



Programmation Java

Partie 2 – Au cœur du Java

Plan du chapitre

1. La Programmation Orientée Objet
2. Les relations entre objets
3. Les Exceptions
4. Créer et organiser son application
5. Les ensembles
6. Aller plus loin en POO



Chapitre 1

La Programmation Orientée Objet

La programmation procédurale	Les méthodes
Principes de la POO	Le constructeur
Java et la POO	La surcharge
Un peu de vocabulaire	La visibilité
Créer une classe	L'encapsulation
Instanciation	Les statiques
Utilisation des attributs	Attributs constants

La programmation procédurale

Ce qu'on appelle programmation **procédurale** est le fait d'exécuter du code **selon une procédure bien définie**, instruction après instruction, **dans l'ordre**.

Il s'agit finalement de la méthode d'exécution de n'importe quel programme et de notre méthode d'écriture actuelle.

```
; Global declarations
STATUS equ 3      ; Status register is File 3
C       equ 0      ; Carry/Not Borrow flag is bit0
        cblock 20h
        NUM:2      ; Number: high byte, low byte
        endc

MAIN    goto  SQR_ROOT

; *****
; * FUNCTION: Calculates the square root of a 16-bit integer *
; * EXAMPLE : Number = FFFFh (65,535d), Root = FFh (255d) *
; * ENTRY   : Number in File NUM:NUM+1 *
; * EXIT    : Root in W. NUM:NUM+1; I:I+1 and COUNT altered *
; *****

; Local declarations
        cblock
        I:2, COUNT ; Magic number hi:lo byte & loop count
        endc

SQR_ROOT
        org 200h    ; Code to begin @ 200h in Program store
        clr  COUNT  ; Task 1: Zero loop count
        clr  I       ; Task 2: Set magic number I to one
        clr  I+1
        incf I+1,f

; Task 3: DO
SQR_LOOP movf I+1,w  ; Task 3(a): Number - I
        subwf NUM+1,f ; Subtract lo byte I from lo byte Num
        movf I,w     ; Get high byte magic number
        btfss STATUS,C ; Skip if No Borrow out
        addlw 1      ; Return borrow
        subwf NUM,f  ; Subtract high bytes

; Task 3(b): IF underflow THEN exit
        btfss STATUS,C ; IF No Borrow THEN continue
        goto SQR_END   ; ELSE the process is complete
        incf COUNT,f   ; Task 3(c): ELSE inc loop count
        movf I+1,w     ; Task 3(d): Add 2 to the magic number
        addlw 2
        btfsc STATUS,C ; IF no carry THEN done
        incf I,f       ; ELSE add carry to upper byte I
        movwf I+1
        goto SQR_LOOP

SQR_END movf COUNT,w  ; Task 4: Return loop count as the root
        return
        end
```

La programmation procédurale

Dans certains cas, l'ordre de nos instructions sera plus ou moins respecté, comme avec les conditions, les boucles ou encore les fonctions.

Cela reste de la programmation procédurale, mais intelligente. Le programme résonne et se déplace en fonction de ses instructions

Un exemple ci-contre d'exécution procédurale d'un programme pourtant lu dans un ordre différent.

```
public static int sum( int ... nbs ) {  
  2 int total = 0 ;  
  3 for( int nb : nbs ) {  
    4,5,6 total += nb ;  
  }  
  7 return total ;  
}  
  
public static void main( String[] args ) {  
  1 System.out.println( sum(10, 20, 30 ) );  
}
```

Principe de la POO

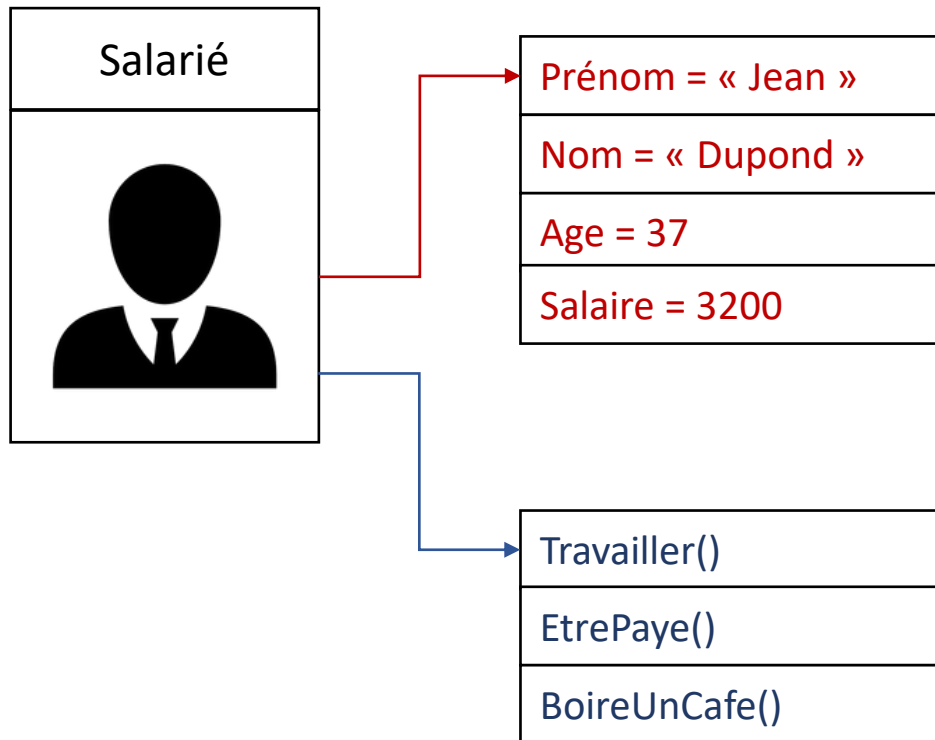
La Programmation Orientée Objet (POO) ou Object Oriented Programming (OOP) en anglais, est donc un style de programmation.

Mais plus qu'un style, il s'agit également d'éléments syntaxiques et de principes spécifiques. La POO entretient également un lien plus simple avec la réalité.

Enfin, en termes de développement, cette méthode **simplifie** grandement le code utilisé, son **sens** et son **organisation**.



Principe de la POO

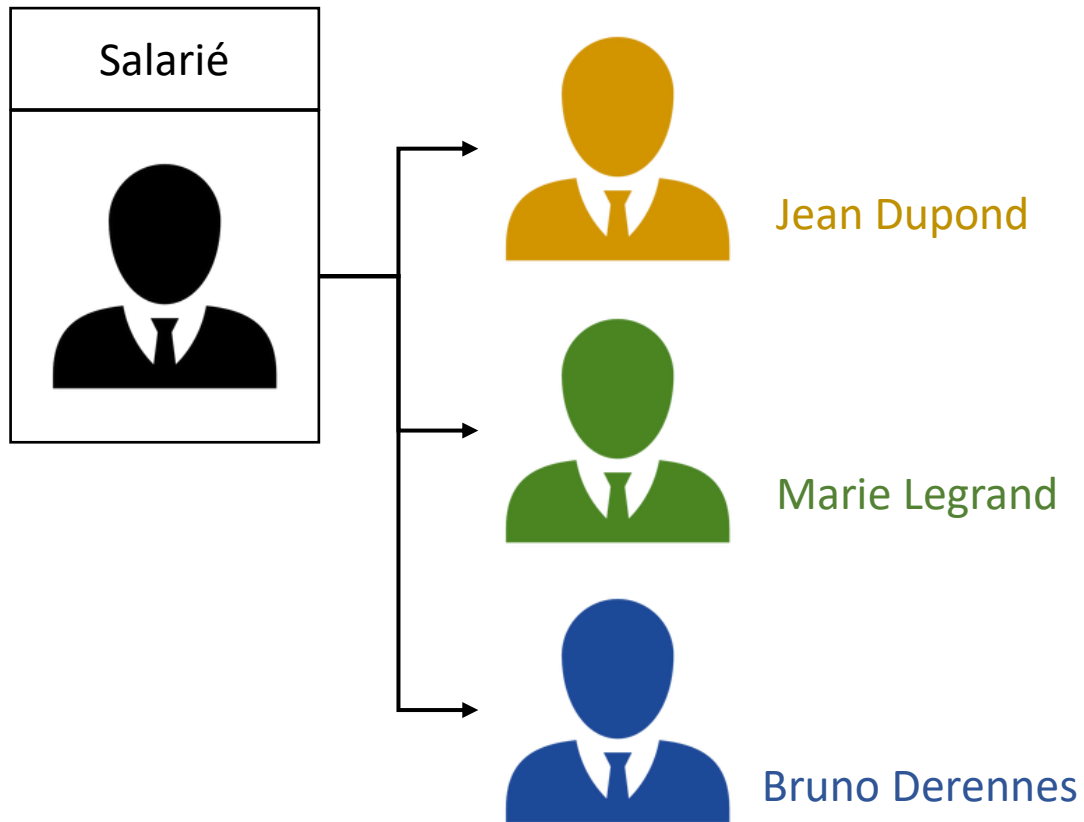


La POO tourne autour de la création d'objets représentant les éléments physiques ou théoriques avec lesquels notre programme interagira.

Par exemple, pour une application de gestion salariale, nous pourrions créer un objet salarié.

Cet objet salarié disposerait de différentes variables (nom, prénom, age, fonction, salaire, ...) et de fonctionnalités propres (travailler, être payé, boire un café, ...)

Principe de la POO

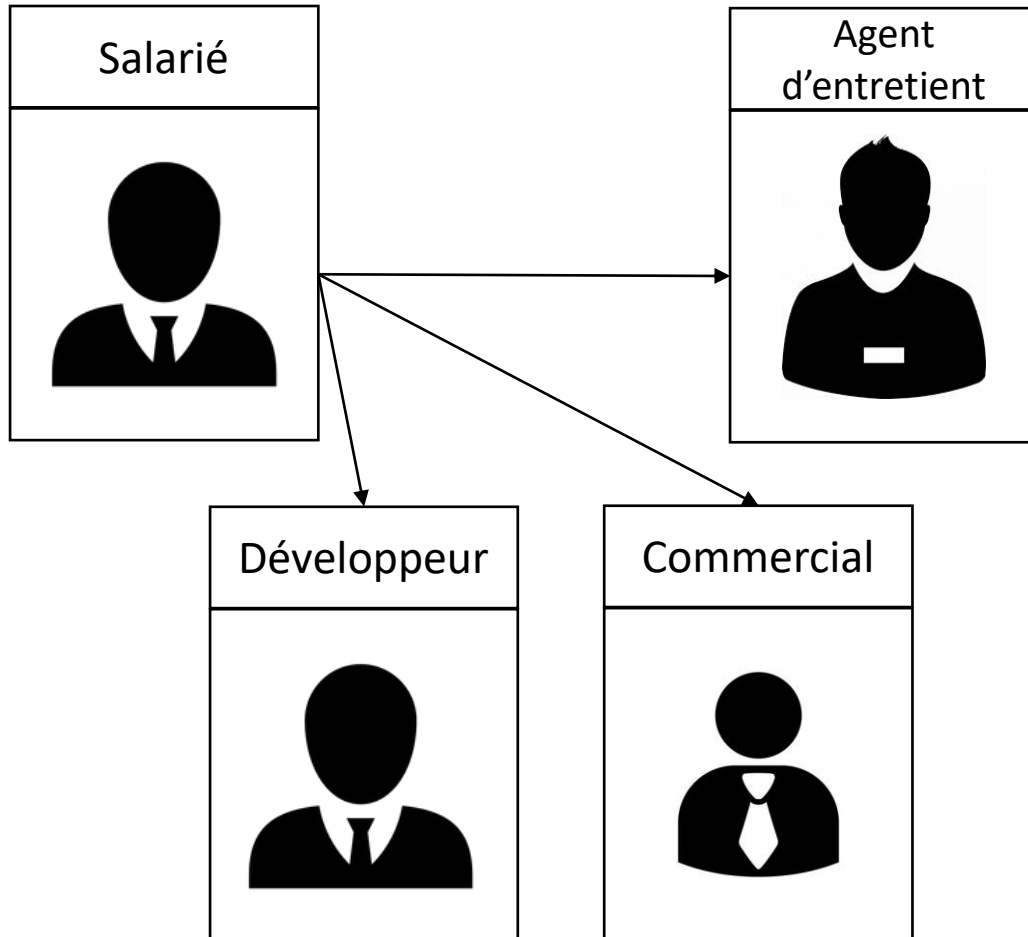


Le gros avantage ici, c'est qu'en ayant créé ce modèle de salarié, il devient très facile d'en créer une multitude sur le même principe.

Dans une écriture plus procédurale, il nous aurait fallu créer pour chaque salarié de nouvelles variables.

On distingue donc le **plan de l'objet**, ici **le salarié**, et ses **instances**, c'est-à-dire les différents objets créés à partir de ce plan, ici **Jean**, **Marie** et **Bruno**.

Principe de la POO

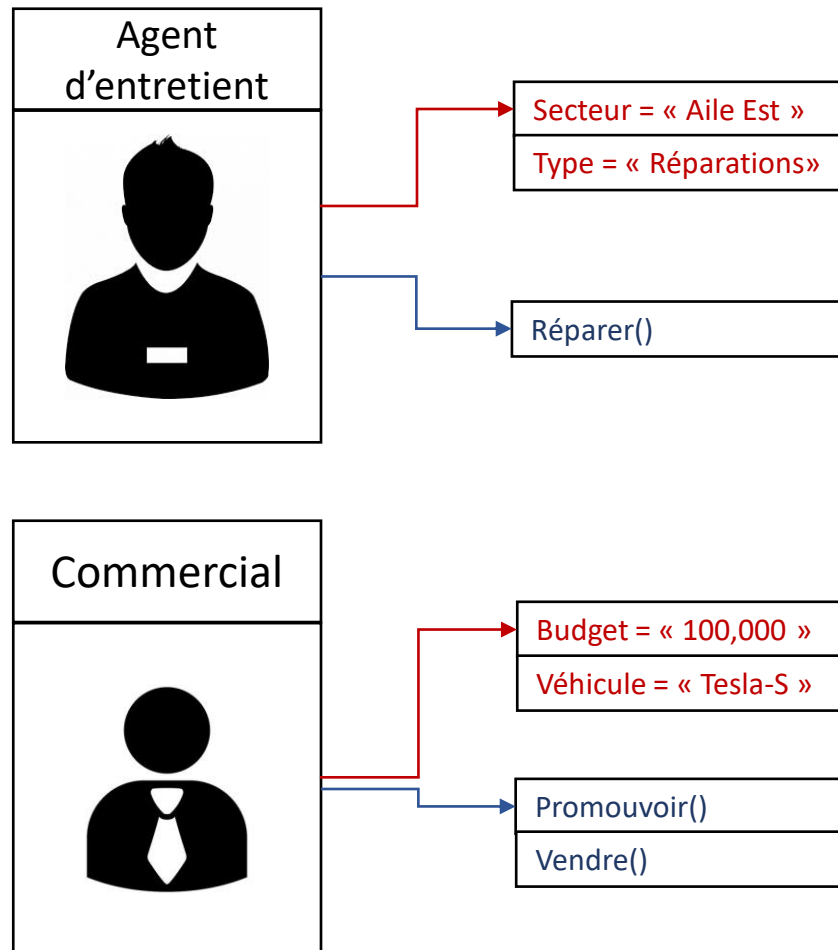


Autrement qu'une simple enveloppe contenant des variables et fonctions, notre objet est aussi une source de principes existants.

Ci-contre, on visualise trois types d'objets : les commerciaux, les développeurs et les agents d'entretien.

Chacun est un salarié, il dispose donc de toutes les variables et fonctions d'un salarié lambda. On peut donc dire que chacun est un salarié spécialisé.

Principe de la POO



Autrement qu'une simple enveloppe contenant des variables et fonctions, notre objet est aussi une source de principes existants.

Ci-contre, on visualise trois types d'objets : les commerciaux, les développeurs et les agents d'entretien.

Chacun est un salarié, il dispose donc de toutes les variables et méthodes d'un salarié lambda. On peut donc dire que chacun est un salarié spécialisé.

Java et la POO

La relation entre JAVA et la POO est très forte puisque ce langage est intégralement orienté objet.

Cela signifie que tout est considéré comme un objet ou en fait partie. Le programme est un objet, un entier est un objet, un module est un objet, bref... tout.

Souvenez vous du code ci-contre qui encadre notre programme, et notamment notre fonction principale main. Il s'agit de la définition de l'objet principal du programme.

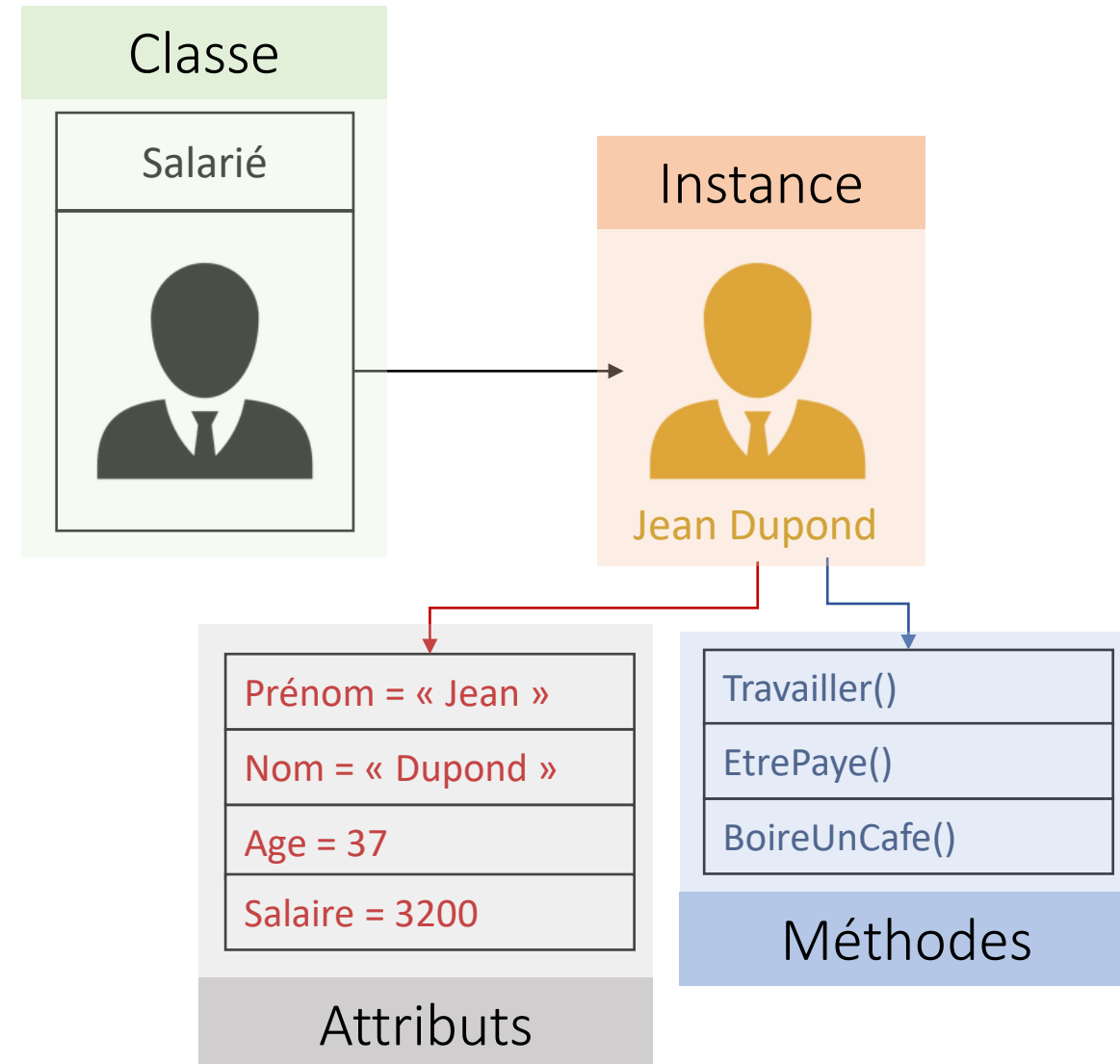
```
public class Main {  
  
}
```

Un peu de vocabulaire

Avant de commencer l'aspect pratique, voyons quelques mots et définitions concernant la POO.

Nous appellerons désormais classe un type d'objet. D'autre part, les objets créés à partir de ces classes sont nommés des instances.

Les variables que contiennent les classes sont des attributs, tandis que leurs fonctions sont des méthodes.



Créer une classe

```
class Main {  
  
}
```

Commençons la pratique ! En réalité, nous l'avons déjà fait. Chaque fichier, dont notre fichier principal Main.java, doit comporter une classe à son nom.

Repartons donc de 0 avec notre compréhension actuelle des classes. Créons un nouveau fichier Main.java et créons y une classe nommée Main.

Pour créer une classe, on en précédera le nom par **class**. Le contenu de celle-ci, ses attributs et ses méthodes, seront définies dans des « {} ».

Créer une classe – *Méthode main*

Ajoutons y la méthode main, qui pour rappel est la fonction principale qui se lancera lors de l'exécution.

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

Concernant les mots clefs **public** et **static**, ils ont à voir avec la POO mais nous en reparlerons plus tard.

Créer une classe – *Nouvelle classe*

La classe Main étant notre programme principal, nous allons créer une seconde classe pour la tester dans notre main.

Créons donc la classe Employee représentant le salarié vu plus tôt.

Pour le moment, nous la placerons après la classe Main, mais plus tard nous la déplacerons dans un fichier à part.

```
class Main {  
    public static void main( String[] args ) {  
    }  
}  
  
class Employee {  
  
}
```

Créer une classe – *Attributs*

Ajoutons à notre salarié ses attributs. Pour rappel il s'agit de variable le concernant.

Ces attributs seront les même pour toutes les instances de la classe, mais chaque instance pourra avoir des valeurs différentes et indépendantes.

Pour ajouter un attribut, il suffit de la déclarer entre les « {} » de la classe, sans valeurs.

```
class Employee {  
    String nom, prenom ;  
    int age ;  
    double salary;  
}
```


Instanciation

Pour créer une instance à partir d'une classe, nous devons le construire à partir du mot clef `new` et du nom de la classe.

Pour pouvoir nous servir correctement de cette instance, le mieux est de la stocker dans une variable typée avec la classe.

```
class Main {  
  
    public static void main( String[] args ) {  
        Employee e1 = new Employee();  
    }  
}  
  
class Employee {  
    String nom, prenom ;  
    int age ;  
    double salary;  
}
```

Instanciación

Il est possible de créer une multitude d'instance à partir de notre classe.

```
public static void main( String[] args ) {  
    Employee e1 = new Employee();  
    Employee e2 = new Employee();  
}
```

Ainsi, e1 et e2 sont tout deux des instances de la classe Employee.

Utilisation des attributs

Chaque instance dispose donc des attributs définis plus tôt. Pour y accéder, nous pouvons les appeler à partir de l'instance en utilisant le « . ».

Par défaut, la valeur de nos attributs sont **null**, mais il sera possible de changer ce fait plus tard.

Il est également possible de redéfinir leur valeur comme avec n'importe quelle variable.

```
public static void main( String[] args ) {  
    Employee e1 = new Employee();  
    System.out.println( e1.prenom );  
    e1.prenom = "Albert" ;  
    System.out.println( e1.prenom );  
}
```

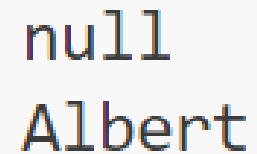


Diagram illustrating the state of the `prenom` attribute. It shows the initial value `null` and the updated value `Albert`.

Utilisation des attributs

Profitons en pour vérifier que chaque instance dispose de ses propres valeurs indépendantes pour chaque attributs.

```
Employee e1 = new Employee();  
Employee e2 = new Employee();  
  
e1.prenom = "Albert" ;  
e2.prenom = "Marie" ;  
  
System.out.println(  
    e1.prenom + " et " + e2.prenom  
    + " sont sur un bateau."  
);
```

```
Albert et Marie sont sur un bateau.
```

Les méthodes

Attelons nous à présent aux fonctions de nos objets : les méthodes. Celles-ci sont de simples fonctions appartenant à une classe spécifique.

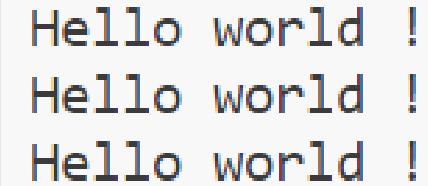
Nous les définirons comme habituellement. On y accède comme les attributs, mais n'oubliez pas les « () » pour exécuter votre fonction (avec leurs éventuels arguments).

```
class Employee {  
    String nom, prenom ;  
    int age ;  
    double salary;  
  
    void dire_bonjour() {  
        System.out.println("Hello world !") ;  
    }  
}
```

Les méthodes

Et bien évidemment, n'importe quelle instance pourra faire usage de ces méthodes.

```
public static void main( String[] args ) {  
    Employee  
        e1 = new Employee(),  
        e2 = new Employee(),  
        e3 = new Employee();  
  
    e1.dire_bonjour();  
    e2.dire_bonjour();  
    e3.dire_bonjour();  
}
```



Hello world !
Hello world !
Hello world !

Ci-dessus, nous faisons appel à 3 instances différentes pour exécuter notre méthode.

Les méthodes – *Usage des attributs*

Mais là où nos méthodes deviennent vraiment intéressantes, c'est lorsque l'on utilise nos attributs dans nos méthodes.

Pour ce faire, nous utiliserons le mot clef **this**. Celui-ci représente l'instance qui a lancé la méthode. Avec le « . », nous pouvons donc accéder à tous ses attributs et méthodes.

```
String full_name() {  
    return this.prenom + " " + this.nom ;  
}
```

Les méthodes – *Usage des attributs*

Dans l'exemple ci-dessous, nous récupérons le nom complet de notre employé pour le présenter dans la méthode **dire_bonjour**.

```
void dire_bonjour() {  
    System.out.println("Hello world !") ;  
    System.out.println("My name is " + this.full_name());  
}  
  
String full_name() {  
    return prenom + " " + this.nom ;  
}
```


Les méthodes – *Usage des attributs*

Bien entendu, pour en faire le meilleur usage, il nous faudra tout d'abord initialiser les attributs de nos différentes instances.

```
Employee e1 = new Employee();  
Employee e2 = new Employee();
```

```
e1.nom = "Dupond" ;  
e1.prenom = "Jean" ;
```

```
e2.nom = "Legrand" ;  
e2.prenom = "Marie" ;
```

```
e1.dire_bonjour();  
e2.dire_bonjour();
```

```
Hello world !  
My name is Jean Dupond  
Hello world !  
My name is Marie Legrand
```

Le constructeur

```
Employee(  
    String prenom, String nom,  
    int age, double salary  
) {  
    this.nom = nom ;  
    this.prenom = prenom ;  
    this.age = age ;  
    this.salary = salary ;  
}
```

Et des attributs, nous pourrions en avoir beaucoup plus à initialiser. Cela devient très vite lourd et source d'erreurs.

C'est pourquoi il existe une méthode exécutée à la création d'une nouvelle instance, qui a pour but d'initialiser ces valeurs, éventuellement à l'aide de paramètres.

Cette méthode s'appelle le constructeur. Elle se nomme par le nom de la classe. Dans notre cas, nous définirons des paramètres pour initialiser tout nos attributs

Le constructeur

Lors de l'instanciation, il faudra bien veiller à respecter le nombre, le type et l'ordre des paramètres à fournir entre les « () ».

Nous pouvons alors grandement raccourcir notre code précédent, pourtant pour le même rendu.

```
Employee e1 = new Employee( "Jean", "Dupond", 37, 2200 );  
Employee e2 = new Employee( "Marie", "Legrand", 41, 2600 );
```

```
e1.dire_bonjour();  
e2.dire_bonjour();
```

```
Hello world !  
My name is Jean Dupond  
Hello world !  
My name is Marie Legrand
```

La surcharge

Une méthode est définie par son prototype, c'est-à-dire sa définition. Elle contient, entre autres, le type de la fonction, son nom et le type de ses arguments.

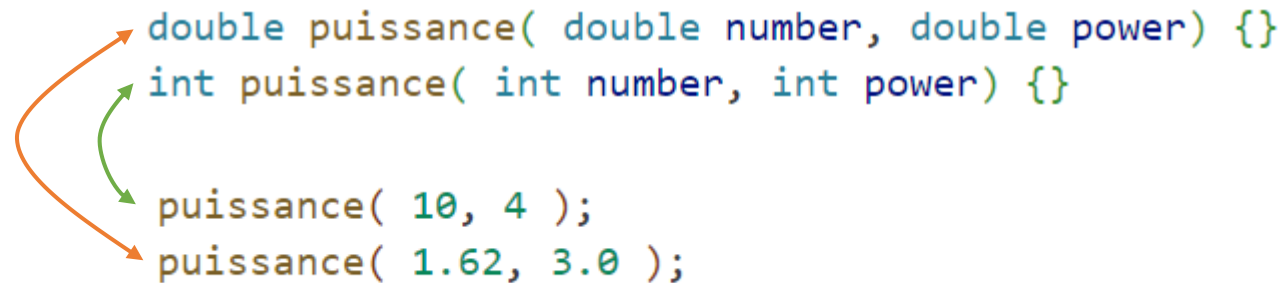
Deux prototypes avec l'un de ces éléments différent sont donc deux prototypes différents qui définissent potentiellement un ensemble d'instructions différent.

```
double puissance( double number, double power)
```

La surcharge

Lorsque deux prototypes ont le même nom de méthode, mais des arguments ou un type de retour différent, alors il s'agit d'une surcharge de la méthode initiale.

Cela signifie que l'on ajoute un cas d'utilisation de la fonction en quelques sorte. Elle s'adapte en fonction du type d'arguments choisi lors de l'utilisation.



```
double puissance( double number, double power) {}  
int puissance( int number, int power) {}  
  
puissance( 10, 4 );  
puissance( 1.62, 3.0 );
```

The diagram illustrates method overloading. It shows two method signatures: `double puissance(double number, double power) {}` and `int puissance(int number, int power) {}`. Below these are two calls: `puissance(10, 4);` and `puissance(1.62, 3.0);`. An orange arrow points from the first call to the first signature, and a green arrow points from the second call to the second signature, demonstrating how the compiler selects the correct method based on the argument types.

La surcharge

La surcharge est également possible avec les constructeurs. On ne redéfinira ici que les types de paramètres.

Cela permet par exemple à notre constructeur d'**Employee** de pouvoir ou non prendre en charge l'âge et le salaire.

```
Employee( String prenom, String nom ) {  
    this.nom = nom ;  
    this.prenom = prenom ;  
}
```

La surcharge – *Appel à un autre constructeur*


Vous remarquerez que l'initialisation du nom et du prénom sont les mêmes dans un constructeur ou dans l'autre.

Cette redondance induit un risque d'oubli en cas de mise à jour. Celle-ci ne serait pas répliquée systématiquement.

Il est donc plus efficace de faire directement appel au constructeur initialisant nos premiers paramètres dans notre constructeur plus complexe avec **this**.

```
Employee( String prenom, String nom ) {  
    this.nom = nom ;  
    this.prenom = prenom ;  
}
```

```
Employee(  
    String prenom, String nom,  
    int age, double salary  
) {  
    this( prenom, nom );  
    this.age = age ;  
    this.salary = salary ;  
}
```



La visibilité

En POO, les attributs et méthodes possèdent des visibilité. Il s'agit de droits d'accès à ces éléments depuis différents niveau de l'application.

Par défaut, tout est **public**, c'est-à-dire que tout est accessible, attribut ou méthode, depuis l'intérieur de la classe ou depuis l'extérieur.

En définissant l'un de ces élément en **private**, il devient alors inaccessible depuis l'extérieur de la classe.

```
class Employee {  
  
    String nom, prenom ;  
    int age ;  
    double salary;  
  
    Employee( String prenom, String nom ) {  
        this.nom = nom ;  
        this.prenom = prenom ;  
    }  
  
    public static void main( String[] args ) {  
  
        Employee e1 = new Employee( "Jean", "Dupond", 37, 2200 );  
        Employee e2 = new Employee( "Marie", "Legrand", 41, 2600 );  
  
        e1.dire_bonjour();  
        e2.dire_bonjour();  
    }  
}
```

Les attributs sont ici utilisés à l'intérieur de leur classe, dans un milieu **privé**.

La méthode `dire_bonjour` est utilisée dans un milieu **public**, à l'extérieur de la classe.

La visibilité

Pour définir la visibilité d'un attribut ou d'une méthode, on précédera sa définition d'un mot clef entre **public** et **private**.

Lors de la tentative d'utilisation d'un élément privé dans un milieu publique, une erreur surviendra pour nous indiquer un problème de visibilité.

```
class Employee {  
  
    private String nom, prenom ;  
  
    Employee( String prenom, String nom ) {  
        this.nom = nom ;  
        this.prenom = prenom ;  
    }  
}  
  
public static void main( String[] args ) {  
  
    Employee e1 = new Employee( "Jean", "Dupond" );  
    System.out.println( e1.nom );  
}
```

```
.\Main.java:11: error: nom has private access in Employee  
        System.out.println( e1.nom );  
                           ^
```

La visibilité

En revanche, il sera tout à fait possible d'utiliser nos attributs privés au sein de méthodes publiques par exemple.

Cela permet de contrôler leur utilisation, de régir la manipulation des données de vos objets depuis l'extérieur.

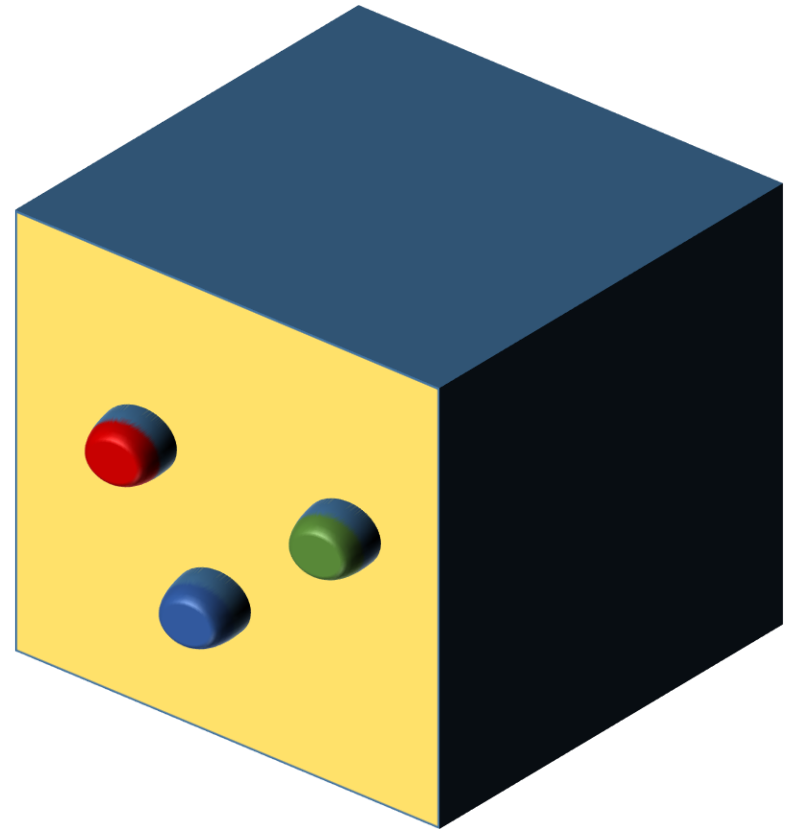
```
class Employee {  
  
    private String nom, prenom ;  
  
    Employee( String prenom, String nom ) {  
        this.nom = nom ;  
        this.prenom = prenom ;  
    }  
  
    public String full_name() {  
        return this.prenom + " " + this.nom ;  
    }  
  
}  
  
public static void main( String[] args ) {  
  
    Employee e1 = new Employee( "Jean", "Dupond" );  
    System.out.println( e1.full_name() );  
  
}
```

L'encapsulation

L'encapsulation est un principe d'organisation de nos classes permettant de mieux gérer et protéger nos attributs et certaines de nos méthodes.

Ce principe est de définir le plus souvent possible nos attributs de manière privée et nos méthodes publiques.

Voyez votre objet comme un mécanisme dont on ne voit pas les rouages intérieurs. On dispose tout de même d'outils extérieurs pour s'en servir correctement.



L'encapsulation - *Getters*

Comment accéder alors correctement à nos attributs ?

La méthode la plus adaptée est de créer un getter, c'est-à-dire une méthode retournant la valeur de votre attribut.

L'intérêt, même si votre attribut est retourné tel quel, est de pouvoir anticiper l'évolution de cet accès. Si un changement doit survenir, ce sera dans le getter et nulle part ailleurs.

```
public String get_prenom() {  
    return this.prenom ;  
}
```

L'encapsulation - *Setters*

Attention cependant, la valeur retournée par notre getter pourra être modifiée, mais sa nouvelle valeur ne sera pas enregistrée dans notre attribut.

Pour permettre la modification d'un attribut depuis l'extérieur, nous créerons un setter.

Là encore, il s'agit d'une méthode, publique, prenant en paramètre la nouvelle valeur et l'appliquant sur l'attribut concerné.

```
public void set_prenom( String nouveau_prenom ) {  
    this.prenom = nouveau_prenom ;  
}
```

Les statiques — *Attributs statiques*

```
class Employee {  
  
    private static int count = 0 ;  
    private String nom, prenom ;  
}
```

Il est possible de rendre un attribut ou une méthode statique. Cela signifie que plutôt que d'être indépendant pour chaque instance, la valeur sera partagée pour toute la classe.

Pour créer un attribut statique, on lui ajoute le mot clef **static** devant sa déclaration.

On pourra également lui définir une valeur par défaut, comme ci-contre avec un compteur d'employés instanciés.

Les statiques — *Attributs statiques*

```
1  class Main {  
2      public static void main( String[] args ) {  
3  
4          new Employee("Jean", "Dupond");  
5          new Employee("Marie", "Legrand");  
6          new Employee("James", "Durand");  
7  
8      }  
9  }  
10  
11 class Employee {  
12  
13     private static int count = 0 ;  
14     private String nom, prenom ;  
15  
16     Employee( String prenom, String nom ) {  
17         this.nom = nom ;  
18         this.prenom = prenom ;  
19         this.count += 1 ;  
20         System.out.println("Employé n°" + this.count + " créé !");  
21     }  
22  
23 }
```

Pour l'utiliser, il vous suffira de l'invoquer dans vos méthodes via **this**, ou depuis l'extérieur via le nom de la classe (à condition que sa visibilité soit publique).

Remarquez que malgré les 3 instances, la valeur est bien partagée.

```
Employé n°1 créé !  
Employé n°2 créé !  
Employé n°3 créé !
```

Les statiques – *Méthodes statiques*

Le même principe est appliqué aux méthodes pour créer des méthodes statiques : on y ajoute `statique` entre la visibilité et le type.

De cette manière, notre méthode sera accessible sans instance préalable.

Un exemple ci-contre avec une méthode affichant le nombre d'employés instanciés jusqu'alors.

```
public static void details() {  
    System.out.println(  
        "Nous avons instancié " + count + " employés"  
    );  
}
```

```
new Employee("Jean", "Dupond");  
new Employee("Marie", "Legrand");  
new Employee("James", "Durand");  
  
Employee.details();
```

```
Employé n°1 créé !  
Employé n°2 créé !  
Employé n°3 créé !  
Nous avons instancié 3 employés
```


Les attributs constants

Il est possible de définir un attribut comme étant constant. Nous utiliserons pour ce faire le même mot clef que pour une constante classique : **final**.

```
private final double salary = 2000 ;
```

L'affectation d'une valeur lors de la déclaration devient obligatoire et toute modification par la suite impossible.



Pause questions

Chapitre 2



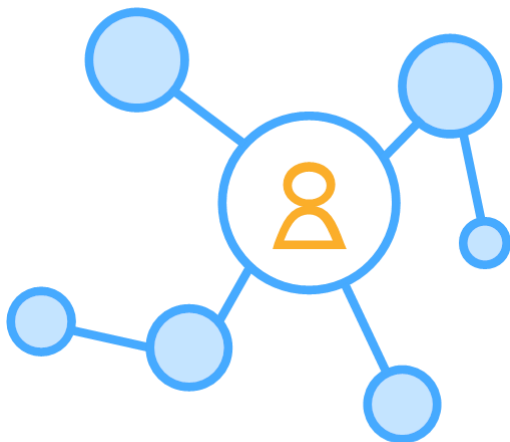
Les relations entre objets

Les différentes relations

L'association

La composition

L'héritage

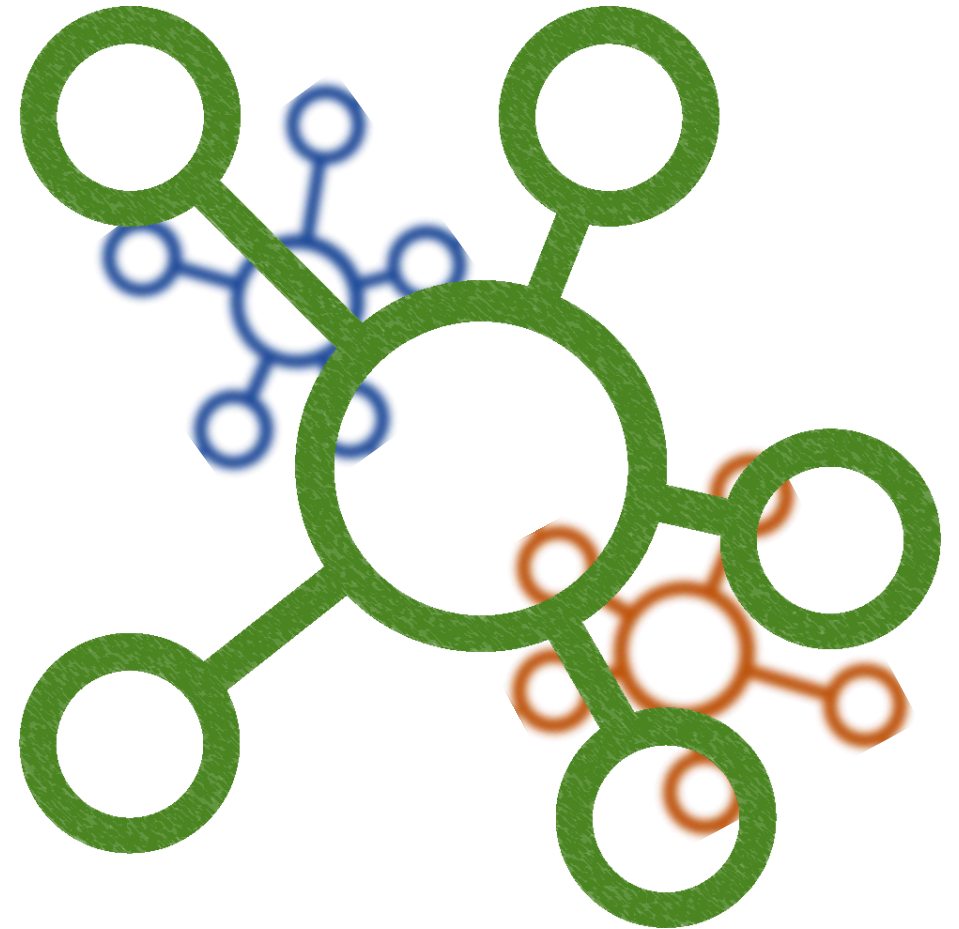


Les différentes relations

Les objets ainsi créés pourront être multiples et avoir des relations, avec leur environnement mais aussi les uns avec les autres.

On distingue 3 grands types de relation :

- L'association
- La composition
- L'héritage



L'association

La relation d'association entre deux objet consiste à les faire collaborer pendant un court instant.

Concrètement, il s'agira par exemple de l'intervention d'un premier objet dans la méthode d'un second. L'association est donc éphémère.

Ci-contre, une classe marchand. Chaque marchand dispose de crédit et de carottes qu'il pourra échanger avec un autre marchand.

```
19 class Merchant {
20
21     private double credits ;
22     private int carrots ;
23
24     Merchant( double credits, int carrots ) {
25         this.credits = credits ;
26         this.carrots = carrots ;
27     }
28
29     public void buy_carrots( Merchant from, int quantity ) {
30         double carrot_price = 2.25 ;
31         from.remove_carrots( quantity ) ;
32         this.carrots += quantity ;
33         from.add_credits( quantity * carrot_price );
34     }
35
36     public void add_credits( double credits ){
37         this.credits += credits ;
38     }
39
40     public void remove_carrots( int quantity ) {
41         this.carrots -= quantity ;
42     }
43
44     public void show_details() {
45         System.out.println(
46             "Credits : " + this.credits +
47             " / Carrots : " + this.carrots
48         );
49     }
50
51 }
```

L'association

On peut déjà s'apercevoir que la méthode `buy_carrots` fait intervenir un autre marchand.

C'est là qu'intervient notre relation éphémère, notre association. Nos deux marchand collaborent le temps d'un échange.

```
public void buy_carrots( Merchant from, int quantity ) {  
    double carrot_price = 2.25 ;  
    from.remove_carrots( quantity ) ;  
    this.carrots += quantity ;  
    from.add_credits( quantity * carrot_price );  
}
```

L'association

Cette association pourra être initiée par l'un ou l'autre des parties.

```
public static void main( String[] args ) {  
  
    Merchant albert = new Merchant( 20, 10 );  
    Merchant lucien = new Merchant( 150, 0 );  
  
    lucien.buy_carrots( albert, 5 );  
  
    albert.show_details();  
    lucien.show_details();  
  
}
```

```
Credits : 31.25 / Carrots : 5  
Credits : 150.0 / Carrots : 5
```

Remarquez que l'échange a bien eu lieu après l'appel à notre fonction **buy_carrots**, le tout en respectant le principe d'encapsulation.

La composition

La composition est une relation plus long terme. Un objet sera composé d'un autre. Il s'agit donc d'une dépendance plus forte que l'association.

Concrètement, il s'agira d'un objet étant attribut d'un autre objet et lui étant utile au cours de différentes méthodes.

Prenons par exemple cette nouvelle classe **Car** ci-contre. Chaque voiture est équipée d'un moteur, ici sous la forme d'une autre classe **Engine**.

```
30 class Car {
31
32     private Engine engine ;
33     private double speed ;
34
35     Car( Engine e ) {
36         this.engine = e ;
37         this.speed = 0 ;
38     }
39
40     public void accelerate() {
41         this.speed += 0.2 * engine.get_power() ;
42     }
43
44     public void show_details() {
45         System.out.println(
46             "This car is equipped by a " + engine.get_power()
47             + " hp engine and goes at " + this.speed + " km/h "
48         );
49     }
50
51 }
```


La composition

Voici d'ailleurs l'implémentation de la classe **Engine**.

Dans notre cas, **Engine** compose **Car** puisqu'il est un de ses attributs.

La relation est donc beaucoup plus long terme qu'une simple association éphémère.

```
--  
16  ✓ class Engine {  
17  
18      private double power ;  
19  
20  ✓  Engine( double power ) {  
21      |      this.power = power ;  
22      |  }  
23  
24  ✓  double get_power() {  
25      |      return this.power ;  
26      |  }  
27  
28  }
```

La composition

Cette composition peut être obligatoire dès le début de l'existence de nos objet, comme c'est le cas ici avec notre constructeur de **Car**.

Mais il pourrait en être autrement. On pourrait imaginer rendre optionnel un moteur, seulement il faudrait certainement adapter notre méthode **accelerate**.

```
public static void main( String[] args ) {  
  
    Engine engine = new Engine( 150 );  
    Car my_car = new Car( engine );  
    my_car.accelerate();  
    my_car.show_details();  
}
```

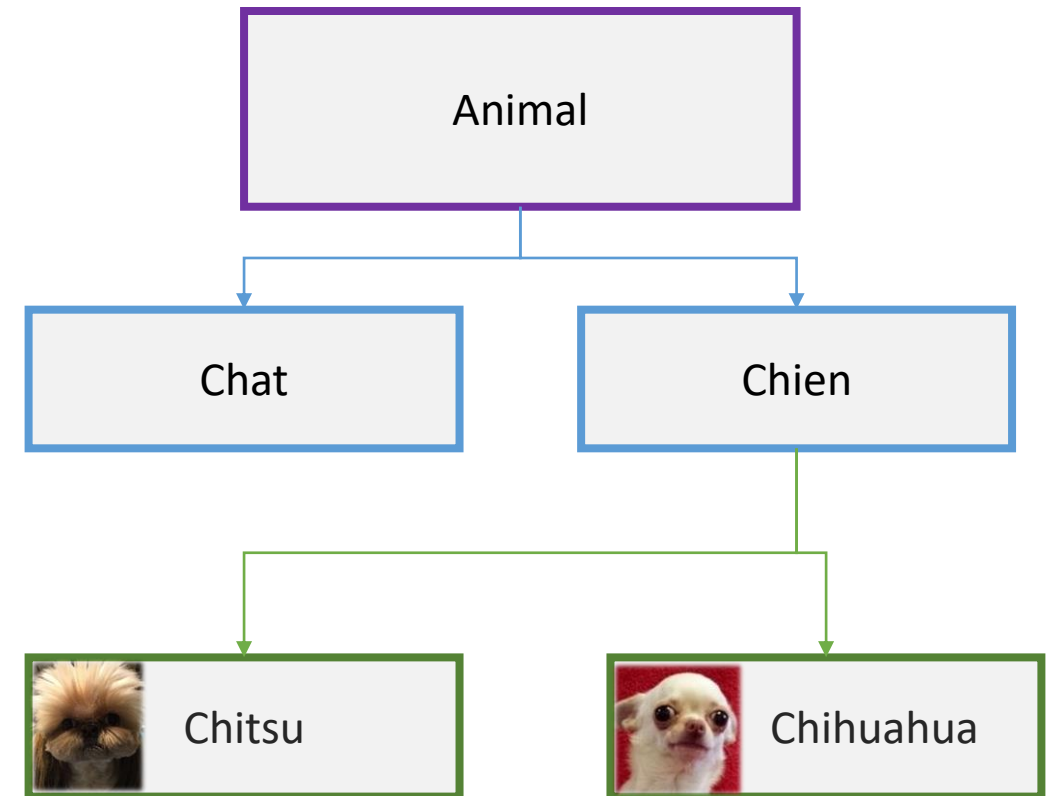
This car is equipped by a 150.0 hp engine and goes at 30.0 km/h

L'héritage

Il existe une relation très importante n POO : l'héritage. Elle permet de créer une nouvelle classe à partir d'une autre.

Ce faisant, la nouvelle classe, ou classe fille, obtiendra d'emblé les attributs et méthodes de la classe d'origine (ou classe mère).

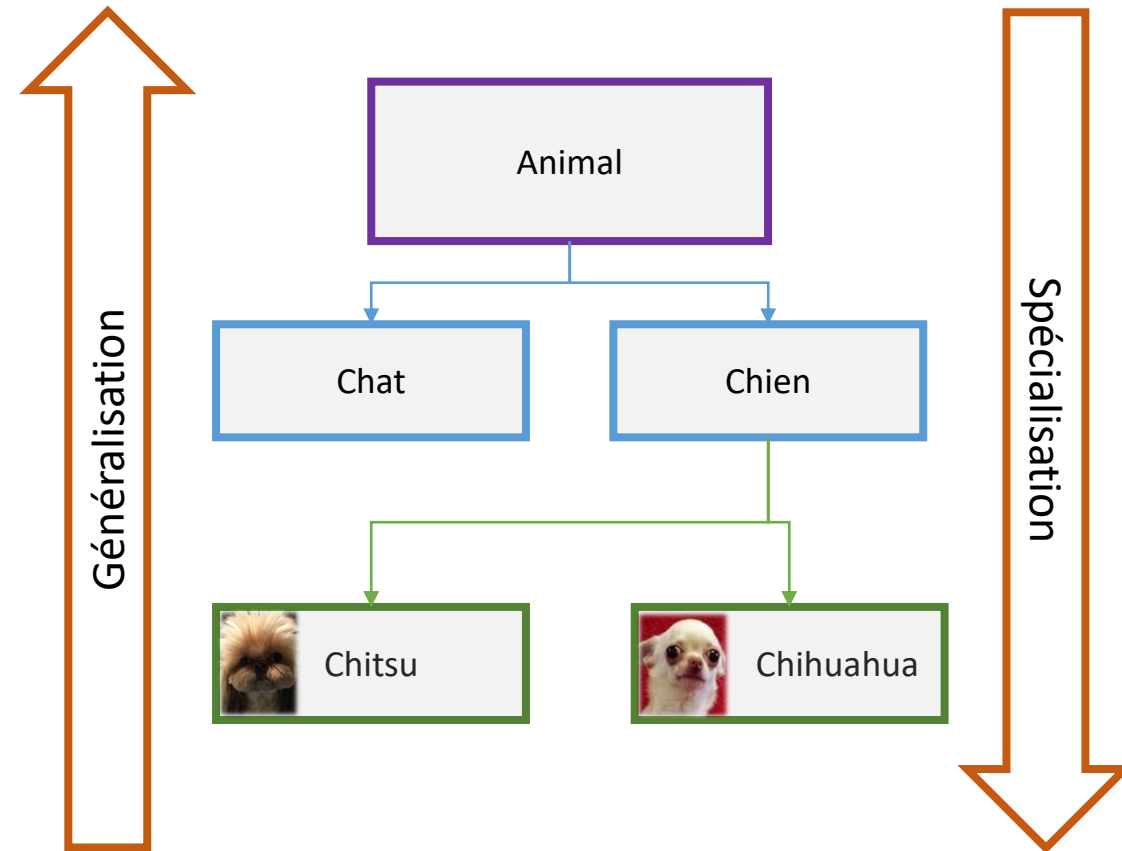
On pourra alors spécialiser la classe fille en lui ajoutant de nouvelles propriétés et méthodes.



L'héritage

On parlera donc, dans le sens de la classe mère vers les filles, de spécialisation, tandis que dans le sens inverse, on parlera de généralisation.

La généralisation c'est donc le regroupement de plusieurs classes sous un même type.

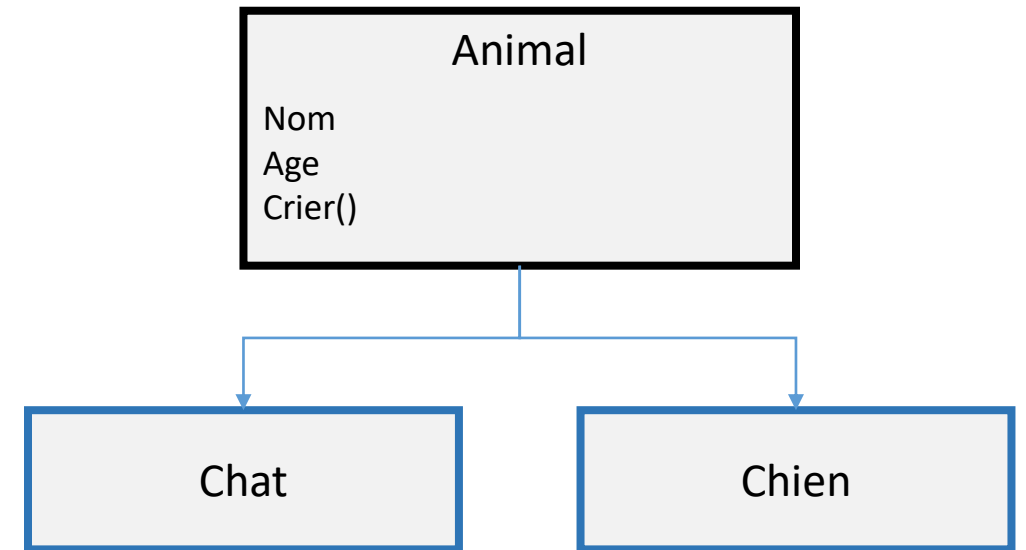


L'héritage

Voici un exemple plus concret de l'héritage précédemment schématisé.

Tout animal pourra posséder les attributs **age** et **nom**, ainsi qu'une méthode **crier**. Bien évidemment, cette méthode est assez vague.

On pourra alors créer par héritage d'autres classes, comme chat et chien, qui hériteront des ces attributs et méthodes.



L'héritage

Venons en à la syntaxe. Pour définir une nouvelle classe en héritant des attributs et méthodes d'une autre, on utilisera le mot clef **extends** à la suite de sa définition.

Attention, on ne peut hériter que d'une seule classe à la fois en Java.

Avec l'exemple ci-contre, remarquez qu'il est possible d'utiliser notre **Chien** comme un **Animal** sans lui avoir défini quoi que ce soit de plus.

```
3  class Main {
4
5      public static void main( String[] args ) {
6
7          Chien max = new Chien("Max", 6);
8          max.crier();
9
10     }
11 }
12
13
14 class Animal {
15
16     private String name ;
17     private int age ;
18
19     Animal( String name, int age ) {
20         this.name = name ;
21         this.age = age ;
22     }
23
24     public void crier() {
25         System.out.println(
26             "L'animal " + this.name + " de " + this.age + " ans crie !"
27         );
28     }
29 }
30
31
32
33 class Chien extends Animal {}
```

L'animal Max de 6 ans crie !

L'héritage - *Surcharge*

Il est maintenant possible de spécialiser notre classe fille, en lui ajoutant des attributs et des méthodes, comme ici la méthode **jouer_avec_un_balle**.

Cependant, on peut aussi spécialiser notre classe en redéfinissant ou complétant des méthodes existantes.

Ci-contre, nous redéfinissons la méthode **crier** pour qu'elle reflète mieux le comportement d'un chien.

```
class Chien extends Animal {  
  
    public void crier() {  
        System.out.println(  
            "Le chien " + this.name + " de " + this.age + " ans aboie !"  
        );  
    }  
}
```

L'héritage - *Surcharge*

En toute logique, nous pouvons donc utiliser notre chien en le faisant crier, ce qui utilisera notre méthode fraîchement mise à jour.

Cependant, nous allons être confrontés à quelques problèmes avant d'en arriver là.

Comme vous le voyez, nous obtenons une erreur lors de l'exécution : **age** et **name** sont des attributs privés !

```
public static void main( String[] args ) {  
  
    Chien max = new Chien("Max", 6);  
    max.crier();  
  
}
```

```
.\Main.java:37: error: name has private access in Animal  
        "Le chien " + this.name + " de " + this.age + " ans aboie !"   
                           ^  
.\Main.java:37: error: age has private access in Animal  
        "Le chien " + this.name + " de " + this.age + " ans aboie !"   
                           ^
```


L'héritage — *Visibilité protected*

Et oui, comment utiliser nos attributs privés à Animal dans notre classe fille Chien.

Il nous serait possible d'utiliser des getters, c'est une solution.

Une seconde solution est de rendre ces attributs protégés avec le mot clef **protected**, ce qui signifie que nos attributs ne seront accessibles que dans la famille de la classe (ses enfants donc).

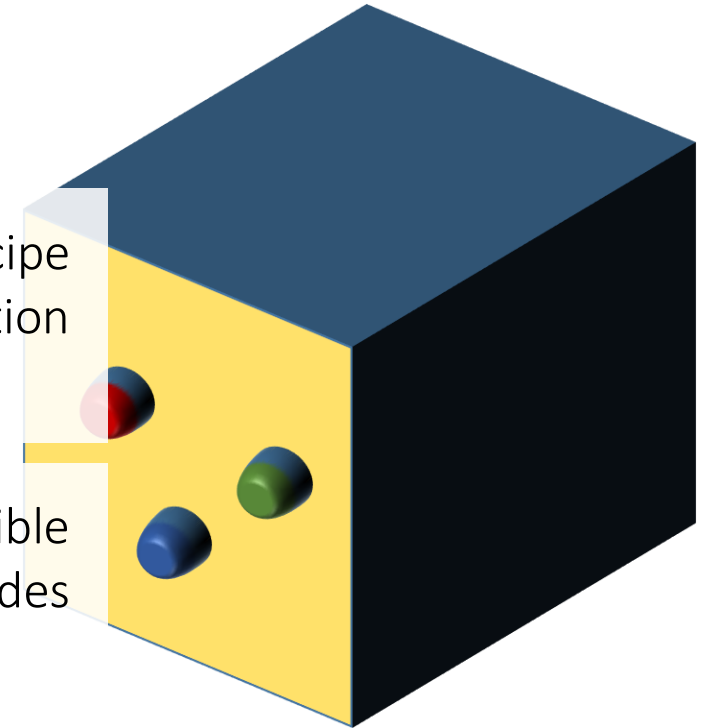
```
class Animal {  
  
    protected String name ;  
    protected int age ;  
}
```

L'héritage — *Visibilité protected*

Pour clarifier ce qui a été dit plus tôt sur le principe d'encapsulation, redéfinissons l'affirmation suivante.



Les attributs doivent être le plus souvent possible **private** ou **protected** et les méthodes **public**.



L'héritage — *Constructeur de classe fille*

Malgré tout, il nous reste encore une erreur après ce problème de visibilité réglé.

```
.\Main.java:33: error: constructor Animal in class Animal cannot be applied to given types;  
class Chien extends Animal {  
^  
  required: String,int  
  found: no arguments  
  reason: actual and formal argument lists differ in length
```

Notre classe Chien ne semble pas pouvoir initialiser ses attributs hérités plus tôt.

L'héritage – *Super*

```
class Chien extends Animal {  
  
    chien( String name, int age ) {  
        |     super( name, age );  
    }  
}
```

Le chien Max de 6 ans aboie !

Pour nous servir correctement de nos attributs hérités, nous allons devoir redéfinir un constructeur propre à notre classe fille.

Il nous faudra obligatoirement faire appel au constructeur de la classe mère.

Nous avons vu qu'il était possible de faire appel à un autre constructeur de notre classe avec **this**, mais pour atteindre notre classe mère, il nous faudra utiliser **super**.

L'héritage — *Super*

Il sera par la suite possible d'ajouter un attribut spécifique au Chien, en l'ajoutant au constructeur du chien après l'usage de **super**.

```
class Chien extends Animal {  
    protected String color ;  
  
    Chien( String name, int age, String color ) {  
        super( name, age );  
        this.color = color ;  
    }  
  
    public void crier() {  
        System.out.println(  
            "Le chien " + this.name + ", " + this.color +  
            ", de " + this.age + " ans, aboie !"  
        );  
    }  
}
```



Pause questions

Chapitre 3



Les Exceptions

Qu'est ce qu'une exception ?

Réagir à une exception

Lancer une exception

Créer ses propres exceptions

Qu'est ce qu'une exception



Lorsqu'une Java détecte un état anormal lors de l'exécution, celui-ci lève ce qu'on appelle une exception.

Ces exception ne sont pas forcément fatales et sont même parfois normales.

Il peut très bien s'agir d'un état particulier comme la perte de connexion à une application. Le tout est d'y réagir !

Qu'est ce qu'une exception

Le code ci-contre par exemple vous semblera sûrement d'emblé problématique avec sa division par 0.

```
double x = 1 / 0 ;
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java va bien évidemment s'en rendre compte et nous indiquera dans la console qu'une erreur **inattendue** est survenue.

L'erreur indiquée mentionne d'ailleurs l'exception qui est levée : **java.lang.ArithmeticException** avec une petite description.

Réagir à une exception

Si Java a dû interrompre le programme, c'est que seul, il ne saura pas comment réagir à ce problème.

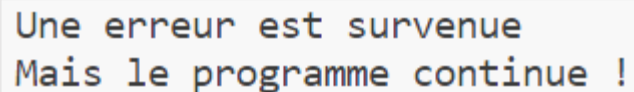
Fort heureusement, il existe une structure permettant de capter ces exceptions et de programmer un scénario en réaction.

Il s'agit de la structure **try catch** dont le **try** englobe le code susceptible de lever une exception et le **catch** permet de programmer une réaction.

```
try {  
    double x = 1 / 0 ;  
}  
catch( Exception e ) {  
    System.out.println("Une erreur est survenue");  
}
```

Réagir à une exception

```
try {  
    double x = 1 / 0 ;  
}  
catch( Exception e ) {  
    System.out.println("Une erreur est survenue");  
}  
  
System.out.println("Mais le programme continue !");
```



Une erreur est survenue
Mais le programme continue !

Outre le fait de pouvoir indiquer à Java comment réagir à une exception, le **try catch** évite également une interruption brutale du programme.

Dans le cas où une exception est résolue par un catch, le programme reprendra son cours après celui-ci.

Réagir à une exception

Notez également qu'une `try` ne déclenche pas systématiquement un `catch`, il est là pour y réagir si nécessaire.

```
Scanner scan = new Scanner( System.in );
System.out.print("Diviseur > ");
int diviseur = scan.nextInt();

try {
    int x = 1 / diviseur ;
}
catch( Exception e ) {
    System.out.println("Une erreur est survenue");
}

System.out.println("Fin du programme");
```

Diviseur > 1
Fin du programme

Choix d'un diviseur à 1

Diviseur > 0
Une erreur est survenue
Fin du programme

Choix d'un diviseur à 0

Réagir à une exception

Notez le Exception e dans les parenthèses du **catch**. Il permet non seulement de récupérer l'erreur, mais aussi de filtrer les erreurs.

Il nous est par exemple possible de l'afficher avec un simple **print**.

```
catch( Exception e ) {  
    System.out.println("Une erreur est survenue : " + e);  
}
```

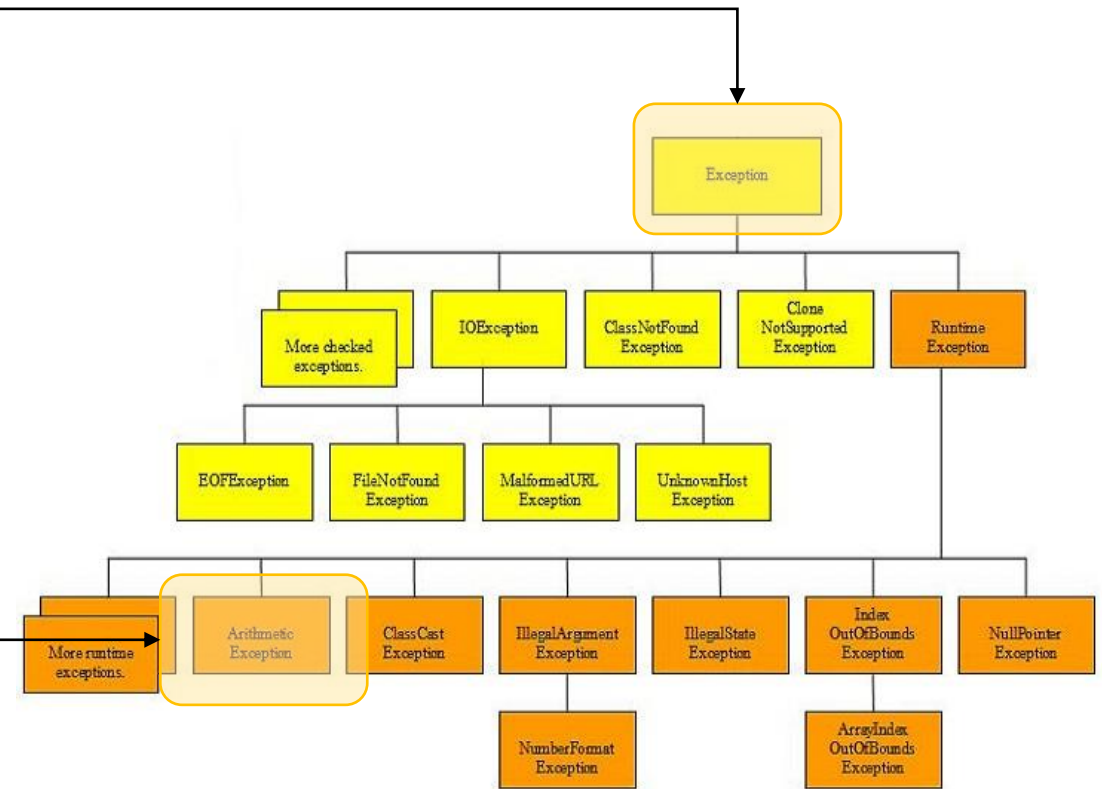
```
Une erreur est survenue : java.lang.ArithmeticException: / by zero
```

Réagir à une exception

Mais pourquoi utilisons nous ici **Exception** alors que nous savons que l'exception déclenchée est ici une **ArithmeticException** ?

En réalité, Exception est la classe mère de toutes les exception. **ArithmeticException** est donc une Exception.

Un catch ne se déclenchera que si l'exception levée correspond à celle qu'on lui a assignée en argument.



Réagir à une exception

Le fait de renseigner **Exception** nous permet donc de réagir à diverses erreurs.

```
try {  
    int[] arr = new int[1];  
    arr[5] = 20 ;  
}  
catch( Exception e ) {  
    System.out.println("Exception détectée : " + e);  
}  
  
System.out.println("Fin du programme");
```

```
Exception détectée : java.lang.ArrayIndexOutOfBoundsException:  
Index 5 out of bounds for length 1  
Fin du programme
```

```
catch( ArithmeticException e ) {  
    System.out.println("Exception détectée : " + e);  
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  
Index 5 out of bounds for length 1  
    at Main.main(Main.java:13)
```

Renseigner une **Exception** plus précise, comme **ArithmeticException**, nous permet alors d'être plus sélectif.

Réagir à une exception

Il est d'ailleurs possible de mettre en place plusieurs `catch` à la suite, de manière à programmer différentes réaction à différentes Exceptions.

Comme pour la structure `if` avec `else if`, les `catch` s'exécuteront dans l'ordre jusqu'à en trouver un qui corresponde.

Dès qu'un `catch` correspond, il s'exécute et ignore les suivants.

```
catch( ArrayIndexOutOfBoundsException e ) {  
    System.out.println("Cet index n'existe pas dans votre Array !");  
}  
catch( ArithmeticException e ) {  
    System.out.println("On ne divise pas par zéro, merci !");  
}  
catch( Exception e ) {  
    System.out.println("Une erreur est survenue : " + e);  
}
```


Réagir à une exception

Enfin, il existe un dernier bloc possible avec le **try** **catch** : le **finally**.

Ce nouveau bloc ne prend aucun paramètre et agira à la manière d'un **else**.

Il s'exécutera à la fin du **try**, quelque soit les circonstances (exception levée ou pas, et même en cas d'erreur).

```
try {  
    int x = 1/0;  
}  
catch( ArrayIndexOutOfBoundsException e ) {  
    System.out.println("Cet index n'existe pas dans votre Array !");  
}  
finally {  
    System.out.println("Fin de try");  
}
```

```
Fin de try  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Main.main(Main.java:12)
```

Lancer une exception

```
try {  
    throw new ArithmeticException( "Hello, je suis une ArithmeticException ! ");  
}  
catch( ArithmeticException e ) {  
    System.out.println("Ho ? Une ArithmeticException : " + e);  
}
```

```
Ho ? Une ArithmeticException : java.lang.ArithmeticException:  
Hello, je suis une ArithmeticException !
```

Il est également possible de lancer soit même une Exception, pour signaler à l'utilisateur de la fonction qu'un événement problématique est survenu.

On utilisera le mot clef **throw** pour ce faire, suivi de l'instance de l'Exception souhaitée.

La plupart des exceptions disposent d'un constructeur simple, prenant comme unique argument une chaîne de caractère descriptive de l'exception.

Créer ses propres exceptions

Le mot clef **throw** est d'autant plus intéressant lorsque l'on crée nos propres exceptions.

Celle-ci pourront alors faire spécifiquement référence aux problèmes de notre application et être plus clairs.

Pour créer notre exception, il suffit de créer une classe héritant de l'exception la plus proche. Le plus général est d'hériter d'**Exception**.

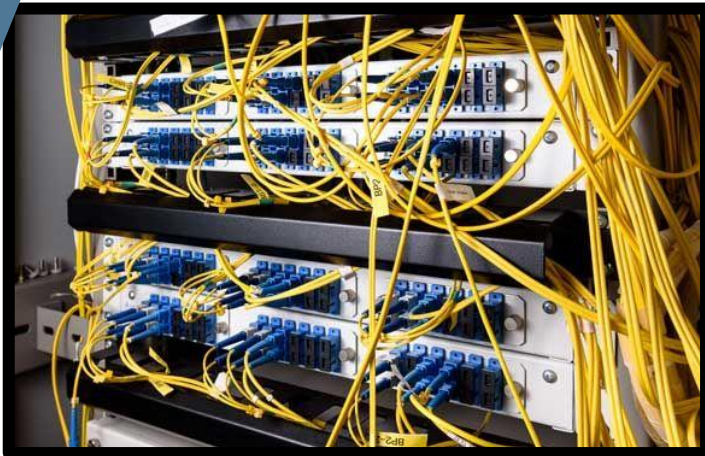
```
class MyException extends Exception {  
    MyException( String infos ) {  
        super( infos );  
    }  
}  
  
try {  
    throw new MyException( "Hello, je suis une MyException ! " );  
}  
catch( MyException e ) {  
    System.out.println("Ho ? Une ArithmeticException : " + e);  
}
```

```
Ho ? Une ArithmeticException :  
MyException: Hello, je suis une MyException !
```



Pause questions

Chapitre 4



Créer et organiser son application

Organiser ses classes

Packages

Compilation

Créer un exécutable

Exécuter l'application

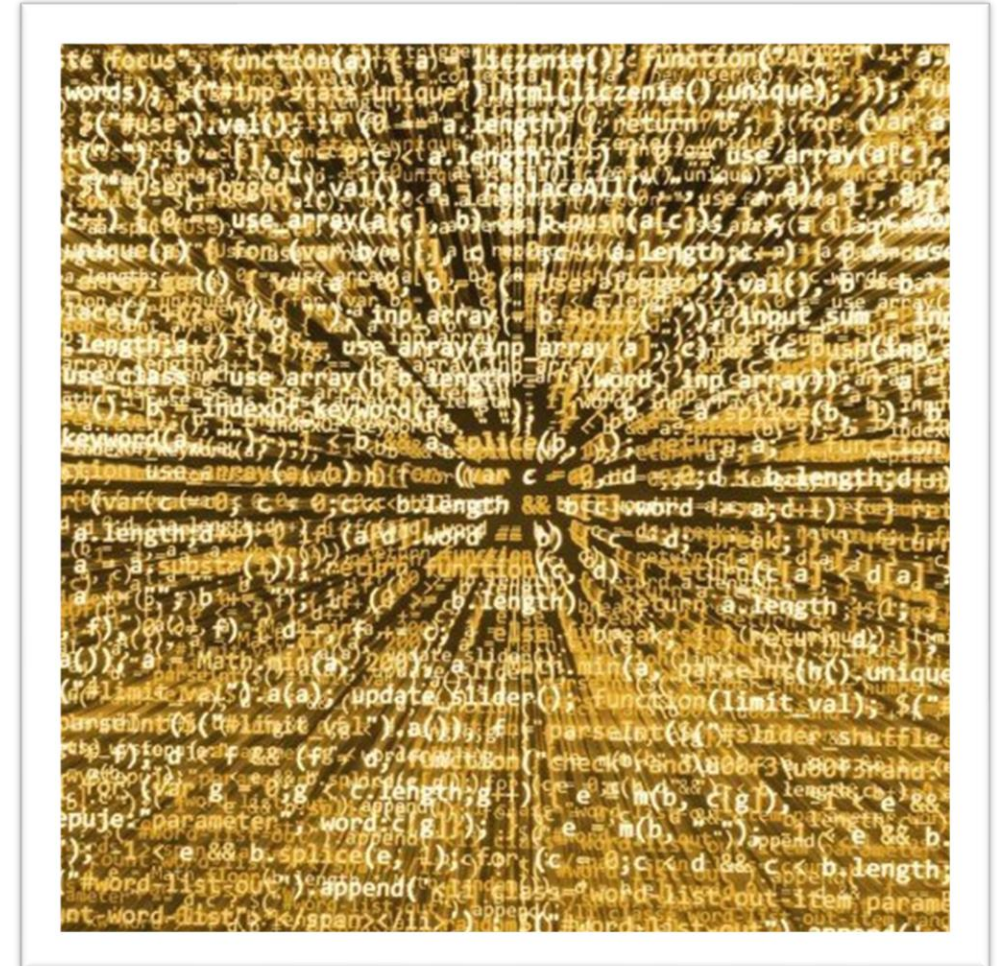
Le Manifest

Organiser ses classes

Jusqu'à présent nous avons toujours utilisé un seul et même fichier pour écrire notre code.

Cela étant, un seul fichier, c'est parfois étroit pour un certain nombre de classes.

C'est pourquoi, il est fortement recommandé de faire usage de plusieurs fichiers, plus exactement d'un fichier par classe.



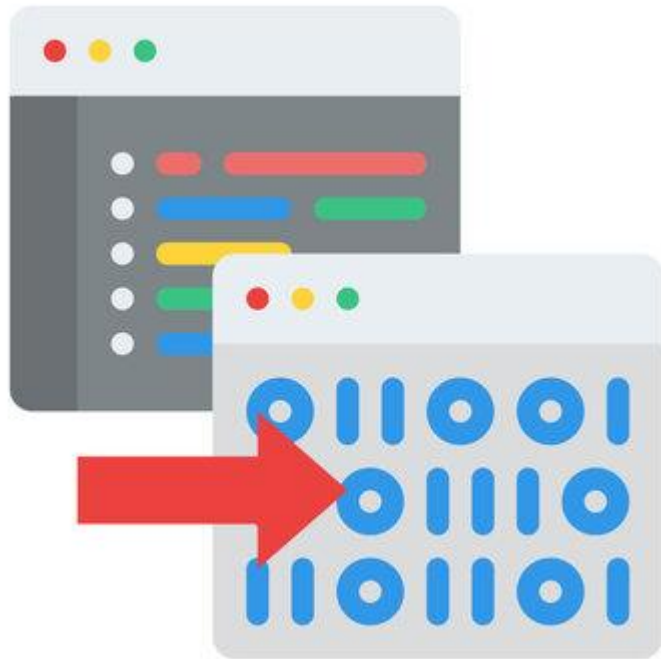
Organiser ses classes

Reprenons notre code sur les animaux par exemple.
Nous avons 3 classes : Main, Animal et Chien.

Créons donc 3 fichiers distincts pour chacune de ces classes, portant le même nom (attention aux majuscules).



Compilation



Après avoir transféré chaque code de classes dans les fichiers appropriés, nous pouvons de nouveau tenter d'exécuter notre Main.

Mais bien évidemment, il ne nous sera pas possible d'utiliser nos classes de fichiers différents comme par magie.

Il va nous falloir les compiler avec **javac**. Etant donné que tous nos fichiers sont dans le même dossier, nous pouvons utiliser la commande :

```
javac *.java
```


Packages



Et effectivement, après avoir exécuté notre Main (avec `java Main` pour rappel), le programme fonctionne.

En réalité, nos classes ont été automatiquement importées comme si elles faisaient partie d'un même ensemble.

Cet ensemble, java l'appelle **package**. Il nous sera possible, surtout lorsque nous utiliserons des sous-dossier, de les définir nous même.

Packages



Créons donc un nouveau sous-dossier pour l'expérience. Ce sera le sous dossier **animals** dans lequel nous déplacerons nos classes.

Nous devons ici indiquer aux classes **Animal.java** et **Chien.java** qu'elles font partie du même package.

D'autre part, il faudra importer notre classe **Animal** dans **Chien.java** et **Chien** dans **Main.java**.

Packages - *Définition*

Pour définir le package d'un fichier, il suffit de le définir, en tout début de fichier, avec le mot clef **package**.

```
1 package animals ;
```

A la suite de celui-ci, on définira le package dont fait partie le fichier.

Ci-contre, un exemple avec nos fichier **Animal.Java** et **Chien.Java** qui feront tous deux partie du package **animals**.

Packages — *Convention de nommage*

Ici notre package a été nommé simplement pour ne pas complexifier l'exemple, mais si notre application prend de l'ampleur, elle pourrait rencontrer des conflits.

Il existe de nombreux packages à travers le monde qui pourraient porter le même nom.

Par convention, on nommera donc nos package sous un ensemble représentant une sorte de nom de domaine inversé.



Packages — *Convention de nommage*

Imaginons par exemple travailler chez javaprogrammers.com.

Chacun de nos packages se trouveraient donc sous l'arborescence `com.javaprogrammer.<package>`.

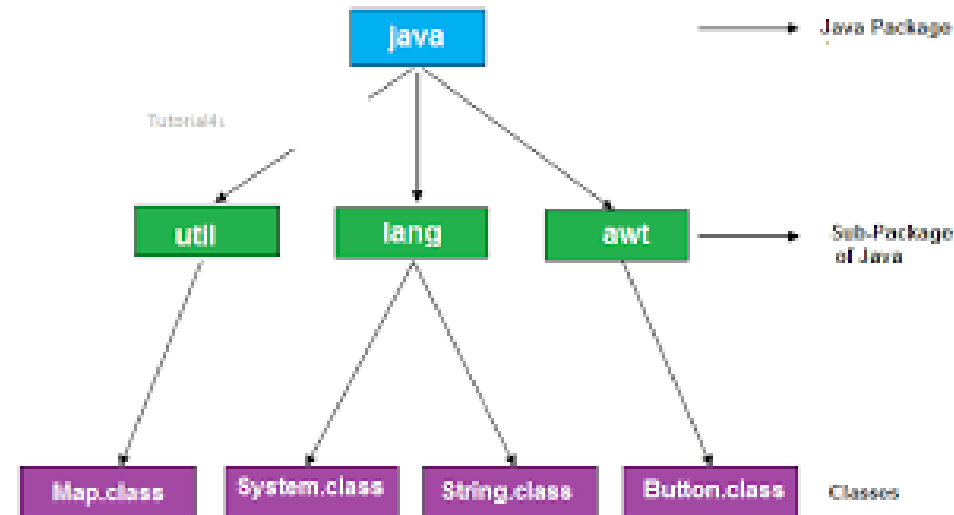
Dans le cas de notre package `animals` donc, il s'agirait du package `com.javaprogrammers.animals`.



Packages — *Convention de nommage*

Attention, cela est valable pour les packages externes à Java.

Vous remarquerez que les packages internes à java, comme `Scanner` utilisé plus tôt, se trouvent dans l'arborescence `java.lang`, `java.util`, etc...



Packages - *Utilisation*

Rendez-vous dans notre fichier **Main.java** à présent. Nous avons besoins de notre classe **Chien**.

Pour l'importer, nous utiliserons le mot clef **import** suivi du nom du package.

A la suite du package, séparé par un « . », nous pourrons sélectionner l'élément à importer, c'est-à-dire dans notre cas, la classe **Chien**.

```
1  import animals.Chien ;
2
3  class Main {
4
5      public static void main( String[] args ) {
6
7          Chien max = new Chien("Max", 6, "noir et blanc");
8          max.crier();
9
10     }
11
12 }
```

Packages - *Utilisation*

Même opération dans **Chien.java** où nous importerons la classe **Animal**.

```
1  package animals ;
2  import animals.Animal ;
3
4  ✓ public class Chien extends Animal {
5      |
6      |     protected String color ;
7      |
8      ✓ public Chien( String name, int age, String color ) {
9          |     super( name, age );
10         |     this.color = color ;
11         | }
```


Créer un exécutable

Lorsque votre programme sera terminé, vous voudrez sans doute pouvoir le distribuer.

Nous pourrions distribuer nos `.class`, les fichiers étant compilés et fonctionnels. Mais c'est potentiellement beaucoup de fichiers à envoyer.

Plus élégant, il est possible en Java de créer un exécutable unique : le `.jar`.



Créer un exécutable

Tout d'abord, par confort, nous allons compiler notre programme différemment.

Nous allons créer un dossier **dist** à la racine de notre projet dans lequel placer tous nos fichiers compilés.

Pour ce faire, ajoutez l'option **-d dist** pour définir le dossier **dist** comme dossier de destination pour nos fichiers compilés.

```
javac -d dist/ animals/*.java  
javac -d dist/ Main.java
```



A file explorer view showing the project structure. It has a tree view on the left and a file list on the right. The tree view shows a folder named 'animals' which is expanded, showing 'Animal.java' and 'Chien.java'. Below 'animals' is a folder named 'dist', which is also expanded. Inside 'dist', there is a sub-folder 'animals' containing 'Animal.class' and 'Chien.class', and a file 'Main.class'. At the bottom of the file list is 'Main.java'.

- ▼ animals
 - Animal.java
 - Chien.java
- ▼ dist
 - ▼ animals
 - Animal.class
 - Chien.class
 - Main.class
- Main.java

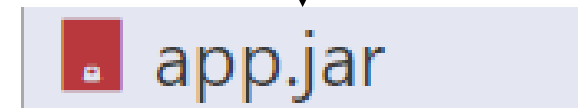
Créer un exécutable

Rendez-vous ensuite dans le dossier **dist** pour y créer l'exécutable.

Nous aurons besoins du programme **jar**, déjà installé avec le programme d'installation de Java.

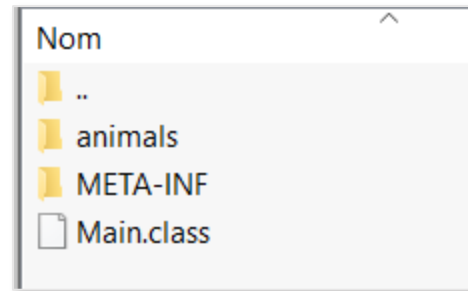
Pour créer votre exécutable avec vos fichiers compilés, exécutez **jar -cvf app.jar ***, où **app.jar** est le nom de l'exécutable à créer et ***** représente toutes les sources du dossier.

```
. cd dist
dist> jar -cvf app.jar *
manifeste ajouté
ajout : animals/(entrée = 0) (sortie = 0)(stockage : 0 %)
ajout : animals/Animal.class(entrée = 1175) (sortie = 644)(compression : 45 %)
ajout : animals/Chien.class(entrée = 1059) (sortie = 569)(compression : 46 %)
ajout : Main.class(entrée = 397) (sortie = 283)(compression : 28 %) -
```



Créer un exécutable

Notez que notre **.jar** n'est autre qu'une archive qu'il est d'ailleurs possible d'ouvrir.



Un aperçu ici de notre **.jar** avec **winrar**. Nous retrouvons notre arborescence précédente ainsi qu'un nouveau fichier : le Manifest.

Exécuter l'application

Pour exécuter notre application fraîchement créée, il va nous falloir faire appel à java avec la commande `java -jar app.jar`.

Cependant, java ne saura pas seul quel fichier, dans notre archive, contient la classe Main.

```
Erreur : impossible de trouver ou de charger la classe principale .\app.jar  
Causé par : java.lang.ClassNotFoundException: /\app/jar
```

Exécuter l'application

Pour le lui indiquer, nous utiliserons l'option `-cp` (classpath) lors de l'exécution :

```
java -cp app.jar Main
```

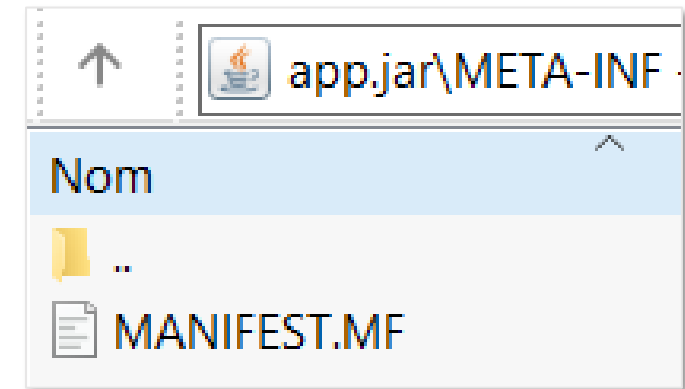
```
Animal n°1 créé !  
Le chien Max, noir et blanc, de 6 ans, aboie !
```

Le Manifest

Vous l'aurez sans doute remarqué, lorsqu'on explore l'archive `.jar`, il y a un dossier qui ne fait pas partie de nos `.class`.

Un dossier **META-INF** a été automatiquement créé par jar avec à l'intérieur un **MANIFEST.MF**.

Ce Manifest est le fichier de configuration de notre `.jar`, et il est possible d'en éditer les propriétés.



Le Manifest

Voici le contenu de ce manifest qui devrait être similaire chez vous. Il contient, par ligne une clef et une valeurs séparés par un « : ».

La clef **Manifest-Version** permet d'indiquer la version de la spécification du manifest (1.0 étant la version en vigueur).

La clef **Created-By** indique le système l'ayant généré ainsi que sa version.

```
1 Manifest-Version: 1.0
2 Created-By: 11.0.12 (Microsoft)
3 |
```


Le Manifest

Il est possible de modifier ce manifest, avant ou après la génération du .jar.

La clef **Main-Class** par exemple, permet de définir la classe où se trouve le main, qu'on appelle aussi l'**entry point**.

Après l'avoir défini, plus besoins de spécifier notre class principale lors de l'exécution, il suffira d'ajouter l'option **-jar**.

```
1 Manifest-Version: 1.0
2 Created-By: 11.0.12 (Microsoft)
3 Main-Class: Main
4 |
```

```
java -jar .\app.jar
Animal n°1 créé !
Le chien Max, noir et blanc, de 6 ans, aboie !
```

Le Manifest

L'option **Class-Path** permettra quant à elle de faire référence à d'autre .jar pour en importer les fonctionnalités.

```
1 Manifest-Version: 1.0
2 Created-By: 11.0.12 (Microsoft)
3 Main-Class: Main
4 Class-Path: other.jar utils.jar
5 |
```

Le Manifest

Voici quelques autres clefs informatives sur la nature du package et son versionnement :

Clef	Description
Name	Nom du package
Specification-Title	Titre de la specification
Specification-Version	Version de la spécification
Specification-Vendor	Vendeur de la spécification
Implementation-Title	Titre de l'implémentation
Implementation-Version	Numéro de build de l'implémentation
Implementation-Vendor	Vendeur de l'implémentation



Pause questions

Chapitre 5



Les ensembles

Les ArrayList

Les LinkedList

Les HashMap

Les HashSet

Les Iterator



Les ArrayList

Souvenez vous dans la partie 1 chapitre 4, nous avons vu les **Array** qui permettent de contenir un ensemble de données d'un même type.

Ceux-ci étaient limités dans leur utilisation par leur impossibilité à être redimensionnés notamment.

A partir de là, leur manipulation devient compliquée et il faut recréer un nouvel **Array** pour tout changement de dimension.



Les ArrayList

Fort heureusement, il existe des Objets très utiles en ce qui concerne la manipulation des ensembles et qui remplaceront la plupart du temps nos **Array**.

Le premier d'entre eux est l'**ArrayList** qui est tout bonnement un **Array** avec des outils facilitant son usage.

Pour nous en servir, il va tout d'abord falloir l'importer depuis **java.util**.

```
import java.util.ArrayList ;
```

Les ArrayList - *Création*

Nous allons ensuite pouvoir l'instancier en lui précisant le type d'élément qu'il devra comporter.

Cette syntaxe nouvelle fait intervenir les « <> » en fin de type dans lesquels nous renseignerons ce type.

```
ArrayList<String> students = new ArrayList<String>();
```


Les ArrayList - *Ajout*

A partir de là, tout devient plus simple. Pour ajouter un élément, on utilisera la méthode **add**.

Il suffit alors de lui fournir en argument l'élément à ajouter, qui doit impérativement être du type précisé lors de l'instanciation.

```
ArrayList<String> students = new ArrayList<String>();  
  
students.add( "George" );  
students.add( "Maria" );  
students.add( "Elvis" );  
students.add( "Jean-Jacques" );
```

Les ArrayList - *Print*

Pour superviser rapidement le contenu de notre ensemble, un simple **print** suffit.

Il sera alors affiché en format JSON, très simple à comprendre : entre « `[]` », chaque élément séparé par une « `,` ».

```
students.add( "Elvis" );  
students.add( "Jean-Jacques" );  
  
System.out.println( students );
```

```
[George, Maria, Elvis, Jean-Jacques]
```

Les ArrayList - *Lecture*

Pour lire un élément à un index précis, il suffira d'utiliser la méthode **get**.

On lui fournira en argument cet index. La valeur de retour sera alors l'élément à cet index.

```
System.out.println( students.get(2) );
```



Elvis

Les ArrayList - *Lecture*

Pour en obtenir la taille, nous pourrions également utiliser la méthode `size`.

```
System.out.println( students.size() );
```

4

```
System.out.println( students.size() );  
students.clear();  
System.out.println( students.size() );
```

4

0

Pour en retirer la totalité du contenu, on utilisera la méthode `clear`.

Les ArrayList - *Modification*

Pour modifier un élément à un index précis, il suffira d'utiliser la méthode **set**.

Celle-ci prend en premier paramètre l'index à modifier et en second sa nouvelle valeur.

```
students.add( "George" );  
students.add( "Maria" );  
students.add( "Elvis" );  
students.add( "Jean-Jacques" );  
  
students.set(1, "Alice");  
System.out.println( students );
```

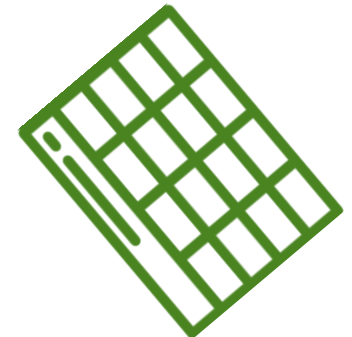
```
[George, Alice, Elvis, Jean-Jacques]
```

Les LinkedList

Les **LinkedList** sont très similaires avec les **ArrayList**, elles possèdent d'ailleurs toutes deux des fonctionnalités similaires.

Cependant, leur principe de fonctionnement n'est pas le même et votre cas d'usage décidera duquel choisir.

Si vous souhaitez stocker de la donnée, on utilisera **ArrayList**, si vous souhaitez manipuler de la donnée, on utilisera **LinkedList**.



Les LinkedList

Elle possède donc toutes les fonctionnalités vues précédemment avec en plus celle-ci-dessous :

Méthode	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list
addFirst()	Adds an item to the beginning of the list.

Les LinkedList

Un petit exemple d'usage des `LinkedList` avec une succession de ces nouvelles méthodes.

```
LinkedList<String> students = new LinkedList<String>();

students.add( "George" );
students.add( "Maria" );
students.add( "Elvis" );
students.add( "Jean-Jacques" );

students.addFirst("Alice");
System.out.println( students );

students.removeLast();
System.out.println( students );

System.out.println( students.getFirst() );
```

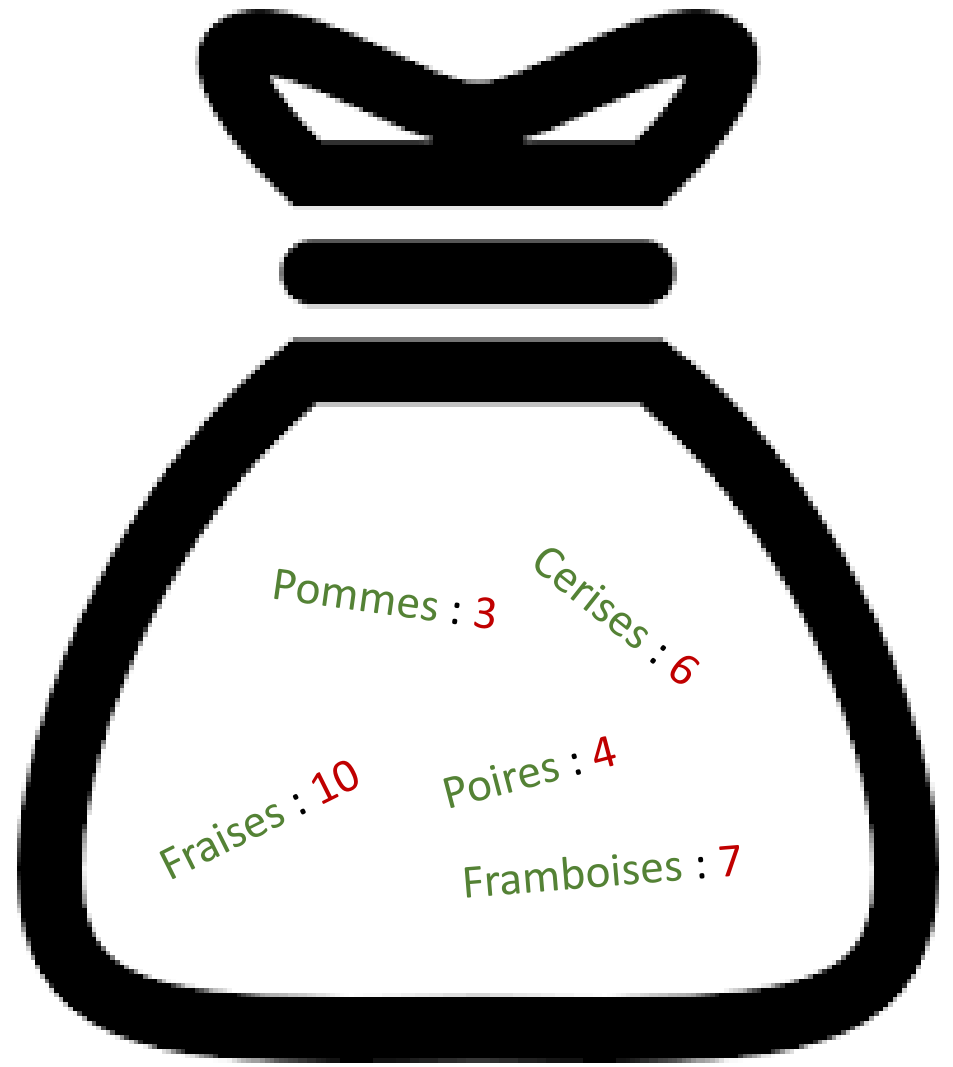
```
[Alice, George, Maria, Elvis, Jean-Jacques]
[Alice, George, Maria, Elvis]
Alice
```


Les HashMap

Le `HashMap` est un autre ensemble adapté à d'autres situation très différentes.

Celui-ci n'est pas ordonné et fonctionne avec des couples **clef** et **valeur**.

On peut voir ça comme un sac dans lequel on stocke pelle-mêle des valeurs auxquelles on attache une étiquette : notre clef.



Les HashMap

Pour définir un **HashMap**, on devra tout d'abord lui spécifier le type de ses clef et le type de ses valeurs.

Dans le cas de notre sac de fruits, il s'agira de clef **String** et de valeurs **int**.

Attention cependant, **HashMap** demande des types sous forme d'objets. Il nous faudra donc remplacer nos types primitifs.

Type primitif	Objet
int	Integer
double	Double
char	Character
bool	Boolean
byte	Byte
long	Long

Les HashMap

Finalement, pour définir les types de notre **HashMap**, nous les engloberons avec des « <> », séparés par une « , ».

```
HashMap< String, Integer > mon_panier = new HashMap< String, Integer >();
```

Les HashMap

HashMap dispose de nombreuses fonctions dont quelques unes ci-dessous :

Méthode	Description
put	Ajoute ou modifie une valeur
get	Lit une valeur
remove	Supprime une valeur et sa clef
clear	Supprime toutes les données
containsValue	Recherche une valeur spécifique

Les HashMap

Un exemple avec différentes manipulations des méthodes précédentes :

```
HashMap< String, Integer > mon_panier = new HashMap< String, Integer >();  
  
mon_panier.put( "Banane", 3 );  
mon_panier.put( "Fraise", 5 );  
  
System.out.println( mon_panier );  
  
mon_panier.remove( "Fraise" );  
  
System.out.println( mon_panier.get("Banane") );  
System.out.println( mon_panier.containsValue(3) );
```

```
{Fraise=5, Banane=3}  
3  
true
```

Les HashSet

Le **HashSet** est lui aussi un ensemble avec un objectif bien particulier : lister des éléments de manière unique et non ordonnés.

On le définit à la manière d'un **ArrayList** et on pourra utiliser la même fonction **add** pour ajouter un élément à l'ensemble.

Pour en retirer un élément, là encore la méthode **remove** fera l'affaire.

```
HashSet< String > users = new HashSet< String >();

users.add("Georges");
users.add("Emilie");

System.out.println( users );

for( int i = 0 ; i < 10 ; i++ ) {
    users.add("Emilie");
}

users.remove("Georges");

System.out.println( users );
```

```
[Georges, Emilie]
[Emilie]
```

Les HashSet

La méthode **contains** nous permettra de savoir si une valeur existe déjà au sein de notre ensemble.

```
users.add("Georges");  
users.add("Emilie");  
System.out.println( users.contains("Georges") );  
System.out.println( users.contains("Jacques") );
```

true
false

```
users.add("Georges");  
users.add("Emilie");  
users.clear();  
  
System.out.println( users );
```

[]

La méthode **clear**, comme d'habitude, permettra de supprimer l'intégralité des valeurs.

Les Itérateurs

```
ArrayList< String > users = new ArrayList< String >();  
users.add("Georges");  
users.add("Stephan");  
users.add("Emilie");  
users.add("Lucie");  
  
Iterator< String > it = users.iterator();
```

Les itérateurs sont des objets permettant de parcourir facilement les ensembles dont ceux que nous avons vu précédemment.

Il s'agit de l'objet **Iterator** importable depuis **java.util**.

Pour obtenir un itérateur de l'un de vos ensembles, il suffira d'utiliser la méthode **iterator**.

Les Itérateurs

Il sera alors possible de lire les différentes valeurs les unes après les autres via un simple usage de la méthode **next** de l'itérateur.

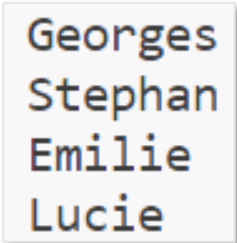
```
// Affichage du premier élément  
System.out.println( it.next() );  
  
// Affichage du deuxième élément  
System.out.println( it.next() );
```

Georges
Stephan

Les Itérateurs

La méthode `hasNext` permet de savoir à tout instant si on a atteint la fin de notre ensemble. Très pratique pour boucler dessus par exemple.

```
// Lecture de la totalité des valeurs
while( it.hasNext() ) {
    System.out.println( it.next() );
}
```



Georges
Stephan
Emilie
Lucie



Pause questions

Chapitre 6

Aller plus loin en POO

Le polymorphisme

Abstraction

Interfaces

Enumérations



Le polymorphisme

Le polymorphisme en POO est la possibilité d'utiliser une même méthode (même prototype) pour différentes classes de la même famille.

Prenons comme exemple cette classe **Animal** qui disposera d'attributs, **name** et **age**, et d'une méthode **crier**.

```
public class Animal {  
  
    protected String name ;  
    protected int age ;  
  
    public Animal( String name, int age ) {  
        this.name = name ;  
        this.age = age ;  
    }  
  
    public void crier() {  
        System.out.println(  
            String.format(  
                "L'animal %s de %d ans crie !",  
                name, age  
            )  
        );  
    }  
}
```

Le polymorphisme

Nous allons créer diverses classes filles à partir de `Animal` comme `Chien`, `Chat` ou encore `Lion`.

Ce qui est intéressant dans chacune de ses 3 classes c'est que chaque `Animal` aura sa propre façon de crier.

Nous devons donc dans chaque cas en redéfinir la méthode `crier`.

```
public class Chat extends Animal {  
  
    public Chat( String name, int age ) {  
        super( name, age );  
    }  
  
    public void crier() {  
        System.out.println(  
            String.format(  
                "Le chat %s de %d ans miaule !",  
                name, age  
            )  
        );  
    }  
}
```

Le polymorphisme

Le polymorphisme est donc, dans notre cas, de pouvoir appeler la même méthode (crier) sur différents objets sans que cela n'ai le même effet.

Notre instance de chat miaule, notre instance de lion rugit, etc... Chacun, exécutant pourtant la même action, produit son propre effet.

```
Chien max = new Chien( "Max", 8 );  
Chat grigri = new Chat( "Grigri", 3 );  
Lion thor = new Lion( "Thor", 7 );
```

```
max.crier();  
grigri.crier();  
thor.crier();
```

```
Le chien Max de 8 ans aboie !  
Le chat Grigri de 3 ans miaule !  
Le lion Thor de 7 ans rugit !
```

Le polymorphisme

Là où c'est encore plus intéressant, c'est quand on souhaite s'adresser à l'ensemble de la famille **Animal**, et donc à n'importe quelle classe fille.

Créons par exemple un **ArrayList** de nos animaux, c'est-à-dire un **ArrayList< Animal >**.

L'avantage de cette définition est qu'il permet d'ajouter n'importe quel classe fille de **Animal** à la liste.

```
ArrayList< Animal > mes_animaux = new ArrayList< Animal >();  
  
mes_animaux.add( new Chien( "Max", 8 ) );  
mes_animaux.add( new Chat( "Grigri", 3 ) );  
mes_animaux.add( new Lion( "Thor", 7 ) );  
  
System.out.println( mes_animaux );
```

```
[Chien@6646153, Chat@21507a04, Lion@143640d5]
```


Le polymorphisme

Et mieux encore, l'effet polymorphique de notre méthode crier nous permet de l'appeler depuis n'importe quel membre de notre liste.

```
for( Animal animal : mes_animaux ) {  
    animal.crier();  
}
```

```
Le chien Max de 8 ans aboie !  
Le chat Grigri de 3 ans miaule !  
Le lion Thor de 7 ans rugit !
```

Pourtant, chaque Animal que comporte cette liste pourra se comporter différemment lors de son cri.

L'abstraction

Dans certains cas, certaines de nos classes mères ne feront que définir les méthodes sans leurs implémentations.

Un exemple très simple avec notre classe **Animal** précédente, il est bien évident qu'on peut pas déterminer le cri d'un **Animal** sans savoir ce qu'il est.

Dans ce cas, la méthode sera définie comme **abstraite**, c'est-à-dire sans implémentation.

```
public abstract void crier();
```

L'abstraction

Bien évidemment, c'est méthode ne sera pas utilisable tant qu'elle n'aura pas été implémentée dans une classe fille.

Une classe contenant une méthode abstraite est elle aussi forcément abstraite. On utilisera là encore le mot clef **abstract**.

De ce fait, une classe abstraite ne pourra pas être instanciée.

```
public abstract class Animal {  
  
    protected String name ;  
    protected int age ;  
  
    public Animal( String name, int age ) {  
        this.name = name ;  
        this.age = age ;  
    }  
  
    public abstract void crier();  
}
```

L'abstraction

Lors d'un héritage, la classe fille est donc elle aussi automatiquement abstraite, puisqu'héritant de la même méthode abstraite.

Pour lever cette abstraction, la classe fille devra donc surcharger la ou les méthodes abstraites de sa classe mère.

```
public class Chien extends Animal {  
  
    public Chien( String name, int age ) {  
        super( name, age );  
    }  
  
    public void crier() {  
        System.out.println(  
            String.format(  
                "Le chien %s de %d ans aboie !",  
                name, age  
            )  
        );  
    }  
}
```

Les interfaces

Les interfaces sont des classes complètement abstraites, ne contenant aucun attributs et uniquement des méthodes abstraites.

Pour définir une interface, on utilisera la même syntaxe que pour une classe mais avec le mot clef `interface`.

```
public interface EtreVivant {  
    public void manger();  
}
```

Les interfaces

Pour en hériter, on n'utilisera pas **extends** mais un mot clef complémentaire **implements**.

Cela signifie qu'il est possible d'hériter d'une classe et d'implémenter une interface en même temps.

```
public class Chien extends Animal implements EtreVivant {
```

Les interfaces

L'avantage des interfaces est qu'il est possible d'en implémenter plusieurs, c'est-à-dire qu'on peut hériter de plusieurs interfaces à la fois.

```
public interface EtreVivant {  
    public void manger();  
}
```

```
public interface EtreMobile {  
    public void se_deplacer();  
}
```

```
public abstract class Animal implements EtreVivant, EtreMobile {
```

Les énumérations

Une énumération est une structure permettant de regrouper un ensemble de valeurs possibles sous un même nom.

Par exemple, tous les états d'un feu tricolore routier. Cela permet de ne pas accepter d'autres valeur et de pouvoir connaître toutes les possibilités.

Pour définir une énumération, on utilisera le mot clef **enum**, suivi de son nom et de toutes ses valeurs entre « {} » séparées par une « , ».

```
enum TrafficLightState {  
    RED,  
    GREEN,  
    ORANGE,  
    BLINK,  
    DISABLED  
}
```


Les énumérations

On pourra alors faire référence à cet ensemble de valeurs autorisées dans des prototypes de méthodes, pour filtrer les entrées.

Un exemple ici avec la méthode `set_state` qui prend en paramètre un `TrafficLightState`, ce qui limite le type de valeurs possibles.

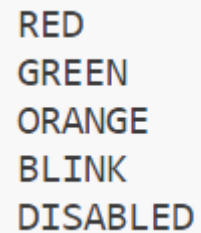
```
public class TrafficLight {  
  
    private TrafficLightState state = TrafficLightState.DISABLED;  
  
    public TrafficLight() {}  
  
    public void set_state( TrafficLightState new_state ) {  
        this.state = new_state ;  
    }  
  
    public void show_status() {  
        System.out.println( this.state );  
    }  
}  
  
public static void main( String[] args ) {  
  
    TrafficLight tl = new TrafficLight();  
    tl.show_status();  
    tl.set_state( TrafficLightState.GREEN );  
    tl.show_status();  
}
```

DISABLED
GREEN

Les énumérations

Enfin, il est possible d'obtenir l'ensemble des valeurs d'une énumération via la méthode **values** que toute énumération possède.

```
for( TrafficLightState tls : TrafficLightState.values() ) {  
    System.out.println( tls );  
}
```



RED
GREEN
ORANGE
BLINK
DISABLED



Pause questions