L’héritage

NFA032

Présentation n°6

Présentation générale

- Motivations :
  - S'appuyer sur des classes existantes pour en définir de nouvelles :
    - sans modifier le code des classes existantes
    - en écrivant un minimum de code pour les nouvelles
  - Réutiliser le code existant en lui permettant de manipuler des objets de ces nouvelles classes
    - le code peut fonctionner avec des objets qui n'existent pas encore…
- Comment définir une nouvelle classe *S* :
  - Une classe *C* existante définit les propriétés de ses instances
    - Propriété (ou membre) = nom + type + valeur/code + protection + …
  - La classe *S* hérite de *C* :
    - Elle fait siennes les propriétés de *C*
      - [**héritage pur**] En les reprenant telles quelles
      - [**redéfinition**] En altérant une partie de leur définition (nom, type, valeur/code, …)
      - [**inhibition**] En signalant qu'elles ne sont plus définies à son niveau
    - Elle ajoute ses propres propriétés
      - [**enrichissement**]
- Comment réutiliser :
  - Là où un objet de classe *C* est attendu, un objet de classe *S* peut le remplacer
    - Principe du *polymorphisme des variables*

Présentation générale

- Si la classe *S* hérite de la classe *C* :
  - *S* est une **sous-classe** (ou *classe dérivée* / *fille*) de *C*
  - *C* est une **super-classe** (ou *classe de base* / *mère*) de *S*
  - La relation d'héritage s'entend généralement au sens large :
    - Si *S* hérite de *C*, c'est que :
      - [**héritage direct**] soit *S* hérite explicitement de *C*
      - [**héritage indirect**] soit *S* hérite explicitement de *T*, et *T* hérite de *C*
    - La relation est réflexive : toute classe *S* hérite d'elle-même (toujours implicite)
    - La relation est transitive : si *A hérite de B* et *B hérite de C*, alors *A hérite de C*
- Pour toute classe *C* :
  - sa **descendance** est l'ensemble des classes qui héritent de *C*, *C* incluse.
  - son **ascendance** est l'ensemble des classes dont *C* hérite, *C* incluse.

Présentation générale

- Le **graphe d'héritage**
  - C'est un *graphe orienté* dont :
    - Les sommets sont les classes
    - Un arc relie *S* vers *C* si *S* hérite de *C*
  - Il en existe une forme canonique : le **graphe de Hasse**
    - réflexivité : inutile de tracer les arcs *S* vers *S*
    - transitivité : si *A hérite de B* et *B hérite de C*, alors *A hérite de C* $\Rightarrow$ inutile de tracer l'arc *A* vers *C*
  - Propriétés :
    - Graphe sans circuit :
      - si *S* hérite de *C* et *C* hérite de *S*, c'est que *S*=*C*
    - Multiplicité de l'héritage :
      - **Héritage simple** : toute classe hérite au mieux (0 ou 1) d'une autre classe (graphe = forêt d'arbres)
      - **Héritage multiple** : au moins une classe hérite d'au moins deux autres classes

L'héritage en Java

- Il existe deux sortes de classes :
  - Les classes
  - Les interfaces
- La relation d'héritage se concrétise sous trois formes distinctes :
  - Une **relation d'extension** entre classes
    - Une classe *S* étend (`extends`) au mieux une autre classe *C* ⇒ héritage simple à ce niveau
  - Une **relation d'implémentation** de classe à interface
    - Une classe *S* implémente (`implements`) autant d'interfaces que nécessaire ⇒ héritage multiple à ce niveau
  - Une **relation d'extension** entre interfaces
    - Une interface *I* étend (`extends`) autant d'interfaces que nécessaire ⇒ héritage multiple à ce niveau

L'héritage en Java : polymorphisme des variables

- [**Interface**] Soit une définition (ou affectation) de variable :

```
I v = e
```

où :

  - `I` est une interface
  - `e` est une expression de type `T`
- Règle du polymorphisme des variables :
  - cette définition (ou affectation) est valide si :
    - `T` est une interface qui hérite de (étend) l'interface `I`
    - `T` est une classe qui hérite de (implémente) l'interface `I`

- [**Classe**] Soit une définition (ou affectation) de variable :

```
C v = e
```

où :

  - `C` est une classe
  - `e` est une expression de type `T`
- Règle du polymorphisme des variables :
  - cette définition (ou affectation) est valide si :
    - `T` est une classe qui hérite de (étend) la classe `C`

L'héritage en Java : liaison dynamique de code

- Soient :
  - `I` une interface, spécifiant une méthode `m(…)`
  - `C` une classe, définissant une méthode `m(…)`
  - `S` une classe héritant de `I` ou `C`
  - `v` une variable :
    - de type `I` ou `C`
    - référençant un objet de classe `S`
- Liaison dynamique de code
  - Un appel `v.m(…)` lance la méthode `m(…)` définie au niveau de `S`
    - soit le code de `m(…)` dans `S` est hérité
    - soit ce code est explicitement redéfini (`@Override`), auquel cas c'est ce dernier qui est considéré
  - En général :
    - Le code appelé à l'exécution ne peut être prédit par le compilateur
    - il est déterminé juste avant l'appel, d'où la notion de *liaison dynamique de code* (aussi appelée *liaison tardive*)

L'héritage en Java – cas des attributs

- Si une classe *A* hérite d'une classe *B* :
  - Pour les attributs (variables d'instance) :
    - tout attribut défini dans une classe est nécessairement nouveau
      - un attribut hérité ne peut pas être altéré
        - » nom, type, initialisation et protection inaltérables
    - Si un attribut nouvellement défini porte le même nom qu'un attribut hérité
      - sa définition **masque** celle héritée
      - les deux attributs coexistent
      - L'attribut hérité reste accessible via une conversion de type

```
class A {
   int x;
}

class B extends A {
   double x;
}
```

```
{
   A a = new A();
   a.x = 1;        // ici int (x de A)

   B b = new B();
   b.x = 2;        // ici double (x de B)
   ((A)b).x = 3; // ici int (x de A)

   A c = b;
   c.x = 4;        // ici int (x de A)
}
```

L'héritage en Java – cas des méthodes

- Si une classe *A* hérite d'une classe *B* :
  - Pour les méthodes (méthodes non statiques) :
    - Toutes les méthodes de *B* sont héritées
      - Attention : les constructeurs de *B* sont hérités, mais ne peuvent pas servir à construire des objets de classe *A*
        - » Il faut donc définir des constructeurs dans *A*
        - » Les constructeurs de *A* peuvent s'appuyer sur ceux de *B* : appel `super`
    - Des méthodes de *B* peuvent être redéfinies
      - Optionnel : écrire l'annotation `@Override` avant la définition
      - Pour la signature d'une méthode redéfinie :
        - » Reste identique dans *A* : il suffit de définir un autre corps
        - » Est **spécialisée** dans *A*, avec un nouveau corps
      - La définition (corps) de la méthode dans *B* reste appelable dans *A* : appel `super`.
    - Toute autre méthode est considérée comme nouvelle
- Que signifie *spécialiser* une signature de méthode :
  - Soit *S* un type qui hérite d'un type *C*
  - Règle de la **covariance** (adoptée en Java) :
    - Soit le type *C* de l'un (au moins) de ses paramètres est changé en *S*
    - Soit le type *C* de son résultat est changé en *S*
  - Il existe une autre règle, dite de **contravariance**, qui diffère en ceci :
    - Soit le type *C* de son résultat est changé en *R*, où *C* hérite de *R*

L'héritage en Java - appel **super**

- Usage de **super** dans un constructeur
  - soit *S* une classe disposant de plusieurs constructeurs
  - appel **this** : déjà étudié

```
class S … {
   S(…) {
      this(…); // appel d'un autre constructeur de S
   …
```

    - il n'y a au mieux qu'un seul appel **this** dans un constructeur
    - cet appel est nécessairement écrit comme 1ère instruction du corps
  - appel **super** : supposons que *S* hérite explicitement de *C*

```
class S extends C … {
   S(…) {
      super(…); // appel d'un constructeur de C
   …
```

    - il n'y a au mieux qu'un seul appel **super** dans un constructeur, qui remplace alors l'appel **this**
    - cet appel est nécessairement écrit comme 1ère instruction du corps
    - signification : l'objet en cours d'initialisation **this** :
      1. s'initialise en tant qu'objet de type *C* : appel **super**
      2. poursuit son initialisation en tant qu'objet de type *S* : instructions suivantes du corps

Un exemple élémentaire : les animaux

- Reprise de l'exemple des animaux (cf. cours précédent sur les *interfaces*), mais légèrement complété :
  - Tout animal porte un nom
  - Tout animal pousse un cri
  - Tout animal appartient à un genre
  - Tout mammifère est un animal
  - Tout chien est un mammifère dont le cri est *ouah ouah*
  - Tout chat est un mammifère dont le cri est *miaou*
  - Toute souris est un mammifère dont le cri est *couic*
  - Tout oiseau est un animal
  - Tout canari est un oiseau dont le cri est *cui cui*
  - *Rintintin* est un chien
  - *Tom* et *Sylvestre* sont des chats
  - *Jerry* est une souris
  - *Titi* est un canari
  - *Jolly Jumper* est un cheval

Un exemple élémentaire : les animaux

- Dans un premier temps, étudions juste le cri :

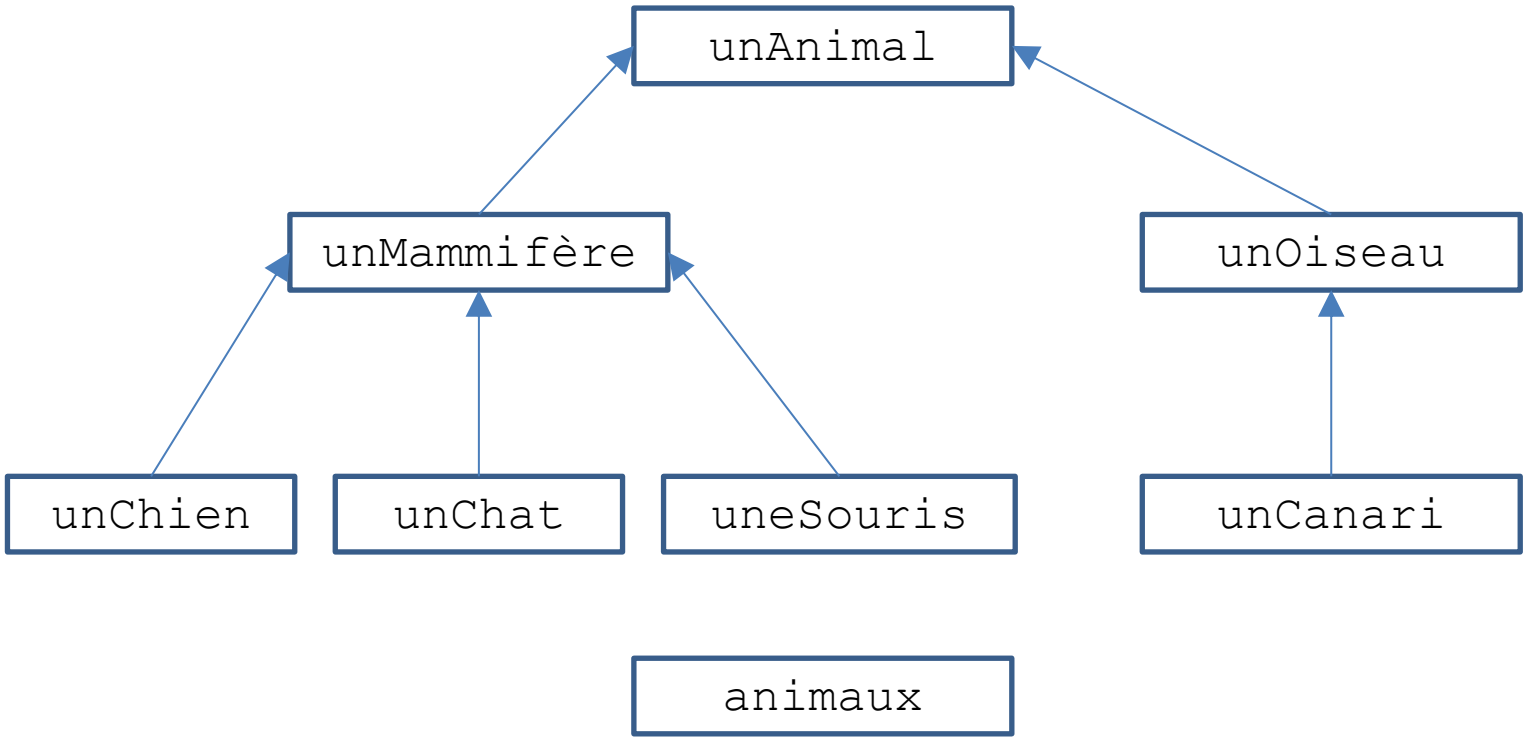
```
class unAnimal {
  public    String sonCri()   { return "un cri"; }
}

class unMammifere extends unAnimal {
}

class unChien extends unMammifere {
  @Override public String sonCri()   { return "ouah ouah"; }
}

class unChat extends unMammifere {
  @Override public String sonCri()   { return "miaou"; }
}

class uneSouris extends unMammifere {
  @Override public String sonCri()   { return "couic"; }
}

class unOiseau extends unAnimal {
}

class unCanari extends unOiseau {
  @Override public String sonCri()   { return "cui cui"; }
}
```

L'héritage en Java – cas des membres statiques

- Si une classe *S* hérite d'une classe *C* :
  - Tout *membre statique* (*variable de classe* ou *fonction* = méthode statique) introduit dans une classe est nécessairement nouveau :
    - membres statiques hérités :
      - Nom, type, initialisation et protection inaltérables
    - si une variable de classe nouvellement définie porte le même nom qu'une variable héritée :
      - sa définition **masque** celle héritée
      - les deux variables de classe coexistent
      - La variable de classe héritée reste accessible (il suffit d'expliciter la classe)
    - si une fonction nouvellement définie a une signature qui spécialise celle d'une fonction héritée :
      - sa définition **masque** celle héritée
      - les deux fonctions coexistent
      - La fonction héritée reste accessible (il suffit d'expliciter la classe)
  - Attention : pas de liaison dynamique en statique !
    - Un appel à une fonction *f* d'une classe *C* exécutera toujours le même code de *f*, quelles que soient les sous-classes de *C* (re)définissant *f*.

Un exemple élémentaire : les animaux

- Le code complet, cette fois :



Autre exemple : les figures planes

- Les données :
  - Une **figure** :
    - se caractérise par les coordonnées dans le plan de son **point de référence**
    - possède une **taille**, qui est un réel
  - Une **figure plane** :
    - est une figure
    - possède une **surface**, qui est un réel
    - sa taille est sa surface
  - Un **carré** :
    - est une figure plane dont le point de référence est son centre
    - se caractérise par la longueur de son **coté**
    - sa surface vaut son coté au carré

Un exemple : les figures planes

```
class uneFigure {
  double x, y; // abscisse et ordonnée du point de référence
  uneFigure(double x, double y) { this.x = x; this.y = y; }
  uneFigure() { this(0, 0); }
  double taille() { return -1; } // taille indéfinie à ce niveau
}

class uneFigurePlane extends uneFigure {
  uneFigurePlane(double x, double y) { super(x, y); }
  double surface() { return -2; } // surface indéfinie à ce niveau
  @Override double taille() { return this.surface(); }
}

class unCarré extends uneFigurePlane {
  double c; // coté
  unCarré(double x, double y, double c) { super(x, y); this.c = c; }
  @Override double surface() { return this.c * this.c; }
}
```

Légende :

- *nouvelle propriété ajoutée*
- *propriété héritée redéfinie*

Un exemple : les figures planes

- Bilan des propriétés définies par classe :

Classe	Attributs	Méthodes
uneFigure	double x double y	uneFigure(double,double) uneFigure() double taille()
uneFigurePlane	●double x ●double y	●uneFigure(double,double) ●uneFigure() uneFigurePlane(double,double) ●double taille() double surface()
unCarré	●double x ●double y double c	●uneFigure(double,double) ●uneFigure() ●uneFigurePlane(double,double) unCarré(double,double,double) ●double taille() ●double surface()

rien = *nouveau*

● = *hérité tel quel*

● = *hérité mais altéré*

● = *hérité tel quel mais non exploitable en tant que tel (constructeur)*

Un exemple : les figures planes

- La magie du *polymorphisme de variable* combiné à la *liaison dynamique de code* :
  - ce programme :

```
public static void main(String[] args) {
  uneFigure[] figures = {
    new uneFigure(1, 2),
    new uneFigurePlane(3, 4),
    new unCarré(5, 6, 7)
  };
  for (uneFigure figure : figures)
    Terminal.ecrireStringln("taille = "+figure.taille());
}
```

  - produit cette trace à l'exécution :

```
taille = -1.0
taille = -2.0
taille = 49.0
```

Un exemple : les figures planes

- Remarques :
  - Une notion n'a pas été traduite :
    - le point de référence d'un carré est son centre
  - Et si nous avions choisi :
    - de définir `uneFigure` comme une interface ?
    - idem pour `uneFigurePlane` ?
- Questions :
  - Si vous deviez définir la notion de point (classe `unPoint`), comment feriez-vous ?
  - Même question si vous deviez définir la notion de rectangle (classe `unRectangle`) ?

L'héritage en Java – appel **super**

- Usage de **super** dans une méthode
  - soit *C* une classe définissant une méthode `m(…)` ; cette méthode pourrait aussi tout simplement être héritée

```
class C … {
   … m(…) { … }
   …
```

  - soit *S* une sous-classe de *C* redéfinissant cette même méthode `m(…)`, avec éventuellement une spécialisation de sa signature

```
class S extends C … {
   @Override … m(…) { … }
   …
```

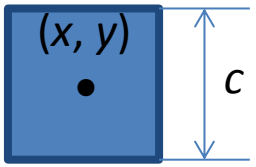
  - appel **super** dans le corps de `m(…)` dans *S* :

```
super.m(…); // appel du code de m défini au niveau de C
```

    - il n'y a aucune limite au nombre d'appels **super** à la méthode
    - il n'y a aucune contrainte quand à leurs places dans le corps
    - attention de ne pas remplacer **super** par **this** : il s'agit alors d'un appel récursif au code de `m` défini dans *S*
    - l'appel **super** ne peut figurer que dans un appel à une méthode de *C* que la méthode `m` de *S* spécialise
      - autrement dit, impossible d'écrire un appel **super**`.n(…)` si la méthode `m` de *S* n'est pas une redéfinition de la méthode `n(…)` de *C*

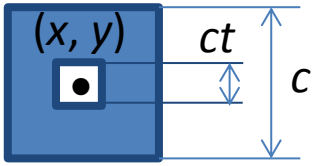
Un exemple : les figures planes

- On ajoute des carrés particuliers, qui ont un trou carré en leur centre :
  - Rappel de la définition de `unCarré`




```
class unCarré extends uneFigurePlane {
  double c; // coté
  unCarré(double x, double y, double c) { super(x, y); this.c = c; }
  @Override double surface() { return this.c * this.c; }
}
```

  - Définition de `unCarréAvecTrouCarré`




```
class unCarréAvecTrouCarré extends unCarré {
  double ct; // coté du trou
  unCarréAvecTrouCarré(double x, double y,
                       double c,
                       double ct) {
    super(x, y, c);
    if (ct>=c) throw new RuntimeException("coté du trou incorrect");
    this.ct = ct;
  }
  @Override double surface() {
    return super.surface() - this.ct * this.ct;
  }
}
```

Autre exemple : les figures planes

- Ajout d'une méthode `afficher` à nos figures :
  - Si *f* est un objet `uneFigure`, affiche sur le terminal

    ```
    Je suis une figure
    Mon point de référence est en (x, y)
    Ma taille vaut z
    ```

  - Si *f* est un objet `uneFigurePlane`, affiche sur le terminal

    ```
    Je suis une figure plane
    Mon point de référence est en (x, y)
    Ma surface vaut z
    ```

  - Si *f* est un objet `unCarré`, affiche sur le terminal

    ```
    Je suis un carré
    Mon centre est en (x, y)
    Ma surface vaut z
    Mon coté vaut c
    ```

Autre exemple : les figures planes

- L'analyse des différents messages montre leur structure commune :

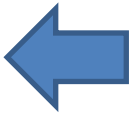
```
Je suis GENRE
REFERENCE est en (x, y)
TAILLE vaut z
COMPLEMENT
```

  - avec :

Fragment de texte	`uneFigure`	`uneFigurePlane`	`unCarré`
GENRE	une figure	une figure plane	un carré
REFERENCE	mon point de référence		mon centre
TAILLE	ma taille	ma surface	
COMPLEMENT			mon coté vaut c

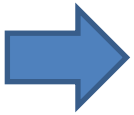
Autre exemple : les figures planes

```
public static void main(String[] args) {
  uneFigure[] figures = {
    new uneFigure(1, 2),
    new uneFigurePlane(3, 4),
    new unCarré(5, 6, 7),
    new unCarréAvecTrouCarré(8, 9, 10, 2)
  };
  for (uneFigure figure : figures) {
    figure.affiche();
    Terminal.sautDeLigne();
  }
}
```



Programme de test

Trace d'exécution



```
je suis une figure
mon point de référence est en (1.0, 2.0)
ma taille vaut -1.0

je suis une figure plane
mon point de référence est en (3.0, 4.0)
ma surface vaut -2.0

je suis un carré
mon centre est en (5.0, 6.0)
ma surface vaut 49.0
mon coté vaut 7.0

je suis un carré avec un trou carré
mon centre est en (8.0, 9.0)
ma surface vaut 96.0
mon coté vaut 10.0
mon trou a un coté de 2.0
```

Autre exemple : les figures planes

```
class uneFigure {
  ...
  void affiche() {
    Terminal.ecrireStringln("je suis "+this.GENRE());
    Terminal.ecrireStringln(this.REFERENCE()+" est en ("
                            +x+", "+y+")");
    Terminal.ecrireStringln(this.TAILLE()+" vaut "+this.taille());
  }
  String GENRE() { return "une figure"; }
  String REFERENCE() { return "mon point de référence"; }
  String TAILLE() { return "ma taille"; }
}

class uneFigurePlane extends uneFigure {
  ...
  @Override String GENRE() { return "une figure plane"; }
  @Override String TAILLE() { return "ma surface"; }
}
```

Autre exemple : les figures planes

```
class unCarré extends uneFigurePlane {
   ...
   @Override String GENRE() { return "un carré"; }
   @Override String REFERENCE() { return "mon centre"; }
   @Override void affiche() {
     super.affiche();
     Terminal.ecrireStringln("mon coté vaut "+this.c);
   }
}

class unCarréAvecTrouCarré extends unCarré {
   ...
   @Override String GENRE() { return "un carré avec un trou carré"; }
   @Override void affiche() {
     super.affiche();
     Terminal.ecrireStringln("mon trou a un coté de "+this.ct);
   }
}
```

Exercices

- Le carré à trou carré :
  - étudier le travail de la liaison dynamique lors de l'affichage des informations de cet objet
    - Indiquer dans quel ordre toutes les méthodes sont appelées, en précisant pour chacune à quelle classe est rattaché le code exécuté
- Exercice 1 : les classes `A` à `D`
  - Compréhension du rôle :
    - Du polymorphisme des variables
    - De la liaison dynamique

De l'usage du lien d'héritage

- Règle du polymorphisme de variables :
  - Si *V* est une variable de type *C*
  - Si *S* est un type qui hérite de *C*
  - Alors *V* peut référencer n'importer quel objet de type *S*
- Autrement dit :
  - Partout où un *C* est attendu, un *S* peut y prendre place
- Conclusion :
  - Le lien d'héritage est presque toujours employé pour concrétiser dans le code la relation abstraite de *généralisation / spécialisation*
    - Un mammifère est une forme particulière (*spécialisation*) d'animal
      - Linguistique : le mot *mammifère* est un hyponyme du mot *animal*
    - Un animal est une forme plus générale (*généralisation*) de mammifère
      - Linguistique : le mot *animal* est un hyperonyme du mot *mammifère*
  - Il n'est pas toujours facile de l'employer à bon escient

De l’usage du lien d’héritage

- Les exemples présentés font un usage adéquat du lien d’héritage
  - Exemple des animaux :
    - le lien d’héritage est exactement la concrétisation de la relation de généralisation / spécialisation.
  - Exemple des figures planes :
    - Idem, quoique :
      - Pourquoi un *carré* ne serait-il pas un *carré avec un trou carré* dont le trou aurait un coté nul ?

De l'usage du lien d'héritage

- Il y a parfois des usages malheureux (à éviter)
  - Exemple :
    - Un point du plan :
      - se caractérise par ses coordonnées cartésiennes *x* et *y*
      - peut être déplacé selon un déplacement Δ_x et Δ_y spécifié

(*x*, *y*)

(*x*, *y*)

    - Un rectangle dont les cotés sont parallèles aux axes :
      - se caractérise par sa hauteur et sa largeur
      - ainsi que par les coordonnées de son coin en bas à gauche
    - Quel choix de codage en Java de toutes ces notions ?

De l'usage du lien d'héritage

```
class unPoint {
  double x, y; // abscisse et ordonnée du point
  unPoint(double x, double y) {
    this.x = x; this.y = y; }
  void déplacer(double dx, double dy) {
    this.x += dx; this.y += dy; }
}

class unRectangle extends unPoint {
  double H, L; // hauteur, largeur
  unRectangle(double x, double y, double L, double H) {
    super(x, y); this.H = H; this.L = L; }
}
```

Héritage exploité pour récupérer les propriétés des points : le rectangle est défini entre-autre par son coin inférieur gauche.

Le bénéfice semble grand : il n'y a rien à écrire de plus pour le coin du rectangle…

Pourquoi est-ce mauvais ?

Le centre d'un cercle est un point.

Le polymorphisme des variables implique que le centre peut être une instance d'une classe héritant de point.

Le centre d'un cercle pourrait donc être un rectangle ! Absurde …

```
class unCercle {
  unPoint c; // centre
  double R; // rayon
  unCercle(double x, double y, double R) {
    this.c = new unPoint(x, y);
    this.R = R; }
}
```

Exercice 2

- Ouvriers et outils
  - Une première partie sans héritage
  - Une seconde avec
  - Imbrication de deux grandes relations :
    - La relation de généralisation / spécialisation
    - La relation de composition