Programmation Objet avec Java

Grâce aux connaissances que vous avez acquises dans les chapitres précédents vous avez à votre disposition les briques de base nécessaire à l'écriture de vos propres classes Java.

Dans ce chapitre nous étudierons notamment :

- L'anatomie d'une classe et comment déclarer des champs, méthodes et constructeurs.
- Comment instancier des objets à partir de ces classes, comment utiliser l'opérateur "." de déréferencement.
- Des concepts avancés sur les classes comme l'utilisation du mot clé "this", le contrôle d'accès, ...
- L'utilisation des annotations.

Programmation Objet avec Java

Classes

Nous avions précédemment utilisé l'exemple d'un vélo pour illustrer les concepts objets. Voici une possible implémentation complète :

```
public class Bicycle {

    // the Bicycle class has three
    // fields

    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one
    // constructor

    public Bicycle(int startCadence,
          int startSpeed, int
          startGear) {

        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods

    public void setCadence(int newValue) {
            cadence = newValue;
        }

    public void setGear(int newValue) {
            gear = newValue;
        }

    public void applyBrake(int decrement){
            speed -= decrement;
        }

    public void speedUp(int increment) {
            speed += increment;
        }
}
```

Programmation Objet avec Java

Classes

En suivant le paradigme objet, un vélo de montagne, **spécialisation** d'un vélo classique, est **une sous-classe** de la classe `Bicycle`.

Cette classe hérite de tous les champs et méthodes de `Bicycle`, ajoute le champs `seatHeight` et une méthode pour le modifier. Voici sa déclaration :

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass has one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass has one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Programmation Objet avec Java

Classes : déclaration

Une déclaration de classe peut contenir les éléments suivants :

- Des modificateurs tels que `public` ou `private`.
- Le nom de la classe (première lettre en majuscule par convention).
- Le nom **de la** classe parente (super classe) si il y en a une précédé du mot-clé `extends`. Attention : une seule classe parente possible.
- La liste **des** interfaces implémentées, si il y en a, précédée du mot-clé `implements`.
- Le corps de la classe entre ‘`{ }`’.

```
class MyClass extends MySuperClass implements YourInterface1, YourInterface2 {
    //field, constructor, and method declarations
}
```

Rappel : variables d'instances = champs non statiques.

Les déclarations sont composées des 3 éléments suivants :

- Un ou plusieurs modificateurs (tels que `public` ou `private`).
  - `public` : le champs est accessible depuis toutes classe externe.
  - `private` : le champs est accessible uniquement depuis cette classe (invisible à l'extérieur de la classe).

    Nb : de manière à préserver **l'encapsulation**, il est usuel d'utiliser ce modificateur et de fournir des méthodes publiques pour accéder aux champs privés.
- Le type de la variable (`int`, `float`, `boolean`, ..., classes de l'API Java, **vos propres classes**).
- Le nom de la variable.

Programmation Objet avec Java

Classes : déclaration des variables d'instances


```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    (...)
}
```

Un champs privé

Son accesseur

Son modificateur

Les définitions sont composées des 6 éléments suivants :

- Modificateurs (tels que `public` ou `private`).
- Type de retour de la méthode ou `void` si pas de valeur de retour.
- Nom de la méthode.
- Liste des paramètres de la méthode entre parenthèses, ‘`( )`’ si pas de paramètres.
- Une liste d’exceptions.
- Le corps de la méthode entre ‘`{ }`’

```
public double calculateAnswer(double wingSpan, int numberOfEngines, double
length, double grossTons) {
    //do the calculation here
}
```

Programmation Objet avec Java

Classes : surcharge de méthodes

La signature d'une méthode est composée de son nom et du type de ses paramètres.
Ex : `calculateAnswer(double, int, double, double)`

Surcharge : en Java des méthodes sont distinctes si elles disposent de signatures différents.

- Corollaire → des méthodes d'une classe peuvent avoir le même nom si elles déclarent des listes différentes de paramètres.

- Les méthodes surchargées sont différenciées sur le nombre et le type de leurs paramètres.

Attention : le compilateur Java **ne considère pas le type de retour** pour la surcharge, vous ne pouvez donc pas déclarer deux méthodes avec la même signature, même si leur type de retour est différent.

Programmation Objet avec Java
Classes : surcharge de méthodes

Ex : surcharge de la méthode `draw` dans une classe de calligraphie.

- Evite d'avoir à nommer différemment chaque méthode `draw` (`drawString`, `drawInteger`, `drawFloat`, ...).
- Chaque méthodes `draw` dispose d'une signature distincte des autres.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Programmation Objet avec Java

Classes : constructeurs

Une classe possède des constructeurs qui sont invoqués pour créer des instances de cette classe (des objets). **Les initialisations de variables d'instance doivent se trouver dans les constructeurs.**

Une déclaration de constructeur ressemble à une déclaration de méthode, mais elle doit posséder le même nom que sa classe et ne pas avoir de déclaration de type de retour.

Ex. Constructeur de la classe Bicycle :

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

Pour créer une instance de la classe Bicycle on appel un constructeur indirectement par l'opérateur `new` : `Bicycle myBike = new Bicycle(30, 0, 8);`

L'appel au constructeur libère un espace mémoire pour contenir l'objet et initialise ses champs.

Le principe de surcharge des méthodes s'applique de la même façon aux constructeurs d'une classe.

- Bien que Bicycle ne possède qu'un seul constructeur il aurait été possible d'en déclarer d'autres. Par exemple, un constructeur sans arguments comme celui ci :

```
public Bicycle() {
  gear = 1;
  cadence = 10;
  speed = 0;
}
```

Il n'est pas forcément nécessaire de fournir des constructeurs pour vos classes, par exemple si il n'y a pas de champs à initialiser :

- Mais il faut garder alors à l'esprit que le compilateur Java va alors automatiquement fournir un constructeur implicite sans arguments.

- Ce constructeur implicite va essayer d'appeler le constructeur sans argument de sa classe parente → si il n'existe pas, erreur de compil !

Programmation Objet avec Java

Classes : arguments

La déclaration d'une méthode ou d'un constructeur indique le nombre et le type des arguments à fournir lors de ses futures invocations. Il est possible d'utiliser n'importe quel type en argument :

- types primitifs (`int`, `double`, `float`, ...)
- types non-primitifs (classes de l'API, classes utilisateur, tables)

En fonction du type des arguments, lors d'un appel de méthode ou constructeur, on distingue 2 types de passage des valeurs :

- Passage des arguments de type primitifs.
- Passage des arguments de type non-primitifs.

Programmation Objet avec Java

Classes : passage de types primitifs

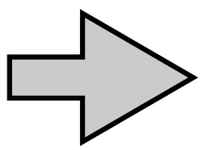
Les arguments de type primitifs sont passés à la méthode **par valeur**.

Cela signifie que les modifications de valeurs apportées aux paramètres par la méthode ne sortent pas de cette méthode.

Corollaire : lorsqu'une méthode termine (`return`), les paramètres sont détruits, de même que les changements qui ont pu leur être appliqué.

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        //invoke passMethod() with x as argument
        passMethod(x);

        // print x to see if its value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

Affiche



```
After invoking passMethod, x = 3
```

Les paramètres de types non-primitifs, comme les objets, sont eux aussi passés aux méthodes et constructeurs **par valeur**.

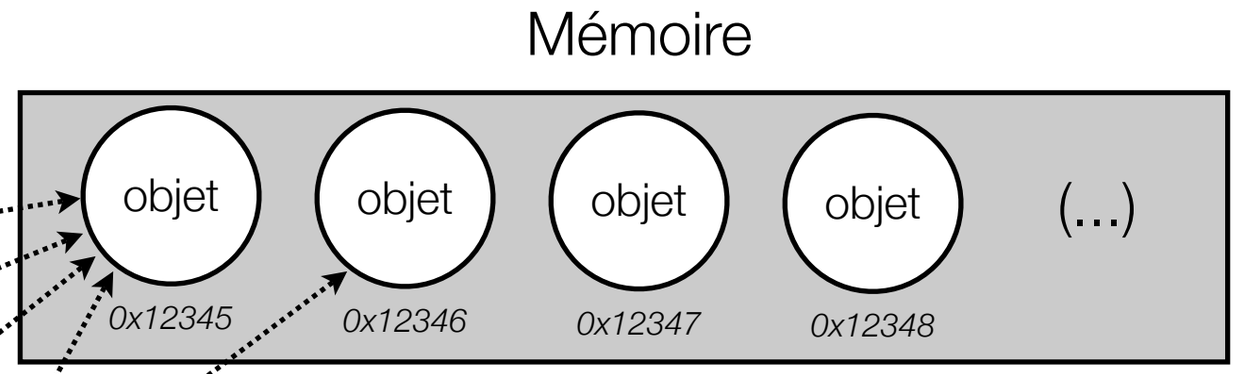
Attention : une variable ou un paramètre de type non-primitif est en fait une référence (une adresse mémoire) vers un objet.

Corollaire : c'est la référence qui est passée par valeur, pas l'objet lui-même.

- Donc si la méthode modifie les champs de l'objet alors **ces modifications perdurent en dehors de la méthode**.
- A contrario, tout modification à la référence (l'adresse mémoire) ne perdurera pas en dehors de la méthode.

Programmation Objet avec Java

Classes : passage de types non-primitifs




```
//Variable      0x12345
Circle myCircle = new Circle();

//Appel de méthode
moveCircle(myCircle, 23, 56)

System.out.println(myCircle);


public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX); //modifie l'objet à l'adresse 0x12345
    circle.setY(circle.getY() + deltaY); //modifie l'objet à l'adresse 0x12345

    //code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

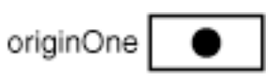
Programmation Objet avec Java

Objets

Un programme Java typique va créer de nombreux objets qui interagissent par invocation de méthodes. Grâce à ces interactions, un programme peut effectuer de nombreuses tâches.

Déclaration d'une variable référençant un objet :

- Ex. `Point originOne;`
- La valeur de la variable reste indéterminée tant que l'opérateur new n'a pas été utilisé pour lui assigner une instance.
- Il faut assigner une instance à la variable avant de l'utiliser.
- Une variable dans cet état, qui ne référence aucun objet, peut être illustré comme suit :



Programmation Objet avec Java

Objets : instanciation

L'opérateur `new` instancie une classe...

- en allouant de la mémoire pour le stockage d'un nouvel objet et retourne une référence vers cet emplacement mémoire (il s'agit en fait d'une adresse même si on ne peut la manipuler en Java).

- puis en appelant le constructeur adéquat de cet objet.

Nb : le terme "instancier" est équivalent à "créer un objet de".

L'opérateur `new` requière que l'on lui indique le constructeur à appeler :
Ex. `Point originOne = new Point(23, 94);`

La référence retournée par l'opérateur `new` n'a pas forcement à être stockée dans une variable, elle peut être utilisée directement.
Ex. `int height = new Rectangle().height;`

Programmation Objet avec Java

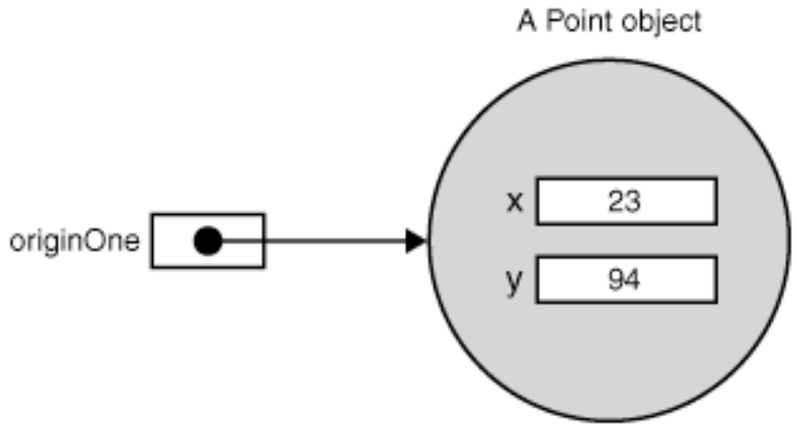
Objets : initialisation

Voici le code de la classe Point :

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

Elle possède un seul constructeur qui sera appelé lors de l'utilisation de new dont le résultat est présenter graphiquement ci-dessous :

```
Point originOne = new Point(23, 94);
```



Programmation Objet avec Java

Objets : utilisation

Référencer les champs d'un objet :

- Depuis l'intérieur de la classe : les champs d'un objet sont accessibles directement par leur nom :
  Ex.

```
Class Rectangle {
  public int width = 0;
  public int height = 0;

  (...)
     System.out.println("Width and height are: " + width + ", " + height);
  (...)
}
```

- Depuis l'extérieur de la classe : il faut utiliser une référence d'objet suivie de l'opérateur " . " :
  Ex.

```
Rectangle rectOne = new Rectangle();
System.out.println("Width of rectOne: " + rectOne.width);
```

Appel des méthodes d'un objet.

Vous pouvez aussi utiliser une référence d'objet pour invoquer une de ses méthodes :

- Depuis l'intérieur de la classe : les méthodes sont accessibles directement par leur nom :

- Depuis l'extérieur de la classe : il faut utiliser une référence d'objet suivie de l'opérateur "`.`" :
  Ex.

```
System.out.println("Area of rectOne: " + rectOne.getArea());
(...)
rectTwo.move(40, 72);
```

Pour les méthodes qui renvoient une valeur, il est possible d'utiliser l'invocation directement dans une expression Java.

Ex. `int areaOfRectangle = new Rectangle(100, 50).getArea();`

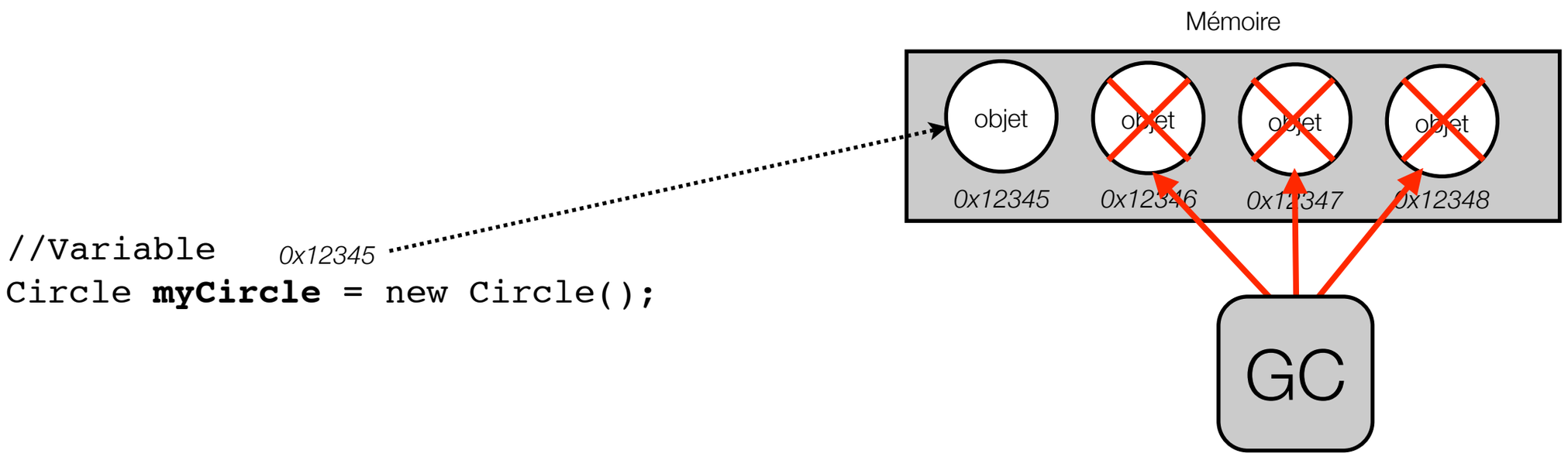
Programmation Objet avec Java

Objets : gestion de la mémoire

Java, langage objet moderne, dispose d'un mécanisme évolué de gestion automatique de la mémoire.

Le but étant de libérer le programmeur de la tâche fastidieuse et source d'erreurs (donc de bugs) de gestion manuelle de la mémoire (comme en C++ par exemple avec les instructions malloc et dealloc).

Ce mécanisme s'appelle le **Garbage Collector** (ou Ramasse Miettes), **il est invisible pour le programmeur** et va libérer automatiquement la mémoire utilisée par des instances qui ne sont plus référencées dans le programme.



Programmation Objet avec Java

Classes : retour de valeurs

Une méthode rend la main (elle retourne) au code qui l'a invoquée lorsque :

- Elle a complétée toutes les instructions de la méthode.
- Ou qu'elle atteint une instruction `return`.
- Ou qu'une exception est levée.

Le type de retour de la méthode est défini dans sa déclaration, la valeur retournée par l'instruction `return` doit s'y conformer.

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

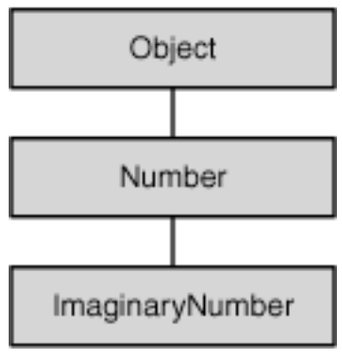
Une méthode déclarée comme `void` n'est pas tenue d'utiliser l'instruction `return` mais elle peut s'en servir malgré tout pour interrompre le flot de contrôle. Dans ce cas, l'instruction s'utilise sans argument.

Programmation Objet avec Java

Classes : retour de valeurs

Si le type de retour déclaré par la méthode est une classe ou une interface alors la méthode doit retourner un objet dont le type **est identique ou un sous-type** du type déclaré.

```
public Number returnANumber() {
    ??
}
```




```
public Number returnANumber() {
    return new Number();
}
```

✓

```
public Number returnANumber() {
    return new Object();
}
```

✗

```
public Number returnANumber() {
    return new ImaginaryNumber();
}
```

✓

Programmation Objet avec Java

Classes : utilisation de ‘this’

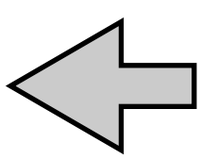
Dans une méthode d’instance ou un constructeur, le mot clé `this` est une **référence à l’objet courant**. C’est à dire l’objet dont les méthodes ou le constructeur est actuellement appelé.

Vous pouvez designer tout membre (champs, méthode) de l’objet courant en utilisant `this`.

Il n’est évidemment **pas accessible depuis une méthode statique** (ou méthode de classe), puisqu’elle n’appartient pas à une instance de la classe désignée.

L’usage le plus courant de `this` est pour accéder à un champ de l’objet qui serait masqué par une déclaration locale à une méthode :

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```



Les paramètres masquent les champs “x” et “y” de l’instance.
On réussi malgré tout à y accéder grâce à **`this`**.

Programmation Objet avec Java

Classes : utilisation de ‘this’

Un autre usage consiste à l’employer, dans un constructeur, pour accéder à un autre constructeur de l’objet. On appel cette technique de l’*invocation explicite de constructeur*.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Invocation de `public Rectangle(int x, int y, int width, int height)`

Programmation Objet avec Java

Classes : contrôle d'accès

Le contrôle d'accès **aux membres d'une classe** (champs, méthodes) est effectué en Java grâce aux modificateurs suivants : `public`, `private`, `protected`.

La table suivante indique, pour l'usage d'un modificateur donné sur un champ d'une classe, quelles entités ont alors accès à ce champs :

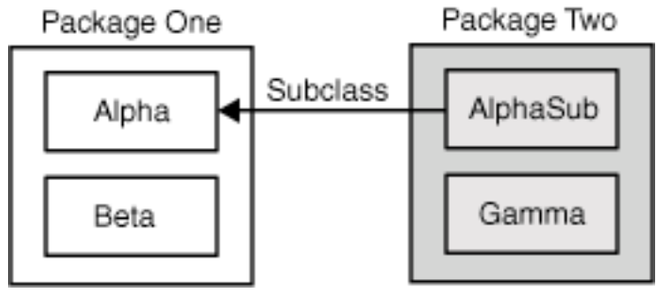
Modificateur	Classe	Package	Sous-classe	Univers
public	O	O	O	O
protected	O	O	O	N
pas de modificateur	O	O	N	N
private	O	N	N	N

Quand vous réutilisez du code externe , le choix de modificateurs effectué par le programmeur affecte votre **visibilité** des membres de ses classes.

Programmation Objet avec Java

Classes : contrôle d'accès

Dans l'exemple suivant on considère 4 classes de manière à déterminer comment l'usage de modificateur impacte la visibilité.



La table suivante montre à quel endroit **les membres de la classe Alpha** sont visibles en fonction des modificateurs de contrôle d'accès qui peuvent leur être appliqués :

Modificateur	Alpha	Beta	Alphasub	Gamma
public	O	O	O	O
protected	O	O	O	N
pas de modificateur	O	O	N	N
private	O	N	N	N

Le contrôle d'accès peut aussi être appliqué directement à une classe. Mais dans ce cas on ne dispose pas des modificateurs `protected` et `private`, on peut seulement apposer le modificateur `public`, ou ne pas en mettre :

- **Modificateur public** : dans ce cas la classe est visible à tout autre classe, quelque-soit son emplacement (même en dehors du package).

- **Pas de modificateur** : par défaut, dans ce cas la classe est visible uniquement aux classes de son package.

Dans tous les cas, la règle à suivre pour le contrôle d'accès de votre propre code si il doit être réutilisé par autrui est la suivante :

1. Choisir le niveau ayant du sens le plus restrictif pour éviter le mauvais usage de vos classes.

2. Eviter autant que possible l'usage de champs public dans vos classes sauf pour les constantes.

Programmation Objet avec Java

Classes : utilisation de ‘static’

Nous avons déjà abordé l’utilisation de static pour la définition de **variables de classes** (ou champs statiques) communes à toutes les instances d’une classe.

Ex. si l’on veut tenir à jour le nombre de vélo qui ont été instanciés on peut utiliser une variable de classe :

```
public class Bicycle{

    private int cadence;
    private int gear;
    private int speed;

    // add a class variable for the number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        // increment number of Bicycles
        ++numberOfBicycles;
    }
    (...)
}
```

Programmation Objet avec Java

Classes : utilisation de ‘static’

Il est aussi possible d’utiliser le mot-clé `static` sur des méthodes de classes.

Ces méthodes peuvent donc être appelées directement sur la classe, sans passer par une instance. Utilisez alors l’opérateur “`.`” de déréférencement sur le nom de la classe :

```
ClassName.methodName(args)
```

Attention : puisqu’elles sont statiques, ces méthodes ne peuvent pas accéder directement aux méthodes ou variables d’instances (non-statiques) de la classe, ni au mot-clé `this` → il faut passer par une instance (fournie par ex. en argument).

Un usage basique des méthodes statiques consiste à les utiliser pour accéder aux champs statiques d’une classe. Ex :

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Programmation Objet avec Java

Classes : ‘static’ + ‘final’ = constante

L’utilisation combinée du mot clé `static` et du mot clé `final` sur une variable de classe permet la définition d’une constante.

Le modificateur final indique que la valeur de la variable ne peut pas changer.

Les constantes définies de cette manière ne peuvent être ré-assignées, cela déclenche une erreur de compilation.

Par convention le nom des constantes est constitué uniquement de caractères majuscules.
Ex. `static final double PI = 3.141592653589793;`

La définition d’une variable comme constante dans votre code permet aux compilateur Java d’effectuer des optimisations lors de sa traduction en “byte-code” (le code intermédiaire multi-plateforme du runtime Java).

Programmation Objet avec Java

Interface

Dans le premier chapitre de ce cours nous avons rapidement abordé la notion d'interface comme concept de base de la programmation orientée objets :

- Il existe effectivement un nombre importants de cas où il est important de s'accorder sur un contrat définissant rigoureusement les interactions entre plusieurs modules de codes séparés.
- Les développeurs de chaque modules ne devraient pas avoir à connaître les détails des autres modules, uniquement leurs offres contractuelles de services.

Les interfaces Java sont une représentation informatique de cette contractualisation, mais sans le processus de négociation propre aux contrats de la vie de tous les jours.

Une interface Java est assez similaire aux classes que nous avons rencontrées jusqu'à présent mais **sans le corps des méthodes**.

Programmation Objet avec Java

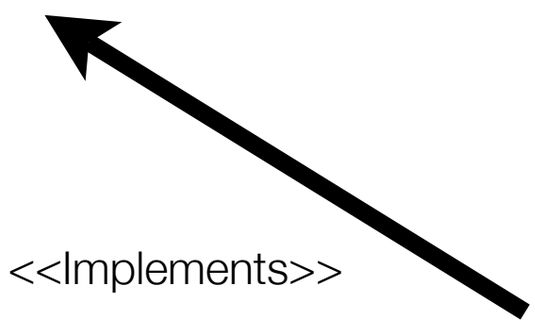
Interface

Une interface Java **ne peut être instanciée**, contrairement à une classe.

Une interface peut uniquement être **implémentée** par une classe, ou **étendue** par une autre interface :

- **Une même classe peut implémenter plusieurs interfaces**, alors qu'elle ne peut étendre qu'une seule classe (pas d'héritage multiple en Java)

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures
   int turn(Direction direction,   // An enum with values RIGHT, LEFT
            double radius, double startSpeed, double endSpeed);
   int changeLanes(Direction direction, double startSpeed, double endSpeed);
   int signalTurn(Direction direction, boolean signalOn);
   int getRadarFront(double distanceToCar, double speedOfCar);
   int getRadarRear(double distanceToCar, double speedOfCar);
         ......
   // more method signatures
}
```




```
public class OperateBMW760i implements OperateCar {

   // the OperateCar method signatures, with implementation --
   // for example:
   int signalTurn(Direction direction, boolean signalOn) {
      //code to turn BMW's LEFT turn indicator lights on
      //code to turn BMW's LEFT turn indicator lights off
      //code to turn BMW's RIGHT turn indicator lights on
      //code to turn BMW's RIGHT turn indicator lights off
   }

   // other members, as needed -- for example, helper classes
   // not visible to clients of the interface
}
```

Programmation Objet avec Java

Interface : définition

Une définition d'interface est constituée :

- De modificateurs :
  - `public` → l'interface est visible à toute classe dans tout package.
  - `private` → l'interface est visible uniquement aux classes du même package.
- D'une liste d'interfaces parentes (si elles existes) précédée du mot clé `extends`.
- Du corps de l'interface, où toute déclaration de méthode est implicitement publique.

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

   // constant declarations
   double E = 2.718282;  // base of natural logarithms

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);

}
```

Programmation Objet avec Java

Héritage

La notion d'héritage est simple mais puissante : lorsque vous voulez écrire une nouvelle classe, et qu'il existe déjà une autre classe qui intègre une partie des fonctionnalités que vous voulez implémenter, alors vous pouvez **dériver** cette nouvelle classe de la classe pré-existante.

- Une classe dérivée est **une sous-classe** d'une autre classe (parente, ou super-classe).

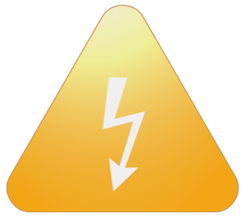
- **En dérivant, on hérite de tous les champs et méthodes 'public' ou 'protected' de la classe parente → réutilisation, capitalisation du code.**

- Les constructeurs ne sont pas hérités, mais il est possible dans une sous-classe d'invoquer les constructeurs de la classe parente.

Programmation Objet avec Java

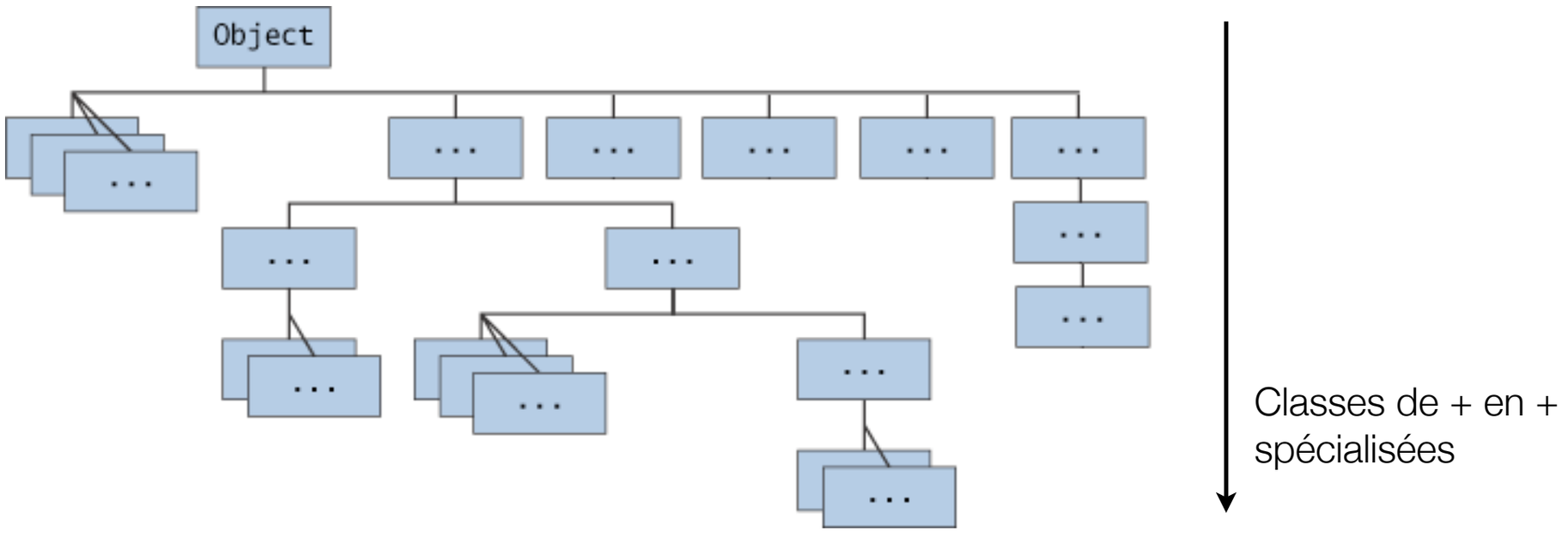
Héritage : classe ‘Object’

Toute classe est **implicitement** une sous-classe de la classe Java `java.lang.Object`.



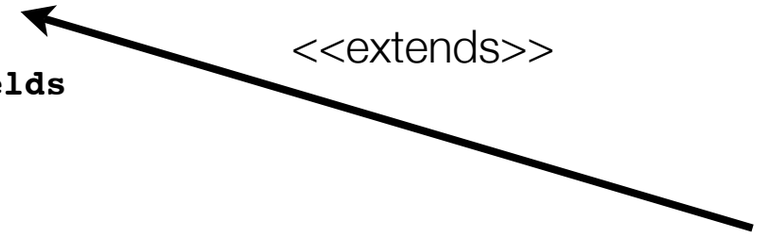
A l’exception de `Object` qui n’a pas de classe parente, **toute classe à une (et une seule !) super-classe directe** (héritage simple).

La classe `Object`, implémente des comportements communs à toutes les classes Java, dont celles que vous écrirez. De ce fait, elle est au sommet de la hiérarchie de classes Java.



Programmation Objet avec Java

Héritage : exemple




```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence,
          int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
     int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

Programmation Objet avec Java

Héritage : possibilités

Dans une sous classe, les actions suivantes sont possibles :

- Les champs et méthodes héritées de la super-classe peuvent être utilisés directement.
- Déclarer de nouveaux champs ou méthodes qui ne sont pas dans la super-classe.
- Déclarer un champs dans la sous-classe qui porte le même nom qu'un champs dans la super-classe : il **masque** alors ce dernier.
- Définir une méthode d'instance ayant la même signature qu'une méthode d'instance de la super-classe : elle **redéfinie** cette dernière.
- Définir une méthode de classe (statique) ayant la même signature qu'une méthode de classe de la super-classe : elle **masque** alors cette dernière.
- Ecrire un nouveau constructeur pour la sous-classe faisant appel à un constructeur de la super-classe, soit implicitement soit par l'utilisation du mot clé `super`.

Programmation Objet avec Java

Héritage : transtypage (“cast”)

Le type de données d’un objet est la classe à partir de laquelle il a été instancié.
Ex. `public MountainBike myBike = new MountainBike();`
→ `myBike` est de type `MountainBike`.

`MountainBike` est un descendant de `Bicycle` et `Object`. Ainsi, une instance de `MountainBike` est aussi de type `Bicycle` ou `Object`
→ elle peut être utilisée partout où une instance de `Bicycle` ou `Object` est attendue.

L’inverse n’est pas forcement vrai : un vélo (`Bicycle`) est peut être un vélo de montagne (`MountainBike`), mais pas forcément !

Le transtypage (ou “cast”) permet l’usage d’un objet d’un type donné, **à la place** d’un objet d’un autre type, en forçant la déclaration de type.

Programmation Objet avec Java

Héritage : transtypage (“cast”)

Par exemple, si on écrit :

```
Object obj = new MountainBike();
```

- Alors `obj` est à la fois un `Object` et un `MountainBike`. **Il s’agit d’un transtypage (“upcast”) implicite.**

A l’inverse, si on écrit maintenant :

```
MountainBike myBike = obj; ✖
```

- Cela va créer une erreur de compilation parceque `obj` n’est pas connu du compilateur comme étant un `MountainBike`.

- Mais il est possible de promettre au compilateur, à l'exécution, un objet de type `MountainBike` dans la variable `obj`.

- **On utilise une opération de transtypage (“downcast”) explicite :**
  `MountainBike myBike = (MountainBike) obj; ✔`

Programmation Objet avec Java

Héritage : redéfinition de méthodes

Si il existe une méthode **d'instance** dans une classe avec *la même signature (nom, type et nombre d'arguments) et type de retour* qu'une méthode d'instance dans une des super-classes de sa hiérarchie d'héritage, alors on dit que cette méthode est **redéfinie**.

Cette fonctionnalité de Java permet d'hériter d'une classe dont le comportement est suffisamment proche de son besoin puis de redéfinir ses méthodes au besoin.

Une méthode redéfinie peut aussi retourner un sous-type du type de retour de la méthode originelle (règle de "covariance").

Si l'on dispose d'une instance de sous-classe qui redéfinie des méthodes d'instances, alors **il n'est pas possible d'accéder aux méthodes originelles**, uniquement aux méthodes redéfinies.

Programmation Objet avec Java

Héritage : masquage de méthodes

Si il existe une méthode **de classe** dans une classe avec *la même signature* qu'une méthode de classe dans une des super-classes de sa hiérarchie d'héritage, alors on dit que cette méthode est **masquée**.

La distinction entre masquage et redéfinition a son importance, car contrairement à la redéfinition, **il est possible de retrouver la méthode originelle par un upcast.**

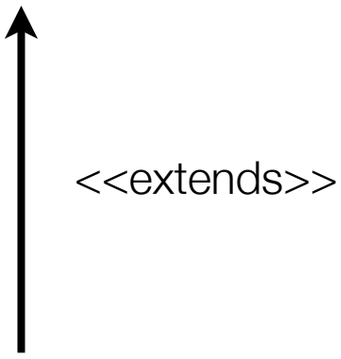
- *Voir exemple au transparent suivant.*

Nb : Dans une sous-classe, vous pouvez **surcharger** des méthodes héritées de classes parentes. Ces méthodes surchargées ne masquent ni ne redéfinissent les méthodes héritées, ce sont de nouvelles méthodes à part entière, propres à la sous-classe.

Programmation Objet avec Java

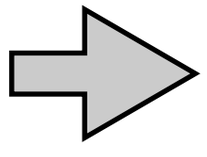
Héritage : masquage de méthodes

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```




```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Affiche



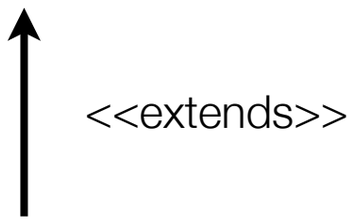
```
The class method in Animal.
The instance method in Cat.
```

Programmation Objet avec Java

Héritage : utilisation de ‘super’

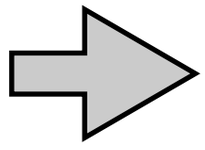
Si une de vos méthodes d’instance effectue la redéfinition d’une méthode de super-classe, alors il est possible d’invoquer la méthode originelle grâce au mot clé `super`.

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```




```
public class Subclass extends Superclass {

    public void printMethod() { //overrides printMethod in Superclass
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {

    Subclass s = new Subclass();
    s.printMethod();
    }

}
```

Affiche



```
Printed in Superclass.
Printed in Subclass
```

Programmation Objet avec Java

Héritage : utilisation de ‘super’

Le mot clé `super` peut aussi être utilisé dans les constructeurs, pour accéder aux constructeurs d’une classe parente.

```
//Rappel : MountainBike est une sous classe de Bike

public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
}
```

L’invocation d’un constructeur parent **doit être la première ligne** du constructeur.

Avec `super()`, le constructeur sans argument de la classe parente est appelé.

Attention : Si votre constructeur n'appelle pas explicitement un constructeur de la super-classe alors un appel automatique au constructeur sans argument de la super-classe est ajouté.
→ Erreur de compilation si il n’existe pas.
→ Si `Object` est la seule super-classe, pas de problème car elle possède un constructeur sans arguments.

Programmation Objet avec Java

Héritage : utilisation de ‘final’

Le mot clé `final` peut être utilisé sur toute déclaration de méthode dans vos classes.

Une méthode déclarée final ne peut pas être redéfinie par d’éventuelles sous-classes.
Nb : la classe `Object` de Java le fait pour plusieurs de ses méthodes.

Une raison possible pour procéder de la sorte pourrait être de s’assurer qu’une méthode vitale pour la consistance de l’état de l’objet ne puisse être altérée.

Ex :

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Nb : Il est aussi possible de définir une classe entière comme étant `final`. Ce qui empêche son sous-classement. C’est par exemple le cas de la classe `String`.

Programmation Objet avec Java

Héritage : utilisation de ‘abstract’

Une classe abstraite est une classe ayant été déclarée avec le mot clé `abstract`. Elle peut inclure des méthodes abstraites, ne peut être instanciée mais peut être sous-classée.

Une méthode abstraite est déclarée sans implémentation :

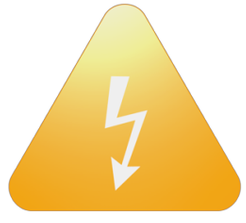
```
abstract void moveTo(double deltaX, double deltaY);
```

Si une classe contiens des méthodes abstraites alors la classe elle-même doit être déclarée abstraite :

```
public abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw();
}
```

Une sous classe d’une classe abstraite doit être elle aussi déclarée abstraite si elle ne fournis pas d’implémentation pour toutes les méthodes abstraites de sa classe parente.

Nb : toutes les méthodes d’une interface sont implicitement abstraites !

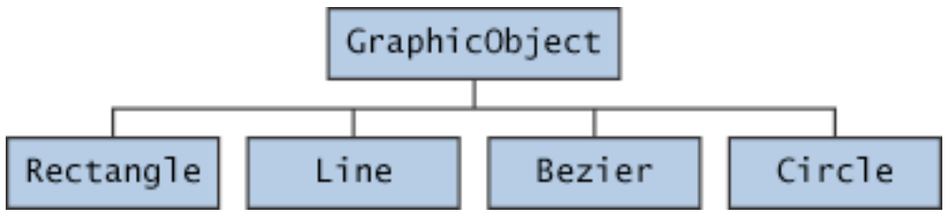


Programmation Objet avec Java

Héritage : utilisation de ‘abstract’

Intérêt par rapport aux interfaces : une classe abstraite peut fournir une implémentation partielle de ses fonctionnalités (des méthodes non-abstraites).

Exemple d’usage : modélisation d’objets graphiques.




```
abstract class GraphicObject {
    //Implémentation partielle
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }

    //Implémentation spécifique dans
    //chaque type d’objet graphique
    abstract void draw();
    abstract void resize();
}
```

<<extends>>

<<extends>>

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Concepts Avancés

Comme son nom l'indique, ce chapitre aborde des notions avancées de la programmation Java :

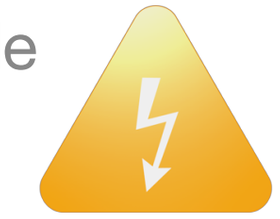
- Les Génériques.
- Les API de Collections (structures de données).

Concepts Avancés

Génériques : introduction

Les génériques sont une fonctionnalité relativement récente du langage Java (depuis Java 5) qui permettent d'**améliorer la stabilité (donc la fiabilité) de votre code**.

Ils permettent de détecter, **au moment de la compilation**, des erreurs de programmation qui n'auraient précédemment été détectés que lors de l'exécution.



Ils sont notamment utilisés de manière extensive par les Collections (chapitre suivant).

Pour simplifier les choses dans ce chapitre et éviter les dépendances circulaires dans le cours, nous allons travailler sur des exemples proches des Collections, sans pour autant les mettre en oeuvre.

Concepts Avancés

Génériques : introduction

On commence par définir une classe non-générique Box (boite) qui travaille sur n'importe quel type d'objet (`Object`). On fournis 2 méthodes :

- `add` : pour ajouter un objet dans la boite.
- `get` : pour obtenir l'objet stocké dans la boite.

```
public class Box {

    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

Puisque les méthodes de `Box` acceptent et retournent des `Objects`, vous pouvez y stocker ce que vous voulez.

A contrario, il est impossible de restreindre une boite, au moment de son instanciation, à un type d'objet particulier (comme à `Integer` par ex.).

➡ On ne peut que mettre une indication en commentaire...

```
public class BoxDemo1 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        integerBox.add(new Integer(10));
        Integer someInteger = (Integer)integerBox.get();
        System.out.println(someInteger);
    }
}
```

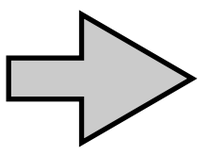
Mais rien n'empêche, dans les faits, un programmeur de passer un type non autorisé en argument.

Dans l'exemple suivant, l'erreur n'est pas détecté à la compilation, uniquement lors de l'exécution !

➡ Bien sur, il n'est pas possible de transtyper une `String` en `Integer`.

```
public class BoxDemo2 {

    public static void main(String[] args) {

        // ONLY place Integer objects into this box!
        Box integerBox = new Box();

        // Imagine this is one part of a large application
        // modified by one programmer.
        integerBox.add("10"); // note how the type is now String

        // ... and this is another, perhaps written
        // by a different programmer
        Integer someInteger = (Integer)integerBox.get(); //BUG !
        System.out.println(someInteger);
    }
}
```

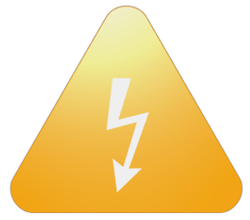
Affiche



```
Exception in thread "main"
    java.lang.ClassCastException:
       java.lang.String cannot be cast
       to java.lang.Integer
       at BoxDemo2.main(BoxDemo2.java:6)
```

Grâce aux génériques, l'erreur de transtypage précédente aurait pu être **détectée par le compilateur Java, avant l'exécution.**

On introduit ci-dessous **une version générique** de la classe `Box`. Cette dernière est maintenant paramétrée par un type (non-primitif) grâce à la **variable de type** `T`.



➡ Au moment de la déclaration et instanciation d'une `Box`, il faudra indiquer la valeur de `T`, c'est à dire fournir un type concret qui sera utilisé pour cette instance.

```
public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

```
//Exemple d'instanciation

Box<Integer> integerBox =
new Box<Integer>();
```

Une fois la variable `integerBox` initialisée, il est possible d'invoquer sa méthode `get` **sans utiliser de transtypage**.

```
public class BoxDemo3 {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
        System.out.println(someInteger);
    }
}
```

De plus, si vous essayer d'ajouter dans la `integerBox` un objet de type incompatible avec `Integer`, comme une `String`, alors **l'erreur est détectée à la compilation !**

```
BoxDemo3.java:5: add(java.lang.Integer) in Box<java.lang.Integer>
    cannot be applied to (java.lang.String)
       integerBox.add("10");
                      ^
    1 error
```

Concepts Avancés

Génériques : méthodes & constructeurs

Des paramètres de types peuvent aussi être déclarés au sein des signatures de méthodes ou constructeurs → méthodes génériques, constructeurs génériques.

Dans ce cas, **la visibilité du paramètre de type est limitée à la méthode ou au constructeur**, et non à la classe.

```
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

Affiche

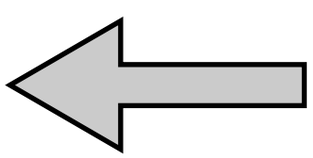
```
T: java.lang.Integer
U: java.lang.String
```

Une utilisation plus réaliste d'une méthode générique serait l'exemple suivant qui défini une méthode statique qui rempli plusieurs boites avec le même objet :

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    }
}
```

Le code suivant utilise cette méthode pour remplir des boites avec des crayons. On s'aperçoit qu'**il n'est pas toujours nécessaire de fournir explicitement le paramètre de type à la méthode**:

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;

Box.<Crayon>fillBoxes(red, crayonBoxes);
//ou :
Box.fillBoxes(red, crayonBoxes);
```



Le compilateur infère le type `Crayon`

Concepts Avancés

Génériques : paramètres de type liés

Grâce aux paramètres de type liés, vous pouvez restreindre le paramètre de type d'une classe, méthode ou constructeur **à une classe donnée, ou une de ses sous-classes**.

Pour se faire on utilise le mot clé `extends` dans la déclaration de paramètre de type :

```
public class Box<T> {
    (...)

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    (...)

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // compile-time error: this is still String!
    }
}
```

Uniquement **`Number`** ou une de ses sous-classes

Concepts Avancés

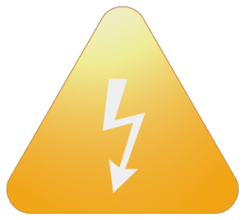
Génériques : sous-typage

Il est possible d'instancier une boite générique avec le paramètre de type `Number`, puis ensuite d'utiliser **un sous-type** de `Number` comme argument de `add()` :

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10)); // OK
box.add(new Double(10.1)); // OK
```

Maintenant, si l'on considère la méthode suivante, est-il possible de lui passer une `Box<Integer>` ou `Box<Double>` en argument ?

→ **Non car ce ne sont pas des sous-types de `Box<Number>` !**



```
public static void boxTest(Box<Number> b){
   // method body omitted
}
```

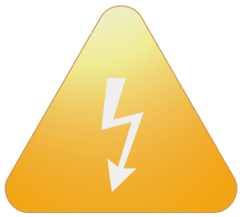
Pour spécifier une méthode qui accepte **un type de boite encore inconnu** mais qui sera un sous-type de `Number` il faut utiliser la syntaxe suivante (marche aussi avec le mot-clé `super` pour accepter des super-types) :

```
Box<? extends Number> n
```

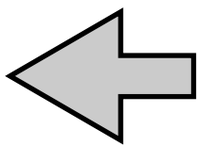
Concepts Avancés

Génériques : sous-typage

En effet, bien que `Box<Integer>` et `Box<Double>` ne soient pas des sous-types de `Box<Number>`, **ils sont des sous types de `Box<? extends Number>`.**



```
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void boxTest(Box<? extends Number> b){
       Number number = b.get(); // OK
       System.out.println(number);

       b.add(new Integer(20)); //ERREUR !
    }

    public static void main(String[] args) {
        Box<Integer> aBox = new Box<Integer>();
        aBox.add(new Integer(10));

        Box.boxTest(aBox);  //OK
    }
}
```



Attention : avec l'utilisation de '?' **la modification** de la boite n'est plus permise par le compilateur Java !

La lecture (get) est toujours possible par contre.

Concepts Avancés
Collections

Une collection est un objet capable de regrouper plusieurs éléments. Elle est utilisée pour stocker, accéder, manipuler et agréger des données.

Le framework de Collections offert par Java est une architecture unifiée pour représenter et manipuler des collections. Il offre :

- **Des interfaces** : types de données abstraits qui représentent les collections. Permet de les manipuler sans connaître les détails d'implémentation.

- **Des implémentations** : il s'agit des collections concrètes, implémentants les interfaces ci-dessus.

- **Des algorithmes** : des méthodes qui effectuent des recherches, de tri ou calculs divers sur les collections.

Concepts Avancés
Collections : bénéfices

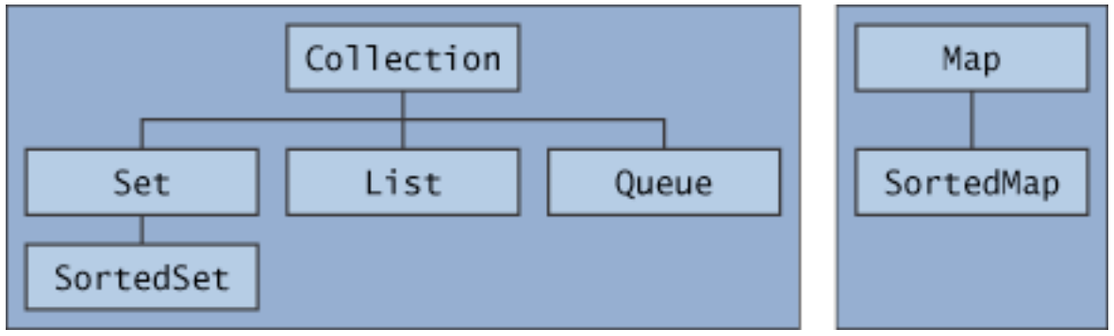
L'utilisation généralisée du framework de collections Java permet :

- **De réduire vos efforts** de programmation en vous concentrant sur le code métier de votre application, pas de sa plomberie.
- **D'améliorer les performances** de vos applications.
- **D'améliorer l'intéropérabilité et la réutilisabilité** de différents modules de code.
- **D'éviter d'avoir à apprendre différentes API** de manipulation de collections.
- **D'éviter d'avoir à ré-inventer la roue** dans chacune de vos applications.

Concepts Avancés

Collections : interfaces

Les interfaces principales du framework représentent différents types de collections. Elles forment une hiérarchie :



Un `Set` (ensemble) est un type particulier de Collection, un `SortedSet` (ensemble trié) est un type particulier de `Set`, etc...

On note que la hiérarchie est constituée de 2 arbres distincts : en fait, un `Map` n'est pas une véritable `Collection`.

Toutes ces interfaces sont génériques. La déclaration de l'interface Collection est la suivante : `public interface Collection<E>...`

Lorsque vous déclarez une instance de `Collection` ou de ses sous-classes dans votre code, **vous devez spécifier le type des objets qu'elle contiendra** (via la variable de type).

Pour maintenir un nombre d'interfaces réduit dans le framework, on ne dispose pas d'une interface distincte pour chaque variante de chaque type de collection :

- En fait, chaque interface regroupe plusieurs implémentations aux propriétés distinctes.
- Ces propriétés peuvent être par exemple la non-modification de la collection ("immutable"), le verrouillage de sa taille, le fait de ne pouvoir supprimer aucun élément...

Concepts Avancés

Collections : interface ‘Collection’

Une `Collection` représente un groupe d’objets appelés **éléments**.

C’est le plus petit dénominateur commun de toutes les catégories et variantes du framework de collections.

On ne dispose pas d’implémentations directes de cette interface, uniquement de sous-interfaces telles que `Set` et `List`.

De plus, la plupart des classes du framework offre un constructeur dit “de conversion”, prenant en argument une Collection et créant une nouvelle instance à partir de cette collection.

- Par exemple, pour créer une liste à partir d’une collection :

```
Collection<String> c = ...;
List<String> list = new ArrayList<String>(c);
```

Concepts Avancés

Collections : interface ‘Collection’


```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Nb : Les méthodes optionnelles de l’interface ne sont pas forcément implémentées par les classes implémentants Collection<E>.

Concepts Avancés

Collections : interface ‘Collection’

2 choix s’offres à vous pour parcourir une collection :

- **for-each :**

```
for (Object o : collection)
    System.out.println(o);
```

- **Iterateur :** récupéré par le biais de la méthode `iterator()` de la Collection :

```
//Interface des itérateurs
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}


//Exemple d’utilisation d’un itérateur sur une Collection<Integer>
for (Iterator<Integer> it = c.iterator(); it.hasNext();) {
   Integer element = it.next();
   (...)
}
```

Les opérations suivantes sont appliquées à l’ensemble de la collection :

- `containsAll` : retourne `true` si la collection contient tous les éléments d’une collection passée en argument

- `addAll` : ajoute tous les éléments de la collection passée en argument à la collection cible.

- `removeAll` : enlève de la collection cible tous les éléments de la collection passée en argument.

- `retainAll` : enlève de la collection cible tous les éléments qui ne sont pas dans la collection passée en argument.

- `clear` : enlève tous les éléments contenus dans la collection.

Concepts Avancés

Collections : interface ‘Set’

Un set (ensemble) est une collection qui **ne peut contenir de doublons**.

- Il est basé sur la notion algébrique d’*ensemble* et peut être utilisé pour représenter par exemple une main de poker, les cours d’un planning d’étudiant ou les processus en cours d’execution sur une machine.

- Il offre les mêmes méthodes que celles de l’interface Collection en leur ajoutant la restriction sur les doublons.

2 instances de Set sont considérées équivalentes (méthode `equals`) si elles contiennent les mêmes éléments.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Concepts Avancés

Collections : interface ‘List’

Une liste (ou séquence) est **une collection ordonnée**, qui peut contenir des doublons.

Son utilisateur a habituellement un contrôle précis sur la position d’insertion des éléments dans la liste et peut donc y accéder par un index (un entier).

En plus des opérations de Collection, elle offre des méthodes pour :

- **L’accès positionnel** aux éléments, par leur index.
- **La recherche** d’éléments dans la liste, qui retourne un index.
- **L’itération**, pour tirer parti de la nature séquentielle de la liste.
- **L’accès par portions** à la liste.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Concepts Avancés

Collections : interface ‘Queue’

Une `Queue` (file) est une collection utilisée pour regrouper plusieurs éléments avant leur traitement.

Mis à part les opérations basiques de l’interface `Collection`, une file fournis des opérations supplémentaires d’insertion, extraction et inspection :

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Les éléments d’une file sont habituellement ordonnés en FIFO (*first-in, first-out*), mais pas nécessairement. En fait, chaque file doit spécifier ses propriétés d'ordonnancement.

Chaque méthode de la file existe sous 2 formes : une qui renvoie une exception si l’opération échoue, la seconde qui retourne une valeur spéciale (`null` ou `false`) :

- Les méthodes d’insertion s’appliquent **à la queue** de la file.
- Les méthodes de suppression et d’inspection s’appliquent **à la tête** de la file.

	Exception	**Valeur spéciale**
Insertion	add(e)	offer(e)
Suppression	remove()	poll()
Inspection	element()	peek()

Nb : Il est possible de créer des files limitées en taille (“Bounded Queues”).

Concepts Avancés

Collections : interface ‘Map’

Une Map (association) va **lier des clés à des valeurs**.

- Elle ne peut contenir des clés en double et chaque clé est associée au plus à une valeur.

- Il s’agit en fait de l’abstraction de la notion mathématique de *fonction*.

Si vous avez déjà utilisé les HashTables (tables de hachage) Java plus anciennes, alors vous êtes familiers avec les concepts basiques d’une Map.

Les principales opérations d’une Map permettent notamment de créer et d’accéder à ces associations clé/valeur :

- `put`, `get`, `containsKey`, `containsValue`

- `size`, `isEmpty`

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Concepts Avancés
Collections : tri

Une liste peut être triée par la méthode statique `sort` de la classe `Collections` : **`Collections.sort(l);`**

- Si la liste contient des `String` elle sera triée par ordre alphabétique, des dates, par ordre chronologique... comment est-ce possible ?

- En fait, la méthode `sort` se base sur la méthode de comparaison 2-à-2 offerte par l'interface `Comparable`.

- Or, `String` et `Date` implémentent l'interface `Comparable`.

→ Implémenter l'interface `Comparable` pour une classe donnée permet de définir un **ordre naturel** des instances de cette classe.

Les classes suivante de l'API Java implémentent `Comparable` :

Classe	Ordre naturel implémenté
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

Concepts Avancés

Collections : tri

Vous ne pouvez trier avec `sort` que des éléments `Comparable` de types compatibles → sinon, `ClassCastException` !

En fait, il existe deux possibilités pour trier des éléments :

- Soit les éléments disposent déjà d'un ordre naturel (ils implémentent `Comparable`).

- Soit il ne dispose pas d'un ordre naturel ou vous souhaitez les trier dans un autre ordre. Dans ce cas il faut créer un `Comparator` qui sera passé en argument de la méthode `Collections.sort(list, comparator)`.

Si vous souhaitez trier vos propres classes dans un ordre donné, les 2 cas de figure s'offrent à vous.

L'interface `Comparable` est donnée ci-dessous :

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- La méthode d'instance `compareTo` compare l'objet cible avec l'objet passé en argument et retourne :
  - **un entier négatif** : si l'objet cible est "inférieur" à l'objet reçu.
  - **0** : si l'objet cible est égal à l'objet reçu.
  - **un entier positif** : si l'objet cible est "supérieur" à l'objet reçu.
- Si l'objet passé en argument ne peut être comparé à l'objet cible → `ClassCastException`.

Concepts Avancés

Collections : tri : Comparator

Comme pour `Comparable`, l'interface `Comparator` est constituée d'une seule méthode :

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- La méthode `compareTo` compare son premier paramètre `o1` avec son second paramètre `o2` et retourne :
  - **un entier négatif** : si `o1` est "inférieur" à `o2`.
  - **0** : si `o1` est égal à `o2`.
  - **un entier positif** : si `o1` est "supérieur" à `o2`.
- Si les objets passés en argument ne peuvent être comparés → `ClassCastException`.

Concepts Avancés

Collections : tri : interface ‘Sorted Set’

Il s’agit d’une déclinaison ordonnée de l’interface `Set` vue précédemment.

Elle maintient ses éléments dans leur ordre naturel, ou en fonction d’un `Comparator` fourni lors de la création d’une instance de `SortedSet`.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Concepts Avancés

Collections : tri : interface ‘SortedMap’

Il s’agit d’une déclinaison ordonnée de l’interface `Map` vue précédemment.

Elle maintient ses éléments dans l’ordre naturel de leurs clés, ou en fonction d’un `Comparator` sur clé fourni lors de la création d’une instance de `SortedSet`.

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Concepts Avancés

Collections : implémentations

Après avoir vu les interfaces du framework `Collection`, il faut maintenant s'attarder sur **les implémentations standards de ces interfaces**.

Les implémentations fournies sont en fait les véritables objets utilisés pour stocker les données contenues dans les collections.

Elles sont catégories en fonction de leur utilisation, bien que dans le cadre de ce chapitre nos n'abordions que les implémentations d'usage courant :

Interfaces	**Implementations d'usage courant**				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
Set	***HashSet***		TreeSet *(implémente aussi SortedSet)*		LinkedHashSet
List		***ArrayList***		LinkedList	
Queue				LinkedList *(car comportement FIFO)*	
Map	***HashMap***		TreeMap *(implémente aussi SortedMap)*		LinkedHashMap

Implementation = implémentation de base à utiliser sauf si besoins particuliers.

Toutes ces implémentations d'usage général ont les caractéristiques suivantes :

- Elles implémentent toutes les méthodes (même optionnelles) de leurs interfaces.
- Elles autorisent des éléments, clés ou valeurs nulles.
- Aucune ne sont synchronisées (thread-safe).
- Elles disposent d'itérateurs capables de détecter des modifications concurrentes à l'exécution (cas d'erreur).
- Elles sont `Serializable` (on peut les stocker dans un fichier par ex.)
- Elles supportent la méthode `clone`.

Bibliographie

The Java Tutorials : http://java.sun.com/docs/books/tutorial/index.html

Le livre de Java premier langage (Eyrolles)

Programmer en Java (Eyrolles)

Introduction à Java (O'Reilly)

Java : Tête la première (O'Reilly)

Java In A NutShell (O'Reilly)

Java Cookbook (O'Reilly)

Learning Java (O'reilly)

Fin du cours #2