Upcasting

- Dans le code précédent, une référence `Point` désigne un objet `PointNomme` : cette affectation est autorisée car le compilateur sait qu'un `PointNomme` *est un* `Point` avec des caractéristiques supplémentaires.
- Ce mécanisme de conversion d'un objet d'une classe dérivée (`PointNomme`) en un objet d'une classe parente (`Point` ou `Object`) s'appelle *upcasting* (on « remonte » dans l'arbre d'héritage).
- Cette conversion est toujours sûre puisque l'on part d'un type particulier pour aller vers un type plus général (on est certain que l'objet récepteur aura les méthodes et attributs de l'objet de départ) : c'est pour cette raison que le compilateur réalise implicitement cette conversion lorsque cela est nécessaire.
- Cela signifie notamment qu'on pourra toujours affecter n'importe quel objet à une référence `Object`.

Downcasting

- Dans le code suivant, par contre :

```
Point point = new PointNomme("A", 1, 2);
PointNomme pointNomme;
...
pointNomme = point;    // Erreur !
```

- Le compilateur interdira la dernière affectation car un `Point` n'est pas *nécessairement* un `PointNomme`. Pourtant, cette affectation aurait du sens lors de l'exécution puisque l'on sait que, ici, `point` désigne, en fait, un `PointNomme`...
- Pour résoudre ce problème, on peut indiquer au compilateur que l'on veut transtyper `point` en `PointNomme` : il s'agit alors d'un *downcasting* puisque l'on « descend » dans l'arbre d'héritage :

```
pointNomme = (PointNomme)p;
```

Remarques sur le downcasting

- Le downcasting présente quelques risques car l'objet récepteur contient éventuellement plus de méthodes et d'attributs que l'objet de départ. Ainsi, le compilateur ne se plaindrait pas du code suivant :

```
Point point = new Point(10, 20);
PointNomme pointNomme;
...
pointNomme = (PointNomme)point;
System.out.println(pointNomme.getNom());  // Erreur à l'exécution !
```

- Pourtant, le code échouera à l'exécution avec `ClassCastException` car `pointNomme` n'était pas une instance de `PointNomme` mais une instance de `Point` et qu'il n'existe donc pas de méthode `getNom()` pour cette instance.

Remarques sur le downcasting

- On pourrait bien sûr utiliser l'instruction *instanceof* avant d'appeler une méthode définie dans une classe dérivée :

```
Point point = new Point(10, 20);
PointNomme pointNomme;
...
if (point instanceof PointNomme) { // Ce code ne sera donc pas exécuté ici
  pointNomme = (PointNomme)point;
  System.out.println(pointNomme.getNom());
}
```

- Mais, en pratique, il est préférable d'éviter le downcasting car il produit du code lourd et peu fiable.
- Ce n'est malheureusement pas toujours possible (cf. redéfinition de la méthode `clone()` et son utilisation...).

Classes et méthodes abstraites

- Dans certains cas, on peut ne rien savoir d'une méthode virtuelle (on ne veut la créer que pour qu'elle apparaisse dans l'arbre d'héritage et pour faire jouer ensuite le polymorphisme).
- On ne sait pas quoi mettre dans la méthode `dessine()` d'une classe `Figure`, par exemple : on sait juste que toutes les figures devront proposer une méthode`dessine()`...
- En ce cas, on peut déclarer une méthode sans corps, c'est-à-dire une *méthode abstraite* (C++ les appelle méthodes virtuelles pures) avec le mot-clé `abstract`.

Classes et méthodes abstraites

- Une classe qui contient *au moins une* méthode abstraite est elle-même abstraite (on ne peut pas créer d'instance de cette classe) et doit donc elle-même être déclarée avec le mot-clé `abstract`.
- Une classe qui dérive d'une classe abstraite et qui ne redéfinit pas toutes ses méthodes abstraites reste abstraite.
- Autrement dit, une classe ne peut être *concrète* (instanciable) que si *toutes* ses méthodes (héritées ou non) ne sont plus abstraites.

Remarques

- Une méthode abstraite ne doit pas être déclarée privée car, sinon, on ne pourra jamais la redéfinir...
- Une méthode abstraite n'est qu'une méthode virtuelle particulière et se redéfinit donc de la même manière.

Classes et méthodes abstraites

Une classe abstraite ne peut évidemment pas être instanciée (avec `new`), mais on peut avoir des références vers une classe abstraite : le polymorphisme jouera alors son rôle lorsque cette référence pointera vers une instance d'une classe concrète qui hérite de cette classe abstraite :

```
class Point { ... }

abstract class Figure {
  public abstract void dessine();
}

class Cercle extends Figure {
  private Point  centre;
  private double rayon;
  public Cercle(Point centre, double rayon) {
    this.centre = centre;
    this.rayon = rayon;
  }
  @Override
  public void dessine() { /*  Code de traçage d'un cercle */ }
}

class Triangle extends Figure {
  private Point sommet1, sommet2, sommet3;
  public Triangle(Point s1, Point s2, Point s3) {
    sommet1 = s1; sommet2 = s2; sommet3 = s3;
  }
  @Override
  public void dessine() { /* Code de traçage d'un triangle */ }
}
```


Classes et méthodes abstraites

Exemple d'utilisation du polymorphisme

```
public class Programme {

  public static void main(String[] args) {

    Figure[] tableau = new Figure[3];
    tableau[0] = new Cercle(new Point(0,0), 6.7);
    tableau[1] = new Triangle(new Point(0,0), new Point(0,2), new Point(1,2));
    tableau[2] = new Cercle(new Point(1,1), 5.6);

    for (Figure fig : tableau) {
      fig.dessine();
    }
  }
}
```

Classes et méthodes abstraites

- Autre cas d'utilisation des méthodes abstraites : la classe de base sait comment effectuer une action, mais cette action dépend d'autres actions effectuées par les classes dérivées (qui n'existent peut-être pas encore)...
- En ce cas, la méthode que sait effectuer la classe de base ne doit pas être abstraite (puisque l'on connait le code à effectuer). En revanche, elle doit définir les méthodes abstraites attendues pour effectuer ce traitement :

```
abstract class Figure {

  public void moveTo(int x, int y) { ... }

  public void deplace(int deltaX, int deltaY) { // Non abstraite
     efface();
     moveTo(deltaX, deltaY);
     affiche();
  }

  public abstract void efface();
  public abstract void affiche();

}
```


Classes et méthodes abstraites

```
class Cercle extends Figure {
 ...
 @Override
 public void efface()  { /* Effacement du cercle */ }
 @Override
 public void affiche() { /* Affichage du cercle */ }
 ...
}

class Triangle extends Figure {
 ...
 @Override
 public void efface()  { /* Effacement du triangle */ }
 @Override
 public void affiche() { /* Affichage du triangle */ }
 ...
}
```

Commentaires

- Lorsqu'un objet `Cercle` ou `Triangle` appelle `deplace(...)`, c'est donc la méthode de `Figure` qui est appelée puisqu'elle n'a pas été redéfinie dans `Cercle` ni dans `Triangle` (premier « dispatch de méthode »).
- Cette méthode appelle à son tour les méthodes `efface()` et `affiche()` de la classe de l'objet cible, c'est-à-dire de `Cercle` ou de `Triangle` (ou de n'importe quelle sous-classe concrète de `Figure` puisque ces deux méthodes *doivent* être redéfinies). On a donc un « double dispatch » : un qui « remonte », un qui « redescend ».

Le mot-clé `final`

On a déjà vu que *final* permettait de définir des constantes. Mais il a également deux autres rôles liés à l'héritage :

- Une *méthode* marquée `final` ne peut pas être redéfinie. Ceci permet de produire un code plus efficace car le compilateur peut remplacer l'appel de la méthode par un code « inline » puisqu'il sait qu'elle ne pourra pas être redéfinie par une sous-classe. La liaison du code de la méthode est donc faite lors de la compilation au lieu de l'être dynamiquement à l'exécution.
  Du point de vue du programmeur, une méthode `final` garantit que celle-ci aura toujours le comportement voulu puisqu'elle ne pourra pas être redéfinie par un code encore inconnu.
- Une *classe* marquée `final` ne peut pas être héritée. Toutes ses méthodes sont donc implicitement considérées comme `final`.

Remarque

Une classe ne peut être à la fois `abstract` et `final` puisqu'une classe abstraite est, par définition, incomplète et repose sur des classes filles pour l'implémentation.

Intérêt des classes abstraites (rappel)

- Permettent de *factoriser le comportement* de plusieurs classes, *même si on n'est pas encore capable de le coder*.
- Permettent de *catégoriser* les objets (figures, figures fermées, figures ouvertes, etc.) : on créee une hiérarchie, même si sa racine est un concept général, qui ne correspond à aucune instance possible.
- Permettent au compilateur de vérifier la définition des méthodes abstraites héritées.

Interfaces

- Le mot clé *interface* permet de totalement séparer une classe de son implémentation. On peut indiquer ce que la classe devra faire, sans expliquer comment.
- Les interfaces ressemblent beaucoup aux classes abstraites mais :
  - elles ne peuvent pas contenir de variables d'instances ;
  - toutes leurs méthodes sont abstraites.
- En pratique, une interface ne peut donc faire aucune hypothèse sur la façon dont elle sera implémentée.
- Une classe peut implémenter *une ou plusieurs* interfaces, ce qui a pour effet d'établir un *contrat* entre cette classe et les interfaces qu'elle implémente : la classe *doit* fournir une implémentation pour chacune des méthodes indiquées dans les interfaces qu'elle s'engage à implémenter.
- Si une classe n'implémente pas toutes les méthodes de l'interface, la classe est abstraite et doit donc être marquée `abstract` : on a donc une *implémentation partielle* de l'interface.

Interfaces et polymorphisme d'interface

- Les interfaces étant indépendantes des hiérarchies de classes, elles permettent de réaliser un *polymorphisme d'interface* (ou de *signature*).
- On ne peut évidemment pas créer d'instance d'une interface mais il est possible de déclarer une référence vers une interface : cette référence sera alors *compatible avec toute classe implémentant l'interface* (un tableau de `Comparable` peut contenir n'importe quel objet implémentant `Comparable`).

Définition d'une interface

Une interface est définie en utilisant le mot-clé *interface* au lieu de *class* :

```
[public] interface NomInterface {
   type1 nomMeth1(liste_params);
   type2 nomMeth2(liste_params);
   ...
   type3 varFinale1 = valeur1;
   type4 varFinale2 = valeur2;
   ...
}
```

Remarques

- Les variables sont implicitement `final` : elles ne peuvent donc pas être modifiées dans la classe qui implémente l'interface (ce sont donc des constantes...)
- Les constantes et les méthodes sont implicitement abstaites et publiques.

Implémentation d'une interface

- Une classe implémente une interface à l'aide du mot-clé *implements*. La forme générale est la suivante :

```
[public] class NomClasse [extends ClasseBase] [implements interface1  [, interface2...] {
   ...
}
```

- Les méthodes de l'interface implémentées par la classe doivent être déclarées `public` et correspondre à la signature indiquée dans l'interface.
- Une classe qui implémente une interface peut bien sûr définir ses propres méthodes et variables d'instance.

Implémentation d'une interface : exemple

```
interface Callback {
  void callback(int param);
}

class Client implements Callback {
  public void callback(int param) {
     System.out.println("Appel de callback avec " + param);
  }

  public void autreMethode() { ... }
}

class AutreClient implements Callback {
    public void callback(int param) {
     System.out.println("Autre appel de callback avec " + param);
}

class Programme {
  public static void main(String[] args) {
    Callback c = new Client();
    AutreClient ac = new AutreClient();

    c.callback(42);

    c = ac;    // c désigne maintenant un AutreClient
    c.callback(42);
  }
}
```


Variables des interfaces

- Les interfaces permettent également d'importer des constantes dans des classes : il suffit de créer une interface ne contenant que des variables initialisées avec les valeurs voulues.
- La classe qui « implémentera » cette interface disposera alors de ces variables comme des constantes.
- Exemple :

```
interface ConstantesPartagees {
   int NON = 0;
   int OUI = 1;
   int PEUT_ETRE = 2;
   int JAMAIS = 3;
}

class Programme implements ConstantesPartagees {
  // Les constantes NON, OUI, PEUT_ETRE et JAMAIS sont visibles
}
```

Extension d'interfaces

Une interface peut hériter d'une *ou plusieurs* interfaces :

```
interface A {
   void meth1();
   void meth2();
}

interface B {
   void meth3();
}

interface C extends A, B {
   void meth4();
}

class MaClasse implements C {
  // doit implémenter meth1(), meth2(),  meth3() et meth(4) comme des méthodes publiques
}
```


Utilisation des interfaces

- Le paquetage *java.lang* définit un certain nombre d'interfaces : `Cloneable`, `Comparable<T>`, `Iterable<T>`, notamment.
- Vos classes doivent implémenter certaines interfaces pour être utilisables dans certains contextes :
  - une classe doit implémenter *Cloneable* pour indiquer qu'elle redéfinit la méthode `clone()`.
  - une classe doit implémenter *Comparable<T>* pour indiquer que ses objets sont ordonnés et peuvent donc être comparés. Elle doit alors fournir une méthode *compareTo()*. `T` est le type des objets auxquels les objets de la classe pourront être comparés.
  - une classe doit implémenter *Iterable<T>* pour que ses objets puissent être parcourus par une boucle « foreach ». Elle doit alors fournir une méthode *iterator()* qui renvoie un objet *Iterator<T>* (`T` est le type des éléments).

Utilisation des interfaces

- Le paquetage *java.io* définit l'interface *Serializable*, qui n'impose aucune implémentation de méthode (c'est une interface « marqueur »). Pour être sérialisable, une classe et *tous ses champs* doivent être sérialisables (voir plus loin).
- Le paquetage *java.util* définit plusieurs interfaces (*Collection<T>*, *List<E>*, *Map<K,V>*, *Set<E>*, etc.) permettant de regrouper les fonctionnalités des différentes collections qu'il définit.

Le framework des collections

Les interfaces collections

- La plupart des classes formant le framework des collections Java sont définies dans le paquetage *java.util*.
- Ce framework est *générique* : les classes et les interfaces ont donc des paramètres de type (avant 1.5, ce framework ne gérait que des collections d'`Object`).
- Les 4 interfaces collection de base de `java.util` sont *Set<T>*, *List<T>*, *Queue<T>* et *Map<K,V>*. Chacune de ces interfaces décrit un type d'organisation des données d'une collection (et les méthodes possibles).
- `Set<T>`, `List<T>` et `Queue<T>` héritent de *Collection<T>* qui, elle-même, dérive de *Iterable<T>*. Les méthodes de ces deux interfaces sont donc disponibles pour les ensembles, les listes et les files d'attente.
- Les interfaces *SortedSet<T>* et *SortedMap<K,V>* sont des versions spécialisées des interfaces correspondantes (elles ajoutent des méthodes d'ordonnancement des objets dans ces types de collections).

Les classes collections

- `Set<T>` est implémentée par les classes *HashSet<T>*, *LinkedHashSet<T>* et *EnumSet<T>*.
- `SortedSet<T>` est implémentée par la classe *TreeSet<T>*.
- `List<T>` est implémentée par les classes *Vector<T>*, *Stack<T>*, *ArrayList<T>* et *LinkedList<T>*.
- `Queue<T>` est implémentée par *PriorityQueue<T>* et *LinkedList<T>*.
- `Deque<T>` (qui hérite de `Queue<T>`) est implémentée par *LinkedList<T>* et *ArrayDeque<T>*.
- `Map<K,V>` est implémentée par *Hashtable<K,V>*, *HashMap<K,V>*, *LinkedHashMap<K,V>*, *WeakHashMap<K,V>*, *IdentityHashMap<K,V>*, *EnumMap<K extends Enum<K>, V>*.
- `SortedMap<K,V>` est implémentée par *TreeMap<K,V>*.
- `NavigableMap<K,V>` est implémentée par *TreeMap<K,V>*

Les classes collections

- La classe `LinkedList<T>` implémente les interfaces `List<T>`, `Queue<T>` et `Deque<T>`. En fonction des besoins, un objet de cette classe peut donc être vu comme une liste, une file d'attente ou une file bidirectionnelle (il dispose de toutes les méthodes adéquates).
- Tout objet d'une classe implémentant `Collection<T>` (liste, ensemble ou file d'attente) peut être référencé par une variable de type `Collection<T>`.
- Une `Map<K, V>` n'est pas une `Collection<T>`, mais on peut obtenir un ensemble à partir d'une map et certaines méthodes des maps renvoient des `Collection<T>`...
- Une `Collection<T>` implémente `Iterable<T>`, ce qui signifie qu'elle dispose d'une méthode `iterator()` renvoyant un objet `Iterator<T>` et qu'on peut la parcourir avec une boucle « foreach ».
- Les objets `List<T>` disposent également d'une méthode `listIterator()` qui renvoient un `ListIterator<T>` permettant de parcourir la liste dans les deux sens.
- `listIterator()` est surchargée pour ne traiter qu'une partie de la liste.

Les principales classes

- `HashSet<T>` implémente un ensemble en utilisant un `HashMap` : les opérations d'écriture et de lecture se font donc en `temps constant`. Par contre, l'ordre des éléments n'est pas défini.
- `TreeSet<T>` implémente un ensemble dans lequel les objets sont triés par ordre croissant. Par conséquent, l'itérateur obtenu à partir d'un `TreeSet<T>` renvoie les objets par ordre croissant.
- `Vector<T>` implémente une liste sous forme de tableau dont la taille varie en fonction des besoins. Les objets sont lus et écrits au moyen d'un indice. Ils peuvent aussi être obtenus via un itérateur. Un `Vector<T>` est synchronisé, ce qui permet d'assurer un comportement cohérent en cas d'accès par plusieurs threads.
- `ArrayList<T>` implémente une liste sous forme de tableau dont la taille varie en fonction des besoins et qui peut également être accédé comme une liste chaînée. Elle fournit les mêmes fonctionnalités que `Vector<T>` mais n'est pas synchronisée (préférable dans un environnement mono-thread).
- `Stack<T>` hérite de `Vector<T>` et ajoute les méthodes propres à une pile.

Les principales classes

- `LinkedList<T>` implémente une liste chaînée. Elle peut également être utilisée comme une pile ou une file d'attente.
- `ArrayDeque<T>` implémente un tableau de taille dynamique sous forme de file d'attente bidirectionnelle.
- `PriorityQueue<T>` implémente une file d'attente à priorité où chaque objet est ordonné par ordre croissant de la tête à la queue. Cet ordre est déterminé soit par un objet `Comparator<T>` fourni au constructeur, soit par la méthode `compareTo()`.
- `Hashtable<K,V>` implémente une table de hachage de clés de type `K` et de valeurs de type `V`, où chaque clé doit être non `null`. La classe des clés doit implémenter les méthodes `hashcode()` et `equals()`. Comme `Vector<T>`, cette classe est synchronisée.
- `HashMap<K,V>` implémente une table de hachage de clés de type `K` et de valeurs de type `V`. Avec cette classe, on peut stocker des valeurs `null` et une clé peut valoir `null`. Cette classe n'est pas synchronisée.

Les principales classes

- `WeakHashMap<K,V>` : comme `HashMap<K, V>`, sauf qu'une entrée clé/valeur est supprimée dès que la clé n'est plus référencée à l'extérieur de la table. Avec `HashMap`, cette entrée est conservée tant que la clé existe dans la table de hachage, même si elle n'est plus accessible par le programme.
- `IdentityHashMap<K,V>` : comme `HashMap<K,V>`, sauf que les comparaisons pour rechercher une clé ou une valeur comparent les références, pas les objets : deux clés (ou deux valeurs) sont égales si et seulement si ce sont les mêmes.
- `TreeMap<K, V>` implémente une table de hachage triée selon l'ordre croissant des clés.

Conversions tableaux/listes

- La méthode `toArray()` permet d'obtenir un tableau à partir d'une `Collection<T>` :

```
ArrayList<String> noms = new ArrayList<>();
...
String[] donnees = noms.toArray(new String[noms.size()]);
```

- Inversement, la méthode statique `java.util.Arrays.asList()` permet de convertir un `T[]` en `Collection<T>` :

```
String[] personnes = {"Jules", "César", "Cléopatre", "Auguste"};
List<String> listeNoms = Arrays.asList(personnes);
...
ArrayList<String> alNoms = new ArrayList<>(Arrays.asList(personnes));
```


Entrées/Sorties