

Les Formations au dojo 道場

La POO en Java

Durée : 1,5 journée

Sommaire :

Introduction : Rappel de la POO

1 – Le concept de classe

- Les composants d'une classe
- Constructeurs
- Les packages
- Déclarer une instance et l'utiliser

2 – L'héritage

- L'héritage simple
- La classe « Object »
- Classes abstraites
- Interface et implémentation

3 – Les visibilité

4 – Le mot-clef static

5 – Les exceptions

Conclusion

Introduction :

Rappel de la POO



Introduction : Rappel de la POO

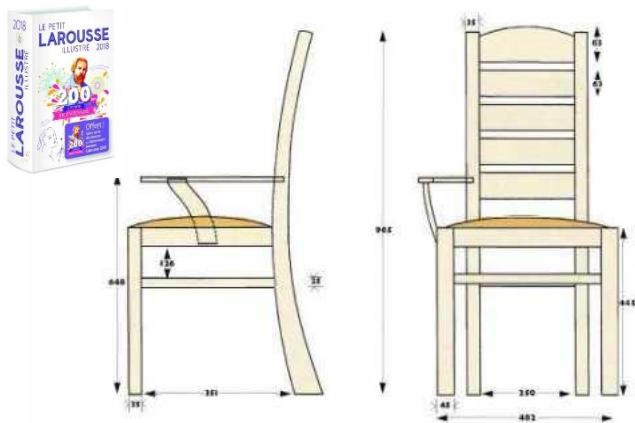
La programmation orientée objet tourne autour de 4 grand concepts :

- L'abstraction, qui permet de décrire des objets de tous les jours de manière simplifiée
- L'encapsulation, qui expose le « quoi » plutôt que le « comment »
- L'héritage, qui permet de mutualiser certains aspects objets à travers une classe « mère »
- Le polymorphisme, qui nous indique qu'un objet peut avoir plusieurs visages

Introduction : Rappel de la POO

Attention ! On ne confond pas :

La définition d'un objet



La classe

L'objet en lui-même



L'objet (ou l'instance de la classe)

Introduction : Rappel de la POO

Une classe se définit par un nombre de variables (attributs) ainsi que des méthodes.

Canard
int age int taille
cancaner() grandir() manger(Graines graines) boire(Eau eau)

Rectangle
int largeur int longueur
calculerSurface()

Ces objets peuvent être dépendant ou indépendant les uns des autres.

Comme dans la vraie vie, un objet ne fait rien de lui-même. Il attend qu'on l'utilise.

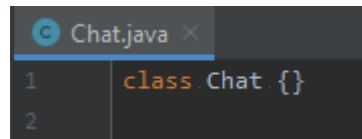


Partie 1 :

Le concept de classe

Le concept de classe en Java

En java, une classe se définit grâce au mot-clef « **class** ».

A screenshot of a code editor window titled 'Chat.java'. The editor shows two lines of code: line 1 contains 'class Chat {' and line 2 contains '}'. The word 'class' is highlighted in orange.

```
1 class Chat {  
2 }
```

Il est techniquement possible d'avoir plusieurs classes au sein d'un même fichier Java, mais pour le moment :

1 classe = 1 fichier



Le concept de classe en Java

Les conventions de nommage de classe en Java :

- Les classes suivent les règles de nommage générales
- Les classes sont des noms (Poire, Voiture, Chat etc...)
- Les classes commencent par des majuscules (par opposition aux variables)
- S'il y a besoin de plusieurs mots, on privilégiera le CamelCase
- Le débat code en français vs code en anglais dépend de votre contexte

Le concept de classe en Java – Vrai ou faux ?



Nom de classe	Compile	Conventionnel
Compte1001Nuits		
Class		
Vouloir		
Etoiles		
\$Dollar\$		
€Euro€		
despote		
Martin-Pecheur		
MartinPecheur		

Le concept de classe en Java – Vrai ou faux ?



Nom de classe	Compile	Conventionnel
Compte1001Nuits	✓	✓
Class	✓	✓
Vouloir	✓	✗
Etoiles	✓	✓
\$Dollar\$	✓	✗
€Euro€	✗	✗
despote	✓	✗
Martin-Pecheur	✗	✗
MartinPecheur	✓	✓





Le concept de classe en Java

L'utilisation de cette classe se fait à travers des instances de cette dernière.

(classe ≠ instance)

La création d'une instance se fait grâce au mot-clef « **new** » suivi d'un **constructeur**.

Plus d'information sur le constructeur dans les prochains slides

Le résultat de cette création est ensuite stockée dans une variable dont le type porte le nom de la classe.

```
public static void main(String[] args) {  
    Chat gribouille = new Chat();  
    Chat felix = new Chat();  
    Chat nougatine = new Chat();  
}
```

On peut créer autant d'instances de notre objet que la mémoire de notre JVM nous le permet.

Le concept de classe en Java

Pour l'instant, la classe « Chat » n'a pas de comportement. Ajoutons-lui une méthode :

```
class Chat {  
  
    void miauler(){  
        System.out.println("Miaou !");  
    }  
  
}
```

Notre chat est maintenant capable de miauler !

Le concept de classe en Java

On ne peut accéder à cette méthode qu'à travers une instance de type chat :

- C'est un chat (une instance) qui miaule
- Il n'y a pas de miaulement sans chat

Pour la déclencher, on utilise la notation suivante :

<variable>. <méthode>(<arguments>);

```
gribouille.miauler();  
felix.miauler();  
felix.miauler();  
nougatine.miauler();
```



```
Miaou !  
Miaou !  
Miaou !  
Miaou !
```

Le concept de classe en Java

Pour le moment, tous les objets de la classe « Chat » sont les mêmes. Ils n'ont que du comportement et n'ont pas d'attributs particuliers.

Ajoutons à la classe « Chat » un attribut « **nom** » et une attribut « **couleur** ».

```
String couleur;  
String nom;  
  
void miauler(){  
    System.out.println(nom+"."+couleur+" dit : Miaou!");  
}
```

Chat
String couleur String nom
miauler()

Le concept de classe en Java

Pour l'instant, les attributs ne sont pas initialisés :

```
gribouille.miauler();    null null dit : Miaou !  
felix.miauler();        null null dit : Miaou !  
nougatine.miauler();    null null dit : Miaou !
```

Il y a quatre moyens de les valoriser :

- Donner des valeurs par défaut
- Valoriser directement les attributs
- Passer par des méthodes
- Utiliser le constructeur pour construire l'objet avec les bonnes valeurs

```
class Chat {  
    ... String nom = "Chat";  
    ... String couleur = "Gris";  
  
    ... void miauler() { ... }  
}
```

```
gribouille.nom = "Gribouille";  
gribouille.couleur = "Tigré blanc et gris";
```

```
felix.setNom("Félix");  
felix.setCouleur("Noir et blanc");
```

```
Chat nougatine = new Chat("Nougatine", "Roux");
```


Le concept de classe en Java

La **valorisation directe** des attributs brise le phénomène d'encapsulation : On agit directement sur la structure interne de l'objet sans passer par les méthodes qu'il expose.



La **valorisation par méthodes** (ou par « setter ») ne brise pas *explicitement* le phénomène d'encapsulation, mais donne tout de même beaucoup d'informations sur la structure interne de l'objet.



De plus, ces deux première méthodes permettent de créer un objet dont la structure n'est pas intègre (ici, on peut très bien créer un chat sans couleur).

Le **constructeur** ne brise pas le phénomène d'encapsulation et encourage le développeur à créer des objets **intègres**. **C'est la solution que l'on va préférer.**



Nb : C'est une méthode qui peut vite devenir complexe en cas de grande combinatoire, mais il existe des astuces pour éviter cette problématique



Le concept de classe en Java : Les constructeurs

Un **constructeur** est une méthode de l'objet dont le nom est le même que celui de sa classe :

```
Chat() {}
```

Il peut coexister plusieurs constructeurs au sein d'une même classe.

Les constructeurs peuvent avoir des paramètres. C'est grâce à ceux-là que l'on va pouvoir valoriser les attributs de la classe.

```
Chat(String param1, String param2) {  
    ... nom = param1;  
    ... couleur = param2;  
}
```



Le concept de classe en Java : Les constructeurs

Dans ce dernier exemple, `param1` et `param2` sont des variables de méthode : Ils n'existent pas en dehors de cette méthode

Les attributs `nom` et `couleur` sont des variables inhérentes à la classe. On peut y accéder depuis toutes les méthodes de cette classe

Si une variable de classe possède le même nom qu'une variable de méthode, on accèdera à la variable de classe grâce au mot-clef « `this` »

```
class Chat {  
  
    ...String nom;  
    ...String couleur;  
  
    ...Chat() {}  
  
    ...Chat(String nom, String couleur) {  
        ...    this.nom = nom;  
        ...    this.couleur = couleur;  
    ...}  
  
    ...void miauler(){...}  
  
}
```



Le concept de classe en Java : Les constructeurs

Dans le cas où il n'y a pas de constructeur déclaré dans un objet Java. Le compilateur ajoute un « **constructeur par défaut** ».

Ce constructeur par défaut est un constructeur qui ne prend pas de paramètres et qui ne contient pas d'instructions.

Si l'objet possède un constructeur, le compilateur ne mettra pas de constructeur par défaut !

Le concept de classe en Java : Les constructeurs



Exercice 1 : Développez une classe « Enfant » qui possède un nom et un prénom

Exercice 2 : Ajoutez une méthode « presentation() » qui affiche « Bonjour, je m'appelle <prenom> <nom> »

//Créez 2 enfants Charlotte et Olivier et faites les se présenter

Exercice 3 : Ajoutez-lui un attribut « âge » qui :

- Commence à 0
- Doit s'incrémenter de 1 à travers la méthode « feterAnniversaire() »
- Ne doit pas dépasser 17 ans

Exercice 4 : Ajoutez l'âge à la méthode de présentation

//Faites vieillir Charlotte et Olivier de quelques années et faites les se présenter

Le concept de classe en Java : Les constructeurs



Exercice 5 : Ajoutez une méthode « joueAuBallon » qui prend en paramètre un autre enfant et qui affiche :

- « <prenom1> joue au ballon avec <prenom2> »

//Faire jouer Charlotte au ballon avec Olivier

Exercice 6 : Ajoutez à la classe « Enfant » un attribut « amoureux »

- Quel type allez-vous utiliser pour un tel attribut?
- Ecrire une méthode « tomberAmoureux » qui va venir valoriser cet attribut
- Ecrire une méthode « estAmoureux » qui renvoie un booléen selon si l'enfant a (ou non) un amoureux

//Faire que Olivier tombe amoureux de Charlotte (et vice-versa)

Le concept de classe en Java : Les constructeurs



Exercice 7 : Dans le code suivant, repérez les constructeurs.

- Les constructeurs sont-ils tous utiles?
- Pourquoi?

```
class Chat {  
    String nom;  
    String couleur;  
  
    Chat() {}  
  
    Chat(String nom, String couleur) {  
        this.nom = nom;  
        this.couleur = couleur;  
    }  
  
    void miauler() {...}  
}
```



Le concept de classe en Java : Les packages

Java possède un système de rangement des classes, appelé « **package** ». Ainsi, une classe est identifiable par :

- Son nom
- Son package

Par exemple la classe **String** native de Java provient du package « **java.lang** ». Son nom complet (ou « **fully qualified name** ») est donc « **java.lang.String** ».

La déclaration du nom du package se fait une fois par fichier et sera pris en compte pour toutes les classes présentes dans ce fichier.

Le mot-clef à utiliser est « **package** » suivi du nom du package :

```
package com.bnpparibas.sit.architecture.demoepita;
```


Le concept de classe en Java : Les packages

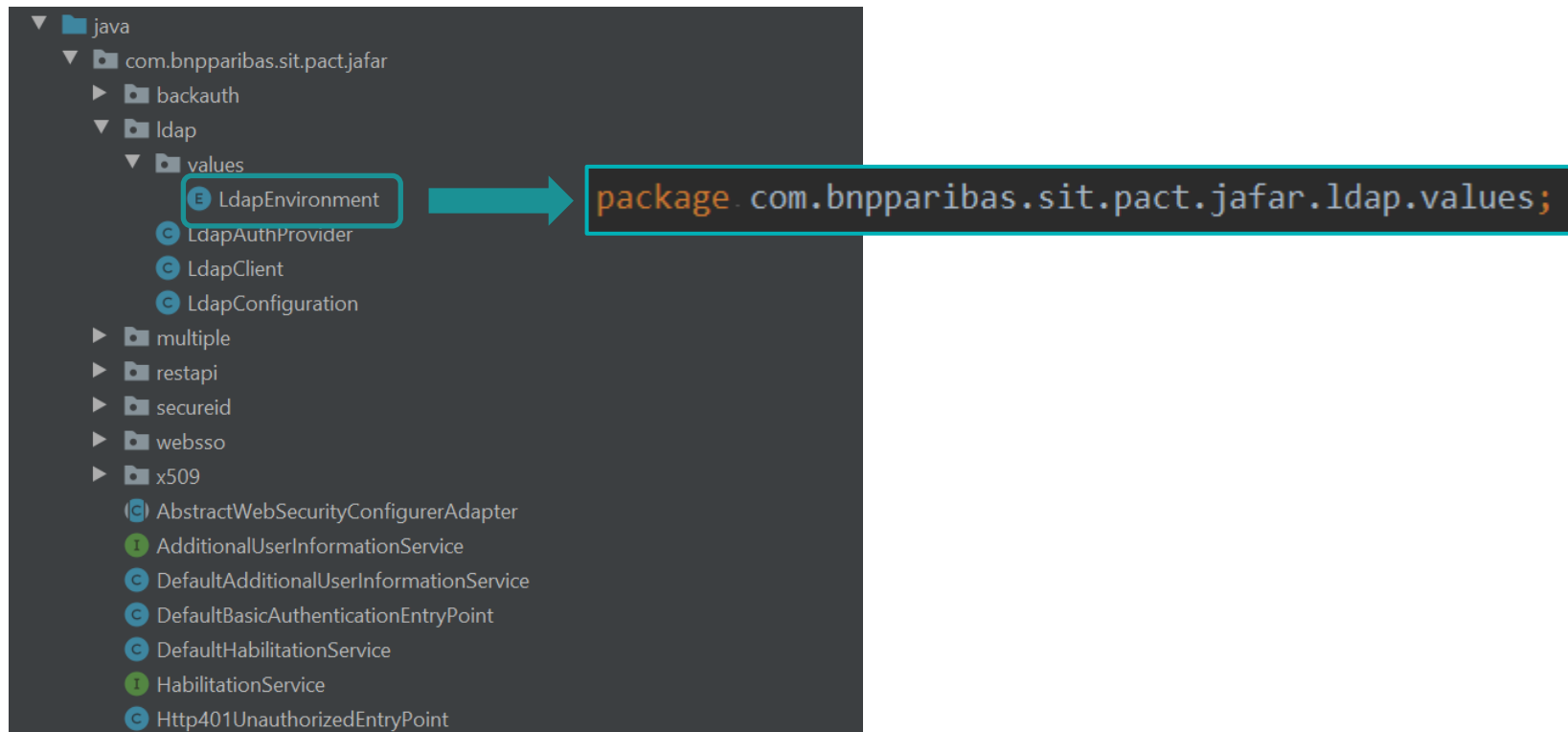
Utiliser les packages de manière intelligente permet d'avoir un classement intuitif des classes. Elle nous permet de savoir d'où elles viennent et ce qu'elles font.

Par exemple chez SIT :



Le concept de classe en Java : Les packages

Autre exemple :



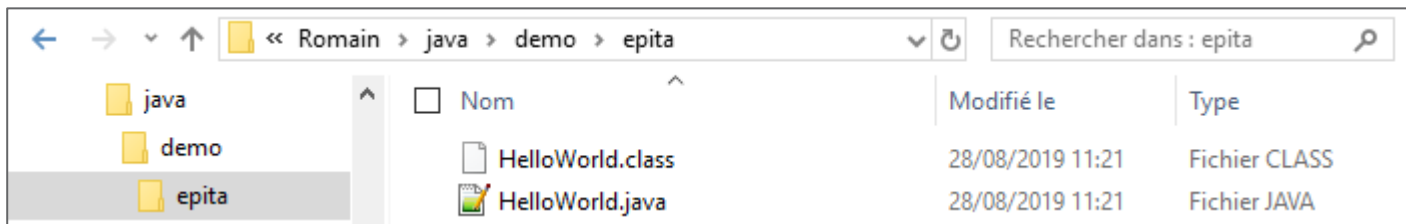
Le concept de classe en Java : Les packages

Le package désigne l'arborescence de fichiers pour pouvoir retrouver la classe.

Par exemple :

```
package demo.epita;  
  
class HelloWorld{  
    ... public static void main(String[] args) {...}  
}
```

L'interpréteur Java s'attend à trouver la classe dans le sous-dossier suivant (en partant du dossier racine « java ») :



A vertical line on the left side of the slide, featuring a series of circles. The second circle from the top is filled with a teal color, while the others are white with black outlines.

Le concept de classe en Java : Les packages

Attention, il ne sera plus possible d'exécuter la classe avec la commande :

```
b65046@PARW00243450 MINGW64 /c/Romain/java/demo/epita
$ java HelloWorld
Erreur : impossible de trouver ou charger la classe principale HelloWorld
```

Pour exécuter la classe présente dans un package, il faut se placer dans le dossier racine et d'appeler la classe par son nom complet :

```
b65046@PARW00243450 MINGW64 /c/Romain/java
$ java demo.epita.HelloWorld
Hello World
```



Le concept de classe en Java : Les packages

S'il n'y a pas de package d'indiqué, Java utilise le package par défaut (le répertoire courant)

Ne pas indiquer de package est une mauvaise pratique qui induira des problèmes par la suite :

- Problématiques de visibilité
- Problématiques d'import



Le concept de classe en Java : Les packages

Les règles de nommage pour les packages :

- Elles suivent les règles de nommages Java standard
- Les packages sont séparés de leur sous-package par un point

La convention de nommage est simple :

- Tous les noms de packages doivent être en caractère alphabétique minuscule

Le concept de classe en Java : Les packages

Par défaut, vous pouvez utiliser toutes les classes d'un même package (ou d'un de ses sous-packages) sans avoir besoin d'indiquer quoi que ce soit à Java.

Si les classes que vous souhaitez utiliser se trouvent dans d'autres packages, il faudra :

- Soit les déclarer grâce au mot-clef « **import** » (après la déclaration du package, mais avant la déclaration de classe).
- Soit les utiliser grâce à leur « **full qualified name** »

```
import demo.autrePackage.Enfant;

class HelloWorld{
    ... public static void main(String[] args){
    ...     Enfant charlotte = new Enfant();
    ... }
}
```

```
class HelloWorld{
    ... public static void main(String[] args){
    ...     demo.autrePackage.Enfant charlotte
    ...     = new demo.autrePackage.Enfant();
    ... }
}
```

Le concept de classe en Java : Les packages



Exercice 1 : Reprendre les sources de l'exercice précédent et les placer dans un même package. N'oubliez pas d'adapter l'arborescence de vos fichiers en conséquence

Exercice 2 : Exécuter le programme avec le nom complet de la classe

Astuce : Pour compiler l'intégralité des classes Java dans vos sous dossiers, vous pouvez utiliser la commande suivante :

```
$ find . -name "*.java" | xargs javac
```


Le concept de classe en Java : Les références

En Java, les objets (non-primitifs) sont des références vers des objets en mémoire.

```
Chat chat = new Chat();  
System.out.println(chat);
```



demo.epita.Chat@311d617d

@311d617d

Variable 'chat'



@311d617d

⋮

@01235457



Mémoire

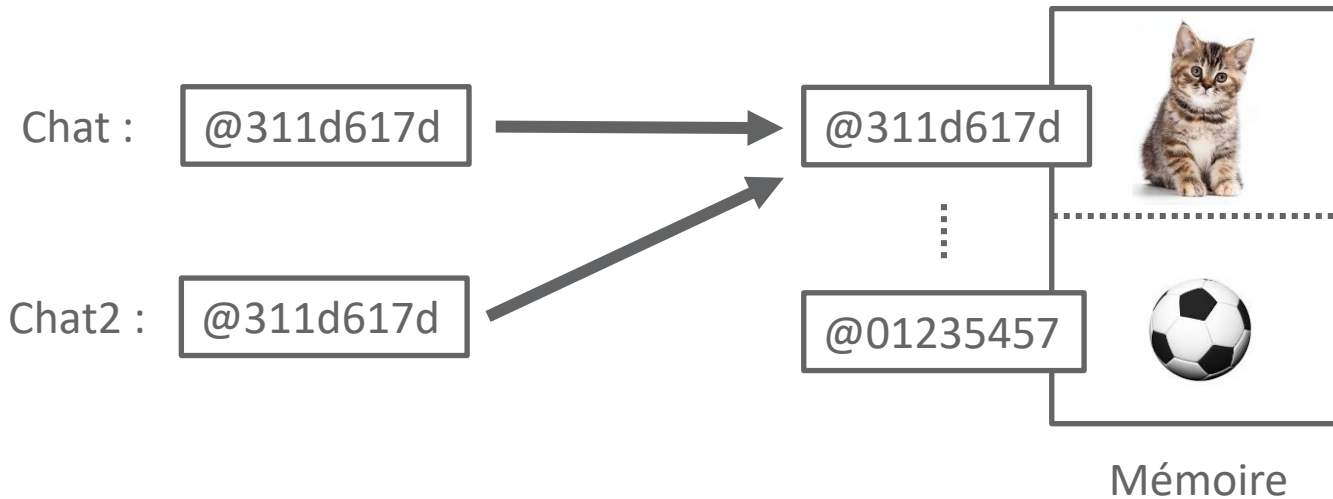
Le concept de classe en Java : Les références

Conséquence direct de cette particularité : deux variables peuvent pointer sur le même objet en mémoire :

```
Chat chat = new Chat();  
Chat chat2 = chat;  
System.out.println(chat);  
System.out.println(chat2);
```



demo.epita.Chat@311d617d
demo.epita.Chat@311d617d





Le concept de classe en Java : Les références

Nouvelle conséquence : Si je modifie l'objet pointé, il sera modifié en mémoire, et donc pour toutes les variables qui pointent dessus.

Il sera donc possible de modifier l'objet pointé par « chat2 » en appelant des méthodes sur « chat »

```
Chat chat = new Chat( name: "Gribouille");  
Chat chat2 = chat;  
chat.setName("Nougatine");  
System.out.println(chat.getName());  
System.out.println(chat2.getName());
```

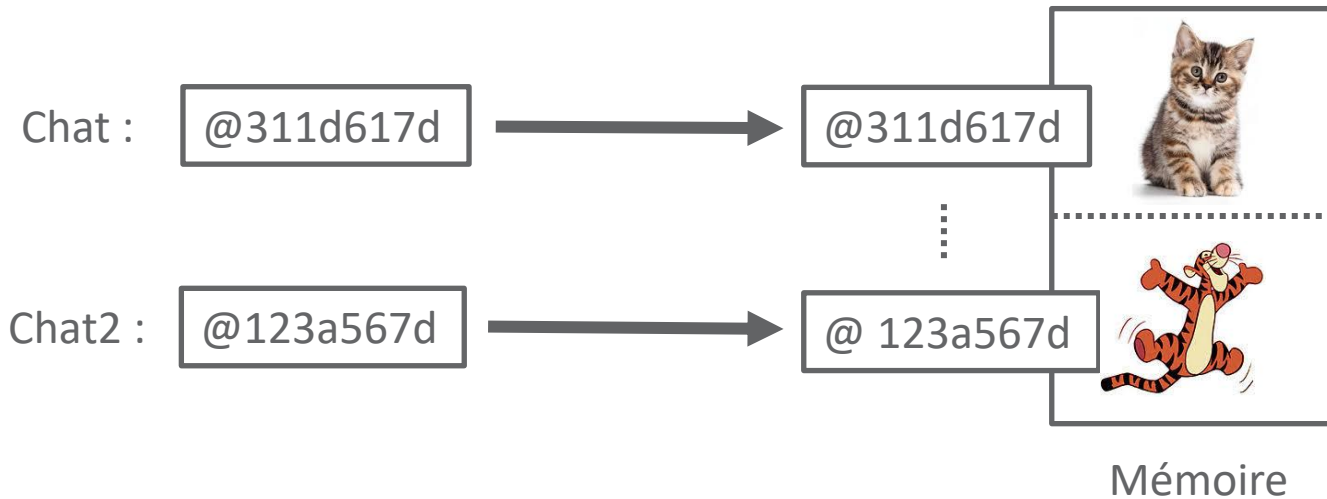
Nougatine

Nougatine

Le concept de classe en Java : Les références

Il se peut que lors d'une manipulation, un objet en mémoire ne soit plus pointé par aucune variable.

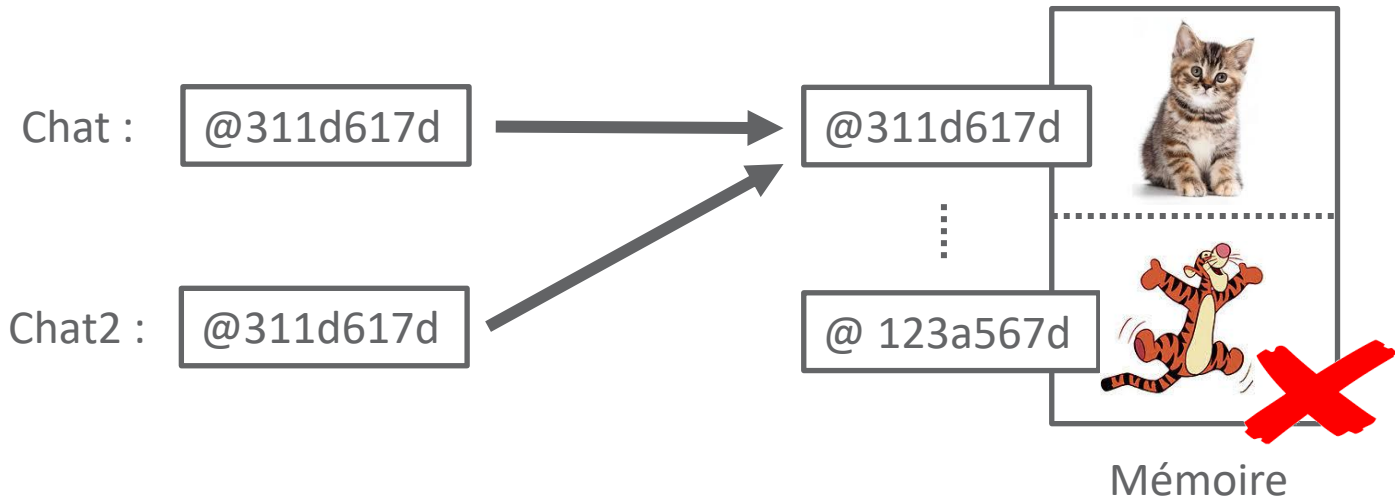
```
Chat chat = new Chat( name: "Gribouille");  
Chat chat2 = new Chat( name: "Tigrou");  
chat2 = chat;
```



Le concept de classe en Java : Les références

Il se peut que lors d'une manipulation, un objet en mémoire ne soit plus pointé par aucune variable.

```
Chat chat = new Chat( name: "Gribouille");  
Chat chat2 = new Chat( name: "Tigrou");  
chat2 = chat;
```



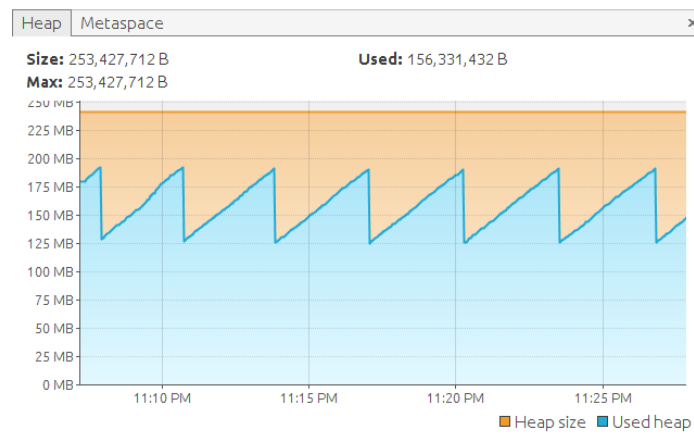
Le concept de classe en Java : Les références

Cet objet ne sert alors plus à rien (si ce n'est occuper de la place dans la mémoire) !

Il est alors repéré et détruit automatiquement par le « **Garbage Collector** »

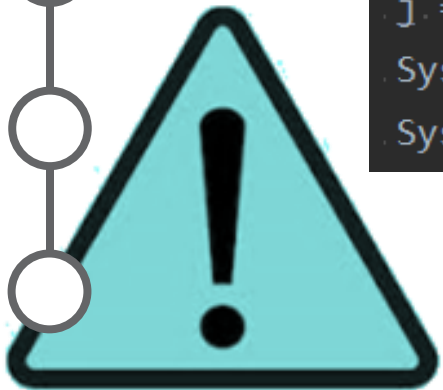
Pour cette raison, il n'y a pas de « destructeur » en Java.

On peut appeler manuellement le Garbage Collector, mais en général, c'est une mauvaise idée.

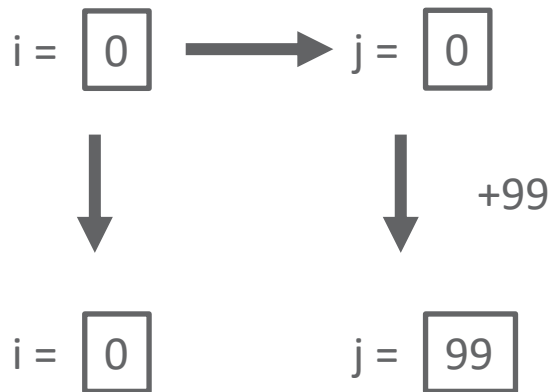


Le concept de classe en Java : Les références

Attention ! Pour le cas des types primitifs, les variables ne contiennent pas d'adresse mémoire mais bien leur valeur réelle !



```
int i = 0;
int j = i;
j = j+99;
System.out.println(i);
System.out.println(j);
```





Le concept de classe en Java

Exercice 1 : Qu'affichent ces codes ?

```
Chat chat = new Chat("Gribouille");  
Chat chat2 = new Chat("Napoleon");  
chat = chat2;  
chat2.setNom("Felix");  
System.out.println(chat.getNom());  
System.out.println(chat2.getNom());
```

```
int i = 1;  
int j = 2;  
i = j;  
j=42;  
System.out.println(i);  
System.out.println(j);
```

```
double[] tab1 = {1.0};  
double[] tab2 = {2.0};  
tab1 = tab2;  
tab2[0] = 42.0;  
System.out.println(tab1[0]);  
System.out.println(tab2[0]);
```

```
String s1 = "1";  
String s2 = "2";  
s1 = s2;  
s2="42";  
System.out.println(s1);  
System.out.println(s2);
```




Le concept de classe en Java

Exercice 1 :

Felix / Felix

2 / 42

```
Chat chat = new Chat("Gribouille");  
Chat chat2 = new Chat("Napoleon");  
chat = chat2;  
chat2.setNom("Felix");  
System.out.println(chat.getNom());  
System.out.println(chat2.getNom());
```

```
int i = 1;  
int j = 2;  
i = j;  
j=42;  
System.out.println(i);  
System.out.println(j);
```

```
double[] tab1 = {1.0};  
double[] tab2 = {2.0};  
tab1 = tab2;  
tab2[0] = 42.0;  
System.out.println(tab1[0]);  
System.out.println(tab2[0]);
```

```
String s1 = "1";  
String s2 = "2";  
s1 = s2;  
s2="42";  
System.out.println(s1);  
System.out.println(s2);
```

42.0 / 42.0

2 / 42



Le concept de classe en Java

Exercice 2 : Qu'affichent le code suivant ?

```
public static void main(String[] args) {  
    Chat gribouille = new Chat("Gribouille");  
    Chat gribouille2 = clonerChat(gribouille);  
    gribouille2.setNom("CloneDeGribouille");  
    System.out.println(gribouille.getNom());  
}  
  
private static Chat clonerChat(Chat gribouille) {  
    Chat clone = new Chat("griffu");  
    clone = gribouille;  
    clone.miauler();  
    return clone;  
}
```



Le concept de classe en Java

Exercice 2 : Qu'affichent le code suivant ?

```
public static void main(String[] args) {  
    Chat gribouille = new Chat("Gribouille");  
    Chat gribouille2 = clonerChat(gribouille);  
    gribouille2.setNom("CloneDeGribouille");  
    System.out.println(gribouille.getNom());  
}  
  
private static Chat clonerChat(Chat gribouille) {  
    Chat clone = new Chat("griffu");  
    clone = gribouille;  
    clone.miauler();  
    return clone;  
}
```

Miaou
CloneDeGribouille



Le concept de classe en Java

Exercice 3 : Quel est le soucis avec cette classe? Une idée pour le résoudre?

```
public class Parent {  
  
    String nom;  
    String prenom;  
    Enfant[] enfants;  
  
    public Parent(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public void avoirEnfant(){  
        /*code compliqué permettant  
        d'ajouter un enfant au tableau*/  
    }  
}
```

```
    public String getNom() {  
        return nom;  
    }  
  
    public String getPrenom() {  
        return prenom;  
    }  
  
    public Enfant[] getEnfants() {  
        return enfants;  
    }  
}
```



Le concept de classe en Java

Exercice 3 : Quel est le soucis avec cette classe? Une idée pour le résoudre?

```
public class Parent {  
  
    String nom;  
    String prenom;  
    Enfant[] enfants;  
  
    public Parent(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public void avoirEnfant(){  
        /*code compliqué permettant  
        d'ajouter un enfant au tableau*/  
    }  
}
```

```
    public String getNom() {  
        return nom;  
    }  
  
    public String getPrenom() {  
        return prenom;  
    }  
  
    public Enfant[] getEnfants() {  
        return enfants;  
    }  
}
```

On renvoie l'adresse mémoire du tableau (et donc on autorise l'écriture) via une méthode de lecture.

Partie 2 : L'héritage





L'héritage

L'héritage en Java se fait grâce au mot-clef

« **extends** »

Rappel : Une classe fille hérite de sa classe mère :

- Ses méthodes
- Ses attributs

```
class Chat {  
    void miauler(){  
        System.out.println("Miaou !");  
    }  
}
```

```
class SuperChat extends Chat{  
    void voler(){  
        System.out.println("Je vole !");  
    }  
}
```

```
public static void main(String[] args) {  
    SuperChat superChat = new SuperChat();  
    superChat.miauler();  
    superChat.voler();  
}
```

L'héritage

Si dans le contexte de la classe fille, la méthode de la classe mère ne nous convient pas, on peut modifier son comportement.

C'est la surcharge :

```
class Chat {  
    void miauler(){  
        System.out.println("Miaou !");  
    }  
}
```

```
class SuperChat extends Chat{  
    void voler(){  
        System.out.println("Je vole !");  
    }  
  
    void miauler(){  
        System.out.println("Super Miaou !");  
    }  
}
```

Ici, un SuperChat ne fait plus **Miaou**, mais **Super Miaou** !

L'héritage

Par convention, on ajoute l'annotation « **@Override** » au dessus d'une méthode surchargée.

Cette annotation n'a qu'un but documentaire. Elle n'est pas obligatoire.

```
class SuperChat extends Chat{  
    ...  
    void voler() {  
        System.out.println("Je vole !");  
    }  
    @Override  
    void miauler() {  
        System.out.println("Super Miaou !");  
    }  
}
```



L'héritage

Il arrive parfois que l'on veuille tout de même accéder à la méthode ou à l'attribut de la classe mère.

Par exemple, si une méthode de la classe mère est une composante de comportement de la classe fille.

On utilisera alors
le mot-clef « **super** »

```
@Override  
void miauler(){  
    ... System.out.print("Super.");  
    ... super.miauler();  
}
```

L'héritage

Lorsque vous avez défini un constructeur dans la classe mère, le compilateur vous **forcera** à écrire au moins un constructeur dans la classe fille.

Chacun de ses constructeurs doit faire appel à un constructeur de la classe mère grâce à la méthode « **super(<arguments>)** »

L'appel à un constructeur parent **doit-être la première instruction** !

```
public class Enfant {  
    public Enfant(String nom,  
                  String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```



```
class EnfantMignon extends Enfant{  
    public EnfantMignon(String nom,  
                        String prenom) {  
        super(nom, prenom);  
    }  
}
```



L'héritage

Exercice 1 : Développer une classe « Véhicule », qui possède en attribut un entier « nombre de roues », et un constructeur associé.

Exercice 2 : Développer une méthode *direBonjour()* qui dit fait dire à tout véhicule « Bonjour, je suis un véhicule à <n> roues ».

Exercice 3 : Développer une classe « Vélo » qui est un type de véhicule. Le vélo possède un constructeur sans paramètres qui initialise le nombre de roues à 2.

Exercice 4 : Créer une vélo et le faire se présenter.



L'héritage

Exercice 5 : Sur le même concept que la classe vélo, créer la classe « Train » qui possède 100 roues. Le train se présente comme tout autre véhicule mais ajoute « TchooTchoo ! » après sa présentation.

Exercice 6 : Créer un train et le faire se présenter.

A vertical line on the left side of the slide, composed of a series of circles connected by a line. The third circle from the top is filled with orange, while the others are white with gray outlines.

L'héritage : La classe Object

En Java, tout objet hérite par défaut de la classe « **Object** »

Ce qui implique que toute class en Java possède des méthodes par défaut.

Certaines de ces méthodes peuvent être utilisées telles-
quelles et n'ont pas vocation à être surchargées :

- **void wait()** et **notifyAll()** qui servent à faire du multithreading
- **getClass()** qui sert à récupérer la classe de l'objet



L'héritage : La classe Object

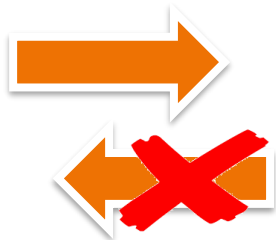
D'autres méthodes par défaut ont vocation à être surchargées :

- **String toString()** : Représentation de l'objet en chaîne de caractères. Par défaut, elle renvoie l'adresse mémoire, mais on peut lui faire afficher ce que l'on veut !
- **int hashCode()** : Fait une « empreinte » de l'objet et le stocke sous forme de nombre
- **boolean equals(Object o)** : Définit l'égalité entre l'objet courant et l'objet passé en paramètre

L'héritage : La classe Object

Contrat entre equals et hashCode :

- Si deux objets ont la même empreinte (hashCode) alors, ils doivent être égaux
- Deux objets peuvent être sémantiquement égaux, mais ne pas avoir la même empreinte



A vertical line on the left side of the slide with a series of circles. The third circle from the top is filled with orange, while the others are white with black outlines.

L'héritage : La classe Object

Focus sur l'égalité :

```
int a = 1;
int b = 1;
System.out.println(a == b);
```

true

Cela fonctionne car le contenu des deux variables est identique. Mais...

```
Chat chat1 = new Chat( nom: "Gribouille");
Chat chat2 = new Chat( nom: "Gribouille");
System.out.println(chat1 == chat2);
```

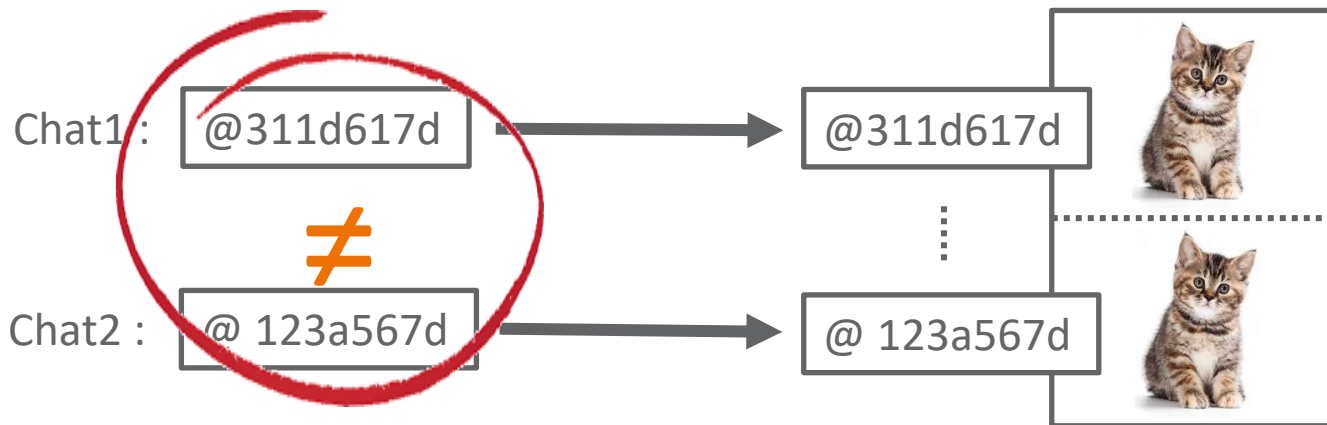
false

Pourquoi ?

L'héritage : La classe Object

...car les variables contiennent des adresse mémoires !

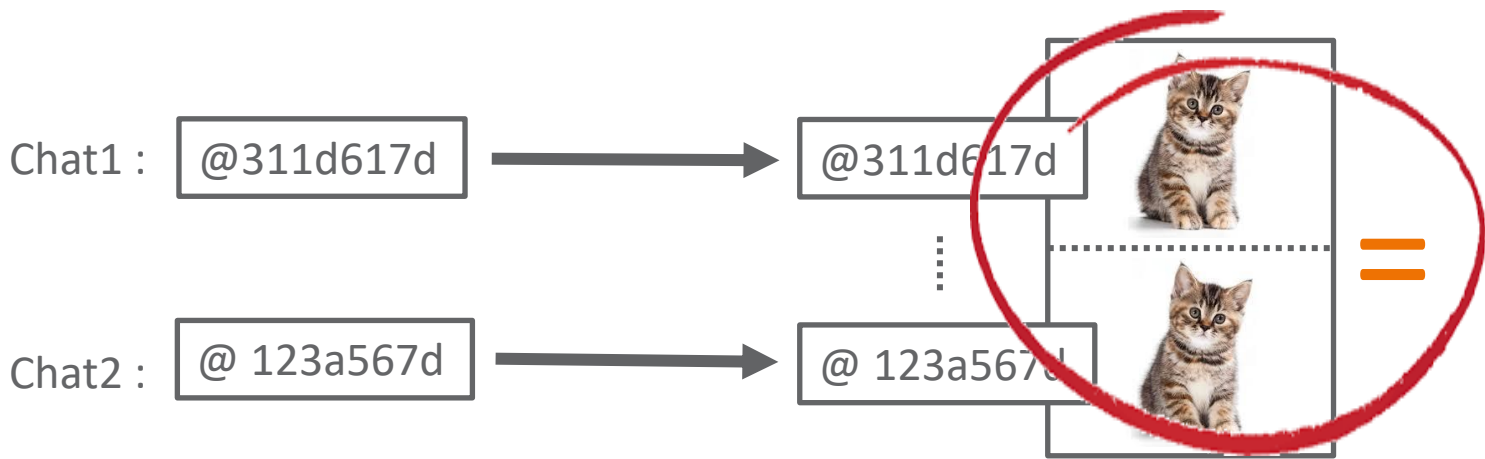
Même si ces deux objets sont sémantiquement identiques, leurs adresses mémoires sont différentes, ce sont deux objets différents !



L'héritage : La classe Object

La méthode « **equals** » sert à effectuer une comparaison sémantique :

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Chat chat = (Chat) o;
    return Objects.equals(this.nom, chat.nom);
}
```





L'héritage : La classe Object

En général, on écrit jamais soi-même une méthode « `equals` » ou « `hashCode` ».

Les outils de développement le font pour nous !

Il existe même des bibliothèques (e.g : Immutable, Lombok...) qui permettent d'ajouter ces méthodes dynamiquement à la compilation.

Certains langages (e.g : Kotlin) vont même jusqu'à intégrer ces fonctionnalités dans le langage.

L'héritage



Exercice 1 : Reprendre le code de l'exercice précédent, mais remplacer la méthode *direBonjour()* par une autre méthode de sorte que *System.out.println(vehicule);* affiche la phrase de présentation.

Exercice 2 : Ce code est-il cohérent? Si non, comment faire pour l'améliorer?

```
String string1 = "chat";
String string2 = "chat";

if(string1 != string2){
    System.out.println("Pas égaux");
}else{
    System.out.println("Egaux");
}
```

L'héritage



Exercice 1 : Reprendre le code de l'exercice précédent, mais remplacer la méthode *direBonjour()* par une autre méthode de sorte que *System.out.println(vehicule);* affiche la phrase de présentation.

Exercice 2 : Ce code est-il cohérent? Si non, comment faire pour l'améliorer?

```
String string1 = "chat";
String string2 = "chat";

if(string1 != string2){
    System.out.println("Pas égaux");
}else{
    System.out.println("Egaux");
}
```

Il faut utiliser la méthode *.equals()*

Mais depuis java 8, cela fonctionne tout de même !

L'héritage : Les interfaces

En POO, il existe une notion de contrat. En Java, les **interfaces** sont une manière de matérialiser ce contrat.

```
interface Caressable {  
    void caresser();  
}
```

Le mot-clef pour respecter ce contrat est « **implements** »

```
class Chien implements Caressable {  
    @Override  
    public void caresser() {  
        System.out.println("**Remue la queue**");  
    }  
}
```

```
class Chat implements Caressable {  
    @Override  
    public void caresser() {  
        System.out.println("RrrrrRrrrrRrrrr");  
    }  
}
```

A vertical line on the left side of the slide, featuring a series of circles. The third circle from the top is filled with orange, while the others are white with black outlines.

L'héritage : Les interfaces

Comme pour les surcharges, on utilise l'annotation « **@Override** » pour signifier que l'on implémente une méthode contractualisée dans une interface.

Si une classe annonce le respect d'un contrat, le compilateur veillera à ce que l'implémentation a bien été réalisée.

On ne peut pas instancier d'interface !



L'héritage : Les interfaces

Rappel : En Java, il est possible de respecter plusieurs contrats, mais il n'est pas possible d'étendre plusieurs classes.

Les interfaces à respecter sont alors listées et séparées par des virgules :

```
class Chat extends Animal implements Caressable, Domestiquable, Joueur { ... }
```



L'héritage

Exercice 1 : Créer une classe Maison avec un attribut température

Exercice 2 : Créer une interface AvecChauffage et une interface AvecClimatisation ayant respectivement un contrat de méthode *chauffer()* et *refroidir()*

Exercice 3 : Faire en sorte que la maison implémente ces deux interfaces. La méthode *chauffer()* fait monter la température d'un degré, et *refroidir()* la fait baisser d'un.

Exercice 4 : Créer une classe Datacenter qui possède lui aussi une température. Un datacenter n'a qu'une climatisation. La climatisation permet de réguler la température à 20°.



L'héritage : Les classes abstraites

Les classes abstraites sont un autre moyen de créer des contrats en Java.

Elle rassemblent à la fois les particularités des interfaces et des classes parents :

- Il faut les étendre (et non les implémenter)
- On ne peut pas instancier de classe abstraite
- Elle peuvent contenir des méthodes abstraites (qui agissent comme une interface)
- Elles peuvent contenir des méthodes « concrètes »
- Elles peuvent contenir des attributs

L'héritage : Les classes abstraites

Le mot-clef à utiliser est « **abstract** » :

```
abstract class Mammifere {  
  
    final int nombreYeux = 2;  
  
    void respirer(){  
        System.out.println("J'inspire de l'air");  
        System.out.println("J'expire de l'air");  
    }  
  
    abstract void reproduire();  
  
}
```



L'héritage

*Rappel : Lorsque l'on est en attente d'une variable d'un certain type, on peut la substituer par toute implémentation ou classe fille de cette dernière. C'est le **polymorphisme** !*

L'héritage

Exercice :

```
class Chat {  
    ... void mange(AlimentPourChat aliment){/*...*/}  
}
```



Si AlimentPourChat est...	Alors on peut passer en paramètres...
Classe Concrète	
Classe Abstraite	
Interface	

L'héritage

Exercice :

```
class Chat {  
    ... void mange(AlimentPourChat aliment){/*...*/}  
}
```



Si AlimentPourChat est...	Alors on peut passer en paramètres...
Classe Concrète	<ul style="list-style-type: none">- Instance de la classe- Instance de classe étendant la classe concrète
Classe Abstraite	<ul style="list-style-type: none">- Instance de classe étendant la classe abstraite
Interface	<ul style="list-style-type: none">- Instance classe implémentant l'interface- Instance de classe étendant une implémentation de l'interface

Partie 3 :

Les visibilités





Les visibilité

Pour le moment, votre encapsulation s'arrête à la création de méthodes, mais rien n'empêche un développeur d'accéder directement aux attributs d'une classe.

```
.Enfant enfant = new Enfant( nom: "DUPONT", prenom: "Olivier");  
.enfant.age = 20;
```

Ici, on agit directement sur l'attribut âge sans passer par la méthode correspondante (qui vérifie que l'âge est inférieur à 18 ans)

A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with a purple color, while the others are white with black outlines.

Les visibilités

Les visibilités sont un moyen d'indiquer au compilateur qui (quels objets) a le droit d'accéder aux propriétés - ou aux méthodes - de la classe.

En Java, il existe 4 visibilités :

- public
- protected
- « default » ou package-private
- private

A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with a purple color, while the others are white with black outlines.

Les visibilités

Public : Tout le monde a le droit d'accéder à l'attribut ou à la méthode. Généralement utilisée pour définir des API publique, ce que l'on a envie d'exposer au monde extérieur.

Private : Les accès ne peuvent se faire que par la classe elle-même. Généralement utilisée pour le fonctionnement interne de la classe.



Les visibilité

Dans cet exemple, le chef cuisinier expose des méthodes publiques :

- On peut lui demander de cuisiner des pâtes
- On peut lui demander de cuisiner du riz

Mais on ne sait pas comment il fait (et nous n'avons pas à le savoir!)

```
public class ChefCuisinier {  
    public void cuisinerPates(){  
        bouillirEau();  
        cuirePates();  
    }  
  
    public void cuisinerRiz(){  
        bouillirEau();  
        cuireRiz();  
    }  
  
    private void bouillirEau() {}  
  
    private void cuirePates() {}  
  
    private void cuireRiz() {}  
}
```



Les visibilité

Default : Accessible uniquement aux objets du même package

Ici, je veux limiter les accès aux méthodes du moteur :

- Uniquement la voiture a le droit d'utiliser le moteur
- Un conducteur d'a pas le droit d'injecter directement de l'essence dans le moteur

Les méthodes du moteur ne sont donc pas privées, ni publique. Elles sont réservées aux objets du package

```
package vehicule.voiture;  
  
public class Voiture {  
    ... private Moteur moteur = new Moteur();  
    ... public void accelerer(){  
        ... moteur.injecterEssence();  
        ... moteur.combustion();  
    ... }  
}
```

```
package vehicule.voiture;  
  
class Moteur{  
    ... void injecterEssence() {}  
    ... void combustion() {}  
}
```



Les visibilités

Protected : Accessible aux objets du même package ainsi qu'aux classes qui l'étendent

Ici, la classe ChefCuisinier est abstraite. Les méthodes normalement privées:

- bouillirEau()
- cuirePates()
- cuireRiz()

doivent pouvoir être accédées de toute classe fille, quelque soit le package dans lequel elle est hébergée.

```
package chef.generique;

public abstract class ChefCuisinier {

    public abstract void cuisinerPates();
    public abstract void cuisinerRiz();

    protected void bouillirEau() {}
    protected void cuirePates() {}
    protected void cuireRiz() {}

}
```



Les visibilités

Par défaut, toute méthode ou attribut dans un objet est « **default/package-private** »

Par défaut, toute méthode ou attribut défini dans une interface est « **public** »

Les visibilité



Exercice :

Visibilité	Classe elle-même	Classes du package	Classes enfants (non package)	En dehors du package (non-enfant)
protected				
private				
default (package-private)				
public				

Les visibilité



Exercice :

Visibilité	Classe elle-même	Classes du package	Classes enfants (non package)	En dehors du package (non-enfant)
protected	✓	✓	✓	✗
private	✓	✗	✗	✗
default (package-private)	✓	✓	✗	✗
public	✓	✓	✓	✓

Les visibilité

Pour résumer (dans l'ordre) :

Visibilité	Classe elle-même	Classes du package	Classes enfants (non package)	En dehors du package (non-enfant)
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default (package-private)	✓	✓	✗	✗
private	✓	✗	✗	✗

Partie 4 :

Méthodes et attributs statiques



A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with a green color, while the others are white with black outlines.

Méthodes et attributs statiques

On appelle élément **statique** d'une classe tout élément attaché à cette classe plutôt qu'à l'une de ses instances.

Le mot-clef à utiliser est « **static** »

Un élément statique peut exister, être référencé, ou s'exécuter même si aucune instance de cette classe n'existe.

Un élément statique ne peut référencer **this**.



Méthodes et attributs statiques

Il est possible de définir quatre types d'éléments statiques :

- Champs statiques
- Méthodes statiques
- Blocs statiques (hors formation)
- Classes membre statiques (hors formation)

Méthodes et attributs statiques

Un attribut statique peut être appelé (si sa visibilité le permet) sans avoir besoin d'instanciation de la classe.

```
public class Chat {  
    ...  
    public static int nombrePattes = 4;  
}
```



```
int nombreDePattesDeChat = Chat.nombrePattes;
```

Attention, un changement sur un attribut statique changera la valeur pour tous les objets de cette classe !

```
Chat chat1 = new Chat();  
Chat chat2 = new Chat();  
chat1.nombrePattes = 3;  
System.out.println(chat2.nombrePattes);
```



3

A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with a green color, while the others are white with black outlines.

Méthodes et attributs statiques

Une des manières de créer une constante en Java est de déclarer une variable comme étant à la fois « **static** » et « **final** ».

De cette manière, on peut accéder aux variables sans avoir besoin d'instancier d'objet, et on ne peut pas la redéfinir :

```
public static final double PI = 3.14159265359;
```



Méthodes et attributs statiques

Les conventions de nommage pour les constantes sont :

- Tout en MAJUSCULES
- Séparés par des sous-tirets (underscores)

Méthodes et attributs statiques

De la même manière qu'un attribut statique, une **méthode statique** peut être appelée directement depuis la classe, sans besoin d'instanciation.

```
public class Calculatrice {  
    ...  
    public static double factorielle(int i){...}  
    public static double exponentielle(int i){...}  
    public static double cosinus(int i){...}  
    ...  
}
```

Une méthode statique ne peut utiliser que des attributs statiques (ou inhérents à la méthode).

Méthodes et attributs statiques



Exercice 1 : Créer une classe « Centaure » qui possède 2 bras et 4 pattes.

Exercice 2 : Ajouter un attribut « population » qui démarre à 0 et qui s'incrémente de 1 à chaque centaure créé.

Exercice 3 : Que se passe-t-il si vous déclarez une nouvelle variable de type Centaure mais sans l'initialiser (ou en l'initialisant à null)?

Méthodes et attributs statiques



Exercice 4 : Chez moi, je n'ai qu'un seul dictionnaire. Je n'achète pas de nouveau dictionnaire à chaque fois que j'ai besoin de faire une recherche.

L'idée est donc de modéliser ce comportement à travers un objet qui limite le nombre d'instances en mémoire à 1 (maximum).

- Pour simplifier, un dictionnaire n'a qu'un titre et celui dont je dispose est un Larousse
- Il ne doit pas être possible de pouvoir faire de « new »
- Il existe une méthode *getInstance()* qui me renvoie l'instance de l'objet qui existe déjà.

Partie 5 :

Les exceptions

SIMPLY EXPLAINED



`NullPointerException`



Les exceptions

Certaines méthodes, ou certaines actions peuvent éventuellement générer des erreurs :

- Un fichier n'a pas pu être ouvert
- Une division par zéro a eu lieu
- Une lecture a été tentée au-delà de la limite d'un tableau
- Etc...

Les exceptions

On dit alors qu'une exception est levée :

```
int cherieCaVaCouper = 1/0;
```

```
java.lang.ArithmeticException: / by zero  
  
    at com.bnpparibas.sit.pact.demo.domain.example.HelloWorld.calculerFactorielle(HelloWorld.java:14)  
    at com.bnpparibas.sit.pact.demo.domain.example.HelloWorld.test(HelloWorld.java:9) <22 internal calls>
```

...et votre programme s'arrête.

La génération d'un objet "exception" est la façon par laquelle la machine Java nous signale qu'une action s'est mal déroulée.



Les exceptions

Définition : *Une exception caractérise le déroulement non nominal d'un programme.*

Une exception peut être déclenchée de deux façons :

- L'exécution du programme ne se déroule pas de la façon prévue, et la machine Java génère une exception
- Le programme décide lui-même de déclencher une exception, afin de signaler à la méthode appelante que quelque chose ne se déroule pas comme prévu

A vertical line on the left side of the slide, featuring a series of circles. The top four circles are white with black outlines, and the bottom circle is solid blue.

Les exceptions

Une exception caractérise le déroulement non nominal d'un programme.

Une exception peut être déclenchée de deux façons :

- L'exécution du programme ne se déroule pas de la façon prévue, et la machine Java génère une exception
- Le programme décide lui-même de déclencher une exception, afin de signaler à la méthode appelante que quelque chose ne se déroule pas comme prévu

Les exceptions

Quelques exemples :

```
// Exemple 1
int i = 0 ;
int j = 0 ;
int k = i / j ; // déclenche une java.lang.ArithmeticException

// Exemple 2
String bonjourLeMonde = null ;
bonjourLeMonde.length() ; // déclenche la célèbre java.lang.NullPointerException

// Exemple 3
int [] quelquesEntier = new int [] { 3, 1, 4, 1, 5, 9 } ;
int z = quelquesEntier[1000] ; // déclenche java.lang.IndexOutOfBoundsException
```



Les exceptions

Pour déclencher vos propres exceptions, il faut utiliser le mot-clef « **throw** »

```
if (nombrePattes > 4) {  
    ... throw (new Exception("Ce chat n'est pas normal !"));  
}
```

Si une méthode porte en elle le déclenchement, (même conditionnel) d'une exception, elle doit le déclarer dans sa signature avec le mot-clef « **throws** »

```
private static void compterPattes() throws Exception { ... }
```

Sauf...

A vertical line on the left side of the slide, featuring a series of circles. The top four circles are white with dark outlines, and the bottom circle is solid blue.

Les exceptions

Il est possible de « lancer » des exceptions sans avoir besoin de le déclarer dans la signature.

Pour cela, il faut que les exceptions soient de type (ou un sous-type) de la classe « **RuntimeException** »

Pour différencier les deux, on parlera de :

- **Checked** (quand il faut les expliciter)
- **Unchecked** (pour les RuntimeException)

A vertical line on the left side of the slide, consisting of a series of circles. The top four circles are white with black outlines, and the bottom circle is solid blue.

Les exceptions

Toute exception jetée explicitement doit être soit propagée vers la méthode appelante, soit traitée localement.

La propagation d'une exception se fait simplement dans la déclaration de la méthode

```
public void checkup() throws Exception{
    ...compterPattes();
    ...//rappelVaccins();
    ...//...
}

private void compterPattes() throws Exception {
    ...if (nombrePattes > 4) {
        ...throw (new Exception("Ce chat n'est pas normal!"));
    }
}
```

Les exceptions

Pour être traitée localement, il faut « attraper » une Exception. Pour cela, on utilise les mots clef « **try** », « **catch** », « **finally** ».

```
public void checkup() {  
    try {  
        compterPattes();  
    } catch (NumberFormatException e) {  
        e.printStackTrace();  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        // ...  
    }  
}
```

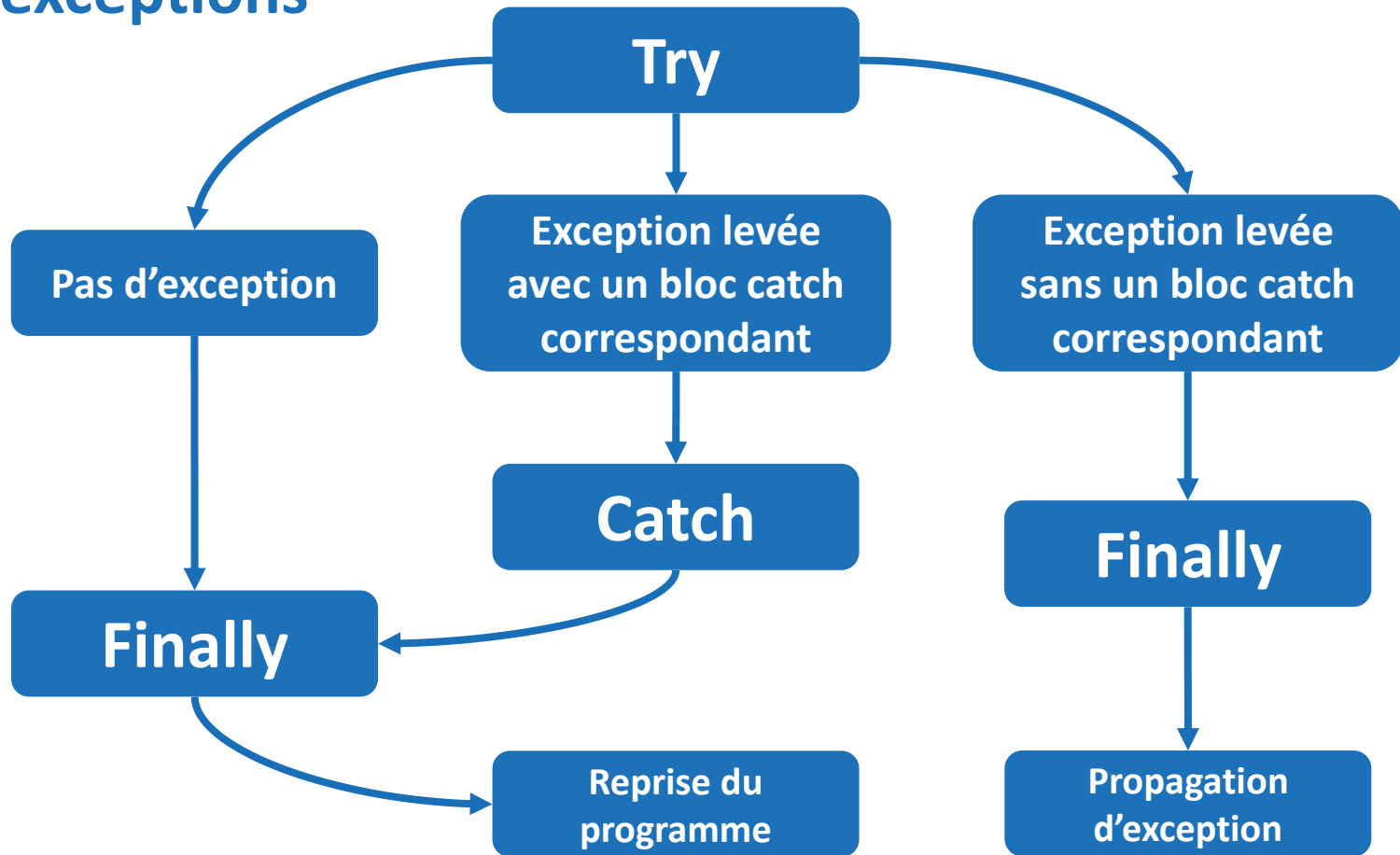
Try : Bloc de code « dangereux »

Catch : Que faire en cas de « NumberFormatException »

Catch : Que faire en cas de « Exception »

Finally (optionnel) : Que faire dans tous les cas

Les exceptions



A vertical line on the left side of the slide, featuring a series of circles. The top four circles are white with dark gray outlines, and the bottom circle is solid blue.

Les exceptions

Java dispose d'une librairie d'exceptions bien fournie, mais il est aussi possible de créer sa propre exception.

Pour cela, il suffit de créer une classe étendant la classe « **Exception** » (ou « **RuntimeException** » pour créer des unchecked exception).

Les exceptions

```
public class CatException extends Exception {  
    public CatException() {  
        //Message pas défaut  
        super("Ce chat n'est pas normal !!");  
    }  
    public CatException(String message) {  
        super(message);  
    }  
}
```

```
private void compterPattes() throws CatException {  
    if (nombrePattes > 4) {  
        throw (new CatException());  
    }  
}
```

```
public void checkup(){  
    try {  
        compterPattes();  
    } catch (CatException e) {  
        e.printStackTrace();  
    }  
}
```

```
demo.epita.CatException: Ce chat n'est pas normal !!  
— at demo.epita.Chat.compterPattes(Chat.java:21)  
— at demo.epita.Chat.checkup(Chat.java:13)  
— at demo.epita.HelloWorld.main(HelloWorld.java:10)
```


Les exceptions



Exercice 1 : Développer une méthode qui calcule le factoriel d'un entier. Que se passe-t-il passé l'itération 14? Pourquoi?

Exercice 2 : Trouver un moyen de tester si l'opération fonctionne correctement et renvoyer une exception de type « FactorielleException » si ce n'est pas le cas.

Exercice 3 : Faire en sorte que cette exception n'apparaisse pas dans la signature de la méthode.

Exercice 4 : Récupérer l'exception proprement dans la méthode main et afficher un message d'erreur.



Les exceptions

Exercice 5 : Créer 2 classes d'exception étendant FactorielleException :

- La première en cas d'incohérence de calcul
- La seconde en cas de nombre négatif en entrée.

Exercice 6 : Lever les exceptions au moment opportun

Exercice 7 : Afficher un message d'erreur différent selon le type d'exception levé

Conclusion

A vertical line on the left side of the slide, decorated with seven colored circles: green, cyan, orange, purple, lime green, and blue.

Conclusion

Félicitations !

Vous avez survécu à la POO en Java !

A ce stade, vous savez :

- Créer des classe (abstraites ou non) avec leurs constructeurs associés
- Créer des interface
- Hériter de ces dernières
- Jouer sur les visibilitées pour préserver l'encapsulation
- Ce qu'est un attribut ou une méthode statique
- Jongler avec des exceptions



Conclusion

Pour réviser (ou aller plus loin) :

<http://blog.paumard.org/cours/java/>

That's all Folks!