La programmation orientée objet 3

Java™

Version 2023/2024

Sources

- *Introduction à la programmation orientée objet 2, CNPI-Gdt-PrograTG'11, 2012*
- *Introduction à la programmation orientée objet, CNPI-Gdt-PrograTG'11, 2011*
- *Introduction à Java, Robert Fisch, 2009*
- *Introduction à la programmation,* Robert Fisch, 11TG/T0IF, 2006
- *Introduction à la programmation,* Simone Beissel, T0IF, 2003
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006
- Wikipedia (Modèle-Vue-Contrôleur)

Rédacteurs

- Fred Faber
- Robert Fisch

Site de référence

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

Le cours est adapté et complété en permanence d'après les contributions des enseignants. Les enseignants sont priés d'envoyer leurs remarques et propositions via mail à Fred Faber ou Robert Fisch qui s'efforceront de les intégrer dans le cours dès que possible.

La version la plus actuelle est publiée sur le site : **http://java.cnpi.lu**

Table des matières

1. Les événements souris

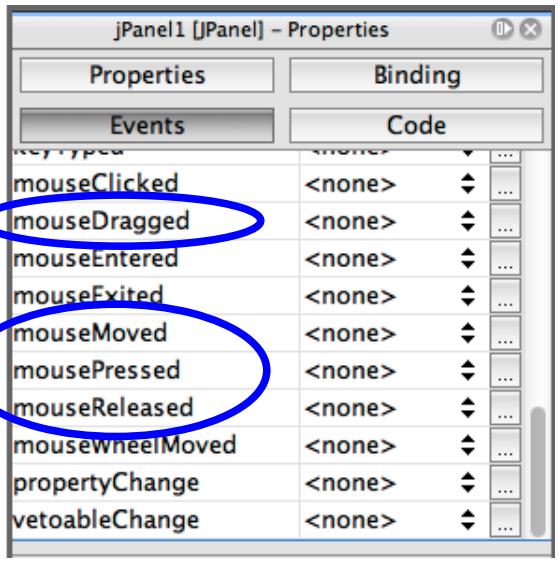
Par **événements souris** on entend le fait qu'un programme réagisse sur un clic de la souris respectivement sur son déplacement.

Le présent chapitre n'expliquera pas en détails 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`**).

1.1. 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 quatre :



- **`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`**

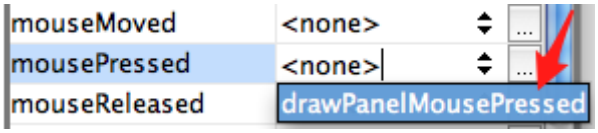
  Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question, a **enfoncé** un bouton de la souris **et déplace** celle-ci.

- **`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.

1.2. 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 quatre types d'événements souris traités dans ce cours.

Vu le modèle MVC, les événements souris sont à développer dans la classe **MainFrame**.

1.3. La classe « MouseEvent »

Voici les méthodes les plus importantes de la classe **MouseEvent**[1].

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** (gauche), **MouseEvent.BUTTON2** (centre) et **MouseEvent.BUTTON3** (droite). **Cette méthode ne fonctionne pas pour l'événement mouseDragged!**

1 http://docs.oracle.com/javase/6/docs/api/java/awt/event/MouseEvent.html

2. Le chronomètre

2.1. 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.

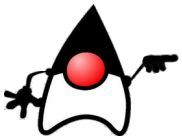
2.2. La classe « Timer »

La classe **`Timer`** fait partie du paquet **`javax.swing`**. De ce fait, il faut 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 le temps en « millisecondes ».



Attention

Il existe plusieurs classes portant le nom **`Timer`** qui possèdent des fonctionnements différents. Le présent cours traite uniquement celle contenue dans la paquet **`javax.swing`**. Il faut donc veiller à importer la bonne classe !

2.3. Propositions 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.

2.3.1. 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 500 ms → 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`**.

2.3.2. Méthode dite « par classe anonyme » - *pour avancés*

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 ...

2.3.3. Méthode par expression lambda - *pour avancés*

Java 8 a introduit la possibilité de définir des "expressions lambda"[2]. 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.]

2 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.

3. Agrégation

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

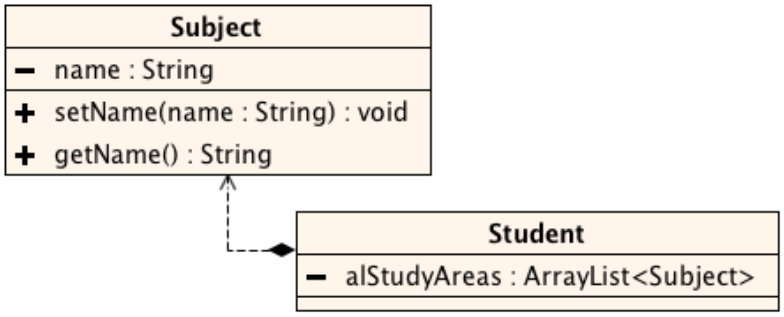
L'**agrégation** est une relation entre deux classes. Elle décrit le fait qu'une classe "**possède**" des instances d'une autre classe ou vice-versa que des instances d'une classe "**font partie**" des instances d'une autre classe (EN : **is-part-of**).

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 K ⇔
  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.



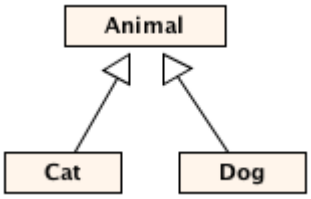
On constate que :

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`**.

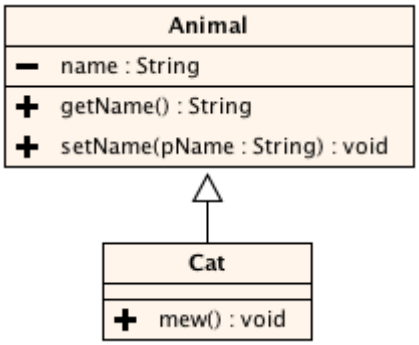
4. Héritage

En classe de 3GIG, la notion de « classe » a été introduite. Elle a été définie comme une description formelle d'une collection d'objets similaires, de même identité.

De façon similaire à ce qui existe dans le monde réel, une classe peut **hériter** d'une autre classe. Ainsi on peut avoir les classes **`Cat`** et **`Dog`** qui héritent toutes les deux de la classe **`Animal`**.



En informatique, il existe un concept assez similaire. La différence majeure réside dans le fait qu'une classe ne peut **hériter** d'une seule autre classe. Vu que le mot « classe » est féminin, on parle en français de la **classe mère** et de la **classe fille**. En anglais on les nomme simplement « parent class » et « child class ».

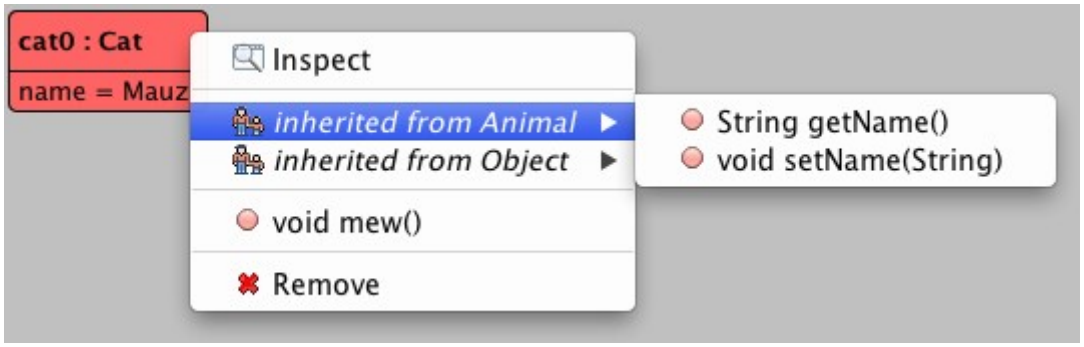


L'héritage est une relation entre deux classes. **L'héritage** est la possibilité de définir une classe en ne formulant que les différences par rapport à une classe existante. Cette relation s'appelle une **relation de généralisation**, **spécialisation** ou encore relation "**est-un**" (EN : "**is-a**").

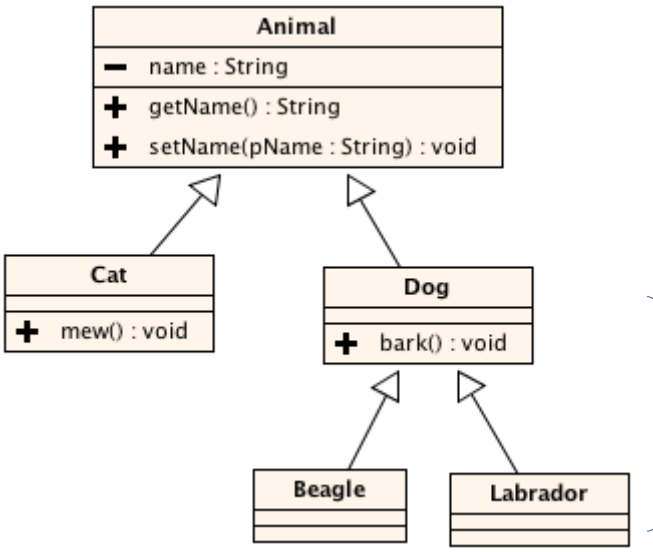
Vocabulaire & explications

- La classe **`Animal`** est la **classe mère** de la classe **`Cat`**.
- La classe **`Cat`** est la **classe fille** de la classe **`Animal`**.
- **`Cat`** est **l'héritier** de **`Animal`**.
- **`Cat`** est une **sous-classe** de **`Animal`**.
- La classe **`Cat`** est **dérivée** de la classe **`Animal`**.
- La classe **`Cat`** possède toutes les propriétés et méthodes de la classe **`Animal`**.
- On dit que **`Cat`** est une **spécialisation** de **`Animal`**.
- La **relation d'héritage** est indiquée dans le schéma UML à l'aide d'une flèche fermée partant de la classe fille et pointant sur la classe mère.

En créant une instance de la classe **`Cat`**, on remarque que le champ **`name`**, défini dans la classe mère **`Animal`**, s'affiche mais que les méthodes ne se trouvent pas dans le menu contextuel. Celles-ci se trouvent en fait dans le sous-menu « inherited from Animal ».



Bien entendu, une classe peut avoir plusieurs héritiers :



Il est possible qu'une classe hérite d'une classe qui elle-même hérite déjà d'une autre classe. Ceci est pourtant évité dans le présent cours.

Traductions

anglais	français	deutsch
to mew	miauler	miauen
to bark	aboyer	bellen

Conclusion

Il est donc possible qu'une classe **B hérite d'une classe A**, ce qui veut dire que la classe **`B`** possède tous les éléments de 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.

Attention :

Les classes filles n'ont pas d'accès aux éléments privés (**`private`**) de la classe mère ! Pour les accéder, il faut utiliser les accesseurs et les manipulateurs (pour autant qu'ils existent).

4.1. Extension d'un classe

Voici le code source de la classe **Animal** de l'exemple précédent :

```
public class Animal
{
     private String name;

     public String getName()
     {
          return name;
     }

     public void setName(String pName)
     {
          name = pName;
     }
}
```

Le code source de la classe **Cat** est le suivant :

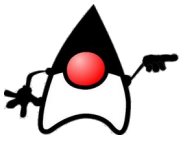
```
public class Cat extends Animal
{
     public void mew()
     {
          System.out.println("mew mew");
     }
}
```

Remarques

- Dans la classe **Cat**, la partie **extends Animal** indique que cette classe hérite de la classe **Animal**.
- Dans la classe **Animal**, il n'y a aucun indice qu'il existe un héritier. La classe de laquelle une autre classe hérite ne possède aucune connaissance de ses héritiers éventuels.

En général

On peut faire hériter une classe d'une autre classe en mettant le mot clé **extends** suivi du nom de la classe de laquelle on veut la faire hériter derrière sa déclaration.



Remarques

- Pour autant qu'elle n'est 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.

4.2. Redéfinition d'une méthode

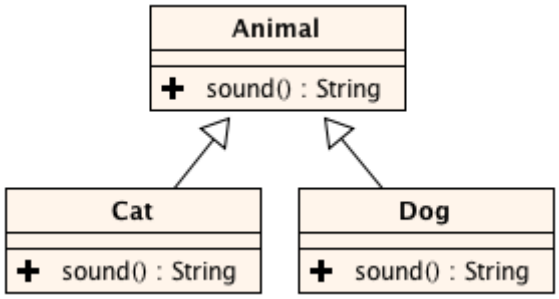
La redéfinition d'une méthode est le fait de définir dans une classe fille une méthode qui existe déjà **sous le même nom et avec les mêmes types de paramètres** dans la classe mère.

Supposons la classe **Animal** suivante :

```
public class Animal
{
     public String sound()
     {
          return "?";
     }
}
```

La méthode **sound** indique le son qu'un animal fait. On sait d'un chat miaule et qu'un chien aboie. Précédemment on avait dit qu'une sous-classe est une spécialisation d'une autre classe.

En effet, il est possible qu'une classe fille redéfinisse une méthode de la classe mère. En reprenant l'exemple des animaux, le schéma UML pourrait être le suivant :

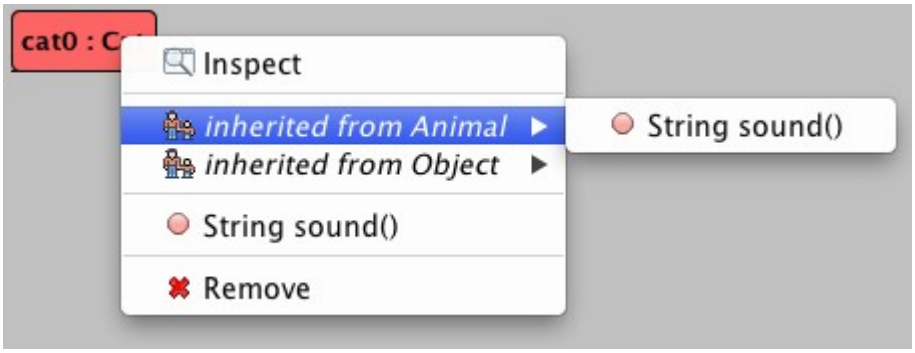


Le code source de la classe **Cat** et **Dog** serait alors le suivant :

```
public class Cat extends Animal
{
     public String sound()
     {
          return "mew";
     }
}
```

```
public class Dog extends Animal
{
     public String sound()
     {
          return "woof";
     }
}
```

En créant une instance de cette version de la classe **Cat**, on remarque que la méthode **sound()** est affichée directement dans le menu, mais qu'elle se trouve aussi dans le menu des méthodes héritées de la classe **Animal**.



On remarque par contre que, peu importe laquelle des deux méthodes **sound()** on exécute (celle de l'instance **cat0** ou celle héritée de **Animal**), le message « mew » est affiché à l'écran.

Réflexion

- Il est normal que nous obtenions l'affichage « mew » si nous appelons la méthode **sound()** de l'objet **cat0**.
- Ce qui est intéressant est que nous obtenons aussi « mew » lorsque nous appelons la méthode **sound()** héritée de **Animal**. Ceci s'explique par le fait que Java 'sait' que **cat0** n'est pas seulement un **Animal**, mais plus spécialement un **Cat**. Ainsi Java va appeler automatiquement la méthode **sound()** la mieux adaptée, c.-à-d. la plus spécialisée qu'elle trouve pour l'objet actuel **cat0**.

Exemple

Considérons attentivement les deux lignes suivantes :

```
Animal myAnimal = new Cat();
System.out.println(myAnimal.sound());
```

Nous voyons que l'objet créé est du type **Cat**, mais que la variable employée est du type **Animal**. Ceci est possible, parce qu'un chat est aussi un animal (bien sûr, puisque **Cat** est dérivé de **Animal**). Cette compatibilité est très pratique pour pouvoir traiter des objets de différents types avec une même variable d'un type plus général.[3]

Si maintenant nous appelons **myAnimal.sound()** nous pourrions croire que nous obtiendrons l'affichage « ?» à l'écran. Mais comme expliqué ci-dessus, Java 'sait' que **myAnimal** a été créé comme une instance de **Cat** et va alors appeler la méthode la mieux adaptée, celle de **Cat**. On obtiendra donc automatiquement l'affichage « mew » à l'écran ![4]

3 Remarque pour avancés : En POO le fait de pouvoir traiter une instance d'une classe comme étant aussi une instance des classes de base de cette classe s'appelle *polymorphisme*.

4 Remarque pour avancés : En POO cette technique utilisée pour lier automatiquement une méthode redéfinie au type réel de l'instance, est connue sous le nom *late binding* ou aussi *dynamic binding*.

4.3. Le mot clé «super»

Une classe peut utiliser les éléments hérités comme s'ils se trouvaient dans la classe elle-même. C'est pour cette raison qu'il existe le mot clé **super**, qui représente dans une classe fille une référence vers la classe mère. Elle permet donc de faire appel à des méthodes de la classe mère qui ont été redéfinies par la classe fille.

Attention

Pour les constructeurs, l'appel au constructeur **super(...)** doit obligatoirement être la première instruction (→ voir aussi chapitre 4.5).

Exemple

En reprenant l'exemple précédent, le code suivant :

```
Cat myCat = new Cat();
System.out.println(myCat.sound());
```

affichera le texte « mew » à l'écran. En modifiant la classe **Cat** de la manière suivante :

```
public class Cat extends Animal
{
    public String sound()
    {
        return super.sound()+"mew";
    }
}
```

puis en exécutant les mêmes deux lignes de code, le texte « ?mew » sera affiché dans la console. Le point d'interrogation « ? est bien entendu le résultat de la partie **super.sound()**.

4.4. 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, ce qui est parfaitement normal, car tout chat est aussi un animal.

4.5. L'héritage et les constructeurs

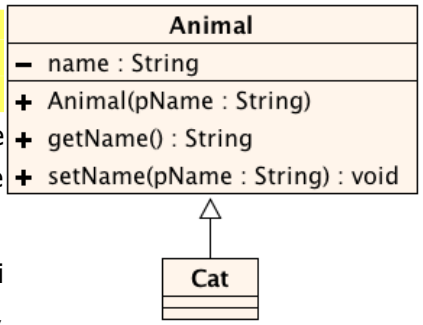
Attention ! Lorsqu'une classe étend une autre, les constructeurs ne sont pas hérités automatiquement. Si la classe mère définit un ou plusieurs constructeurs, la classe fille doit également définir (au moins) un constructeur. En plus, il faut que la classe fille fasse appel en tant que **première instruction** dans son constructeur à l'un des constructeurs de la classe mère à l'aide de la méthode **`super(...)`**.

Explication : Si la première ligne d'un constructeur n'est pas un appel au constructeur hérité, alors Java effectue automatiquement un appel au constructeur par défaut (**`super();`**). Un problème peut alors parvenir, si dans la classe mère nous avons remplacé le constructeur par défaut par un constructeur avec paramètres :

Problème

Prenons l'exemple que voici :

```
public class Cat extends Animal
{
}
```



La classe **`Cat`** hérite de la classe **`Animal`**. La classe **`Cat`** possède donc aussi un attribut **`name`** et peut y accéder via le manipulateur ou l'accesseur.

On pourrait s'imaginer que la classe **`Cat`** possède aussi automatiquement un constructeur (hérité) qui initialise le nom,

mais ceci est faux !

Le compilateur indique l'erreur :

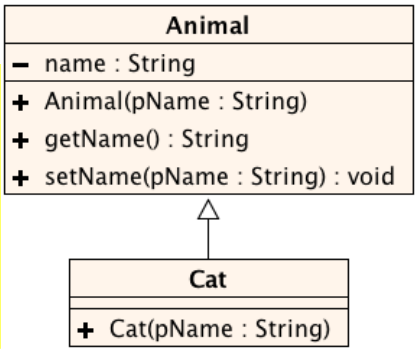
```
Cat.java @ line 1: cannot find symbol
```

Ceci est dû au fait que dans **`Animal`** le constructeur par défaut (sans paramètres) a été remplacé par le constructeur avec un paramètre. En construisant un **`Cat`**, le constructeur de **`Cat`** fait automatiquement un appel au constructeur par défaut de **`Animal`** … qui n'existe plus.

Solution

En modifiant la classe **`Cat`** de la manière suivante :

```
public class Cat extends Animal
{
     public Cat(String pName)
     {
          super(pName);
          // autres initialisations
     }
}
```

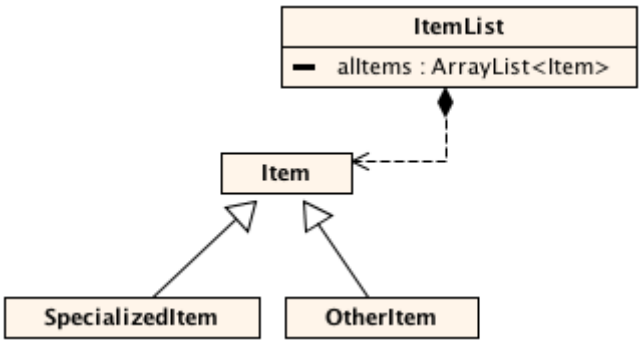


Le compilateur n'émet plus de message d'erreur. En fait, le compilateur veut forcer le développeur à initialiser correctement la classe fille. C'est pour cette raison que, exception faite des constructeurs par défaut, tous les constructeurs doivent être déclarés explicitement.

5. Application type

Le présent chapitre décrit de manière générique le développement d'une application type. Bien entendu cette « recette » doit être adaptée à une problématique spécifique et ne peut s'utiliser telle quelle. Malgré cela la manière de procéder décrite ici peut être appliquée pour résoudre un grand nombre de problèmes.

5.1. Le modèle

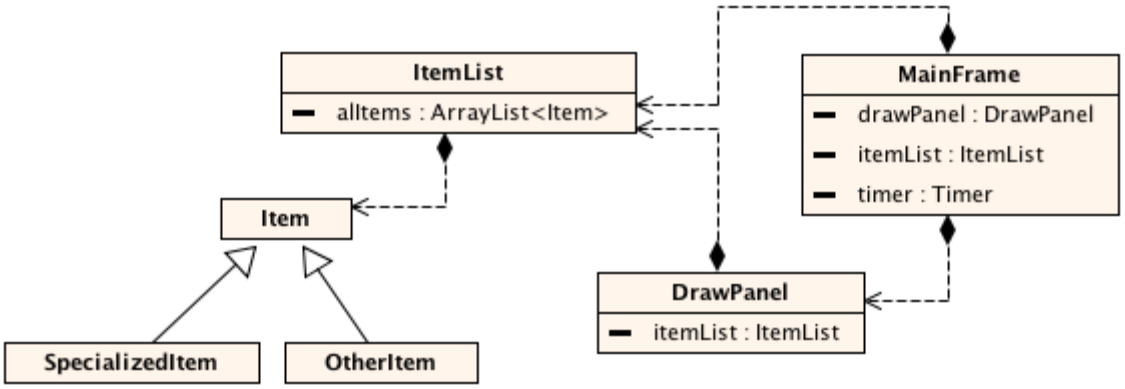


Le modèle de notre application type possède une classe **`Item`** qui représente un certain élément. Cette classe peut avoir des classes fille, dénommées **`SpecializedItem`** et **`OtherItem`** dans le schéma ci-dessus. Finalement, il existe une certaine classe, nommé ici **`ItemList`**, qui gère un ensemble d'éléments.

Remarques

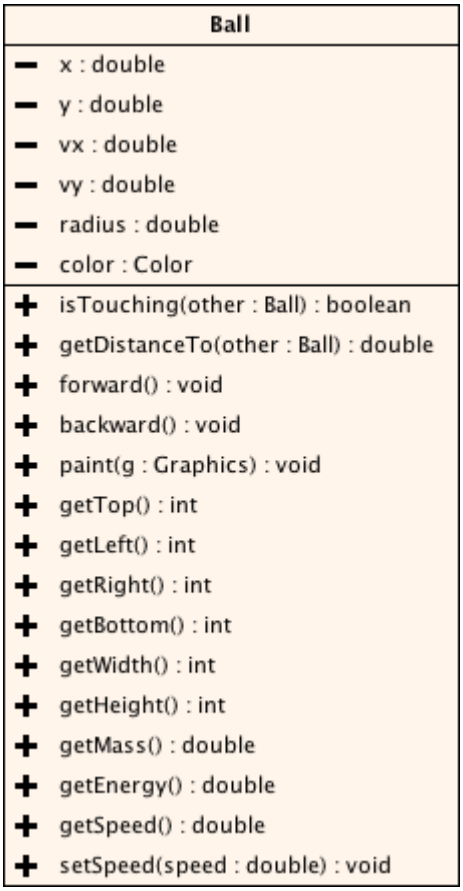
- Le schéma ci-dessus ne montre que les attributs de base. Ceux-ci dépendent bien évidemment du vrai caractère de l'application à implémenter.
- Sur le schéma ci-dessus ne sont pas non plus indiquées les différentes méthodes. Celles-ci dépendent évidemment aussi de la problématique posée.
- Le modèle comporte donc les notions suivantes :
  - une liste (**`ArrayList`**),
  - de l'héritage.

5.2. L'interface graphique



1. L'interface graphique consiste en deux classes : La fiche principale, dénommée comme toujours **`MainFrame`** ainsi qu'un canevas appelé comme d'habitude **`DrawPanel`**. Ce dernier est bien-sûr posé sur la fiche principale.
2. Les deux classes de l'interface graphique possèdent une référence (= un lien) vers le modèle, c'est-à-dire vers la classe **`ItemList`**.
   - La classe **`MainFrame`** a besoin de ce lien afin de pouvoir modifier / gérer les données contenues dans le modèle. C'est elle qui « possède » la liste.
   - La classe **`DrawPanel`** nécessite cette référence afin de pouvoir accéder aux données du modèle et ce dans le but de réaliser un dessin. Elle reçoit le lien vers le modèle de la part de la classe **`MainFrame`**.
3. La classe **`MainFrame`**, qui est toujours le contrôleur (cf. modèle MVC), possède un chronomètre qui lui permet d'exécuter une action dans des intervalles réguliers.

5.3. Exercice résolu « Bouncing Balls »



On aimerait écrire un simulateur capable de simuler des chocs élastiques entre deux boules. La partie "physique" ne faisant pas partie du problème, toutes les formules et explications sont fournies.

Il est assez clair, qu'il faudra modéliser une boule (EN: ball). Une boule en mouvement possède bien sûr les coordonnées de son centre, un vecteur vitesse - décomposé selon les deux axes et un rayon. Le vecteur vitesse ne donne pas uniquement la vitesse, mais aussi la direction dans laquelle la boule se déplace. Afin de mieux distinguer les boules, on leur donne encore une couleur.

Finalement, une boule pourra être modélisée de la façon suivante (pour des raison de lisibilité, les accesseurs et modificateurs ainsi que les constructeurs ne sont pas indiqués) :

Explications

- La méthode **`getDistance`** calcule la distance de cette boule à celle passée en tant que paramètre.
- La méthode **`isTouching`** permet de tester si cette boule touche une autre, c'est-à-dire si la distance entre les centres des deux boules est inférieure à la somme de leurs rayons.
- Les méthodes **`forward`** et **`backward`** font avancer, respectivement reculer la boule selon son vecteur vitesse actuel.
- Pour être aussi précis que possible, les attributs de la boule sont des nombres réels. Les méthodes **`getLeft`**, **`getTop`**, **`getRight`** et **`getBottom`** constituent des méthodes pour accéder facilement aux extrémités de la boule. Vu que ces méthodes seront utilisées par les parties de code responsables pour le dessin, elles fournissent immédiatement des nombres entiers en arrondissant les valeurs réelles. Il en est de même pour les méthodes **`getWidth`** et **`getHeight`**.
- Afin de pouvoir simuler des chocs entre des boules de différentes masses, la méthode **`getMass`** retourne la masse de la boule. En effet, elle ne fait que calculer le volume car on suppose que toutes les boules sont de la même matière.
- La méthode **`getEnergy`** calcule l'énergie cinétique de la boule.
- La méthode **`getSpeed`** retourne la vitesse de la boule, c'est-à-dire la norme du vecteur vitesse. En contre-partie, la méthode **`setSpeed`** met à jour la vitesse, sans pour autant changer la direction de la boule. Elle modifie donc le vecteur vitesse en multipliant ses coordonnées par le rapport de la vitesse actuelle avec la nouvelle vitesse.
- La méthode **`paint`** dessine la boule sur le canevas passé en tant que paramètre.

Réflexions préalables :

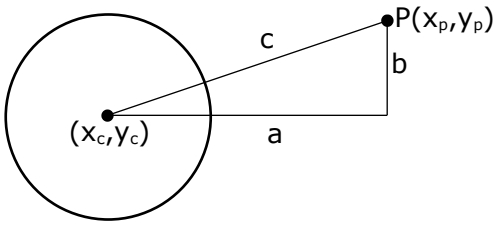
Dans cet exercice, il faut déterminer, si deux balles se touchent. Il est utile, de se poser la question plus généralement :

1. ***Comment déterminer si un Point P(xp,yp) se trouve à l'intérieur d'un cercle de centre (xc,yc) et de rayon r?***

et pour notre cas :

2. ***Comment déterminer si 2 cercles se superposent?***

1. Comment déterminer si un Point P(x_p,y_p) se trouve à l'intérieur d'un cercle de centre (x_c,y_c) et de rayon r?



Calculer la distance ***c*** du point P au centre du cercle : $c=\sqrt{a^2+b^2}$

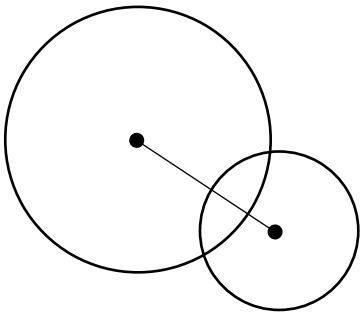
en Java : c = Math.sqrt(Math.pow(a,2) + Math.pow(b,2)) ;

avec $a=x_p-x_c$

$b=y_p-y_c$

Si $c \leq r$ alors le point P se trouve à l'intérieur du cercle (bordure incluse).

2. Comment déterminer si 2 cercles se superposent?



Calculer la distance ***c*** entre les centres des deux cercles.

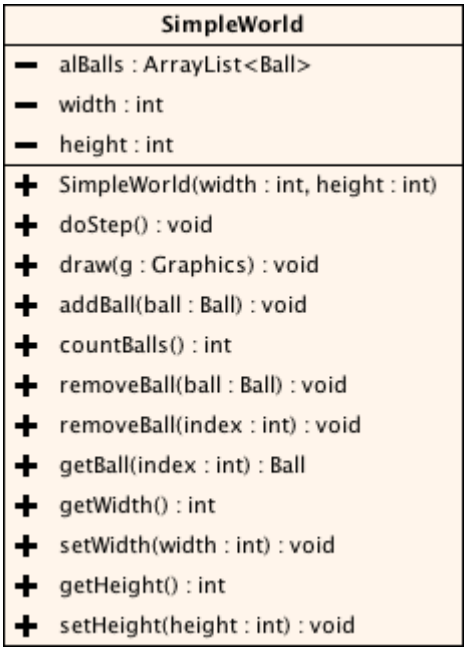
Si $c \leq r_1+r_2$ alors les 2 cercles se touchent.

Les parties de code les plus complexes de cette classe sont les suivantes :

```
    public boolean isTouching(Ball other)
    {
        return (getDistanceTo(other) <= radius+other.radius );
    }

    public double getDistanceTo(Ball other)
    {
        return Math.sqrt(Math.pow(x-other.x,2)+Math.pow(y-other.y,2));
    }

    public double getMass()
    {
        return Math.PI*radius*radius;
    }

    public double getEnergy()
    {
        return getMass()*Math.pow(getSpeed(), 2);
    }

    public double getSpeed()
    {
        return Math.sqrt(vx*vx+vy*vy);
    }

    public void setSpeed(double speed)
    {
        Double q = speed/getSpeed();
        if(!q.isNaN())
        {
            vx=vx*q;
            vy=vy*q;
        }
    }
```

Maintenant que nous avons modélisé une boule, il est temps de placer les boules dans un environnement qui sera responsable des lois physiques agissant sur elles. Pour commencer, nous allons développer un environnement très simple: Une fois qu'on place une boule dans cet environnement, elle se déplace avec la vitesse indiquée dans la direction donnée. Dans cet environnement les boules ne sont pas capables d'avoir des chocs avec d'autres boules. Dès qu'elles dépassent les limites de l'environnement, les boules disparaissent. On peut en déduire que cet environnement doit posséder une largeur et une hauteur. Bien sûr, cette classe a aussi besoin d'une liste de boules et elle dispose de plusieurs méthodes de gestion pour ces dernières.

A droite, vous trouvez le schéma UML proposé pour la classe **`SimpleWorld`** :

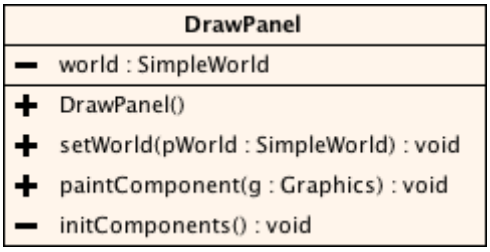


Explications

- La méthode **`doStep`** fait bouger toutes les boules. En plus, elle est responsable de supprimer les boules dès qu'elle dépasse les limites de l'environnement. Cette méthode sera appelée ultérieurement de manière périodique par un chronomètre, ce qui fera en sorte que les boules bougent. La vitesse des boules sera donc déterminée à la fois par leur vitesse propre, mais aussi par la vitesse du chronomètre. Afin de disposer d'une meilleure précision, on peut augmenter à un maximum la fréquence du chronomètre et adapter la vitesse des boules en fonction de ceci.
- La méthode **`paint`** dessine toutes les boules sur le canevas passé en tant que paramètre.

Voici les parties les plus importantes du code source :

```
public void doStep()
{
    // faire avancer toutes les boules
    for(int i=0;i<balls.size();i++)
        balls.get(i).forward();

    // supprimer les boules qui sont sorties du canevas
    for(int i=balls.size()-1;i>=0;i--)
    {
        Ball b = balls.get(i);
        if (b.getBottom()<0 || b.getRight()<0 || b.getLeft()>width ||b.getTop()>height)
            balls.remove(b);
    }
}

public void paint(Graphics g)
{
    for(int i=0;i<balls.size();i++)
        balls.get(i).paint(g);
}
```



La première version de notre modèle étant finie, il est temps d'implémenter un contrôleur et une vue afin de pouvoir vérifier que notre modèle est correct. Nous avons donc besoin d'une classe **DrawPanel** ainsi que d'une classe **MainFrame**. Comme d'habitude, une instance de la classe **DrawPanel** est ajoutée à la fiche principale **MainFrame**. La classe **DrawPanel** a comme tâche unique de dessiner un environnement. Le schéma UML y relatif sera le suivant :

Explications

- La méthode **paintComponent** vide le canevas et dessine, s'il existe, l'environnement.
- La méthode **setWorld** permet de faire passer un environnement à cette classe.

En supprimant les parties automatiquement générées par NetBeans, le code source de cette classe est le suivant :

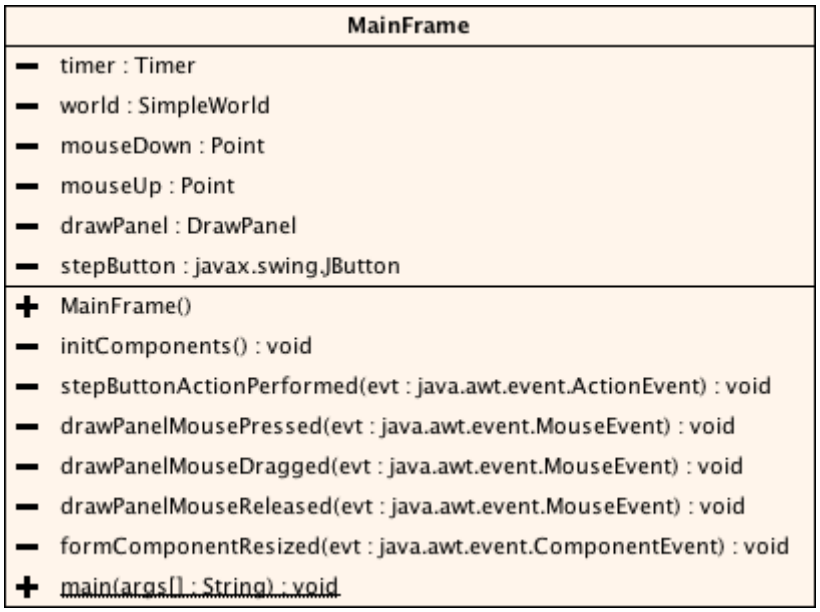
```
public class DrawPanel extends javax.swing.JPanel
{
    /** l'environnement */
    private SimpleWorld world = null;
    /** la ligne */
    private Line line = null;

    /** le constructeur */
    public DrawPanel()
    {
        initComponents();
        // améliorer les performances graphiques
        setDoubleBuffered(true);
    }

    /** modificateur pour l'environnement */
    public void setWorld(SimpleWorld pWorld)
    {
        world=pWorld;
        repaint();
    }

    /** modificateur pour la ligne */
    public void setLine(Line pLine)
    {
        line=pLine;
        repaint();
    }
```

```
@Override
    public void paintComponent(Graphics g)
    {
        // vider le canevas
        g.setColor(Color.WHITE);
        g.fillRect(0,0,getWidth(),getHeight());
        // dessiner l'environnement, c-à-d les boules
        if(world!=null) world.draw(g);
        if(line!=null) line.draw(g);
    }
}
```

Ensuite nous devons développer la classe **`MainFrame`**. Celle-ci possède les tâches suivantes :

- créer un nouvel environnement,
- créer le lien entre l'environnement et la classe **`DrawPanel`**,
- réagir au clics de l'utilisateur en créant de nouvelles balles,
- gérer le chronomètre.

On aboutit donc au schéma UML suivant :



Explications :

- La méthode de réaction du bouton, qui est appelée périodiquement par le chronomètre, fait bouger les boules, puis rafraichit le canevas.
- Afin de pouvoir ajouter des boules avec des vitesses et des directions différentes, nous allons programmer aussi les événements **`mousePressed`**, **`mouseDragged`** et **`mouseReleased`**. La méthode de réaction du premier événement enregistre la position de la souris. Celle du deuxième enregistre aussi la position de la souris, mais dans un autre attribut. Celle du dernier ajoute une boule de taille et de couleur aléatoire à la position de la souris en calculant la vitesse et la direction à partir de la position initiale de la souris. La boule n'apparaît donc qu'à partir du moment où on relâche le bouton de

la souris. Pour ne pas ajouter les boules les unes sur les autres ou à l'extérieur du canevas, certains tests sont effectués avant l'ajout de la boule. Afin de "voir" le déplacement de la souris, on peut ajouter quelques lignes de code afin de tracer une ligne entre le point initial et la position actuelle de la souris.

- Vu que le canevas ne possède pas encore de dimension lors du lancement du programme, l'environnement faisant bouger les boules doit être redimensionné lors du redimensionnement de la fenêtre principale. La taille de l'environnement est toujours égale à la taille de la vue. Pour ceci, il faut programmer l'événement **componentResized**.

En supprimant les parties automatiquement générées par NetBeans, le code source de cette classe est le suivant :

```
public class MainFrame extends javax.swing.JFrame
{
    /** le chronomètre */
    private Timer timer;
    /** notre environnement */
    private SimpleWorld world = null;
    /** point auquel le bouton de la souris est enfoncé */
    private Point mouseDown = null;
    /** point auquel le bouton de la souris est relâché */
    private Point mouseUp = null;

    public MainFrame()
    {
        initComponents();
        // créer un nouvel environnement
        world = new BounceWorld(getWidth(),getHeight());
        // passer l'environnement à la vue
        drawPanel.setWorld(world);
        // créer et démarrer le chronomètre
        timer = new Timer(10,stepButton.getActionListeners()[0]);
        timer.start();
        // cacher le bouton
        stepButton.setVisible(false);
    }

    private void stepButtonActionPerformed(java.awt.event.ActionEvent evt)
    {
        // si l'environnement existe,
        if (world!=null)
        {
            // faire bouger les boules
            world.doStep();
            repaint();
        }
    }
```

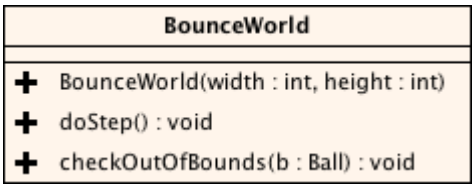
```
    private void drawPanelMousePressed(java.awt.event.MouseEvent evt)
    {
        // mémoriser ce point
        mouseDown = evt.getPoint();
        // dessiner la ligne
        drawPanel.setLine(new Line(evt.getPoint(),evt.getPoint()));
    }


    private void drawPanelMouseDragged(java.awt.event.MouseEvent evt)
    {
        // dessiner la ligne
        drawPanel.setLine(new Line(mouseDown,evt.getPoint()));
    }

    private void drawPanelMouseReleased(java.awt.event.MouseEvent evt)
    {
        // choisir une couleur aléatoire
        Color c = new Color((float)Math.random(),(float)Math.random(),(float)Math.random());
        // choisir un rayon aléatoire mais qui ne dépasse
        // pas les limites de l'environnement
        double max = Math.min(evt.getX(),evt.getY());
        max = Math.min(max,drawPanel.getWidth()-evt.getX());
        max = Math.min(max,drawPanel.getHeight()-evt.getY());
        max = Math.min(max,100);
        double r = 20 + Math.random()*(max-20);
        // créer une nouvelle boule
        Ball b = new Ball(evt.getX(), evt.getY(),
                    (mouseDown.getX()-evt.getX())/100.0,
                    (mouseDown.getY()-evt.getY())/100.0, r, c);
        // tester si la nouvelle boule ne touche pas une autre
        boolean touching = false;
        for(int i=0;i<world.countBalls();i++)
        {
            Ball o = world.getBall(i);
            if (o.isTouching(b)) touching=true;
        }
        // ajouter la boule si elle ne touche pas une autre
        // et si son rayon est au moins 20
        if(!touching && max>=20)
        {
            world.addBall(b);
            repaint();
        }
        // remettre à zéro des points de la souris
        mouseDown=null;
        mouseUp=null;
        drawPanel.setLine(null);
    }
```

```
    private void formComponentResized(java.awt.event.ComponentEvent evt)
    {
        // si la hauteur et la largeur ne sont pas zéro
        if (getWidth()!=0 && getHeight()!=0)
        {
            if (world!=null)
            {
                // adapter la taille de l'environnement
                world.setWidth(drawPanel.getWidth());
                world.setHeight(drawPanel.getHeight());
            }
        }
    }

}
```

Si tout est codé correctement, nous pouvons faire les choses suivantes en démarrant l'application :

- Enfoncer le bouton de la souris à un endroit donné du canevas et
- lorsque nous relâchons le bouton de la souris, une balle de taille et de couleur aléatoire apparaît à la position actuelle de la souris et bouge avec une vitesse relative à la longueur du trait bleu dans la même direction que ce dernier.

Maintenant, il est temps de développer la partie la plus intéressante, c'est-à-dire les chocs entre les boules. Pour ceci, il est important de savoir que le procédé indiqué ici ne fonctionne plus dès qu'il y a trop de boules sur la surface. Comme cette nouvelle classe est une spécialisation de notre environnement très simpliste, le nouvel environnement peut hériter de ce dernier. Appelons-le **`BounceWorld`**.



Explications

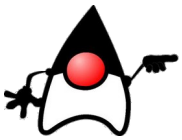
- La méthode **`doStep`** est responsable de faire bouger les boules et de calculer les chocs entre deux boules.
- La méthode **`checkOutOfBounds`** redirige une boule dès qu'elle veut dépasser les limites de l'environnement. De cette manière il est garanti que les boules ne soient pas perdues.

```
public class BounceWorld extends SimpleWorld
{
    public void checkOutOfBounds(Ball b)
    {
         // si la boule sort du haut ou du bas
        if (b.getTop()<0 || b.getBottom()>getHeight())
        {
            // invertir la vitesse verticale
            b.setVy(-b.getVy());
        }
        // si la boule sort de l'un des côtés
        if (b.getLeft()<0 || b.getRight()>getWidth())
        {
            // invertir la vitesser horizontale
            b.setVx(-b.getVx());
        }
        // si la boule est sortie quelque part
        if (b.getLeft()<0   || b.getRight()>getWidth() ||
               b.getTop()<0 || b.getBottom()>getHeight())
        {
            b.forward();      // faire avancer la boule
        }
    }
```

```
    @Override
    public void doStep()
    {
        for(int i=0;i<countBalls();i++)
        {
            Ball a = getBall(i);
            // faire avancer la boule et tester qu'elle n'est pas sortie du canevas
            a.forward();
            checkOutOfBounds(a);
            // tester les collisions avec les boules restantes
            for(int j=i+1;j<countBalls();j++)
            {
                Ball b = getBall(j);
                // si la boule a touche la boule b
                if(a!=b && a.isTouching(b))
                {
                    // reculer la boule afin de garantir qu'elle ne se
                    // trouve pas à l'intérieur de l'autre boule.
                    a.backward();
                    // calculer le choc
                    // source: http://fr.wikipedia.org/wiki/Choc_élastique
                    double d = a.getDistanceTo(b);
                    double nx = (b.getX() - a.getX())/d;
                    double ny = (b.getY() - a.getY())/d;
                    double gx = -ny;
                    double gy = nx;
                    double v1n = nx*a.getVx() + ny*a.getVy();
                    double v1g = gx*a.getVx() + gy*a.getVy();
                    double v2n = nx*b.getVx() + ny*b.getVy();
                    double v2g = gx*b.getVx() + gy*b.getVy();
                    double nax = nx*v2n;
                    double nay = ny*v2n;
                    double nbx = nx*v1n;
                    double nby = ny*v1n;
                    double mA = a.getMass();
                    double mB = b.getMass();
                    double nxv2n = (mA-mB)/(mA+mB)*nbx + 2*mB/(mA+mB)*nax;
                    double nyv2n = (mA-mB)/(mA+mB)*nby + 2*mB/(mA+mB)*nay;
                    double nxv1n = 2*mA/(mA+mB)*nbx + (mB-mA)/(mA+mB)*nax;
                    double nyv1n = 2*mA/(mA+mB)*nby + (mB-mA)/(mA+mB)*nay;
                    a.setVx(nxv2n +  gx*v1g);
                    a.setVy(nyv2n +  gy*v1g);
                    b.setVx(nxv1n +  gx*v2g);
                    b.setVy(nyv1n +  gy*v2g);
                    // faire avancer les deux boules
                    a.forward();
                    checkOutOfBounds(a);
                    b.forward();
                    checkOutOfBounds(b);
                }
            }
        }
    }
}
```

Afin d'utiliser ce nouvel environnement, il faut adapter la ligne adéquate dans la classe **`DrawPanel`**.

```
world = new BounceWorld(getWidth(),getHeight());
```



Remarques

- Le code source du présent exercice est téléchargeable sur le site officiel de la programmation au régime technique : http://java.cnpi.lu
- Dans la présente application, l'héritage ne s'applique pas aux éléments (ici des boules), mais à la classe contentant la liste (ici l'environnement).
- Dans un tel système, l'énergie totale reste constante. Afin de vérifier ceci, on pourrait calculer l'énergie totale de toutes les boules lors de chaque étape et l'afficher dans la console.
- La classe **`Line`** sert à tracer la ligne bleue lors de l'ajout d'une nouvelle boule.
- On pourrait s'imaginer d'autres environnements :
  - un environnement avec des forces de frottement,
  - un environnement avec gravitation,
  - un environnement dans lequel les boules changent de couleur en fonction de leur position ou de leur état de choc,
  - ...

Le schéma UML global (les méthodes ne sont pas affichées pour des raisons de lisibilité) de toute l'application sera donc le suivant :