La programmation orientée objet 3+ 2GIN

Java™

version 2022/2023

Sources

- *Introduction à Java, Robert Fisch, 2009*
- *Introduction à la programmation,* Robert Fisch, 11TG/T0IF, 2006
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006

Rédacteur

- Fred Faber

Site de référence

- **http://java.cnpi.lu**

Autres sites intéressants

- **http://java.developpez.com/**
- **http://www.javabuch.de**

Table des matières

1. Le temps et la date

1.1. Mesurer le temps écoulé

Parfois il est utile de mesurer le temps entre deux points précis d'un programme. Java procure un plusieurs possibilités, et nous allons en discuter les plus utiles.

`long System.nanoTime()`

retourne le temps actuel en nanosecondes (ns). Il est à noter que la précision n'est pas nécessairement donnée à une nanoseconde près, puisqu'elle dépend de l'horloge interne de l'ordinateur. Aucune garantie ne peut être donnée quant à la fréquence de changement de **`nanoTime()`**. La valeur de **`nanoTime()`** n'est pas donnée par rapport à une date ou un temps précis, elle indique le temps écoulé depuis le démarrage de la JVM. Elle peut seulement être employée pour calculer le temps écoulé (différences de deux appels de la méthode).

Utilisation :

```
long startTime ;
long stopTime ;
startTime = System.nanoTime() ;
//action à tester
…
stopTime = System.nanoTime() ;
timeLabel.setText((stopTime-startTime)/1000000000.0 + "s");
```

1.2. Les fonctions du calendrier

Les classes **LocalDate**, **LocalTime**, **LocalDateTime** du paquet **java.time** nous fournissent la date et le temps actuel. L'appel **LocalDateTime.now()** nous fournit une instance de la classe **LocalDateTime** initialisée au temps et à la date actuelle en considérant la zone temporaire de votre ordinateur.

Par la ligne suivante, nous obtenons une représentation préformatée de la date et du temps actuels :

```
    System.out.println(LocalDateTime.now());
```

P.ex : **2016-10-13T21:05:54.864**

Pour pouvoir formater ces informations individuellement, nous pouvons accéder à une instance de la classe **LocalDateTime** et lui demander les informations qui nous intéressent une à une. Pour cela, nous employons ici des méthodes prédéfinies de la classe **LocalDateTime**.

Exemple :

```
LocalDateTime now = LocalDateTime.now();
int year    = now.getYear();
int month   = now.getMonthValue();
int day     = now.getDayOfMonth();
int hour    = now.getHour();
int min     = now.getMinute();
int sec     = now.getSecond();
int nano    = now.getNano();
System.out.println("Date : "+day+"-"+month+"-"+year);
System.out.println("Time : "+hour+":"+min+":"+sec+" ("+nano/1000000+"ms)");
```

P.ex : **Date : 13-10-2016**
Time : 21:05:54 (864ms)

Remarques :

- Le paquet **java.time** qui a été introduit avec Java 8 contient une foule de classes et de méthodes pour travailler avec les dates et les temps. Celles-ci remplacent les anciennes classes **Calendar**, **Date**, **TimeZone** du paquet **java.util** qui sont dorénavant déconseillées. Une présentation de toutes les fonctionnalités de **java.time** remplirait des dizaines de pages. Nous allons nous limiter ici aux fonctions de base nommées ci-dessus et énumérer les classes les plus intéressantes :
  - Il existe des méthodes et des classes pour représenter les dates sous forme préformatée selon les formats usuels de différents pays (**DateTimeFormatter, FormatStyle, Locale**).

- Les classes **`ZoneDateTime`**, **`ZoneId`**, **`ZoneOffset`** permettent de travailler avec des dates et des temps de différents fuseaux horaires.

1.3. Le chronomètre

1.3.1. Fonctionnement

Un chronomètre est un objet qui exécute périodiquement une méthode donnée, c'est-à-dire qui exécute des actions à intervalles réguliers. Entre deux appels consécutifs à cette méthode s'écoule toujours un temps donné fixe.

Exemple

Voici l'axe du temps représentant l'exécution d'un chronomètre qui est déclenché chaque seconde :

Le trait orange représente le temps d'exécution de la méthode que le chronomètre exécute périodiquement.

1.3.2. La classe « Timer »

La classe **`Timer`** fait partie du paquet **`javax.swing`**. De ce fait, il faux indiquer en haut d'une classe utilisant un **`Timer`** la ligne d'importation correspondante :

```
import javax.swing.Timer;
```

La classe **`Timer`** dispose de plusieurs méthodes, dont voici celles qui nous intéressent le plus :

Méthodes	Description
`void start()`	Démarre le chronomètre.
`void stop()`	Arrête le chronomètre.
`boolean isRunning()`	Détermine si le chronomètre est actif ou non.
`void setDelay(int)`	Permet de modifier l'intervalle d'exécution du chronomètre. Le paramètre indique en temps en « millisecondes ».
`int getDelay()`	Permet de lire l'intervalle d'exécution du chronomètre. Le paramètre indique en temps en « millisecondes ».

1.4. Proposes d'implémentation

Il existe plusieurs méthodes ou « recettes », suivant lesquelles on peut mettre en œuvre un chronomètre. La méthode dite « par bouton caché » est la plus simple et la méthode préférée dans notre cours. La deuxième méthode est à considérer comme une information supplémentaire pour ceux qui désirent approfondir la matière.

1.4.1. Méthode dite « par bouton caché »

1. Ajoutez un bouton à votre fiche. Nous appelons ce bouton **`stepButton`** parce que chaque clic sur le bouton effectue un pas des opérations (EN : step). Plus tard, le chronomètre va exécuter ces pas automatiquement et périodiquement.
2. Développez la méthode de réaction du bouton **`stepButton`** et testez-la !

   (Pour tester, on peut cliquer régulièrement sur ce bouton, pour simuler l'action du chronomètre.)
3. Créez un nouveau chronomètre **`timer`** (l'instance **`timer`** peut être un attribut ou une variable du type **`Timer`**).
   Le premier paramètre du constructeur indique la **période** de la répétition **en millisecondes**. (Ici : périodicité 500ms → activation 2 fois par seconde).
   Le second paramètre indique au chronomètre d'exécuter la 1ère méthode de réaction liée au bouton **`stepButton`** :

```
timer = new Timer( 500 , stepButton.getActionListeners()[0] );
```

4. Démarrer le chronomètre :

```
timer.start();
```

5. Finalement on peut rendre invisible le bouton :

```
stepButton.setVisible(false);
```

Dans l'exemple donné, on crée donc un chronomètre qui exécute toutes les demi secondes le code attaché à la méthode de réaction du bouton **`stepButton`**.

1.4.2. Méthode par classe anonyme

Pour cette méthode, il n'y a pas besoin de créer un bouton. Par contre, le code à écrire pour la création d'un chronomètre est nettement plus complexe :

```
timer = new Timer( 500, new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        // code à exécuter périodiquement
    }
});
timer.start();
```

⇨ Tapez <Ctrl><Espace> après le mot **`ActionListener`** afin d'activer la complétion automatique de code ...

1.4.3. Méthode par expression lambda

Java 8 a introduit la possibilité de définir des "expressions lambda"[1]. Nous pouvons utiliser cette technique pour alléger davantage la définition d'une réaction au chronomètre :

```
timer = new Timer( 500, event -> {
        // code à exécuter périodiquement
});
timer.start();
```

[⇨ Si vous avez défini une réaction par classe anonyme, NetBeans vous propose automatiquement de transformer la définition en une expression lambda.]

1 Les expressions lambda ont été introduites en Java pour faire un pas vers la programmation fonctionnelle, ce qui permet p.ex. de passer des suites de traitements comme paramètres.

2. Les événements de la souris

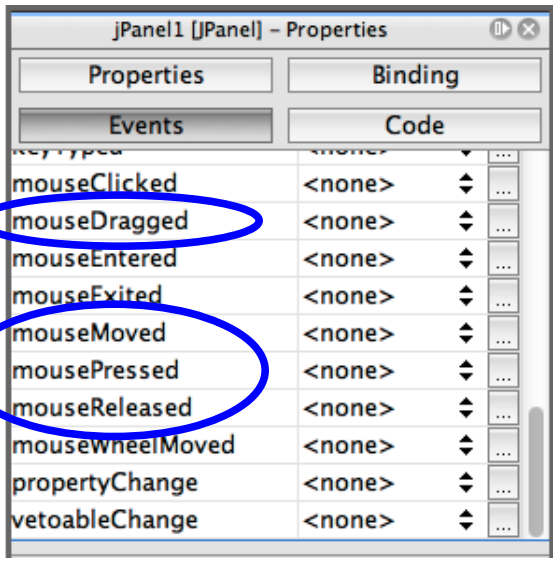
Par **événements souris** on entend le fait qu'un programme réagisse sur un clic de la souris respectivement sur son déplacement.

Le présent chapitre n'expliquera pas en détail le fonctionnement des événements souris mais se concentre uniquement sur leur utilisation tel que le propose l'éditeur NetBeans.

Les événements souris existent pour la plupart des composants qu'on peut ajouter sur une fenêtre, comme par exemple les boutons (**`JButton`**), les champs d'édition (**`JTextField`**) et les panneaux (**`JPanel`**). Tous les événements souris sont définis dans la classe **`MouseEvent`**.

2.1. *Types d'événements*

Dépendant du composant sélectionné, l'éditeur de propriété affiche un certain nombre d'événement relatifs à l'utilisation de la souris. Il s'agit de tous les événements préfixés avec le mot « mouse ». Malgré le fait qu'il existe plusieurs événements, le présent cours n'en traite que trois :



- **`mousePressed`**

  Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question et **enfonce** l'un des boutons de la souris.

- **`mouseReleased`**

  Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question et **relâche** l'un des boutons de la souris.

- **`mouseDragged`**

  Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question, a **enfoncé** un bouton de la souris **et déplace** celle-ci.

- **`mouseMoved`**

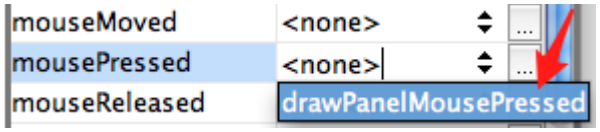
  Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question, **et déplace** la souris **sans avoir enfoncé** un bouton.

Pour avancés :

Pour pouvoir réagir aux événements **`mousePressed`** et **`mouseReleased`**, la classe doit implémenter l'interface **`MouseListener`**.
Pour pouvoir réagir aux événements **`mouseMoved`** et **`mouseDragged`**, la classe doit implémenter l'interface **`MouseMotionListener`**.

2.2. Méthodes de réaction

Afin d'attacher une méthode de réaction à un composant, sélectionnez le composant à l'aide de la souris, puis rendez vous dans l'onglet « Events » de l'éditeur des propriétés. Cliquez sur les petites flèches derrière l'événement, puis cliquez sur l'entrée présentée dans le menu contextuel du nom du composant suivi du nom de l'événement.



L'éditeur saute ensuite automatiquement dans le mode « Source » de NetBeans et place le curseur dans la nouvelle méthode de réaction qu'il vient de créer.

Pour le composant **`drawPanel`** et l'événement **`mousePressed`**, la méthode de réaction, telle que générée par NetBeans, est la suivante :

```
    private void drawPanelMousePressed(java.awt.event.MouseEvent evt) {
        // TODO add your handling code here:
    }
```

Afin de pouvoir réagir à un tel événement, il est essentiel de disposer de certaines informations relatives à la souris, comme par exemple :

- la position de la souris ou
- le bouton enfoncé ou relâché.

Ces informations sont passées à la méthode de réaction via le paramètre **`evt`** qui est du type **`MouseEvent`**. Ce paramètre est présent pour tous les trois types d'événements souris traités dans ce cours.

2.3. La classe « MouseEvent »

Voici les méthodes les plus importantes de la classe **`MouseEvent`**[2].

Méthodes	Description
`int getX()` **`int getY()`**	Retournent les coordonnées **`x`** ou **`y`** du point auquel se trouvait le curseur de la souris lorsque l'événement a été déclenché.
`Point getPoint()`	Retourne le point où se trouvait le curseur de la souris lorsque l'événement a été déclenché.
`int getButton()`	Indique quel bouton de la souris a été enfoncé ou relâché. Voici les valeurs les plus courantes : **`MouseEvent.BUTTON1`**, **`MouseEvent.BUTTON2`** et **`MouseEvent.BUTTON3`**.
`boolean isShiftDown()` **`boolean isAltDown()`** **`boolean isControlDown()`**	Permettent de détecter si au moment de l'événement les touches **Shift**, **Alt** ou **Control** du clavier étaient enfoncées.

2 https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/awt/event/MouseEvent.html

3. Les quatre principes de la POO

Maintenant que nous connaissons les techniques de base de la programmation orientée objet (POO) et de la programmation graphique en Java, il est temps de nous occuper plus en détail de l'organisation et de la structuration des éléments à l'intérieur des classes et des classes entre elles. Nous allons d'abord revenir sur quelques points que vous connaissez déjà pour les approfondir. Ensuite, nous allons ajouter des notions qui auront un effet massif sur la manière dont vous pouvez organiser votre application. En fait du point de vue technique, il n'y aura pas beaucoup de nouveautés à 'apprendre' (le plus grand bond en avant sera causé par un seul mot clé : *'extends'*), mais vous aurez la possibilité de voir la programmation à un niveau plus élevé, plus conceptionnel.

Résumons d'abord : la POO est basée sur quatre principes fondamentaux que nous allons éclairer dans ce chapitre :

1. Abstraction
2. Encapsulation
3. Héritage
4. Polymorphisme

3.1. *Abstraction*

L'abstraction est le fait d'ignorer les propriétés marginales et de se concentrer sur les aspects essentiels dans le cadre du projet à réaliser. C'est un principe essentiel pour pouvoir gérer la complexité du monde réel dans le domaine informatique.

En informatique, l'abstraction nous permet de ne considérer que les caractéristiques les plus importantes des objets que nous manipulons. Ce qui est important dépend du projet que nous sommes en train de réaliser. Lorsque nous devons réaliser une classe **`Personne`**, nous allons inclure des attributs pour le nom et le prénom des personnes, mais probablement pas le groupe sanguin, qui est alors victime de l'abstraction. Si par contre nous réalisions une application pour les patients d'une clinique, le groupe sanguin serait un attribut très important à inclure dans la classe.

3.2. *Encapsulation*

L'encapsulation est le fait de cacher la structure et le fonctionnement interne d'un objet et de ne rendre accessible que les éléments et les opérations absolument nécessaires à l'utilisation.

On sépare donc l'interface (d'utilisation) de la réalisation (l'implémentation). Nous avons déjà appliqué le principe de l'encapsulation lorsque nous avons protégé les attributs et donné l'accès uniquement par un accesseur. De la même façon, des méthodes et des attributs peuvent être cachés à l'intérieur de la classe sans qu'on en ait conscience à l'extérieur.

Le principe de l'encapsulation est le même dans d'autres domaines, p.ex. en électronique où on cache la complexité des appareils électroniques dans un boîtier en ne donnant accès aux fonctionnalités que par des boutons, des connecteurs normés ou des télécommandes. L'utilisateur n'a donc pas besoin de connaître toute la complexité de la **réalisation** des circuits électroniques et des connexions entre les différentes platines, mais il lui suffit de dominer ce que les constructeurs ont prévu comme **interface utilisateur**.

En informatique comme en électronique ou en mécanique, la conception de l'encapsulation et de l'interface utilisateur définit le confort d'utilisation et la sécurité des objets.

3.2.1. Avantages

L'encapsulation a plusieurs avantages :

- cacher la complexité d'un objet et faciliter son utilisation *(EN : information hiding)*
- protéger l'objet contre des opérations illicites de l'extérieur
- savoir modifier les interna d'un objet sans que l'utilisateur de l'objet ne doive changer son code

3.2.2. Niveaux d'accessibilité

En Java, l'encapsulation est réalisée par les niveaux d'accessibilité. Jusqu'ici, vous connaissez deux niveaux d'accessibilité (**`private`** et **`public`**) pour les classes, attributs et méthodes. En Java, il existe 4 niveaux différents d'accessibilité :

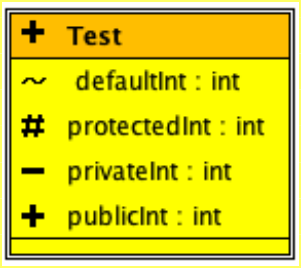
- **Accessibilité par défaut**
  Mot clé : (aucun)
  Symbole UML : **~**
  Si le niveau d'accessibilité d'un élément n'est pas spécifié, toutes les classes du même **paquet** ont accès à cet élément.
- **Accessibilité protégée**
  Mot clé : **`protected`**
  Symbole UML : **#**
  Si un élément est préfixé du mot clé **`protected`**, toutes les classes du même paquet ainsi que tous les **héritiers** directs (→ voir chapitre 3.4) ont accès à cet élément.
- **Accessibilité privée**
  Mot clé : **`private`**
  Symbole UML : **-**
  Un élément préfixé par le mot clé **`private`** est uniquement accessible depuis la classe elle-même. (Les instances d'une classe ont quand même accès aux éléments privés des autres instances de la même classe.)
- **Accessibilité publique**
  Mot clé : **`public`**
  Symbole UML : **+**
  Si un élément est marqué comme étant **`public`**, il n'existe aucune restriction par rapport à la visibilité de cet élément.

Exemple

Soient les deux classes suivantes qui se trouvent dans le même paquet :

```
public class Test
{
     int defaultInt = 0;
     protected int protectedInt = 1;
     private int privateInt = 2;
     public int publicInt = 3;
}


public class Launcher
{
     public static void main(String[] args)
     {
          Test test = new Test();
          test.defaultInt   = 10;         // OK, car même paquet
          test.protectedInt = 10;         // OK, car même paquet
          test.privateInt   = 10;         // !! ERROR !!
          test.publicInt    = 10;         // OK
     }
}
```



3.3. Agrégation

Dans la POO, il peut exister différentes relations entre les classes. Une relation que nous connaissons déjà sans lui avoir donné un nom est **l'agrégation**.

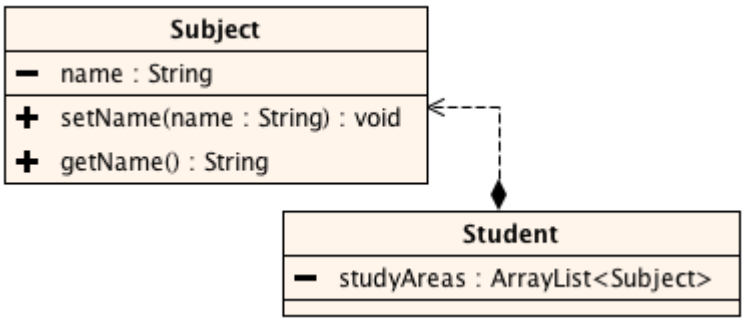
L'**agrégation** est une relation entre deux classes. Elle décrit le fait qu'une classe "**possède**" des instances d'une autre classe ou vice-versa que des instances d'une classe "**font partie**" des instances d'une autre classe (EN : **is-part-of**).

En effet, nous avons employé l'agrégation à chaque fois que nous avons intégré une instance d'un objet comme attribut dans une autre classe. On utiliserait p.ex. l'agrégation pour modéliser le fait que

- les élèves *font partie* d'une classe <=>
  une classe *possède* des élèves
- un (ou plusieurs) moteurs *font partie* d'un avion <=>
  un avion *possède* un (ou plusieurs) moteurs,
- ...

Exemple

Considérons les classes **`Student`** et **`Subject`** qui représentent un étudiant respectivement une branche.



On constate que :

1. Un étudiant (**`Student`**) **possède** une liste de domaines dans lesquels il réalise ses études.
2. La liste (**`ArrayList`**) de branches (**`Subject`**) est un attribut de la classe **`Student`**.
3. La classe **`Subject`** **est une partie** de la classe **`Student`** puisque la classe **`Student`** ne peut pas exister sans la classe **`Subject`**.

3.4. Héritage

L'héritage est une relation entre deux classes. **L'héritage** est la possibilité de définir une classe en ne formulant que les différences par rapport à une classe existante. Cette relation s'appelle une **relation de généralisation**, ou encore relation "**est-un**" (EN : "**is-a**").

Dans la POO il est possible qu'une classe **B hérite d'une classe A**, ce qui veut dire que la classe **B** possède tous les éléments la classe **A** (sans même écrire une ligne de code). Après, évidemment, on ajoute des éléments à la classe **B** ce qui rend la classe **B** plus **spécifique** que **A**. Ce concept est très fréquent dans la vie réelle.

Exemple : Prenons les termes 'personne', 'employé' et 'client'

*Un employé **est une** personne* et possède donc tout ce qui définit une personne. Dire qu'une personne est un employé est cependant plus spécifique et implique qu'elle possède des caractéristiques supplémentaires (employeur, salaire, …).

→ 'Personne' est donc un terme plus général,

→ 'Employé' est un terme plus spécifique et plus restrictif,

→ 'Employé' possède toutes les caractéristiques de 'Personne' et

→ 'Employé' possède des caractéristiques supplémentaires.

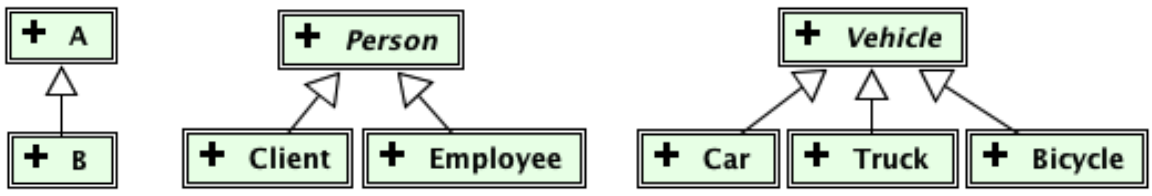
*Un client **est une** personne* et nous pouvons répéter les mêmes constatations que pour 'employé' et 'personne'.

Si nous devons modéliser un programme comprenant employés et clients, il est donc très utile de regrouper dans une classe **`Person`** tous les points communs entre clients et employés. Les classes **`Client`** et **`Employee`** seront définies sur base de **`Person`**. Ainsi, il suffira de définir seulement les caractéristiques spécifiques qui distinguent **`Employee`** et **`Client`** de **`Person`** => On évitera la duplication du code.

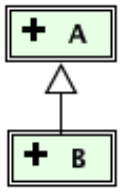
On dira donc que

`Employee` **est dérivé de** **`Person`**	ou que	**`Employee`** **hérite de** **`Person`**.
`Client` **est dérivé de** **`Person`**	ou que	**`Client`** **hérite de** **`Person`**.

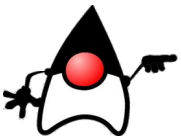
En UML, on représente cette relation de généralisation comme suit :



Pour exprimer cette relation d'héritage entre **A** et **B** on dit que :



- **A** est la **classe de base**
- **B** est une **spécialisation** de **A**
- **A** est la **généralisation** de **B**
- **B** est une **sous-classe** de **A**
- **A** est la **super-classe** de **B**
- **B** est la **classe fille** de **A**
- **A** est la **classe mère** de **B**



Remarques

- Pour autant qu'elle ne soit pas protégée, il est même possible d'étendre une classe dont on ne possède pas le code source !
- Avec les raccourcis <Alt-F12> en Windows et <Ctrl-F12> en Mac OSX, vous pouvez faire afficher la hiérarchie d'héritage de la classe actuelle.

3.4.1. Le mot clé "extends"

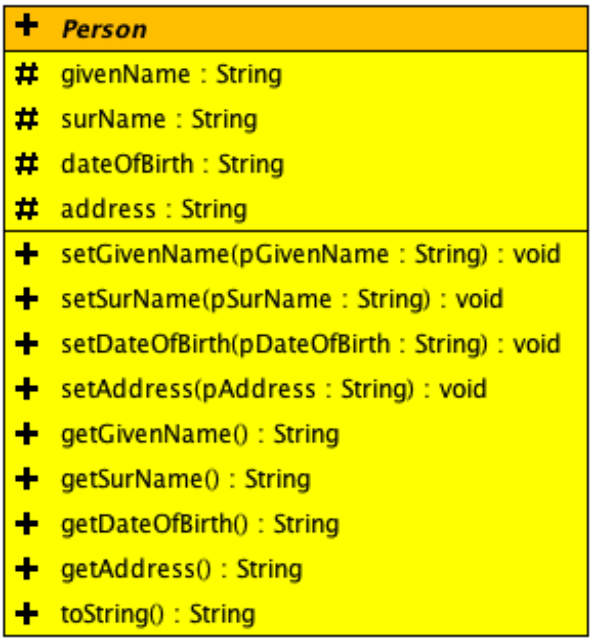
Pour exprimer l'héritage en Java, il suffit d'ajouter dans la définition de la sous-classe le mot clé **`extends`** suivi du nom de la classe de base :

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

Maintenant, B contient tous les éléments de A et nous pouvons ajouter les attributs et méthodes qui distinguent B de A.

3.4.2. Exercice résolu

Imaginons que nous devons écrire un programme pour une petite entreprise. Nous savons déjà que nous devrons implémenter des classes pour gérer les clients et les employés. Tous les deux possèdent des éléments communs qui nécessitent des attributs, avec des méthodes de gestion (nom, prénom, adresse, date de naissance, etc.). Au lieu de répéter les éléments dans les deux classes, nous rassemblons les éléments dans une **classe de généralisation** **`Person`**.



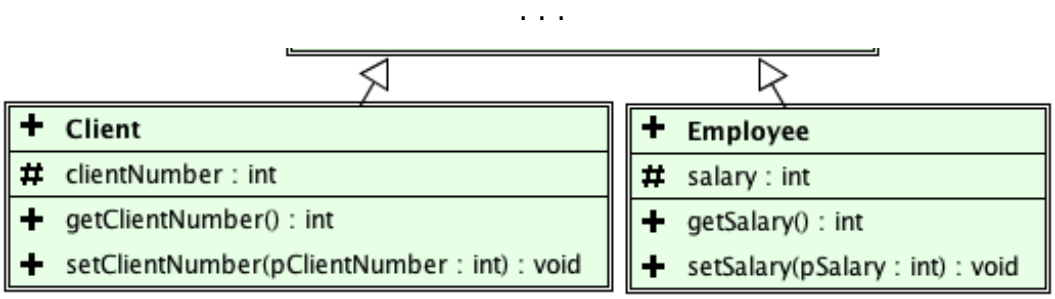
Nous créons ensuite la classe **`Client`** en ajoutant '**`extends Person`**' derrière la déclaration. (En Unimozer, vous pouvez alternativement employer l'assistant '**`Add Class...`**' et entrer '**`Person`**' dans le champ '**`extends`**').

La méthode **`toString`** est définie de façon à ce qu'elle retourne un texte de la forme :

```
<givenName> <surName> *<dateOfBirth> (<address>)
```

Si vous créez maintenant un nouveau **`Client`**, vous verrez qu'il dispose déjà des attributs et méthodes de **`Person`**. Ajoutez aussi une classe **`Employee`** pour les employés.

Ajoutons maintenant les éléments spécifiques des clients (no. carte client) et des employés (salaire) dans les sous-classes **`Client`** et **`Employee`**.

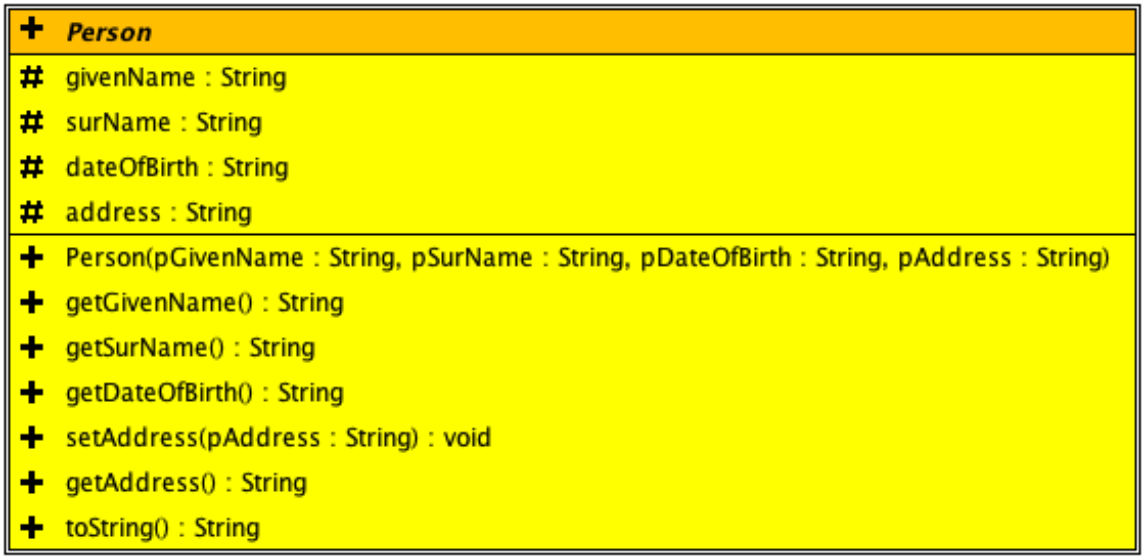


Si vous créez maintenant un nouveau **`Client`**, vous verrez qu'il dispose de l'attribut **`clientNumber`** en plus des attributs et méthodes de **`Person`**.

En vérifiant les trois classes, nous voyons que la réalisation est loin d'être parfaite dans l'esprit de **l'encapsulation**. En ce qui concerne l'**exactitude des données**, il est même possible de créer des clients sans nom, sans date de naissance et sans numéro de client. En plus, il est possible de modifier le nom ou la date de naissance des personnes plusieurs fois.

La meilleure solution est donc d'ajouter des constructeurs qui nous forcent d'initialiser les instances correctement, puis de supprimer la possibilité de modification des données persistantes (nom, prénom, date de naissance, numéro de client). L'adresse d'une personne ou le salaire d'un employé peuvent aussi changer plus tard.

Changeons d'abord la classe de base **`Person`** :



Avant de continuer, nous constatons plusieurs choses :

- La méthode **`toString`** héritée de **`Person`** n'est plus correcte (complète) pour **`Client`** et **`Employee`** (le numéro de client et le salaire manquent).
- Les constructeurs de **`Client`** et **`Employee`** différeront de celui de **`Person`**, mais ils auront une grande partie commune.
- En plus, le compilateur affiche deux messages d'erreur du genre '**… `cannot find symbol`**' aussitôt que nous avons ajouté un constructeur (avec des paramètres) à **`Person`**. Les messages d'erreur sont affichés au début des classes **`Client`** et **`Employee`** (même si nous n'avons rien changé dans ces classes...).

Commençons par nous occuper du premier "problème" ...

3.4.3. Redéfinition de méthodes

La redéfinition d'une méthode est le fait de définir dans une sous-classe une méthode qui existe déjà **sous le même nom** (et avec le même nombre et type de paramètres) dans une classe de base de la classe. Dans ce cas la méthode redéfinie de la classe de base ne sera plus disponible (directement) pour les instances de la sous-classe.

Attention :

- Les méthodes du même nom mais avec d'autres paramètres ne sont pas touchées par la redéfinition.
- Dans le code de la sous-classe nous avons toujours accès aux méthodes redéfinies, mais de façon indirecte : c.-à-d. nous pouvons toujours les appeler par "**super**" (→ voir aussi chapitre 3.4.4), même si elles ne sont plus disponibles directement dans les instances de la sous-classe.

Exercice résolu (suite) :

Redéfinissons la méthode **`toString`** pour notre classe **`Client`** :

```
public String toString()
{
  return givenName + " " + surName + " *" + dateOfBirth
                                  + " (" + address + ") no.:" + clientNumber;
}
```

Maintenant, un appel de **`toString`** d'un client va employer cette nouvelle méthode. La méthode originale de **`Person`** ne sera plus disponible aux instances de **`Client`**.

Adaptez aussi la méthode **`toString`** de **`Employee`**.

3.4.4. Accès aux éléments de la super-classe - *super*

Une classe peut utiliser les éléments hérités comme s'ils se trouvaient dans la classe elle même. Dans la classe **Client**, nous pouvons donc écrire directement :

```
System.out.println(givenName +" " surName);
setAddress("57, rue Bellevue L-3883 Manternach");
```

> Si par contre :
>
> - nous devons appeler une méthode de la super-classe qui existe **sous le même nom** dans la sous-classe, ou
> - si nous voulons appeler le **constructeur** de la super-classe,
>
> alors nous devons placer le mot-clé **super** devant le nom de la méthode.

Le mot clé **super** peut être considéré comme une référence à la super-classe.

Exercice résolu (suite) :

En redéfinissant la méthode **toString**, vous avez certainement remarqué que vous avez dû répéter du code qui se trouvait déjà dans la classe **Person**. On peut bien sûr copier le code par *copy-paste*, mais ce serait bien plus flexible de faire une référence à la méthode héritée au lieu de la copier.

Dans la classe **Client**, nous pouvons écrire très élégamment:

```
public String toString()
{
  return super.toString() + " no.:" + clientNumber;
}
```

Ceci sera encore plus important dans des méthodes plus complexes, où il faudrait à chaque modification recopier le code de la super-classe dans toutes les sous-classes...

3.4.5. Redéfinition du constructeur

ATTENTION : LES CONSTRUCTEURS NE SONT PAS HÉRITÉS !

Mais : Lors de sa construction, la sous-classe appelle automatiquement le constructeur par défaut de sa super-classe (s'il existe) !

Constructeur par défaut : Si nous ne définissons pas de constructeur pour une classe, alors la classe dispose d'un constructeur par défaut sans paramètres. Ce constructeur par défaut est hérité de la classe **`java.lang.Object`** qui est directement ou indirectement la super-classe de toutes les classes. Le constructeur par défaut construit une instance et initialise les attributs à leurs valeurs par défaut (0, null, false).

Dès que nous définissons notre propre constructeur, celui-ce remplace le constructeur par défaut. Si dans une super-classe nous définissons un constructeur avec des paramètres, il n'existe plus de constructeur sans paramètres (sauf, si nous nous définissons en plus un constructeur sans paramètres). La sous-classe ne trouve donc plus de constructeur par défaut qu'elle peut appeler et en conséquence elle produit une erreur du type '**… `cannot find symbol`**'.

Exercice résolu (suite) :

Enfin, nous pouvons nous expliquer les messages d'erreur dans notre projet et y et remédier : Les messages d'erreur dans **`Client`** et **`Employee`** s'expliquent par le fait que nous avons remplacé le constructeur par défaut (sans paramètres) de la super-classe **`Person`**. par un constructeur avec des paramètres.

Les constructeurs de **`Client`** et **`Employee`** doivent donc appeler le constructeur de **`Person`** explicitement. Pour ceci, nous aurons besoin de la référence **`super`**. L'appel au constructeur par défaut de la super-classe se ferait tout simplement par **`super()`**, mais comme le constructeur que nous voulons appeler a des paramètres, nous devons les indiquer entre parenthèses derrière **`super`**.

> **Le constructeur d'une sous-classe a typiquement deux charges :**
>
> 1. **appeler le constructeur de la super-classe,**
> 2. **initialiser ses propres attributs** (ici : `clientNumber`)**.**

Dans **`Client`**, nous définirons le constructeur donc comme suit :

```
public Client(String pGivenName, String pSurName, String pDateOfBirth,
                                        String pAddress, int pClientNumber)
{
    super(pGivenName, pSurName, pDateOfBirth, pAddress);   //(1.)
    clientNumber = pClientNumber;                          //(2.)
}
```

Définissez aussi le constructeur pour **`Employee`** et testez les deux classes.

3.4.6. Classes abstraites

Une **classe abstraite** est une classe qui n'est pas utilisée pour en créer des instances, mais uniquement comme classe de généralisation pour pouvoir en dériver des classes plus spécialisées. En UML le nom de la classe abstraite est inscrit en italique.

Une classe abstraite est le plus souvent employée pour regrouper les éléments (méthodes et attributs) communs d'un ensemble de classes qui seront spécialisées par la suite.

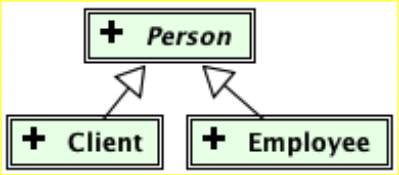
Par opposition, une **classe concrète** est une classe qui est utilisée pour en créer des instances.

En Java, les classes abstraites sont précédées du mot clé **abstract** pour que le compilateur les reconnaisse comme telles et puisse émettre une erreur de compilation lorsqu'on essaie d'en instancier un objet.

Exercice résolu (suite) :

Lors de la modélisation de notre petite entreprise, il est clair que la classe **Person** doit être réalisée comme classe abstraite puisqu'elle sert seulement comme 'dénominateur commun' aux classes **Client** et **Employee**, sans jamais être utilisée pour en dériver des instances.

```
public abstract class Person
{
    . . .
}
```



Dans la représentation UML, la classe **Person** apparaît alors en italique et un appel du genre **new Person(...)** va provoquer une erreur de compilation. (Évidemment les appels de **new Client(...)** ou **new Employee(...)** seront toujours possibles).

3.5. Polymorphisme

Le **polymorphisme** est la possibilité de considérer une instance d'une classe comme étant aussi une instance des classes de base de cette classe.

Dans notre exemple, il est possible de traiter une instance de **Employee** comme étant aussi une instance de **Person** ou même de **Object**. Ceci est extrêmement pratique parce qu'on peut envoyer une instance de **Employee** à une méthode désirant comme paramètre une instance de **Object** ou **Person**. Ainsi on peut définir des méthodes générales qui fonctionnent pour tout un sous-arbre de la hiérarchie des classes.

De même on peut définir un attribut ou une variable d'un type de base (p.ex. **Object** ou **Person**) et on peut lui affecter des instances d'une de ces sous-classes (directes ou indirectes).

```
Person p = new Employee("James","Black","12/3/1960","2323 Downtown NY",5800);
Object o = new Employee("James","Black","12/3/1960","2323 Downtown NY",5800);
```

Le polymorphisme fonctionne uniquement dans une seule direction, c.-à-d. une instance d'une classe ne peut pas être considérée comme instance d'une de ses sous-classes. L'affectation suivante est donc incorrecte :

```
Employee e = new Person("James","Black","12/3/1960","2323 Downtown NY"); //FAUX !!
```

Compatibilité des affectations :

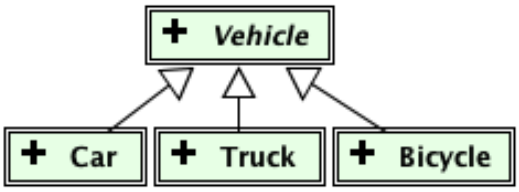
Une classe est compatible avec ses classes ancêtres.

Remarque : Polymorphisme et la 'vie réelle'

Cette façon de traiter l'héritage correspond bien à notre conception des choses de la vie réelle : Effectivement, un employé est une personne et peut être traité comme telle, mais inversement on ne peut pas dire que toute personne est un employé, et la traiter comme tel …

Exercice :

Soient les classes **`Vehicle, Car, Truck, Bicycle`** définies comme suit :



Que pouvez-vous dire de la classe **`Vehicle`** ?

Soient les déclarations suivantes :

```
Car     myCar;
Truck   myTruck;
Bicycle myBicycle;
Vehicle myVehicle;
```

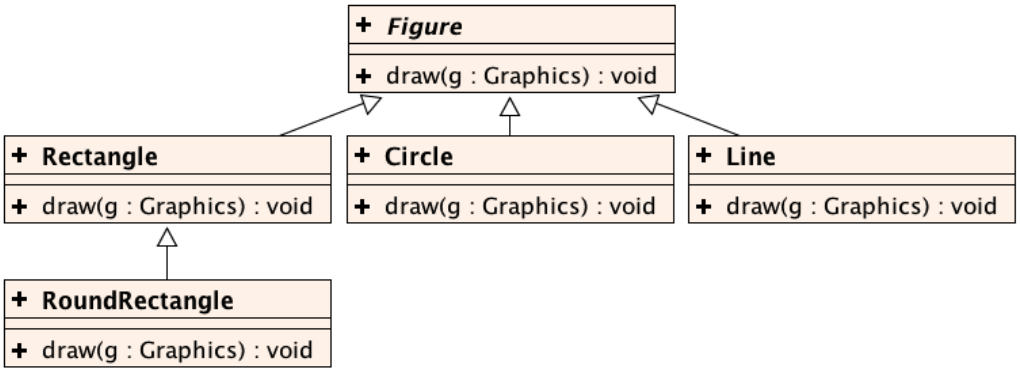
Lesquelles des affectations suivantes sont compatibles ?

Lesquelles sont incompatibles, ou incorrectes pour d'autres raisons ?

```
1. Truck t = myVehicle;
2. myVehicle = myBicycle;
3. Vehicle v = myTruck;
4. myCar = new Vehicle();
5. Car t = new Truck();
6. Vehicle v = new Car();
```

3.5.1. Polymorphisme et redéfinition de méthodes *(late binding)*

Le concept du polymorphisme devient particulièrement intéressant si on considère en plus qu'il est possible de redéfinir des méthodes qui ont déjà été définies dans des classes de base (→ voir chapitre 3.4.3).



Pour illustrer ceci, considérons l'exemple suivant qui est issu d'un programme de dessin :

```
public abstract class Figure
{    //définition des méthodes et attributs communs à toutes les figures
     . . .
     public void draw(Graphics g)
     {    // ? ? ?
     }
}
```

```
public class Rectangle extends Figure
{
     public void draw(Graphics g)
     {    //instructions pour dessiner un rectangle
          . . .
     }
}
```

```
public class Circle extends Figure
{
     public void draw(Graphics g)
     {    //instructions pour dessiner un cercle
          . . .
     }
}
```

```
public class Line extends Figure
{
     public void draw(Graphics g)
     {    //instructions pour dessiner une ligne
          . . .
     }
}
```

```
public class RoundRectangle extends Rectangle
{
     public void draw(Graphics g)
     {    //instructions pour dessiner un rectangle aux coins arrondis
          . . .
     }
```

```
}
```

Nous voyons que chaque classe dérivée de **`Figure`** a redéfini la méthode héritée **`draw`** pour que les objets soient dessinés de façon individuelle et correcte. Il faut cependant se poser la question ce qui se passe si, en appliquant le polymorphisme, on procède comme suit :

```
Figure f1 = new Rectangle();
Figure f2 = new RoundRectangle();
f1.draw(g);
f2.draw(g);
```

`f1` et **`f2`** sont du type **`Figure`**, mais ils font référence à des instances du type **`Rectangle`** et **`RoundRectangle`**.

Quelle méthode **`draw`** sera alors appelée, celle de **`Figure`**, celle de **`Rectangle`** ou celle de **`RoundRectangle`** ?

Java suit le principe le plus confortable (pour nous) en appelant automatiquement la méthode qui correspond à la nature de l'instance (et non celle de la référence).

Donc notre appel de **`f1.draw(g)`** produira effectivement un rectangle à l'écran et l'appel de **`f2.draw(g)`** produira un rectangle aux coins arrondis.

Si maintenant au lieu de deux figures **`f1`** et **`f2`**, vous vous imaginez toute une liste (*array*, **`Vector`** ou **`ArrayList`**) de figures, qui est remplie en désordre de rectangles, cercles, lignes etc, vous saurez apprécier l'utilité de cette façon de procéder... !

Remarque pour avancés : '***late binding' vs. 'early binding'***

Cette façon de lier automatiquement une méthode redéfinie au type réel de l'instance, nécessite une technique connue sous le nom de ***late binding*** ou aussi ***dynamic binding***.

Expliquons ceci en reprenant l'exemple d'une liste **`ArrayList<Figures> figureList`** remplie de toutes sortes de figures : En effet, lors de la compilation, on ne peut pas encore savoir quelle méthode doit être liée à l'appel de **`figureList.get(i).draw()`** (celle de **`Rectangle`**, celle de **`RoundRectangle`**, celle de **`Circle`**, …). Cette liaison de l'appel de **`draw`** au code de la 'bonne' méthode (celle correspondant au type réel de l'instance) a lieu seulement lors de l'exécution du programme – donc très tard et de façon dynamique. En principe, Java emploie le principe du *late binding* pour tout appel d'une méthode d'instance.

Lorsque le compilateur sait résoudre les adresses lors de la compilation (p.ex : les adresses des attributs ou des méthodes statiques), on appelle cela ***early binding*** ou aussi ***static binding***.

3.5.2. Méthodes abstraites

En regardant de près l'exemple du chapitre 3.5.1, nous remarquons une incohérence dans la définition de la méthode **`Figure.draw`**. En effet, quelle est l'utilité du bloc d'instructions de **`Figure.draw`** ? Cette méthode est redéfinie complètement dans chaque sous-classe et il n'y a pas une ligne de code utile que l'on pourrait écrire dans son bloc d'instructions. Cependant on ne peut pas la supprimer, sinon le mécanisme de la redéfinition polymorphe de la méthode (décrit dans le chapitre précédent) ne fonctionnerait pas.

En Java, une telle méthode est définie comme méthode abstraite.

Une **méthode abstraite** est une méthode qui n'est pas implémentée, mais qui est déclarée uniquement pour pouvoir être surchargée dans les classes dérivées.

abstract : mot clé marquant une méthode dont l'implémentation est reportée à ses classes descendantes.

Attention :

- Une méthode abstraite n'a pas de bloc d'instructions, mais sa déclaration est suivie d'un point-virgule.
- Une méthode abstraite peut seulement se trouver dans une classe abstraite.
- Chaque classe descendante doit nécessairement redéfinir la méthode abstraite.

Dans notre exemple :

```
public abstract class Figure
{
     //définition des méthodes et attributs communs à toutes les figures
     . . .

     public abstract void draw(Graphics g);
}
```

3.6. L'opérateur « instanceof »

Cet opérateur permet de tester si un objet est une instance d'une classe donnée.

Exemple

Prenons le code suivant :

!!
```
Animal myAnimal = new Cat();

if(myAnimal instanceof Animal)
{
    System.out.println("It is an animal ...");
}
if(myAnimal instanceof Cat)
{
    System.out.println("It is a cat ...");
}
if(myAnimal instanceof Dog)
{
    System.out.println("It is a dog ...");
}
```

Après exécution, le code affiche que **myAnimal** est un animal et que **myAnimal** est un chat.

4. Modificateurs *static* et *final*

4.1. Les éléments statiques - static

En Java, on dit qu'un élément (attribut ou méthode) est **statique**, s'il est partagé par toutes les instances d'une même classe. Cet élément existe alors au niveau de la classe et il est utilisable même sans qu'il n'y ait des instances de la classe. On n'a donc pas besoin de créer des objets avant de pouvoir accéder à des méthodes ou attributs statiques.

En conséquence, un élément statique (niveau de la classe) ne peut pas accéder directement à des éléments non statiques (niveau des instances) de sa propre classe.

Cependant, chaque instance peut accéder aux attributs et aux méthodes statiques.

En UML, les éléments statiques sont soulignés.

On dit aussi **variables de classe** ou **méthodes de classe** pour les **attributs statiques** ou les **méthodes statiques**.

Exemple 1 : attribut statique

Nous voulons compter le nombre d'instances qui ont été créées d'une classe. Il est évident qu'un tel compteur ne peut pas faire partie de chaque instance de la classe, mais qu'il doit se situer au niveau de la classe elle même. Nous créons donc un variable statique :

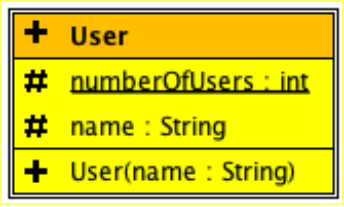
```
protected static int numberOfUsers = 0;
```

Cette variable existe une seule fois pour toute la classe (même s'il n'existe pas encore d'instances) et elle aura la même valeur pour toutes les instances de la classe.

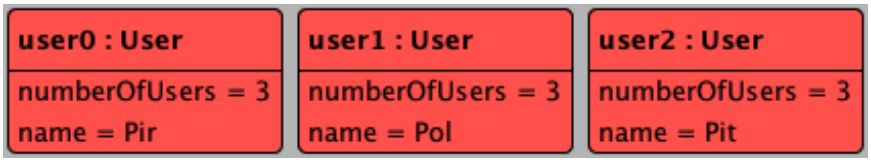
Chaque instance de la classe peut changer la valeur de **`numberOfUsers`**. Pour compter les instances, on peut ajouter une ligne au constructeur de la classe. Le code se présenta alors comme suit :

```
public class User
{
     protected static int numberOfUsers = 0; //existe 1 fois pour la classe
     protected String name; //non statique: existe 1 fois dans chaque objet

     public User(String name)
     {
          this.name = name;
          numberOfUsers = numberOfUsers + 1;
     }
}
```



Après avoir créé trois instances avec les noms Pir, Pol, Pit, les instances peuvent se présenter comme suit :



En ajoutant un utilisateur, **`numberOfUsers`** est incrémenté pour tous les objets à la fois.

Exemple 2 : méthode statique

Pour accéder à **numberOfUsers**, il faut écrire un accesseur. En principe, cet accesseur peut être statique ou non statique. Pour l'utilisateur de la classe **User**, il est quand même très utile de pouvoir demander à la classe combien d'utilisateurs existent. Si l'accesseur n'était pas statique, il faudrait connaître ou créer une instance pour pouvoir demander le nombre d'utilisateurs...

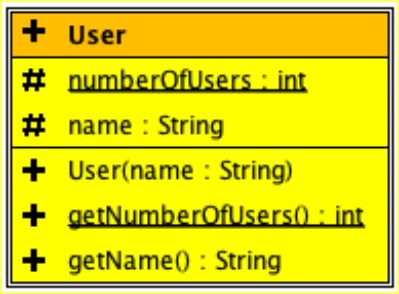
On décide donc de déclarer une méthode **getNumberOfUsers** comme étant statique (ce qui est d'ailleurs préférable pour tous les accesseurs d'attributs statiques) :

```
public class User
{
     protected static int numberOfUsers = 0; //existe 1 fois pour la classe
     protected String name; //non statique: existe 1 fois dans chaque objet

     public User(String name)
     {
          this.name = name;
          numberOfUsers = numberOfUsers + 1;
     }

     public static int getNumberOfUsers()
     {
          return numberOfUsers;
     }

     public String getName()
     {
          return name;
     }
}
```



On peut donc faire ceci:

```
public class Main
{
    public static void main(String[] args)
    {
      System.out.println( User.getNumberOfUsers()); //affiche 0
      User user0 = new User("Pir");
      User user1 = new User("Pol");
      User user2 = new User("Pit");
      System.out.println( User.getNumberOfUsers()); //affiche 3
     }
}
```

Notez que la **méthode** **getNumberOfUsers()** est directement appelée sur la **classe** **User** et non sur l'une des instances !

La méthode **main() :**

En regardant le dernier exemple de plus près, vous comprendrez enfin, pourquoi la méthode **`main()`** d'une application peut être démarrée sans avoir besoin de créer une instance de la classe ...

Remarquez aussi que la méthode statique **`getNumberOfUsers`** n'a pas d'accès direct à l'attribut **`name`** et ne peut pas appeler **`getName()`** sauf pour une instance dont elle reçoit éventuellement la référence. A l'intérieur de **`getNumberOfUsers`** les instructions suivantes provoquent donc une erreur avec les messages :

"non-static variable name *cannot be referenced from a static context"* *et*

"non-static method getName() *cannot be referenced from a static context"*

```
public static int getNumberOfUsers()
{
    name = "Hallo";                    // impossible !
    System.out.println(getName());     // impossible !
    return numberOfUsers;
}
```

Autres méthodes statiques connues :

- Toutes les méthodes de la classe **`Math`** (**`Math.sqrt(...)`**, **`Math.sin(...)`**, …) sont des méthodes statiques. En effet, nous n'avons jamais défini d'instance de cette classe et nous avons appelées ses méthodes en commençant par le nom de la **classe**. De telles classes sont rares puisqu'elles contiennent des méthodes d'utilité générale, non liées à un type d'objet spécifique.
- La méthode **`Calendar.getInstance()`** fournissant une instance du calendrier contenant la date actuelle est une méthode statique.
- La méthode **`JColorChooser.showDialog(...)`** ouvrant un dialogue de sélection de couleurs est une méthode statique (→ voir exercice F.2).

4.2. Les éléments constants - final

Un élément peut être préfixé du mot clé **final**, ce qui a comme effet qu'il ne peut plus être modifié par la suite. Il s'agit donc de quelque chose de constant.

4.2.1. Définition de constantes (attributs constants)

En Java, il n'existe pas de mot clé pour définir une constante, mais on peut quand même définir des constantes (en fait des attributs de classe constants) en combinant les modificateurs **static** et **final** :

final => la valeur ne peut plus être modifiée après l'initialisation

static => la constante n'existe qu'une seule fois et ceci au niveau de la classe

Les constantes de classe sont écrites entièrement en majuscules.

Les constantes de classe doivent être initialisées tout de suite lors de la déclaration.

Exemple :

```
public class Circle
{
     public static final double PI = 3.141592;
     protected static final double RADIUS = 100.4;
     static final double MAX_RADIUS = 500;
     //...
}
```

Attention:

Le modificateur **final** appliqué à un objet fixe la référence (l'adresse) de l'objet. En conséquence, on ne peut pas attribuer un nouvel objet à l'attribut, mais on peut changer le contenu de l'objet.

Exemple :

```
  final Point p = new Point(10, 20);
  p.x = 20;              //Permis, le contenu de l'objet p est changé
  p = new Point(20, 20); //ERREUR! essai de changer la référence de l'objet p
```

Evidemment, si un objet ne permet pas la modification de ses attributs (c.-à-d. si la classe est '***immutable***', comme p.ex. **String**), alors aucune modification de l'objet n'est possible.

Exemple :

```
  final String name = "John Doe"; //L'attribut name ne peut plus être changé
```

4.2.2. Pour avancés : Paramètres et variables locales constants

Plus rarement, le mot clé **`final`** est appliqué aux paramètres et aux variables locales pour éviter qu'ils soient changés après leur initialisation. Dans ce cas, il suffit d'indiquer le modificateur **`final`** devant la déclaration. Il n'est pas nécessaire d'initialiser ces éléments immédiatement lors de la déclaration, mais il ne peut y avoir qu'une seule affectation au cours de leur vie.

Pour les paramètres objets, **`final`** peut être utile pour éviter que le paramètre pointe sur un autre objet que celui reçu initialement.

4.2.3. Pour avancés : Classes non dérivables

Si le modificateur **`final`** est appliqué à une classe, alors on ne peut pas hériter de cette classe pour en dériver des sous-classes. Ce blocage de l'héritage peut avoir différentes raisons :

- la conception de la classe est telle qu'on n'a jamais besoin de la modifier,
- on veut garantir la sécurité de la classe,
- on veut optimiser la performance : on veut éviter que les méthodes soient redéfinies, ce qui évite aussi la recherche dynamique des méthodes (late binding).
- on veut travailler avec des objets 'immuables' pour éviter des précautions supplémentaires dans des applications multi-tâches.

Exemple : la classe **`String`** est définie comme **`final`** pour des raisons de performance.

4.2.4. Pour avancés : Méthodes non redéfinissables

Si le modificateur **`final`** est appliqué à une méthode, alors on ne peut pas redéfinir cette méthode dans les sous-classes. Ce blocage de l'héritage peut avoir différentes raisons :

- on veut garantir que la méthode fasse exactement ce qui est décrit dans cette méthode,
- on veut optimiser la performance : le compilateur sait produire un code plus efficace s'il n'y a pas la possibilité d'une redéfinition de la méthode. Il n'y aura pas de recherche dynamique pour la méthode *(late binding)*.

5. Gestion des paquets

Au début du cours de NetBeans, vous avez vu que les classes prédéfinies sont organisées dans une hiérarchie de paquets. Vous savez aussi comment importer les classes d'un paquet existant dans votre projet (à l'aide de l'instruction **`import`**).

Dans vos projets, vous avez certainement remarqué que le nombre de vos classes augmente rapidement avec la taille du projet. Jusqu'ici, nous avons défini toutes nos classes dans le paquet par défaut (sans nom) du projet, c.-à-d. les fichiers **`.java`** étaient sauvés directement dans le dossier **`src`** du projet. Cette façon de procéder est déconseillée pour des projets plus complexes. Il devient donc nécessaire d'organiser vos propres classes dans des paquets (p.ex. pour séparer les classes du modèle des classes de la vue et du contrôleur).

- Pour créer un nouveau **paquet**, il suffit de créer un nouveau dossier de ce nom dans l'arbre des paquets du dossier **`src`** du projet et d'y placer les fichiers contenant l'instruction **`package`** pour ce paquet.
- Pour placer une classe dans un **paquet**, il faut faire deux choses :
  - Placer le fichier « **`.java`** » dans le répertoire en question (c.-à-d. dans le dossier qui porte le nom du paquet) et
  - indiquer dans la classe le nom du paquet à l'aide du mot clé « **`package`** ». Cette instruction doit être la première instruction dans le fichier, elle doit précéder les instructions **`import`**.

Convention : Les noms des paquets sont écrits entièrement en minuscules.

Exemple

```
package lu.ecole.exemple;

public class Voiture
{
    // ...
}
```

En pratique, un grand nombre des opérations sont effectuées automatiquement :

En Unimozer, il suffit d'écrire l'instruction **`package`** avec le nom du (nouveau) paquet au début du fichier. Lors de la sauvegarde, les dossiers pour les paquets sont automatiquement créés s'ils n'existent pas encore et les fichiers **`.java`** y sont copiés automatiquement.

En NetBeans, on peut créer un nouveau paquet par un clic droit sur un paquet, puis copier les fichiers **`.java`** par 'drag & drop' d'un paquet vers un autre. NetBeans demande s'il doit adapter toutes les références.

Remarques :

- Il n'existe pas d'endroit central où on peut placer les paquets qu'on veut réutiliser dans plusieurs projets. Il faut recopier les paquets vers les dossiers **`src`** de chaque projet en question.
- On peut regrouper les fichiers compilés de paquets dans un fichier **`.jar`** (non exécutable). Ce paquet archivé peut alors être redistribué avec l'application dans le dossier **`dist/lib`**.
- Le compilateur Java n'ajoute pas automatiquement les paquets **`.jar`** précompilés aux dossiers **`dist/lib`** (à l'exception des librairies du JDK et des gestionnaires '*LayoutManagers*' du système NetBeans). Ceci doit être configuré p.ex. en NetBeans avec '**Libraries → Add Library...**'.

(→ Organiser le projet MiniDraw en paquets)

xx simple >Utilisation< d'interfaces

6. Annexe A

6.1. *Live Debugging*

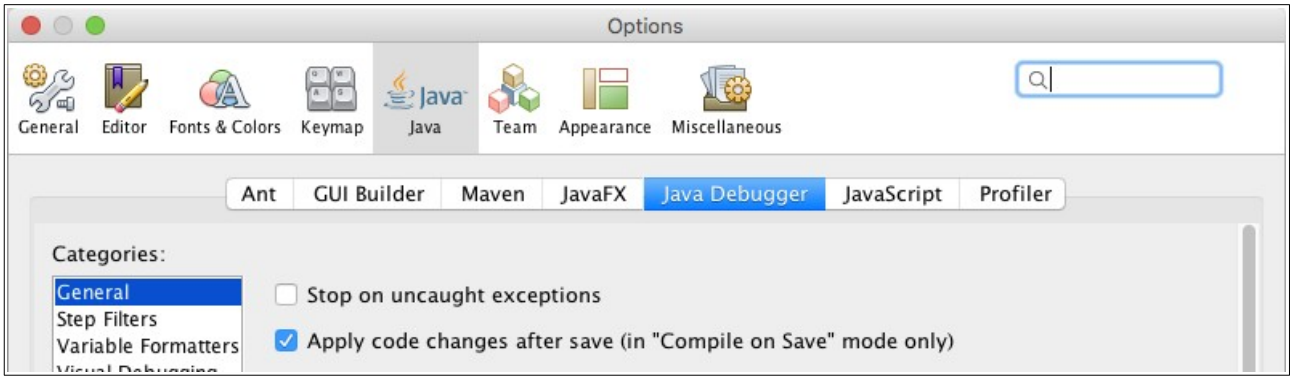
NetBeans offre la possibilité de modifier le code pendant une session de débogage et ainsi de tester des modifications sans devoir recompiler ou redémarrer le programme. Ceci peut être pratique pour tester de petites modifications dans un projet plus volumineux.

Pour ce faire, on peut procéder comme suit :

- démarrer le programme en mode de débogage  ,
- effectuer les modifications (il n'est pas possible d'ajouter ou de créer des méthodes),
- sauvegarder,
- cliquer '**Apply code changes**' dans le menu de débogage,
- si nécessaire, activer *repaint* (p.ex. en redimensionnant le formulaire).

On peut ajouter un peu de confort en automatisant l'étape '*Apply code changes*' comme suit :

- dans les préférences de NetBeans choisir **Java**,
- choisir l'onglet **Java Debugger**,
- activer : "**Apply code changes after save (in "Compile on Save" mode only)**".



Maintenant, il suffit de modifier le code et de pousser sur 'save' pour que les modifications soient appliquées.

[Par défaut dans tous les projets NetBeans, l'option '**Compile on Save**' est déjà activée, mais si vous voulez vérifier, elle se trouve dans les propriétés du projet sous **Build-Compiling**.]