de cas à tester/vérifier.

1. C'est donc un type scellé (très contraint) : clairement, si un type n'est pas scellé, il ne peut pas être fini.

Types finis

Pour quoi faire ?

Pourquoi définir un type fini plutôt que réutiliser un type existant ?

- Typiquement, types de Java trop grands [1]. Si utilisés pour représenter un ensemble fini, difficile voire impossible de prouver que les variables ne prennent pas des valeurs absurdes.
- Même si on l'a prouvé sur papier, le programe peut comporter des typos (ex : `"lnudi"` au lieu de `"lundi"`), que le compilateur ne les verra pas.

Avec un type fini, le compilateur garantit que la variable reste dans le bon ensemble. [2]

---

1. Soit très grands (p. ex., il y a 2^{32} **`int`**s), soit quasi-infinis (il ne peut pas exister plus de 2^{32} références en même temps, mais à l'exécution, un objet peut être détruit et un autre recréé à la même adresse...).
2. Il pourrait aussi théoriquement vérifier l'exhaustivité des cas d'un **`switch`** (sans **`default`**) ou d'un **`if`** / **`else if`** (sans **`else`** seul) : ça existe dans d'autres langages, mais `javac` ne le fait pas [3]. Intérêt : éviter des **`default`** et des **`else`** que l'on sait inatteignables.

Écrire un type fini

Comment faire ?

- **Mauvaise idée :** réserver un nombre fini de constantes dans un type existant (ça ne résout pas les problèmes évoqués précédemment).

  Remarque : c'est ce que fait la construction **enum** du langage C. Les constantes déclarées sont en effet des **int**, et le type créé est un *alias* de **int**.
- On a déjà vu qu'il fallait créer un nouveau type.
- Il faut qu'il soit impossible d'en créer des instances en dehors de sa déclaration...
- ... qu'elles soient directes (appel de son constructeur) ou indirectes (via extension).
- **Bonne idée :** implémenter le type fini comme classe à constructeurs privés et créer les instances du type fini comme constantes statiques de la classe :

```
public class Piece { // peut être final... mais le constructeur privé suffit
    private Piece() {}
    public static final Piece PILE = new Piece(), FACE = new Piece();
}
```

→ les **enum** de Java sont du sucre syntaxique pour écrire cela (+ méthodes utiles).

Classes d'énumération

```
public enum ETAT { SOLIDE, LIQUIDE, GAZ, PLASMA }
```

Une **classe d'énumération** (ou juste énumération) est une classe particulière, déclarée par un bloc synaxique **`enum`**, dans lequel est donnée la liste (exhaustive et définitive) des instances (= « constantes » de l'**`enum`**.).

Elle définit un **type énuméré**, qui est un type :

- fini : c'est la raison d'être de cette construction ;
- pour lequel l'opérateur « == » teste bien l'égalité sémantique [1]
  (toutes les instances représentent des valeurs différentes) ;
- utilisable en argument d'un bloc **`switch`** ;
- et dont l'ensemble des instances s'itère facilement :
  `for (MonEnum val: MonEnum.values()){...}`

1. Pour les enums, identité et égalité sont synonymes.

Énumérations

La base

Exemple simple :

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public class Test {
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            switch (d) {
            case SUNDAY:
            case SATURDAY:
                System.out.println(d + ": sleep");
                break;
            default:
                System.out.println(d + ": work");
            }
        }
    }
}
```

Énumération = classe particulière

Il est possible d'écrire une classe équivalente sans utiliser le mot-clé **enum** [1].

L'exemple précédent pourrait (presque [2]) s'écrire :

```
public final class Day extends Enum<Day> {
    public static final Day SUNDAY = new Day("SUNDAY", 0),
            MONDAY = new Day("MONDAY", 1), TUESDAY = new Day("TUESDAY", 2),
            WEDNESDAY = new Day("WEDNESDAY", 3), THURSDAY = new Day("THURSDAY", 4),
            FRIDAY = new Day("FRIDAY", 5), SATURDAY = new Day("SATURDAY", 6);

    private Day(String name, int ordinal) {
        super(name, ordinal);
    }

    // plus méthodes statiques valueOf() et values()
}
```

Enum<E> est la superclasse directe de toutes les classes déclarées avec un bloc **enum**. Elle contient les fonctionnalités communes à toutes les énumérations.

1. Puisque c'est du sucre syntaxique !
2. En réalité, ceci ne compile pas : `javac` n'autorise pas le programmeur à étendre la classe Enum à la main. Cela est réservé aux vraies **enum**. Si on voulait vraiment toutes les fonctionnalités des **enum**, il faudrait réécrire les méthodes de la classe Enum.

Énumérations

Extensions

On peut donc y ajouter des membres, en particulier des méthodes :

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
    public boolean isWorkDay() {
        switch (this) {
        case SUNDAY:
        case SATURDAY:
            return false;
        default:
            return true;
        }
    }
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            System.out.println(d + ": " + (d.isWorkDay() ? "work" : "sleep"));
        }
    }
}
```

Énumérations

Extensions

On peut même ajouter des constructeurs (privés seulement). Auquel cas, il faut passer les paramètres du(d'un des) constructeur(s) à chaque constante de l'enum :

```
public enum Day {
    SUNDAY(false), MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY(false);
    final boolean isWorkDay;
    private Day(boolean work) {
        isWorkDay = work;
    }
    private Day() { // constructeur sans paramètre -> on peut aussi déclarer les
         constantes d'enum sans paramètre
        isWorkDay = true;
    }
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            System.out.println(d + ": " + (d.isWorkDay ? "work" : "sleep"));
        }
    }
}
```

Énumérations

Extensions

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {
    SUNDAY { @Override public boolean isWorkDay() { return false; } },
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY { @Override public boolean isWorkDay() { return false; } };

    public boolean isWorkDay() { return true; }

    public static void main(String[] args) {
        for (Day d : Day.values())
            System.out.println(d + ": " + (d.isWorkDay() ? "work" : "sleep"));
    }
}
```

Dans ce cas, la constante est l'instance unique d'une sous-classe [1] de l'**`enum`**.

Remarque : comme d'habitude, toute construction basée sur des `@Override` et la liaison dynamique est à préférer à un **`switch`** (quand c'est possible et que ça a du sens).

Énumérations

Types énumérés et sous-typage

- Tous les types énumérés étendent la classe `Enum` [1].
  Donc une énumération ne peut étendre aucune autre classe.
- En revanche rien n'interdit d'écrire `enum Truc implements Machin { ... }`.
- Les types enum sont des classes à constructeur(s) privé(s). [2]
  Ainsi aucune instance autre que les constantes déclarées dans le bloc `enum` ne pourra jamais exister [3].
- On ne peut donc pas non plus étendre un type énuméré [4].

1. Version exacte : l'énumération `E` étend `Enum<E>`. Voir la généricité.
2. Elles sont mêmes `final` si aucune des constantes énumérées n'est muni d'un corps de classe.
3. Ainsi, toutes les instances d'une `enum` sont connues dès la compilation.
4. On ne peut pas l'étendre « à la main », mais des sous-classes (singletons) sont compilées pour les constantes de l'enum qui sont munies d'un corps de classe.

Énumérations

Méthodes

Toute énumération `E` a les méthodes d'instance suivantes, héritées de la classe `Enum` :

- `int compareTo(E o)` (de l'interface `Comparable`, implémentée par la classe `Enum`) : compare deux éléments de `E` (en fonction de leur ordre de déclaration).
- `String toString()` : retourne le nom de la constante (une chaîne dont le texte est le nom de l'identificateur de la constante d'enum)
- `int ordinal()` : retourne le numéro de la constante dans l'ordre de déclaration dans l'enum.

Par ailleurs, tout type énuméré `E` dispose des deux méthodes statiques suivantes :

- `static E valueOf(String name)` : retourne la constante d'enum dont l'identificateur est égal au contenu de la chaîne `name`
- `static E[] values()` : retourne un tableau contenant les constantes de l'enum dans l'ordre dans lequel elles ont été déclarées.

Énumérations

Quand les utiliser

- **Évidemment :** pour implémenter un type fini (cf. intro de ce cours). Remarquez au passage toutes les erreurs potentielles si on utilisait, à la place d'une `enum` :
  - des `int` : tentation d'utiliser directement des littéraux numériques (`1`, `0`, `-42`) peu parlants au lieu des constantes (par flemme). Risque très fort d'utiliser ainsi des valeurs sans signification associée.
  - des `String` sous forme littérale : risque fort de faire une typo en tappant la chaîne entre guillemets.
- **Cas particulier :** quand une classe ne doit contenir qu'une seule instance (singleton) → le plus sûr pour garantir qu'une classe est un singleton c'est d'écrire une enum à 1 élément.

```
enum MaClasseSingleton /* insérer implements Machin */{
    INSTANCE; // <--- l'instance unique !
    /* insérer ici tous les membres utiles */
}
```

Tout cela peut être fait sans les `enum`s mais c'est fastidieux et risque d'être mal fait.

Énumérations

Bien les utiliser

- Les **`enum`**s sont bien pensés et robustes. Il est assez difficile de mal les utiliser.
- **Piège possible :** compter sur les ordinaux (**`int`** retourné par `ordinal()`) ou l'ordre relatif des constantes d'une **`enum`** $\rightarrow$ fragilité en cas de mise à jour de la dépendance fournissant l'**`enum`**.

Bonne pratique pour utiliser une enum fournie par un tiers : (EJ3 Item 35) ne compter ni sur le fait qu'une constante possède un ordinal donné, ni sur l'ordre relatif des ordinaux (= ordre des constantes dans tableau `values()`).

Énumérations et collections (1)

Il existe des implémentations de collections optimisées pour les énumérations.

- `EnumSet<E extends Enum<E>>`, qui implémente `Set<E>` : on représente un ensemble de valeurs de l'énumération `E` par un champ de bits (le bit n°*i* vaut 0 si la constante d'ordinal *i* est dans l'ensemble, 1 sinon). Cette représentation est très concise et très rapide.

  Création via méthodes statiques
  `Set<DAY> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY)`, voire
  `Set<Day> week = EnumSet.allOf(Day.class)`.

  L'usage d'`EnumSet` est à préférer à l'usage direct des champs de bits [1] (EJ3 Item 36). On gagne en clarté et en sécurité.

1. Vous savez, ces entiers qu'on manipule bit à bit via les opérateurs <<, >>, |, & et ~ et dont les programmeurs en C sont si friands...

Énumérations et collections (2)

- `EnumMap<K extends Enum<K>, V>` qui implémente `Map<K,V>` : une `Map` représentée (en interne) par un tableau dont la case d'indice *i* référence la valeur dont la clé est la constante d'ordinal *i* de l'enum `K`.

  Construire un `EnumMap` :
  `Map<Day, Activite> edt = new EnumMap<>(Day.class);`.

  `EnumMap` est à préférer à tout tableau ou toute liste où l'on utiliserait les ordinaux des constantes d'une **`enum`** en tant qu'indices (EJ3 Item 37).

Autour de l'héritage

Le bon usage de l'héritage de classe (1)

On a coutûme de dire que :

- la classe B hérite de la classe A seulement si un B **est un** A.
- on compose [1] A dans B quand un B **possède un** A.

Mais « **est un** » peut être interprété de plusieurs façons :

- une instance de A **peut être utilisée à la place d'** une instance de B $\leftrightarrow$ sous-typage.
- une instance de A **est faite comme** une instance de B [2] $\leftrightarrow$ héritage.

(Comme l'héritage implique le sous-typage, la seconde interprétation est plus « forte ».)

1. C'est-à-dire qu'on met dans A un attribut d'instance de type B. On reparle de composition juste après.
2. Mêmes champs en mémoire, appel du constructeur parent ; même code sauf si redéfini.

Autour de l'héritage

Le bon usage de l'héritage de classe (2)

On peut donc envisager de déclarer une classe B sous-classe d'une classe A existante par une classe B lorsque :

- B doit pouvoir être utilisée à la place de A,
- l'implémentation de B semble pouvoir se baser sur celle de A,
- et A est faite telle que l'héritage est possible.
  C.-à-d. : ni A ni les méthodes à redéfinir ne sont **`final`** et le code à ajouter ou modifier est en accord avec les instructions données dans la documentation de A. [1]

Mais...

1. Faute de documentation, évitez les constructions fragiles, comme par exemple, appeler une méthode héritée `f` depuis une méthode redéfnie `g` de B. En effet : sauf indication contraire, `f` est susceptible d'appeler `g` → risque de `StackOverFlowError`.

Autour de l'héritage

Limites de l'héritage de classe et avertissements

... ce n'est pas parce qu'on peut le faire que c'est une bonne idée !

- Si la classe B hérite de A, elle récupère toutes les fonctionnalités héritées de A, y compris celles qui n'auraient pas de rapport avec l'objectif de B. [1]

  (C'est le principe-même du sous-typage. Mais la vraie question est : est-ce mon intention de créer un sous-type ? Cette classe sera-t-elle utilisée dans un contexte polymorphe ?)
- Les instances de B contiennent tous les champs de A (y compris privés), même devenus inutiles → « surpoids » et risque d'incohérences.
- Étendre une classe qui n'était pas conçue pour cela expose à des comportements inattendus [2] (non documentés par son auteur... qui n'avait pas prévu ça !).

1. Et si ce n'est pas le cas maintenant, quid de la prochaîne version de A ?
2. Cf. cas de la « classe de base fragile » vu précédemment.

Pour des objectifs simples, préférer des techniques alternatives :

- créer du sous-typage → implémentation d'interface [1] [2].
  Une interface ne craint pas le syndrôme de la « classe de base fragile ». [3]
- réutiliser des fonctionnalités déjà programmées → **composition** [4] (utiliser un objet auxiliaire possédant les fonctionnalités voulues pour les ajouter à votre classe).
  Ici aussi, on ne risque pas de « perturber » des fonctionnalités déjà programmées.

1. Obligatoire et assumé dans des langages comme Rust, où l'héritage ne crée pas de sous-typage.
2. EJ3 20 : « *Prefer interfaces to abstract classes* »
3. Faux en cas de méthodes `default` → même besoin de documentation que pour l'héritage de classe.
4. EJ3 18 : « *Favor composition over inheritance* »

Autour de l'héritage

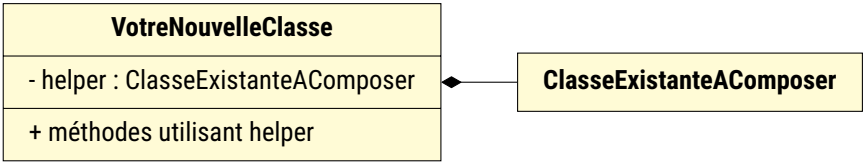
Alternative : ce qu'on entend par « composition »

Composition : utilisation d'un objet à l'intérieur d'un autre pour réutiliser les fonctionalités codées dans le premier. Exemple :

```
class Vendeur {
    private double marge;
    public Vendeur(double marge) { this.marge = marge; }
    public double vend(Bien b) { return b.getPrixRevient() * (1. + marge); }
}

class Boutique {
    private Vendeur vendeur;
    private final List<Bien> stock = new ArrayList<>();
    private double caisse = 0.;

    public Boutique(Vendeur vendeur) { this.vendeur = vendeur; }

    public void vend(Bien b) {
        if (stock.contains(b)) { stock.remove(b); caise += vendeur.vend(b); }
    }
}
```

`Boutique` réutilise des fonctionalités de `Vendeur` sans en être un sous-type.

Ces mêmes fonctionalités pourraient aussi être réutilisées par une autre classe `SuperMarche`.



Notez le losange plein.

UML distingue la composition de l'aggrégation (losange vide). La différence est subtile :

- composition : l'objet du côté du losange est considéré comme propriétaire de l'autre objet, dont le cycle de vie est lié à celui du premier.
- aggrégation : simple utilisation d'un objet par un autre sans que ce dernier ne soit propriétaire de l'autre.

En Java, la composition se traduit par l'absence de référence externe vers l'objet utilisé (le ramasse-miette peut le détruire dès que son propriétaire est détruit).

Objectif : ne pas hériter d'un « bagage » inutile

Exemple problématique

Mathématiquement les entiers sont sous-type des rationnels. Mais comment le coder ?

Pas terrible :

```
public class Rationnel {
    private final int numerateur, denominateur;
    public Rationnel(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}
public class Entier extends Rationnel { public Entier (int n) { super(n, 1); } }
```

Ici, toute instance d'entiers contient 2 champs (certes non visibles) : numérateur et dénominateur. Or 1 seul `int` aurait dû suffire.

→ utilisation trop importante de mémoire (pas très grave)

→ risque d'incohérence à cause de la redondance (plus grave)

Autre problème : la classe `Rationnel` visant à être immuable (attributs `final`) serait typiquement `final` (pour empêcher des sous classes avec attributs modifiables).

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide d'interfaces

Mieux :

```
public interface Rationnel { int getNumer(); int getDenom(); // + operations }
public interface Entier extends Rationnel {
    int getValeur();
    default int getNumer() { return getValeur(); }
    default int getDenom() { return 1; }
}
public final class RationnelImmuable implements Rationnel {
    private final int numerateur, denominateur;
    public RationnelImmuable(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}
public final class EntierImmuable implements Entier {
    private final intValue;
    public EntierImmuable (int n) { intValue = n; }
    @Override public int getValeur() { return intValue; }
}
```

Ainsi nos types existent en version immuable (via les classes) et en version à mutabilité non précisée (via les interfaces), le tout sans trainer de « bagage » inutile.

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide de classes scellées

Dans la solution précédente, nous avons perdu le sous-typage entre les types immuables.

Cela peut encore être amélioré : en rendant privées les implémementations immuables et en scellant tous les types publics de cette hiérarchie.

```
public abstract class Nombre {
    private Nombre() {} // Scellage !
    // hiérarchie publique
    public static abstract class Rationnel extends Nombre { private Rationnel() {} }
    public static abstract class Entier extends Rationnel { private Entier() {} }
    // implémentations immuables
    private static final class RationnelImpl extends Rationnel {
        final int numerateur, denominateur;
        RationnelImpl(int p, int q) { numerateur = p; denominateur = q; }
    }
    private static final class EntierImpl extends Entier {
        final int valeur;
        EntierImpl(int valeur) { this.valeur = valeur; }
    }
    // fabriques statiques
    public static Rationnel rationnel(int p, int q) { return new RationnelImpl(p, q); }
    public static Entier entier(int n) { return new EntierImpl(n); }
} // il faudra bien sûr ajouter getters et opérations arithmétiques dans toutes les classes
```

Les types publics ont alors la bonne relation de sous-typage et leur scellage garantit leur immuabilité.

Objectif : ne pas laisser l'héritage casser l'encapsulation

Exemple problématique


```java
/**
*   ReelPositif représente un réel positif modifiable.
*   Contrat : getValeur et racine retournent toujours un réel positif.
*/
class ReelPositif {
    double valeur;
    public ReelPositif(double valeur) { setValeur(valeur) ; }
    public getValeur() { return valeur; } // on veut retour >= 0
    public void setValeur(double valeur) {
        if (valeur < 0) throw new IllegalArgumentException(); //crash
        this.valeur = valeur;
    }
    public double racine() { return Math.sqrt(valeur); }
}

class ReelPositifArrondi extends ReelPositif{
    public ReelPositifArrondi(double valeur) { super(valeur) ; }
    public void setValeur(double valeur) { this.valeur = Math.floor(valeur); }
}

public class Test {
    public static void main(String[] args) {
        ReelPositif x = new ReelArrondi(— Math.PI);
        System.out.println(x.racine()); // ouch! (affiche "NaN")
    }
}
```

Objectif : ne pas laisser l'héritage casser l'encapsulation

Solutions (1)

1. L'évidente : rendre `valeur` privé pour forcer l'accès via `getValeur` et `setValeur`.
   Point faible : ne résiste toujours pas à certaines extensions

```
class ReelPositifArrondi extends ReelPositif{
    double valeur2; // et hop, on remplace l'attribut de la superclasse
    public ReelPositifArrondi(double valeur) { this(valeur); }
    public void getValeur() { return valeur2; }
    public void setValeur(double valeur) { this.valeur2 = Math.floor(valeur); }
}
```

   Ici, `racine` peut toujours retourner `NaN`. Pourquoi ?

2. La solution la plus précise : rendre `valeur` privé **et** et passer `getValeur` en `final`. Cette solution garantit que le contrat sera respecté par toute classe dérivée.
   Point fort : on restreint le strict nécessaire pour assurer le contrat.
   Point faible : il faut réfléchir, sinon on a vite fait de manquer une faille.

Objectif : ne pas laisser l'héritage casser l'encapsulation

3. La sûre, simple mais rigide : `valeur` → **`private`**, `ReelPositif` → **`final`**.

   Point fort : sans faille et très facile

   Point faible : on ne peut pas créer de sous-classe `ReelPositifArrondi`, mais on peut contourner grâce à la composition (on perd le sous-typage) :

```
class ReelPositifArrondi {
    private ReelPositif valeur;
    public ReelPositifArrondi(double valeur) { this.valeur = new
        ReelPositif(Math.floor(valeur)); }
    public void getValeur() { return valeur.getValeur(); }
    public void setValeur(double valeur) {
        this.valeur.setValeur(Math.floor(valeur)); }
}
```

   Pour retrouver le polymorphisme : écrire une interface commune à implémenter (argument supplémentaire pour toujours programmer à l'interface).

   → on a alors mis en œuvre le patron de conception « **décorateur** » (GoF).

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (1)

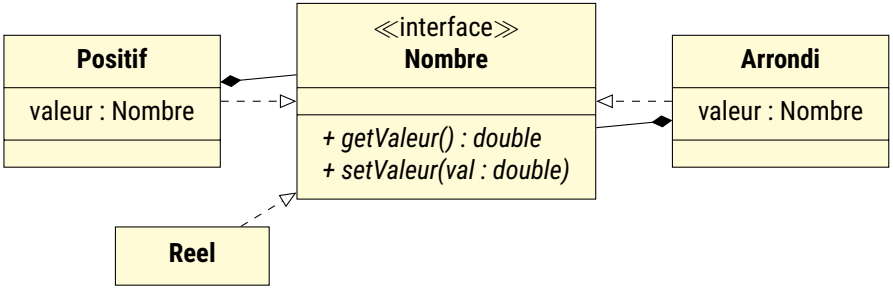

```java
interface Nombre {
    double getValeur();
    void setValeur(double valeur);
}

final class Reel implements Nombre {
    private double valeur;
    public Reel(double valeur) { this.valeur = valeur };
    @Override public double getValeur() { return valeur; }
    @Override public void setValeur(double valeur) { this.valeur = valeur; }
}

final class Arrondi implements Nombre {
    private final Nombre valeur;
    public Arrondi(Nombre valeur) { this.valeur = valeur; }
    @Override public double getValeur() { return Math.floor(valeur.getValeur()); }
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }
}

final class Positif implements Nombre {
    private final Nombre valeur;
    public Positif(Nombre valeur) { this.valeur = valeur; }
    @Override public double getValeur() { return Math.abs(valeur.getValeur()); }
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }
}
```

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (2)

Principe du patron décorateur : on implémente un type en utilisant/composant un objet qui est déjà instance de ce type, mais en lui ajoutant de nouvelles responsabilités.

L'intérêt : on peut décorer plusieurs fois un même objet avec des décorateurs différents.



Dans l'exemple, les décorateurs sont les classes `Positif` et `Arrondi`. Pour obtenir un réel positif arrondi, on écrit juste : `new Arrondi(new Positif(new Reel(42)))`. On n'a pas eu besoin de créer la classe `ReelPositifArrondi`.

Comment « hériter » de plusieurs classes ?

- Le patron décorateur permet, via la composition, d'ajouter/modifier plusieurs fois du comportement en réutilisant plusieurs clases existantes.
- Mais, ce patron est limité à créer des objets d'interface constante [1].
- Pour obtenir à la fois le bénéfice de la réutilisation d'implémentation et d'un type enrichi (plus de méthodes), il faut s'y prendre autrement.
- Le besoin décrit serait pourvu si la clause `extends` admettait plusieurs superclasses. Malheureusement, Java ne permet pas l'héritage multiple.
- À la place, il faut donc « bricoler » avec la composition et l'implémentation d'interfaces → **patron délégation** [2].

1. L'ajout de méthodes n'est pas une fonctionalité de ce patron de conception : en effet, seules les méthodes ajoutées par le dernier décorateur seront utilisables dans l'objet final.
2. Patron décrit et nommé par les auteurs du langage Kotlin, pas par le « Gang of Four », bien qu'il ressemble à d'autres patrons comme décorateur ou adaptateur.

Objectif : simuler un héritage multiple

Via les interfaces et la composition → patron « délégation »

Supposons que vous ayez 2 interfaces avec leurs implémentations respectives : [1]

```
interface AvecPropA { void setA(int newA); int getA(); }
interface AvecPropB { void setB(int newB); int getB(); }

class PossedePropA implements AvecPropA { int a; /* +methodes setA et getA... */ }
class PossedePropB implements AvecPropB { int b; /* +methodes setB et getB... */ }
```

On peut alors écrire une classe ayant les 2 propriétés de la façon suivante :

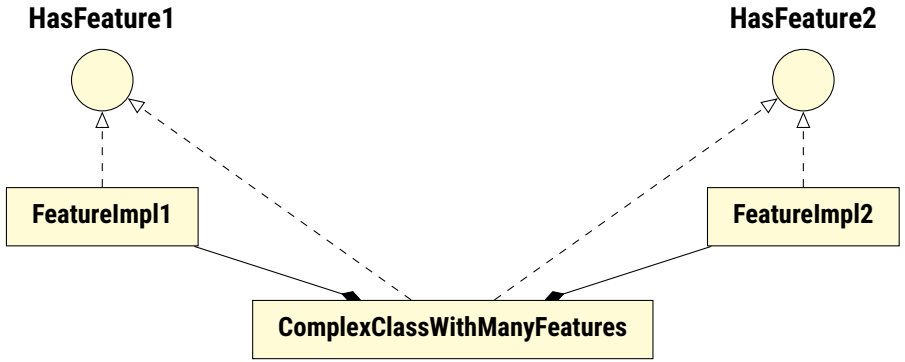
```
class PossedePropAetB implements AvecPropA, AvecPropB {
    PossedePropA aProxy; PossedePropB bProxy;
    void setA(int newA) { aProxy.setA(newA); }
    int getA() { return aProxy.getA(); }
    void setB(int newB) { bProxy.setB(newB); }
    int getB() { return bProxy.getB(); }
}
```

Si les classes auxiliaires sont fournies par un tiers et n'implémentent pas d'interface, on peut créer les interfaces manquantes et les implémenter dans `PossedePropAetB`. [2]

1. Supposées moins triviales que dans l'exemple (sinon c'est le marteau-pilon pour écraser une mouche !).
2. On revient au patron adaptateur déjà introduit dans ce cours.

Patron « délégation » en UML

Diagramme à 2 interfaces déléguées (mais ça pourrait être 1, 3 ou autant qu'on veut) :



Ce qu'on ne voit pas sur le diagramme : les implémentations dans `ComplexClassWithManyFeatures` des méthodes de `HasFeatureX` ne comportent qu'un simple appel vers la méthode de `FeatureImplX` de même nom.