13

Abstraite, cette classe est sans objet

Ce chapitre introduit la notion de classe abstraite et son exploitation lors du polymorphisme.

Sommaire : Classe abstraite — Méthode abstraite, virtuelle pure — Polymorphisme, encore — Les interfaces graphiques

Doctus — Les caractéristiques de nos objets sont un moyen de les identifier. L'ensemble de leurs méthodes permet de savoir ce qu'ils représentent et ce qu'ils font. Elles permettent même de définir nos objets.

Candidus — Nous avons déjà parlé de ça, où veux-tu en venir ?

Doc. — Rappelle-toi que la définition d'une méthode est constituée de sa signature : type retourné, nom de méthode et liste de ses arguments. Son implémentation est l'affaire du mécanisme d'héritage. Nous pouvons donc nous contenter, dans un premier temps, de déclarer l'existence de certaines méthodes tout en remettant à plus tard leur réalisation concrète.

Cand. — Et qu'est-ce qu'on y gagne ?

Doc. — Cela revient à dire que les objets savent faire certaines choses mais sans dire tout de suite comment.

Cand. — Je pourrai donc créer une classe d'objets comme je le fais d'habitude mais, je pourrai, pour certaines méthodes, dire que l'implémentation doit être recherchée dans le type concret de l'objet ?

Doc. — Il vaut mieux préciser que tu diras comment ailleurs plutôt que tout de suite. C'est en fait dans des sous-classes, bien concrètes celles-la, que tu devras réaliser les méthodes concernées.

Cand. — Nos classes concrètes représentent les différentes formes que peuvent prendre les objets qu'on manipule par leur poignée abstraite, c'est ça ?

Doc. — Ta poignée s'appelle une *classe abstraite*. Tu pourras t'en servir pour manipuler ces objets mais elle ne sera, du moins en partie, qu'une sorte de squelette que tu utiliseras pour regrouper tout ce qui est concret et commun à un groupe d'objets, tout en mentionnant des méthodes abstraites que tu te proposes de réaliser dans des sous-classes qui vont en hériter.

De Canaletto à Turner

Le peintre vénitien Canaletto a peint de multiples vues de Venise au XVIII^e siècle. Elles sont extraordinaires par la précision et la profusion de détails qui nous sont rapportés. Canaletto réalisait ses œuvres afin de satisfaire des commandes de notables anglais, exigeant une vue précise de Venise, non avare de détails, sous la forme d'un reportage fidèle à la réalité. Il y a bien évidemment une « aspiration photographique » dans ce travail. Elles sont précises au point que, grâce à elles, on a pu déduire exactement de quelle hauteur, depuis l'époque du peintre, les eaux avaient monté dans Venise.

Turner a lui aussi peint Venise quelque 100 ans plus tard. Venise y est moins nette, bien qu'on en devine les caractéristiques essentielles. Nous sommes aux sources de l'abstraction picturale, où la peinture exprime davantage la vision intérieure de l'artiste que la réalité. Il cherche à communiquer sa Venise à lui, et, ce faisant, à suggérer les émotions qu'elle provoque en lui. Néanmoins, cette abstraction conserve de nombreux traits de Venise, faisant l'économie de leur implémentation détaillée. Venise est entre les lignes. C'est la signature de Venise, bien plus que sa photo. Cela présente l'avantage de bien mieux vieillir et de ressembler à Venise aujourd'hui, bien plus que l'œuvre de Canaletto, qui n'est plus à jour. Les abstractions tiennent mieux la distance. Il en va un peu ainsi des classes abstraites par rapport aux classes concrètes. Dans une des vues de Venise de Turner, on devine un petit personnage peignant dans un coin. Vous aurez deviné de qui il s'agit.

Des classes sans objet

En se replongeant dans les deux petits programmes illustrant les mécanismes d'héritage : l'écosystème et le match de football, force serait de faire le constat suivant : dès qu'une superclasse apparaît dans le code, elle n'a plus l'occasion de donner naissance à des objets. Dans le code de l'écosystème, ne figure aucun objet de type faune ou ressource, et dans le match de football ne joue aucun joueur. Au moment de la création de l'objet à proprement parler, lorsque les joueurs montent sur le terrain, lors de l'utilisation du « new », la classe qui suit ce « new » et qui type dynamiquement l'objet n'a plus lieu d'être une superclasse.

Vous aurez tôt fait de nous rétorquer qu'en étant instance de la sous-classe, tous les objets le sont automatiquement de la superclasse. C'est exact, d'un point de vue déclaration, ou typage statique, et c'est vrai pour le compilateur (ce qui n'est déjà pas rien), mais cela ne reste que partiellement vrai lors de l'exécution. Les objets sont d'abord d'un type dynamique avant d'être également du type statique, comme nombre d'immigrés vous diront qu'ils sont d'abord français (ou devenus tels) avant d'être italien, algérien, polonais ou argentin. Tout objet peut être de plusieurs types statiques, hérités de leurs parents et grands-parents, mais ne sera que d'un, et un seul, type dynamique, sa véritable et ultime nature.

Rien n'interdit, pour l'instant, de créer dynamiquement des objets de type superclasse. Mais on conçoit aisément que, dès que l'univers conceptuel qui nous intéresse est couvert de sous-classes, c'est-à-dire, et pour reprendre la théorie des ensembles, lorsque chaque élément de l'ensemble est repris dans un sous-ensemble, il ne soit plus justifié de créer encore des objets de la superclasse. Votre voiture est une Renault avant d'être une voiture, votre chien est un cocker avant d'être un chien, le joueur de football est un attaquant ou un défenseur avant d'être un joueur. Cela pourrait néanmoins être le cas, si on vous demande de rajouter un animal dans votre logiciel sans préciser son espèce, ou si on vous offre une voiture sans préciser sa marque, ou si l'entraîneur décide d'envoyer un joueur sur le terrain sans lui indiquer quel poste il occupe, mais c'est plutôt rare. On sait pertinamment de quelle nature intime sont les objets auxquels on a affaire. Dans la pratique courante de l'OO, les superclasses, bien qu'indispensables à la factorisation des caractéristiques communes aux sous-classes, ne donnent que très rarement naissance à des objets. La figure 13-1 dépeint la situation idéale dans laquelle les objets sont tous issus d'une sous-classe ; aucun objet n'est laissé au niveau de la super classe.

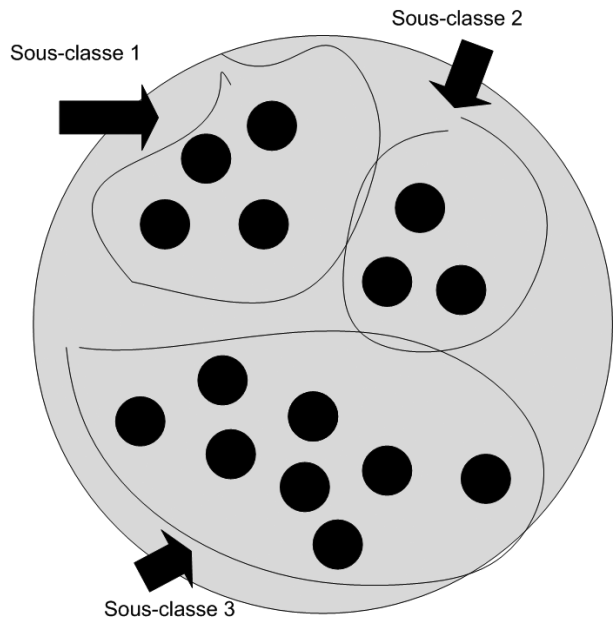


Figure 13-1
Une situation d'héritage idéale, avec tous les objets issus des seules sous-classes

Du principe de l'abstraction à l'abstraction syntaxique

Ayez à l'esprit que ce ne serait pas une bourde syntaxique de créer des objets instances d'une superclasse, sauf dans un cas précis, que nous allons maintenant détailler, et qui se produit quand vous déclarez explicitement la superclasse comme étant « abstraite ». Nous nous baserons pour comprendre la nature et le rôle des classes abstraites sur le modèle de l'écosystème. Dans le code, la classe `Jungle` envoie de manière répétée le même message `evolue()` aux objets issus des deux sous-classes de `Ressource : Eau` et `Plante`. L'exécution de ce message n'a à ce point rien de commun entre l'eau (elle s'assèche) et la plante (elle pousse) qu'aucun corps d'instruction n'est repris dans la classe `Ressource`. L'eau et la plante, bien qu'évoluant toutes deux, et capable de recevoir ce même message, d'où qu'il provienne, ne partagent rien dans l'exécution de celui-ci.

Dans le code Java qui suit, tant la classe `Eau` que la classe `Plante` intègrent la méthode `évolue() :`

```
public class Plante {
  ……  ……..
  ……  ……..

 public void evolue() {

     ……….
    ……….. .
  }
}
```

```
public class Eau {


 ……  ……..
 public void evolue() {
   ……..
    ……..
   ……….. .
     }
}
```

Vous pourriez décemment vous demander à quoi cela sert de nommer ces méthodes de la même manière, si elles décrivent des réalités si distinctes. Rappelez-vous ce que nous vous disions sur la pauvreté de notre langage, quand il s'agit de décrire des modalités actives par rapport aux modalités structurelles.

Voilà une première raison. Il en est une seconde qui tient plus à la pratique logicielle. Il est intéressant de pouvoir écrire le code de la jungle, la « tierce » classe, « cliente » de l'eau et de la plante, comme envoyant indifféremment un même message aux points d'eau et aux plantes. Une économie d'écriture sera véritablement réalisée s'il est possible, à l'instar des joueurs de football recevant le message `avance()` de l'entraîneur, de permettre à la `Jungle` d'envoyer le même message `évolue()`, en boucle, à toutes les ressources auxquelles elle est associée (sans se préoccuper du nombre et de la nature de celles-ci), comme ci-après :

```
for (int i=0; i<lesRessources.length; i++)
  lesRessources[i].evolue();
```

On pourrait imaginer créer un ensemble de 100 points d'eau, par la simple instruction :

```
for (int i=0 ; i<100 ; i++)
  lesRessources[i] = new Eau() ;
```

Et 200 plantes, au moyen de :

```
for (int i=0 ; i<200 ; i++)
  lesRessources[100+i] = new Plante() ;
```

Et d'envoyer ensuite le message `évolue()` sur ces 300 ressources. C'est en effet ce que l'on cherche à faire, en tous les cas, au moment de l'exécution du code. On désirerait ajouter un nouveau type de ressource, par exemple, des cadavres en décomposition d'autres animaux, que le code de la jungle ne se modifierait en rien.

Malheureusement pour nous (mais heureusement dans pratiquement tous les cas de figure), l'exécution est toujours précédée par une étape de compilation (comme nous le savons, Python et PHP 5 se distinguent ici) qui, parmi d'autres choses, fait office de correcteur syntaxique plutôt sévère. Or, comme dans tous les langages de programmation, un tableau se doit d'être typé. Si nous voulons donc installer toutes les ressources dans un tableau, il faudra typer ce dernier au moyen d'une instruction telle que :

```
Ressource [] lesRessources = new Ressource[300].
```

Pouvions-nous typer ce tableau comme `Plante` ? Non, car il y a des points d'eau dans l'affaire. Et comme `Eau` ? Non, car il y a des plantes dans l'affaire. La seule solution est de le typer comme `Ressource`, puisqu'en effet, tant les plantes que les eaux en sont. Et nous nous retrouvons, comme dans le chapitre précédent, en présence d'objet dont le type statique, `Ressource`, diffère du type dynamique : `Eau` ou `Plante`. Nous nageons à nouveau, avec bonheur, en plein polymorphisme.

La classe `Jungle` pourrait également recevoir dans une de ses méthodes un argument de type `Ressource`, sur lequel elle enverrait un message commun à la plante ou l'eau, et qui serait par la suite exécuté différemment. Une nouveauté, essentielle ici, est que ni la classe `Eau` ni la classe `Plante` ne redéfinisse une méthode `evolue()`, qui aurait une part d'instructions déjà prévue dans la superclasse. Pourtant, si nous typons le tableau des ressources comme `Ressource`, et que nous envoyons le message « évolue » sur chacun de ces objets, le compilateur, pour qui seul le type statique a voix au chapitre, ne pourra accepter qu'aucune méthode `evolue()` ne soit en effet présente dans la classe `Ressource`.

Dilemme, dont la seule issue possible est d'installer une méthode `evolue()` dans la classe `Ressource`, tout en déclarant cette méthode « abstraite », c'est-à-dire sans corps d'instruction. Une méthode abstraite est une méthode qui se limite à sa seule signature, une méthode qui ne fait rien, à part se présenter.

En Java, en C# et en PHP 5, nous la déclarons dans la classe `Ressource` de la manière suivante :

```
    abstract public void evolue();
```

En C++, elle serait dite méthode « virtuelle pure », et se déclare ainsi :

```
    public :
      void virtual evolue() = 0;
```

Elle est virtuelle par la présence de `virtual`. En ajoutant `= 0`, on la rend abstraite. Nous verrons par la suite une manière de réaliser l'abstraction dans Python.

Classe abstraite

Toute classe contenant au moins une méthode abstraite devient d'office abstraite. D'ailleurs, tant Java que C# et PHP 5 forcent le trait, en vous obligeant à rajouter le mot-clé `abstract` dans la déclaration de la classe, comme suit :

```
public abstract class Ressource extends ObjetJungle {
     …….
}
```

C++ reste plus sobre et sait que l'abstraction d'au moins une méthode entraîne l'abstraction de toute la classe. Il n'y a d'autre moyen de rendre une classe abstraite qu'en y installant une méthode abstraite ou virtuelle pure. En fait, Java, C# et PHP 5 n'accepteraient pas qu'une méthode abstraite ne fût définie dans une classe, elle-même déclarée comme abstraite, mais le contraire ne s'applique pas. Les trois langages acceptent d'une classe qu'elle soit abstraite, alors qu'aucune méthode abstraite ne s'y trouve. Ils bloquent ainsi la possibilité pour certaines classes de donner naissance à des objets, indifféremment du fait qu'elles intègrent une méthode abstraite. Dans la pratique, très logiquement, une classe ne sera généralement abstraite que si une méthode abstraite s'y trouve.

« new » et « abstract » incompatibles

Au début de ce chapitre, nous vous expliquions que, souvent, les superclasses ne donnent pas naissance à des objets. Dorénavant, elles le pourront d'autant moins qu'elles seront déclarées abstraites. `new` et `abstract` sont deux mots-clés totalement incompatibles, en ce sens qu'aucune allocation de mémoire ne peut être effectuée pour des instances de classe abstraite. Si nous revenons à la définition première des classes abstraites, c'est-à-dire qu'elles contiennent au moins une méthode abstraite, cette interdiction doit vous paraître logique.

Supposons une classe contenant une méthode abstraite et pouvant donner naissance à des objets. Tout objet se doit être capable d'exécuter tous les messages reçus. Qu'en serait-il du message issu de la méthode abstraite ? Le compilateur ne tiquerait pas, car la syntaxe du message est parfaitement correcte. Mais que faire à l'exécution, face à un corps d'instruction absent ? On enverrait un message qui dit de ne rien faire ? Cette possibilité a d'office été bannie par les langages OO, car un message se doit de faire quelque chose.

Notez pour l'anecdote qu'un corps d'instruction vide est considéré comme distinct de pas de corps d'instruction du tout : `public void evolue() {}` est différent de `abstract public void evolue()`. Tous les langages OO interdisent l'envoi de messages à partir de méthodes sans corps d'instruction, mais cette interdiction est levée pour des méthodes dont le corps d'instruction, bien qu'existant, est vide. Seules les premières méthodes sont abstraites, les autres sont stupides mais concrètes !

Abstraite de père en fils

Au contraire des superclasses concrètes, les superclasses abstraites obligent à redéfinir (ne serait-il sans doute pas plus approprié de simplement dire « définir » ?) les méthodes abstraites dans leurs sous-classes. Tant que la méthode abstraite n'est pas redéfinie dans les sous-classes, chacune de ces sous-classes se doit de rester abstraite,

et aucune ne donnera naissance au moindre objet. Le compilateur se chargera de vérifier que vous maintenez l'abstraction, de sous-classes en sous-classes, jusqu'à ce que toutes les méthodes abstraites soient redéfinies.

Ci-après, vous voyez le code Java de la superclasse abstraite `Ressource` et de la sous-classe concrète `Eau`. La méthode `dessineToi()`, qui représente graphiquement la ressource, est abstraite dans la classe `Ressource`, car il est nécessaire de savoir de quelle ressource il s'agit avant de la dessiner. Tous les objets se dessinent, mais tous le feront à leur manière. La méthode `evolue()`, pour des raisons déjà évoquées, est également abstraite. Les plantes évoluent en grandissant, les points d'eau en diminuant de taille.

```
public abstract class Ressource extends ObjetJungle { /* classe abstraite */
  private int temps;
  private int quantite;

  Ressource () {
    super();
    temps = 0;
    quantite = 100;
  }
  abstract public void dessineToi(Graphics g);  /* méthode abstraite */
  abstract public void evolue();                /* méthode abstraite */
  public void incrementeTemps() {
    temps++;
  }
  public int getTemps() {
    return temps;
  }
  public void diminueQuantite() {
    decroitTaille(2);
  }
}
public class Eau extends Ressource {
  Eau() {
    super();
  }
  public void dessineToi(Graphics g) {  /* définition de cette méthode abstraite */
    g.setColor(Color.blue);
    g.fillOval(getMaZone().x, getMaZone().y, getMaZone().width, getMaZone().height);
  }
  public void evolue() {                /* définition de cette méthode abstraite */
    incrementeTemps();
    if ((getTemps()%10) == 0)
      decroitTaille(2);
  }
}
```

Un petit exemple dans les cinq langages de programmation

Ci-après, vous trouverez en Java, C#, PHP 5 et C++ un même exemple d'une superclasse abstraite, dû à la présence en son sein d'une méthode abstraite, ainsi que deux sous-classes concrétisant cette même méthode de deux manières différentes.

En Java

```
abstract class O1 {
  abstract public void jexisteSansRienFaire();
}
class FilsO1 extends O1 {
  public void jexisteSansRienFaire() {
    System.out.println("ce n'est pas vrai, je fais quelque chose");
  }
}
class AutreFilsO1 extends O1 {
  public void jexisteSansRienFaire() {
    System.out.println("c'est de nouveau faux, moi aussi je fais quelque chose");
  }
}
public class ExempleAbstract {
  public static void main(String[] args) {
    /* O1 unO1 = new O1(); impossible */
    O1 unFilsO1         = new FilsO1();
    O1 unAutreFilsO1    = new AutreFilsO1();
    unFilsO1.jexisteSansRienFaire();
    unAutreFilsO1.jexisteSansRienFaire();
  }
}
```

Résultat

```
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
```

Remarquez que nous avons délibérément typé statiquement et dynamiquement nos objets de manière différente, le type statique ne pouvant être qu'une superclasse du type dynamique. Alors qu'il n'est pas possible de typer dynamiquement un objet par une classe abstraite, comme le montre le code (impossible de créer un objet comme étant typé « définitivement » par une classe abstraite), il n'y a aucun problème pour le typer statiquement avec une classe abstraite. C'est de fait une pratique très courante et inhérente au polymorphisme.

En C#

```
using System;
abstract class O1 {
  abstract public void jexisteSansRienFaire();
}
class FilsO1 : O1 {
  public override void jexisteSansRienFaire() {
    Console.WriteLine("ce n'est pas vrai, je fais quelque chose");
  }
}
```

```
class AutreFilsO1 : O1{
  public override void jexisteSansRienFaire() {
    Console.WriteLine("c'est de nouveau faux, moi aussi je fais quelque chose");
  }
}
public class ExempleAbstract {
  public static void Main() {
    /* O1 unO1 = new O1(); impossible */
    O1 unFilsO1 = new FilsO1();
    O1 unAutreFilsO1 = new AutreFilsO1();
    unFilsO1.jexisteSansRienFaire();
    unAutreFilsO1.jexisteSansRienFaire();
  }
}
```

Résultat

```
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
```

Rien d'essentiellement différent par rapport au code Java, si ce n'est l'addition du mot-clé `override`, lors de la concrétisation des méthodes abstraites. Le mot-clé `virtual`, lors de la déclaration des méthodes abstraites, n'est plus nécessaire, comme il l'est lors de la déclaration des méthodes, non plus abstraites, mais concrètes et à redéfinir.

En PHP 5

```
<html>
<head>
<title> Héritage et abstraction </title>
</head>
<body>
<h1> Héritage et abstraction </h1>
<br>

<?php
     abstract class O1 {
          abstract public function jexisteSansRienFaire();
     }

     class FilsO1 extends O1 {
          public function jexisteSansRienFaire() {
               print ("ce n'est pas vrai, je fais quelque chose <br> \n");
          }
     }

     class AutreFilsO1 extends O1 {
          public function jexisteSansRienFaire() {
               print ("c'est de nouveau faux, moi aussi je fais quelque chose <br> \n");
          }
     }

     $unFilsO1 = new FilsO1();
     $unAutreFilsO1 = new AutreFilsO1();
     $unFilsO1->jexisteSansRienFaire();
```

```
    $unAutreFils01->jexisteSansRienFaire();
?>
</body>
</html>
```

Point de typage statique différent d'un typage dynamique, car point de compilation. Mais à cette différence essentielle près, l'abstraction se réalise comme dans les deux langages précédents (et elle est plutôt très inspirée de Java, comme l'est toute la partie « héritage » de PHP 5).

En C++

```
#include "stdafx.h"
#include "iostream.h"
class O1 {
  public:
    virtual void jexisteSansRienFaire() = 0;
};
class FilsO1 : public O1 {
  public:
    void jexisteSansRienFaire() {
      cout <<"ce n'est pas vrai, je fais quelque chose" << endl;
    }
};
class AutreFilsO1 : public O1 {
  public:
    void jexisteSansRienFaire() {
      cout <<"c'est de nouveau faux, moi aussi je fais quelque chose" << endl;
    }
};
int main(int argc, char* argv[]) {
  FilsO1 unFilsO1;
  AutreFilsO1 unAutreFilsO1;
  /* O1 unO1; impossible */
  unFilsO1.jexisteSansRienFaire();
  unAutreFilsO1.jexisteSansRienFaire();

  O1* unFilsO1Pointeur = new FilsO1();
  O1* unAutreFilsO1Pointeur = new AutreFilsO1();

  unFilsO1Pointeur->jexisteSansRienFaire();
  unAutreFilsO1Pointeur->jexisteSansRienFaire();

  return 0;
}
```

Résultat

```
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
ce n'est pas vrai, je fais quelque chose
c'est de nouveau faux, moi aussi je fais quelque chose
```

En C++, le mot-clé `abstract` disparaît. La déclaration de la méthode comme « virtuelle pure » suffit à rendre la classe abstraite. Nous présentons deux versions du programme, selon que les objets sont créés dans la mémoire pile ou la mémoire tas. Nous voyons que, dans le premier cas, il n'est pas possible de typer statiquement des objets par une classe abstraite car, comme dans ce cas, la seule déclaration suffit à la création de l'objet, les deux types se doivent d'être égaux. L'utilisation du polymorphisme à partir de classe abstaite n'est donc possible qu'avec des pointeurs et sur des objets installés dans la mémoire tas.

Classe abstraite

Une classe abstraite en Java, C##, PHP 5 et C++ (dans ce cas, en présence d'une méthode virtuelle pure) ne peut donner naissance à des objets. Elle a comme unique rôle de factoriser des méthodes et des attributs communs aux sous-classes. Si une méthode est abstraite dans cette classe, il sera indispensable de redéfinir cette méthode dans les sous-classes, sauf à maintenir l'abstraction pour les sous-classes et à opérer la concrétisation quelques niveaux en dessous.

L'abstraction en Python

Bien qu'il ne soit pas possible de déclarer une classe abstraitre en Python (à cause de la simplification de la syntaxe et de l'affaiblissement du typage), une petite pirouette, illustrée dans le code ci-dessous, permet d'empêcher la classe `01` de donner naissance à des objets.

```
class 01:
       def __init__(self):
       assert self.__class__ is not 01
       def jexisteSansRienFaire(self):
              pass

class Fils01(01):
       def jexisteSansRienFaire(self):
              print "ce n'est pas vrai, je fais quelque chose"

class AutreFils01(01):
       def jexisteSansRienFaire(self):
              print "c'est de nouveau faux, moi aussi je fais quelque chose"

# un01=01() devenu impossible
unFils01=Fils01()
unAutreFils01=AutreFils01()
unFils01.jexisteSansRienFaire()
unAutreFils01.jexisteSansRienFaire()
```

L'intruction « assert » évalue la condition qui suit (ici, dans le constructeur, vérifie si l'objet en création n'est pas de la classe `01`). Si cette condition est vérifiée, cette instruction ne fait rien. Dans le cas contraire, une exception est produite et levée (elle peut alors être `try-catch` comme expliqué dans le chapitre 7). La classe `01` pourra être héritée et aura dès lors comme seul rôle de factoriser des méthodes à redéfinir dans les classes filles. Comme Python n'a que faire du typage statique, les classes abstraites ne joueront pas un rôle exactement semblable à celui joué dans les situations polymorphiques possibles et fréquentes dans les trois autres langages.

Un petit supplément de polymorphisme

Les enfants de la balle

Monsieur Loyal, dans son costume rouge mité, centré au milieu du disque lumineux, d'un revers de la main interrompt les cuivres et les cymbales un peu rouillés, et dans un éclat de voix strident hurle : « Que tous les artistes fassent leur numéro. » Et, bientôt, l'un après l'autre, tous les artistes viendront s'exécuter sur la piste. Mais ce qu'ils ne savent pas tous ces artistes, ces jongleurs, ces clowns, ces trapézistes, ces funambules et ces dompteurs, tous ces enfants ou, devrais-je dire, toutes ces sous-classes de la balle, c'est qu'ils feront leur numéro, ils le feront mais ne le feront pas à la manière Bouglione ou à la manière Pinter, ils le feront à la manière polymorphique.

Tous ont reçu cinq sur cinq le message de Monsieur Loyal, tous exécuteront leur numéro, tous l'exécuteront avec méthode, mais chacun à sa façon. Monsieur Loyal envoie un même message à un tableau d'artistes de cirque, artiste abstrait (la méthode `faireMonNuméro` étant abstraite dans la classe `Artiste`), et chacun se lancera dans le numéro qu'il a concrétisé et si longtemps répété dans sa sous-classe d'artiste, devenue pour le coup concrète, elle aussi : `Jongleur`, `Trapéziste`, `Dompteur`…

Il est important pour M. Loyal de savoir que la classe `Artiste` existe pour pouvoir interagir avec tous ces artistes d'une seule et même manière, quitte à ce que ce qui leur soit demandé se prête à une réalisation différente. S'il convoque un de ces artistes dans son bureau pour le payer, il lui enverra un message, ayant comme but l'établissement des prestations. Il est, là encore, bien possible que ces prestations doivent être établies différemment selon l'artiste en question. Un clown pourrait coûter moins cher qu'un dompteur ou un trapéziste, d'où sa tristesse.

Cliquez frénétiquement

Empoignez votre souris d'ordinateur, et cliquez frénétiquement, un clic, deux clics, cliquez où vous pouvez, cliquez où vous voulez, mais cliquez. Ce qui se produit sur l'écran, en réponse à ces clics, dépend de l'endroit où vous cliquez, de l'objet graphique sur lequel vous cliquez. Un menu apparaît, une fenêtre se ferme, une autre s'ouvre, un onglet passe au premier plan, une nouvelle police de caractère est mémorisée, le curseur se transforme en croix, etc. Il s'en passe des choses en réaction à ce clic, et pourtant le clic est toujours le même.

Parfois, vous pouvez juste le doubler rapidement, parfois votre souris possède un, deux ou trois boutons, et le clic se fait sur l'un ou l'autre. Cela laisse malgré tout très peu de modalités d'action, en comparaison au nombre d'objets graphiques qui réagiront différemment en réaction à ces quelques modalités d'action. Une poignée de modalités d'action, effectives sur une large panoplie d'objets, réagissant tous différemment en regard de ces actions, les interfaces graphiques des systèmes exploitations, Windows, Mac OS ou Linux, sont de merveilleux exemples de polymorphisme. Une souris, un clic, et une véritable taxonomie d'objets graphiques, capables de réagir à ce clic, voilà comment on peut résumer très schématiquement l'interaction de l'utilisateur avec ces systèmes d'exploitation.

Pourquoi ces objets graphiques sont-ils organisés de façon hiérarchique ? Car une fenêtre est un de ces objets qui peut se déplacer, s'agrandir, et possède un système de glissement qui permet de dévoiler, en partie seulement, la fenêtre. Une icône est un objet graphique qui peut juste se déplacer, un menu est un objet graphique qui ne peut pas se déplacer mais peut s'ouvrir, etc. Il est clair que certaines modalités d'action sont partagées par certains de ces objets et, ainsi, sont-elles factorisables dans des superclasses abstraites. D'autres seront spécifiées de manière polymorphique, au bas de l'arbre taxonomique, comme les effets de la souris. La figure 13-2 montre les différents objets graphiques Java, et leur structure d'héritage.

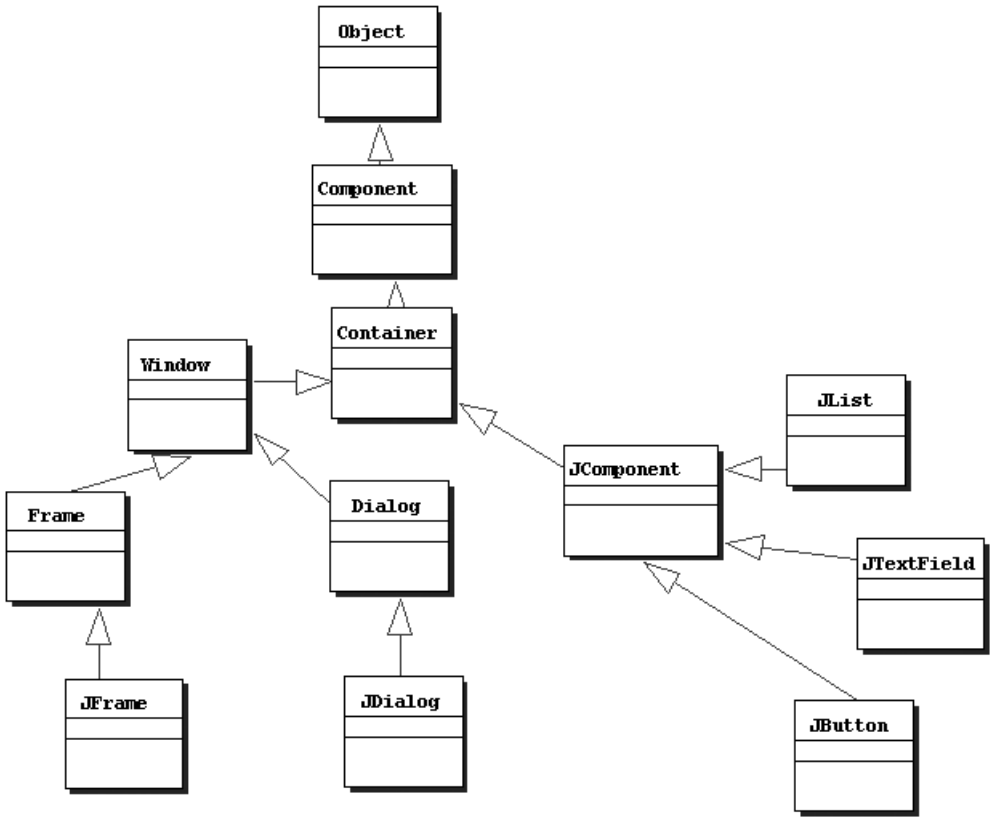


Figure 13-2
La hiérarchie des classes graphiques en Java.

On constate que Java, comme recommandé par la charte du bon programmeur objet, n'est pas économe des niveaux d'héritage. Ici on en décompte jusque six.

Ce détour par les interfaces graphiques des systèmes d'exploitation est loin d'être innocent, car l'histoire de l'orienté objet est concomitante, en partie, à l'histoire des interfaces graphiques. Lorsque Steve Jobs, célèbre gourou de l'informatique et créateur des Macintosh, se rend au Xerox PARC en 1979, Alan Kay, leader d'un groupe de recherche, lui présente les trois technologies innovantes sur lesquelles il planche. Tout d'abord, la mise en réseau des ordinateurs selon un protocole encore balbutiant : Internet. Steve Jobs n'y voit rien de très prometteur. Ensuite, une nouvelle manière de programmer, implémentée, en grande partie, dans un nouveau langage de programmation, inspiré de Simula, mais largement amélioré : Smalltalk.

À nouveau, Steve Jobs ne voit pas là de quoi fouetter un programmeur. Alan Kay décrit pourtant cette manière de programmer, OO comme il se doit, comme l'approche la plus élégante et la plus directe pour réaliser ce qui est son troisième domaine de recherche : la conception de nouvelles modalités d'interaction avec l'ordinateur : fenêtres, souris, menus… On connaît aujourd'hui cette musique par cœur, mais, en 1979, toute interaction se faisait *via* des lignes de commande tapées au clavier. En découvrant cette dernière recherche, Steve Jobs est subjugué, il n'en croit pas ses oreilles et ses yeux, et il comprend que c'est la manière la plus innovante et à la fois la plus naturelle de penser l'utilisation de l'ordinateur. Il part avec, sous le bras, son projet d'un nouveau système d'exploitation pour les Macs. On connaît la suite de l'histoire…

Un certain Bill Gates passa aussi par là, jeta un œil par la fenêtre, et Windows, comme par hasard, vi le jour. Chaque fois que vous cliquez à l'écran, ouvrez une fenêtre ou déroulez un menu, c'est en partie à Alan Kay que vous le devez.

Alan Kay

Il obtient son doctorat de l'université d'Utah en 1969 ; le sujet en est le développement d'interfaces graphiques 3-D. L'approche OO lui semble la plus naturelle pour la réalisation de ces interfaces graphiques. De ses 10 années passées au légendaire Xerox PARC (et il n'est pas pour rien dans la naissance de cette légende), il aura apporté une contribution essentielle à la mise au point du langage Smalltalk qui, bien que venant après Simula, est sans conteste le premier langage OO populaire et diffusé. Il participe aussi activement à la mise au point des protocoles réseaux Ethernet et Internet.

Son dernier sujet de préoccupation, et qui l'occupe encore activement aujourd'hui, est de repenser l'interaction homme-ordinateur, afin de ne pas laisser l'ordinateur se cantonner à une machine à écrire sophistiquée, mais de le transformer en un réel support à la créativité et l'imagination. C'est en observant, auprès de Seymour Papert, les enfants utiliser l'ordinateur, qu'il se rend compte qu'il y a dans cette « machine » un potentiel largement sous-exploité pour l'enrichissement intellectuel des enfants et des adultes en général. Depuis toutes ces années d'observation, l'éducation des enfants (dès l'âge de 6 ans), par le biais d'une utilisation mieux pensée des technologies de l'information, est devenue pour lui une croisade.

On sait tout ce que Steve Jobs lui doit et, de fait, il s'associe au succès d'Apple pendant les 10 années qui suivent. Il apporte sa touche essentielle dans la mise au point du Netwon d'Apple, mais le succès n'est pas au rendez-vous, car son rêve d'un système permettant une vraie interaction intuitive et créative, un compagnon aussi indispensable que discret, ne parvient à aboutir, suite à des difficultés techniques et des freinages commerciaux.

Il devient ensuite pendant 5 ans le vice-président pour la recherche et le développement de la « Walt Disney Company » où il a l'occasion de se convaincre chaque jour davantage que, selon ses propres termes, la « révolution informatique ne s'est toujours pas produite » (mais qu'est-ce qui lui faut ?) et que « la meilleure manière de prédire le futur est de l'inventer ». Sa présence auprès de Walt Disney lui permet d'approfondir l'interaction entre les enfants et l'ordinateur, afin de faire de ce dernier une aide véritable à la pédagogie, une stimulation à la créativité et une vraie réponse à la soif insatiable d'apprentissage de l'enfant. Il est aujourd'hui, en compagnie de son ami Nicolas Negroponte du MIT, derrière le projet OLPC « un laptop pour chaque enfant », facilitant par des prix dérisoires (100 $) l'acquisition d'un portable pour chaque enfant. Ces ordinateurs sont largement inspirés de ses premiers travaux sur une interface homme/machine extrêmement conviviale dénommée « dynabook ».

À plus de 60 ans, et de façon à définitivement se donner les moyens de ses ambitions, il fonde sa propre compagnie : « Viewpoints Research Institute », consacrée au développement d'outils informatiques, permettant une meilleure appréhension des systèmes complexes, et destinés tant aux adultes qu'aux enfants. Une des productions de cette nouvelle compagnie est le langage de développement orienté objet, « Squeak », au sujet duquel la maison d'édition Eyrolles a récemment produit un livre[a]. De par sa grande facilité d'utilisation, ce langage devrait permettre tant aux enfants qu'aux éducateurs de « jouer » et « d'expérimenter » de manière créative, ludique et plus intuitive, les maths et la science. Ce nouveau langage, directement issu de Smalltalk, devrait aider Alan Kay à imposer tant ses nouvelles idées sur la pédagogie que sa vision très personnelle, et dont le manque de réceptivité l'obsède, sur l'interaction homme-machine.

Enfin, ne soyez plus étonnés que ce livre reprenne des exemples de biologie et de chimie, nous ne pouvons rêver d'une meilleure compagnie, car Alan Kay obtint son premier diplôme universitaire en biologie moléculaire. C'est à partir de là, et sans arrêt depuis, qu'il se mit à penser l'informatique, son informatique à lui, en des termes biologiques. Ainsi l'idée de messages entre objets est-elle directement inspirée de l'idée de messages chimiques entre les cellules.

a. *Squeak*, Xavier Briffault, Stéphane Ducasse, Eyrolles 2001.

Alan Kay considère qu'un ordinateur idéal se doit de fonctionner comme un organisme vivant, complexe mais se divisant la tâche en un ensemble de modules simples, chaque module devant agir de concert avec les autres (les communications ayant lieu par messages chimiques) afin de réaliser une fonctionnalité commune. Mais tous les modules doivent conserver une certaine autonomie et préserver leur intégrité.

Steve Jobs doit se mordre les doigts de n'avoir saisi qu'une seule innovation importante parmi tout ce dont lui parla Alan Kay, car tout de ce qui fait l'informatique aujourd'hui : les réseaux, la programmation OO, les interfaces graphiques, les imprimantes laser, les ordinateurs de poche, l'informatique ubiquitaire, est redevable en partie à l'extraordinaire imagination et la créativité débordante d'Alan Kay. Allant de déception en déception, à force de voir ses idées spoliées en ce qu'elles ont de plus rémunératrices, il espère par cette nouvelle entreprise arriver enfin, sans doute par sa juste réappropriation et son adoption par les enfants, à redonner toutes ses lettres de noblesse à l'ordinateur, tel qu'il souhaiterait le voir utiliser. Ayant lu tout cela, vous ne serez guère surpris d'apprendre qu'il a reçu en 2004 le prix Turing (la version informatique du Nobel).

Le Paris-Dakar

L'organisateur du Paris-Dakar doit planifier à l'avance la consommation de tout le convoi de véhicules qui participent à cette course : moto, camions, buggy, voiture, hélicoptère… Tous ces véhicules consomment, mais tous le font différemment en fonction des kilomètres parcourus. Il est capital de comprendre ici que c'est la manière de calculer la consommation qui diffère d'un véhicule à l'autre. Si la seule différence se ramenait à une valeur, par exemple, simplement la consommation par km, et que le calcul de la consommation revenait à multiplier cette valeur par le nombre de km à parcourir, il n'y aurait nul besoin d'héritage, nul besoin de polymorphisme. Il n'y aurait, qui plus est, qu'une seule classe de véhicule, dont les objets se distingueraient, notamment par la valeur de cette consommation kilométrique.

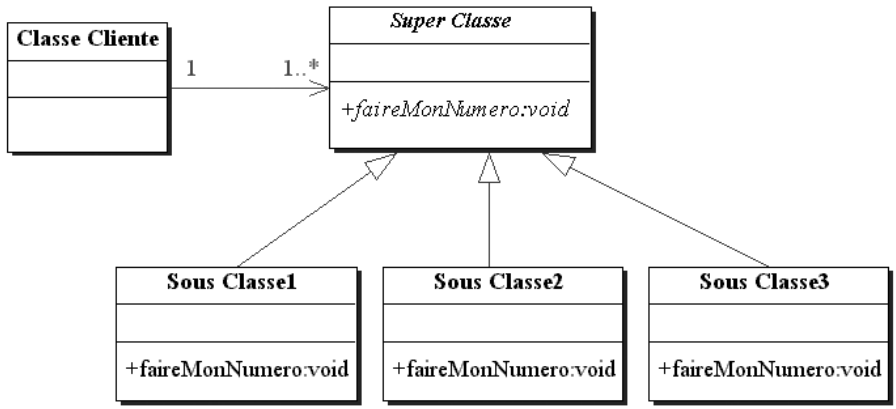
Si seule la valeur des attributs différencie les objets entre eux, point n'est besoin de sous-classe. L'usage des langues naturelles, en général, ne permet pas de distinguer le lien existant entre un objet et sa classe d'un côté, et le lien entre une sous-classe et sa superclasse de l'autre. Ma Renault, objet, est une Renault, classe. Une Renault, sous-classe, est une voiture, superclasse. La même expression « est une » est utilisée ici, alors qu'en programmation OO, ces deux contextes d'utilisation mènent à des développements logiciels très différents : simple instanciation d'objet dans le premier, mise en place d'une structure d'héritage à deux niveaux dans le second. Ne vous trompez pas et n'abusez pas d'héritage inutile. Il se doit d'alléger et non pas d'alourdir votre conception. Le gain de son apport, en clarté, économie, maintenance et extensibilité, doit être suffisamment important. Autrement, n'y songez même pas et contentez-vous d'un seul niveau taxonomique. C'est déjà bien assez.

Le polymorphisme en UML

Dans les trois exemples de polymorphisme, présentés plus haut, ce qui apparaît à chaque fois est un type d'architecture logicielle comme celle décrite en UML ci-après.

Dans UML, toute classe ou méthode abstraite est indiquée en italique. Ici, la superclasse et la seule méthode de la classe sont, de fait, abstraites. Trois sous-classes concrètes redéfinissent la méthode `faire-Mon-Numero()`, une méthode dont le corps d'instruction sera radicalement différent pour chacune des sous-classes.

Figure 13-3
Diagramme de classes UML classique associé au polymorphisme.



Une « classe cliente » envoie indifféremment le message `faireMonNumero()` à tous les référents qu'elle possède comme attribut. Comme indiqué dans le diagramme, cet ou ces attributs seront typés par la superclasse. Il peut s'agir d'un et un seul référent, typé statiquement d'une unique manière, mais qui, lors de l'exécution du message, peut endosser plusieurs types dynamiques différents. Cela peut être un tableau de référents, qui lors de l'envoi d'un même message, en boucle sur tous les éléments du tableau, donnera lieu à des exécutions différentes.

Aucun objet n'est ici encore créé, sauf un tableau de référents. Il n'y a donc, à ce stade, pas de tentatives, qui seraient rejetées à la compilation, de création d'objet d'une classe abstraite (la superclasse ici). Les référents pointeront vers des objets, qui eux, lors de leur création, seront de type dynamique, type correspondant à une des sous-classes héritant de la superclasse. L'opération de création d'objets se fera à partir des sous-classes, bien concrètes cette fois. Chaque objet sera donc typé statiquement par son référent superclasse, et typé dynamiquement par sa sous-classe.

Il n'y aurait aucun problème à approfondir l'arbre d'héritage, et à laisser, nonobstant le lien client-serveur entre la classe cliente et la superclasse, la manière dont le message serait exécuté être définie bien plus bas dans l'héritage. Plusieurs couches de classes abstraites peuvent précéder l'arrivée, tout en bas, des classes concrètes. Rappelez-vous qu'à entendre certains gourous de l'OO, les superclasses ne devraient être, en principe, qu'abstraites. Principe discutable, nullement contraint par la syntaxe des langages de programmation, mais dont, néanmoins, on perçoit la pertinence en jetant un simple coup d'œil au monde qui nous entoure.

Exercices

Exercice 13.1

Expliquez pourquoi la présence d'une méthode abstraite dans une classe interdit naturellement la création d'objets issus de cette classe.

Exercice 13.2

Justifiez pourquoi l'absence de concrétisation dans une sous-classe d'une méthode définie abstraite dans sa superclasse oblige, sans autre forme de procédé, la sous-classe à devenir abstraite.

Exercice 13.3

Expliquez pourquoi C++ ne recourt pas à l'utilisation du mot-clé `abstract` pour définir une classe abstraite.

Exercice 13.4

Justifiez pourquoi C# n'impose pas de définir une méthode abstraite `virtual` pour pouvoir la redéfinir.

Exercice 13.5

Réalisez un code dans un des trois langages, dans lequel une superclasse `MoyenDeTransport` contiendrait une méthode `consomme()` abstraite, qu'il faudrait redéfinir dans les trois sous-classes `Voiture`, `Moto`, `Camion`.

Exercice 13.6

Réalisez un code dans un des trois langages, dans lequel une superclasse `ExpressionAlgebrique` contiendrait un `String` comme « 2 + 5 » ou « 7 * 2 » ou « 25 : 5 » et une méthode abstraite `evalue()`, et trois sous-classes `Addition`, `Multiplication`, `Division`, code qui redéfinirait cette méthode selon que l'expression mathématique en question serait une addition, une multiplication ou une division.

Exercice 13.7

Retrouvez ce qu'écrirait à l'écran le code Java suivant. Dessinez également le diagramme de classe UML correspondant.

```
abstract class Militaire {
  private int age;
  private String nationalite;
  private int QI;

  public Militaire(int age, String nationalite, int QI) {
    this.age = age;
    this.nationalite = nationalite;
    this.QI = QI;
  }
  abstract public void partirEnManoeuvre();
  public void deserter() {
    System.out.println("Salut les cocos");
  }
  public void executer() {
    System.out.println("A vos ordres chef");
  }
  public int getQI() {
    return QI;
  }
}
abstract class Plouc extends Militaire {
  public Plouc(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  abstract public void partirEnManoeuvre();
```

```
}
abstract class Grade extends Militaire {
  public Grade(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void commander (Militaire unTroufion) {
    unTroufion.executer();
  }
  abstract public void partirEnManoeuvre();
}
class Colonel extends Grade {
  private Plouc[] mesTroufions;

  public Colonel(int age, String nationalite, int QI, Militaire[] mesTroufions) {
    super(age,nationalite,QI);
    this.mesTroufions = new Plouc[4];
    for (int i=0; i<4; i++) {
      this.mesTroufions[i] = (Plouc)mesTroufions[i];
    }
  }
  public void partirEnManoeuvre() {
    for (int i=0; i<4; i++) {
      commander(mesTroufions[i]);
      System.out.println();
    }
  }
}
class General extends Grade {
  private Colonel monColonel;

  public General(int age, String nationalite, int QI, Colonel monColonel) {
    super(age,nationalite,QI);
    this.monColonel = monColonel;
  }
  public void partirEnManoeuvre() {
    commander(monColonel);
  }
}
class Abruti extends Plouc {
  public Abruti(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void partirEnManoeuvre() {
    System.out.println("C'est super");
  }
}
class TireAuFlanc extends Plouc {
  public TireAuFlanc(int age, String nationalite, int QI) {
    super(age,nationalite,QI);
  }
  public void executer() {
    super.executer();
    deserter();
    System.out.println("et merde");
```

```
  }
  public void partirEnManoeuvre() {
    if (getQI() < 5)
      System.out.println("vivement le bar");
    else
      System.out.println("vivement la bibliotheque");
  }
}
public class Armee {
  public static void main(String[] args) {
    Militaire[] unRegiment = new Militaire[6];
    unRegiment[0] = new Abruti(20, "Belge", 1);
    unRegiment[1] = new Abruti(23, "Francais", 8);
    unRegiment[2] = new TireAuFlanc(20, "Italien", 1);
    unRegiment[3] = new TireAuFlanc(25, "Italien", 8);
    unRegiment[4] = new Colonel(50, "Belge", 2, (Militaire[])unRegiment);
    unRegiment[5] = new General(60, "Francais", 2, (Colonel)unRegiment[4]);
    for (int i=0; i<6; i++)
      unRegiment[i].partirEnManoeuvre();
  }
}
```

Exercice 13.8

Retrouvez ce qu'écrirait à l'écran le code C# suivant :

```
using System;
abstract class Animaux {
  private int monAge;
  private String monNom;

  public Animaux(int age, String nom) {
    monAge = age;
    monNom = nom;
  }
  public virtual void crierSuivantLesAges() {
    if (monAge <= 2) {
      Console.WriteLine("je fais un petit");
    }
    else {
      Console.WriteLine("je fais un gros");
    }
  }
  protected virtual void crierAToutAge() {
    Console.WriteLine("chuuuuut");
  }
  abstract public void dormir();
}
class Fermier {
  private int nbreAnimaux;
  private Animaux[] mesAnimaux;
```

```
  public Fermier(Animaux[] lesAnimaux, int nombre) {
    mesAnimaux = lesAnimaux;
    nbreAnimaux = nombre;
  }
  public void jeFaisMaTourneeDuSoir() {
    for (int i=0; i<nbreAnimaux; i++) {
      mesAnimaux[i].dormir();
      Console.WriteLine();
    }
  }
}
class Cochon : Animaux {
  public Cochon(int age, String nom) : base(age,nom) {}
  protected override void crierAToutAge() {
    Console.WriteLine("groin groin");
  }
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors en fermant les yeux");
  }
}
class Poule : Animaux {
  public Poule(int age, String nom) : base(age,nom) {}
  protected override void crierAToutAge() {
    Console.WriteLine("cot cot");
  }
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors sur mon perchoir");
  }
}
class Taureau : Animaux {
  public Taureau(int age, String nom) : base(age, nom) {}
  public override void dormir() {
    crierSuivantLesAges();
    crierAToutAge();
    Console.WriteLine("et je m'endors a côte de ma vache");
  }
}
public class Ferme {
  public static void Main() {
    Animaux[] laFamilleRoyale = new Animaux[10];
    Fermier AlphonseII = new Fermier(laFamilleRoyale,3);
    laFamilleRoyale[0] = new Cochon(1,"Phil");
    laFamilleRoyale[1] = new Poule(1,"Astrud");
    laFamilleRoyale[2] = new Taureau(3,"Lorenzio");
    AlphonseII.jeFaisMaTourneeDuSoir();
  }
}
```

Exercice 13.9

Retrouvez ce qu'écrirait à l'écran le code Java suivant. Ce code utilise la classe `Vector` qui est un tableau dynamique dont nous utilisons les méthodes suivantes :

- `size()` pour obtenir la taille du tableau,
- `elementAt(i)` pour extraire le énième élément du tableau (attention, cette méthode renvoie un `object`, et il est nécessaire d'utiliser le `casting` pour récupérer le vrai type),
- `addElement()` rajoute un nouvel élément en queue du tableau.

Dessinez également le diagramme de classe UML correspondant.

```
import java.util.*;
public class UneSerre {
  Vector maSerre = new Vector();
  Jardinier j;

  public static void main(String[] args) {
    new UneSerre();
  }
  public UneSerre() {
    maSerre.addElement(new Bananier(3,150));
    maSerre.addElement(new Olivier(5,300));
    maSerre.addElement(new Magnolia());

    j=new Jardinier(maSerre);

    j.occupeToiDesPlantes(0,2,1);
    j.occupeToiDesPlantes(3,3,4);
    j.occupeToiDesPlantes(4,0,1);
  }
}
class Jardinier {
  Vector maSerre;

  public Jardinier(Vector maSerre) {
    this.maSerre = maSerre;
  }
  public void occupeToiDesPlantes(int periode, int lumiere, int humidite) {
    for (int k=0; k<maSerre.size();k++) {
      Vegetal v = (Vegetal)maSerre.elementAt(k);
      v.jeGrandis(lumiere,humidite,periode);
    }
  }
}
abstract class Vegetal {
  protected int age;
  private int hauteur;
  private int etat;
  protected static String[] humidite=   {   "je me desseche !",
                                            "plus d'eau", "ok pour l'eau",
```

```
                                        "Mes racines pourrissent",
                                        "blou bloup"
                                    };
  protected static String[] lumiere=    {   "more light please",
                                        "lumosite parfaite",
                                        "je suis aveuglee",
                                        "je crame!!"
                                    };
  public Vegetal(int a, int h) {
    age = a;
    hauteur = h;
  }
  abstract public void jeGrandis(int lumiere, int humidite, int periode);
}
abstract class Fruitier extends Vegetal {
  Fruitier(int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFruits() {
    System.out.println(" .... et je donne des fruits");
  }
}
class Bananier extends Vegetal {
  Bananier (int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFruits() {
    System.out.println(" .... et je donne de bonnes bananes");
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("le bananier dit: ");
    if (l>1) l=1;
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==3 && age>3 && age>10) jeDonneDesFruits();
    age++;
  }
}
class Olivier extends Fruitier {
  Olivier(int a, int h) {
    super(a,h);
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("l'olivier dit: ");
    if (l>1) l=1;
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==1 && age>3 && age>10) jeDonneDesFruits();
    age++;
  }
}
```

```
abstract class Plante extends Vegetal {
  Plante(int a, int h) {
    super(a,h);
  }
  public void jeDonneDesFleurs() {
    System.out.println("je donne des jolies fleurs");
  }
}
class Magnolia extends Plante {
  Magnolia() {
    super(0,0);
  }
  Magnolia(int a, int h) {
    super(a,h);
  }
  public void jeGrandis(int l, int h, int periode) {
    System.out.println("le magnolia dit: ");
    System.out.println(lumiere[l]+" ");
    System.out.println(humidite[h]);
    if (periode==1 && age>1 && age>6) jeDonneDesFleurs();
    age++;
  }
  public void jeDonneDesFleurs() {
    System.out.println("Les Magnolias fleurissent");
  }
}
```

Exercice 13.10

Retrouvez ce qu'écrirait à l'écran le code C++ suivant. Dessinez également le diagramme de classe UML correspondant.

```
#include "stdafx.h"
#include "iostream.h"

class Instrument {
  public:
    Instrument() {}
    virtual void joue() = 0;
};
class Guitare : public Instrument {
  public:
    void joue() {
      cout << "je fais ding ding" << endl;
    }
};
class Trompette : public Instrument {
  public:
    void joue() {
      cout << "je fais pouet pouet" << endl;
    }
```

```
};
class Tambour : public Instrument {
  public:
    void joue() {
      cout << "je fais badaboum" << endl;
    }
};
class Musicien {
  private:
    Instrument* monInstrument;
  public:
    Musicien(Instrument *monInstrument) {
      this->monInstrument = monInstrument;
    }
    void joue() {
      monInstrument->joue();
    }
};
class Orchestre {
  private:
    Musicien *lesMusiciens[3];
    int nombreMusicien;
  public:
    Orchestre(Musicien* lesMusiciens[3]) {
      for (int i=0; i<3; i++) {
        this->lesMusiciens[i] = lesMusiciens[i];
      }
    }
    void joue() {
      for (int i=0; i<3; i++) {
        lesMusiciens[i]->joue();
      }
    }
};
int main() {
  Instrument* lesInstruments[10];
  Musicien* lesMusiciens[8];

  lesInstruments[0] = new Guitare();
  lesInstruments[1] = new Guitare();
  lesInstruments[2] = new Trompette();
  lesInstruments[3] = new Trompette();
  lesInstruments[4] = new Guitare();
  lesInstruments[5] = new Tambour();
  lesInstruments[6] = new Tambour();
  lesInstruments[7] = new Tambour();
  lesInstruments[8] = new Trompette();
  lesInstruments[9] = new Guitare();
  lesMusiciens[0]   = new Musicien(lesInstruments[2]);
```

```
  lesMusiciens[1]   = new Musicien(lesInstruments[5]);
  lesMusiciens[2]   = new Musicien(lesInstruments[0]);
  lesMusiciens[3]   = new Musicien(lesInstruments[9]);
  lesMusiciens[4]   = new Musicien(lesInstruments[9]);
  lesMusiciens[5]   = new Musicien(lesInstruments[4]);
  lesMusiciens[6]   = new Musicien(lesInstruments[2]);
  lesMusiciens[7]   = new Musicien(lesInstruments[1]);

  Musicien* lesMusiciensDOrchestre[3];
  lesMusiciensDOrchestre[0] = lesMusiciens[2];
  lesMusiciensDOrchestre[1] = lesMusiciens[5];
  lesMusiciensDOrchestre[2] = lesMusiciens[3];

  Orchestre *unOrchestre    = new Orchestre(lesMusiciensDOrchestre);
  unOrchestre->joue();

  return 0;
}
```

Exercice 13.11

Corrigez le code des classes `A` et `B` pour que le code compile et que son exécution affiche à l'écran : « 1, 2, trois, quatre ». Expliquez et corrigez les erreurs à même le code.

```
abstract class A extends Object {
    private int a,b ;
    private String c ;

    public A(int a,int b, String c) {
       super() ;
       this.a=a ;
       this.b=b ;
       this.c=c ;
    }

    public abstract void decrisToi() ;

}


class B extends A {
   private String d ;

    public B(int a, int b, String c, String d)  {
       this.a=a ;
       this.b=b ;
       this.c=c ;
       this.d=d ;
    }
}
```

```
public class Correction1 {
    public static void main(String[] args)  {

            B b=new B(1,2,"trois","quatre") ;
            b.decrisToi() ;
    }
}
```

Exercice 13.12

Dans le code C++ qui suit, seuls la classe `C` et le `main` contiennent des erreurs. Supprimez-les sans altérer les fonctionnalités du code et indiquez ce que ce code écrirait dans sa version correcte.

```
#include "stdafx.h"
#include "iostream.h"

class A {
private:
        int a ;

public:
        A(int a) {
               this->a=a ;
        }
               int getA() {
               return a ;
        }

        virtual void action(){
               cout << "je travaille" << endl ;
        }
        virtual void actionA() = 0 ;
        void actionA2() {
               cout << "je travaille pour A et A" << endl ;
        }
} ;

class B : A {
private:
        int a,b ;

public:
        B(int a, int b): A(a) {
               this->a=a ;
               this->b = b ;
        }

        void action(){
               actionA() ;
               cout << "je ne travaille pas" << endl ;
        }
               virtual void actionA(){
```

```
                cout << "je travaille pour A" << endl ;
        }
        void actionA2() {
                cout << "je travaille pour A et A" << endl ;
        }
} ;

class C : public A,B {
public:
        C(int a, int b):A(a),B(a,b)
        {}

        void action() {
                cout << getA() << "je travaille pour C"<<endl ;
        }

        void actionA(){
                cout << "je travaille pour C" << endl ;
        }
} ;

int main()
{
        A *a = new A(1) ;
        A *c1 = new C(1,2) ;
        B c2 ;
        c1->action() ;
        c2.action() ;
        return 0 ;
}
```