



Global Knowledge®

GKJAVA

Septembre 2019

Objectifs

Les fondamentaux de la programmation Java

Acquérir les compétences et connaissances nécessaires pour prendre en main l'environnement standard JAVA (JSE) et utiliser les outils de développement

- Développer du code Java, en manipulant Eclipse, en respectant les concepts objets et de bonnes pratiques de qualité de codage
- Etre capable de décrire les différentes technologies de Java SE, EE, ME
- Maîtriser les types de données et la syntaxe du langage Java
- Manipuler les données avec JDBC et connaitre les problématiques de performance et de cohérence des données

Présentation

- Qui suis-je ?



Présentation

- Qui êtes-vous ?



Logistique



- **Pause en milieu de session**



- **Vos questions sont les bienvenues. N'hésitez pas !**



- **Feuille d'évaluation à remettre remplie en fin de session**



- **Merci d'éteindre vos téléphones**

Sommaire

	Page
➤ Chapitre 1 : Introduction	7
➤ Chapitre 2 : Le langage JAVA	59
➤ Chapitre 3 : Compléments au langage Java	149
➤ Chapitre 4 : Classes Abstraites & Interfaces	215
➤ Chapitre 5 : Exceptions	229
➤ Chapitre 6 : Les génériques	261
➤ Chapitre 7 : Les expressions lambda	286
➤ Chapitre 8 : Les Streams	342
➤ Chapitre 9 : Thread	365
➤ Chapitre 10 : JavaBean	401
➤ Chapitre 11 : Accès au base de données	408
➤ Chapitre 12 : Entrées Sorties	478
➤ Chapitre 13 : JavaFX	521
Conclusion	576

Objectifs du chapitre

- L'orienté Objet
 - Objet & Classe
 - Encapsulation & Polymorphisme
 - Héritage & Associations
- JAVA
 - Historique
 - Versions
 - Gestion de la mémoire et de la sécurité

Qu'est-ce qu'une classe ?

- C'est un modèle de quelque chose d'abstrait.
- C'est la définition qui permet de créer des objets.
- Une classe peut être définie comme un **modèle caractérisant un ensemble d'éléments du système d'information**

Qu'est-ce qu'une classe ?

Actrice

- Un prénom
- Un nom
- Une date de naissance
- Une adresse
- Un lieu de naissance

Ce sont les propriétés

Elle a des attributs



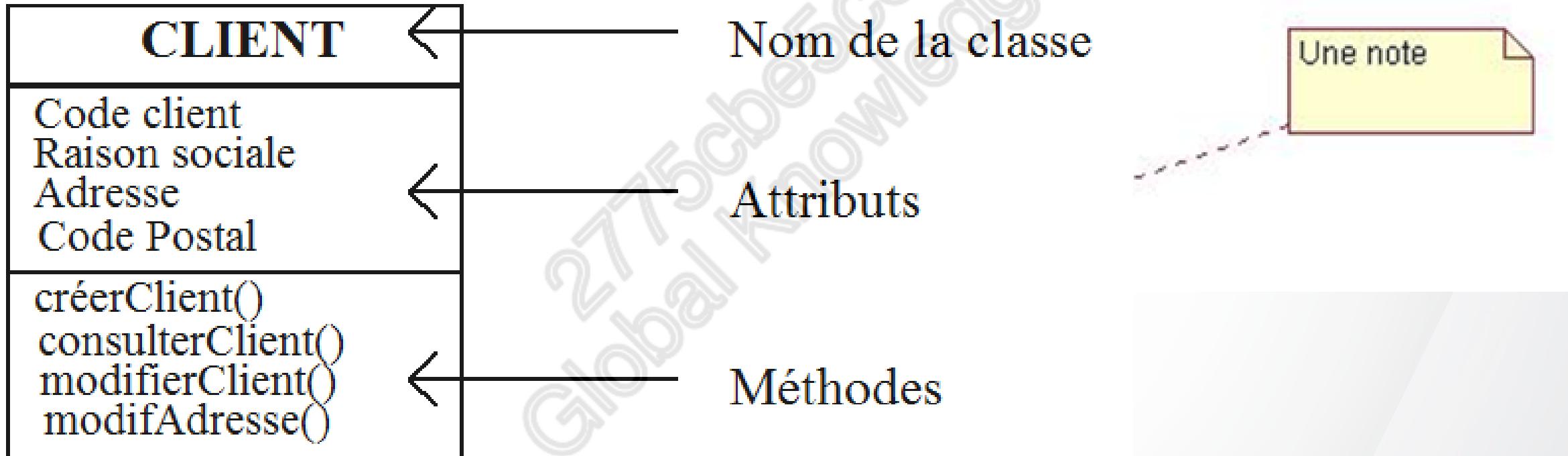
- Manger
- Boire
- Parler
- Dormir
- Rouvrir

Ce sont les actions
qu'elle peut réaliser
des méthodes

La Classe : Méthodes

- La liste des **méthodes** permet de décrire de manière exhaustive le comportement des éléments du système d'information qu'elle caractérise.
- Les trois principaux types de méthodes sont :
 - Les **constructeurs** qui interviennent lors de la création d'un élément ;
 - Les **accesseurs** qui permettent d'accéder à l'élément soit en consultation (on parle alors de sélecteur), en mise à jour (modificateur) ou dans un ordre défini (itérateur) ;
 - Les **destructeurs** qui interviennent lors de la suppression d'un élément

La Classe : Exemple

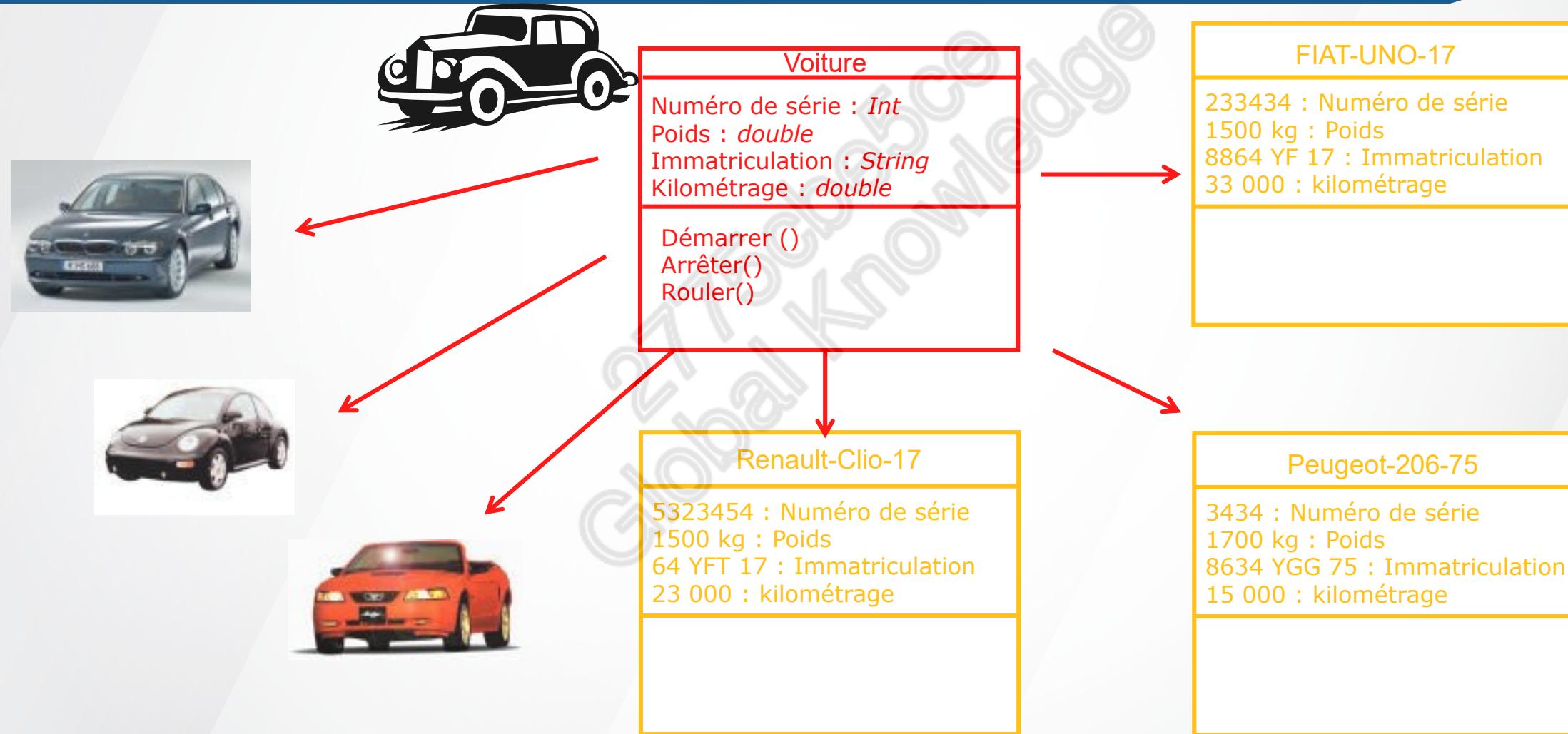


- **NB1** : La notation adoptée ci-dessus ainsi que dans la suite de la formation est le formalisme UML (Unified Modeling Language).

L'objet

- L'objet est l'**occurrence ou l'instance d'une classe**.
- Le procédé qui consiste à créer un objet à partir d'une classe est appelé **instanciation**.
- L'instanciation d'un objet est effectuée par l'activation d'un de ses constructeurs. De même, sa suppression ne pourra être obtenue que par l'activation de son destructeur

L'objet : Exemple



Classe/objet : Exemple

➤ Première étape :

On dispose d'une classe
Client = Le Modèle

Cette classe a été écrite par
un développeur.

```
package demoCours;  
  
public class Client {  
    //Attributs  
  
    int numero ;  
  
    String nom ;  
  
    public Client()    {// Constructeur - Initialise les attributs  
        nom = "inconnu" ;  
        numero = 0 ;  
    }  
  
    public String retourneInfo()    {  
        // Méthode qui retourne une chaîne de caractères  
        // contenant les info client  
  
        return " Le client est " + numero + " - " + nom;  
    }  
}  
} // fin classe
```

Classe/objet : Exemple

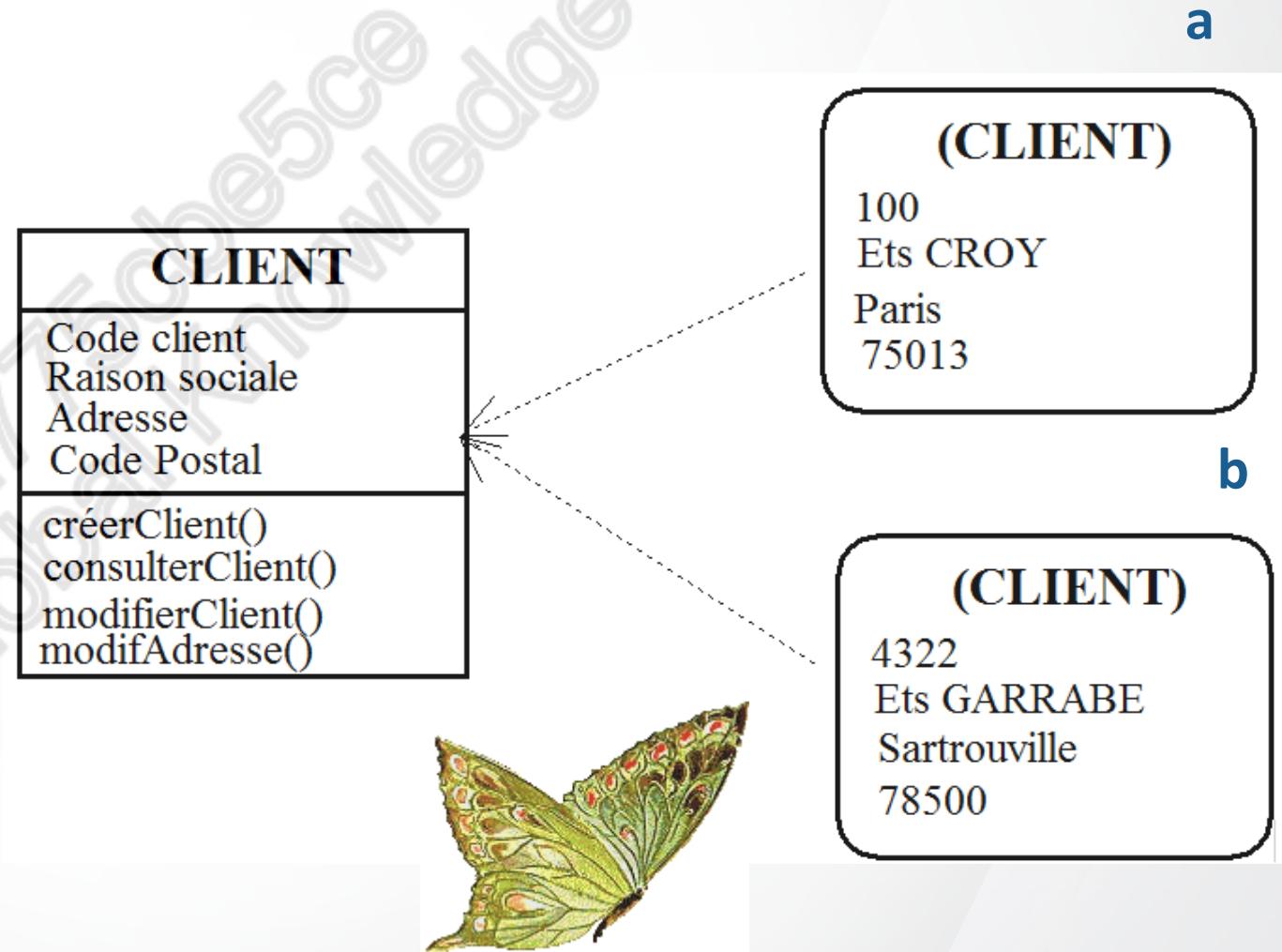
➤ Deuxième étape :

On crée un objet nommé **a**, instance de la classe Client ayant comme attributs :

- Un code client valant 100
- Une raison sociale valant Ets CROY

On crée un objet nommé **b**, instance de la classe Client

Les objets **a** et **b** sont en mémoire



Classe/objet : Exemple

- La troisième étape consiste à manipuler les objets, par l'envoi de messages = les méthodes.

Les étapes 2 et 3 s'écrivent dans un code indépendant du modèle, ici un exécutable Java

Les objets sont stockés dans des variables et se manipulent comme tout autre type de variable

Résultats console :

Le client est 0 – inconnu

```
package demoCours;
public class TestClient
{
    public static void main(String[] args)
    {
        // Manipulation d'un entier
        // Déclaration de la variable x de type int
        int x;
        // Initialisation de la variable
        x = 100;
        // Manipulation de la variable
        x = x + 200;

        // Manipulation d'un Client
        // Déclaration de la variable a de type Client
        Client a;
        // Initialisation de la variable : consiste à appeler le constructeur
        a = new Client();
        // Manipulation de la variable
        String info = a.retoumeInfo();

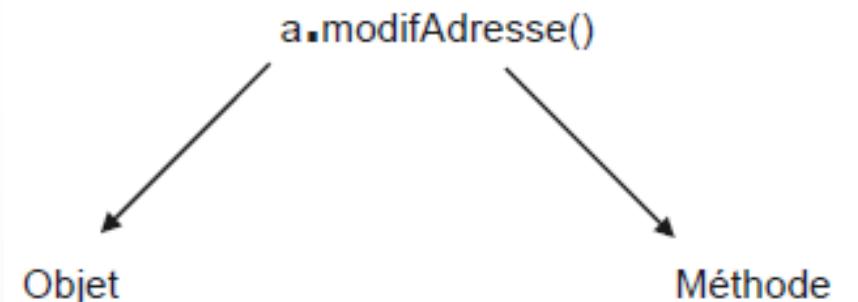
        // Affichage à la console système
        System.out.println(info);
    }
}
```

Les méthodes : Présentation

- Les méthodes contiennent les traitements à effectuer pour gérer les objets.
- La classe dont est issu l'objet doit contenir tous les traitements possibles sur cet objet.
- Pour manipuler un objet on lui applique une de ces méthodes
- Exemple :

Dans l'exemple du **Client**, on ajoute un attribut adresse et une méthode modifAdresse(...) dans le modèle.

Si on veut modifier l'objet **a**, déclaré dans notre exécutable, on lui envoie un message, c'est à dire on applique la méthode modifAdresse() à la variable **a**



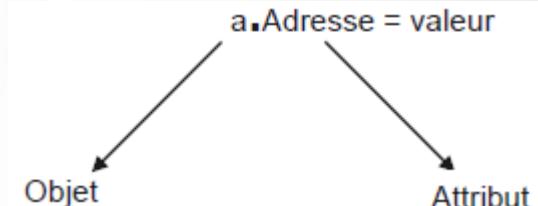
Les méthodes : Accès à un attribut

- On peut accéder à la valeur d'un attribut directement par une notation pointée, cette caractéristique n'est possible que si l'on ne fait pas d'encapsulation, elle est donc à éviter :
- Exemple de code Java :

Résultats console

Le client est 100 - Ets CROY

```
package demoCours;  
  
public class TestClient  
{  
  
    public static void main(String[] args)  
    {  
  
        // Instanciation d'un Client  
        // Déclaration et initialisation de la variable a de type Client  
        Client a = new Client();  
  
        // Manipulation des attributs nom et numero du client a  
        a.nom = "Ets CROY";  
        a.numero = 100;  
  
        // Affichage à la console système en utilisant la méthode de l'obj  
        System.out.println(a.retourneInfo());  
    }  
}
```



La Classe : L'encapsulation

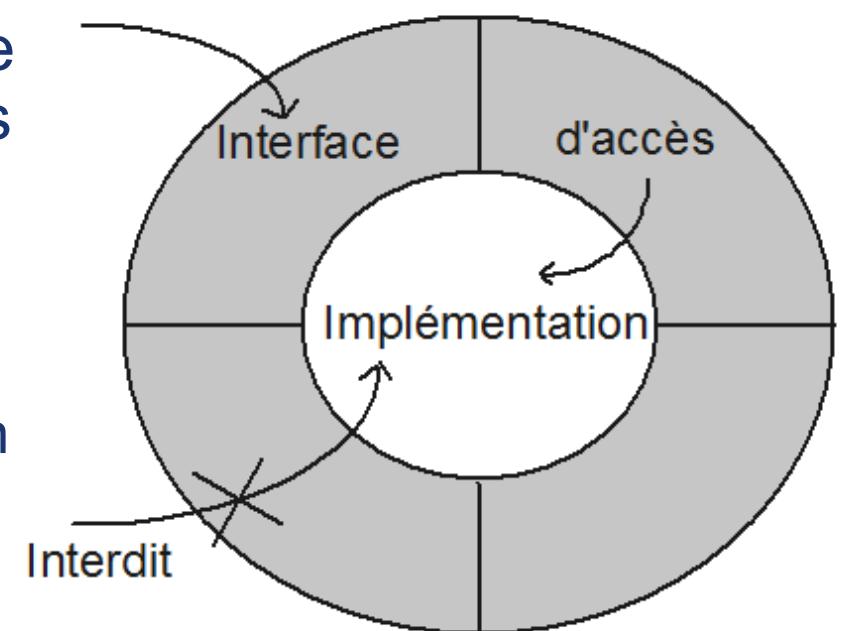
- Pour garantir la cohérence des valeurs d'attributs des occurrences d'une classe, il est intéressant de pouvoir forcer l'utilisateur de l'occurrence (le développeur d'applications) à n'accéder à son état que par l'intermédiaire de ces méthodes.
- Ainsi si l'on veut garantir le respect de la règle de gestion du *nom client* de l'exemple page précédente (*le nom doit être en majuscule*), on protège cette zone qui devient inaccessible directement depuis l'environnement extérieur à la classe, sa gestion étant prise en charge uniquement par la méthode *modifNom(...)*.

La Classe : L'encapsulation

- On fera de même avec les autres attributs de la classe si l'on souhaite s'assurer que pour tout client, les zones *adresse* et *numero* ont été valorisées. Il sera possible aussi de valoriser le numéro du client dans le constructeur de manière automatique et empêcher ainsi sa mise à jour.
- Ce procédé de masquage de tous les attributs d'une classe qui ne seront donc accessibles que par le filtre des méthodes, aussi bien en consultation qu'en mise à jour, est connu sous le nom d'**encapsulation**.

La Classe : L'encapsulation

- Loin d'être une caractéristique anodine du modèle objet, l'encapsulation en est un des points forts puisqu'elle induit un couplage faible entre les programmes applicatifs et les éléments du système d'information, ce qui est particulièrement appréciable, entre autres, dans un contexte de maintenance.
- **Remarque :**
 - L'implémentation du modèle objet varie fortement d'un langage de programmation à l'autre. En Java, en C# et en C++ par exemple, l'encapsulation n'est qu'une possibilité offerte alors qu'en Smalltalk elle est incontournable.
 - Quoiqu'il en soit, il est fortement conseillé de la mettre en œuvre dans tous les contextes



La Classe : L'encapsulation

➤ Exemple de code java - Encapsulation

L'encapsulation s'effectue au niveau de la définition, donc de la classe. Si l'on interdit l'accès aux attributs, il faut mettre à disposition des méthodes permettant de lire l'attribut et de le mettre à jour. On nomme ces méthodes des getter/setter.

```
package demoCours;
public class Client {

    // Attributs
    private int numero;
    private String nom;

    public Client() {
        // Constructeur - Initialise les attributs
        nom = "inconnu";
        numero = 0;
    }

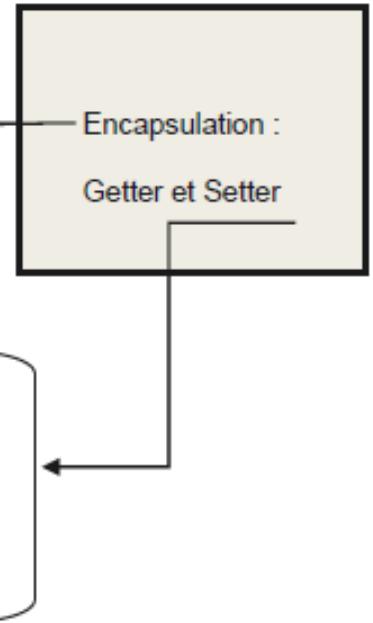
    public int getNumero() {return numero; }

    public void setNumero(int num) {numero = num; }

    public String getNom() {return nom; }

    public void setNom(String n) {nom = n; }

    public String retourneInfo(){
        return " Le client est " + numero + " - " + nom;
    }
}
```



La Classe : L'encapsulation

Résultats console

```
package demoCours;
public class TestClient {

    public static void main(String[] args) {

        // Instanciation d'un Client
        // Déclaration et initialisation de la variable a de type Client
        Client a = new Client();

        // Manipulation des attributs nom et numéro du client a
        a.setNom("Ets CROY");
        a.setNumero(100);
        //Affichage à la console système en utilisant la méthode de l'obj
        System.out.println(a.retourneInfo());
    }
}
```

Le client est 100 - Ets CROY

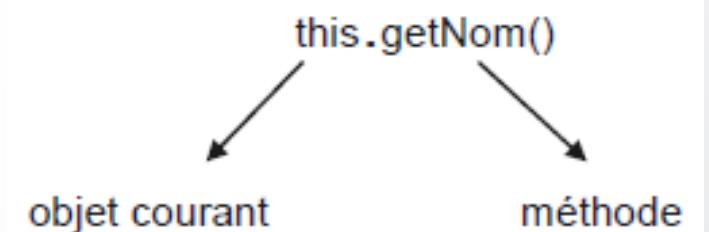
La Classe : L'encapsulation - Réutilisation & this

- On regarde de plus près la méthode `retourneInfo()`. Celle-ci doit retourner le nom et le numéro du client.
- Si l'on choisit de faire de l'encapsulation, il faut utiliser ce principe, même à l'intérieur de son propre code
 - **On réutilise donc dans cette méthode les méthodes `getNom()` et `getNuméro()`.**

Lorsque l'on appelle une méthode, on doit lui préciser sur quel objet elle s'applique.

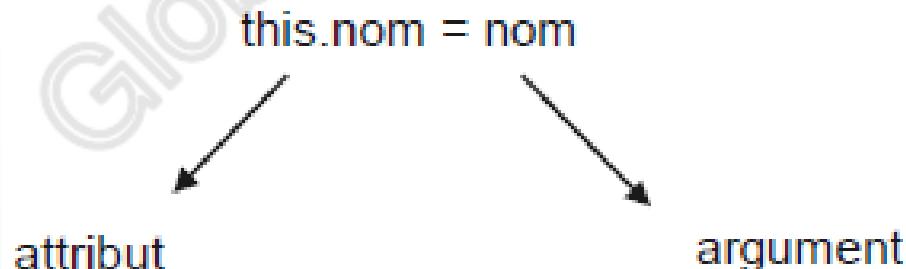
Dans la définition de `retourneInfo()` on ne connaît pas encore l'objet en cours. On a donc recours à une pseudo-variable qui permet de préciser l'objet courant.

En java on utilise le « **this** »



La Classe : L'encapsulation - Réutilisation & this

- Dans les « setter », on passe en argument une variable (type et nom) qui renseignera l'attribut. En général, cette variable porte le même nom que l'attribut. Ce qui provoquerait la situation suivante :
« nom = nom ».
- Il faudra donc différencier les variables. Le this permet de référencer l'attribut, on aura donc



La Classe : L'encapsulation - Réutilisation & this

L'exécutable reste inchangé

```
package demoCours;
public class Client {

    // Attributs
    private int numero;
    private String nom;

    public Client() {
        // Constructeur - Initialise les attributs
        nom = "inconnu";
        numero = 0;
    }

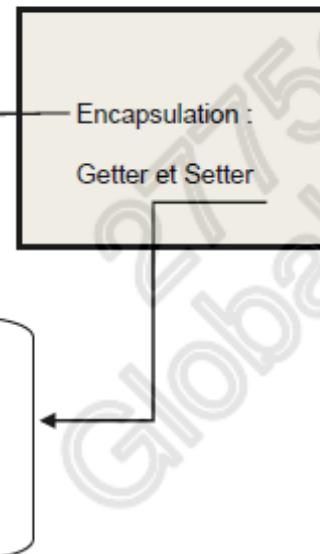
    public int getNumero() {return numero; }

    public void setNumero(int num) {numero = num;}

    public String getNom() {return nom; }

    public void setNom(String n) {nom = n}

    public String retourneInfo(){
        return " Le client est " + numero + " - " + nom;
    }
}
```



```
package demoCours;
public class Client {
    // Attributs
    private int numero;
    private String nom;

    public Client() {
        // Constructeur - Initialise les attributs
        nom = "inconnu";
        numero = 0;
    }
    public int getNumero() {
        return numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String retourneInfo(){
        return " Le client est " + this.getNumero() + " - " + this.getNom();
    }
}
```

Polymorphisme

- On appelle **polymorphisme**, la caractéristique d'une méthode à recouvrir des réalités différentes de manière transparente pour l'utilisateur des classes.
- Le polymorphisme offre aux objets la possibilité d'appartenir à plusieurs catégories à la fois.
- En effet, nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges.
- Mais, on peut écrire l 'expression suivante :

3 pommes + 5 oranges = 8 fruits

Polymorphisme

- Dans l'exemple précédent la méthode **retourneInfo()**, consiste à retourner l'ensemble des informations du client.
 - On appelle la méthode sans argument.
- Il est possible de créer une autre méthode **retourneInfo(argument)**, celle-ci reçoit en argument une chaîne de caractères qui retourne cette chaîne et les informations du client.
 - On manipule une méthode de même nom, le code qui sera exécuté est différent mais cela est transparent pour l'utilisateur.

Polymorphisme

- Dans une même classe, plusieurs méthodes peuvent aussi porter le même nom, l'évaluateur sait quelle méthode il doit utiliser en fonction des arguments reçus (combinaison nombre d'arguments + types).
- Ce type de polymorphisme est nommé une **surcharge**
- On parle de **signature** de la méthode :
 - Son nom
 - Son nombre d'arguments
 - Les types d'arguments



signature

Polymorphisme

➤ Exemple de code java – La classe - Polymorphisme

➤ L'exécutable

```
package demoCours;
public class TestClient {

    public static void main(String[] args) {
        // Instanciation d'un Client
        // Déclaration et initialisation de la variable a de type Client
        Client a = new Client();
        // Manipulation des attributs nom et numero du client a

        a.setNom("Ets CROY");
        a.setNumero(100);
        // Affichage à la console système en utilisant la méthode de l'obj
        System.out.println(a.retourneInfo());
        System.out.println(a.retourneInfo("Bonjour"));

    }
}
```

```
package demoCours;
public class Client {
    //Attributs
    private int numero;
    private String nom;

    public Client() {
        // Constructeur
        nom = "inconnu";
        numero = 0;
    }

    public int getNumero() { return numero; }

    public void setNumero(int numero) {this.numero = numero; }

    public String getNom() {return nom;}

    public void setNom(String nom) {      this.nom = nom; }

    public String retourneInfo(){
        return " Le client est " + this.getNumero() + " - " + this.getNom();
    }

    public String retourneInfo(String info){
        return info + " : " + this.retourneInfo();
    }
}
```

Résultats console :

```
Le client est 100 - Ets CROY
Bonjour : Le client est 100 - Ets CROY
```

Héritage

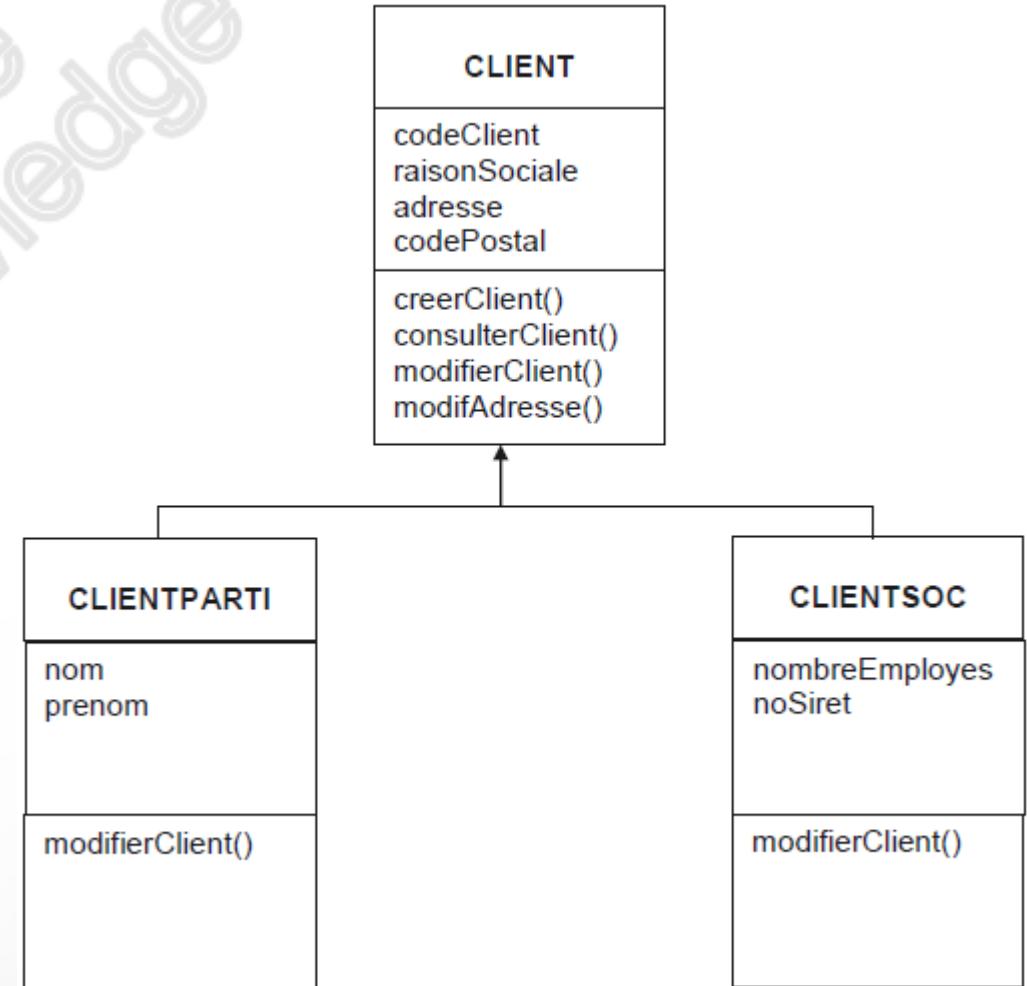
- Deux classes, dont la distinction est justifiée, peuvent néanmoins présenter un certain nombre de similitudes, qu'il peut être intéressant de factoriser afin d'éliminer tout risque de redondance et de faciliter les maintenances ultérieures.
- Ce processus est connu sous le nom de **généralisation** et se traduit par l'établissement d'un lien hiérarchique entre une classe, dénommée **sur-classe** ou **classe-mère**, et sa **sous-classe** ou **classe-fille**.
- Le processus inverse de la généralisation est la **spécialisation**. Une classe définie de manière trop globale peut être détaillée si les règles de gestion à implémenter le justifient.
- On parle d'**héritage**, la sous-classe récupère les attributs et les méthodes de la sur-classe.
- Remarque : on n'hérite pas des constructeurs.

Héritage

- On peut redéfinir une méthode de la sur-classe dans la sous-classe, on utilise encore le polymorphisme.
- On dit qu'il y a **redéfinition** d'une méthode lorsqu'une méthode héritée d'une sur-classe est annulée et remplacée par une méthode d'une sous-classe.

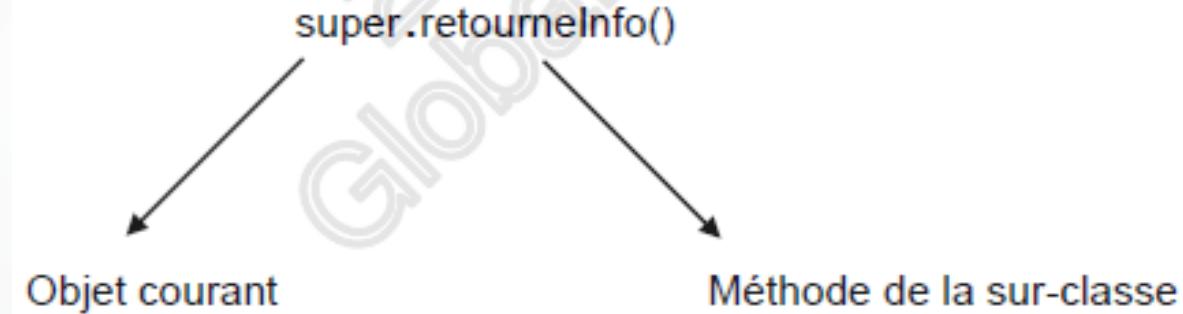
Héritage : Exemple

- Dans l'exemple précédent, on décide :
 - De gérer des clients particuliers, afin de connaître le nom, le prénom...
 - De gérer des clients « société » pour connaître le nombre d'employés, le numéro de Siret...
- Ces deux types de clients sont des clients, ils vont donc hériter de la classe client.



Héritage : Exemple

- Dans notre exemple, on souhaite dans la redéfinition de la méthode `retourneInfo()` faire référence à la méthode du même nom dans la sur-classe.
- Pour faire référence à la méthode de la classe mère, on utilise la pseudo-variable « **super** ».



Héritage : Exemple

➤ Exemple de code java - Héritage et super

L'exécutable

```
package demoCours;
public class TestHeritage {
    public static void main(String[] args) {

        // Déclaration et init d'un Client
        Client a = new Client();
        // Manipulation des attributs
        a.setNom("Ets CROY");
        a.setNumero(100);
        // Affichage à la console système
        System.out.println(a.retourneInfo());

        // Déclaration et init d'une Societe
        ClientSociete s= new ClientSociete();
        // Affichage à la console système
        System.out.println(s.retourneInfo());
    }
}
```

Résultats console :

```
package demoCours;
public class ClientSociete extends Client {
    // Attributs spécifiques aux Societes
    private int nbEmployes;
    private String siret;

    public ClientSociete() {
        // On n'hérite pas des constructeurs - il faut les définir
        // On initialise les attributs de la classe fille
        nbEmployes = 0;
        siret = "numero de siret";
    }

    public int getNbEmployes() {return nbEmployes;}

    public void setNbEmployes(int nbEmployes) {this.nbEmployes = nbEmployes;}

    public String getSiret() {return siret;}

    public void setSiret(String siret) {this.siret = siret;}

    public String retourneInfo() {
        // On ajoute au code de la classe mère le spécifique de la fille
        return super.retourneInfo() + " Societe: " + this.getSiret() + " Nb d'employés " +
    this.getNbEmployes();
    }
}
```

```
Le client est 100 - Ets CROY
Le client est 0 - inconnu Societe: numero
de siret Nb d'employés 0
```

Héritage : Exemple

➤ Remarque :

- La méthode « **retourneInfo(String info)** » de la classe mère n'a pas été redéfinie dans la classe fille.
- Si dans notre exécutable, on exécute la méthode du même nom pour l'objet de type ClientSociete, on provoquera en fait l'exécution de la méthode « **retourneInfo()** » de la classe fille.
- En effet, dans la classe mère, on fait appel dans « **retourneInfo(String info)** » à « **this.retourneInfo()** », comme « **this** » est l'objet en cours, c'est bien « **retourneInfo()** » de ClientSociete qui est sollicité.

Résultats console

```
package demoCours;  
  
public class TestHeritage {  
  
    public static void main(String[] args) {  
  
        // Déclaration et int d'un Client  
        Client a = new Client();  
  
        // Manipulation des attributs  
        a.setNom("Ets CROY");  
        a.setNumero(100);  
  
        //Affichage à la console système  
        System.out.println(a.retourneInfo());  
  
        // Déclaration et int d'une Societe  
        ClientSociete s = new ClientSociete();  
  
        // Affichage à la console système  
        System.out.println(s.retourneInfo());  
        System.out.println(s.retourneInfo("Soc"));  
    }  
}
```

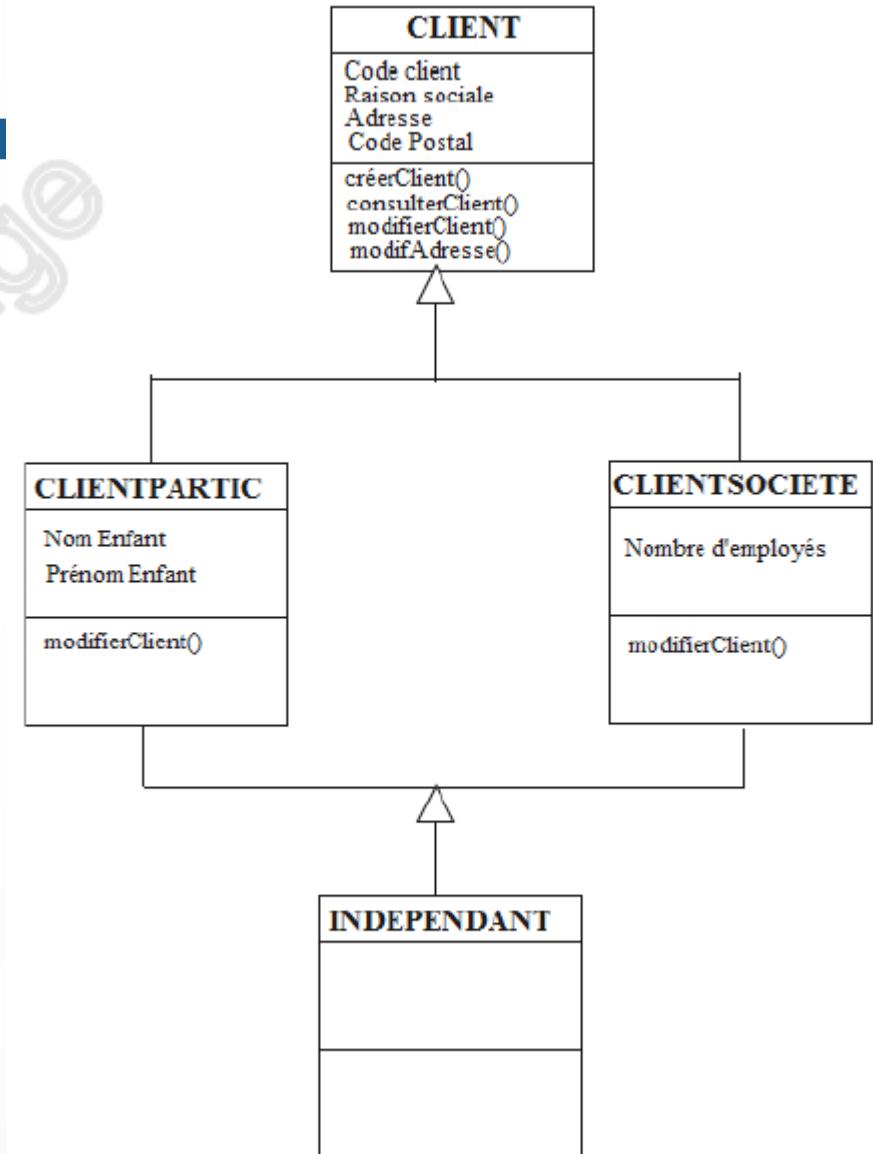
Le client est 100 - Ets CROY

Le client est 0 - inconnu Societe: numero de siret Nb d'employés 0

Soc : Le client est 0 - inconnu Societe: numero de siret Nb d'employés 0

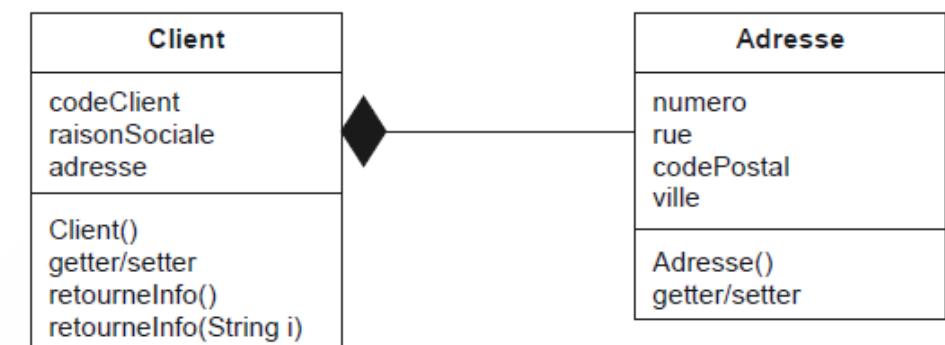
Héritage multiple

- Le modèle objet autorise l'**héritage multiple**, capacité dont dispose une sous-classe d'hériter de plusieurs classes mères de même niveau
- L'héritage multiple n'existe pas en Java.
 - Cependant, les Interfaces permettent de reconstituer une sorte de pseudo héritage.



Association – Composition et agrégation

- Un autre lien entre les classes existe.
- Il se nomme lien d'association et/ou lien de composition.
- Dans l'exemple ci-dessous, on dit que le Client est composé d'Adresse, c'est-à-dire que dans ses attributs la classe Client définit un attribut Adresse.
- On crée une classe Adresse pour définir un nouveau type, qui représente une adresse telle qu'on la souhaite.
 - Cette adresse sera un modèle, utilisé dans toutes nos définitions, pour le fournisseur, le salarié...
 - Notre client peut avoir plusieurs adresses, qui seront définies toujours sur le même modèle.



Association – Exemple

➤ La classe Adresse

➤ La classe Client

```
package demoCours;
public class Client {
    // Attributs
    private int numero;
    private String nom;
    private Adresse adresse;

    public Client() {
        // Constructeur - Initialise les attributs
        nom = "inconnu";
        numero = 0;
        adresse = new Adresse();
    }
    public Adresse getAdresse() {return adresse;}
    public void setAdresse(Adresse adresse) {this.adresse = adresse; }

    // Autres getter/setter .../...

    public String retourneInfo(){
        return "Le client est " + this.getNumero() + " - " + this.getNom()
            + " il habite: " + this.getAdresse().getVille();
    }
}
```

```
package demoCours;

public class Adresse {
    private int numero;
    private String ville;

    public Adresse() {
        // init attributs
        numero = 0;
        ville = "à renseigner";
    }

    public int getNumero() {return numero;}

    public void setNumero(int numero) {this.numero = numero;}

    public String getVille() {return ville;}

    public void setVille(String ville) {this.ville = ville;}
}
```

Association – Exemple

- L'exécutable
- Il est possible de relancer les exécutables TestClient et TestHeritage sans les avoir modifiés. La nouvelle information concernant l'adresse est bien prise en compte.
- C'est aussi une des forces des technologies objet : pouvoir faire évoluer sans avoir à tout reconstruire.

TestClient

```
Le client est 100 - Ets CROY il habite: à renseigner
Bonjour : Le client est 100 - Ets CROY il habite: à renseigner
```

Résultats console

TestHeritage

```
Le client est 100 - Ets CROY il habite: à renseigner
Le client est 0 - inconnu il habite: à renseigner Societe: numero de siret
Nb d'employés 0
Soc : Le client est 0 - inconnu il habite: à renseigner Societe: numero de
siret Nb d'employés 0
```

Objectifs du chapitre

- L'orienté Objet
 - Objet & Classe
 - Encapsulation & Polymorphisme
 - Héritage & Associations
- JAVA
 - Historique
 - Versions
 - Gestion de la mémoire et de la sécurité

Historique de Java (1)

- Java a été développé à partir de décembre 1990 par une équipe de Sun Microsystems dirigée par James Gosling
- Au départ, il s'agissait de développer un langage de programmation pour permettre le dialogue entre de futurs ustensiles domestiques
- Or, les langages existants tels que C++ n'étaient pas à la hauteur : recompilation dès qu'une nouvelle puce arrive, complexité de programmation pour l'écriture de logiciels fiables...



Historique de Java (2)

- 1990 : Ecriture d'un nouveau langage plus adapté à la réalisation de logiciels embarqués, appelé OAK
 - Petit, fiable et indépendant de l'architecture
 - Destiné à la télévision interactive
 - Non rentable sous sa forme initiale
- 1993 : le WEB « décolle ». Sun redirige ce langage vers Internet : les qualités de portabilité et de compacité du langage OAK en ont fait un candidat parfait à une utilisation sur le réseau. Cette réadaptation prit près de 2 ans.
- 1995 : Sun rebaptisa OAK en Java (*café en argot américain, machine à café : endroit où se réunissait James Gosling et ses collaborateurs*)

Historique de Java (3)

- Les développeurs Java ont réalisé un langage indépendant de toute architecture de telle sorte que Java devienne idéal pour programmer des applications utilisables dans des réseaux hétérogènes, notamment Internet.
- Le développement de Java devint alors un enjeu stratégique pour Sun et l'équipe écrivit un navigateur appelé HotJava capable d'exécuter des programmes Java.
- La version 2.0 du navigateur de Netscape a été développée pour supporter Java, suivie de près par Microsoft (Internet Explorer 3)
- L'intérêt pour la technologie Java s'est accru rapidement : IBM, Oracle et d'autres ont pris des licences Java.

Les différentes versions de Java

- De nombreuses versions de Java depuis 1995
- Évolution très rapide et succès du langage
- Une certaine maturité atteinte avec Java 2
- Mais des problèmes de compatibilité existaient :
 - Entre les versions 1.1 et 1.2/1.3/1.4
 - Avec certains navigateurs

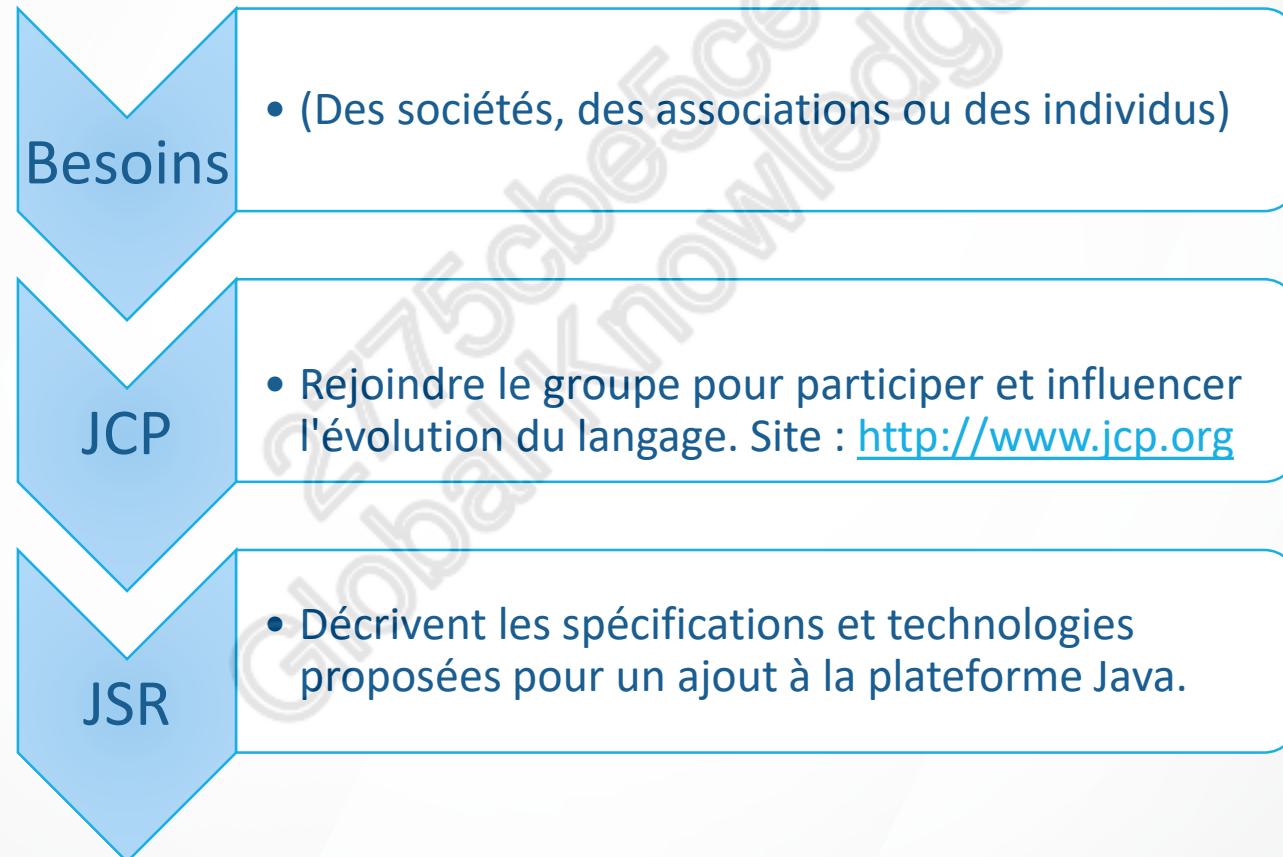
Année	Événements
1995	Premier lancement commercial en mai.
1996	Sortie en janvier du JDK 1.0 et en septembre lancement d'une association regroupant des développeurs Java (JDC Java Developer Connection)
1997	Sortie en février du JDK 1.1
1998	Lancement en décembre de J2SE et de la communauté JCP (Java Community Process)
1999	Lancement en décembre de J2EE
2000	Sortie en mai de J2SE 1.3
2002	Sortie de J2SE 1.4, nom de code Merlin
2004	Sortie de la version 1.5 mais connue sous l'appellation Java SE 5.0 , nom de code Tiger
2006	Sortie de la version Java SE 6 connue sous son nom de code Mustang
2009	Oracle rachète Sun
2011	Java SE 7, nom code Dolphin, première version sous la licence GPL
2014	Sortie de Java SE 8 nouveautés majeures les <i>closures</i> (Période de maintenance jusqu'en 2017 support étendu en 2025)
2016	Lancement de la version 9 (Plus modulaire, plus performant, intégration de JSON : format de données), nom code Umbrella. Après cette version tant attendue, Oracle accélère les upgrades, tous les 6 mois. Java 10 et 11 voient le jour en 2018.
2018	
2019	Java 12 et Java 13 (en septembre) Depuis 2018, les versions apportent des nouvelles fonctionnalités pour les développeurs mais surtout des améliorations du JDK.

Editions et versions

- Java ME, Java Micro Edition,
- Java SE, Java Standard Edition,
- Java EE, Java Enterprise Edition.

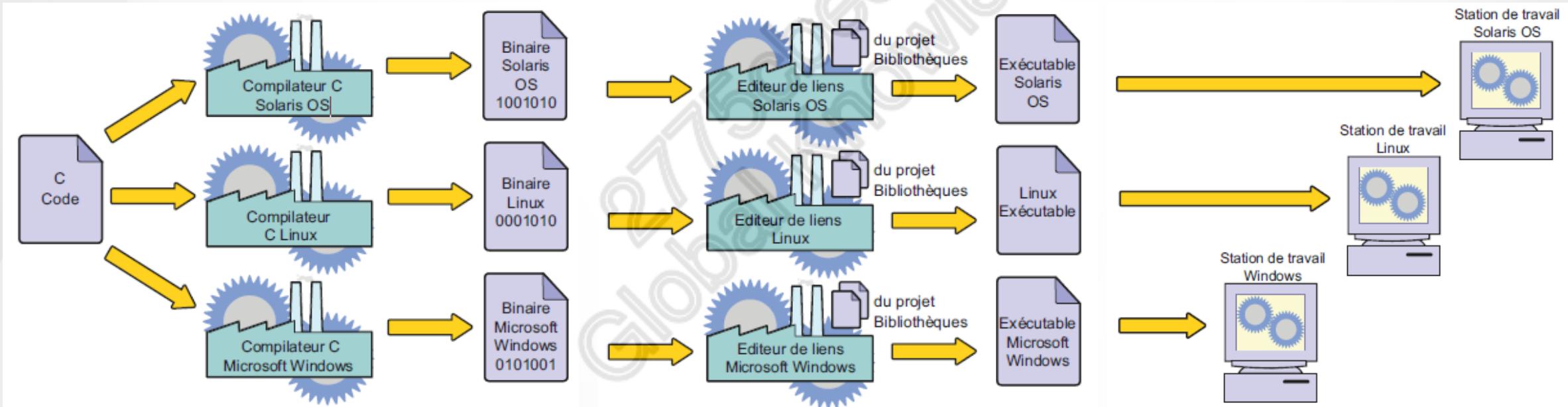


Comment se fait l'évolution de Java ?



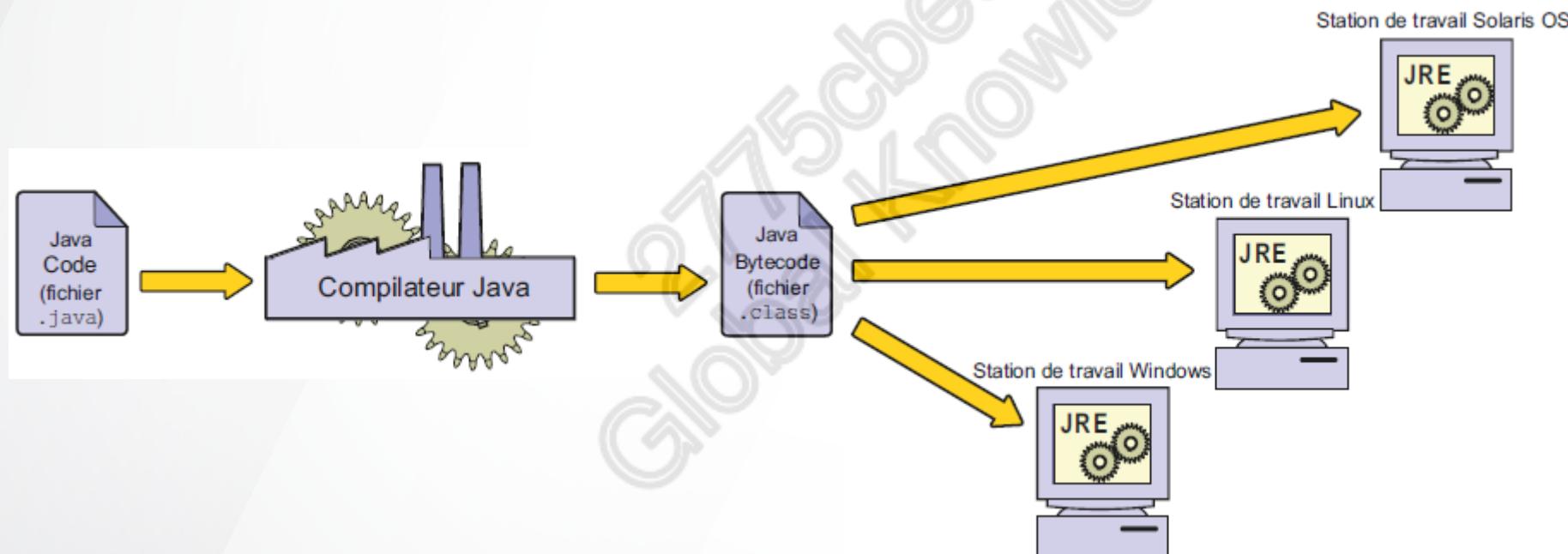
Architecture & caractéristiques du langage

➤ Programmes dépendants de la plate-forme



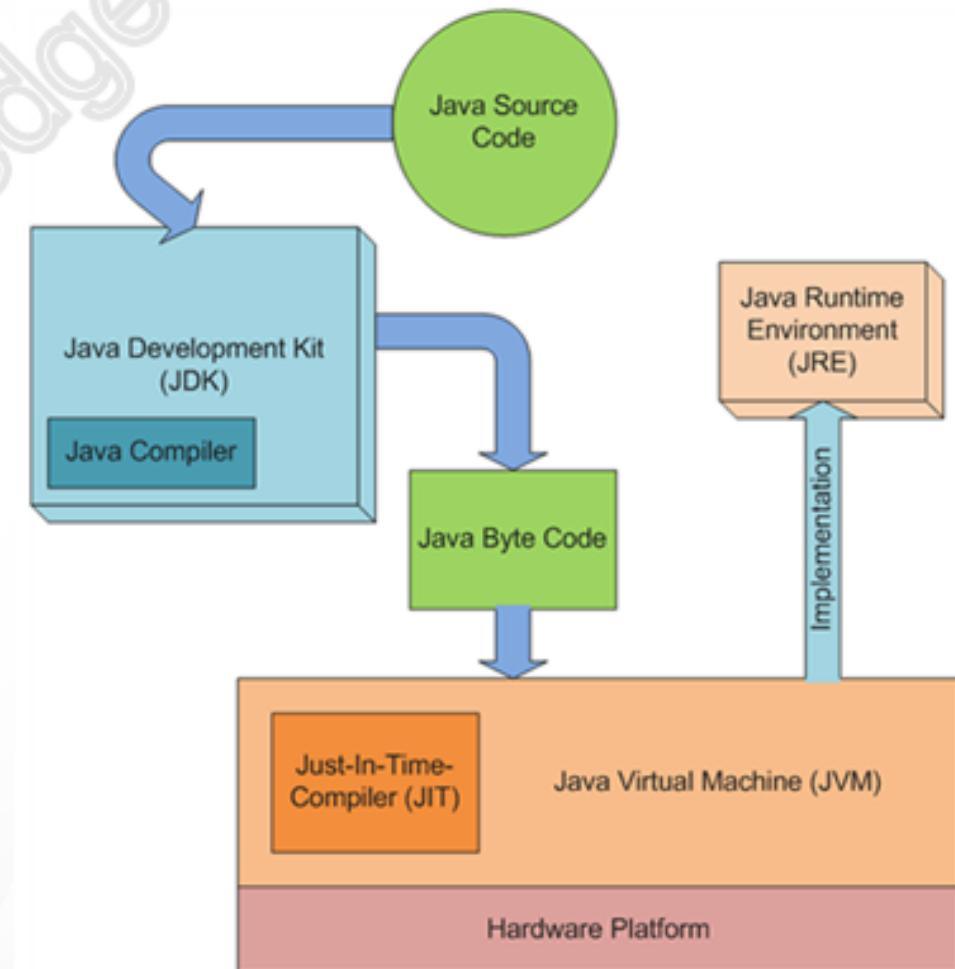
Architecture & caractéristiques du langage (2)

➤ Programmes indépendants de la plate-forme



Architecture & caractéristiques du langage (3)

- **JDK** : Il s'agit d'un paquet (bundle) de logiciels que vous pouvez utiliser pour développer des applications Java
- **JRE** : Il s'agit d'une implémentation de la machine virtuelle Java qui exécute des programmes Java
- **JVM** : est un appareil informatique fictif (abstrait) qui exécute des programmes compilés sous forme de bytecode Java.



Architecture & caractéristiques du langage

(4)

- Caractéristiques :
 - Java est un langage orienté objet.
 - Le langage Java est fortement typé : toutes les variables sont typées et il n'existe pas de conversion implicite qui risquerait une perte de données.
 - Si une telle conversion doit être réalisée, le développeur doit obligatoirement l'expliciter.
 - L'héritage multiple n'existe pas, mais pourra être simulé en utilisant les interfaces.
 - L'encapsulation est fortement conseillée, elle s'obtient en déclarant chaque attribut comme privé (mot clé private).
 - Java permet le polymorphisme : il est possible de surcharger / redéfinir des méthodes.

Architecture & caractéristiques du langage (5)

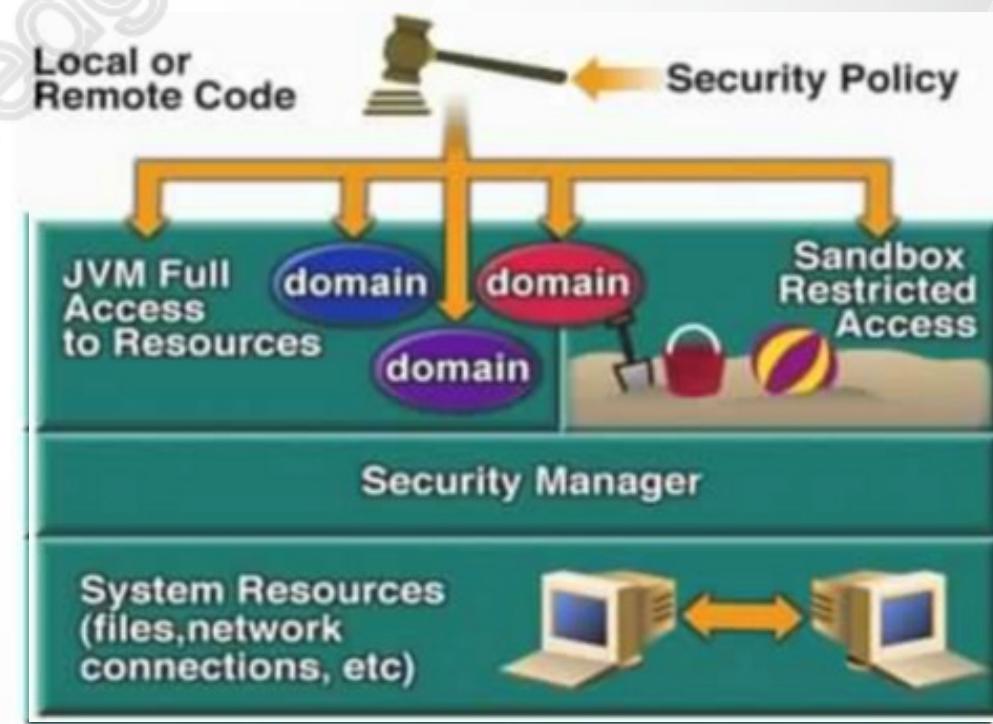
- Caractéristiques
 - Il est possible d'archiver un ensemble de classes, on parle alors de fichiers JAR, l'archive est un fichier ayant comme extension un **.jar**.
 - Le fichier archive permet au navigateur d'utiliser une seule connexion quand la classe utilise plusieurs classes, images, fichiers audio et autres.
 - Le fichier archive est compressé.
- Java est indépendant de toute plate-forme.

Gestion de la mémoire

- Dans certains langages de programmation, le programmeur doit s'occuper lui même de détruire les objets inutilisables.
 - La gestion de la mémoire est entièrement prise en compte par Java. (Pas de pointeur)
 - La fonctionnalité de **ramasse-miettes** (garbage collector) est automatisée.
 - Ses rôles sont de garantir :
 - L'allocation de la mémoire pour un objet lors de sa création
 - La libération de toute place allouée directement ou indirectement par un objet, lors de sa destruction.
- ➔ Ce phénomène ralentit parfois le fonctionnement de java.

Gestion de la sécurité

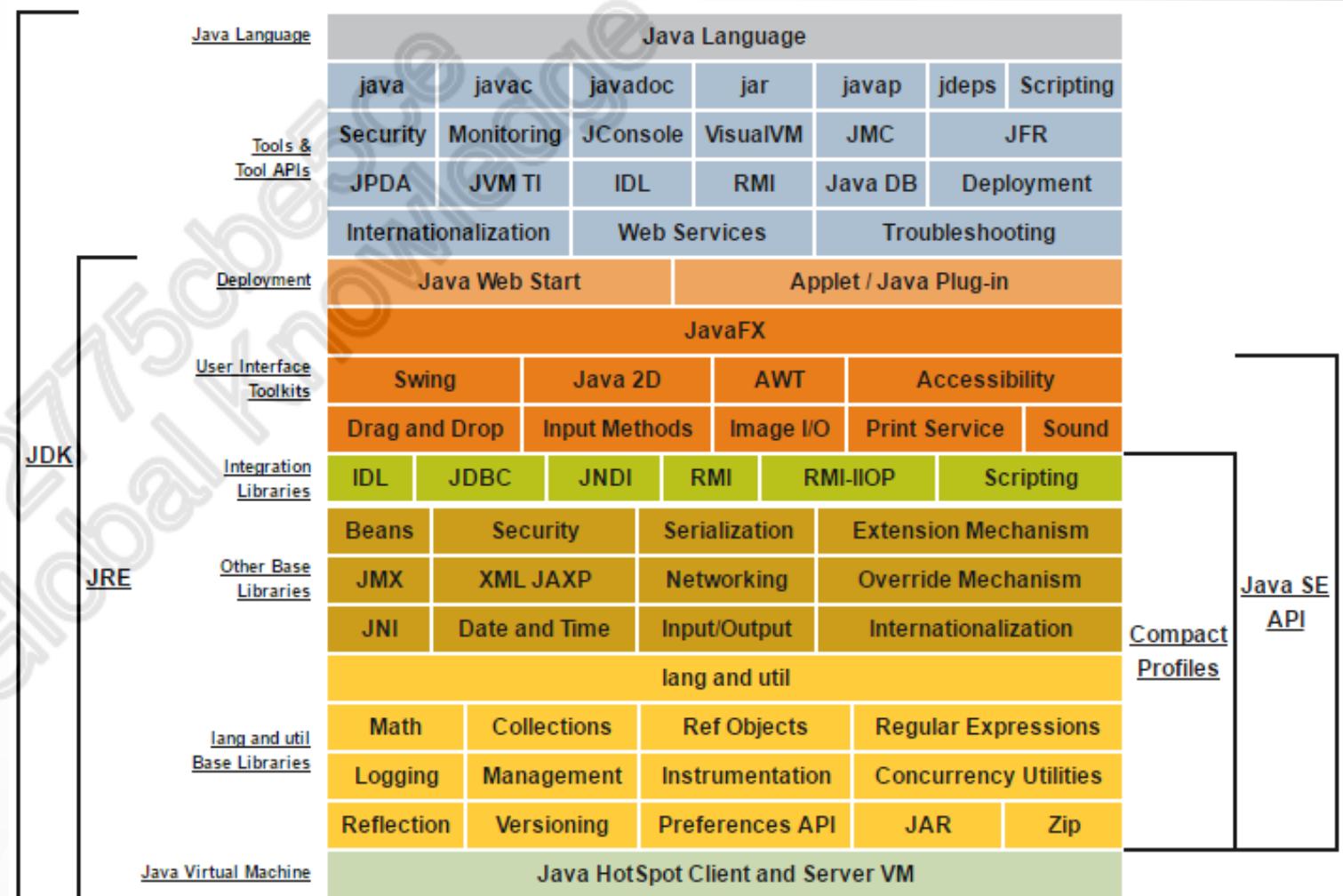
- La sécurité fait partie intégrante du système d'exécution et du compilateur.
 - Un programme Java « planté » ne menace pas le système d'exploitation, car il n'a pas directement accès à la mémoire.
- Sur le Web, l'accès au disque dur est également réglementé (pour les applets). (Principe de SandBox)
- Toute fenêtre créée par un programme java est clairement identifiée comme étant une fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe.



Possibilité de signer les Applet
Possibilité de définir une politique de sécurité

La plateforme Java SE

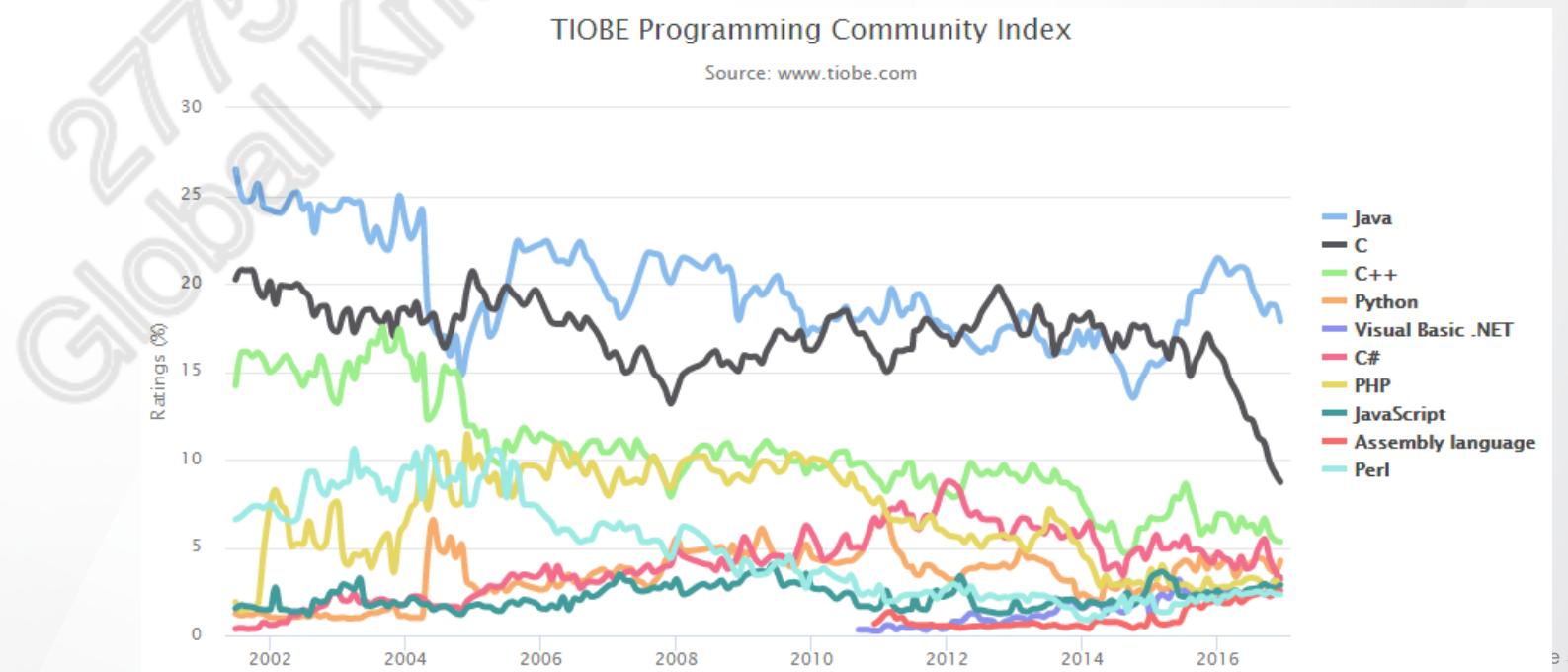
- Ce diagramme conceptuel illustre les technologies de composants Java, *ici la version 8 de Java SE*
- Cet ensemble permet de voir que Java n'est pas qu'un langage
- Site d'Oracle :
<https://docs.oracle.com/en/java/javase/12/>



Pour conclure

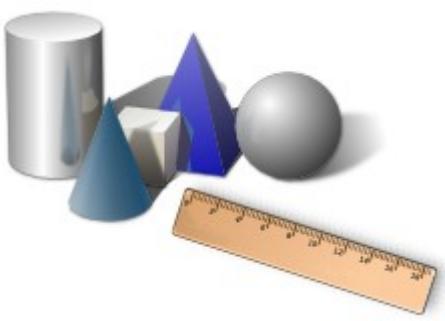
- Java compte aujourd’hui plus de 13 millions de développeurs et est déployé sur des milliards d’appareils électroniques.
- Il reste le langage préféré des développeurs, le plus utilisé à travers le monde, et le plus enseigné dans les écoles et universités. La plate-forme est soutenue par Oracle en premier lieu, mais aussi par des sociétés majeures telles que Google, IBM, Intel ou Red Hat.

(Source *le journal du net - 2014*)



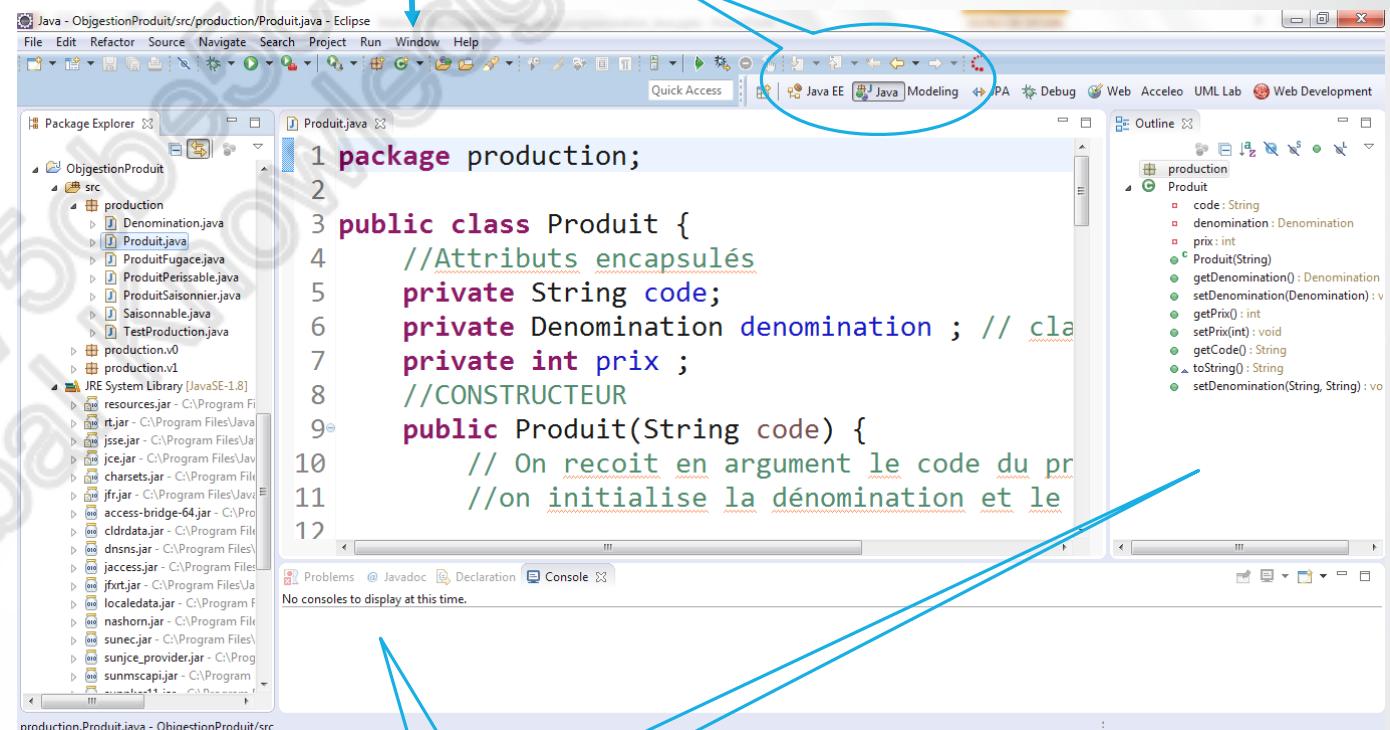
Pour conclure

- Java bénéficie également d'un écosystème gratuit et open source extrêmement développé, sans équivalent ailleurs.
- Java reste une plate-forme stable, mature et pérenne.
- La version 8 apporte à Java ce qui lui manquait pour traiter les problématiques des applications d'entreprise des années qui viennent : notamment le traitement massivement parallèle de très grands volumes de données.
- Java est sans aucun doute un des meilleurs environnements et un des plus pertinents pour ces applications, maintenant et pour les dix prochaines années, et peut être plus.



Install Lab

- Tour d'horizon d'Eclipse
 - Installation du JDK (version stable)
 - Paramétrage Eclipse
 - Quelques astuces :
 - Création Projet
 - Création de classe
 - Auto complétion
 - Génération des sources (Getter / setter)



Vues

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Généralité : Les commentaires

- /* commentaire sur une ou plusieurs lignes */
 - **Identiques à ceux existant dans le langage C**
- // commentaire de fin de ligne
 - **Identiques à ceux existant en C++**
- /** commentaire d'explication */
 - **Les commentaires d'explication se placent généralement juste avant une déclaration (d'attribut ou de méthode)**
 - **Ils sont récupérés par l'utilitaire javadoc et inclus dans la documentation ainsi générée.**

Généralité

- Les expressions
 - Une expression se termine toujours par le symbole « ; » .
- Les blocs d'instructions
 - Un bloc permet de regrouper plusieurs instructions, il se déclare par « { » et se termine par « } » .
 - **Toute variable déclarée dans un bloc n'est connue que de ce bloc.**
- Les identificateurs
 - Attention, Java différencie majuscules et minuscules :
 - « Exemple » et « exemple » sont pour lui deux identificateurs différents.
- Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.

Généralité : Les annotations

- Les annotations, aussi appelées « méta-données », permettent de marquer différents éléments du langage Java avec des attributs particuliers, dans le but d'automatiser certains traitements.
 - Java a toujours proposé une forme ou une autre de méta programmation
 - Dès l'origine, l'outil "javadoc" permettait d'exploiter automatiquement des méta-données à but documentaire

Méthode
nommée *foo*

```
/**  
 * Méthode inutile  
 * @param param Un paramètre (non utilisé)  
 * @return Une valeur fixe : "foo"  
 * @throws Exception N'arrive jamais (promis!)  
  
public String foo(String param) throws Exception {  
    return "foo";  
}
```

Généralité : Les annotations

- Reconnaissant le besoin d'un système de méta-programmation plus robuste et plus flexible, Java 5.0 introduit les Annotations
- **Java 5** propose quelques annotations et « méta-annotations » standards. Il nous donne également la possibilité de créer les nôtres.
- Exemple :
 - Une annotation indiquant qu'un des paramètres transmis à une méthode ne doit pas avoir la valeur « null » :
« @NotNull (param='xx', message='le paramètre xx doit être renseigné') ».
 - Des outils de gestion des annotations peuvent ensuite générer des lignes de code supplémentaires, garantissant l'application de cette contrainte.

Généralité : Les annotations

- S'appliquent à :
 - Des méthodes
 - Des classes
 - Des attributs
- Sont optionnelles
- Peuvent contenir des couples clé/valeur

```
@Deprecated  
public class Pojo {  
  
    @Deprecated  
    private int foo;  
  
    @Deprecated  
    public Pojo() {  
  
        @Deprecated  
        int localVar = 0;  
  
    }  
  
    @Deprecated  
    public int getFoo() {  
        return foo;  
    }  
  
    public void setFoo(@Deprecated int foo) {  
        this.foo = foo;  
    }  
}
```

Généralité : Les annotations standards

- Disponibles dans `java.lang`
- **@Overrides**
 - Lance une erreur si la méthode annotée ne redéfinit pas celle de sa super-classe.
 - Cette annotation ne peut porter que sur une méthode.

```
// L'annotation commence par une majuscule
@Override
public boolean equals(Object o) { ... }
```

Généralité : Les annotations standards

➤ **@SuppressWarnings**

- « **@SuppressWarnings** » demande au compilateur de ne pas afficher certains warnings, dont le nom est passé en paramètre.

```
@SuppressWarnings("deprecation")
public class VieilleClasse { ... }
```

- Le principal intérêt de cette annotation est de pouvoir cacher des Warnings sur des parties anciennes de code, sans pour autant les cacher dans toute l'application.

Généralité : Les annotations standards

➤ @SuppressWarnings

- Exemple de saisie de code sous Eclipse,
 - Cet outil propose un Warning dans la marge.
 - On peut sélectionner le Warning, obtenir la liste des possibilités de corrections et choisir de faire disparaître ce Warning.

The screenshot shows a Java code editor in Eclipse. A tooltip is open over the line of code:

```
private Adresse adresse;
```

The tooltip contains several options for the selected warning:

- Remove 'adresse', keep side-effect assignments
- Create getter and setter for 'adresse'
- Rename in file (Ctrl+2, R)
- Rename in workspace (Alt+Shift+R)
- @Add @SuppressWarnings('unused') to 'adresse'

The tooltip also shows a preview of the code with the annotation added:

```
private String nom;
@SuppressWarnings("unused")
private Adresse adresse;
```

At the bottom right of the tooltip, there is a note: "Press 'Tab' from proposal table or click for fo".

Généralité : Les annotations standards

➤ @Deprecated

- « **@Deprecated** » signale que l'élément marqué ne devrait plus être utilisé.
- Redéfinit le tag Javadoc
- Lance un Warning quand le membre annoté est utilisé.

```
public class Exemple {  
    // Le tag javadoc  
    /** @deprecated Cette méthode est remplacée aujourd'hui par ...  
     */  
    // L'annotation commence par une majuscule  
    @Deprecated  
    public void laMethodeObsolète() { ... }  
}
```

Objectifs du chapitre

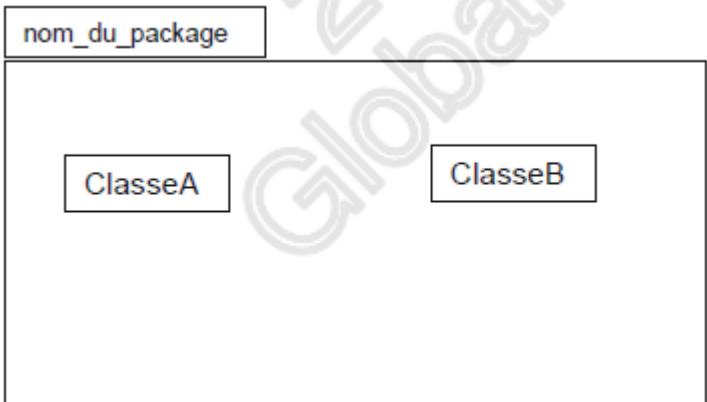
- Généralité du langage JAVA
- **Packages**
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Packages : intérêt

- Un package permet de **regrouper** dans un répertoire des classes et des interfaces concernant un même domaine.
- Il n'existe qu'**un seul niveau** de regroupement.
- On peut définir un ‘sous-package’, via la notation pointée, cependant il sera reconnu comme un package à part entière.
- Un package **influe sur la visibilité** des définitions qu'il contient : par défaut, toutes les classes et interfaces d'un même package peuvent se référencer les unes les autres.

Packages : intérêt

- Java propose différents packages prédéfinis.
Un petit aperçu →
- La représentation UML du package est la suivante



- ▷ java.applet
- ▷ java.awt
- ▷ java.awt.color
- ▷ java.awt.datatransfer
- ▷ java.awt.dnd
- ▷ java.awt.dnd.peer
- ▷ java.awt.event
- ▷ java.awt.font
- ▷ java.awt.geom
- ▷ java.awt.im
- ▷ java.awt.im.spi
- ▷ java.awt.image
- ▷ java.awt.image.renderable
- ▷ java.awt.peer
- ▷ java.awt.print
- ▷ java.beans
- ▷ java.beans.beancontext
- ▷ java.io
- ▷ java.lang
- ▷ java.lang.annotation
- ▷ java.lang.instrument
- ▷ java.lang.management
- ▷ java.lang.ref
- ▷ java.lang.reflect
- ▷ java.math
- ▷ java.net

Packages : Déclaration

- Chaque fichier source java doit contenir **en première instruction** la déclaration du package auquel il appartient.
- Le package sera créé s'il n'existe pas.
- Par convention un nom de package est en **minuscules**, et ne doit pas contenir de caractère spécial (espace, *, /, etc.).

```
package nom_du_package ;  
  
package nom_du_package.nomsouspackage ;
```

Packages : Importation

➤ Utilité :

- Quand une classe doit utiliser une classe définie dans un autre package, elle est obligée de donner le chemin complet de l'endroit où se trouve cette autre classe c'est à dire le package auquel elle appartient.
- Il existe trois manières de faire :
 - Sans importation préalable, indication du package élément par élément (vite illisible).
 - Importation préalable de l'ensemble d'un package (moins performant).
 - Importation préalable de chaque élément utilisé d'un package (le mieux !).

```
package nom_du_package1 ;  
class ClasseA {...}  
class ClasseB {...}  
// Exemple sans importation préalable  
package nom_du_package2 ;  
class ClasseC { nom_du_package1.ClasseA ...  
               nom_du_package1.ClasseB ...  
}  
// Exemple avec importation totale  
package nom_du_package2 ;  
import nom_du_package1.* ;  
class ClasseC { ClasseA ...  
               ClasseB ...  
}  
// Exemple avec importation de chaque élément utilisé  
package nom_du_package2 ;  
import nom_du_package1.ClasseA ;  
import nom_du_package1.ClasseB ;  
class ClasseC { ClasseA ...  
               ClasseB ...  
}
```

Packages : Importation

➤ Les imports statiques

- Les imports statiques permettent de ne pas écrire le nom de la classe devant les accès aux membres « de classe » (attributs ou méthodes « static »).
- Cette possibilité est apparue avec la version 5 de Java, elle permet de rendre le code plus lisible.

```
package demoCours;  
import static java.lang.System.out;  
  
public class TestHeritage  
{  
  
    public static void main(String[] args)  
{  
  
        // Déclaration et int d'un Client  
        Client unClient = new Client();  
  
        // Manipulation des attributs  
        unClient.setNom("Ets CROY");  
        unClient.setNumero(100);  
  
        // Affichage à la console système  
        out.println(unClient.retourneInfo());  
    }  
}
```

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- **Classes**
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Classes : Déclaration

- Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelée classe.
- La classe décrit le domaine de définition d'un ensemble d'objets
- La déclaration s'effectue par le mot clé « **class** » suivi du nom de la classe.
- Par convention un nom de classe débute par une **majuscule**, et ne doit pas contenir de caractère spécial.
 - Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

Classes : Déclaration

➤ **Syntaxe :**

```
[Modificateurs] class NomClasse [extends NomSuperClasse] [implements Interface1,  
Interface2] {...}
```

- Les accolades { } délimitent le contenu de la classe (càd de la définition).
- « **extends** » pour hériter d'une classe.
- « **implements** » pour implémenter des interfaces.

```
class ClasseA {...}  
// Exemple avec héritage  
class ClasseB extends ClasseA{...}  
// Exemple avec modificateurs  
public abstract class ClasseC {...}
```

Classes : Déclaration - Modificateurs

- La visibilité d'une classe est indiquée par les modificateurs.
- Par défaut une classe est visible à l'intérieur d'un même package.
 - **public** : la classe est visible pour les autres packages.
 - **final** : la classe ne pourra pas être sur-classe
 - **abstract** : la classe ne pourra pas être instanciée, elle sert de référence à des sous-classes.
- **Exemple :**
 - Déclaration d'une classe abstraite, dans un sous-package, définissant un attribut adresse, appartenant à un autre package, d'où l'importation de cette définition.

```
package demoCours.syntaxe;  
  
import demoCours.Adresse;  
  
public abstract class Personne  
{  
    private int numero;  
    private String nom;  
    private Adresse adresse;  
  
    public Personne()  
    {  
        .../...  
    }  
    .../...  
}
```

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- **Méthodes**
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Méthodes

➤ Intérêt :

- Une méthode **définit un traitement à effectuer, elle a un nom.**
- Une méthode peut recevoir des arguments (ou paramètres : définis dans les parenthèses après le **nom de méthode**, séparés par des virgules) pour faire varier le traitement à effectuer (on indique le *type* et le *nom* de l'argument)
- Un méthode peut retourner une information, l'information est alors typée (à définir devant le nom de la méthode), s'il n'y a pas d'info : on indique *void*

Remarque : Un constructeur est une méthode particulière de la classe, car il n'indique aucune information de retour, le nom du constructeur est le même que celui de la classe. Il permet notamment l'initialisation des attributs.

➤ Syntaxe :

```
[Modificateurs] typeDeRetour nomMethode ([typeArg1 nomArg1, typeArg2 nomArg2]) {...}  
protected String monAction ( int n_Parm1, String chaine ) {...}  
public void monAction ( String n_Parm1 ) {...}
```

Méthodes : Déclaration

- La déclaration d'une méthode peut commencer par un modificateur, qui indique le degré de visibilité de la méthode.
- Entre parenthèses on indique les paramètres reçus (séparés par des virgules).
- Par convention un nom de méthode débute par une **minuscule**, et ne doit pas contenir de caractère spécial. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

```
public class ClasseA {  
    // Constructeur  
    public ClasseA() {...}  
    // Autres méthodes  
    void neRetourneRien() {...}  
  
    public boolean retourneBooleen() {  
        return false;  
    }  
  
    protected void recoitBooleen (boolean a)  
    {...}  
    ... / ...
```

Méthodes : Modificateurs

- La visibilité d'une méthode est déterminée par les modificateurs.
- Par défaut une méthode est visible à l'ensemble du package.
 - **private** : méthode visible par la classe uniquement.
 - **public** : méthode visible par tous les packages.
 - **protected** : méthode visible par les sous-classes et l'ensemble du package.
 - **static** la méthode pourra être invoquée pour la classe elle-même (sans avoir besoin d'instancier un objet)

Méthodes : Types de Méthodes

- On distingue trois types de méthodes :
 - Une fonction
 - Une procédure
 - Un constructeur
 - **Fonction :**
 - Une « fonction » renvoie toujours une valeur dont elle doit indiquer le type.
 - Son code contiendra le mot clé **return**
- Définition d'une méthode : boolean controlSaisie(int arg) { ... return true ; }*
- **Son appel** peut figurer dans une expression ou être l'affectation d'une variable
 - Expression : *if (controlSaisie(saisie)) {} //saisie est une variable int*
 - Affectation : *boolean test = controlSaisie(saisie) ;*

Méthodes : Types de Modificateurs

➤ Procédure :

- Une « **procédure** » ne retourne aucune valeur.
- Sa déclaration contiendra le mot clé **void**.
- Son appel équivaut à celui d'une instruction.

➤ Exemple :

- Définition de la méthode : **void afficheListe(int entier) {...}**
- Appel de la méthode : **afficheListe (x) ; // x est une variable int**
 afficheListe(20) ;

Méthodes : Types de Modificateurs

➤ Constructeur :

- Un « **constructeur** » porte le même nom que la classe. Le nom débute donc par une **majuscule**.
- Il ne retourne rien, mais cette fois il ne faut pas noter le mot clé *void*.
- L'appel du constructeur ne peut se faire que via l'opérateur « **new** ». Cette appel permet **d'instancier** l'objet. (*new Client(); new Voiture("Audi");*)
- Lors d'un héritage, on ne pourra pas hériter du constructeur, il faudra définir cette méthode pour chaque classe.
- Lors d'une surcharge (polymorphisme à l'intérieur d'une même classe), la réutilisation du code d'un constructeur déjà défini s'effectuera en utilisant la pseudo-variable **this**, car nommer le constructeur ne peut se faire que via *new*.

Méthodes : Exemple

➤ Exemple de constructeurs et de méthodes

```
package demoCours;
public class Client {
    //Attributs
    ...
    // Constructeurs - Initialise les attributs
    public Client() {
        this.numero = 0;
        // Création de l'objet adresse
        this.adresse = new Adresse();
    }
    public Client(int numero) {
        // Appel du constructeur sans argument
        this();
        //init du numéro
        this.numero = numero;
    }
    // Méthode type fonction qui retourne un type Adresse
    public Adresse getAdresse() {
        return adresse;
    }
    // Méthode type procédure qui ne retourne rien mais reçoit un argument
    public void setAdresse(Adresse adresse) {
        this.adresse = adresse;
    }
    // Méthode type fonction qui retourne un type Chaine et qui reçoit un argument
    // qui appelle d'autres méthodes
    public String retourneInfo(String info){
        return info + " : " + this.retourneInfo();
    }
}
```

Deux constructeurs
= Surcharge

Le constructeur avec
argument appelle le
constructeur sans
argument
Pas d'instanciation

Méthodes : arguments

- Il est désormais possible de passer à une méthode un nombre non défini d'arguments.
- Ces paramètres sont traités comme un tableau et doivent donc tous être du même type.
- La seule limite est qu'on ne peut pas, dans la liste des éléments fournis, alterner tableau et éléments unitaires.

Méthodes : arguments

```
package demoCours.syntaxe;
public class Calcul {

    public Calcul() {
    }

    public int additionner ( int ... entiers ) {
        //Méthode ayant plusieurs arguments de type int
        int somme = 0 ;
        for (int valeur : entiers)
            somme += valeur ;
        return somme ;
    }
}
```

```
package demoCours.syntaxe;
public class TestCalcul {
    public static void main(String[] args) {
        // Instanciation de l'objet définissant la méthode à arg variables
        Calcul cal = new Calcul();

        //manip de la méthode en passant différents paramètres
        System.out.println("Add de 3 éléments " + cal.additionner ( 1, 2, 3 ) );
        System.out.println("Add de 6 éléments " + cal.additionner ( 1, 2, 3, 4, 5, 6 ) );

        //Déclaration d'un tableau (les tableaux seront vus dans un prochain chapitre)
        int [ ] tableau = { 1, 2, 3, 4 } ;
        System.out.println("Addition via 1 tableau " + cal.additionner ( tableau ) );
    }
}
```

➤ Résultats à la console :

```
Add de 3 éléments 6
Add de 6 éléments 21
Add via 1 tableau 10
```

Méthodes : Transmission de paramètres

- En Java, les arguments sont toujours passés par **valeur** :
 - Le « passage par valeur » : l'appel ne transmet qu'une copie de la valeur. Si le traitement de la méthode appelée change cette valeur, l'originale (la valeur au niveau de l'appelant) n'est pas modifiée.
 - Pour les paramètres de type objets, l'adresse de l'objet est passée par valeur. Si cette adresse est remplacée dans la méthode par celle d'un autre objet, l'objet d'origine restera intègre après l'exécution de la méthode.

Méthodes : Polymorphisme : Surcharge et Redéfinition

- Un nom de méthode n'est pas unique.
- Une même classe peut avoir plusieurs constructeurs et plusieurs fois le même nom de méthode.
- La combinaison
 - Du nom de la méthode
 - Du nombre de paramètres reçus
 - Et de leurs types respectifs

...constitue la **signature** de la méthode et doit être unique au sein d'une même classe
- Quand une classe contient des méthodes ayant le même nom, on parle de **surcharge**.
- On parle de **redéfinition**, quand on redéfinit une méthode de la classe-mère dans la classe fille. On reprend dans la fille exactement la même signature que celle de la mère.

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- **Attributs – variables**
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itératives
- Rupture et continuation
- Structure d'une application

Attributs - variables

➤ Déclaration

- La déclaration d'une donnée (d'une variable) en Java **doit toujours** être précédée du type de donnée que l'on veut manipuler.
- Par convention un nom de donnée débute par une **minuscule**

➤ Syntaxe

[Modificateurs] *type de la variable* **nomVariable** [=valeur];

➤ Exemples, ici 3 types de données : int, String et boolean

```
int monEntier ;  
String prenom ;  
boolean ok ;
```

Noter la différence
Majuscule/Minuscule

Attributs - variables

➤ Attribut d'instance :

- Une classe peut posséder plusieurs attributs, dits propriétés, variables d'instance, champs ou fields en anglais.
- Un attribut d'instance (ou « variable d'instance ») est une donnée définie en dehors de toute méthode. Chaque objet, instance de cette classe, possèdera sa propre valeur : par exemple, le nom d'une personne.
- C'est une variable globale à l'ensemble des méthodes de la classe : toutes peuvent y accéder.

```
package modeles;  
public class Personne {  
    String prenom;  
    int age;  
}
```

Attributs - variables

➤ Attribut de classe :

- Une classe peut posséder un attribut qui sera partagé par toutes ses instances (par exemple le nombre d'instances créées).

```
private static int compt ;
```

- Une « variable de classe » est créée lors du chargement de la classe en mémoire vive (et non lors de linstanciation d'objets). Elle doit être créée avec le modificateur « **static** ».
- Une donnée définie avec le modificateur « **final** » est une constante. Elle mérite donc d'être également *static*. Le nom des constantes est en majuscule.

```
private static final int AGE_MIN = 25;
```

Attributs - variables

➤ Variable de travail :

- Une méthode peut déclarer une variable, dans ce cas elle reste locale à la méthode et n'est pas visible par d'autres méthodes.

➤ Déclaration

- La déclaration d'une donnée doit toujours être précédée du type de donnée que l'on veut manipuler.
- Dans une méthode on peut initialiser au moment de la déclaration la valeur de la variable
- On peut aussi définir plusieurs variables de même type ;

```
public void methode(){  
    int a, b, c;  
    int x = 1;  
    boolean ok = false;  
}
```

Attributs – variables : exemples

➤ Exemple :

```
// Déclaration de variable
int nomVarEntier ;
String nomChaine ;
Adresse adresse ;

// Déclaration de plusieurs variables
int nomVarEntier1, nomVarEntier2, nomVarEntier3 ;
String nomChaine1, chain2, chain3;
Adresse adr1, adr2, adresse3;

// Déclaration et initialisation
int nomVarEntiern = 5 ;
String maChaine = "Bienvenue chez GK";
Adresse adress = new Adresse(); //init = appel constructeur
```

Attributs - variables : modificateurs

- La visibilité des attributs (d'instance ou de classe) est déterminée par les modificateurs.
- Par défaut un attribut est visible à l'ensemble du package.
 - private : attribut visible par la classe uniquement.
 - public : attribut visible par tous les packages.
 - protected : attribut visible par les sous-classes et le même package.
 - final : déclaration d'une constante.
 - static : attribut partagé par toutes les instances.
- Seul le modificateur *final* est possible avec une variable de travail.

Attributs – variables : les types primitifs

- Java a conservé trois catégories de types dits « primitifs » (qui ne sont pas des classes) :
 - Les booléens, les nombres et les caractères

Attention :

Java initialise par défaut les attributs (données déclarées en dehors de toute méthode), mais **pas** les variables locales (données déclarées dans une méthode).

Type	Taille	Valeur par défaut
boolean	8	false
char	16	'0x00' (norme Unicode)
byte	8	0
short	16	0
int	32	0
long	64	0
float	32	0.0F
double	64	0.0D

Attributs - variables : exemple

```
package modeles;
public class Personne {
    //Attributs d'instance = propriétés GLOBAL
    private String prenom;
    private int age;
    private boolean majeur ;
    //Constante
    private static final int AGE_MIN = 25;
    //constructeurs
    //BUT : initialiser les variables d'instance
    public Personne() {
        // la variable age est initialisée par défaut à 0
        // car c'est un type primitif
        // modification de age par la constante
        age = AGE_MIN ;
        // initialisation autres variable
        prenom = "inconnu" ; //Objet
        _majeur = true ;      //primitif
    } .../...
```

```
//autre constructeur Polymorphisme
public Personne(String chaine) {
    // modification de prenom par le paramètre chaine
    prenom = chaine ;
    // les variables age et majeur sont initialisées
    // par défaut respectivement à 0 et false
}
// Autre méthode Reçoit un argument de type int
public void uneMethode(int a){
    // déclaration variables de travail LOCAL
    int x, y ;
    x = AGE_MIN ;
    y = x ;
}           // fin méthode
}// fin classe
```

Attributs – variables : les valeurs numériques

- Les données numériques sont représentées en base 10, 16 ou 8, avec ou sans point décimal.
- Base 16 : précédés des symboles 0x ou 0X.
- La valeur des entiers de plus de 32 bits (long) finissent par le symbole L ou l.
- Les valeurs des réels peuvent :
 - être en simple précision
 - terminer par d ou D pour un double
 - terminer par f ou F pour un float

Attributs – variables

➤ Les booléens

- Les valeurs booléennes « *true* » et « *false* » ne sont pas des valeurs numériques

➤ Les caractères

- La représentation interne des caractères en java est de l'unicode (sur 16 bits).
- Un caractère est placé entre apostrophes.
- Dans les deux cas, certains caractères ont une signification particulière

Ligne suivante	\n
Paragraphe suivant	\r
Tabulation	\t
Apostrophe	\'
Guillemet	\“
Backslash	\\\

Attributs – variables :

➤ Exemples de manipulations de variables

```
package demoCours.syntaxe;
public class Declaration
{
    static int un, deux;

    public static void main(String[] args)
    {
        Boolean ok = true;
        char monCaractere;
        double monDouble;
        int resultat;

        // Manip variables globales
        un = 1;
        deux = 2;

        // Manip variables locales
        resultat = un + deux;
        monCaractere = 'A';
        monDouble = 100;

        System.out.println("int " + resultat + " valeur du booléen : " + ok);
        System.out.println("Char " + monCaractere + " double " + monDouble );
    }
}
```

Résultats console :

```
int 3 valeur du booléen : true
Char A double 100.0
```

Attributs – variables :

➤ Exemples de manipulations de variables et de fonctions

```
package demoCours_Syntaxe;
public class Calcul {
    public static void main(String[] args) {
        System.out.println("début du traitement principal");
        //déclaration et init des variables
        int a = 10;
        int b = 5;
        //déclaration d'une variable pour récupérer le résultat de la fonction
        double c;
        System.out.println("Addition de " + a + " + " + b);
        c = addition(a, b);
        System.out.println(c);
        //manipulation de variable;
        a = a * 3 ;
        c = b; //conversion implicite de int en double

        System.out.println("Addition sans récupération de résultat" );
        c = addition(a, b);

        System.out.println("Division de " + a + " + " + b + " = " + division(a, b));
        System.out.println("End");

    }
    // Méthode addition
    public static double addition(int a, int b) { return a + b; }
    // Méthode division sans gestion de la div par 0 !
    public static double division(int a, int b) {return a / b; }
} //fin classe
```

Résultats console :

```
début du traitement principal
Addition de 10 + 5
15.0
Addition sans récupération de résultat
Division de 30 + 5 = 6.0
End
```

Attributs – variables : type enum

- Il est régulièrement utile de pouvoir définir un ensemble fini de valeurs pour des données. Par exemple : {nord, sud, est, ouest}
- Jusqu'à présent, la façon la plus pratique était de déclarer des constantes dans une classe :

```
public class PointCard  
{  
    public static final int NORD = 0 ;  
    public static final int SUD = 1 ;  
    public static final int EST = 2 ;  
    public static final int OUEST = 3 ;  
}
```

- L'inconvénient de ce procédé est qu'il n'offre pas de contrôle de type possible des valeurs – surtout si les constantes ne sont pas utilisées.

Attributs – variables : type enum

- **Syntaxe** [Modificateurs] **enum** IdentType { identValeur1, identValeur2, ... };
- Lorsqu'il rencontre le mot-clé « enum », le compilateur crée automatiquement une classe :
 - Cette classe hérite de java.lang.Enum
 - Elle est déclarée « final » pour empêcher toute modification par héritage
 - Elle implémente les interfaces Comparable et Serializable
 - Elle ne possède pas de constructeur public
 - Chacune des valeurs de l'énumération est une instance de cette classe, déclarée « public static final »
 - Les méthodes « *toString()* », « *equals()* » et « *compareTo()* » (notamment) sont redéfinies
 - Une méthode « *values()* » renvoie un tableau contenant toutes les valeurs possibles

Attributs – variables : type enum

- Exemples de manipulation du type enum :

Résultats console :

```
roi avant as -1
roi après valet 3
0
direction EST
Vive les Bretons
```

```
package demoCours.syntaxe;

public class Declaration
{
    // Déclarations
    enum PointCard {NORD, SUD, EST, OUEST};
    public enum CarteAJouer {VALET, CAVALIER, DAME, ROI, AS};

    public static void main(String[] args)
    {
        // Manipulations
        CarteAJouer as = CarteAJouer.AS;
        CarteAJouer roi = CarteAJouer.ROI;
        CarteAJouer valet = CarteAJouer.VALET;

        if (roi.compareTo(as) < 0)
            {System.out.println("roi avant as " + roi.compareTo(as));}
        if (roi.compareTo(valet) > 0)
            {System.out.println("roi après valet " +
                roi.compareTo(valet));}
        System.out.println(roi.compareTo(roi));

        PointCard direction = PointCard.EST;
        if (!direction.equals(PointCard.SUD))
            {System.out.println("direction " + direction);}

        switch (direction)
        {
            case OUEST:
                System.out.println("les chapeaux ronds");
                break;
            default:
                System.out.println("Vive les Bretons");
                break;
        }
    }
}
```

Attributs – variables

➤ Pseudo-variables :

- **super** : ce mot clé provoque une recherche dans la super–classe :
 - Déclenchement d'une méthode définie dans la classe mère :
 - `super.methode(...);`
 - Déclenchement d'un constructeur de la classe mère :
 - `super(...);`
- **this** : ce mot clé référence l'objet dynamiquement actif au moment de l'exécution du code défini dans la classe :
 - `this.attribut ;` // référence 1 attribut d'instance
 - `this.méthode() ;` // appel la méthode nommée méthode (this non obligatoire)
 - `this() ;` // Utilisable dans un constructeur pour faire appel au constructeur

Attributs - variables : exemple

```
package modeles;
public class Personne {
    //Attributs d'instance = propriétés
    private String prenom;
    private int age;
    private boolean majeur ;
    //Constante
    private static final int AGE_MIN = 25;
    //constructeurs
    //BUT : initialiser les variables d'instance
    public Personne() {
        super(); // Appel du constructeur de la classe mère
        // la variable age est initialisée par défaut à 0
        // car c'est un type primitif
        // initialisation autres variable
        prenom = "inconnu" ; //Objet
        _majeur = true ;      //primitif
    } .../...
```

```
public Personne(String prenom) {
    // Appel du constructeur de cette classe
    this();
    // modification de prenom par le paramètre
    this.prenom = prenom;
    // modification de age par la constante
    age = AGE_MIN ;
}
// Reçoit un argument de type int
//effectue des opérations arithmétiques
public void uneMethode(int a, int b){
    int x, y, z, w ;
    x = a + b ;
    y = x - b ;
    z = b * a ;
    w = z * a ;
}           // fin méthode
}// fin classe
```

Attributs – variables

- **La valeur « null » :**
 - Le mot clé « **null** » permet d'affecter n'importe quelle variable de type objet.
 - Il permet d'initialiser certaines variables avec une valeur par défaut.
 - Certaines méthodes retournent comme valeur « **null** » pour indiquer qu'elles n'ont pas été capables d'allouer un objet spécifique.

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- **Objets**
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Objets

- Les objets manipulés dans les traitements doivent être déclarés dans des variables.
- L'instanciation d'un objet s'effectue avec l'opérateur ***new***, puis on utilise le nom du constructeur de la classe que l'on veut instancier.

```
// Déclaration d'une référence  
ClasseA objetA;  
Personne p1, p2 ;  
// Instanciation d'un objet  
objetA = new ClasseA() ;  
// Déclaration et instantiation  
ClasseA autreObjetA = new ClasseA() ;  
Personne p = new Personne() ;
```

- La déclaration d'une référence à un objet n'alloue aucune zone mémoire à l'objet lui-même. Cette allocation est faite à l'appel d'un constructeur par *new*.
- Si la ClasseA contient un attribut statique (mot-clé *static*), l'allocation mémoire pour cette référence sera unique pour tous les objets de ClasseA.

Objets

La définition = la classe

```
package exemple ;  
public class Personne {  
    //attributs  
    int nbAdresse ;  
    static int compteur ;  
  
    //constructeur  
    public Personne(){  
        nbAdresse =1 ;  
        compteur = compteur +1 ;  
    }  
}
```

Variable de classe

L'exécutable :

```
package exemple ;  
public class TestPersonne {  
    public static void main(String[] args) {  
  
        // Déclaration de 2 références  
        Personne p1, p2 ;  
        //Instanciation d'objets  
        p1 = new Personne() ;  
        p2 = new Personne() ;  
    }  
}
```

Référence mémoire de
TestPersonne

p1 p2



A la première instantiation, la variable de classe :
compteur =1

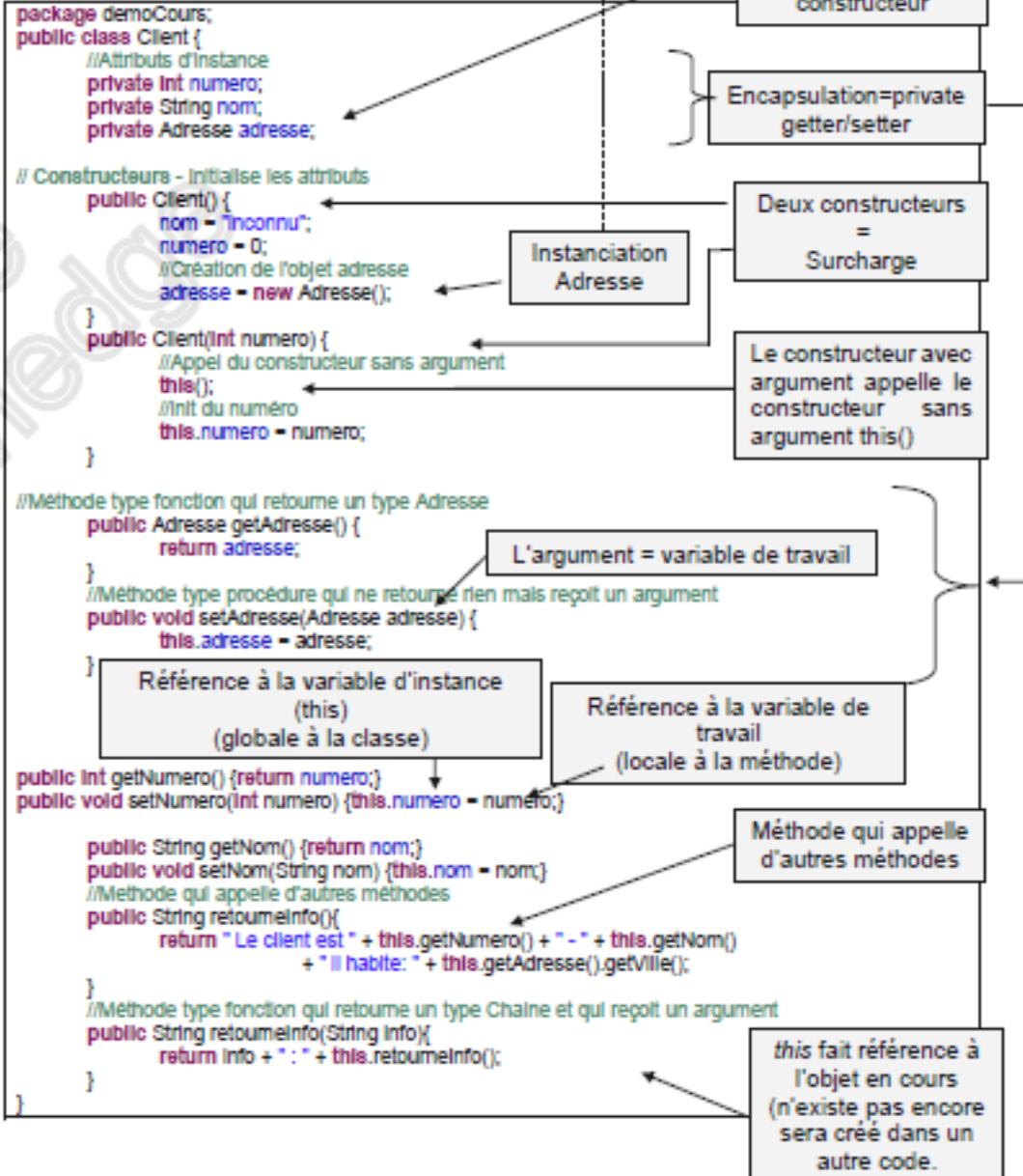
A la deuxième instantiation
compteur= 2

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- **Exemple**
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Exemple :

- La définition d'un concept métier le Client, c'est-à-dire la **classe Client**.
- La classe Client est composée d'une autre classe métier, définie sous le nom **Adresse** et d'une classe String du monde Java. Ce sont des liens de composition.



Exemple

- La définition d'une classe fille.
- La classe ClientSociete "est une sorte de" Client, aussi elle hérite de la classe mère Client (*extends*). C'est un lien de généralisation/spécialisation

```
package demoCours;

public class ClientSociete extends Client {
    //Attributs spécifiques aux Sociétés
    private Int nbEmployes;
    private String siren;

    public ClientSociete() {
        // On inhérite pas des constructeurs - Il faut les définir
        // On initialise les attributs de la classe filie
        // Appel du constructeur de la classe mère est implicite
        nbEmployes = 0;
        siren="numero de siren";
    }
    public ClientSociete(Int numero) {
        super(numero);
    }

    public Int getNbEmployes() {return nbEmployes;}
    public void setNbEmployes(Int nbEmployes) {this.nbEmployes = nbEmployes;}
    public String getSiret() {return siren;}
    public void setSiret(String siren) {this.siren = siren;}

    @Override
    public String retoumeInfo() {
        // On ajoute au code de la classe mère le spécifique de la fille
        return super.retoumeInfo() + " Société: " + this.getSiret() + " Nb d'employés " +
    this.getNbEmployes();
    }
}
```

Héritage

Définition des deux constructeurs (comme la classe mère)

Appel du constructeur de la mère (super)

Redéfinition Annotation@Override

**Appel de la méthode de la mère (super.retoumeInfo())
Et redéfinition**

Exemple

➤ Les tests de ce concept :

- Définition d'un exécutable TestHeritage (nécessite une classe) :
- On instancie des objets de type Client et de type ClientSociete

Résultats à la console

Le client est 100 - Ets CROY il habite: à renseigner

Le client est 200 - inconnu il habite: à renseigner Societe: numero de siren Nb d'employés 0

Le client est 250 - inconnu il habite: à renseigner Societe: null Nb d'employés 0

```
package demoCours;  
  
public class TestHeritage {  
  
    public static void main(String[] args) {  
  
        // Déclaration et Int d'un Client  
        Client unClient = new Client();  
        //Manipulation des attributs  
        unClient.setNom("Ets CROY");  
        unClient.setNumero(100);  
  
        // Affichage à la console système  
        System.out.println(unClient.retourneInfo());  
  
        // Déclaration des variables de type Societe  
  
        ClientSociete uneSociete, uneAutreSociete;  
  
        // Instanciation  
        uneSociete = new ClientSociete();  
        // Manipulation de l'objet uneSociete  
        uneSociete.setNumero(200);  
  
        // Instanciation  
        uneAutreSociete = new ClientSociete(250);  
  
        // Affichage à la console système  
        System.out.println(uneSociete.retourneInfo());  
        System.out.println(uneAutreSociete.retourneInfo());  
    }  
}
```

Méthode indiquant qu'il s'agit d'une application autonome Java

Pas d'instanciation d'objet System
on n'utilise que la définition de System
Déclaration statique
à cette définition, on manipule la variable out pour laquelle on applique la méthode println

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- **Les opérateurs**
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Les Opérateurs

➤ Les opérateurs arithmétiques

Java utilise les quatre opérateurs arithmétiques de base :

- + addition
 - - soustraction
 - * multiplication
 - / division
- et le modulo
- % modulo (reste de la division entière)

Les Opérateurs

➤ Les opérateurs unaires

Les opérateurs sont dits unaires lorsqu'ils portent sur un seul terme.

- L'inverse : -
- L'incrément : ++
 - La pré-incrémantation si le signe ++ est à gauche de l'élément. (++i)
 - $i = ++j; \rightarrow j = j+1$ puis $i = j;$
 - La post-incrémantation si le signe ++ est à droite. (i++)
 - $i = j++; \rightarrow i = j$ puis $j = j+1;$
- La décrémentation : --
 - La pré-décrémentation si le signe -- est à gauche de l'élément. (--i)
 - La post-décrémentation si le signe -- est à droite. (i--).

Les Opérateurs

➤ Les opérateurs de comparaison

Six opérateurs de comparaison sont utilisables :

- == égal,
- != différent,
- > strictement supérieur,
- >= supérieur ou égal,
- < strictement inférieur,
- <= inférieur ou égal.

➤ Les opérateurs de comparaison peuvent être utilisés dans les instructions de contrôle du langage, mais aussi dans des instructions d'affectation.

➤ Ils **comparent des données de même nature**. Le résultat est une valeur booléenne.

Les Opérateurs

➤ Exemple

```
boolean resultat ;  
resultat = var1 > var2 ;
```

➤ Résultat vaudra :

- *true*, si var1 est strictement supérieur à var2 ,
- *false*, si var1 est inférieur ou égal à var2,

➤ Le signe = (“ égal ”) n'est utilisé que pour l'affectation, la comparaison s'effectue avec le signe ==

Les Opérateurs

➤ L'opérateur de concaténation

- La concaténation de (chaînes de caractères) est effectuée avec le symbole “+”.

```
String ch = "jour";
String resultat = "bon" + ch + "!";
```

➤ Les opérateurs logiques :

- ! Non pour la négation,
- & Et,
- | Ou,
- ^ Ou exclusif,
- && Et évalué, pour intersection,
- || Ou évalué, pour la réunion.

```
if (Annuaire.val(s1.code) == null &&
Annuaire.val(s2.code) == null) {  
}
```

Les Opérateurs

➤ Les opérateurs associatifs

L'opérateur d'affectation “=” peut se combiner avec des opérateurs arithmétiques ou booléens .

Exemple	Équivalence
$i = 1$	$i = i - 1$
$i += j$	$i = i + j$
$i *= j$	$i = i * j$

➤ L'opérateur conditionnel

Cet opérateur ternaire « ? » teste une expression et renvoie en résultat une autre expression, en fonction du test.

```
A = (b < c) ? (d - 1) : (d + 1) ;
```

Signification

- Si b est plus petit que c alors a vaut $d-1$ Sinon a vaut $d+1$.

Les Opérateurs

➤ L'opérateur **instanceof**

L'opérateur « **instanceof** », permet de déterminer la classe d'appartenance d'un objet.

➤ Cet opérateur attend deux opérandes :

- A gauche : un objet
- A droite : une classe

➤ Il retourne un booléen.

Les Opérateurs

```
package demoCours.syntaxe;

import demoCours.Client;
import demoCours.ClientSociete;

public class TestOperateurs {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Client monClient = new Client(300);
        ClientSociete maSociete = new ClientSociete();
        String maChaine = "toto";

        if (monClient instanceof Client) {
            System.out.println(monClient.retourneInfo());
        }

        if (maSociete instanceof Client) {
            System.out.println(maSociete.retourneInfo());
        }

        if (maChaine instanceof Client) {
            System.out.println(maSociete.retourneInfo());
        }
    }
}
```

maSociete est une instance de client par héritage

Dans cet exemple il est impossible d'écrire à la console système car il y a automatiquement détection de l'erreur.

Description	Resource	Path	Location	Type
<ul style="list-style-type: none">✖ Errors (1 item)				
<ul style="list-style-type: none">✖ Incompatible conditional operand types String and Client	TestOperateu...	/stageJavaDev/src/...	line 25	Java Problem

Les Opérateurs

➤ Priorité des opérateurs

- La priorité des opérateurs est la suivante : (de la plus grande à la plus petite).
- Les opérateurs de même niveau sont évalués depuis la gauche.

.	[]	()		
++	--	!	~	instance of
*	/	%		
+	-			
<<	>>	>>>		
<	>	<=	>=	
==	!=			
&				
^				
&&				
? :				
=				
:				

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- **Exécution conditionnelle**
- Exécution itérative
- Rupture et continuation
- Structure d'une application

Exécution Conditionnelle : if / else if / else if

➤ Syntaxe

```
if (expression) instruction ;  
if (expression) {instructions ;}  
if (expression) instruction ; else instruction ;  
if (expression) {instructions ;} else {instructions ;}  
  
if (expression) {instructions ;}  
else if (expression) {instructions ;}
```

- L'expression de test doit toujours être une expression booléenne.
- Si l'expression est vraie l'instructions se trouvant derrière les parenthèses est exécutée, il faut mettre des { } si l'on souhaite exécuter plusieurs instructions.
- Si l'expression est fausse les instructions suivant le « else » sont exécutées. Le « else » est facultatif.

Exécution Conditionnelle : if / else if / else if

➤ Exemple

```
package demoCours.syntaxe;
import demoCours.Client;
public class TestOperateurs {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Client monClient= new Client(300);
        String maChaine="toto";

        int un = 1;
        int deux = 2;

        if (monClient instanceof Client) {
            System.out.println(monClient.retoumeInfo());
        }

        if (un == deux) System.out.println("impossible");
        else if (un < deux) {
            System.out.println("ok");
        }
    }
}
```

Le client est 300 - inconnu il habite : à renseigner
ok

Exécution Conditionnelle : switch

➤ Syntaxe

```
switch (expression)
{
    case valeur1 : instructions ;
    case valeur2 : instructions ;
    ...
    default : instructions ;
}
```

- L'expression est évaluée, on compare la valeur rentrée avec celles de chaque valeur de `case`. S'il y a égalité, les instructions du `case` sont effectuées.
- S'il n'y a pas d'égalité, les instructions du `default` sont exécutées. Le `default` est facultatif.
- Si l'on veut isoler chaque `case`, il faut terminer les instructions par un `break`.

Exécution Conditionnelle : switch

```
switch (un) {  
    case 2:  
        System.out.println("impossible");  
        break;  
    case 3:  
        System.out.println("impossible");  
        break;  
    default:  
        System.out.println("défaut");  
        break;  
}  
  
enum PointCard {NORD, SUD, EST, OUEST};  
PointCard direction = PointCard.EST;  
switch (direction) {  
    case OUEST:  
        System.out.println("En route vers la Bretagne !");  
        break;  
  
    default:  
        System.out.println("Ailleurs que dans l'Ouest");  
        break;  
}
```

break : permet de quitter le switch, sans avoir à tester les autres cas.

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- **Exécution itérative**
- Rupture et continuation
- Structure d'une application

Exécution Itérative : for

➤ Syntaxe

```
for (expression1 ; expression2 ; expression3) instruction ;
for (expression1 ; expression2 ; expression3) {instructions ;}
```

- Expression1 : expression d'initialisation de l'itérateur. L'itérateur peut aussi être déclaré dans cette expression. L'expression n'est effectuée qu'une seule fois. On peut utiliser plusieurs itérateurs, séparés dans ce cas par des virgules.
- Expression2 : condition de test, tant que la condition est vraie on continue la boucle.
- Expression3 : cette expression modifie le ou les itérateurs à chaque itération.
- Ces deux dernières expressions sont facultatives. Dans ce cas, il faut prévoir dans les instructions :
 - Une sortie de la boucle par un *break*,
 - Une modification de l'itérateur.

Exécution Itérative : for

➤ Exemple

```
package demoCours.syntaxe;

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

        for (int i = 0; i < 5; i++)
        {
            System.out.print(" Valeur de i = " + i);
        }
    }
}
```

Résultats console :

```
Valeur de i = 0 Valeur de i = 1 Valeur de i = 2
Valeur de i = 3 Valeur de i = 4
```

Exécution Itérative : for (each)

➤ **Syntaxe**

```
for (TypeElement nomVariableElement : nomVariableTableau) { instructions ; }
```

- Boucle apparue avec la version 5 du langage.
 - Cette boucle « for » est aujourd’hui nettement moins fastidieuse que la précédente (le compilateur se charge de générer le code nécessaire).
- Nous n’avons pas encore abordé les tableaux. Cette boucle est utilisée quand on doit parcourir un ensemble de données de même type, donc des tableaux ou des listes que l’on abordera dans un prochain chapitre.

Exécution Itérative : for (each)

➤ Exemples

Résultats console:

```
package demoCours.syntaxe;

public class TestBoucles
{
    public static void main(String[] args)
    {
        //déclaration et initialisation d'un tableau d'entier

        int tableau [] = {1,2,3,4,5};

        for (int unElement : tableau)
        {
            System.out.print(" valeur de l'élément=" + unElement);
        }

        System.out.println();
        System.out.println("Ancienne écriture (avant 1.5)");

        for (int i = 0; i < tableau.length; i++)
        {
            System.out.print(" valeur de l'élément=" + tableau[i]);
        }
    }
}
```

Valeur de l'élément=1 valeur de l'élément=2 valeur de l'élément=3 valeur de l'élément=4
Ancienne écriture (avant 1.5)
valeur de l'élément=1 valeur de l'élément=2 valeur de l'élément=3 valeur de l'élément=4

Exécution Itérative : while

➤ Syntaxe

```
while (condition)
    instruction ;
while (condition)
{instructions ;}
```

- Si la condition est vérifiée on entre dans la boucle et tant qu'elle est vraie on continue la boucle. La condition doit être booléenne.

➤ Exemple

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

        int i = 5;
        while (i<=5)
        {
            System.out.print(" valeur de i=" + i);
            i+=1;
        }
    }
}
```

Exécution Itérative : do / while

- **Syntaxe**

```
do instruction while (condition);  
do {instructions;} while (condition);
```
- On exécute les instructions, si la condition est vérifiée on recommence la boucle. La condition doit être de type booléen

Résultats console :

```
Affichage des valeurs i=1 j=5  
valeur de l'élément=4 valeur de l'élément=3  
valeur de l'élément=2 valeur de l'élément=1  
valeur de l'élément=0 fin boucle  
Affichage des valeurs i=1 j=0
```

```
package demoCours.syntaxe;  
  
public class TestBoucles  
{  
    public static void main(String[] args)  
    {  
        int i = 1;  
        int j = 5;  
  
        System.out.println("Affichage des valeurs i=" + i + " j=" + j);  
  
        do  
        {  
            j = j - i;  
            System.out.print(" valeur de l'élément=" + j);  
        } while (j >= i);  
  
        System.out.println(" fin boucle");  
        System.out.println("Affichage des valeurs i=" + i + " j=" + j);  
    }  
}
```

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- **Rupture et continuation**
- Structure d'une application

RUPTURE ET CONTINUATION

➤ **label :**

- Il est possible d'attribuer un label à une instruction ou à un bloc d'instructions. Un label est identifié quand il termine par :

• **break :**

- L'instruction « **break** », valable aussi avec le *switch*, peut être utilisée dans les boucles. Dans ce cas, on quitte la boucle.

• **continue :**

- L'instruction « **continue** » peut être utilisée dans les boucles. Dans ce cas, on sort de la boucle mais on recommence une nouvelle itération.
 - Il est possible d'utiliser un nom de label, dans ce cas le contrôle est repris dans la boucle dépendant du label.

Rupture et Continuation

➤ Exemple

```
package demoCours.syntaxe;
public class TestBoucleMbriquees
{
    public static void main(String[] args)
    {

        boucle1: for (int i = 0; i < 7; i++)
        {
            System.out.println(" valeur de l'élément i=" + i);
            int j = 0;
            while (j <= i)
            {
                System.out.print(" valeur de j=" + j);
                j += 1;
                if (i == 5)
                    continue boucle1;
                if (j == 5)
                    break;
            }
            System.out.println(..");
        }
    }
}
```

Objectifs du chapitre

- Généralité du langage JAVA
- Packages
- Classes
- Méthodes
- Attributs – variables
- Objets
- Exemple
- Les opérateurs
- Exécution conditionnelle
- Exécution itérative
- Rupture et continuation
- **Structure d'une application**

Structure d'une application

➤ Programme principal :

- Un programme Java est une classe, pour être exécutable, la définition doit posséder une méthode «**main**».
- Cette méthode ne retourne aucun argument et doit être déclarée avec les modificateurs « *public* » et « *static* ».
- Elle reçoit en paramètre un tableau de chaîne de caractères.

```
public class MonExec {  
  
    public static void main (String args[])  
    {  
        //code java à exécuter  
    }  
} //fin classe
```

Structure d'une application

➤ Communiquer avec la machine

- Pour communiquer avec la machine virtuelle (JVM), on utilise la classe `System` (package `java.lang`).
- Dans un premier temps, nous utilisons, pour afficher des informations sur la console système, les méthodes `print` ou `println()` de cette classe `System` sur sa variable de classe `out`.

`System.out.println()`

```
public class Test
{
    //programme principal
    public static void main (String args[])
    {
        int var1 = 32 ;
        System.out.println( "la variable var1 vaut :" + var1) ;
    }
}
```

Structure d'une application

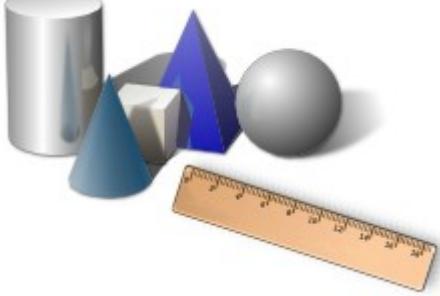
➤ Séparation Concept métier et Visuel

- Pour n'avoir à retoucher la définition d'une classe que si et seulement si les règles de gestion qui la concernent sont modifiées, nous décrirons dans des classes différentes :
 - Une ou plusieurs **classes métier**, contenant les informations et les comportements des objets « métier » Par exemple, pour tout client : mot de passe, numéros de comptes, ... ; les actions possibles étant de : se connecter, consulter un compte, déposer de l'argent, etc.
 - Une **classe ayant une méthode main**, permettant l'utilisation et les tests de ces objets : créations, manipulations, affichages.

Structure D'une Application : exemple

```
package exemple ;  
public class Personne  
{  
    // Attributs  
    private String nom ;  
    private int nbEnfants ;  
  
    // Constructeur  
    public Personne(String paramNom)  
    { nom = paramNom ; }  
  
    // Autres méthodes  
    public String getNom() { return nom ; }  
    public void enregistreNaissance(int nb) { nbEnfants += nb ; }  
    public int getNbEnfants() { return nbEnfants ; }  
}
```

```
package exemple ;  
public class Test  
{  
    //programme principal  
    public static void main (String args[])  
    {  
        Personne p1 = new Personne("Untel") ;  
        p1. enregistreNaissance( 2 ) ;  
        Personne p2 = new Personne("Machin") ;  
        System.out.println( "Monsieur " + p1.getNom()  
                            + " a " +p1.getNbEnfants()+" enfants" );  
        System.out.println( "Monsieur "+p2.getNom()  
                            + " a " +p2.getNbEnfants()+" enfants" );  
    }  
}
```



Exercice 1 - Se référer au cahier d'exercices

Eléments de syntaxe / Concepts

➤ GESTION BANCAIRE - ELEMENTS DE SYNTAXE :

- *Gestion des dépôts et retraits sur un compte en banque*
- *Gestion du compte avec encapsulation*
- *Suppression de la variable solde, ajout de 2 attributs*
- *Gestion des découverts*

➤ La classe Compte à définir devra toujours avoir les signatures :

- public void **depotDe(int montant)** //pour déposer 1 montant
- public void **retraitDe(int montant)** //pour retirer 1 montant
- public int **getSolde()** //pour obtenir le solde du Compte
- public **Compte()** //Le constructeur

La classe Compte **ne doit pas contenir d'affichage console !!**

Compte
//Attributs int solde ;
//Méthodes Compte() void depotDe(int m) void retraitDe(int m) int getSolde()

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** **Compléments au langage Java**
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Les tableaux
- La classe String
- Les classes « ENVELOPPES »
- Les vecteurs
- Autres collections
- Les itérateurs
- Les Tris

Les tableaux : Intérêt

- Les tableaux permettent de mémoriser une collection d'éléments de même type.
- La taille est fixée lors de la déclaration.
 - On a donc une borne inférieure et une borne supérieure.
 - Lors de l'accès à un élément, à l'aide d'un indice mis entre crochet « [] », il y a vérification de l'existence de la borne et une erreur d'exécution si la borne n'est pas trouvée.

Les tableaux : Déclaration

- La déclaration d'un tableau est similaire à la déclaration d'une variable.
 - C'est à dire, qu'il doit toujours être précédé du **type de l'élément** que l'on veut stocker puis de **son nom**.
 - Pour indiquer qu'il s'agit d'un tableau, derrière le type de l'élément on met des crochets « [] ».
- On peut déclarer plusieurs tableaux de même type, en même temps, dans ce cas on les sépare par des virgules.
- On peut initialiser un tableau au moment de sa déclaration. Dans ce cas, on utilise les accolades et l'on sépare les valeurs par des virgules.
- Pour indiquer le nombre d'éléments à stocker, on crée une instance avec l'opérateur **new** et on met dans les crochets « [] » le nombre d'éléments

Les tableaux : Déclaration

```
// Déclaration d'un tableau
double [] nomTableau;
Client [] listeClients ;

// Déclaration de plusieurs tableaux
int [] nomTableau1, nomTableau2 ;

// Déclaration et initialisation
int [] nomTableau4 = {1,2,3,4,5};

// Déclaration du nombre d'éléments
nomTableau = new double [3];
```

Les tableaux : Valorisation

- Le premier poste a l'indice 0.
- Pour valoriser un élément, on utilise les crochets [].
- Pour connaître le nombre de postes déclarés on utilise la variable *length* appliquée au tableau.

```
// Déclaration et initialisation
int [] nomTableau1 = {1,2,3,4,5};

// Valorisation
nomTableau1[0] = 6;

// Affichage du nombre de postes
System.out.println(nomTableau1.length);
```

Les tableaux : Exemple

- Nous reprenons notre client et nous décidons d'ajouter un attribut sous forme de tableau :
 - Ce tableau *mails* contiendra au maximum 3 chaînes de caractères.
 - Nous ajoutons un entier pour connaître le nombre de mail déjà renseigné.
- Nous modifions :
 - Le constructeur de Client, afin d'initialiser le tableau,
 - La méthode `retourneInfo()`, pour retourner l'ensemble du tableau
- Nous avons créé deux méthodes `get/set`, non pas pour retourner et mettre à jour tout le tableau, mais pour gérer un seul élément du tableau.

Les tableaux : Exemple 1/2

```
package demoCours;
public class Client {
    // Attributs
    private int numero;
    private String nom;
    private Adresse adresse;
    private String [] mails;
    private int nbMails;

    public Client() {
        // Constructeur - Initialise les attributs
        nom = "inconnu";
        numero = 0;
        // Création de l'objet adresse
        adresse = new Adresse();
        nbMails = 0;
        mails = new String[3];
    }
    public void setMails(String mail) {
        this.mails[nbMails] = mail;
        this.nbMails += 1;
    }
    public String getMails() {
        // Si aucun mail enregistré on n'exécute pas la boucle, on retourne l'info
        if (nbMails == 0) return "aucun mail";

        String chaine = "";
        for (int i = 0; i < nbMails; i++) {
            chaine += mails[i] + " | ";
        }
        return chaine;
    }
    public String retourneInfo(){
        return " Le client est " + this.getNumero() + " - " + this.getNom()
            + " il habite: " + this.getAdresse().getVille()
            + " ses mails : " + this.getMails();
    }
}
```

Déclaration du tableau qui contiendra des objets String

Initialisation du tableau qui contiendra 3 éléments

Valorisation d'un élément du tableau

Boucle pour récupérer tous les éléments du tableau

```
package demoCours;
public class TestClientTab {
    public static void main(String[] args) {
        // Manipulation d'un Client
        // Déclaration de la variable c1 de type Client
        Client c1;

        // Initialisation de la variable :
        // consiste à appeler le constructeur
        c1 = new Client();

        // Manipulation des attributs
        c1.setNom("Ets CROY");
        c1.setNumero(100);

        // Affichage à la console système
        System.out.println(c1.retourneInfo());

        c1.setMails("c.roy@gk.com");
        System.out.println(c1.retourneInfo());
        c1.setMails("deuxième adresse");
        c1.setMails("3eme adresse");

        // Affichage à la console système
        System.out.println(c1.retourneInfo());
    }
}
```

Le test =
Exécutable

Les tableaux : Exemple 2/2

Exécution du test

Résultats console :

Le client est 100 - Ets CROY il habite: à renseigner ses mails : aucun mail
Le client est 100 - Ets CROY il habite: à renseigner ses mails : c.roy@gk.com |
Le client est 100 - Ets CROY il habite: à renseigner ses mails : c.roy@gk.com |
deuxième adresse | 3eme adresse |

```
package demoCours;  
  
public class TestClientTab  
{  
  
    public static void main(String[] args)  
    {  
        //Manipulation d'un Client  
  
        // Déclaration de la variable c1 de type Client  
        Client c1;  
  
        // Initialisation de la variable :  
        // consiste à appeler le constructeur  
        c1 = new Client();  
  
        // Manipulation des attributs  
        c1.setNom("Ets CROY");  
        c1.setNumero(100);  
  
        // Affichage à la console système  
        System.out.println(c1.retourneInfo());  
  
        c1.setMails("c.roy@gk.com");  
        System.out.println(c1.retourneInfo());  
        c1.setMails("deuxième adresse");  
        c1.setMails("3eme adresse");  
  
        // Affichage à la console système  
        System.out.println(c1.retourneInfo());  
    }  
}
```

Les tableaux : Copie

- La classe System contient une méthode de classe (déclarée en static), nommée « **arraycopy** » qui permet de copier le contenu d'un tableau.
- Il existe une version de cette méthode pour chaque type de tableau. (Utilisation du polymorphisme).
- La version 6 de Java SE a ajouté la présence des méthodes **Arrays.copyOf()** et **Arrays.copyOfRange()** permettant de redimensionner/tronquer/copier des tableaux de tous types plus simplement qu'avec la méthode **System.arraycopy()** quelque peu complexe...

Les tableaux : Copie

➤ Exemple

```
// Déclaration et initialisation
int [] nomTableauO = {1,2,3,4,5};
int [] nomTableauD = new int[3];
int debutIndiceO = 1 ;
int debutIndiceD = 0 ;
int nombreElement = 3 ;

// Copie
System.arraycopy(nomTableauO, debutIndiceO, nomTableauD, debutIndiceD,
nombreElement);
```

Le tableau nomTableauD contient alors : {2,3,4}.

Les tableaux : Copie

➤ Exemple de copie de tableau :

Résultats console :

```
Le client est 100 - Ets CROY il habite: à renseigner ses mails : aucun mail
Le client est 100 - Ets CROY il habite: à renseigner ses mails :
c.roy@gk.com | 
Le client est 100 - Ets CROY il habite: à renseigner ses mails :
c.roy@lgk.com | deuxième adresse | 3eme adresse |
null|null|null|c.roy@gk.com|deuxième adresse|3eme adresse|null|null|null|
```

```
package demoCours;
public class TestClientTab
{

    public static void main(String[] args)
    {
        // Manipulation d'un Client
        Client c1 = new Client();
        c1.setNom("Ets CROY");
        c1.setNumero(100);
        System.out.println(c1.retourneInfo());
        c1.setMails("c.roy@gk.com");
        System.out.println(c1.retourneInfo());
        c1.setMails("deuxième adresse");
        c1.setMails("3eme adresse");
        System.out.println(c1.retourneInfo());

        // Déclaration de tableaux
        String[] infoClient, infoCopie;

        // Initialisation du tableau
        infoClient = c1.getTabMails();

        // Copie du tableau
        infoCopie = new String[9];

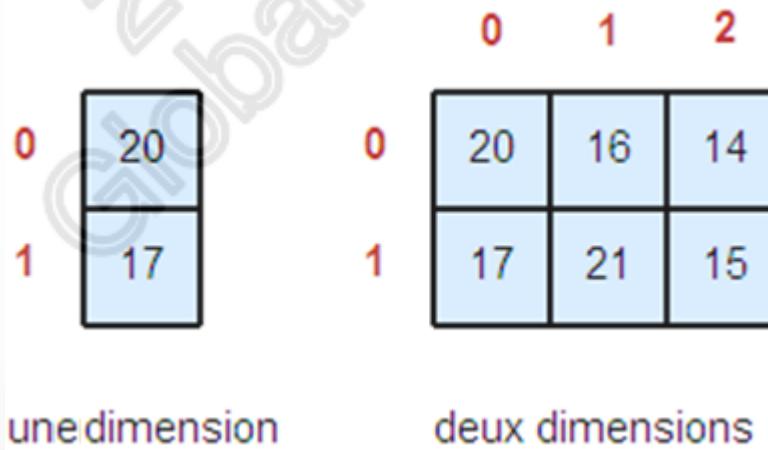
        System.arraycopy(infoClient, 0, infoCopie, 3, 3);

        // Affichage du tableau
        String affic = "";
        for (String string : infoCopie)
        {
            affic += string + "|";
        }

        System.out.println(affic);
    }
}
```

Les tableaux : Multi-dimensions

- Il est parfois utile de stocker un tableau à deux dimensions, que l'on compare à la notion courante de tableau avec des lignes et des colonnes.
- Pour accéder à un élément d'un tel tableau, il faut maintenant utiliser deux indices.
 - Le premier représentant la ligne et le second la colonne.

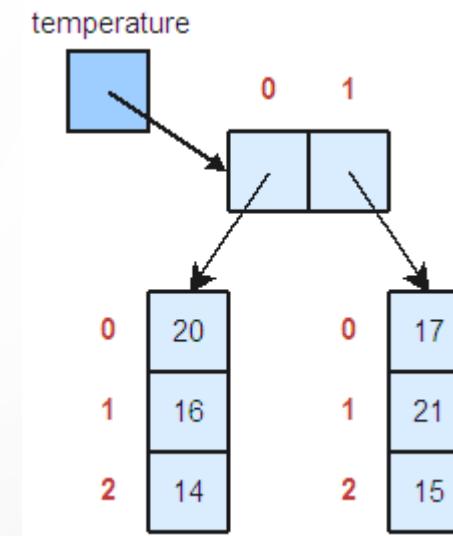


Les tableaux : Multi-dimensions

- Pour déclarer un tel tableau, on utilise comme précédemment des crochets mais comme on est à deux dimensions, on va utiliser deux crochets.
- Pour le tableau ci-dessus, on va écrire
- Mais qu'est-ce qu'en réalité un tableau à deux dimensions ? C'est en fait un tableau à une dimension qui contient des références vers des tableaux

```
int[][] temperature = new int[2][3]
```

```
int [] [] nomTab
```



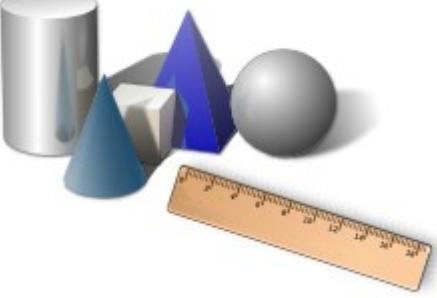
Les tableaux : Multi-dimensions

- Pour parcourir ce genre de tableau, il faudra deux boucles « for » imbriquées.
- Comment connaître les «largeur» et «hauteur» de ce tableau ?
 - En utilisant la variable **length** on obtiendra la longueur du premier tableau c'est à dire 2
 - Ensuite, comme on a vu que c'est un tableau de tableaux, il suffit de consulter la variable **length** non plus sur **temperature** mais sur **temperature[0]** ou **temperature[1]**

```
public class MultiDim {  
    public static void main(String[] args) {  
        int [][] liste = { {1,2,3}, {4,5,6} };  
        for (int i = 0; i < liste.length; i++) {  
            System.out.println("indice i=" + i);  
            for (int j = 0; j < liste[i].length; j++) {  
                System.out.print(liste[i][j] + "|");  
            }  
            System.out.println();  
        }  
    }  
}
```

Résultat console
→

indice i=0
1|2|3|
indice i=1
4|5|6|



Exercice 2 - Tableaux cahier d'exercices)

Changer les types int en int []

Compte
//Attributs
int somDepot ;
int somRetrait;

//Méthodes
Compte()
void depotDe(int m)
void retraitDe(int m)
int getSolde()

Changer les contenus pas les signatures

GESTION BANCAIRE - TABLEAUX

- L'idée est maintenant de conserver dans 2 tableaux, **à 1 dimension**, les montants déposés et les montants retirés.
- Le type des attributs somDepot et somRetrait passe donc de int à int[]
- Pour cela, il sera nécessaire d'ajouter, à notre classe Compte, deux attributs: iD et iR, pour gérer le remplissage d'élément dans le tableau
- Nous rappelons que les **signatures des méthodes ne changent pas !**
- Vous pouvez toutefois ajouter d'autres méthodes si vous le souhaitez.
- Votre test ne doit pas changer puisqu'il utilise les méthodes de Compte

Objectifs du chapitre

- Les tableaux
- **La classe String**
- Les classes « ENVELOPPES »
- Les vecteurs
- Autres collections
- Les itérateurs
- Les Tris

La classe String : Intérêt

- Java permet de définir des chaînes de caractères à l'aide de la classe « **String** ». Cela permet de stocker du texte et d'avoir accès à différentes fonctions servant à manipuler du texte.
- Cette classe appartient au package « `java.lang` ».
- Nous vous indiquons ici quelques éléments de cette définition, il faudra par la suite, par vous-même approfondir.

La classe String : Déclaration

- La classe « String » offre un raccourci pour créer une instance en utilisant les guillemets

```
String uneChaine = "Bonjour";
```

```
= String uneChaine = new String("Bonjour");
```

- Cette classe contient aussi plusieurs constructeurs

```
public String (String valeur)
public String (char [] valeur)
public String (char [] valeur, int debut, int long)
```

La classe String : Principales Méthodes

- Package `java.lang.*`
- Cette liste n'est pas exhaustive. Il existe bien d'autres méthodes utiles telles que :
 - `toLowerCase()`,
 - `toUpperCase()`,
 - `equalsIgnoreCase(String s)`,
 - etc.
- L'intérêt ici est la méthode **`equals`** pour comparer l'égalité des objets

Méthodes	Définition
<code>int length()</code>	Retourne la longueur réelle de la chaîne
<code>char charAt(int i)</code>	Retourne le caractère à l'indice donné
<code>String substring(int début)</code>	Retourne la sous-chaîne à partir de l'indice donné inclus
<code>String substring(int début, int fin)</code>	Retourne la sous-chaîne à partir du 1 ^{er} indice donné jusqu'au 2 ^{ème} indice donné -1 inclus
<code>int indexOf(char c)</code>	Retourne la position du caractère
<code>int indexOf(char c, int i)</code>	Retourne la position du caractère à partir de i-1 inclus
<code>int lastIndexOf(char c)</code>	Retourne la position du caractère en partant de la fin
<code>int lastIndexOf(char c, int i)</code>	Retourne la position du caractère en partant de la fin à partir de i inclus
<code>boolean equals(String s)</code>	Retourne true si les 2 chaînes ont les mêmes valeurs

La classe String : Exemple

Suite du résultat :

```
2
30
Globa
recomptez
```

```
package mesTests;

public class Test
{
    public static void main (String [] args)
    {
        String s = "Bienvenue chez Global Knowledge";
        int i ;
        System.out.println(s.length()) ;

        for (i = 0; i < s.length(); i++)
        {
            System.out.println(s.charAt(i) ) ; //Affichage de chaque caractères 1 à 1 →
        }
        System.out.println(s.indexOf('e') ) ;
        System.out.println(s.lastIndexOf('e') );
        String nom= s.substring(15, 20);
        System.out.println(nom );
        if (nom.equals("Global"))
        {
            System.out.println("Extraction ok");
        }
        else
        {
            System.out.println("recomptez");
        }
    }
}
```

La classe String : Conversions

- La méthode de classe **valueOf()** permet de convertir un primitif en chaîne

```
public class Test
{
    public static void main (String [] args)
    {
        String s ;
        boolean b = true;
        int i = 2000;
        long l = 2001;
        float f = (float) 3.14;
        double d = 3.1415926;

        s= String.valueOf(b);
        System.out.println(s);      → true
        s= String.valueOf(i);
        System.out.println(s);      → 2000
        s= String.valueOf(l);
        System.out.println(s);      → 2001
        s= String.valueOf(f);
        System.out.println(s);      → 3.14
        s= String.valueOf(d);
        System.out.println(s);      → 3.1415926
    }
}
```

La classe String : Concaténations

- Pour concaténer des chaînes, l'opérateur **+** est à utiliser.
 - Cependant, avec les String de nombreux objets intermédiaires peuvent être créés par la JVM.
 - Il est préférable d'utiliser un **StringBuffer** ou un **StringBuilder(Java5)** pour ajouter des chaînes de caractères

```
String s1 = "Bienvenue" ;  
String s2 = "chez Global Knowledge !" ;  
StringBuilder sb = new StringBuilder() ;  
sb.append(s1).append(" ").append(s2) ;  
String s3 = sb.toString() ;  
// A privilégier par rapport à  
String s3 = s1 + " " + s2 ;
```

Objectifs du chapitre

- Les tableaux
- La classe String
- **Les classes « ENVELOPPES »**
- Les vecteurs
- Autres collections
- Les itérateurs
- Les Tris

Les classes « Enveloppes »

- Les classes « **enveloppes** », ou « **wrapper** » permettent d'encapsuler des données de type primitif dans des objets.
 - En effet, les données de type primitif ne sont pas des objets et certaines méthodes ne traitent que des objets.
- Les wrappers appartiennent au package « `java.lang` ».

Type simple	Classe
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>double</code>	<code>Double</code>
<code>float</code>	<code>Float</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>

Les classes « Enveloppes » : Déclaration

- Les wrappers ont des méthodes statiques pour éviter l'instanciation.
- Les wrappers ont des constructeurs qui attendent une donnée du type simple en paramètre.
- Les wrappers offrent des services comme la conversion.

```
Integer unEntier= new Integer(10) ;  
boolean b = true ;  
Boolean unBooleen = new Boolean(b) ;
```

Les classes « Enveloppes » : Utilisation

- Chaque wrapper contient une méthode unique d'accès à la valeur du type simple.

Type simple	Méthode d'accès
boolean	booleanValue()
char	charValue()
double	doubleValue()
float	floatValue()
int	intValue()
long	longValue()

- Il existe une méthode commune pour la représentation externe, sous forme de chaîne de caractères en utilisant la méthode `toString()`.

```
public class Test
{
    public static void main (String [] args)
    {
        String s;
        int i = 2000;
        Integer unEntier = new Integer(i);
        s = unEntier.toString();
        System.out.println(s);           → 2000
    }
}
```

Les classes « Enveloppes » : Conversions

- Pour convertir une chaîne en type primitif, on utilise les méthodes `xxValue` (où `xx` représente le primitif).
- Pour les wrappers « `Integer` » et « `Long` », on peut appliquer les méthodes de conversion «`parseXx` ».
- Attention, java vérifie, à l'exécution, les opérations faites avec des entiers (division ou modulo 0), mais **pas avec des réels**

```
public class Test
{
    public static void main (String [] args)
    {
        String s;
        boolean b;
        int i;
        long l;
        float f;
        double d;

        s = "false";
        b = new Boolean(s).booleanValue();
        System.out.println(b); → false

        s = "1234";
        i = Integer.parseInt(s);
        System.out.println(i);
        System.out.println(new Integer(s).intValue()); → 1234 → 1234

        s = "6789";
        l = Long.parseLong(s); → 6789

        s = "6.78";
        f = new Float(s).floatValue(); → 6.78

        s = "3.1e5";
        d = new Double(s).doubleValue(); → 310000.0
    }
}
```

Objectifs du chapitre

- Les tableaux
- La classe String
- Les classes « ENVELOPPES »
- **Les vecteurs**
- Autres collections
- Les itérateurs
- Les Tris

Les vecteurs : Définition

- Les vecteurs sont issus de la classe « **Vector** ». Cette classe appartient au package « `java.util` » qu'il faut importer.
- Ils représentent ce que l'on appelle des collections dynamiques, des structures de données.
- La classe Vector du package `java.util` permet de stocker des objets dans un tableau dont la taille évolue avec les besoins.

Les vecteurs : Intérêt

- Les vecteurs ont une taille quelconque, ils ne sont donc pas bornés, comme les tableaux.
- Les éléments que l'on peut stocker sont uniquement des objets. Un **type primitif doit être enveloppé** pour pouvoir être mémorisé.
 - Depuis la version 5 du langage, l'emballage du primitif sous sa forme objet est fait automatiquement.
- Dans un vecteur, on peut mémoriser des types d'objets différents. Il existe une gestion dédiée aux vecteurs : accès, énumération.

Les vecteurs : Utilisation

➤ Déclaration et initialisation

- Instance de la classe Vector. Le vecteur créé ne contient aucun élément.

```
Vector unVecteur = new Vector();
```
- Ici, le vecteur unVecteur peut contenir tout type d'objet.
- Il est possible depuis la version 5, de préciser quels seront les types d'objet pouvant être stockés dans le vecteur.
 - Ainsi, la machine contrôlera le type d'objet avant de le stocker.
- Par exemple, on souhaite créer :
 - Un vecteur listeDesClients ne contenant que des objets de type Client,
 - Un vecteur maListe ne contenant que des entiers. Le type int étant primitif il faut passer par la classe enveloppe Integer.

```
Vector<Client> listeDesClients = new Vector<Client>();  
Vector<Integer> maListe = new Vector<Integer>();
```

Les vecteurs : Ajout

- Le premier élément a l'indice 0.
 - `addElement()`
- L'ajout d'un objet dans un vecteur s'effectue par la méthode « **addElement()** » qui attend en argument l'objet à stocker.
(Nécessité de passer par les wrappers pour les primitifs).

maListe →

0	Integer
	value=0
1	Integer
	value=0
2	Integer
	value=1
3	Integer
	value=1
4	Integer
	value=2
5	Integer
	value=2

```
import java.util.*;
public class Vect
{
    public static void main (String [] args)
    {
        Vector maListe = new Vector();
        for (int i = 0 ; i < 3 ; i++)
        {
            // Ecriture obligatoire avant la version 5
            maListe.addElement(new Integer (i));
            // Ecriture à partir de la version 5
            maListe.addElement(i);
        }
    }
}
```

Les vecteurs : Ajout

➤ insertElementAt()

- Il est possible d'insérer un élément à un endroit particulier, en utilisant la méthode « **insertElementAt()** », qui attend en argument l'objet et l'indice du vecteur.
- L'indice doit avoir été créé au préalable, sinon une erreur est générée. (On ajoute un nouvel élément.)

```
unVecteur.insertElementAt(new Integer(12), 0);
```

Les vecteurs : Manipulation

➤ Remplacement

- Pour mettre un objet à la place d'un autre, on utilise la méthode `setElementAt()`.
- Cette méthode attend en argument l'objet et l'indice à modifier.

`unVecteur.setElementAt(new Integer(12), 0);`

➤ Taille :

- Pour connaître le nombre d'éléments stockés dans un vecteur on utilise la méthode `size()`.

```
public static void main (String [] args)
{
    Vector unVecteur = new Vector() ;
    for (int i=0 ; i<3 ; i++) {
        unVecteur.addElement(new Integer (i)) ;
    }
    unVecteur.insertElementAt(new Integer(12), 1);
    System.out.println(unVecteur.size()) ; → 4
}
```

Les vecteurs : Accès

➤ Caste

- On a placé un Integer (sous-classe de Object) dans un Vector.
- On va récupérer un objet (appartenant à la classe Object).
- Si on applique une méthode définie dans Integer à l'objet récupéré cela ne fonctionne pas, car Object ne connaît pas la méthode de sa sous-classe Integer.
- Java est conscient que cela est pratique, il considère que l'on doit prendre nos responsabilités. → On doit donc effectuer une conversion dynamique de type, on dit que l'on « caste ». A chaque fois que l'on manipule les vecteurs on doit « caster » !
- Depuis Java5 si le vecteur est créé avec un générique, la caste n'est plus nécessaire.

➤ elementAt()

- Pour récupérer l'objet stocké on utilise la méthode « **elementAt()** » qui attend en argument l'indice de stockage.

```
import java.util.*;
public class Vect
{
    public static void main (String [] args)
    {
        int mt = 0 ;
        Vector unVecteur = new Vector() ;
        // Remplissage
        for (int i=0 ; i<3 ; i++) {
            unVecteur.addElement(new Integer (i)) ;
        }
        unVecteur.insertElementAt(new Integer(12), 1);
        // Récupération
        Integer elt;
        for (int i=0 ; i<unVecteur.size(); i++) {
            elt = (Integer) unVecteur.elementAt(i); //caste
            System.out.println("entier "+ elt); → Affiche 1 à 1 le contenu
            // On cumule dans un montant de type « int » les éléments du vecteur
            // Il faut donc récupérer la valeur primitive de l'Integer (via la méthode)
            mt = mt + elt.intValue(); // La JVM fait implicitement le intValue() depuis V5
        }
        System.out.println("montant = "+ mt); → montant = 15
    }
}
```

Nous remplissons
3 éléments de type
Integer dans le
Vector

Nous insérons un
4ème élément de
type Integer dans
ce vecteur

Caste obligatoire,
car pas de
généricité

```
import java.util.*;
public class Vect
{
    public static void main (String [] args)
    {
        int mt = 0 ;
        Vector <Integer> unVecteur = new Vector<Integer>() ;
        // Remplissage
        for (int i=0 ; i<3 ; i++) {
            unVecteur.addElement(i) ;
        }
        unVecteur.insertElementAt(12, 1);
        // Récupération
        int elt;
        for (int i=0 ; i<unVecteur.size(); i++) {
            elt = unVecteur.elementAt(i);
            System.out.println("entier "+ elt); → Affiche 1 à 1 le contenu
            // On cumule dans un montant de type « int » les éléments du vecteur
            // écriture simplifiée avec le générique
            mt = mt + elt ;
        }
        System.out.println("montant = "+ mt); → montant = 15
    }
}
```

Même exemple
avec généricité

Nous ajoutons des
éléments sans
passer par la classe
enveloppe

Pas de caste, car
généricité

Les vecteurs : Recherche

- Méthodes de Vector du package java.util.*
- Suppression

Méthodes	Définition
boolean contains(Object o)	Retourne vrai si l'objet est trouvé
int indexOf(Object o)	Retourne l'indice de l'objet Retourne -1 si non trouvé
int indexOf(Object o, int i)	Retourne l'indice de l'objet à partir de l'indice i inclus Retourne -1 si non trouvé)
int lastIndexOf(Object o)	Retourne le dernier indice de l'objet Retourne -1 si non trouvé
int lastIndexOf(Object o, int i)	Retourne le dernier indice de l'objet à partir de l'indice i Retourne -1 si non trouvé
Object firstElement()	Retourne le premier objet stocké
Object lastElement()	Retourne le dernier objet stocké

Méthodes	Définition
boolean removeElement(Object o)	Supprime la 1 ^{ère} occurrence de l'objet indiqué
void removeElementAt(int i)	Supprime l'objet à l'indice i
void removeAllElements()	Supprime tous les objets

Vector : exemple, définition d'une Personne ayant plusieurs mails possibles (reprise ex avec tableau)

```
package modeles;
import java.util.Vector;
public class Personne {
// Attributs d'instance = propriétés
private String prenom;
private Vector<String> mails;
// constructeur
// BUT : initialiser les variables d'instance
public Personne() {
prenom = "inconnu";
mails = new Vector();
}
// ajout d'un nouveau mail
public void setMail(String mail) {
mails.addElement(mail);
}
```

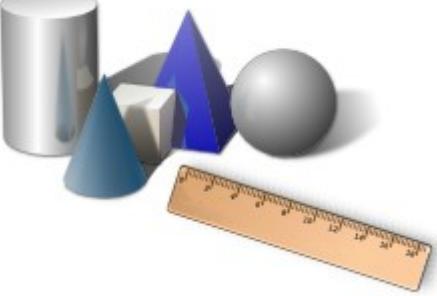
```
// retourner 1 mail
public String getMail(int index) {
if (index > mails.size())
return "index trop grand ! Mail inexistant";
return mails.elementAt(index);
}
public Vector<String> getMails() {
return mails; }
public void setMails(Vector<String> mails) {
this.mails = mails; }
public String getPrenom() {return prenom;}
public void setPrenom(String prenom) {
this.prenom = prenom; }
} // fin classe
```

Vector : exemple

- L'exécutable pour tester la classe Personne
- Le résultat à la console

```
la personne se nomme : inconnu
les mails
[aline.cassar@globalknowledge.fr,
aline.cassar@gmail.com]
Premier aline.cassar@globalknowledge.fr
index trop grand ! Mail inexistant
```

```
package modeles;
public class PersonneTest {
public static void main(String[] args) {
//initialisation d'un Objet
Personne p = new Personne();
System.out.println("La personne se nomme : " +
p.getPrenom());
//Ajout d'un mail
p.setMail("aline.cassar@globalknowledge.fr");
//Ajout d'un autre mail
p.setMail("aline.cassar@gmail.com");
//Affichage de tout le vecteur
System.out.println("Les mails");
System.out.println(p.getMails());
//Affichage du 1er mail
System.out.println("Premier " + p.getMail(0));
//affichage d'un élément inexistant
System.out.println( p.getMail(5));
} //main } //classe
```



Exercice 3 (voir cahier d'exercices)

Vector

➤ GESTION BANCAIRE - VECTEURS

- **Exploitation traditionnelle de Vector**
- **Nous avons pu remarquer que les tableaux ne sont pas adaptés à ce type de gestion, nous changeons donc les types int[] en Vector.**
- **RAPPEL : pas de modification de signature dans Compte. Le test reste inchangé (s'il est bien défini ☺)**

Changer les types
int [] en Vector

Supprimer iD et
IR puisque la
classe Vector gère
l'indice de
remplissage

Changer les
contenus pas les
signatures

```
Compte
//Attributs
int[] somDepot ;
int[] somRetrait;
int iD, iR;
//Méthodes
Compte()
void depotDe(int m)
void retraitDe(int m)
int getSolde()
```

Objectifs du chapitre

- Les tableaux
- La classe String
- Les classes « ENVELOPPES »
- Les vecteurs
- **Autres collections**
- Les itérateurs
- Les Tris

Autres collections, itérateurs et tris

- La suite de ce chapitre est à titre d'information
- Le but est de vous indiquer qu'il existe de multiples collections en Java, et qu'il faudra choisir celle le plus appropriée à la gestion à mettre en œuvre
- Le principe des collections reste identique à notre présentation de Vector, les méthodes changent pour ajouter/récupérer un élément, nous vous invitons donc à vous référer à la documentation officielle ou non

Les ArrayList : Intérêt

- Comme la classe « Vector », la classe `java.util.ArrayList` représente un tableau d'objets dont la taille est dynamique.
- La différence avec la classe `Vector` est que, dans cette dernière, toutes les méthodes sont synchronisées, ce qui est utile pour le multi-tâches, mais la plupart du temps inutile et coûteux si mono-tâches.
- Il est donc le plus souvent préférable d'utiliser un objet de la classe `ArrayList`, quand il n'y a pas plusieurs utilisateurs susceptibles de modifier en même temps la collection.
- Dans nos exercices, le `Compte` peut être utilisé par plusieurs personnes en même temps d'où la manipulation de `Vector`.

Les ArrayList : méthodes

Méthodes	Définition
<code>boolean add(Object)</code>	Ajoute un élément à la liste
<code>Object get(index)</code>	Renvoie l'élément dont la position est précisée
<code>int indexOf(Object)</code>	Renvoie la position de la première occurrence de l'objet fourni en paramètre
<code>int lastIndexOf(Object)</code>	Renvoie la position de la dernière occurrence de l'objet fourni en paramètre
<code>boolean isEmpty()</code>	Indique si la liste est vide
<code>Object remove(int)</code>	Supprime l'objet dont la position est fournie en paramètre
<code>void clear()</code>	Supprime tous les éléments de la liste
<code>Size()</code>	Taille d'un ArrayList

Tables de hash (Hashtable) : Intérêt

- La classe Dictionary permet d'utiliser des structures de données ayant comme index non pas un indice mais une clé.
- Cette classe est abstraite, pour manipuler des informations similaires aux vecteurs mais avec une clé, on instancie la classe **Hashtable**.
- La clé est n'importe quel objet.

Tables de hash (Hashtable) : méthodes

➤ Package java.util.*

Méthodes	Définition
Object put(Object cle, Objet o)	Place l'objet o ayant la clé cle retourne l'objet
Object get(Object cle)	Retourne l'objet ayant la clé cle
boolean contains(Object o)	Retourne vrai si o existe dans la table
boolean containsKeys(Object cle)	Retourne vrai si la clé existe
Object remove(Object cle)	Supprime l'objet ayant la clé cle retourne l'objet
void clear()	Supprime tout (clés et objets)
Enumeration elements()	Retourne une énumération (tout le contenu)
int size()	Retourne le nombre d'éléments stockés
protected void rehash()	Force le « rehashage »
boolean isEmpty()	Teste si la table est vide

Les API java: java.util.Hashtable

- Cette classe gère une collection d'objets au travers d'une “table de hachage”
- La HashTable équivaut à un Vector sauf que les clés sont des **String** au lieu de **numériques**
- Pas de notion d'ordre comme dans un vecteur !!

```
Hashtable ht = new Hashtable();  
  
ht.put("noel", new Date("25 Dec 1997"));  
ht.put("une chaine", "abcde");  
ht.put("un vecteur", new Vector());  
  
Vector v = (Vector)ht.get("un vecteur");  
System.out.println(ht.get(" une chaine ")); // --> abcde
```

- La HashTable permet de retrouver la liste des valeurs des clés présentes

Piles (stack) : Intérêt

- La classe « **Stack** » permet de stocker des données sous forme de pile de type « dernier entré », « premier sorti ».
 - « Stack » hérite de « **Vector** », il s'agit aussi d'une structure de données.
- En utilisant le constructeur, on crée un objet de type pile vide.

Piles (stack) : méthodes

- Les méthodes *elements()* et *size()* sont héritées de la classe Vector.

Méthodes	Définition
Object push(Object o)	Place l'objet o sur le dessus de la pile. Retourne l'objet.
Object pop()	Supprime l'objet du dessus de la pile. Retourne l'objet.
Object peek()	Récupère l'objet du dessus de la pile. Retourne l'objet.
int search(Object o)	Retourne la distance séparant l'objet du haut de la pile. Retourne -1 si objet non trouvé.
boolean empty()	Teste si la pile est vide

Collections type Queue

- Java 5 a ajouté une file d'attente **Queue** (qui hérite de collection) où le développeur définit le type FIFO (qui signifie first in first out) et/ou LIFO (last in, first out).
- Méthodes :
 - add(T t) et offer(T t) permettent d'ajouter un élément à la liste
 - remove() et poll() retirent toutes les deux les éléments de cette file d'attente
 - element() et peek() examinent toutes les deux l'élément disponible, sans le retirer de la file d'attente

Interface Deque

- En Java 6, **Deque**, extension de Queue voit son apparition.
- Elle définit la notion de file d'attente à double extrémité : il est possible d'ajouter des éléments au début de la file ou à la fin.

	Lève une exception	Retourne false
Insertion	<code>addFirst(T t) / addLast(T t)</code>	<code>offerFirst(T t) / offerLast(T t)</code>
Retrait	<code>removeFirst(T t) / removeLast(T t)</code>	<code>pollFirst(T t) / pollLast(T t)</code>
Examen	<code>getFirst(T t) / getLast(T t)</code>	<code>peekFirst(T t) / peekLast(T t)</code>

Objectifs du chapitre

- Les tableaux
- La classe String
- Les classes « ENVELOPPES »
- Les vecteurs
- Autres collections
- **Les itérateurs**
- Les Tris

Les itérateurs : Concepts

- Les itérateurs servent à parcourir les collections (de type séquence comme les vecteurs, liste, piles, files ou autres).
- L'itérateur est simplement un indice qui accède aux éléments de la collection, sans se préoccuper des objets de la collection.
- La plupart du temps, l'itérateur se déplace en avant. Il existe des itérateurs permettant de se déplacer d'avant en arrière.
- Le package `java.util` offre différents itérateurs, nous détaillons ici l'itérateur `Enumeration`, un exemple d'`Iterator` sera aussi fourni.
- Nous rappelons que Java5 à simplifier l'écriture de code et l'arrivée de la boucle `for (each)` est maintenant plus souvent utilisée que les itérateurs.

Les itérateurs : Enumeration

➤ Intérêt d'Enumeration

- Les énumérations fournissent un moyen standard d'effectuer des itérations au sein d'une liste d'éléments stockés
- Avec les Vector, Hashtable, Stack, ... on peut donc manipuler les énumérations pour récupérer les éléments.
- On ne manipule pas une énumération seule, on l'utilise avec une collection de données

Les itérateurs : Enumeration

➤ Déclaration de l'itérateur-Enumeration

- Les types d'objets « **Enumeration** » appartiennent au package `java.util`.
(Les énumérations sont des interfaces).
- L'énumération s'obtient via la méthode « **elements()** » appliquée à un objet de type collection, tel `Vector`.

```
import java.util.*;
public class Test
{
    public static void main (String [] args)
    {
        Vector unVecteur = new Vector();
        ...
        Enumeration ensemble = unVecteur.elements();
        ...
    }
}
```

Objet : `unVecteur`

Objet1	Objet2	Objet3	Objet4	Objet5
--------	--------	--------	--------	--------

`unVecteur.elements() → ensemble (Enumeration)`



ensemble est un itérateur, positionné sur le premier élément du vecteur

Les itérateurs : Enumeration

- **Méthodes de l'interface Enumeration**
 - Une énumération contient deux méthodes :
 - hasMoreElements() qui détermine si l'énumération contient d'autres éléments.
 - nextElement() qui récupère l'élément suivant au sein d'une énumération.
 - S'il n'y a plus d'élément une erreur est générée.
 - Pour éviter de rencontrer l'erreur on utilise la méthode précédente pour savoir s'il reste ou pas des éléments.
- **L'énumération n'a d'intérêt que s'il y a parcours du contenu d'un ensemble de données.**

Les itérateurs : Enumeration

- **Exemple :** Nous aurons donc, dans cet exemple, un Vecteur que nous remplissons avec des entiers

```
public class ExempEnum
{
    private static Vector unVecteur;

    private static void remplissage()
    {
        for (int i = 0; i < 5; i++)
        {
            unVecteur.addElement(new Integer(i));
        }
    }
}
```

```
...
public static void main(String[] args)
{
    //Instanciation
    unVecteur = new Vector();

    // Remplissage d'un vecteur d'entiers via une méthode
    remplissage();
    //Affichage à la console système du contenu Vecteur
    System.out.println("Affichage du contenu du Vector "
        + unVecteur);

    //Cumul des éléments du vecteur
    cumul();
    //Gestion par 1 Enumeration
    enumeration();
}
```

Les itérateurs : Enumeration

➤ Exemple : suite du code

```
//Cumul des éléments du vecteur sans énumération
private static void cumul()
{
    int somme = 0;
    for (int i = 0; i < unVecteur.size(); i++)
    {
        somme += (Integer)unVecteur.elementAt(i);
        //Version du JRE >= 5 la méthode intValue est sous-entendue
        // - la cast reste à faire car pas d'utilisation des génériques
    }
    System.out.println("Cumul des éléments: " + somme );
}
.../...
```

```
//Gestion du cumul avec 1 Enumeration
private static void enumeration()
{
    int somme = 0;
    Enumeration ensemble = unVecteur.elements();
    while (ensemble.hasMoreElements())
    {
        somme += (Integer)ensemble.nextElement();
        // Version du JRE >= 5
        // la méthode intValue est sous-entendue
    }
    System.out.println("Cumul des éléments: " + somme );
}
```

Résultats
console

```
Affichage du contenu du Vector [0, 1, 2, 3, 4]
Cumul des éléments: 10
Cumul des éléments: 10
```

Iterator

- Sur le même principe que l'Enumeration, le package propose aussi Iterator. Avec cette définition, il sera possible de supprimer l'objet de la collection, via la méthode remove(), après l'avoir lu.
- Iterator possède deux méthodes :
 - hasNext() qui renvoie true s'il y a encore au moins un élément à parcourir, false sinon,
 - next() qui renvoie le prochain élément à parcourir.
- Pour utiliser un Iterator, il suffit d'appeler la méthode iterator() sur la collection et de la parcourir.
- Cette méthode n'existe pas sur toutes les collections !

Iterator : exemple

```
package tests;
import java.util.ArrayList;
import java.util.Iterator;
public class TestIterator {
    public TestIterator() {}

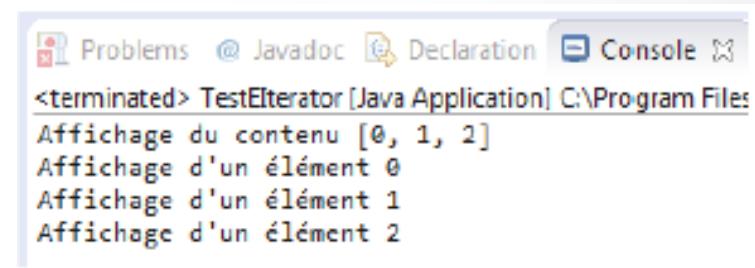
    public static void main(String[] args)
    {
        //Instanciation
        ArrayList<Integer> uneListe = new ArrayList<Integer>();

        // Rémpissage d'entiers
        for (int i = 0; i < 3; i++)
        {
            uneListe.add(i);
        }

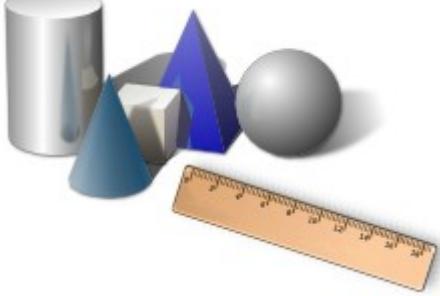
        //Affichage à la console système du contenu
        System.out.println("Affichage du contenu " + uneListe);

        // gestion via un iterateur
        Iterator iterator = uneListe.iterator();

        for (Integer element : uneListe) {
            System.out.println("Affichage d'un élément " + element);
        }
    }
}
```



```
Problems @ Javadoc Declaration Console
<terminated> TestIterator [Java Application] C:\Program Files
Affichage du contenu [0, 1, 2]
Affichage d'un élément 0
Affichage d'un élément 1
Affichage d'un élément 2
```



Exercice 3 suite (voir cahier d'exercices)

Enumeration / Concepts Objet

➤ GESTION BANCAIRE - VECTEURS :

- Exploitation de Vector via l'itérateur Enumeration
- Vous pouvez aussi écrire du code manipulant la boucle for (each)
- Le but est de vous familiariser avec Java et les collections. Notre Compte bancaire conserve tout de même ses caractéristiques, cependant vous pouvez écrire/tester du code pour cette familiarisation.

Compte
//Attributs
Vector somDepot ;
Vector somRetrait;
//Méthodes
Compte()
void depotDe(int m)
void retraitDe(int m)
int getSolde()

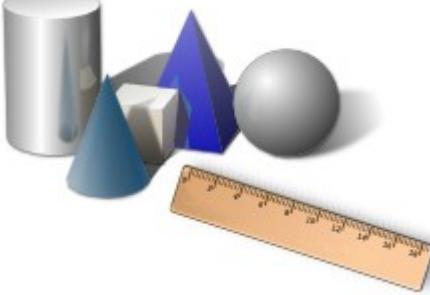
Objectifs du chapitre

- Les tableaux
- La classe String
- Les classes « ENVELOPPES »
- Les vecteurs
- Autres collections
- Les itérateurs
- **Les Tris**

Les Tris

- La classe « `java.util.Collections` » offre une méthode de classe qui permet de trier facilement toute liste constituée de `String`, `Date` ou de données primitives encapsulées dans leur classe enveloppe.
- Ceci est possible grâce à l'interface « `Comparable` » que toutes ces classes implémentent.
 - Cette interface ne définit qu'une seule méthode :
« `public int compareTo(Object o)` ; »
 - « `compareTo` » renvoie :
 - Un entier négatif si l'objet passé en paramètre est plus petit (ou passe avant dans l'ordre alphabétique ou dans l'ordre chronologique) que l'objet pour lequel elle est invoquée ;
 - Zéro, s'ils sont égaux ;
 - Un entier positif s'il est plus grand (ou passe après).

```
import java.util.*;  
public class Test {  
    public static void main (String [] args){  
        ArrayList liste = new ArrayList();  
        // Remplissage  
        liste.add ("ccc");  
        liste.add ("aaa");  
        liste.add ("bbb");  
  
        // Tri  
        Collections.sort(liste);  
    }  
}
```

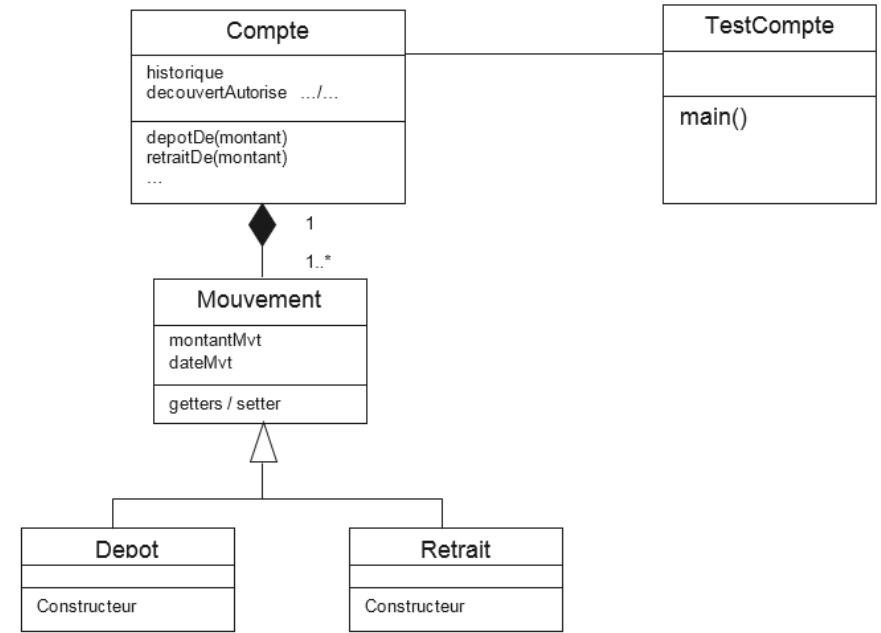


Exercice 4 (voir cahier d'exercices)

Concepts Objet (héritage, polymorphisme)

➤ GESTION BANCAIRE - NOTIONS D'OBJETS :

- Lien de composition : le Compte contient des objets Mouvement dans les Vector
- Lien de composition et un seul vecteur
- Lien d'héritage : Mouvement a 2 filles :
Depot Retrait
Vector contient des objets
Depot et/ou Retrait
- Héritage et polymorphisme

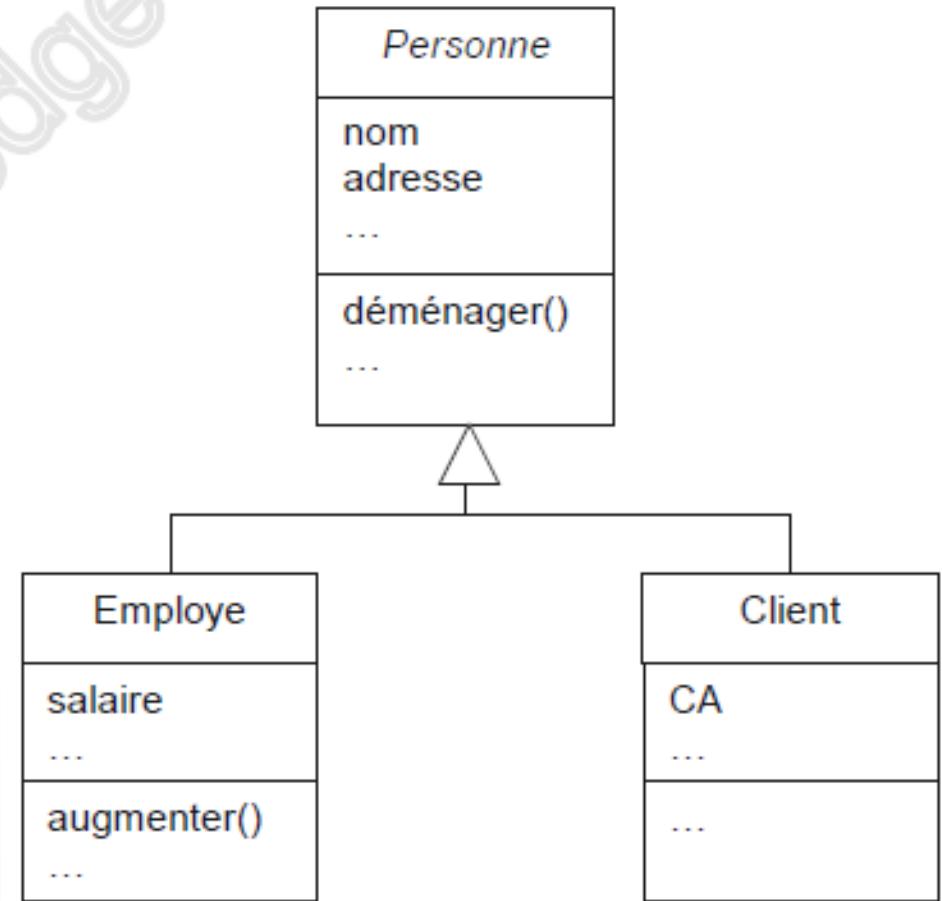


Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** **Classes Abstraites & Interfaces**
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Classes Abstraites : Intérêt

- Une classe **abstraite** est une classe qui ne peut être instanciée.
- En revanche, toutes ses filles pourront le cas échéant profiter par héritage de ses attributs et de ses méthodes.
- Par exemple, une classe « Personne », créée uniquement pour mettre en commun des attributs et des traitements utiles, aussi bien dans la classe « Employe » que dans la classe « Client » sera définie comme abstraite → *public abstract class Personne*



Classes Abstraites : Méthodes abstraites

- Une **méthode** abstraite est une méthode sans corps (sans traitement).
 - Elle sert à indiquer qu'il faudra déclencher un traitement, sans se préoccuper de comment sera réalisé ce traitement.
 - Elle correspond à une méthode dont on veut forcer l'implémentation dans toutes les classes filles.
- ➔ **Une classe doit être déclarée abstraite dès qu'une de ses méthodes est abstraite**

Classes Abstraites : exemples

- Par exemple, supposons une classe abstraite Image dont héritent des classes filles Gif, Jpeg, ...
- Cette classe mère déclare le prototype des méthodes pour compacter et restituer l'image → méthodes abstraites sans code java.
- Les classes filles sont obligées de donner un corps à ces méthodes, c'est à dire de fournir du code pour compacter et restituer l'image. S'il n'y a pas d'implémentation, la classe fille doit être abstraite

```
public abstract class Image
{
    ...
    public abstract void compacter();
    public abstract void restituer();
}

public class Gif extends Image
{
    ...
    public void compacter() { // traitement }
    public void restituer() { // traitement }
}

public class Jpeg extends Image
{
    ...
    public void compacter() { // traitement }
    public void restituer() { // traitement }
}
```

Classes Abstraites : exemples

- L'intérêt est de manipuler des objets de type Jpeg et/ou Gif comme étant des objets de type Image (la classe mère) et être sûr que les méthodes compacter() / restituer() sont applicables.
- Ainsi l'objet, vu comme une Image, s'il est de type Gif et que l'on utilise la méthode compacter(), on s'assure d'avoir le bon traitement.

```
public class Test
{
    public static void main(String [] args)
    {
        ArrayList liste = new ArrayList();

        liste.add (new Jpeg(...));
        liste.add (new Gif(...));
        ...

        Image image ;      // pas d'instanciation,
                           // mais on peut utiliser une référence
                           // de ce type

        for (int i = 0 ; i < liste.size() ; i++)
        {
            image = (Image) liste.get(i) ;
            image.compacter() ;
            // c'est bien le traitement défini dans la classe
            // de l'objet (Jpeg ou Gif) qui sera déclenché
        }
    }
}
```

Interfaces : intérêt

- Une interface est un ensemble de méthodes abstraites. Une interface n'est donc pas instanciable.
 - Une interface peut également définir des attributs, à condition qu'ils soient de classe (static) et constants (final). Autrement dit, les seuls attributs possibles d'une interface sont des constantes.
 - Toutes les classes qui implémentent une (ou des) interface(s) possèdent les méthodes et les constantes déclarées dans celle(s)-ci.
 - L'utilisation d'interfaces permet donc à plusieurs classes de partager un même protocole de comportement, sans avoir de lien d'héritage entre elles.
- Nous en avons bien vu l'avantage avec l'interface `java.util Enumeration` qui permet de parcourir avec la même syntaxe un `Vector`, une `HashTable`, ou toute autre collection d'objets.

Interfaces : Déclaration

- La déclaration s'effectue par le mot clé **interface** puis le nom de l'interface, qui termine souvent par *able* (être capable de).
- Devant le mot clé, on peut trouver le modificateur public : sinon, par défaut, une interface n'est visible que pour les classes définies à l'intérieur de son package.
- Par définition une interface est abstraite le modificateur **abstract** n'est donc pas nécessaire.
- Derrière le nom, on note l'héritage par le mot clé **extends** , une interface peut hériter de plusieurs interfaces (séparées par des virgules).

[Modificateurs] **interface** NomInterf **extends** Inteface1, Interface2...

Interfaces : Définition d'interfaces

- Par défaut, toutes les méthodes déclarées dans une interface sont abstraites, il est donc inutile de le préciser.

```
public interface Geometrie
{
    //Déclaration d'une constante
    static final double PI=3.14159 ;

    public interface Courbe extends Geometrie
    {
        // Courbe connaît donc PI
        // Déclaration de méthodes abstraites
        double longueur();
        void doubler();
    }

    public interface Surface extends Geometrie
    {
        // Surface connaît donc PI
        // Déclaration d'une méthode abstraite
        double surface();
    }
}
```

Interfaces : Implémentation d'une interface dans une classe

- Une classe qui implémente une ou plusieurs interfaces s'engage à définir toutes les méthodes déclarées dans les interfaces.
- Une classe qui n'implémente pas l'ensemble des méthodes de l'interface devient abstraite.
- La première génération de sous-classe qui termine l'implémentation de l'interface devient instanciable.

```
public class Rectangle extends GeometriePlane implements Courbe, Surface {  
    // Déclaration d'attributs  
    private double lg ;  
    // Implémentation des méthodes des interfaces  
    public double longueur()  
    { return lg ; }  
    public void doubler()  
    { //code }  
    public double surface()  
    { //code }  
}  
  
public class Disque extends GeometriePlane implements Courbe, Surface  
{  
    // Implémentation des méthodes des interfaces  
    public double longueur()  
    { //code }  
    public void doubler()  
    { //code }  
    public double surface()  
    { //code }  
}
```

Interfaces : Exemple

- Reprenons l'exemple de la classe Compte des exercices de la gestion bancaire

Classes filles

Classe Mère

```
package gestionBanqueDiffCpt;  
  
public class Compte  
{  
    //exercices de la gestion bancaire  
    .../...  
}
```

Interface

```
package gestionBanqueDiffCpt;  
  
public interface Remunerable  
{  
    public double calcullInteret(double taux);  
}
```

```
package gestionBanqueDiffCpt;  
  
public class PEL extends Compte implements Remunerable  
{  
    public PEL()  
    {  
        // TODO Auto-generated constructor stub  
    }  
  
    public double calcullInteret(double taux)  
    {  
        // TODO Auto-generated method stub  
        return (super.getSolde()* taux)/100;  
    }  
}
```

```
package gestionBanqueDiffCpt;  
  
public class Livret extends Compte implements Remunerable  
{  
    public Livret()  
    {  
        // TODO Auto-generated constructor stub  
    }  
  
    public double calcullInteret(double taux)  
    {  
        // TODO Auto-generated method stub  
        return (super.getSolde()* taux)/100;  
    }  
}
```

```

package gestionBanqueDiffCpt;
public class TestComptes
{
    static Compte cpt;
    static Livret livret;
    static PEL pel;
    public static void main(String[] args)
    {
        System.out.println("Complement Plusieurs comptes ");
        creationDesComptes();
        ensembleComptes();
        ensembleRemunerables();
    }

    private static void creationDesComptes()
    {
        // Un compte courant le Compte cpt est initialisé avec un découvert de 500
        cpt = new Compte(500);
        // on dépose/retire et affiche les valeurs
        cpt.depotDe(100);
        cpt.depotDe(200);

        // Un compte Plan épargne - On ne retire pas d'argent !
        pel = new PEL();
        pel.depotDe(1000);

        // Un compte livret on dépose on retire
        livret = new Livret();
        livret.depotDe(1000);
        livret.depotDe(1000);
    }
}

```

```

private static void ensembleComptes()
{
    // Manipulation des 3 instances comme étant des Comptes
    // Memorisation des objets dans un tableau de type Compte
    // Possible puisque tous objets sont des Comptes
    Compte comptes[] = { cpt, livret, pel };

    // Affichage de chaque solde de compte
    System.out.println("solde");
    for (Compte compte : comptes)
    {
        System.out.println(compte.getSolde());
    }

    // Appel sous entendu de toString
    System.out.println("Definition ");
    for (Compte compte : comptes)
    {
        System.out.println(compte);
    }
}

```

```

// Manipulation des instances non pas comme Compte
// mais comme un autre type => l'interface

private static void ensembleRemunerables()
{
    // Memorisation des objets dans un tableau de type Remunerable
    // Possible pour les objets implémentant l'interface
    Remunerable comptes[] = { livret, pel };

    // Affichage du calcul d'intérêt
    System.out.println("Interet 2% ");
    for (Remunerable compte : comptes) {
        System.out.println(compte.calculInteret(2));
    }
}

```

Résultat

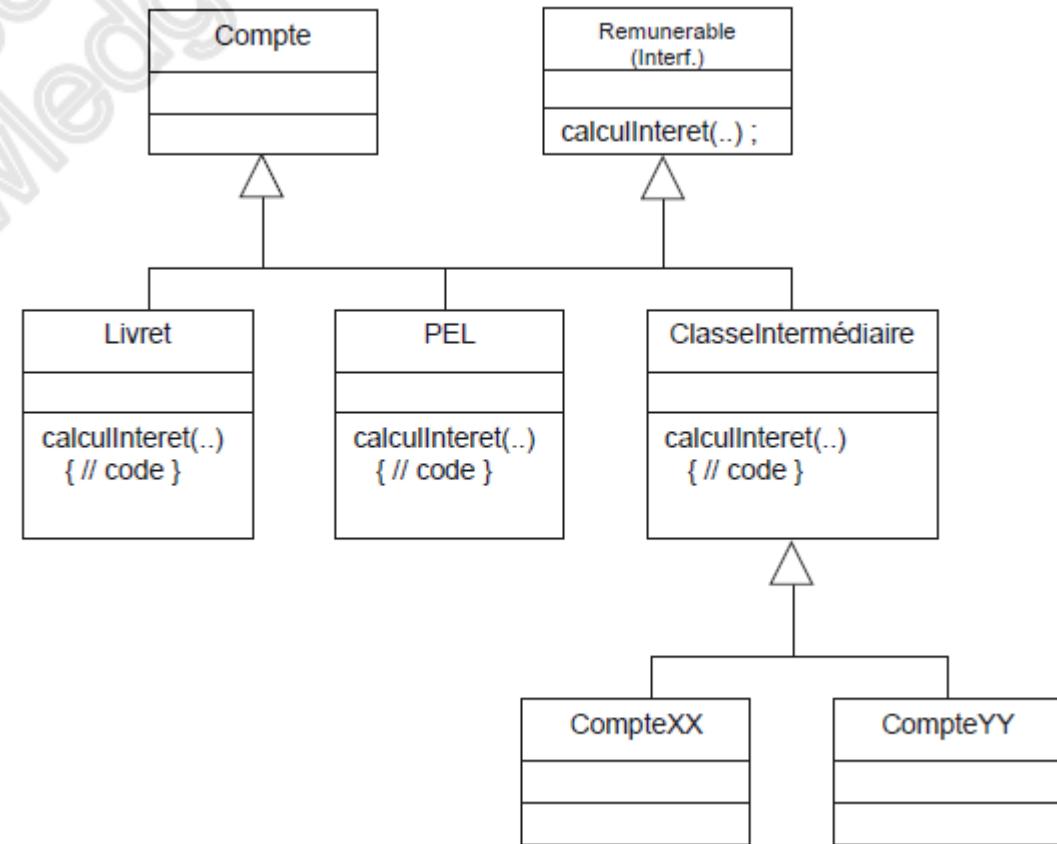
```

Console X
<terminated> TestComptes [Java Application] D:\Environnement JavaEE\java\jre1.5.0_11\bin\javaw.exe (8 févr. 2013 16:01:29)
Complement Plusieurs comptes
solde
0
1500
1000
Definition
Compte ayant un découvert autorisé de 500 le solde est 0 Historique [Depot Mouvement crée le Fri Feb 08 16:01:29 CET 2013, Montant =100, De
Compte ayant un découvert autorisé de 0 le solde est 1500 Historique [Depot Mouvement crée le Fri Feb 08 16:01:29 CET 2013, Montant =1000,
Compte ayant un découvert autorisé de 0 le solde est 1000 Historique [Depot Mouvement crée le Fri Feb 08 16:01:29 CET 2013, Montant =1000]
Interet 2%
30
20

```

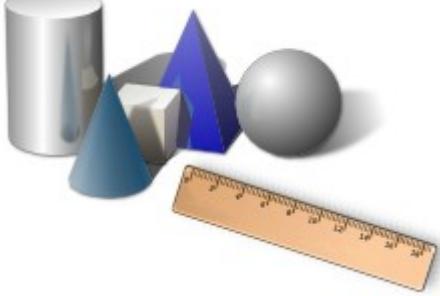
Limites

- Notons qu'à la différence de l'héritage, l'implémentation d'interfaces ne permet pas l'économie de code : on n'hérite dans ce cas que d'une contrainte (réaliser le traitement), pas du traitement lui-même.
- Exemple : si plusieurs classes de comptes à rémunération avaient le même mode de calcul, il faudrait créer une classe intermédiaire.



Conclusion sur les interfaces

- Un moyen d'écrire du code générique
- Une solution au problème de l'héritage multiple
- Un outil pour concevoir des applications réutilisables
- La force du polymorphisme



Exercice 5 (voir cahier d'exercices)

Concepts classe abstraite, override

- **GESTION BANCAIRE - COMPLEMENTS :**
 - Mouvement en classe abstraite
 - Redéfinition de `toString()` dans les classes filles de `Mouvement`
 - Compte – méthode pour retourner l'historique sous forme de chaîne de caractères
- **Nous manipulerons des interfaces dans d'autres exercices, si vous n'avez pas géré Enumeration, faites-le pour vous rendre compte de l'utilisation d'une interface en tant que type d'objet.**

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Intérêt
- Utilisation
- RuntimeException
- Définir ses exceptions

Prévoir les erreurs d'utilisation

- Certains cas d'erreurs peuvent être prévus à l'avance par le programmeur.
- Exemples :
 - Erreurs d'entrée-sortie (I/O fichiers)
 - Erreurs de saisie de données par l'utilisateur
- Le programmeur peut :
 - «Planter» le programme à l'endroit où l'erreur est détectée
 - Manifester explicitement le problème à la couche supérieure
 - Tenter une correction

Notion d'exception

- En Java, les erreurs se produisent lors d'une exécution sous la forme d'exceptions.
- Une exception :
 - Est un objet, instance d'une classe qui hérite d'Exception
 - Provoque la sortie d'une méthode
 - Correspond à un type d'erreur
 - Contient des informations sur cette erreur

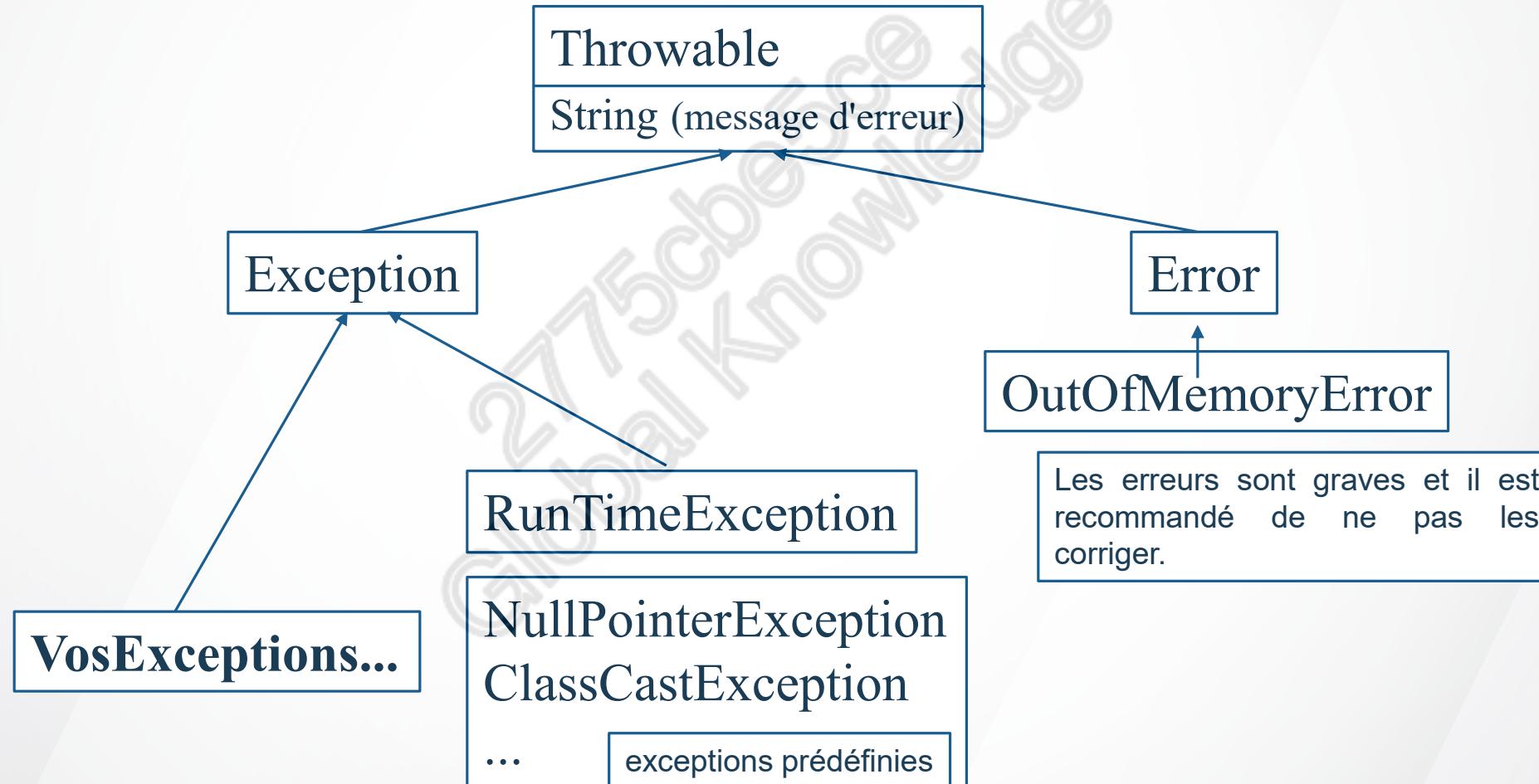
Les exceptions

- Exception
 - Situation particulière imposant une rupture dans le cours d'un programme : erreur, impossibilité...
 - Un Objet **JAVA Exception** est une « bulle » logicielle produite dans une situation qui va remonter la pile d'exécution pour trouver une portion de code apte à la traiter.
 - Si cette portion n'existe pas, le programme s'arrête en affichant la pile d'exécution. Sinon, la portion de code sert à pallier au problème (poursuite éventuelle du traitement, ou sortie = arrêt).

Terminologie

- Une **exception** est un signal qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution.
- Deux solutions alors :
 - Laisser le programme se terminer avec une erreur,
 - Essayer, malgré l'exception, de continuer l'exécution normale.
- Lever une exception consiste à signaler quelque chose d'exceptionnel.
- Capturer l'exception consiste à signaler qu'on va la traiter.

Arbre des exceptions



Nature des exceptions

- En Java, les exceptions sont des objets ayant 3 caractéristiques :
 - Un type (Héritant de la classe *Exception*)
 - Une chaîne de caractères (option).
 - Un « instantané » de la pile d'exécution au moment de la création.
- Les exceptions construites par l'utilisateur étendent la classe *Exception*
- *RunTimeException*, *Error* sont des exceptions et des erreurs prédéfinies et/ou gérées par Java.

Quelques exceptions prédéfinies en Java

- Division par zéro pour les entiers : `ArithmeticException`
- Référence nulle : `NullPointerException`
- Tentative de forçage de type illégal : `ClassCastException`
- Tentative de création d'un tableau de taille négative :
`NegativeArraySizeException`
- Dépassemement de limite d'un tableau : `ArrayIndexOutOfBoundsException`

Des exceptions pour écrire un code fiable

- Java exige qu'une méthode susceptible de lever une exception (hormis les **Error** et les **RuntimeException**) indique quelle doit être l'action à réaliser.
 - Sinon, il y a erreur de compilation.
- Le programmeur a le choix entre :
 - Ecrire un bloc **try / catch** pour traiter l'exception,
 - Laisser remonter l'exception au bloc appelant grâce à un **throws**.
- C'est ce qu'on appelle : "Déclarer ou traiter".

Capture d'une exception

- Les sections **try** et **catch** servent à capturer une exception dans une méthode (attraper la bulle...)
- Exemple :

```
public void XXX(.....) {  
    try{ ..... }  
    catch {  
        .....  
    }  
}
```

Si une erreur
se produit
ici....

Elle sera récupérée là.

try / catch / finally

```
try
{
    ...
}
catch (<une-exception>)
{
    ...
}
catch (<une_autre_exception>)
{
    ...
}
...
finally
{
    ...
}
```

- ➊ Autant de blocs **catch** que l'on veut.
- ➋ Bloc **finally** facultatif.

Traitements des exceptions (1)

- Le bloc `try` est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- Dans ce dernier cas, les clauses `catch` sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exception (ou une superclasse).
- Les clauses `catch` doivent donc traiter les exceptions de la plus spécifique à la plus générale.
 - La présence d'une clause `catch` qui intercepte une classe d'exception avant une clause qui intercepte une sous-classe d'exceptions déclenche une erreur de compilation.
- Si une clause `catch` convenant à cette exception a été trouvée alors le bloc de ce `catch` est exécuté, l'exécution du programme reprend son cours après ce `catch`.

Traitement des exceptions (2)

- Si elles ne sont pas immédiatement capturées par un bloc `catch`, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode `main()`, ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- L'interpréteur Java affiche un message identifiant :
 - L'exception,
 - La méthode qui l'a causée,
 - La ligne correspondante dans le fichier.

Bloc finally

- L'écriture d'un bloc **finally** permet au programmeur de définir un ensemble d'instructions qui est toujours exécuté, que l'exception soit levée ou non, capturée ou non.
- Le bloc **finally** s'exécute même si le bloc en cours d'exécution (**try** ou **catch** selon les cas) contient un **return**, un **throw**, un **break** ou un **continue**. Dans ce cas, le bloc **finally** est exécuté juste avant le branchement effectué par l'une de ces instructions.
- La seule instruction qui peut faire qu'un bloc **finally** ne soit pas exécuté est **System.exit()**.

Interception vs propagation

- Si une méthode peut émettre une exception (ou appelle une autre méthode qui peut en émettre une) il faut :
 - Soit **propager** l'exception (la méthode doit l'avoir déclarée);
 - Soit **intercepter** et traiter l'exception.

Exemple de propagation

```
public int ajouter(int a, String str) throws NumberFormatException  
    int b = Integer.parseInt(str);  
    a = a + b;  
    return a;  
}
```

Exemple d'interception

```
public int ajouter(int a, String str) {  
    try {  
        int b = Integer.parseInt(str);  
        a = a + b;  
    } catch (NumberFormatException e) {  
        System.out.println(e.getMessage());  
    }  
    return a;  
}
```

Les objets Exception

- La classe **Throwable** définit un message de type **String** qui est hérité par toutes les classes d'exception.
- Ce champ est utilisé pour stocker le message décrivant l'exception.
- Il est positionné en passant un argument au constructeur.
- Ce message peut être récupéré par la méthode **getMessage()**.

Exemple

```
public class MonException extends Exception
{
    public MonException()
    {
        super();
    }
    public MonException(String s)
    {
        super(s);
    }
}
```

Levée d'exception

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé `throw`.
- `throw` prend en paramètre un objet instance de `Throwable` ou d'une de ses sous-classes.
- Les objets `Exception` sont souvent alloués dans l'instruction même qui assure leur lancement.

```
throw new MonException("Mon exception s'est produite !!!");
```

Emission d'une exception

- L'exception elle-même est levée par l'instruction `throw`.
- Une méthode susceptible de lever une exception est identifiée par le mot-clé `throws` suivi du type de l'exception.
- Exemple :

```
public void ouvrirFichier(String name) throws MonException
{
    if (name==null) throw new MonException("Le nom du fichier doit être renseigné !");
    else
    {
        ...
    }
}
```

throws (1)

- Pour "laisser remonter" à la méthode appellante une exception qu'il ne veut pas traiter, le programmeur ajoute le mot réservé **throws** à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
{
    // ne traite pas l'exception IOException
    // mais est susceptible de la générer
}
```

throws (2)

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée.
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,).
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans les méthodes qu'elle appelle (et ceci récursivement).

Objectifs du chapitre

- Intérêt
- Utilisation ..
- Runtimeexception
- **Définir ses exceptions**

Définir ses exceptions

- Pour créer ses propres exceptions, il suffit de créer une classe qui hérite de « Exception »

```
public class ClientException extends Exception
{
    // Des attributs sont possibles
    // Des méthodes sont possibles
    // Le constructeur contenant le message d'erreur
    public ClientException (String message)
    {
        super(message);
    }
}
```

Définir ses exceptions

- **Déclenchement :** Pour lever une exception applicative on doit instancier, lors de l'instruction « throw », un objet de la classe créée.

```
public class Client
{
    ...
    public void isBonPayeur() throws ClientException
    {
        if(getSolde() < 0)
        {
            // Création de l'erreur :
            throw new ClientException("mauvais payeur ! ");
        }
    }
}
```

L'utilisateur de cette méthode sera obligé de gérer l'erreur

Définir ses exceptions

- Différents cas de test : Gestion de l'erreur via du try/catch

```
package lesConcepts;
public class TestClient1
{
    public static void main(String[] args)
    {
        Client c =new Client("Dupont");
        System.out.println("Création de : " + c.getNom());
        c.setSolde(-10);
        try
        {
            c.isBonPayeur();
            System.out.println("suite du code si pas d'erreur");
        }
        catch (ClientException e)
        {
            System.out.println(e.getMessage());
        }
        c.setSolde(50);
        try
        {
            c.isBonPayeur();
            System.out.println("suite du code car pas d'erreur");
        }
        catch (ClientException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Résultats Console :

```
Création de : Dupont
mauvais payeur !
suite du code car pas
d'erreur
```

Définir ses exceptions

➤ Différents cas de test : Sans gestion de l'erreur via un throws

```
package lesConcepts;  
public class TestClient2  
{  
    public static void main(String[] args) throws ClientException  
    {  
        Client c = new Client("Dupont");  
        c.setSolde(-10);  
        System.out.println("Création de : " + c.getNom());  
        c.isBonPayeur();  
        c.setSolde(50);  
        c.isBonPayeur();  
        System.out.println("suite du code car pas d'erreur");  
    }  
}
```

Plantage à l'appel de la méthode et pas d'exécution de la suite.

Résultats Console :

```
Création de : Dupont  
Exception in thread "main" lesConcepts.ClientException: mauvais payeur !  
at lesConcepts.Client.isBonPayeur(Client.java:17)  
at lesConcepts.TestClient.main(TestClient.java:13)
```

Définir ses exceptions

- Différents cas de test : Un mixte des 2 : avec try / catch et avec un throws

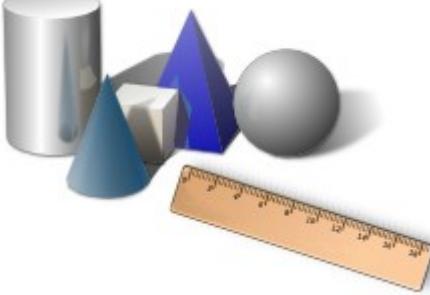
```
package lesConcepts;
public class TestClient3
{
    public static void main(String[] args) throws ClientException
    {
        Client c = new Client("Dupont");
        c.setSolde(-10);
        System.out.println("Création de :" + c.getNom());
        try
        {
            c.isBonPayeur();
        }
        catch (ClientException e)
        {
            System.out.println(e.getMessage());
        }
        c.setSolde(50);
        c.isBonPayeur();
        System.out.println("suite du code car pas d'erreur");
    }
}
```

Résultats Console :
Création de : Dupont
mauvais payeur !
suite du code car pas d'erreur

Pas d'obligation de try/catch
car la méthode à un throws

Conclusion

- Les exceptions rendent la gestion des erreurs **plus simple et plus lisible**.
- Le code pour gérer les erreurs peut être **regroupé en un seul endroit** : là où on a besoin de traiter l'erreur.
- Possibilité de **se concentrer sur l'algorithme** plutôt que de s'inquiéter à chaque instruction de ce qui peut mal fonctionner,
- Les erreurs **remontent la hiérarchie d'appels** grâce à l'exécutif du langage et non plus grâce à la bonne volonté des programmeurs.



Exercice 6 (voir cahier d'exercices)

Exception

➤ GESTION BANCAIRE – EXCEPTIONS

Nous souhaitons gérer l'impossibilité de retirer un certain montant par un objet Exception et non pas par un booléen ou une chaîne de caractères.

- Création d'une classe CompteException
- **La méthode retraitDe(int montant) doit créer un objet de type CompteException et propager l'erreur.**
- Vos tests doivent manipuler :
 - L'interception de CompteException et l'affichage du message d'erreur.
 - La relégation de l'erreur à l'appelant avec l'arrêt du traitement
 - Un mixte des deux précédentes utilisations

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** **Les génériques**
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Le polymorphisme paramétrique
- Paramètres de type, les génériques
- Les classes génériques
- Les méthodes génériques
- Les paramètres de type bornés
- Les génériques et le sous-typage
- Les génériques et les jokers
- Les génériques et l'effacement du type
- Les génériques et le transtypage

Le polymorphisme paramétrique

- Le polymorphisme paramétrique: possibilité pour une fonction ou un type d'être écrit de façon à ce qu'il traite les valeurs de manière identique sans dépendre de la connaissance de leurs types.
- Une telle fonction ou type s'appelle une fonction générique ou un type de données.
- Introduit pour la première fois en langue ML en 1976.
- Fait maintenant partie de nombreux autres langages (Java, C #, Delphi).
- Motivation : le polymorphisme paramétrique permet d'écrire du code général flexible sans sacrifier la sécurité du type.
- Le plus couramment utilisé en Java avec des collections.

Les collections avant Java 1.5

- Les collections Java initiales stockaient des valeurs de type Object.
- Elles pouvaient stocker n'importe quel type, car tous les types sont des sous-classes d'Object.
- Mais vous deviezcaster les résultats, ce qui était fastidieux et source d'erreurs.
- L'examen des éléments d'une collection n'était pas digne de confiance.

Les collections avant Java 1.5

// En Java 1.4:

```
ArrayList nomsEtudiant = new ArrayList();
nomsEtudiant.add("Jean");
nomsEtudiant.add("Alain");
String nomEtudiant = (String) nomsEtudiant.get(0);
```

// Le code compile,

// mais il génère une exception au moment de l'exécution

```
Point point = (Point) nomsEtudiant.get(1);
```

Paramètres de type, les génériques



Paramètres de type, les génériques

- Depuis Java 1.5, lors de la construction de java.util.ArrayList, nous spécifions le type d'éléments que la liste contiendra entre < et >.

```
ArrayList<Type> nomsEtudiant= new ArrayList<Type>();
```

- Nous disons que la classe ArrayList accepte un paramètre de type, ou qu'il s'agit d'une classe générique.
- L'utilisation d'un "ArrayList" de type "brut" sans <> entraîne des avertissements du compilateur.

Paramètres de type, les génériques

```
ArrayList<String> nomsEtudiant = new ArrayList<String>();  
nomsEtudiant.add("Jean");  
nomsEtudiant.add("Alain");  
//Il n'y a plus besoin de transtypage  
String nomEtudiant = nomsEtudiant.get(0);  
  
// Erreur de compilation  
Point point= (Point) nomsEtudiant.get(1);
```

Les classes génériques

```
// Classe générique  
public class name<Type> {  
ou  
public class name<Type, Type, ..., Type> {
```

- En plaçant le type entre <>, vous exigez que tout client qui construit votre objet fournisse un paramètre de type.
- Vous pouvez exiger plusieurs paramètres de type séparés par des virgules.

Les classes génériques

- Le reste du code de votre classe peut faire référence à ce type par son nom.
 - La convention est d'utiliser un nom d'une lettre tel que: T pour Type, E pour Element, N pour Number, K pour Key ou V pour Value, etc.
- Le paramètre type est instancié par le client. (par exemple, E → String)

Instanciation des classes génériques

- Vous ne pouvez pas créer d'objets ni de tableaux d'un type paramétré.

```
public class Foo<T> {  
    // Déclaration ok  
    private T myField;  
    // Déclaration ok  
    private T[] myArray;  
    public Foo(T param) {  
        // erreur de compilation  
        myField = new T();  
        // erreur de compilation  
        myArray = new T[10];  
    }  
}
```

Instanciation des classes génériques

- Vous pouvez créer des variables d'un type paramétré, les accepter en tant que paramètres, les renvoyer ou créer des tableaux en convertissant Object [].
- La conversion en types génériques n'est pas sécurisée contre le type, elle génère donc un avertissement.

...

```
private T myField;  
private T[] myArray;  
public Foo(T param) {  
    myField = param;  
    myArray = (T[]) (new Object[10]);  
}
```

...

La comparaison des objets génériques

- Lors de la comparaison des objets de type E, il faut utiliser la méthode equals :

```
public class ArrayList<E> {  
    ...  
    public int indexOf(E value) {  
        for (int i = 0; i < size; i++) {  
            if (elementData[i].equals(value)) {  
                return i;  
            }  
        }  
        return -1;  
    }  
}
```

Les interfaces génériques

```
public interface List<E> {  
    public void add(E value);  
    public void add(int index, E value);  
    public E get(int index);  
    public int indexOf(E value);  
    public boolean isEmpty();  
    public void remove(int index);  
    public void set(int index, E value);  
    public int size();  
}
```

```
public class ArrayList<E> implements List<E> { ...  
public class LinkedList<E> implements List<E> { ...
```

Les méthodes génériques

```
public static <Type> returnType nomMethode(params) {...}
```

- Lorsque vous voulez créer une seule méthode générique (souvent statique) dans une classe, faites précéder son type de retour par son paramètre type.

```
public class Collections {
```

```
...
```

```
public static <T> void copy(List<T> dst, List<T> src) {
```

```
    for (T t : src) {
```

```
        dst.add(t);
```

```
}
```

```
...
```

Les paramètres de type bornés

- Une limite supérieure de type borné, accepte un super-type donné ou l'un de ses sous-types :
<Type extends SuperType>
- Fonctionne pour plusieurs superclasses / interfaces avec & :
<Type extends ClassA & InterfaceB & InterfaceC & ...>
- Une limite inférieure de type borné, accepte un super-type donné ou l'un de ses super-types.
<Type super SuperType>

Les paramètres de type bornés

- Exemple :

```
// TreeSet fonctionne pour tout type comparable  
public class TreeSet<T extends Comparable<T>> {  
    ...  
}
```

Les génériques et le sous-typage

- List <String> est-il un sous-type de List <Object> ?
- Set<Lion> est-il un sous-type de Collection<Animal> ?
- La réponse est non, cela violerait le principe de substitution de Liskov.
- Si nous pouvions passer un Set <Lion> à une méthode en attente d'un Collection <Animal>, cette méthode pourrait ajouter d'autres animaux dans la collection.

```
Set<Lion> set1 = new HashSet<Lion>();  
Set<Animal> set2 = set1;      // Interdit  
...  
set2.add(new Zebre());  
Lion lion = set1.get(0);      // Erreur
```

Les génériques et les jokers

- ? indique un paramètre de type joker, qui peut être n'importe quel type.

`List <?> liste ; // Prendra n'importe quoi`

- La différence entre `List <?>` Et `List <Object>`:

- ? peut devenir n'importe quel type particulier; l'objet est juste un tel type.
- Lister <Object> est restrictif; ne prendrait pas de liste <String>

Ex : `List<String> listChaine = new ArrayList<String>(); List<?> list2 = new ArrayList ();`
`list2.addAll(listChaine) ; // Incorrect car list2 ne sait pas ce qu'elle attend, elle ne peut pas traiter les objets`

- La différence entre `List <Foo>` et `List <? extends Foo>`:

- `List <? extends Foo>` se lie à un sous-type particulier de Foo et ne permet que cela.
- Par exemple : `List <? extends Animal>` ne stocke que les objets Lion mais pas les objets Zebre.

Les génériques et les jokers

- List <Foo> permet de stocker tout ce qui est un sous-type de Foo dans la même liste.
- Par exemple: List <Animal> peut stocker des Lions et des Zebres.

Les génériques et l'effacement du type

- Tous les types génériques deviennent du type Object une fois compilé.
- Une des raisons : assurer la compatibilité ascendante avec l'ancien byte code.
- Au moment de l'exécution, toutes les instances génériques ont le même type.

```
List <String> lst1 = new ArrayList <String> ();  
List <Integer> lst2 = new ArrayList <Integer> ();  
lst1.getClass () == lst2.getClass () // retourne true
```

Les génériques et l'effacement du type

- Vous ne pouvez pas utiliser instanceof pour découvrir un paramètre de type :

```
Collection <?> cs = new ArrayList <String>();
```

```
if (cs instanceof Collection <String>) {  
    // instruction non autorisée  
}
```

Les génériques et le transtypage

- La conversion en type générique génère un avertissement.

```
List <?> lst = new ArrayList <String> (); // Instruction valide
```

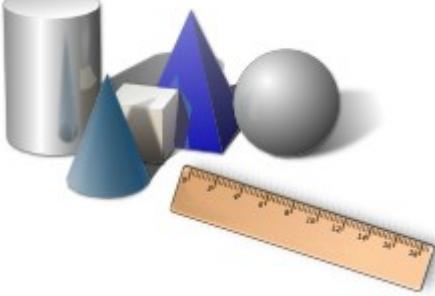
```
List <String> ls = (List <String>) lst; //Génère un avertissement
```

- Le compilateur émet un avertissement non vérifié, car ce n'est pas quelque chose que le système d'exécution va vérifier pour vous.
- Habituellement, si vous pensez avoir besoin de le faire c'est que vous n'avez pas choisi la bonne manière pour implémenter le code.

Les génériques et le transtypage

- Il en va de même pour les variables de type générique :

```
public static <T> T badCast (T t, Object o) {  
    // Génère un avertissement non vérifié  
    return (T) o;  
}
```



Exercice 7 (voir cahier d'exercices) Génériques

➤ GESTION BANCAIRE - Evolution :

- Utilisation des génériques : votre Vector historique ne doit accepter que des objets de type Mouvement
- Mettez en œuvre une méthode qui retourne l'historique sous forme de List<Mouvement>
- Mouvement - Agio : ajouter une classe fille Agio à Mouvement, redéfinissez les méthodes adéquates. Manipuler cette classe Agio dans Compte

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** **Les expressions lambda**
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Introduction
- La programmation fonctionnelle en Java
- Le Lambda calcul en Java
- Les interfaces fonctionnelles
- Références de méthode

Introduction

- Java SE 8 a ajouté le support de la programmation fonctionnelle.
- Les nouvelles fonctionnalités de langage et de bibliothèque prenant en charge la programmation fonctionnelle ont été ajoutées à Java dans le cadre du Project Lambda.
- Ce chapitre présente de nombreux exemples de programmation fonctionnelle, montrant souvent des manières plus simples d'implémenter les tâches que vous aviez fait jusque là.

La programmation fonctionnelle

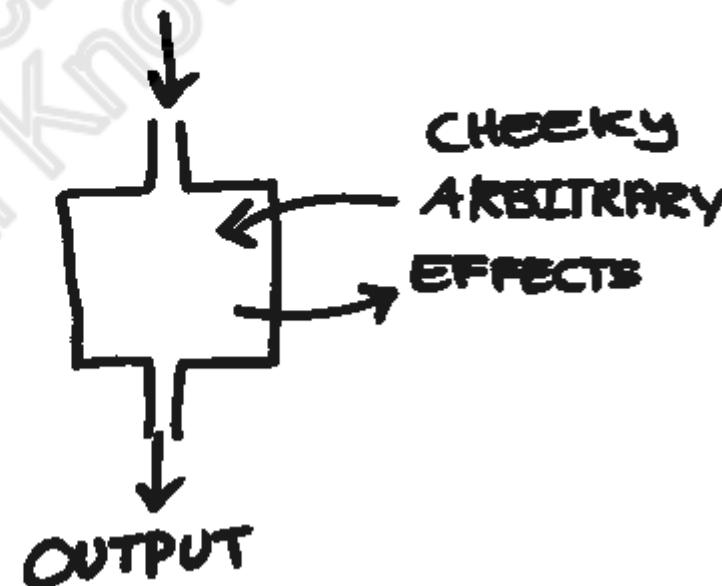
Functions

INPUT



Procedures

INPUT



La programmation fonctionnelle

- Avant la programmation fonctionnelle, vous avez généralement déterminé ce que vous vouliez accomplir, puis spécifié les étapes précises à suivre pour accomplir cette tâche.
- Pour implémenter une itération :
 - Utilisation d'une boucle pour parcourir une collection d'éléments.
 - Nécessite d'accéder aux éléments de manière séquentielle.
 - Requiert l'utilisation des variables mutables.

La programmation fonctionnelle

- En programmation fonctionnelle vous allez spécifiez ce que vous voulez accomplir dans une tâche, mais pas comment l'accomplir.
- Pour implémenter une itération interne :
 - Laissez la bibliothèque déterminer comment itérer sur une collection d'éléments connus.
- La programmation fonctionnelle met l'accent sur l'immuabilité, sans modifier la source de données en cours de traitement ou tout autre état du programme.

La programmation fonctionnelle

- La programmation fonctionnelle est un style de programmation qui considère le calcul comme étant l'évaluation de fonctions mathématiques.
- Traite les données comme étant immuables.
- Les fonctions peuvent prendre des fonctions comme arguments et renvoyer des fonctions comme résultats.

Intérêt de la programmation fonctionnelle

- Permet d'écrire des programmes plus compréhensibles, plus déclaratifs et plus concis que les programmes impératifs.
- Nous permet de nous concentrer sur le problème plutôt que sur le code.
- Facilite le parallélisme.
- Facile à exécuter les tests unitaires et le débogage du code.
- Permet le déploiement de code à chaud et tolérance aux pannes.
- Augmentation de la productivité du développeur.
- Prise en charge des fonctions imbriquées.
- Permet l'utilisation efficace du Lambda Calcul.

Programmation impérative vs programmation déclarative

- Programmation impérative: est un style de programmation dans lequel vous programmez l'algorithme avec un flux de contrôle et des étapes explicites. Pensez à un algorithme pseudo-code avec une logique de branchement et de boucle.
- Programmation déclarative : est un style de programmation où vous déclarez ce qui doit être fait sans se soucier du flux de contrôle.
- Programmation fonctionnelle : est un paradigme de programmation déclarative qui traite le calcul comme une série de fonctions et évite les données d'état mutables pour faciliter la concurrence et notamment le parallélisme.

Programmation fonctionnelle vs programmation orientée objet

Programmation fonctionnelle	Programmation orientée objet
Utilise des données immuables.	Utilise des données mutables.
Suit le modèle de programmation déclarative.	Suit le modèle de programmation impératif.
Se concentre sur: "Ce que vous faites dans le programme."	Se concentre sur "Comment vous faites votre programmation."
Prend en charge la programmation parallèle.	Pas de support pour la programmation parallèle.
Le contrôle de flux est effectué à l'aide d'appels de fonction et d'appels de fonction avec récursivité.	Le processus de contrôle de flux est effectué à l'aide de boucles et d'instructions conditionnelles.
L'ordre d'exécution des instructions n'est pas très important.	L'ordre d'exécution des instructions est important.
Prend en charge à la fois "Abstraction sur les données" et "Abstraction sur le comportement".	Prend uniquement en charge "Abstraction sur les données".

Programmation fonctionnelle vs programmation orientée objet

➤ Exemple de code

- Soit une liste suivante : `ArrayList<String> list1 = new ArrayList<String>();`
- Contenant 5 éléments A, B, C, D, E
- Nous souhaitons afficher le contenu de la liste

Programmation impérative	Programmation fonctionnelle
il faut deux traitements 1. la boucle 2. l'affichage de l'élément	Le code de la boucle est factorisé dans une méthode qui attend en paramètre une fonction qui détient le traitement à réaliser pour chaque élément
<pre>for (String string : list1) { System.out.println(string); }</pre>	<pre>list1.forEach(System.out::println);</pre> <p style="text-align: center;">//Avec 1 expression lambda <pre>list1.forEach(x -> System.out.println(x));</pre></p>

Nouvelles fonctionnalités Java 8 :

- Des méthodes par défaut `foreach()` sur `Iterable`
- Des **interfaces fonctionnelles** ici **Consumer** (paramètre attendu de `forEach`)
- Des expressions **lambda** pour permettre de passer en paramètre une fonction, ici sous la forme d'une **référence de méthode**

Le Lambda calcul



Le Lambda calcul

- Le Lambda calcul a été introduit dans les années 1930 par Alonzo Church en tant que système mathématique permettant de définir des fonctions calculables.
- Le Lambda calcul est équivalent en pouvoir de définition à celui des machines de Turing.
- Le Lambda calcul sert de modèle informatique sous-jacent aux langages de programmation fonctionnels tels que Lisp, Haskell et Ocaml.
- Des fonctionnalités du calcul Lambda telles que les expressions Lambda ont été incorporées dans de nombreux langages de programmation largement utilisés tels que C ++ et Java.

Le Lambda calcul

- Le concept central du Lambda calcul est une expression générée par la grammaire suivante qui peut désigner une définition de fonction, une application de fonction, une variable ou une expression entre parenthèses :

$$\text{expr} \rightarrow \lambda \text{ var. } \text{expr} \mid \text{expr expr} \mid \text{var} \mid (\text{expr})$$

- Nous pouvons penser à une expression de lambda-calcul comme à un programme qui, lorsqu'il est évalué par des bêta-réductions, renvoie un résultat consistant en une autre expression de Lambda calcul.

Exemple d'expression Lambda

- L'expression Lambda suivante $\lambda x. (+ x 1) 2$ représente l'application d'une fonction $\lambda x. (+ x 1)$ avec un paramètre formel x et un corps $+ x 1$ pour l'argument 2.
- Remarquez que la définition de la fonction $\lambda x. (+ x 1)$ n'a pas de nom; c'est une fonction anonyme.
- En Java 8, nous représenterions cette définition de fonction par l'expression lambda Java 8 :

$x \rightarrow x + 1$

Les expressions Lambda en Java



Les expressions Lambda en Java

- Une expression Lambda en Java 8 est une méthode sans déclaration, généralement sous la forme **(paramètres) -> {body}**.
- Exemples :
 - (int x, int y) -> {return x + y; }
 - x -> x * x
 - () -> x
- Une expression Lambda peut avoir zéro paramètre ou plus, séparés par des virgules, et leur type peut être explicitement déclaré ou déduit du contexte.
- Les parenthèses ne sont pas nécessaires autour d'un paramètre unique.

Les expressions Lambda en Java

- () est utilisé pour indiquer zéro paramètre.
- Le corps peut contenir zéro ou plusieurs instructions.
- Les accolades ne sont pas nécessaires autour d'un corps à déclaration unique.

Intérêt des expressions Lambda pour Java

- Ajout du support de la programmation fonctionnelle.
- Permettre l'écriture de code plus compact plus maigre.
- Faciliter la programmation parallèle.
- Développement d'API plus génériques, flexibles et réutilisables.
- Permettre la transmission des comportements ainsi que des données à des fonctions.

Exemples d'expression Lambda en Java

- Afficher une liste d'entiers avec une expression Lambda :

```
List<Integer> intSeq = Arrays.asList(1,2,3);
```

```
intSeq.forEach(x -> System.out.println(x));
```

- **x -> System.out.println (x)** est une expression Lambda qui définit une fonction anonyme avec un paramètre nommé x de type Integer et qui permet d'afficher x dans la sortie standard.

Exemples d'expression Lambda en Java

- Une expression Lambda multi-lignes :

```
List<Integer> intSeq = Arrays.asList(1,2,3);
```

```
intSeq.forEach(x -> {  
    x += 2;  
    System.out.println(x);  
});
```

- Des accolades sont nécessaires pour entourer le corps multi-lignes dans une expression Lambda.

Exemples d'expression Lambda en Java

- Une expression Lambda avec une variable locale définie :

```
List<Integer> intSeq = Arrays.asList(1,2,3);
```

```
intSeq.forEach(x -> {  
    int y = x * 2;  
    System.out.println(y);  
});
```

- Comme pour les fonctions ordinaires, vous pouvez définir des variables locales dans le corps d'une expression lambda.

Exemples d'expression Lambda en Java

- Une expression Lambda avec un type de paramètre déclaré :

```
List<Integer> intSeq = Arrays.asList(1,2,3);
```

```
intSeq.forEach((Integer x -> {  
    x += 2;  
    System.out.println(x);  
}));
```

- Vous pouvez, si vous le souhaitez, spécifier le type de paramètre.

Compilateur Java et expressions Lambda

JDK8



Compilateur Java et expressions Lambda

- Le compilateur Java 8 convertit d'abord une expression lambda en une fonction.
- Il appelle ensuite la fonction générée.
- Par exemple, `x -> System.out.println (x)` peut être converti en une fonction statique générée.

```
public static void genName (Integer x) {  
    System.out.println (x);  
}
```

Compilateur Java et expressions Lambda

- Mais quel type doit être généré pour cette fonction ?
- Comment devrait-il s'appeler ?
- Dans quelle classe devrait-il aller ?



Interfaces fonctionnelles

- Java 8 a défini de nombreuses **interfaces fonctionnelles à utiliser de manière extensive dans les expressions lambda**.
- Décision de conception:
 - Les lambdas Java 8 sont affectées à des interfaces fonctionnelles.
 - Les lambdas Java 8 devraient fonctionner avec le code Java existant sans nécessiter de recompilation.
- Une interface fonctionnelle est une interface Java avec exactement 1 seule méthode abstraite.
- L'annotation `@FunctionalInterface` est ajoutée afin que nous puissions marquer une interface en tant qu'interface fonctionnelle.

Interfaces fonctionnelles

- Par exemple :

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
}
```

- Il n'est pas obligatoire d'utiliser `@FunctionalInterface`, mais il est recommandé de l'utiliser avec des interfaces fonctionnelles pour éviter d'ajouter accidentellement des méthodes supplémentaires.
- Si l'interface est annotée avec l'annotation `@FunctionalInterface` et que nous essayons d'avoir plus d'une méthode abstraite, elle génère une erreur du compilateur.

Interfaces fonctionnelles

- L'API Collections de Java 8 a été réécrite et une nouvelle API de flux utilisant de nombreuses interfaces fonctionnelles est introduite.
- Java 8 a défini de nombreuses interfaces fonctionnelles dans le package `java.util.function`.
- Les interfaces fonctionnelles Java 8 sont notamment les suivantes : `Function`, `Consumer`, `Supplier` et `Predicate` ; avec quelques interfaces filles sur ces dernières.

Des exemples de code, vous sont fournis, juste derrière, nous n'avons pas écrit le code de l'exécutable pour des questions de clarté.

Interfaces fonctionnelles Java 8

- **Function** Une Function prend 1 argument et retourne un résultat

```
Function<Integer, String> maFonction = n -> String.valueOf(n);  
System.out.println(maFonction.apply(100));
```

Méthode
apply pour
appliquer
la fonction

- **BiFunction** Est une spécialisation de Function prend 2 arguments et retourne un résultat

```
BiFunction<Integer, Integer, Integer > monAddition = (a, b) -> a + b;  
System.out.println(monAddition.apply(10, 15));
```

D'autres
méthodes
existent
cf. doc

- **UnaryOperator** Est une Function, prend 1 argument et retourne le même type que l'argument

```
UnaryOperator<Integer> carre = n -> n * n;
```

- **BinaryOperator** Est une spécialisation de BiFunction arguments et résultat ont le même type

```
BinaryOperator<String> concat = (ch1, ch2) -> ch1 + " a dit : '" + ch2 + "'";
```

Interfaces fonctionnelles Java 8

- **Predicate** Prend 1 argument et retourne un booléen

```
Predicate<String> palindrome = s-> {  
    String chain = s.replace(" ", ""); supprime les espaces  
    StringBuffer buffer = new StringBuffer(chain); // méthode inversion  
    buffer.reverse();  
    return chain.equals(buffer.toString()); }; //fin fonction  
//Execution
```

```
String palin = "engage le jeu que je le gagne";  
System.out.println(palin + " est-ce un palindrome ? " + palindrome.test(palin));
```

```
System.out.println(palindrome.negate().test("kayak"));
```

Méthode test pour appliquer la fonction

D'autres méthodes existent cf. doc

- **BiPredicate** Est une spécialisation de Predicate prend 2 arguments et retourne un résultat

Interfaces fonctionnelles Java 8

- **Supplier** : n'a pas d'argument et retourne un résultat
- **Consumer** : prend 1 argument et ne retourne pas de résultat

```
Supplier<String> hola = () -> "bienvenue à tous";  
Consumer<String> affiche = s-> System.out.println(s);
```

```
affiche.accept(hola.get());
```

Méthode
get pour
appliquer
la fonction

Méthode
accept pour
appliquer la
fonction

Interfaces fonctionnelles à retenir

Un petit aperçu du package →

- S'il n'y a pas de retour, on utilise un Consumer
- S'il faut retourner un booléen, on utilise un Predicate
- S'il faut retourner une valeur sans prendre d'argument, c'est un Supplier
- Si la fonction prend deux arguments, c'est une Bixx (où xx est une interface telle Predicate)
- Si la fonction prend deux arguments de même type, c'est un BinaryOperator
- Si une fonction retourne une valeur de même type que son unique argument, c'est un UnaryOperator
- Si la fonction prend en argument un type primitif numérique et un autre type sans retourner de valeur, c'est un ObjyyConsumer (où xx est Int Double ou Long)
- Si l'unique argument de la fonction est de type int, double, long on utilise les interfaces Intxx, Doublexx, Longxx (où xx est une interface telle Predicate)
- S'il faut retourner un primitif numérique (int, double, long), on utilise un yyToIntFunction, yyToDoubleFunction, yyToLongFunction, (où xx est Int Double ou Long)

java.util.function
▷  BiConsumer.class
▷  BiFunction.class
▷  BinaryOperator.class
▷  BiPredicate.class
▷  BooleanSupplier.class
▷  Consumer.class
▷  DoubleBinaryOperator.class
▷  DoubleConsumer.class
▷  DoubleFunction.class
▷  DoublePredicate.class
▷  DoubleSupplier.class
▷  DoubleToIntFunction.class
▷  DoubleToLongFunction.class
▷  DoubleUnaryOperator.class
▷  Function.class
▷  IntBinaryOperator.class
▷  IntConsumer.class
▷  IntFunction.class
▷  IntPredicate.class
▷  IntSupplier.class
▷  IntToDoubleFunction.class
▷  IntToLongFunction.class
▷  IntUnaryOperator.class
▷  LongBinaryOperator.class
▷  LongConsumer.class
▷  LongFunction.class

.../...

Interfaces fonctionnelles existantes

- Exemples d'interfaces fonctionnelles existantes dans le JDK : Runnable, Comparator<T>, Callable<V>.

1 `java.lang.Runnable`
 `public abstract void run();`
 `// only one abstract method`

2 `java.util.Comparator`
 `int compare(T o1, T o2); //abstract method`
 `boolean equals(Object obj);`
 `//non abstract metho of object class`

3 `java.util.concurrent.Callable`
 `V call();`
 `// only one abstract method`

4 `java.awt.event.ActionListener`
 `void actionPerformed(ActionEvent e);`
 `// only one abstract method`

Assigner une expression Lambda à une variable locale

```
public interface Consumer<T> {  
    void accept(T t);  
}  
...  
void forEach(Consumer<Integer> action {  
    for (Integer i:items) {  
        action.accept(t);  
    }  
}
```

Définitions
existantes de Java 8

```
List<Integer> intSeq = Arrays.asList(1,2,3);
```

```
Consumer<Integer> cnsmr = x -> System.out.println(x);  
intSeq.forEach(cnsmr);  
...
```

Assigner une expression Lambda à une variable locale

- Voici une interface appelée Consumer avec une seule méthode appelée accept.
- La méthode forEach parcourt les éléments de l'objet Consumer
`void forEach(Consumer<Integer> action {.../...})` et exécute l'action accept sur chaque élément `action.accept(t);`.
- L'expression lambda devient le corps de la fonction dans l'interface.
`Consumer<Integer> cnsmr = x -> System.out.println(x);`
- La signature de la fonction est définie par l'interface.

Propriétés de la méthode générée

- La méthode générée à partir d'une expression lambda Java 8 a la même signature que la méthode dans l'interface fonctionnelle.
- Le type est le même que celui de l'interface fonctionnelle à laquelle l'expression lambda est affectée.
- L'expression lambda devient le corps de la méthode dans l'interface.
- Toute interface avec une seule méthode non définie par défaut est considérée comme une interface fonctionnelle par Java 8.

Capture de variable

- Les expressions Lambda peuvent interagir avec des variables définies en dehors du corps de l'expression Lambda.
- L'utilisation de ces variables est appelée capture de variable.

```
public class VariableCaptureExample {  
    public static void main(String[] args) {  
        List<Integer> intSeq = Arrays.asList(1,2,3);  
  
        int var = 10;  
        intSeq.forEach(x -> System.out.println(x + var));  
    }  
}
```

Capture de variable static

```
public class StaticVariableCaptureExample {  
    private static int var = 10;  
  
    public static void main(String[] args) {  
  
        List<Integer> intSeq = Arrays.asList(1,2,3);  
        intSeq.forEach(x -> System.out.println(x + var));  
    }  
}
```

Références de méthode

Method Reference Java 8



Références de méthode

- En Java, nous pouvons utiliser des références aux objets, soit en créant de nouveaux objets soit en utilisant des objets existants:

```
List list = new ArrayList();  
List list2 = list;  
isFull(list2);
```

- Si nous n'utilisons qu'une méthode d'un objet dans une autre, nous devons néanmoins passer l'objet complet en tant qu'argument. Ne serait-il pas plus pratique de simplement passer la méthode comme argument?

Par exemple : `isFull(list.size());`

Références de méthode

- En Java 8, grâce aux expressions Lambda, nous pouvons utiliser des méthodes comme s'il s'agissait d'objets ou de valeurs primitives.
- Une référence de méthode est la syntaxe abrégée d'une expression Lambda qui exécute une seule méthode.
- Voici la syntaxe générale d'une référence de méthode:

Object :: methodName

- Les références de méthode peuvent être utilisées pour transmettre une fonction existante aux endroits où une expression Lambda est attendue.
- La signature de la méthode référencée doit correspondre à la signature de la méthode d'interface fonctionnelle.

Références de méthode

- Parfois, l'expression lambda n'est en réalité qu'un appel à une méthode, par exemple:
`Consumer<String> c = s -> System.out.println(s);`
- Pour rendre le code plus clair, vous pouvez transformer cette expression lambda en une référence de méthode:
`Consumer<String> c = System.out::println;`
- Dans une référence de méthode, vous placez l'objet (ou la classe) qui contient la méthode avant l'opérateur :: et le nom de la méthode après, sans arguments.
- Les références de méthode ne peuvent être utilisées que pour remplacer une expression lambda à méthode unique.

Références de méthode

- Il existe quatre types de références de méthodes:
 - Une référence de méthode à une méthode static.
 - Une référence de méthode à une méthode d'instance d'un objet d'un type particulier.
 - Une référence de méthode à une méthode d'instance d'un objet existant.
 - Une référence de méthode à un constructeur.

Références de méthode static

- L'expression Lambda ci-dessous :
`(args) -> Class.staticMethod(args)`
- Peut être transformée en la référence de méthode suivante :
Class::staticMethod
- En général, il n'est pas nécessaire de passer des arguments aux références de méthodes. Cependant, les arguments sont traités en fonction du type de référence de la méthode.
- Où que nous puissions passer une expression lambda qui appelle simplement une méthode static, nous pouvons utiliser une référence de méthode. Par exemple :

Références de méthode - Exemple

```
class Numbers {  
    public static boolean isMoreThanFifty(int n1, int n2) {  
        return (n1 + n2) > 50;  
    }  
    public static List<Integer> findNumbers(  
        List<Integer> l, BiPredicate<Integer, Integer> p) {  
        List<Integer> newList = new ArrayList<>();  
        for(Integer i : l) {  
            if(p.test(i, i + 10)) {  
                newList.add(i);  
            }  
        }  
        return newList;  
    }  
}
```

Références de méthode - Exemple

- Nous pouvons appeler la méthode `findNumbers()` de l'une des manières suivantes :

```
List<Integer> list = Arrays.asList(12,5,45,18,33,24,40);
```

```
// En utilisant une classe anonyme
findNumbers(list, new BiPredicate<Integer, Integer>() {
    public boolean test(Integer i1, Integer i2) {
        return Numbers.isMoreThanFifty(i1, i2);
    }
});
```

Références de méthode - Exemple

- Nous pouvons appeler cette méthode en utilisant:

```
// En utilisant une expression Lambda  
findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));
```

```
// En utilisant une référence de méthode  
findNumbers(list, Numbers::isMoreThanFifty);
```

Référence de méthode d'instance d'un objet d'un type particulier

```
class Shipment {  
    public double calculateWeight() {  
        double weight = 0;  
        // Calculate weight  
        return weight;  
    }  
  
    public List<Double> calculateOnShipments(  
        List<Shipment> l, Function<Shipment, Double> f) {  
        List<Double> results = new ArrayList<>();  
        for(Shipment s : l) {  
            results.add(f.apply(s));  
        }  
        return results;  
    }  
}
```

Type particulier
Shipment

Référence de méthode d'instance d'un objet d'un type particulier

- Nous pouvons appeler cette méthode en utilisant:

```
// En utilisant une expression Lambda  
calculateOnShipments(l, s -> s.calculateWeight());
```

```
// En utilisant une référence de méthode  
calculateOnShipments(l, Shipment::calculateWeight);
```

Référence de méthode d'instance d'un objet existant

- L'expression Lambda suivante :
`(args) -> obj.instanceMethod(args)`
- Peut être transformée en la référence de méthode suivante :
`obj::instanceMethod`
- Une instance définie ailleurs est utilisée et les arguments (le cas échéant) sont passés en arrière plan, comme dans le cas de la méthode static.

Référence de méthode d'instance d'un objet existant

- Par exemple :

```
class Car {  
    private int id;  
    private String color;  
}
```

```
class Mechanic {  
    public void fix(Car c) {  
        System.out.println("Fixing car " + c.getId());  
    }  
}
```

Référence de méthode d'instance d'un objet existant

- Nous pouvons appeler la méthode ci-dessus en utilisant:

```
// En utilisant une expression Lambda  
execute(car, c -> mechanic.fix(c));
```

```
// En utilisant une référence de méthode  
execute(car, mechanic::fix);
```

Référence de méthode à un constructeur

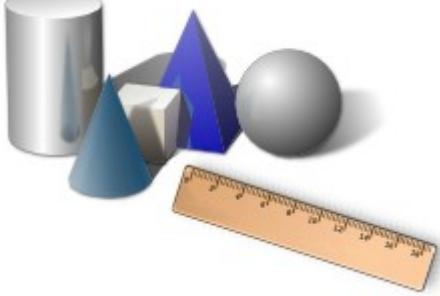
- L'expression Lambda suivante :
`(args) -> new ClassName(args)`
- Peut être transformée en la référence de méthode suivante :
`ClassName::new`
- La seule chose que cette expression Lambda fait est de créer un nouvel objet et nous référons simplement un constructeur de la classe avec le mot-clé `new`.
- Les arguments (le cas échéant) ne sont pas passés dans la référence de la méthode.

Référence de méthode à un constructeur

- Nous pouvons appeler cette méthode en utilisant:

```
// En utilisant une expression lambda  
Supplier<List<String>> s = () -> new ArrayList<String>();  
List<String> l = s.get();
```

```
// En utilisant une référence de méthode  
Supplier<List::new>;  
List<String> l = s.get();
```



Exercice 8 (voir cahier d'exercices)

Lambda et interfaces fonctionnelles

➤ Manipulation de lambda et d'interfaces fonctionnelles

- Nous disposons d'une classe Person et d'un ancien traitement pour afficher les personnes ayant plus de 20 ans.
 - Le but est de modifier l'exécutable pour manipuler des lambdas et des interfaces fonctionnelles. Ces modifications permettent de pouvoir écrire du code plus évolutif, car les critères de recherche peuvent évoluer.
- Nous pouvons dans TestCompte, de notre gestion bancaire, utiliser une expression lambda pour afficher l'historique des mouvements de notre compte :

```
cpt.getHistorique().foreach(x -> System.out.println(x));
```

Ou utiliser une référence de méthode System.out::print

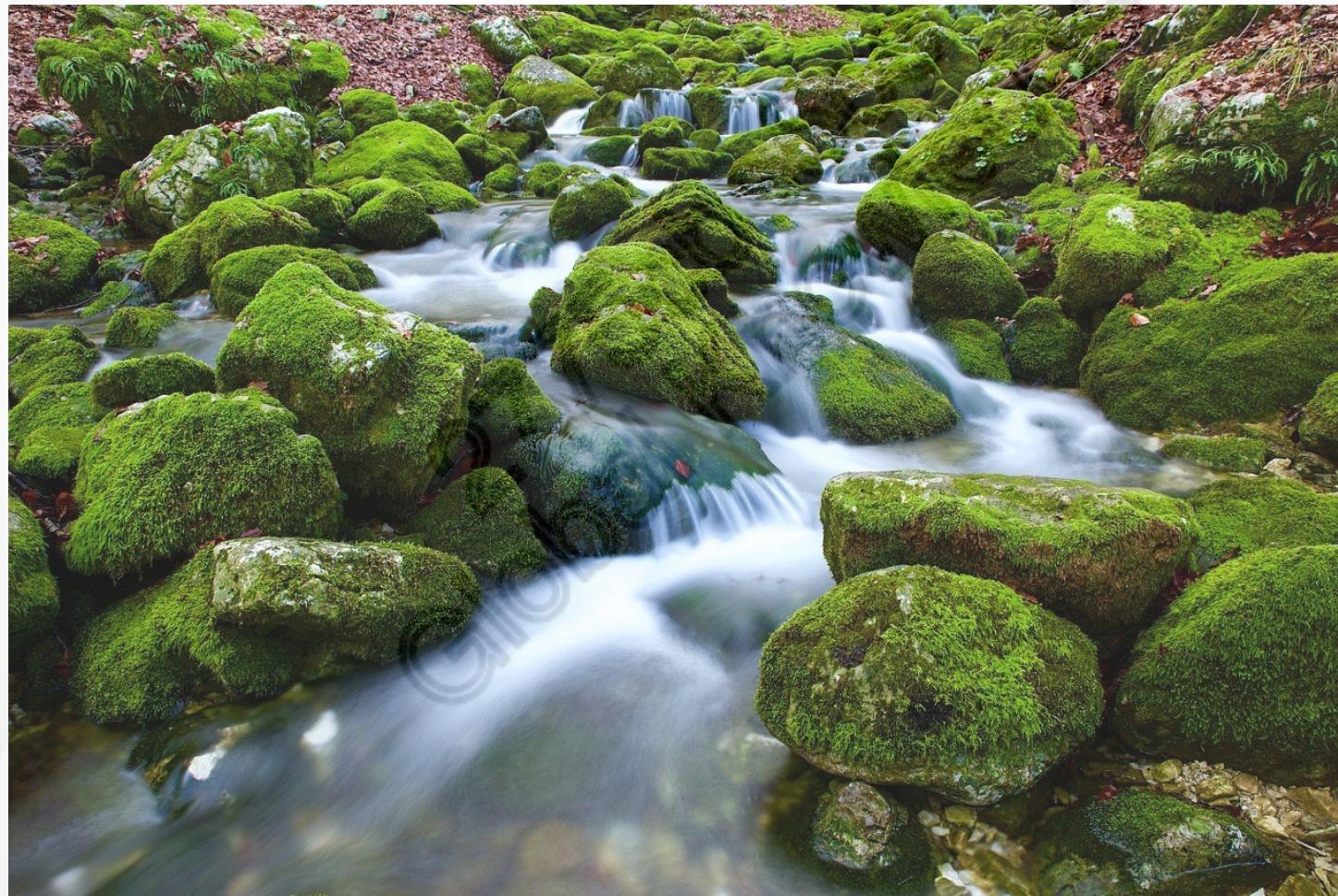
Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** **Les Streams**
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Qu'est-ce qu'un Stream ?
- Architecture des Streams
- Java Stream API

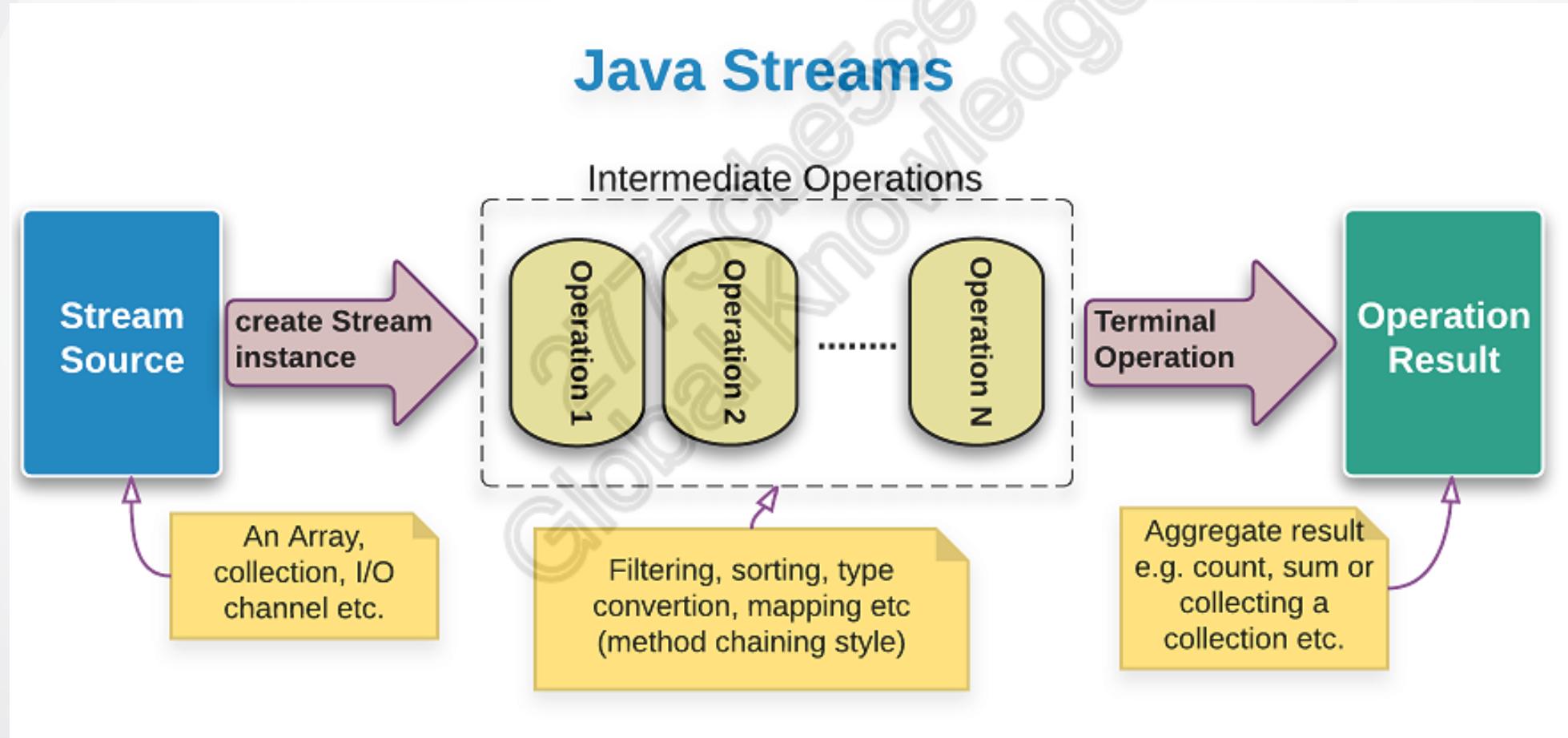
Qu'est-ce qu'un Stream ?



Qu'est-ce qu'un Stream ?

- Un stream, ou flux, est « une séquence d'éléments d'une source prenant en charge des opérations de traitement de données » → ITÉRATION INTERNE.
- Les collections sont conservées dans l'espace mémoire. Vous avez besoin de toute la collection pour itérer → ESPACE.
- Les flux sont créés à la demande et ils sont infinis → TEMPS.
- Les flux sont :
 - Déclaratif : plus concis et lisible.
 - Composable : une plus grande flexibilité.
 - Parallélisable : meilleure performance.

Architecture des Streams



Architecture des Streams

- Un flux est traité par un pipeline d'opérations.
- Un flux commence par une structure de données source.
- Les méthodes intermédiaires sont exécutées sur les éléments du flux. Ces méthodes produisent des flux et ne sont pas traitées jusqu'à l'appel de la méthode terminale.
- Le flux est considéré comme consommé lorsqu'une opération terminale est appelée. Aucune autre opération ne peut être effectuée sur les éléments du flux par la suite.

Architecture des Streams

- Un pipeline de flux contient des méthodes de court-circuit (pouvant être des méthodes intermédiaires ou des méthodes terminales) qui entraînent le traitement des méthodes intermédiaires antérieures uniquement jusqu'à ce que la méthode de court-circuit puisse être évaluée.

Séquence d'éléments

- À l'instar d'une collection, un flux fournit une interface pour un ensemble séquencé de valeurs d'un type d'élément spécifique.
- Les collections étant des structures de données, elles consistent principalement à stocker et à accéder à des éléments, mais les flux consistent également à exprimer des calculs tels que filtrer, trier et mapper.

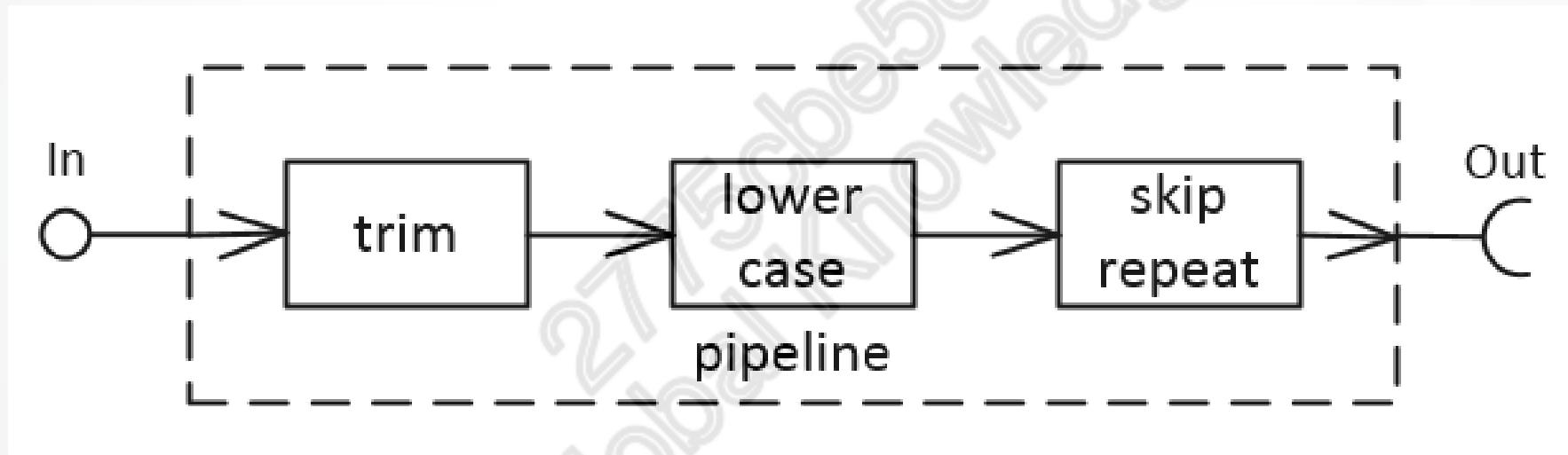
Source

- Les flux utilisent une source de données, telle que des collections, des tableaux ou des ressources d'E / S.
- Notez que la génération d'un flux à partir d'une collection ordonnée préserve le classement. Les éléments d'un flux provenant d'une liste auront le même ordre que la liste.

Operation

- Les flux supportent les opérations de type base de données et les opérations courantes depuis des langages de programmation fonctionnels pour manipuler des données, telles que filtrer, mapper, réduire, trouver, faire correspondre, trier, etc.
- Les opérations de flux peuvent être exécutées séquentiellement ou en parallèle.
- Une opération intermédiaire maintient un flux ouvert pour d'autres opérations. Les opérations intermédiaires sont paresseuses.
- Une opération terminale doit être l'opération finale sur un flux. Une fois qu'une opération terminale est appelée, le flux est consommé et n'est plus utilisable.

Pipeline



Pipeline

- Un pipeline de flux a trois composants :
 - Une source telle qu'une collection, un tableau, une fonction de générateur ou un canal d'E / S.
 - Zéro ou plusieurs opérations intermédiaires.
 - Une opération terminale.
- De nombreuses opérations de flux retournent un flux elles-mêmes, ce qui permet d'enchaîner les opérations et de former un plus grand pipeline.
- Un pipeline d'opérations peut être visualisé comme une requête de type base de données sur la source de données.

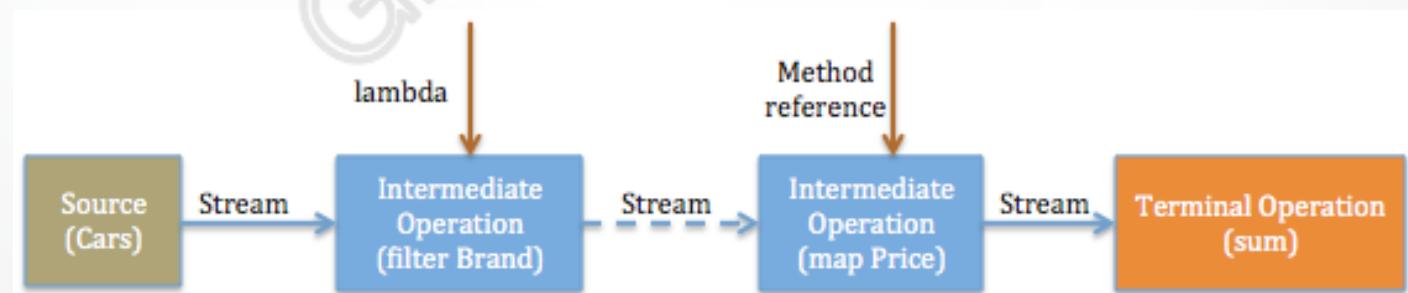
Itération interne

- Contrairement aux collections, qui sont itérées explicitement à l'aide d'un itérateur, les opérations de flux effectuent l'itération en coulisse pour vous.



Consommation des flux

- Les flux sont consommés. Vous ne pouvez les parcourir qu'une seule fois.
- Vous pouvez demander à un nouveau flux provenant de la source de données initiale de le traverser à nouveau, comme pour un itérateur (en supposant qu'il s'agisse d'une source reproductible comme une collection. S'il s'agit d'un canal d'E / S, ceci n'est pas possible).
- Seule l'opération terminale peut consommer un flux, ce qui est fait de manière paresseuse ou à la demande.



Java Stream API

- Le nouveau package `java.util.stream` fournit des utilitaires permettant de prendre en charge des opérations de style fonctionnel sur des flux de valeurs.
- Un moyen courant d'obtenir un flux consiste à partir d'une collection:
- `Stream <T> stream = collection.stream();`
- Les flux peuvent être séquentiels ou parallèles.
- Les flux sont utiles pour sélectionner des valeurs et effectuer des actions sur les résultats.

Java Stream API – Les Stream

- Les Stream ne sont pas liés aux InputStreams, OutputStreams, etc.
- Les Stream ne sont pas des structures de données, mais sont des enveloppes autour de Collection qui transportent des valeurs provenant d'une source via un pipeline d'opérations.
- Les Stream sont plus puissants, plus rapides et plus efficaces en mémoire que les listes.
- Les Stream sont conçus pour les expressions Lambda.
- Les Stream peuvent facilement être sortis sous forme de tableaux ou de listes.
- Les Stream utilisent l'évaluation paresseuse et sont parallélisables.

Java Stream API – Les Stream

➤ La création des Stream peut se faire à partir :

- De valeurs individuelles : Stream.of (val1, val2,...)

```
Stream<String> stream = Stream.of("hello", "hola", "hallo", "ciao");
```

- De tableau : Stream.of (someArray) ou Arrays.stream (someArray)

```
String[] words = {"hello", "hola", "hallo", "ciao"};
```

```
Stream<String> stream = Stream.of(words);
```

- D'une liste (et autres collections) : someList.stream () ou someOtherCollection.stream ()

```
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});  
Stream<String> stream = words.stream();
```

Java Stream API - Opérations intermédiaires

- Une méthode qui prend un flux et renvoie un flux.
- Elles sont chargées paresseusement et ne seront exécutées que si vous incluez une opération terminale à la fin.

La méthode filter () : exclut tous les éléments qui ne correspondent pas à un prédictat.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
long count = strings.stream().filter(string -> string.isEmpty()).count();
```

Comptage
élément
vide

map () : effectue une transformation individuelle des éléments à l'aide d'une fonction.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
```

```
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

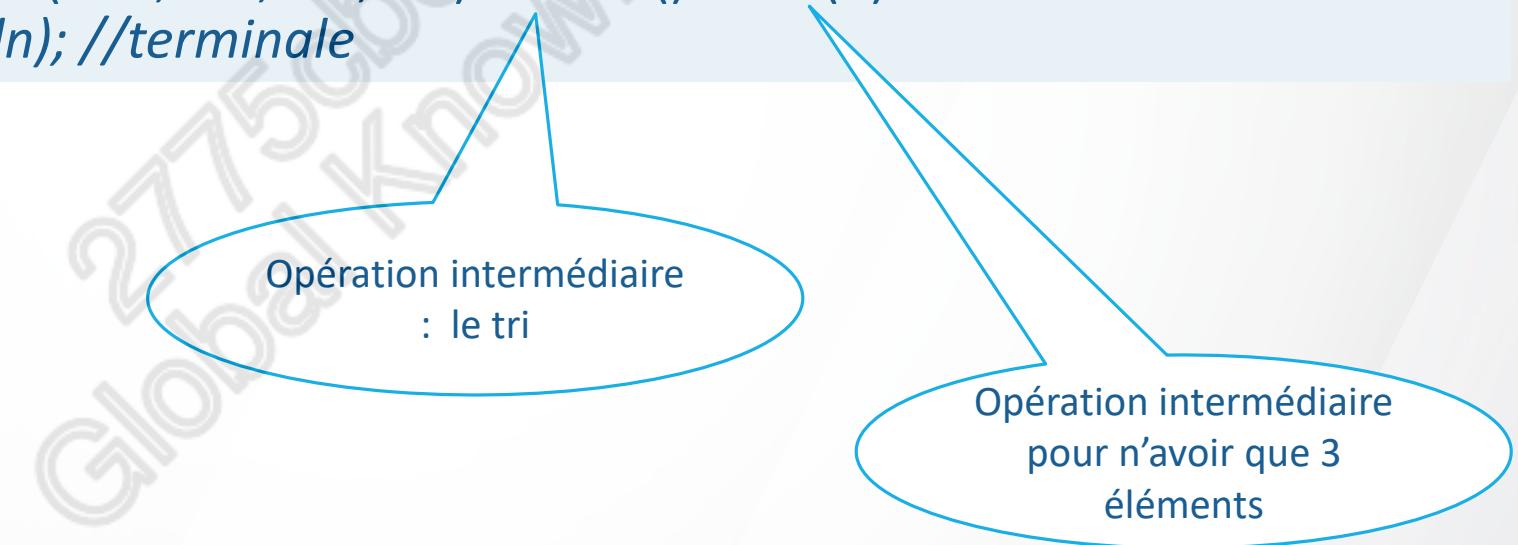
Mise au carré des éléments avec suppression
des doublons

Opération terminale : réunir tous les
éléments en utilisant un Collectors

Java Stream API - Opérations intermédiaires

- La méthode sorted () : renvoie une vue triée du flux. Les éléments sont triés dans un ordre naturel sauf si vous transmettez un comparateur personnalisé.

```
Stream<String> s = Stream.of("m", "k", "c", "t") .sorted() .limit(3)  
s.forEach(System.out::println); //terminale
```



Java Stream API - Opérations terminales

- Une méthode qui prend un flux <T> et retourne void.
- Elles sont chargées immédiatement et entraîneront l'exécution de tout le pipeline.
- Une opération terminale "dépensera" le flux.
- Exemples :
 - La méthode `forEach ()` : effectue une action pour chaque élément de ce flux.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```
 - La méthode `count ()` : renvoie le nombre d'éléments dans ce flux.

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
IntStream s = IntStream.of(digits);    long n = s.count();
```

Java Stream API - Opérations terminales

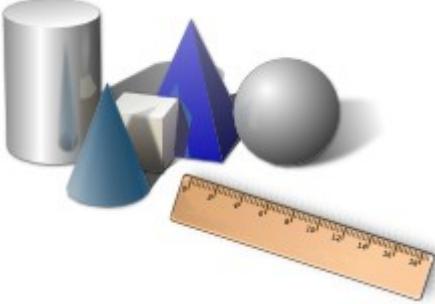
- La méthode `max()` : renvoie l'élément maximum de ce flux en fonction du comparateur fourni.
- La méthode `collect()` : effectue une opération de réduction mutable sur les éléments de ce flux à l'aide d'un collecteur.
- La méthode `reduce()` : effectue une réduction sur les éléments de ce flux à l'aide de la valeur d'identité fournie et d'une fonction d'accumulation associative, puis renvoie la valeur réduite.

Exemple de flux

```
List<Integer> list = Arrays.asList(1,2,3);  
int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();  
System.out.println(sum);
```

Résultat
14

- map ($x \rightarrow x * x$) met au carré chaque élément de la liste puis reduce($(x, y) \rightarrow x + y$) réduit tous les éléments en un seul nombre qui correspond au cumul des éléments.



Exercice 8 (Suite)

Lambda et Stream (cahier d'exercices)

➤ Manipulation de lambda et Stream

- **Calculer la somme des dépôts avec mapToInt et sum**
- **Compter le nombre de dépôt supérieur à 60, faire un prédictat pour filter les dépôts obtenus par mapToInt**

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Muti-tâches

- Multi-tâches : exécution de plusieurs processus simultanément.
 - Un processus est un programme en cours d'exécution.
 - Le système d'exploitation distribue le temps CPU entre les processus.
- Un processus peut être dans différents états.
 - En exécution (running) : il utilise le processeur
 - Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution).
 - Bloqué

Parallélisme

- Parallélisme : pouvoir faire exécuter plusieurs tâches à un ordinateur avec plusieurs processeurs.
- Si l'ordinateur possède moins de processeurs que de processus à exécuter :
 - Division du temps d'utilisation du processeur en tranches de temps (time slice en anglais)
 - Attribution des tranches de temps à chacune des tâches de façon telle qu'on ait l'impression que les tâches se déroulent en parallèle.
 - On parle de pseudo-parallélisme.
- Les systèmes d'exploitation modernes gèrent le multi-tâche et le parallélisme.

Qu'est-ce qu'un Thread ?

- Les threads sont différents des processus :
 - Ils partagent code, données et ressources : « processus légers »
 - Mais peuvent disposer de leurs propres données.
 - Ils peuvent s'exécuter en "parallèle"
- Avantages :
 - Légèreté grâce au partage des données
 - Meilleures performances au lancement et en exécution
 - Partage les ressources systèmes (pratique pour les I/O)
- Utilité :
 - Puissance de la modélisation : un monde multithread
 - Puissance d'exécution : parallélisme
 - Simplicité d'utilisation : c'est un objet Java (java.lang)

Création d'un Thread

- La classe `java.lang.Thread` permet de créer de nouveaux threads
- Un thread doit implémenter obligatoirement l'interface `Runnable`
 - **Le code exécuté se situe dans sa méthode `run()`**
- 2 méthodes pour créer un Thread :
 1. **Une classe qui dérive de `java.lang.Thread`**
 - `java.lang.Thread` implémente `Runnable`
 - Il faut redéfinir la méthode `run()`
 2. **Une classe qui implémente l'interface `Runnable`**
 - Il faut implémenter la méthode `run()`

Méthode 1 : Sous-classer Thread

```
class Proc1 extends Thread {  
    Proc1() {...} // Le constructeur  
    ...  
    public void run() {  
        ... // Ici ce que fait le processus : boucle infinie  
    }  
}  
//Dans une autre classe  
Proc1 p1 = new Proc1(); // Création du processus p1  
p1.start(); // Démarrer le processus qui exécute run() de l'objet p1
```

Méthode 2 : une classe qui implémente Runnable

```
class Proc2 implements Runnable {  
    Proc2() { ... } // Constructeur  
    ...  
    public void run() {  
        ... // Ici ce que fait le processus  
    }  
}  
  
Dans une autre classe...  
Proc2 p = new Proc2();  
Thread p2 = new Thread(p);  
...  
p2.start(); // Démarre un processus qui execute p.run()
```

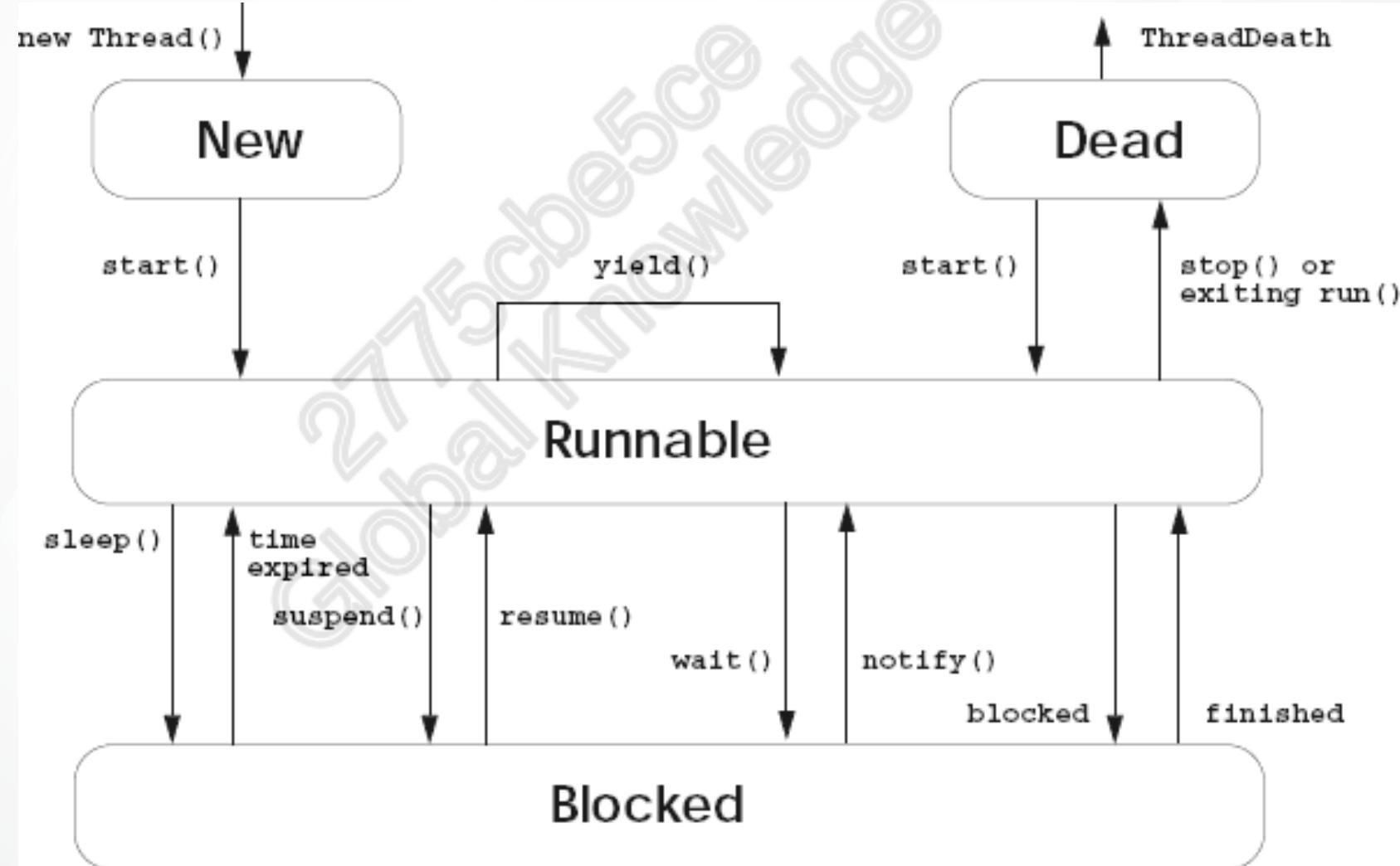
Quelle solution choisir ?

- Méthode 1 : sous-classer Thread
 - Lorsqu'on désire paralléliser une classe qui n'hérite pas déjà d'une autre classe
 - Cas des applications autonomes
- Méthode 2 : implémenter Runnable
 - Lorsqu'une super-classe est imposée
 - Cas des applets (cependant, il est possible de créer une définition qui hérite de Thread et l'applet gère la communication)

```
public class MyThreadApplet  
    extends Applet implements Runnable {
```

- Avec strictement le prototype indiqué (*il faut redéfinir Thread.run et non pas la surcharger*). Distinguer la méthode run (qui est le code exécuté par l'activité) et la méthode start (méthode de la classe Thread qui rend l'activité exécutable) ;
- Dans la première méthode de création, attention à définir la méthode run.

Le cycle de vie



Les états d'un thread

- Crée :
 - Comme n'importe quel objet Java
 - ... mais n'est pas encore actif
- Actif :
 - Après la création, il est activé par start() qui lance run().
 - Il est alors ajouté dans la liste des threads actifs pour être exécuté par l'OS en temps partagé
 - Peut revenir dans cet état après un resume() ou un notify()

Exemple

```
public class ThreadCompteur extends Thread {  
  
    int no_fin;  
  
    // Constructeur  
    ThreadCompteur(int fin, String name) {  
        super(name);  
        no_fin = fin;  
    }  
  
    // On redéfinit la méthode run()  
    public void run() {  
        for (int i = 1; i <= no_fin; i++) {  
            System.out.println(this.getName() + ":" + i);  
        }  
    }  
}
```

```
public class testThread {  
  
    public static void main(String[] args) {  
        // On instancie les threads  
        ThreadCompteur cp1 = new ThreadCompteur(50, "1");  
        ThreadCompteur cp2 = new ThreadCompteur(50, " | 2");  
        ThreadCompteur cp3 = new ThreadCompteur(50, " | | 3");  
        ThreadCompteur cp4 = new ThreadCompteur(50, " | | | 4");  
  
        cp1.start();  
        cp2.start();  
        cp3.start();  
        cp4.start();  
    }  
}
```

```
1:29  
| | | 4:1  
|2:46  
| | | 4:2  
1:30  
1:31  
| | | 4:3  
| | 3:1  
|2:47  
|2:48  
|2:49  
|2:50  
| | | 4:4
```

Les états d'un Thread (suite)

- Endormi ou bloqué :
 - Après sleep() : endormi pendant un intervalle de temps (ms)
 - Suspend() endort le Thread mais resume() le réactive
 - Une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread
- Mort :
 - Si stop() est appelé explicitement
 - Quand run() a terminé son exécution

Exemple d'utilisation de sleep

```
public class ThreadCompteur2 extends Thread {  
  
    int no_fin, attente;  
  
    // Constructeur  
    ThreadCompteur2(int fin, String name, int att) {  
        super(name);  
        no_fin = fin;  
        attente = att;  
    }  
  
    // On redéfinit la méthode run()  
    public void run() {  
        for (int i = 1; i <= no_fin; i++) {  
            System.out.println(this.getName() + ":" + i);  
            try {  
                sleep(attente);  
            } catch (InterruptedException e) {  
                // TODO: handle exception  
            }  
        }  
    }  
}
```

```
public class testThread2 {  
  
    public static void main(String[] args) {  
        // On instancie les threads  
        ThreadCompteur2 cp1 = new ThreadCompteur2(50, "1", 100);  
        ThreadCompteur2 cp2 = new ThreadCompteur2(50, "2", 50);  
  
        cp1.start();  
        cp2.start();  
    }  
}
```

1:1
| 2:1
| 2:2
| 2:3
1:2
| 2:4
1:3
| 2:5
| 2:6
1:4
| 2:7
| 2:8
1:5
| 2:9
| 2:10
1:6
| 2:11
| 2:12
1:7
| 2:13
| 2:14

Les priorités

- Principes :
 - Java permet de modifier les priorités (niveaux absous) des Threads par la méthode `setPriority()`
 - Par défaut, chaque nouveau Thread a la même priorité que le Thread qui l'a créé
 - Rappel : seuls les Threads actifs peuvent être exécutés et donc accéder au CPU
 - La JVM choisit d'exécuter le Thread actif qui a la plus haute priorité : priority-based scheduling
 - Si plusieurs Threads ont la même priorité, la JVM répartit équitablement le temps CPU (time slicing) entre tous : round-robin scheduling

Les priorités (suite)

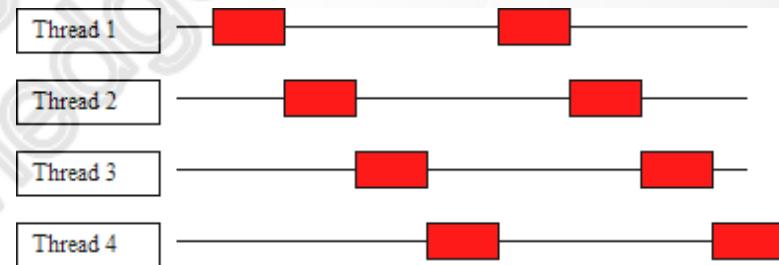
➤ Les méthodes :

- `setPriority(int)` : fixe la priorité du receveur.
 - Le paramètre doit appartenir à :
[`MIN_PRIORITY`, `MAX_PRIORITY`]
 - Sinon `IllegalArgumentException` est levée
- `int getPriority()` : pour connaître la priorité d'un Thread
 - `NORM_PRIORITY` : donne le niveau de priorité "normal"

La gestion du CPU

➤ Time-slicing (ou round-robin scheduling) :

- La JVM répartit de manière équitable le CPU entre tous les threads de même priorité.
Ils s'exécutent en "parallèle".



➤ Préemption (ou priority-based scheduling) :

- Le premier thread du groupe des threads à priorité égale monopolise le CPU.
Il peut le céder :

- Involontairement : sur entrée/sortie
 - Volontairement : appel à la méthode statique `yield()`

Attention : ne permet pas à un thread de priorité inférieure de s'exécuter
(seulement de priorité égale)

- Implicitement en passant à l'état endormi (`wait()`, `sleep()` ou `suspend()`)

La concurrence d'accès

- Le problème : espace de travail commun, pas de "mémoire privée" pour chaque thread :
 - Inconvénient : accès simultané à une même ressource
Il faut garantir l'accès exclusif à un objet pendant l'exécution d'une ou plusieurs instructions
- Pour se faire : le mot-clé synchronized permet de gérer les concurrences d'accès :
 - D'une méthode
 - D'un objet
 - Ou d'une instruction (ou d'un bloc)

La synchronisation

- Basée sur la technique de l'exclusion mutuelle :
 - A chaque objet Java est associé un « verrou » géré par le thread quand une méthode (ou un objet) `synchronized` est accédé.
 - Garantit l'accès exclusif à une ressource (la section critique) pendant l'exécution d'une portion de code.
- Une section critique :
 - Une méthode : déclaration précédée de `synchronized`
 - Une instruction (ou un bloc) : précédée de `synchronized`
 - Un objet : le déclarer `synchronized`
- Attention à l'inter-blocage !!
(problème du dîner des philosophes)

Utiliser synchronized

- Pour gérer la concurrence d'accès à une méthode :
 - Si un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet
 - En revanche, il peut exécuter cette méthode pour un autre objet

```
public synchronized void maMethode() {....}
```

- Pour Contrôler l'accès à un objet :

```
public void maMethode() { ...  
    synchronized(objet) {  
        objet.methode(); } }
```

- L'accès à l'objet passé en paramètre de synchronized(Object) est réservé à un unique thread.

Exemple de synchronisation

```
class Impression {  
    synchronized public void imprime(String t) {  
        for (int i=0; i<t.length(); i++) { System.out.print(t.charAt(i));  
    } } }  
  
class TPrint extends Thread {  
    static Impression mImp = new Impression();  
    String txt;  
    public TPrint(String t) {txt = t;}  
  
    public void run() {  
        for (int j=0; j<3; j++) {mImp.imprime(txt);}}  
  
    static public void main(String args[]) {  
        TPrint a = new TPrint("bonjour ");  
        TPrint b = new TPrint("au revoir ");  
        a.start();  
        b.start();  
    } }
```

Daemons

- Un thread peut être déclaré comme daemon :
 - Comme le "garbage collector", l'"afficheur d'images", ...
 - En général de faible priorité, il "tourne" dans une boucle infinie
 - Arrêt implicite dès que le programme se termine
- Les méthodes :
 - `setDaemon()` : déclare un thread daemon
 - `isDaemon()` : ce thread est-il un daemon ?

Les « ThreadGroup »

Pour contrôler plusieurs threads :

- Plusieurs processus (Thread) peuvent s'exécuter en même temps, il serait utile de pouvoir les manipuler comme une seule entité
 - Pour les suspendre
 - Pour les arrêter, ...

Java offre cette possibilité via l'utilisation des groupes de threads :

`java.lang.ThreadGroup`

- On groupe un ensemble nommé de threads
- Ils sont contrôlés comme une seule unité

Les groupes de threads

- Une arborescence :
 - La classe `ThreadGroup` permet de constituer une arborescence de `Threads` et de `ThreadGroups`
 - Elle donne des méthodes classiques de manipulations récursives d'un ensemble de threads : `suspend()`, `stop()`, `resume()`, ...
 - Et des méthodes spécifiques : `setMaxPriority()`, ...
- Fonctionnement :
 - La JVM crée au minimum un groupe de threads nommé `main`
 - Par défaut, un thread appartient au même groupe que celui qui l'a créé (son père)
 - `getThreadGroup()` : pour connaître son groupe

Création d'un groupe de threads

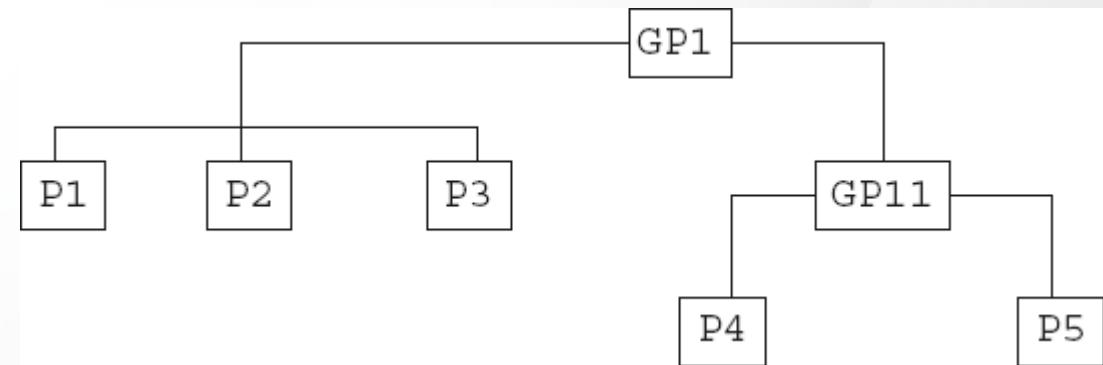
➤ Pour créer un groupe de processus :

```
ThreadGroup groupe = new ThreadGroup("Mon groupe");  
Thread p1 = new Thread(groupe, "P1");  
Thread p2 = new Thread(groupe, "P2");  
Thread p3 = new Thread(groupe, "P3");
```

- ## ➤ On peut créer des sous-groupes de threads pour la création d'arbres sophistiqués de processus :
- Des ThreadGroup contiennent des ThreadGroup
 - Des threads peuvent être au même niveau que des ThreadGroup

Création de groupe de threads (suite)

```
ThreadGroup group1 = new ThreadGroup("GP1");  
Thread p1 = new Thread(group1, "P1");  
Thread p2 = new Thread(group1, "P2");  
Thread p3 = new Thread(group1, "P3");  
  
ThreadGroup group11 = new ThreadGroup(group1, "GP11");  
Thread p4 = new Thread(group11, "P4");  
Thread p5 = new Thread(group11, "P5");
```



Contrôler les ThreadGroup

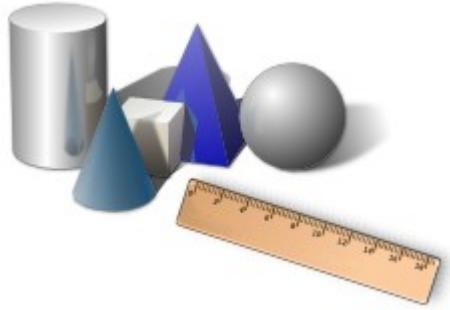
- Le contrôle des ThreadGroup passe par l'utilisation des méthodes standards qui sont partagées avec Thread :

`resume()`, `suspend()`, `stop()`, ...

- Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode
- Ce sont des méthodes de manipulation récursive

Avantages / Inconvénients des threads

- Programmer facilement des applications où des traitements se résout de façon concurrente (applications réseaux, par exemple)
- Améliorer les performances en optimisant l'utilisation des ressources
- Code plus difficile à comprendre, peu réutilisable et difficile à débuguer

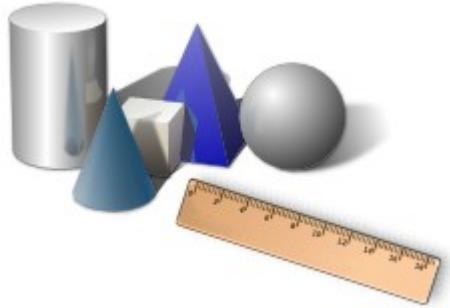


DEMO

- Dans cet atelier, nous simulons une banque possédant plusieurs comptes.
- Nous générerons au hasard des transactions qui déplacent de l'argent entre ces comptes.
 - Chaque compte possède un thread.
 - Chaque transaction déplace des quantités aléatoires d'argent, du compte desservi par le thread vers un autre compte choisi aléatoirement.
- Le code de cette simulation est assez simple. Il est principalement constitué de la classe **Bank** et de la méthode **transfer**.
 - Cette méthode transfère une certaine somme d'argent d'un compte vers un autre (nous ne traiterons pas des soldes négatifs).

Voici le code de la méthode transfer de la classe Bank :

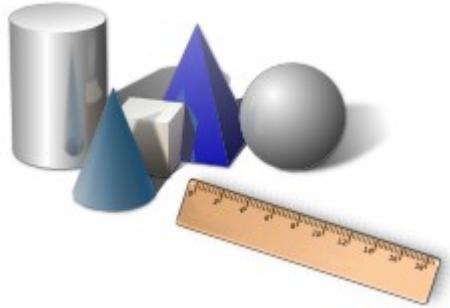
```
public void transfer(int from, int to, double amount)
// ATTENTION : cette méthode n'est pas sûre lorsqu'elle est appelée
// à partir de plusieurs threads
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.print(" %10.2f de %d à %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
}
```



DEMO

- Voici le code de la classe TransferRunnable. Sa méthode run sort en permanence de l'argent d'un compte bancaire spécifié. A chaque itération, la méthode run choisit au hasard un compte cible et un montant d'argent aléatoire, puis elle appelle transfer sur l'objet banque et s'endort.

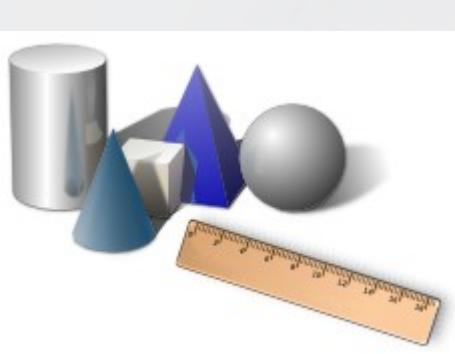
```
class TransferRunnable implements Runnable
{
    ...
    public void run()
    {
        try
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
        catch(InterruptedException e) {}
    }
}
```



DEMO

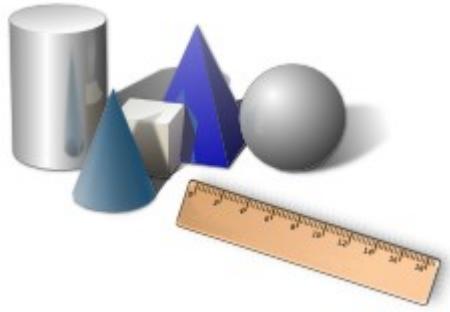
- Lorsque cette simulation est exécutée, nous ne savons pas combien d'argent se trouve dans chaque compte bancaire.
- En revanche, nous savons que la somme de tous les comptes doit rester constante, puisque nous effectuerons uniquement des transferts d'argent entre ces comptes.
- A la fin de chaque transaction, la méthode transfer recalcule le total et l'affiche.

```
/**  
 * Récupère la somme de tous les soldes.  
 * @return le solde  
 */  
public double getTotalBalance()  
{  
    double sum = 0;  
    for (double a : accounts)  
        sum += a;  
    return sum;  
}
```



DEMO

- Ouvrir le même projet threads et ouvrir le package : com.formation.async
- Parcourir les 3 classes avec le formateur :
 - Bank.java
 - TransferRunnable.java
 - UnsynchBankTest.java
- Exécuter la classe Test
 - Que remarquez-vous ?

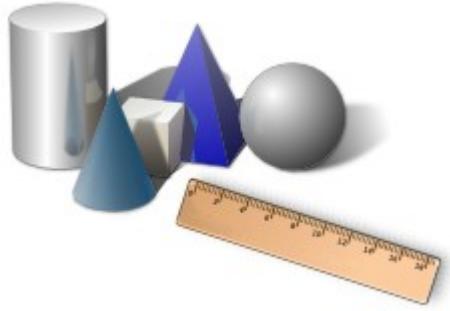


DEMO

➤ Voici un résultat typique :

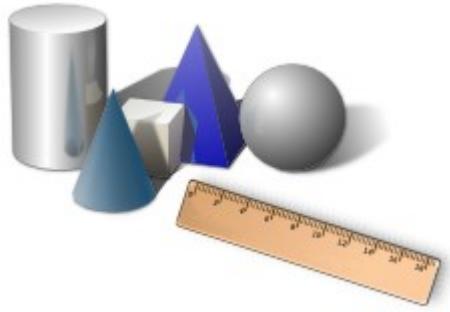
```
....  
Thread[Thread-11,5,main] 588.48 from 11 to 44 Total Balance: 100000.00  
Thread[Thread-12,5,main] 976.11 from 12 to 22 Total Balance: 100000.00  
Thread[Thread-14,5,main] 521.51 from 14 to 22 Total Balance: 100000.00  
Thread[Thread-13,5,main] 359.89 from 13 to 81 Total Balance: 100000.00  
....  
Thread[Thread-36,5,main] 401.71 from 36 to 73 Total Balance: 99291.06  
Thread[Thread-35,5,main] 691.46 from 35 to 77 Total Balance: 99291.06  
Thread[Thread-37,5,main] 78.64 from 37 to 3 Total Balance: 99291.06  
Thread[Thread-34,5,main] 197.11 from 34 to 69 Total Balance: 99291.06  
Thread[Thread-36,5,main] 85.96 from 36 to 4 Total Balance: 99291.06  
....  
Thread[Thread-4,5,main] Thread[Thread-33,5,main] 7.31 from 31 to 32
```

- Comme vous pouvez le constater, il y a une erreur importante.
- Pour certaines transactions, le solde reste à 100 000 \$, ce qui correspond au total correct pour 100 comptes de 10 000 \$ chacun.
 - Mais après un certain moment, le solde total est légèrement modifié.



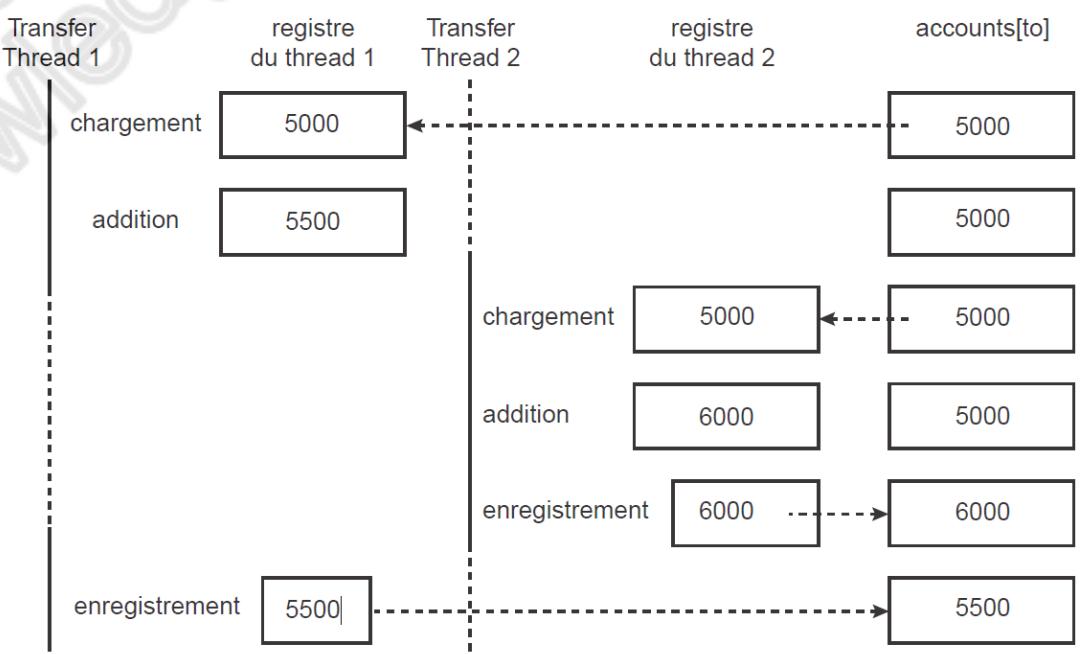
DEMO

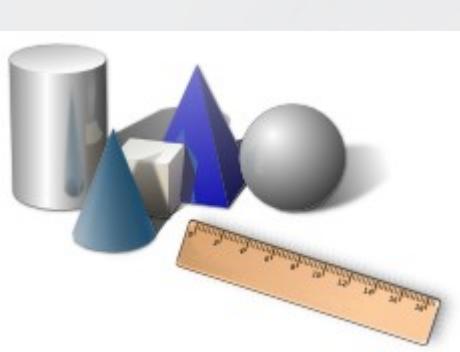
- **Explication :**
- Après un certain temps, des erreurs apparaissaient et une certaine quantité d'argent était soit perdue, soit ajoutée spontanément.
- Ce problème se présente lorsque deux threads essaient simultanément de mettre à jour un compte.
- Supposons que deux threads essaient d'exécuter l'instruction suivante simultanément :
 - `accounts[to] += amount;`
- Le problème de cette instruction est qu'elle n'est pas composée d'opérations **atomiques**. Cette instruction peut en effet être décomposée comme suit :
 1. Charger `accounts[to]` dans un registre.
 2. Ajouter `amount`.
 3. Placer le résultat dans `accounts[to]`.



DEMO

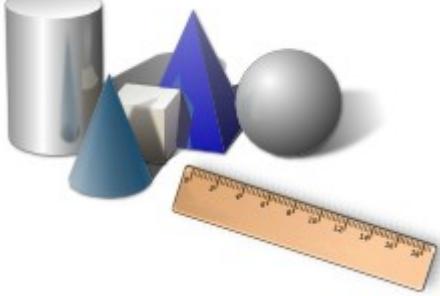
- Maintenant, supposons que le premier thread exécute les deux premières étapes, puis qu'il soit préempté.
- Supposons ensuite que le second thread se réveille et qu'il mette à jour la même entrée dans le tableau account. Le premier thread se réveille alors et termine sa troisième étape.
- Cette action annule la modification de l'autre thread. Par conséquent, le total n'est plus correct





DEMO

- Identifier dans le même exemple le code sensible qui doit être atomique
- Ajouter le nécessaire
- Exécuter
- Que remarquez-vous ?



Exercice 9 (voir cahier d'exercices)

Communication objets - Thread

- **Communication entre composants**
 - Affichage statique via une Applet et déplacement en boucle
 - Animation d'une balle avec Thread
 - Animation d'une balle avec plusieurs Thread
- **Bien que les Applet ne soient plus d'actualité, l'idée est de les utiliser afin de comprendre :**
 - la gestion d'un composant pris en charge par la JVM : l'Applet
 - d'avoir un visuel autre que la console système
 - de faire communiquer des objets entre eux : Applet et Thread
 - de mieux comprendre les threads via ce visuel

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Java Bean : Introduction

- Un JavaBean désigne tout simplement un composant réutilisable. Il est construit selon certains standards, définis dans les spécifications de la plate-forme et du langage Java eux-mêmes.
- Ainsi, **tout objet conforme à ces quelques règles peut être appelé un bean.**

Java Bean : Objectifs

- Un bean est un simple objet Java qui suit certaines contraintes, et représente généralement des données du monde réel.
 - **Les propriétés** : un bean est conçu pour être **paramétrable**. On appelle "propriétés" les champs non publics présents dans un bean. Qu'elles soient de type primitif ou objets, les propriétés permettent de paramétrer le bean, en y stockant des données.
 - **La sérialisation** : un bean est conçu pour pouvoir être **persistent**. La sérialisation est un processus qui permet de sauvegarder l'état d'un bean, et donne ainsi la possibilité de le restaurer par la suite. Ce mécanisme permet une persistance des données, voire de l'application elle-même.

Java Bean : Objectifs

- **La réutilisation** : un bean est un composant conçu pour être **réutilisable**. Ne contenant que des données ou du code métier, un tel composant n'a en effet pas de lien direct avec la couche de présentation, et peut également être distant de la couche d'accès aux données (nous verrons cela avec le modèle de conception DAO). C'est cette indépendance qui lui donne ce caractère réutilisable.
- **L'introspection** : un bean est conçu pour être **paramétrable de manière dynamique**. L'introspection est un processus qui permet de connaître le contenu d'un composant (attributs, méthodes et événements) de manière dynamique, sans disposer de son code source. C'est ce processus, couplé à certaines règles de normalisation, qui rend possible une découverte et un paramétrage dynamique du bean !

Java Bean : Structure

- Un bean :
 - Doit être une **classe publique**
 - Doit avoir au moins **un constructeur par défaut, public et sans paramètres**
 - Java l'ajoutera de lui-même si aucun constructeur n'est explicité ;
 - Peut implémenter l'interface **Serializable** il devient ainsi persistant et son état peut être sauvegardé
 - Ne doit pas avoir de champs publics
 - Peut définir **des propriétés (des champs non publics), qui doivent être accessibles via des méthodes publiques *getter* et *setter*, suivant des règles de nommage**

```
/* Cet objet est une classe publique */
public class MonBean{
    /* Cet objet ne possède aucun constructeur, Java lui assigne donc un constructeur par défaut public et sans paramètre. */
    /* Les champs de l'objet ne sont pas publics (ce sont donc des propriétés) */
    private String proprieteeNumero1;
    private int proprieteeNumero2;

    /* Les propriétés de l'objet sont accessibles via des getters et setters publics */
    public String getProprieteNumero1() {
        return this.proprieteNumero1;
    }

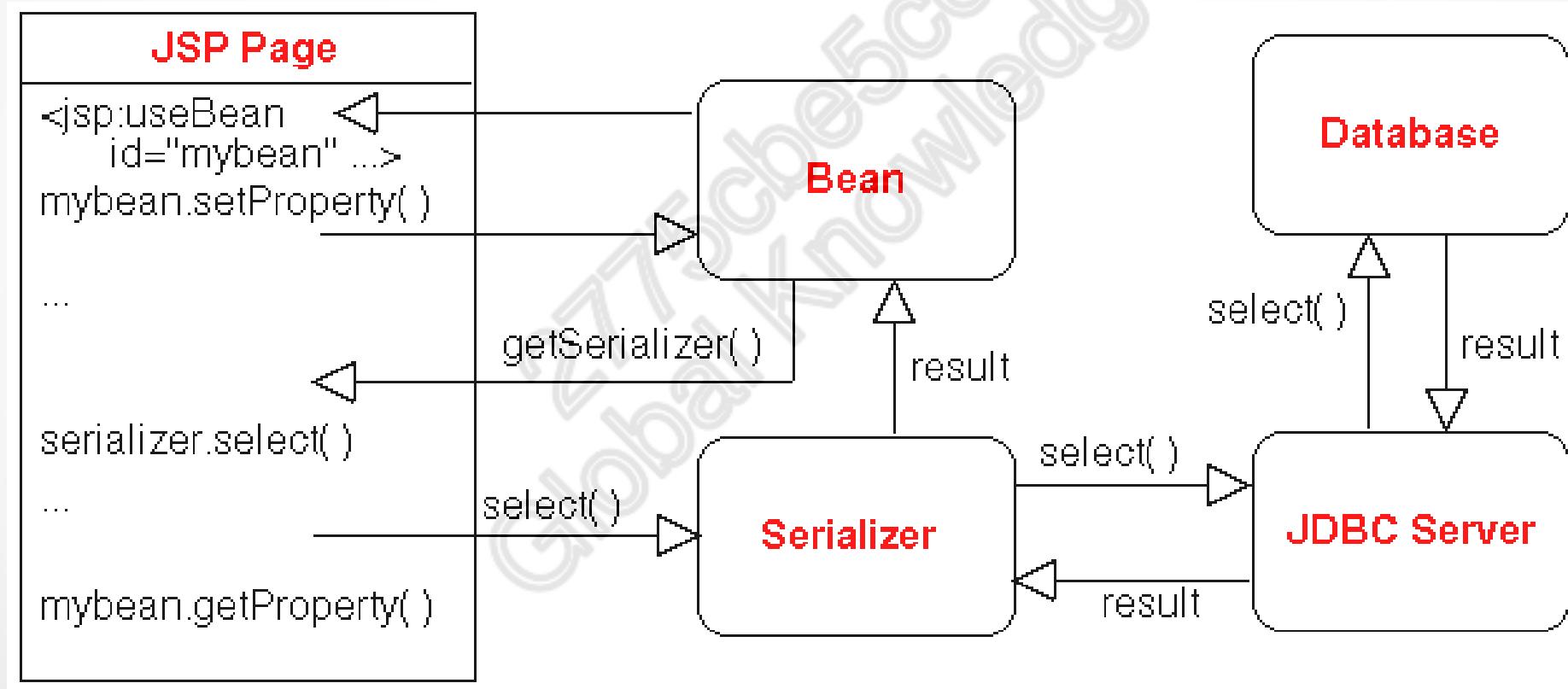
    public int getProprieteNumero2() {
        return this.proprieteNumero2;
    }

    public void setProprieteNumero1( String proprieteeNumero1 ) {
        this.proprieteNumero1 = proprieteeNumero1;
    }

    public void setProprieteNumero2( int proprieteeNumero2 ) {
        this.proprieteNumero2 = proprieteeNumero2;
    }

    /* Cet objet suit donc bien la structure énoncée : c'est un bean ! */
}
```

Java Bean : Utilisation



Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

JDBC : Introduction

- JDBC est un ensemble de classes et d'interfaces, composé de deux sous-ensembles :
 - JDBC API,
 - Destiné aux développeurs d'application Java, désirant interagir avec un SGBD R (Système de Gestion de Base de Données Relationnel).
 - JDBC Driver API,
 - Destiné aux développeurs de drivers (pilote), désirant interfacer un SGBD R en utilisant JDBC

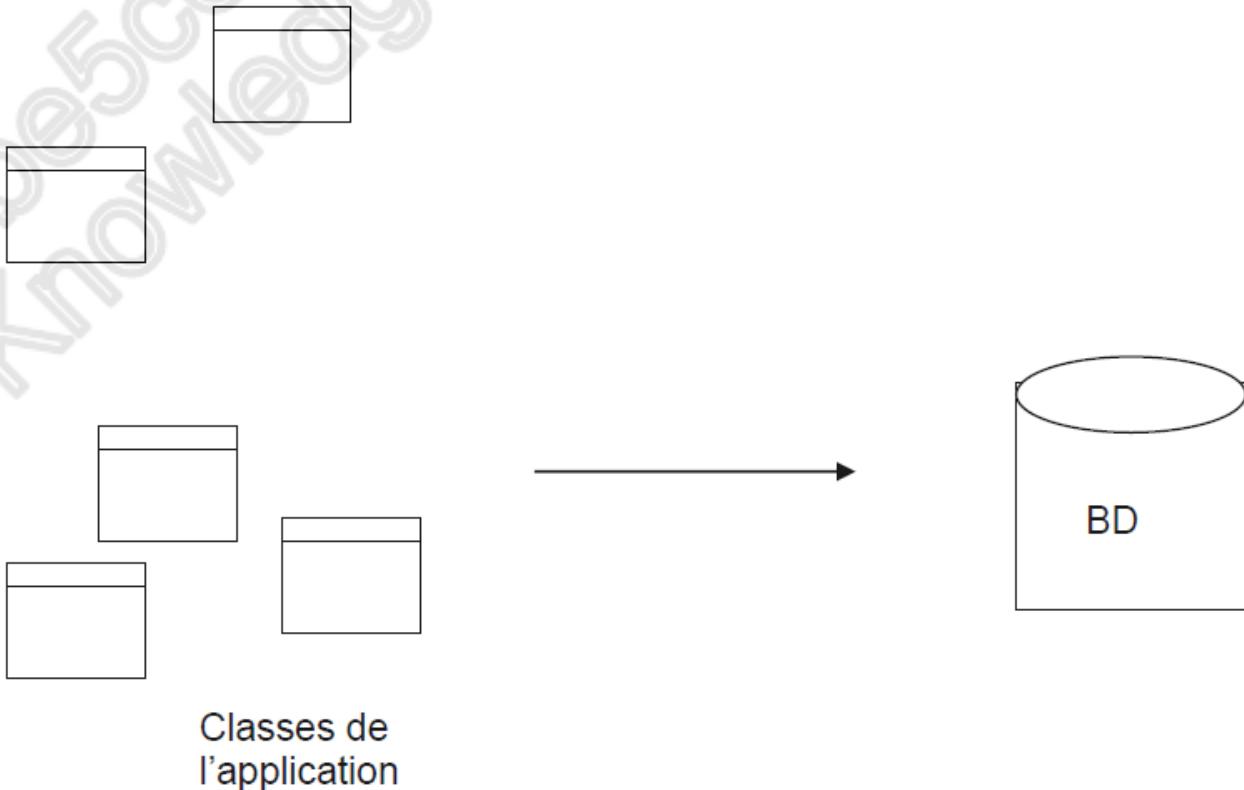
JDBC : Introduction

- Différentes architectures existent pour l'accès aux données :
 - L'accès direct
 - Les beans d'accès aux données
 - L'utilisation de classes dédiées, les DAO (Data Access Object)
 - Les EJB, Enterprise JavaBeans « entity »
 - JPA, Java Persistence API (Application Programming Interface)

JDBC : Architecture

➤ Accès direct :

- Solution la plus simple
 - Peut se faire depuis un servlet, via JDBC
 - Uniquement pour des applications très simples
 - Couplage fort avec la base de données :
 - Non extensible
 - Non réutilisable
 - Maintenance difficile
 - Ne supporte pas un modèle complexe



JDBC : Architecture

➤ Les beans d'accès aux données

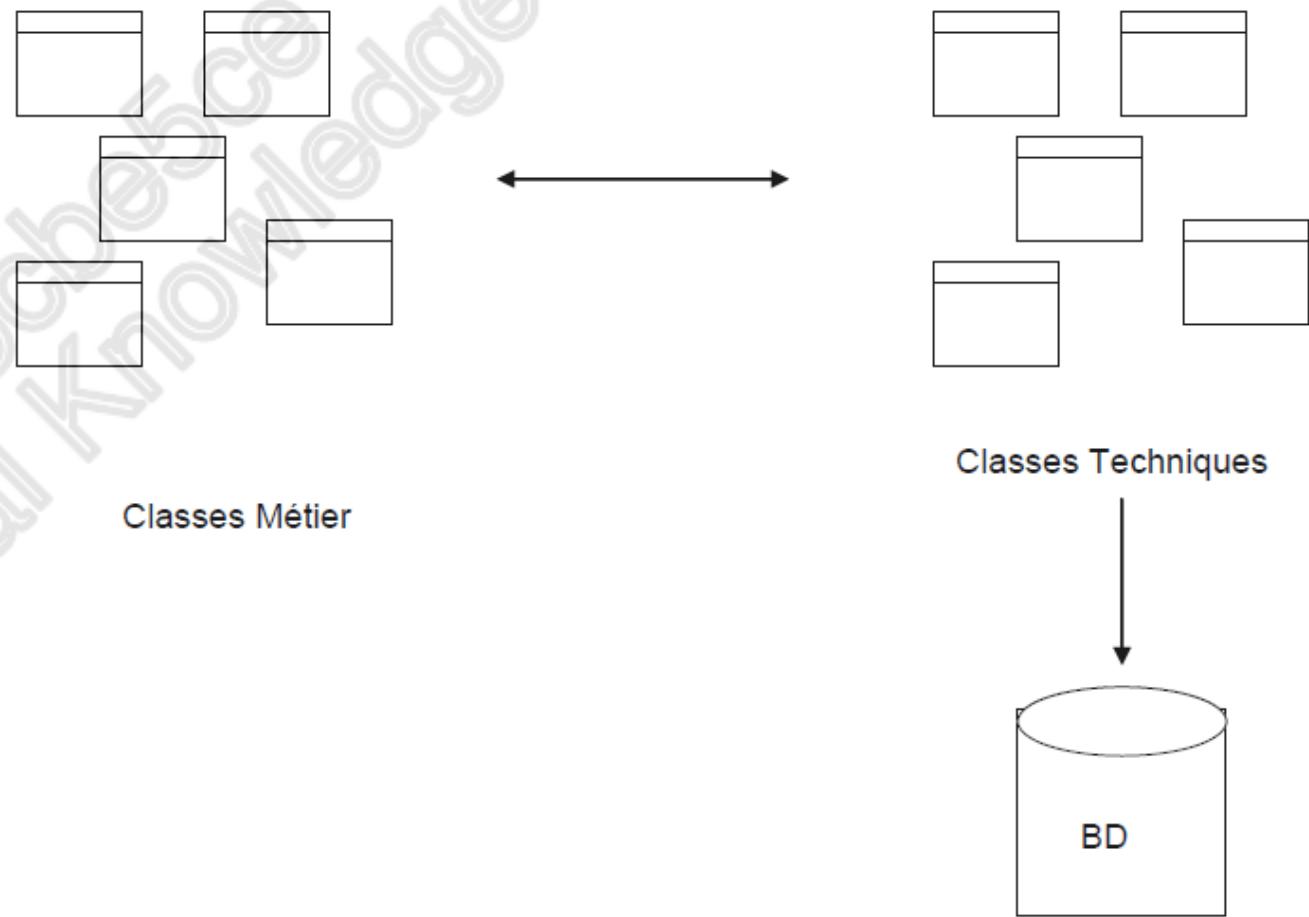
- Outils de génération automatique (avec WebSphere Studio d'IBM on utilise Database Access Servlet)
 - Uniquement pour des applications simples :
 - Courte durée de vie
 - Requêtes simples
 - Relation entre objets peu complexes
 - Peu extensible
 - Inadapté à un modèle complexe.
 - Nécessite l'achat d'un outil.

JDBC : Architecture

➤ Les classes dédiées :

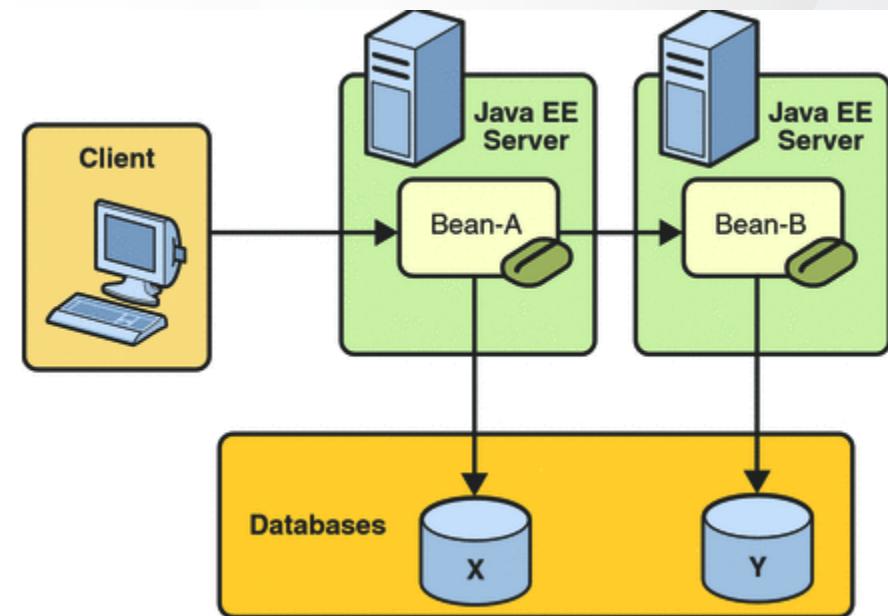
- Création de classes dédiées :
 - Doivent gérer la **persistence**
 - Se chargent des relations entre objets, des requêtes
 - **Séparation** de la logique métier et de la logique d'accès aux données

➤ Cette représentation est proposée par un Design pattern nommé DAO, Data Access Object. Le but de ce pattern est de gérer la persistence des données.



JDBC : Architecture

- **Les EJB, Enterprise JavaBeans « entity »**
- Les EJB appartiennent à la plateforme Java EE, pour les utiliser il faut un serveur d'applications en plus de la JVM.
- La technologie EJB, contient 3 types d'EJB, ceux permettant de manipuler les bases de données sont les EJB Entity. Ils permettent :
 - L'encapsulation d'une classe technique dans un composant dont les objets seront gérés par le serveur d'applications
 - De gérer la persistance des objets à l'aide d'une base de données.



JDBC : Architecture

- L'API JPA (Java Persistence API)
- L'utilisation pour la persistance d'un **mapping O/R** permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC :
 - Ce mapping permet d'assurer la transformation d'objets vers la base de données et vice versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression).
 - Des frameworks ORM existent pour éviter de tout coder, tels que Hibernate, EclipseLink, TopLink...
- JPA n'est qu'une API, EclipseLink en est l'implémentation officielle.

JDBC : l'API

➤ L'API JDBC :

- JDBC (*Java Data Base Connectivity*) permet l'accès à des bases de données avec le langage SQL, à partir d'un programme en Java
- Il est fourni par le paquetage **java.sql**
- JDBC est indépendant des SGBDs

JDBC : l'API

➤ Les composants majeurs de JDBC :

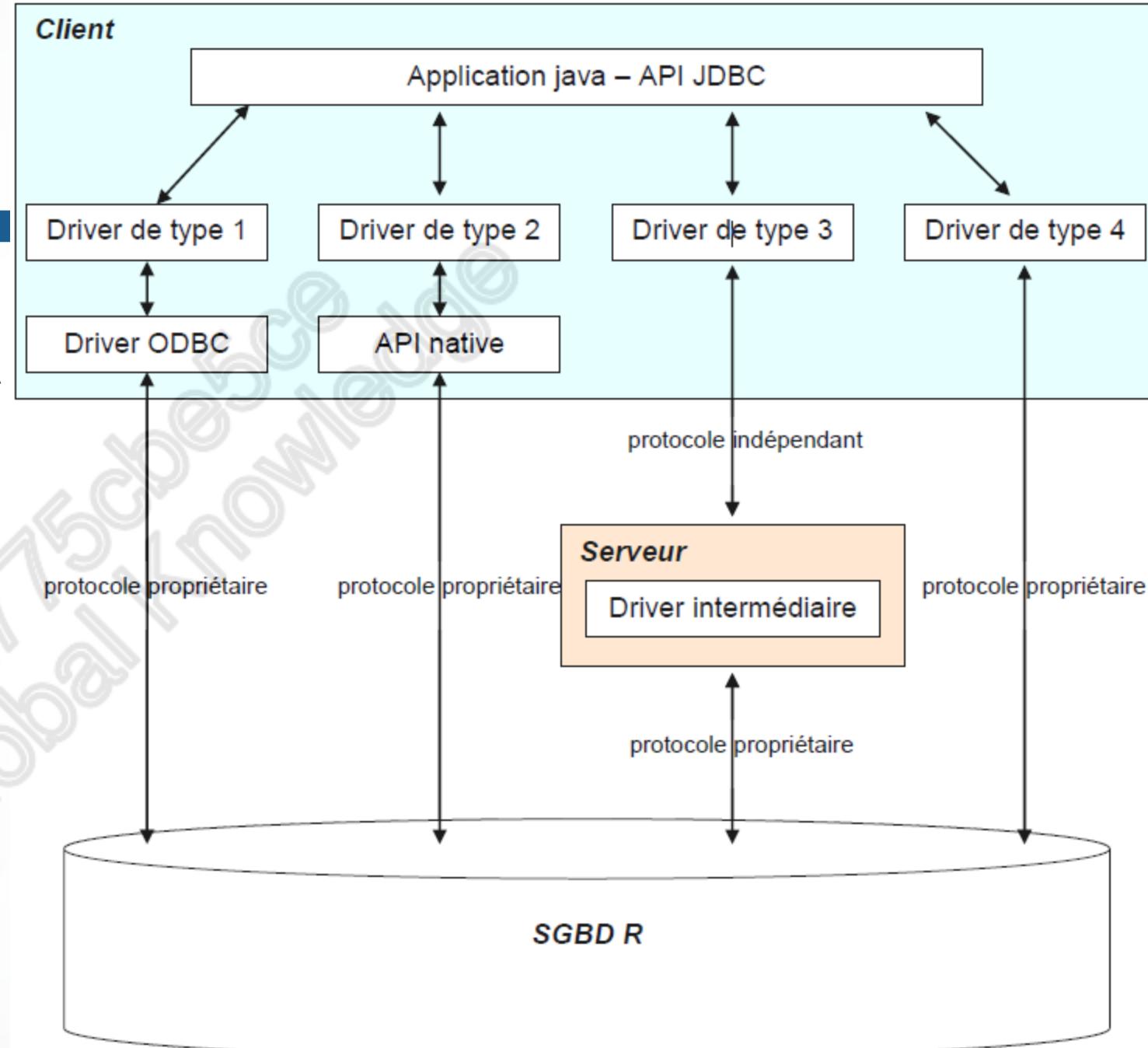
- Il faut manipuler quatre composants majeurs :
 - Les drivers : **DriverManager**
 - Les connexions : **Connection**
 - Les requêtes : **Statement**
 - L'ensemble de résultats : **ResultSet**

Drivers

- Pour travailler avec un SGBD, il faut disposer de classes Java qui implémentent les interfaces de JDBC.
- Un ensemble de telles classes est désigné sous le nom de ***driver***.
- Les drivers dépendent du SGBD auquel ils permettent d'accéder.
- Tous les SGBD importants du marché ont un driver (et même plusieurs), drivers fournis par l'éditeur du SGBD ou par des éditeurs de logiciels indépendants.

Types de drivers

- Type 1 : pont JDBC-ODBC
- Type 2 : driver qui fait appel à des fonctions natives non Java (le plus souvent en langage C) de l'API du SGBD que l'on veut utiliser
- Type 3 : driver qui permet l'utilisation d'un serveur *middleware*
- Type 4 : driver écrit entièrement en Java, qui utilise le protocole réseau du SGBD



Travailler avec JDBC

- 0. Chargement du driver
- 1. Ouvrir une connexion (**Connection**)
- 2. Créer des instructions SQL (**Statement**, **PreparedStatement** ou **CallableStatement**)
- 3. Lancer l'exécution de ces instructions :
 - Interroger la base (**executeQuery()**)
 - ou modifier la base (**executeUpdate()**)
 - ou tout autre ordre SQL (**execute()**)
- 4. Fermer la connexion (**close()**)

Chargement du driver

- Il existe plusieurs manières de charger le driver ou pilote. Cependant,
 - Le chargement est à faire une seule fois par JVM.
 - Plusieurs drivers peuvent être chargés en même temps.
- La classe **DriverManager** gère les drivers (instances de **Driver**) disponibles pour les différents SGBD utilisés par le programme Java
 - Pour qu'un driver soit utilisable, on doit charger sa classe en mémoire :
Class.forName("oracle.jdbc.driver.OracleDriver");
 - La classe crée alors une instance d'elle même et enregistre cette instance auprès de la classe **DriverManager**

Chargement du driver

- *Exemple de chargement de driver “MySQL” avec « Class » et gestion des exceptions :*
- *A privilégier pour nos tests.*

- N'oubliez surtout pas que la JVM a besoin de connaître les chemins d'accès où aller chercher les fichiers de code associés à vos classes. Pour ce faire l'environnement d'exécution Java utilise une variable d'environnement nommée CLASSPATH.

```
package labase;

public class TestLecture
{

    public static void main(String[] args)
    {
        try
        {
            // Enregistrement du driver de type 4 "Mysql"
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (ClassNotFoundException e)
        {
            // Affichage du message d'erreur
            System.out.println("Erreur charg. Driver " + e.getMessage());
        }
    }
}
```

Chargement du driver

- *Exemple de chargement de driver “MySQL” avec instantiation « **DriverManager** » et gestion des exceptions :*

- N'oubliez surtout pas que la JVM a besoin de connaître les chemins d'accès où aller chercher les fichiers de code associés à vos classes. Pour ce faire l'environnement d'exécution Java utilise une variable d'environnement nommée CLASSPATH.

```
package labase;

import java.sql.DriverManager;
import java.sql.SQLException;

public class TestLecture
{
    public static void main(String[] args)
    {
        try
        {
            DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        }
        catch (SQLException e)
        {
            // Affichage du message d'erreur
            System.out.println("Erreur charg. Driver " + e.getMessage());
        }
    }
}
```

Connexion à la base de données

- Pour obtenir une connexion à un SGBD, on demande cette connexion à la classe gestionnaire de drivers

```
// Chargement du driver :  
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");  
  
// Création de la connexion à une Base de données locale:  
String url = new String("jdbc:db2:exemple");  
Connection connex = DriverManager.getConnection(url);  
  
// Création de la connexion pour l'utilisateur nommé user dont le mot // de passe est « mdp »:  
String url = "jdbc:db2:exemple";  
Connection connex = DriverManager.getConnection(url, "user", "mdp");
```

- La classe **DriverManager** s'adresse à tour de rôle à tous les drivers qui se sont enregistrés (méthode **connect**), jusqu'à ce qu'un driver lui fournisse une connexion (ne renvoie pas **null**)

Connexion à la base de données

- Transactions :
 - Par défaut la connexion est en « *auto-commit* » :
 - Un commit est automatiquement lancé après chaque ordre SQL qui modifie la base
 - On peut enlever l'auto-commit par **conn.setAutoCommit(false)** (où conn est l'objet de type Connection utilisé pour accéder à la base de donnée)
 - **conn.commit()** valide la transaction
 - **conn.rollback()** annule la transaction

Créer des instructions SQL (Rappel)

- SQL : Structured Query Language
 - Un langage de requêtes pour communiquer avec une BDR
 - De nombreuses fonctionnalités !
 - Gestion, insertion, suppression, modification de données
 - Opérations arithmétiques et de comparaison
 - Affichage des données
 - Affectation
 - Fonctions d'agrégations

Créer des instructions SQL (Rappel)

➤ Concepts de base :

- **Table** : structure de données formée de colonnes et de lignes
- **Champ** :
 - Plus petit élément d'information (insécable)
 - Intersection d'une **colonne** et d'une ligne
- Enregistrement : collection de champs dont l'unité repose sur une ou plusieurs relations → **ligne**
- **Les clés** :
 - Clé primaire : identifiant unique d'un enregistrement dans une table (PK = Primary Key)
 - Clé étrangère : attribut correspondant à la clé primaire d'une autre table (FK = Foreign Key)

Table PROJET		
codpro	ville	type
1	LEVALLOIS-PERRET	4
2	COURBEVOIE	4
6	LOUVECIENNES	4
11	LES AUBRAIS	4
12	Paris Champ	3
17	PUTEAUX	4
22	COURBEVOIE	4
62	LILLE	3
120	WALLON-CAPPEL CEDEX	4
899	TRAPPES CEDEX	4
990	CARPENTRAS	4

↑
PK

↑

FK sur Table Type

Créer des instructions SQL (Rappel) : Construction d'une requête SQL

- Requête de sélection
- Le raisonnement est le suivant :

Quoi ? quelles informations obtenir ?

Où ? dans quelles tables ?

Comment ? (jointures, restrictions...)

Créer des instructions SQL (Rappel) : La projection

➤ Thème - Requête n° 1 :

- Afficher la liste des adhérents (Nom, Prénom, Code postal, Ville)

Quoi ? **SELECT** Nom_adh, Prénom_adh, Cp_adh, Ville_adh

Où ? **FROM** ADHERENT

	Nom_adh	Prénom_adh	Cp_adh	Ville_adh
▶	Dragan	Henri	42300	Roanne
	Ducreux	Albert	42300	Roanne
	Lamure	Alain	42300	Villerest
	Bonneval	Matthieu	42153	Riorges
	Bonneval	Jeanine	42153	Riorges
	Bonneval	Bruno	42153	Riorges
	Bonneval	Sophie	42153	Riorges
	Cartron	Jean-Pierre	42300	Villerest
	Cartron	Emmanuelle	69210	L'Arbresle
	Cartron	Agnès	42300	Roanne
	Favre	Dominique	71340	Iguerande
	Favre	Alexandre	71340	Iguerande
	Favre	Marie	71340	Iguerande
	Rocher	Romain	42430	Cherier
	Samuel	Danielle	42370	Villemontais
	Samuel	André	42370	Villemontais
	Samuel	Léa	42370	Villemontais
	Cherpin	Cindy	42330	St Galmier
	Kechida	Salim	42300	Roanne
	Penel	Kathleen	69240	Bourg de Thizy
	Chapuis	Nathalie	42300	Roanne
	Chapuis	Cyril	42300	Roanne
	Buisson	Samuel	42335	Roanne
	Meunier	Françoise	42370	St Alban les eaux
	Guenat	Maryline	42300	Roanne

Créer des instructions SQL (Rappel) : La projection

SELECT

Critère de projection. C'est un **ordre** qui décrit les champs que l'on désire extraire et afficher. **Au moins 1** champ à extraire.

FROM

C'est une clause qui précise les tables nécessaires à la requête.

La présence de **FROM** est obligatoire et suit **SELECT**.

Créer des instructions SQL (Rappel) : La restriction

➤ Thème - Requête n° 2 :

- Afficher la liste des adhérents (numéro, nom, prénom, index) dont l'index est inférieur à 20

	Num_adh	Nom_adh	Prénom_adh	Index_adh
▶	4	Bonneval	Matthieu	17,0
	43	Mineret	Nathalie	14,6
	50	Gerthoux	Myriam	19,9
	72	Lancet	Bernard	16,0
*				

Quoi ? SELECT Num_adh, Nom_adh, Prénom_adh, Index_adh

Où ? FROM ADHERENT

Comment ? WHERE Index_adh < 20

Créer des instructions SQL (Rappel) : La restriction

WHERE

C'est une clause qui décrit les critères de restriction.

S'il n'y a pas de restriction, la ligne WHERE n'existe pas.

Créer des instructions SQL (Rappel) : Le Tri

➤ Thème - Requête n° 3 :

- Afficher la liste des adhérents (nom, prénom, code postal, ville) dans l'ordre alphabétique du nom.

Quoi ? **SELECT** Nom_adh, Prénom_adh, Cp_adh, Ville_adh

Où ? **FROM** ADHERENT

ORDER BY Nom_adh

Tri sur le nom

Requête3 : Requête Sélection				
	Nom_adh	Prénom_adh	Cp_adh	Ville_adh
▶	Ballarin	Cécile	42120	Le Coteau
	Ballarin	Germinal	42120	Le Coteau
	Belgier	Céline	42155	Pouilly les Nonains
	Belivaud	André	42120	Perreux
	Berthy	Vanessa	42300	Roanne
	Bonneval	Sophie	42153	Riorges
	Bonneval	Matthieu	42153	Riorges
	Bonneval	Jeanine	42153	Riorges
	Bonneval	Bruno	42153	Riorges
	Briery	Pierre	42300	Roanne
	Brizard	Sébastien	42300	Villerest
	Brunelin	Frank	42300	Villerest
	Buisson	Samuel	42335	Roanne
	Carrera	Antonio	42300	Villerest
	Cartron	Agnès	42300	Roanne
Enr : ◀ ◀ 1 ▶ ▶ ▶* sur 87				

Créer des instructions SQL (Rappel) : Le Tri

ORDER BY

C'est une clause qui décrit les critères de TRI.

L'ordre croissant est traduit par ASC.

L'ordre décroissant est traduit par DESC.

En l'absence de paramètre, c'est ASC qui est retenu par défaut.

Créer des instructions SQL (Rappel) : La jointure

➤ Thème - Requête n° 4 :

- Afficher la liste des adhérents seniors (nom, prénom, intitulé de la formule) dans l'ordre croissant du nom et du prénom.

Quoi ? **SELECT** Nom_adh, Prénom_adh, intitulé_for

Où ? **FROM** ADHERENT

JOIN FORMULE

Comment ? **ON** ADHERENT.Code_for=FORMULE.Code_for

WHERE Code_cat=6

ORDER BY Nom_adh, Prénom_adh

Requête4 : Requête Séle...		
Nom_adh	Prénom_a	Intitulé_for
Belivaud	André	Semaine
Bonneval	Jeanine	Complet
Cartron	Jean-Pierre	Semaine
Cherpin	Maryline	Semaine
Cherpin	Thomas	Semaine
Combe	Laurence	Semaine
Combe	Maxime	Semaine
Ducreux	Albert	Complet
Lancet	Bernard	Semaine
Samuel	André	Semaine
Vacheron	Roger	Semaine
*		

Créer des instructions SQL (Rappel) : La jointure

JOIN

C'est une clause (associée au FROM) qui décrit la table de jointure.

Lorsqu'il y a jointure entre deux tables, les tables sont mentionnées en ligne FROM / JOIN et le critère de jointure (les colonnes qui concourent à la jointure) est décrit dans l'élément ON (Associé à JOIN).

ON

Le préfixage est nécessaire pour préciser de quelle table sont issus les champs concernés par la jointure quand il y a ambiguïté de nom de colonne.

La ligne WHERE sert aux autres critères de restriction.

Créer des instructions SQL

- Instance de l'interface **Statement**
- La méthode à appeler est différente suivant la nature de l'ordre SQL que l'on veut exécuter :
 - Consultation (select) : **executeQuery()** renvoie un **ResultSet** pour récupérer les lignes une à une
 - Modification des données (update, insert, delete) ou autres ordres SQL (create table,...) : **executeUpdate()** renvoie le nombre de lignes modifiées
 - Si on ne connaît pas à l'exécution la nature de l'ordre SQL à exécuter ou si l'ordre peut renvoyer plusieurs résultats : **execute()**

Créer des instructions SQL

```
// Création de la connexion à une Base de données locale:  
String url = "jdbc :db2 :exemple" ;  
Connection connex = DriverManager.getConnection(url, "user", "mdp") ;  
  
// Création de requête:  
Statement requete = connex.createStatement() ;  
  
// Exécution de la requête  
ResultSet resultat = requete.executeQuery("Select * from table") ;
```

Extraction des données : Interface ResultSet

- **executeQuery()** renvoie une instance de **ResultSet**
- L'interface **ResultSet** contient des méthodes pour récupérer dans le code Java les valeurs des colonnes des lignes renvoyées par le **SELECT** :
 - `getXXX(int numéroColonne)`
 - `getXXX(String nomColonne)`
- **XXX** désigne le type Java de la valeur que l'on va récupérer, par exemple `String`, `int` ou `double`

Extraction des données : Types JDBC/SQL

- Tous les SGBD n'ont pas les mêmes types SQL ; même pour les types de base on peut avoir des différences importantes
- Pour cacher ces différences, JDBC définit ses propres types SQL (représentés par des entiers)
- Ces types JDBC/SQL sont définis dans la classe **Types** sous forme de constantes nommées
- Ils sont utilisés par les programmeurs quand ils doivent préciser un type SQL (`setNull`, `setObject`, `registerOutParameter`)

Extraction des données : Types JDBC/SQL

- Il reste le problème de la correspondance entre les types Java et les types SQL
- Dans un programme JDBC, les méthodes getXXX, setXXX servent à préciser cette correspondance
- Par exemple, getString indique que l'on veut récupérer la donnée SQL dans une String
- C'est le rôle du **driver particulier** à chaque SGBD de faire les traductions correspondantes ; une exception peut être lancée si ça n'est pas possible

Exemple

0. Chargement du driver

1. Ouvrir une connexion

2. Créer une instruction SQL

3. Lancer l'exécution de l'instructions SQL

Récupération et conversion
de Type SQL vers Java (via
le Driver de Mysql)

4. Fermer la connexion

```
public static void main(String[] args) {  
    try {  
        // Enregistrement du driver :  
        Class.forName("com.mysql.jdbc.Driver");  
        String url = "jdbc:mysql://localhost:3306/BddJava";  
        // Connexion :  
        Connection connex = DriverManager.getConnection(url, "root", "root");  
        // Création de requête:  
        Statement requete = connex.createStatement();  
        // Exécution de la requête  
        ResultSet resultat = requete.executeQuery("Select Mle, Nom, Datnais from ingenieur");  
        // Parcours de l'ensemble résultat  
        while (resultat.next()) {  
            System.out.println(resultat.getInt("Mle") + "-"  
                + resultat.getString("Nom") + "-"  
                + resultat.getDate("Datnais"));  
        }  
        connex.close();  
    }  
    catch (ClassNotFoundException e) {  
        // Affichage du message d'erreur  
        System.out.println("erreur chargement Driver " + e.getMessage());  
    } catch (SQLException e) {  
        System.out.println("erreur SQL " + e.getMessage());  
    }  
}
```

Requêtes autres que SELECT

➤ Méthode **executeUpdate()**

- Pour toutes les requêtes de :
 - Mises à jour de données (INSERT, UPDATE ou DELETE),
 - Définition d'objets de la base (CREATE, ALTER, ...),
 - Gestion des droits (GRANT, REVOKE)...
- La méthode **executeUpdate** de l'objet Statement.
 - Attend en argument une requête SQL
 - Retourne un entier (type int) correspondant au nombre de lignes affectées par une requête de mise à jour de données. (0 si aucune mise à jour).
- Attention à l'auto commit.

Requêtes autres que SELECT

0. Chargement du driver

1. Ouvrir une connexion

➤ Exemple méthode **executeUpdate()**

➤ Le mode auto commit est désactivé

2. Créer une instruction SQL

3. Lancer l'exécution de l'instructions SQL

➤ La mise à jour a été effectuée, mais par la suite, on ne valide pas cette transaction. Retour arrière par la méthode *rollback()*

4. Fermer la connexion

```
public static void main(String[] args)
{
    Connection connex = null;
    try
    {
        // Enregistrement du driver :
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/logica";

        // Connexion :
        connex = DriverManager.getConnection(url, "root", "root");

        // Création de requête :
        Statement requete = connex.createStatement();

        // Désactivation de l'autovalidation
        connex.setAutoCommit(false);

        // Exécution de la requête :
        int nbLignes = requete.executeUpdate("UPDATE INGENIEUR" +
            " SET VILLE = 'MONTGERON' WHERE mle > 7000 ");
        System.out.println("Nb de mises à jour : " + nbLignes);

        connex.rollback();
        connex.close();

    } catch (ClassNotFoundException e)
    {
        // Affichage du message d'erreur
        System.out.println("Err. Charg. Driver " + e.getMessage());
    } catch (SQLException e)
    {
        System.out.println("erreur SQL " + e.getMessage() );
    }
}
```

Mises à jour groupées

- Depuis l'API JDBC 2.0, les objets issus des classes « Statement », « PreparedStatement » et « CallableStatement » sont capables d'enregistrer une liste de requêtes qui seront soumises ensemble au SGBD, en tant que lot, ce qui contribue à améliorer les performances du serveur.
- Ces objets sont créés et associés à une liste initialement vide. Vous pouvez exercer les actions suivantes :
 - Ajouter des commandes SQL à cette liste par la méthode « **addBatch()** »
 - Envoyer l'ensemble au SGBD par la méthode « **executeBatch()** »
 - Vider la liste grâce à la méthode « **clearBatch()** »

Mises à jour groupées

➤ Exemple

➤ Pour permettre une gestion correcte des erreurs, vous devrez désactiver le mode auto-commit.

```
// Création de la requête  
Statement requete = connex.createStatement();  
  
// Désactivation de l'autovalidation  
connex.setAutoCommit(false);  
  
// Préparation des requêtes :  
requete.addBatch("INSERT INTO Ingenieur (mle, nom, prenom)" +  
    "VALUES (9000, 'VANZO', 'ALAIN')");  
  
requete.addBatch("INSERT INTO Ingenieur (mle, nom, prenom)" +  
    "VALUES (9001, 'STADER', 'MARIA')");  
  
requete.addBatch("INSERT INTO Ingenieur (mle, nom, prenom)" +  
    "VALUES (9002, 'PANERAI', 'ROLANDO')");  
  
// Exécution de la requête et récupération du nombre de mises à jour:  
int[] nombreMaj = requete.executeBatch();
```

Autres types de requêtes : Instruction SQL paramétrée

- La plupart des SGBD (dont Oracle) peuvent n'analyser qu'une seule fois une requête exécutée un grand nombre de fois
- JDBC permet de profiter de ce type de fonctionnalité par l'utilisation de requêtes paramétrées ou de procédures stockées
- Les requêtes paramétrées sont associées aux instances de l'interface **PreparedStatement** qui héritent de l'interface Statement

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE emp SET sal = ?  
WHERE mle = ?");
```

- Les "?" indiquent les emplacements des paramètres

Autres types de requêtes : Instruction SQL paramétrée Avantages

- Un requête paramétrée est plus sûre, plus performante qu'une requête lancée par un Statement, surtout lorsque l'on exécute plusieurs fois la même requête avec des valeurs qui varient.
- L'ordre SQL est plus simple à écrire en **PreparedStatement** qu'en Statement

```
PreparedStatement pstmt = conn.prepareStatement("UPDATE emp SET sal = ?  
WHERE ville= ?");
```

```
Statement stmt = conn.createStatement();  
stmt.executeQuery("UPDATE emp SET sal =" + varSalair + " WHERE ville = '" +  
varVille + "');
```

Les " sont les chaînes en Java, ici ouverture de l'ordre

Concaténation obligatoire entre une chaîne et une variable

Définition d'une chaîne en SQL qu'il faut fermer

" fermeture de l'ordre

Autres types de requêtes : Instruction SQL paramétrée Avantages

- Elles améliorent aussi la portabilité car les méthodes setXXX n'ont pas à tenir compte des différences entre SGBD
- En effet, les SGBD n'utilisent pas tous les mêmes formats de date ('JJ/MM/AA' ou 'AAAA-MM-JJ' par exemple) ou de chaînes de caractères (pour les caractères d'« échappement »)
- Mais on peut aussi utiliser pour cela la syntaxe (un peu lourde) « SQL Escape »

Autres types de requêtes Requête paramétrée

– Valeurs des paramètres

- Les valeurs des paramètres sont données par les méthodes `setXXX(n, valeur)` où n représente la position du paramètre
- On choisit la méthode `setXXX` suivant le type SQL de la valeur que l'on veut mettre dans la base de données
- C'est au programmeur de passer une valeur Java du bon type à la méthode `setXXX`

Noter la différence avec `createStatement()` qui n'attend pas d'argument !

Syntaxe - Avec une requête paramétrée :

```
ResultSet resultat2;  
String sql2 = "select mle, nom, ville, cpt from ingenieur "  
        + "where ville = ? and cpt > ?";  
PreparedStatement requete2 = connex.prepareStatement(sql2);  
  
requete2.setString(1, "paris");  
requete2.setDouble(2, 10d);  
resultat2 = requete2.executeQuery();  
.../... // Exploitation du résultat de la 1ère exécution de la requête  
requete2.setString(1, "lille");  
requete2.setDouble(2, 9d);  
resultat2 = requete2.executeQuery();  
.../... // Exploitation du résultat de la 1ère exécution de la requête
```

Autres types de requêtes : Création d'une procédure stockée

- Les procédures stockées sont associées aux instances de l'interface **CallableStatement** qui héritent de l'interface PreparedStatement
- La création d'une instance de CallableStatement se fait par l'appel de la méthode prepareCall de la classe Connection
- On passe à cette méthode une chaîne de caractères qui décrit comment sera appelée la procédure stockée, et si la procédure renvoie une valeur ou non

Autres types de requêtes : Création d'une procédure stockée

On transmet un code d'article, on récupère un nombre d'articles :

```
.../...

// Création de requête liée à la procédure :
CallableStatement requete = connex.prepareCall("CALL nbArticles(?,?)");

// Valorisation des paramètres
requete.setString(1, "A350C");
requete.registerOutParameter(2, java.sql.Types.INTEGER);

// Exécution
int resultat = requete.executeUpdate();
int variable1 = requete.getInt(2);

.../...
```

Fermeture de la connexion

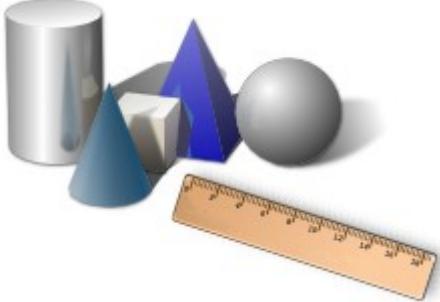
- Pour terminer, il faut fermer la connexion, en utilisant la méthode close() à l'objet Connection.

```
.../...
// Connexion
connex = DriverManager.getConnection(url, "root", "root");

// Création de requête
Statement requete = connex.createStatement();

// Exécution
ResultSet resultat = requete.executeQuery("Select codpro, libpro " +
                                              "from projet");
while (resultat.next()) {
    System.out.println(resultat.getInt("codpro") + " - " + resultat.getString("libpro"));
}

connex.close();
.../...
```



Exercice 10 (cahier d'exercices)

Base de données

➤ BASE DE DONNEES

- *Présentation*
- *Les requêtes*
- Compléments - Modèle Objet (Pour ceux qui maîtrisent le SQL et souhaitent réfléchir à une conception Orientée Objet)

DAO : Principe de base

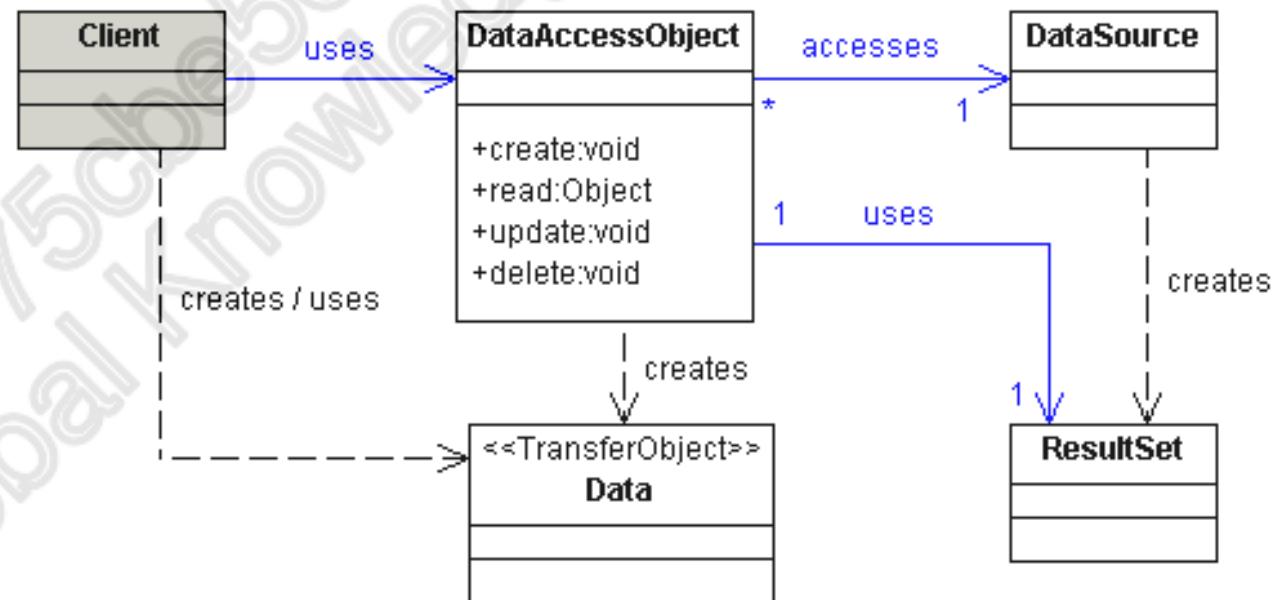
- Il est plus fréquent de changer la façon d'effectuer la persistance que de changer le modèle « métier »
- Pour faciliter les changements dans la persistance il faut isoler le plus possible le code qui gère la persistance
- La persistance est isolée dans des objets spécifiques, les **DAO** (Data Access Objects)

DAO : Principe de base

- Le DAO fournit les méthodes :
 - De lecture d'objet (récupération de données depuis la base de données),
 - De modification et de suppression d'objet,
 - D'ajout d'objet (sauvegarde en base de données).
- Il cache la connexion à la base de données ou tout autre outil d'accès aux données (XML, réseau ...).

DAO : Principe de base

- Le schéma suivant présente le pattern. Il décrit un DAO utilisant JDBC (DataSource et ResultSet).



- L'application Java, ici Client, n'a aucune information sur la base de données. Elle utilise le DAO. La couche DAO s'occupe de l'accès à la base et du langage SQL.

- **Un fait important**
 - Les appels de méthode distants sont beaucoup plus coûteux que les appels locaux
- **Le problème à résoudre**
 - Un client souhaite récupérer des données en interrogeant des objets distants non facilement transportables sur le réseau
 - Exemple : récupérer les nom, prénom, salaire et lieu de travail d'un employé
 - S'il utilise les accesseurs des classes des objets (getNom, getPrenom, getSalaire, getLieu), plusieurs appels distants sont nécessaires

La solution DTO

- Le client demande un objet qui contient toutes les valeurs dont il a besoin
- Cet objet, un Data Transfert Object (DTO), est construit sur le site distant et passé en une seule fois au client
- Un DTO contient l'état d'un ou de plusieurs objets métier, mais pas leur comportement
- Synonyme : Transfert Object (TO)

La solution DTO

➤ Exemples d'utilisation des DTO :

- Transporter les données d'un objet distant pas transportable sur le réseau (pas sérialisable)
- Transporter plusieurs objets distants en un seul appel distant ; par exemple une facture avec toutes les lignes de facture et les informations sur les produits
- Pour éviter les complications inutiles il faut éviter les DTOs si l'application est locale (pas distribuée)

DAO : Gestion de la persistance

➤ Le problème à résoudre

- Le code pour la persistance varie beaucoup
 - Avec le type de stockage (BD relationnelles, BD objet, fichiers simples, etc.)
 - Avec les implémentations des fournisseurs de SGBD
- Si les ordres de persistance sont imbriqués avec le code « métier », il est difficile de changer de source de données

➤ La solution

- Encapsuler le code lié à la persistance des données dans des objets DAO dont l'interface est indépendante du support de la persistance
- Le reste de l'application utilise les DAOs pour gérer la persistance, en utilisant des interfaces abstraites, indépendantes du support de persistance ; par exemple : **Employe getEmploye(int matricule)**

Utilité des DAOs

- Plus facile de modifier le code qui gère la persistance (changement de SGBD ou même de modèle de données)
- Factorise le code d'accès à la base de données
- Plus facile pour le spécialiste des BD d'optimiser les accès (ils n'ont pas à parcourir toute l'application pour examiner les ordres SQL)
- Sans doute le modèle de conception le plus utilisé dans le monde de la persistance

Gestion de la persistance : CRUD

- Cet acronyme désigne les opérations de base de la persistance : create, retrieve, update et delete
- Ces 4 opérations de base sont implémentées dans un DAO :
 - **Create** pour créer une nouvelle entité dans la base
 - **Retrieve** pour retrouver une ou plusieurs entités de la base
 - **Update** pour modifier une entité de la base
 - **Delete** pour supprimer une entité de la base
 - Plusieurs variantes pour les signatures de ces méthodes dans les DAOs

CRUD : create - paramètres

- Prend en paramètre l'état de la nouvelle entité
- Cet état peut être donné :
 - Par une série de paramètres des types des données :
 - **create(int id, String nom,...)**
 - Par un DTO :
 - **create(DTOxxx dto)**
 - Par l'objet métier que l'on veut rendre persistant :
 - **create(Article article)**

CRUD : create – type retour

- Le type retour peut être :
 - void (la variante la plus utilisée)
 - boolean pour indiquer si la création a pu avoir lieu (on peut utiliser une exception à la place)
 - L'identificateur de l'entité ajoutée (utile si la clé primaire est générée automatiquement dans la base de données)
 - Un objet métier ou un DTO correspondant à l'entité ajoutée

CRUD : retrieve

- 3 types de méthodes, suivant qu'elle retourne :
 - Un seul objet
 - Une collection d'objets
 - Une valeur calculée à partir de plusieurs entités (agrégation)
- Une méthode (ou un objet) qui retourne des données de la base de données est souvent appelée **finder**

CRUD : retrieve

- Exemples de finders :
 - On trouvera le plus souvent :
 - Une méthode **findById(id)** (ou d'un nom semblable...) qui retrouve une entité en donnant son identificateur dans la base
 - Une méthode **findAll()** qui retrouve toutes les entités du type géré par le DAO
 - Mais on trouvera aussi des finders qui cherchent suivant des critères quelconques ou suivant des critères bien précis (ces finders dépendent des traitements métier)

CRUD : retrieve

➤ **Finder qui retourne un objet :**

- On lui passe en paramètre un identificateur de l'entité cherchée
- Il retourne un objet métier qui correspond à l'entité cherchée, ou un DTO qui contient les données de l'entité cherchée
- Si le finder retourne un objet unique, il retourne null si rien n'a été trouvé

➤ **Finder qui retourne une collection :**

- On lui passe en paramètre le critère de sélection, sous une forme quelconque :
 - Objet ou valeurs « critère de sélection »
 - Objet « exemple » (à la « query by example »)
- Le type retour peut être très divers :
 - ResultSet
 - RowSet
 - Collection (Collection, List, Set,...) d'objets métier ou de DTOs
 - Tableau (rare)

CRUD : retrieve

- **Résultat vide**
 - Si le critère de la requête n'est vérifiée par aucune valeur, le finder doit retourner une « collection » (collection, resultset rowset ou tableau) vide
 - Retourner la valeur null obligerait à un cas particulier pour le traitement de la valeur renournée (« if (result == null) » avant une boucle qui parcourt le résultat)
- **Finder qui retourne une valeur calculée**
 - Les valeurs calculées à partir des données de plusieurs entités (exemple : total des salaires) peuvent s'obtenir à partir d'objets chargés en mémoire
 - Mais il peut être préférable de ne pas créer les objets et d'interroger directement la base de données qui est optimisée pour ce type de requête
 - Un DAO peut ainsi comporter une méthode qui renvoie le total des salaires des employés

CRUD : update

- Des variantes diverses pour les paramètres :
 - Identificateur + valeurs (plusieurs paramètres pour les valeurs ou un seul DTO)
 - L'objet métier dont on veut sauvegarder les modifications (nécessite un accès public aux valeurs qui seront modifiées)
- Le type retour peut être :
 - void
 - boolean pour indiquer si la modification a pu avoir lieu

CRUD : delete

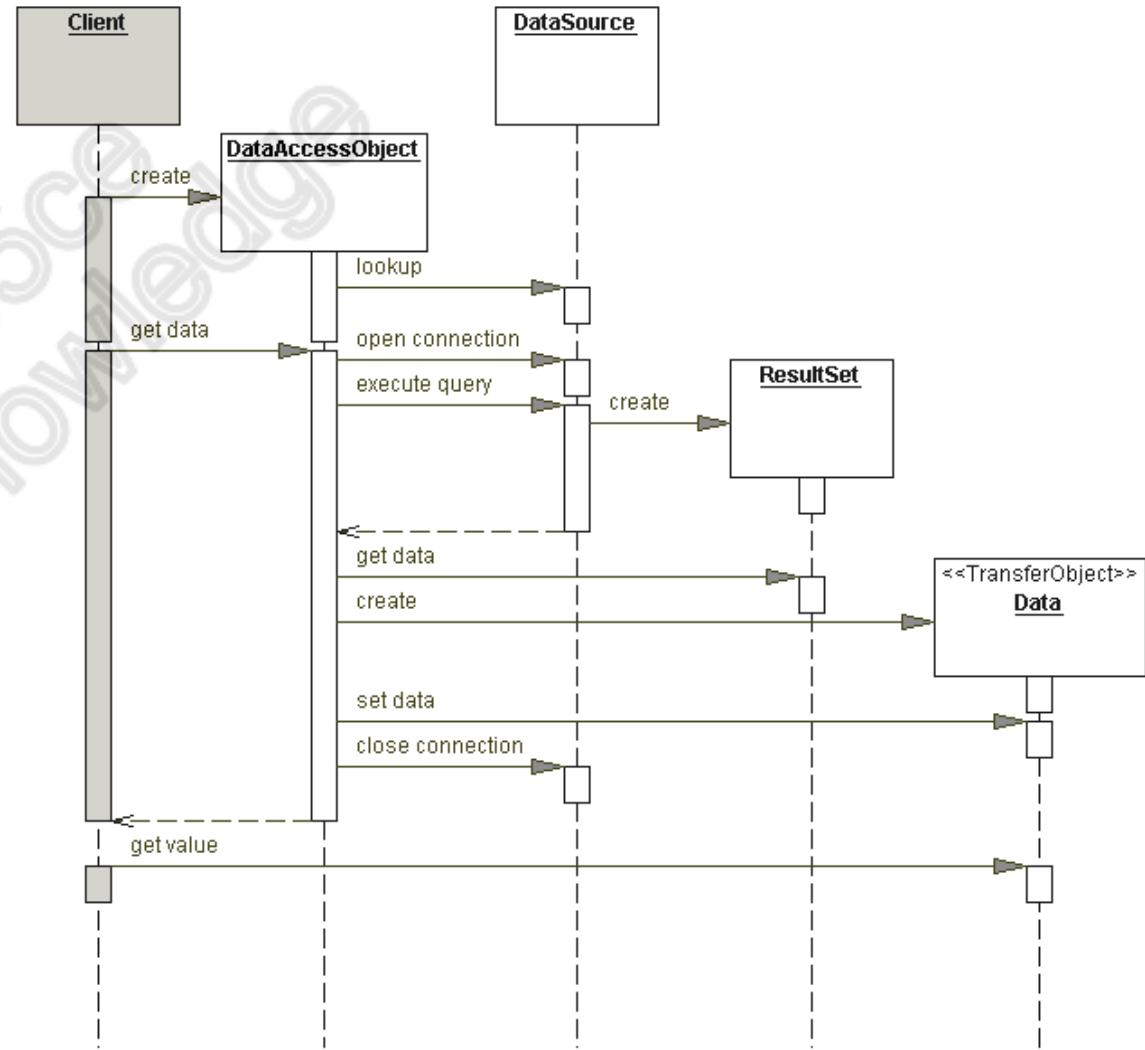
- Variantes pour les paramètres :
 - Identificateur de l'entité à supprimer dans la base
 - L'objet métier (ou un DTO) correspondant à l'entité à supprimer dans la base
- Variantes pour le type retour :
 - Void
 - boolean pour indiquer si la suppression a pu avoir lieu

DAO et exceptions

- Les méthodes des DAO peuvent lancer des exceptions puisqu'elles effectuent des opérations d'entrées-sorties
- Les exceptions ne doivent pas être liées à un type de DAO particulier si on veut pouvoir changer facilement de type de DAO
- Pour cela, on crée une ou plusieurs classes d'exception indépendantes du support de persistance, désignons-les par **DAException** (ou **DataAccessException** ou **DaoException**)
- Les méthodes des DAO attrapent les exceptions particulières, par exemple les **SQLException**, et relancent des **DAException** (auxquels sont chaînées les exceptions d'origine pour faciliter la mise au point)

Utilisation des DAO

- L'application cliente qui manipule les objets métier utilise les DAO.
- Les objets métier n'ont pas de référence à un DAO.



DAO et connexions

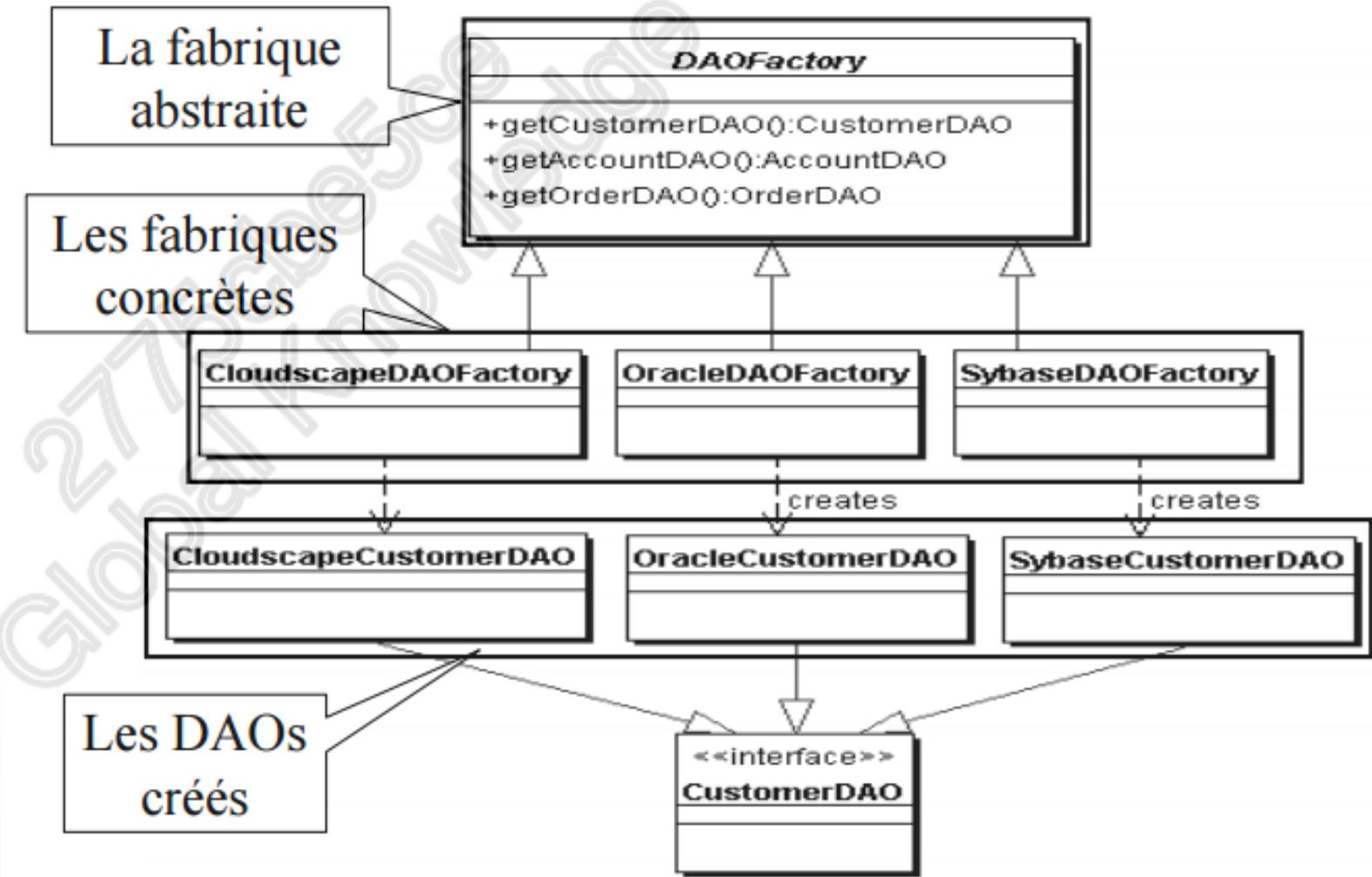
- Une connexion peut être ouverte au début des méthodes du DAO, et fermée à la fin des méthodes
- Cette stratégie va coûter chère si un pool de connexions n'est pas utilisé
- Il est préférable que les connexions soient ouvertes par les clients du DAO
- En ce cas, les connexions ouvertes doivent être passées au DAO
- Pour cela le DAO peut comporter une méthode
setConnection(Connection c)
 - (La façon de faire dépend de l'API de persistance que l'on utilise ; avec JPA on passera le manager d'entité et avec Hibernate la session)

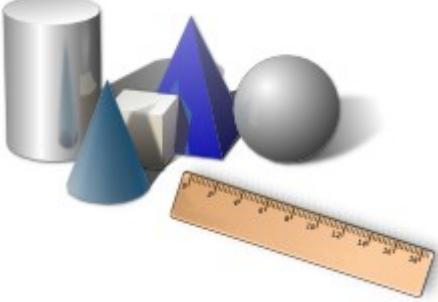
Design pattern Factory

- Une fabrique de création (ou factory) est une classe qui a pour rôle de construire des objets.
- Cette classe utilise des interfaces ou des classes abstraites pour masquer l'origine des objets.
- Ce pattern convient à la création d'un DAO, car il permet de cacher le type concret de la classe de l'instance créée. Ainsi une fabrique de DAO peut retourner un objet DAO sur Oracle ou un DAO DB2 pour IBM.

Design pattern Factory

➤ Exemple





Exercice 10 (suite) DAO

➤ BASE DE DONNEES - DAO

- Il s'agit d'un exercice de **maintenance d'application**.
- 1. Après avoir importé les sources dans un nouveau package, nous vous demandons, de faire fonctionner dans un premier temps l'exécutable **TestDAOList**. Pour cela, il vous faut créer l'interface **IngenieurDAO** mettant à disposition, des clients, les méthodes de MySQLIngenieurDAO. Nous vous conseillons de lire les sources impactés par cet exécutable, pour appréhender le pattern DAO.
- 2. Ensuite, il vous faudra lancer **TestDAO**, Cet exécutable crée un ingénieur en base, recherche et modifie un ingénieur. Après avoir testé ces méthodes il vous faut créer la suppression d'un ingénieur.

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Les entrées / sorties

- Dans la plupart des langages de programmation les notions d'entrées / sorties sont considérées comme une technique de base, car les manipulations de fichiers, notamment, sont très fréquentes.
- En Java, et pour des raisons de sécurité, on distingue deux cas :
 - Le cas des applications Java autonomes, où, comme dans n'importe quel autre langage, il est généralement fait un usage important de fichiers,
 - Le cas des applets Java qui, ne peuvent pas, en principe, accéder, tant en écriture qu'en lecture, aux fichiers de la machine sur laquelle s'exécute le navigateur (machine cliente).

La gestion des fichiers (1)

- La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.
- Un objet de la classe File peut représenter un fichier ou un répertoire.

La gestion des fichiers (2)

- Voici un aperçu de quelques constructeurs et méthodes de la classe **File** :
 - **File (String name)**
 - **File (String path, String name)**
 - **File (File dir, String name)**
 - boolean **isFile() / boolean isDirectory()**
 - boolean **mkdir()**
 - boolean **exists()**
 - boolean **delete()**
 - boolean **canWrite() / boolean canRead()**
 - File **getParentFile()**
 - long **lastModified()**

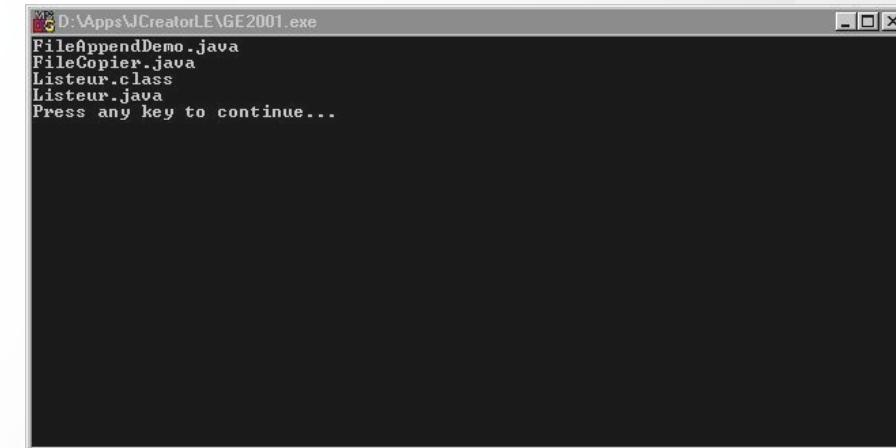
La gestion des fichiers (3)

```
import java.io.*;  
public class Listeur  
{  
    public static void main(String[] args)  
    {  
        litrep(new File("."));  
    }  
    public static void litrep(File rep)  
    {  
        if (rep.isDirectory())  
        { //liste les fichier du répertoire  
            String t[]=rep.list();  
            for (int i=0;i<t.length;i++)  
                System.out.println(t[i]);  
        }  
    }  
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire



La gestion des fichiers (4)

```
import java.io.*;
public class Listeur
{
    public static void main(String[] args)
    { litrep(new File( "c:\\\"));}

    public static void litrep(File rep)
    {
        File r2;
        if (rep.isDirectory())
        {String t[]={};rep.list();
        for (int i=0;i<t.length;i++)
        {
            r2=new File(rep.getAbsolutePath()+"\\\"+t[i]);
            if (r2.isDirectory()) litrep(r2);
            else System.out.println(r2.getAbsolutePath());
        }
    }
}
```

Le nom complet du fichier est rep\fichier

Pour chaque fichier, on regarde s'il est un répertoire.

Si le fichier est un répertoire litrep s'appelle récursivement elle-même

Notion de flux (1)

- Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).
- Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.
- Un flux peut être :
 - Soit une source d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
 - Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.

Notion de flux (2)

- Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :
 - Les fichiers,
 - Les tableaux de données en mémoire,
 - Les lignes de communication (connexion réseau)

Notion de flux (3)

- L'intérêt de la notion de flux est qu'elle permet une gestion homogène :
 - Quelle que soit la ressource associée au flux de données,
 - Quel que soit le flux (entrée ou sortie).
- Certains flux peuvent être associés à des filtres :
 - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

Notion de flux (4)

- Les flux sont regroupés dans le paquetage java.io
- Il existe de nombreuses classes représentant les flux
 - Il n'est pas toujours aisé de se repérer.
- Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :
 - E / S bufferisées, traduction de données, ...
- Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.

Flux d'octets et flux de caractères

- Il existe des flux de bas niveau et des flux de plus haut niveau (travaillant sur des données plus évoluées que les simples octets). Citons :
 - Les flux de caractères
 - Classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.
 - Les flux d'octets
 - Classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives.

Lecture de fichier

```
import java.io.*;
public class LireLigne
{
    public static void main(String[] args)
    {
        try
        {
            FileReader fr=new FileReader("c:\\windows\\system.ini");
            BufferedReader br= new BufferedReader(fr);
            while (br.ready())
                System.out.println(br.readLine());
            br.close();
        }
        catch (Exception e)
        {
            System.out.println("Erreur "+e);
        }
    }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileReader puis à partir de celui-ci, on crée un BufferedReader

Dans l'objet BufferedReader on dispose d'une méthode `readLine()`

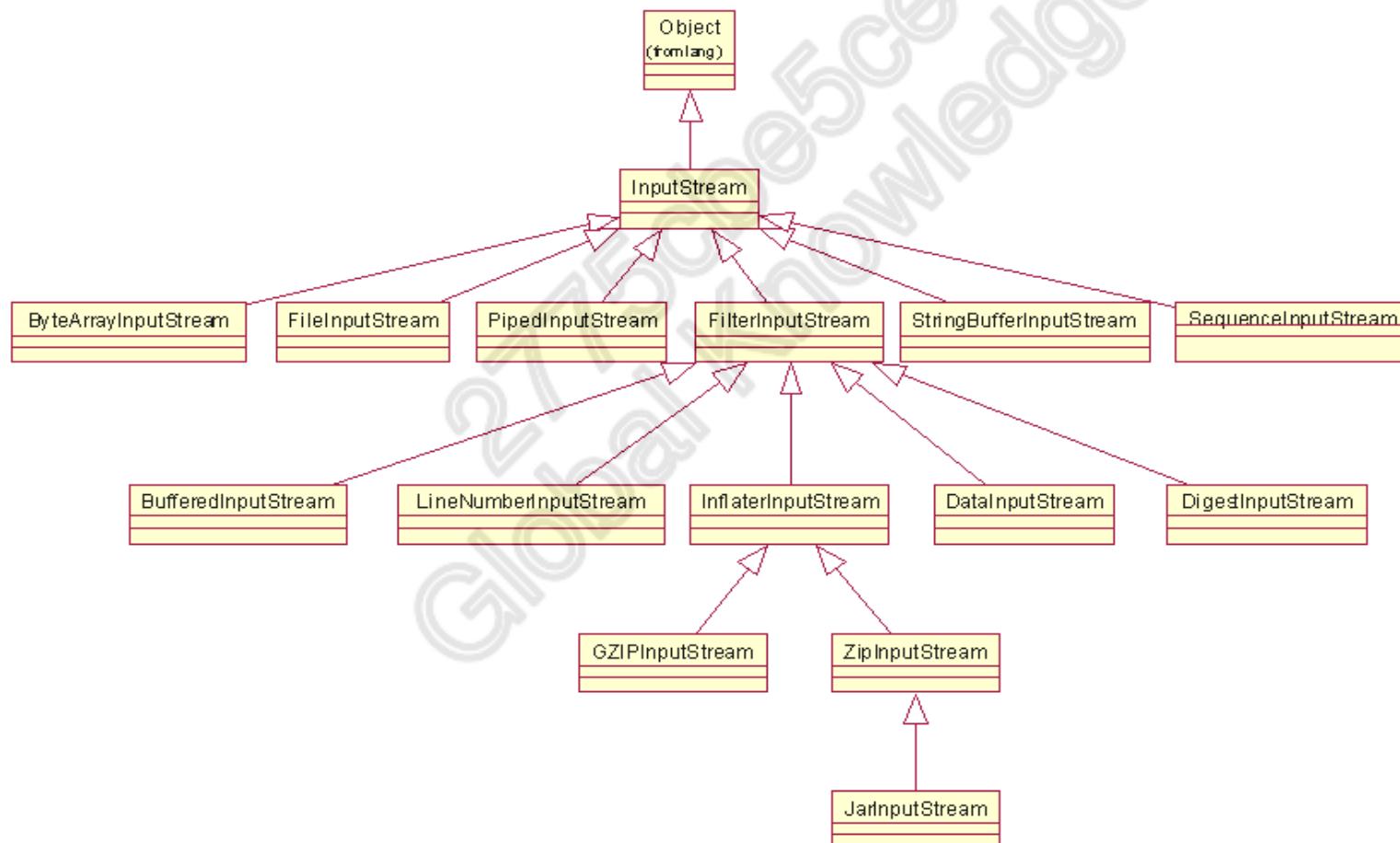
Ecriture dans un fichier

```
import java.io.*;
public class Ecrire
{
    public static void main(String[] args)
    {
        try
        {
            FileWriter fw=new FileWriter("c:\\temp\\essai.txt");
            BufferedWriter bw= new BufferedWriter(fw);
            bw.write("Ceci est mon fichier");
            bw.newLine();
            bw.write("Il est à moi...");
            bw.close();
        }
        catch (Exception e)
        { System.out.println("Erreur "+e);}
    }
}
```

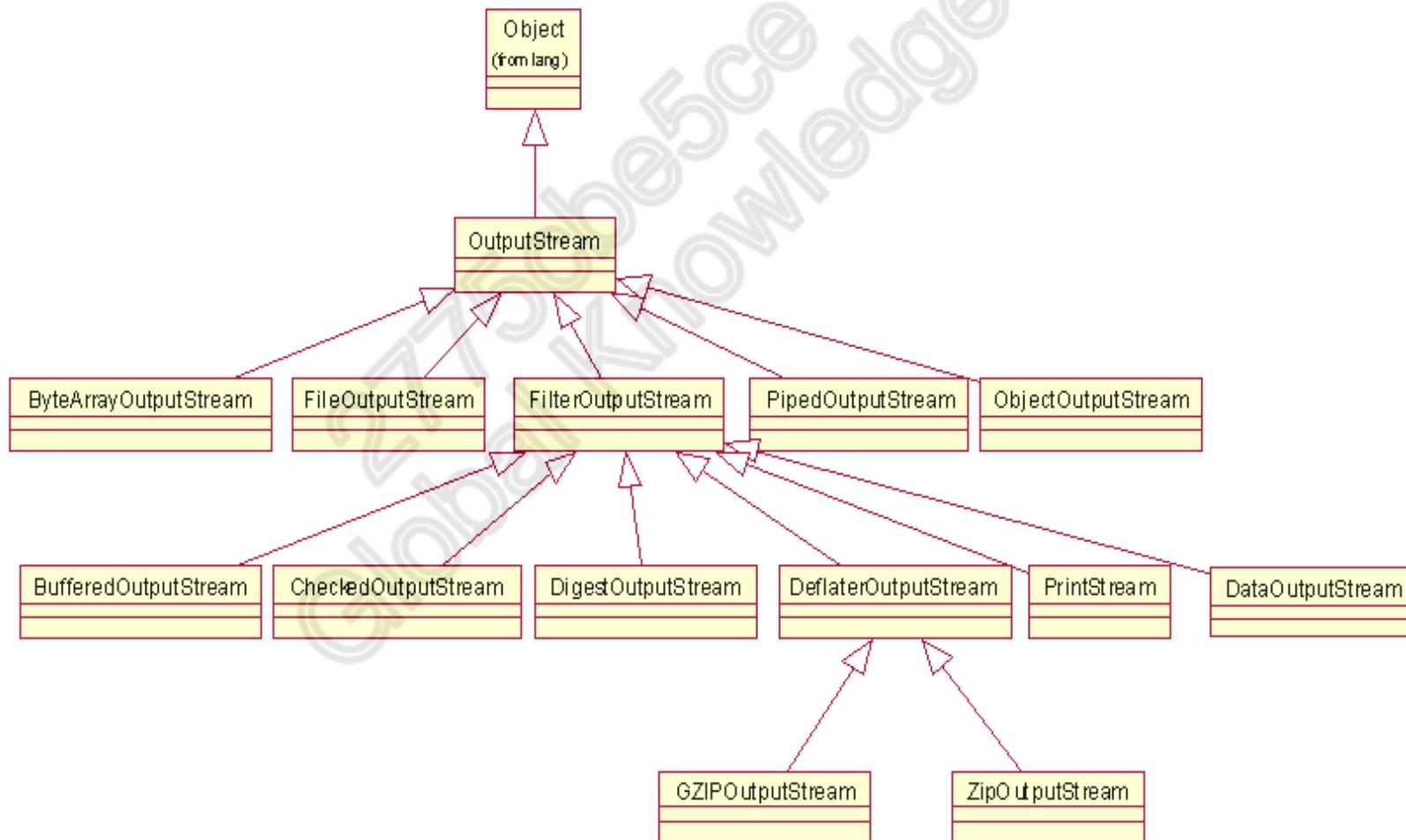
A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileWriter puis à partir de celui-ci, on crée un BufferedWriter

Attention, lorsqu'on a écrit, il ne faut pas oublier de fermer le fichier

La hiérarchie des flux d'octets en entrée



La hiérarchie des flux d'octets en sortie



Les flux d'octets : La classe InputStream (1)

- Un InputStream est un flux de lecture d'octets.
- InputStream est une classe abstraite.
 - Ses sous-classes concrètes permettent une mise en œuvre pratique.
 - Par exemple, **FileInputStream** permet la lecture d'octets dans un fichier.

La classe InputStream (2)

- Les méthodes principales qui peuvent être utilisées sur un InputStream sont :
 - **public abstract int read () throws IOException** qui retourne l'octet lu ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.
 - **int read (byte[] b)** qui emplit un tableau d'octets et retourne le nombre d'octets lus
 - **int read (byte [] b, int off, int len)** qui emplit un tableau d'octets à partir d'une position donnée et sur une longueur donnée
 - **void close ()** qui permet de fermer un flux,
 - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.

La classe InputStream (3)

- Les méthodes principales qui peuvent être utilisées sur un InputStream sont (suite) :
 - **int available ()** qui retourne le nombre d'octets prêts à être lus dans le flux,
 - Attention : Cette fonction permet d'être sûr qu'on ne fait pas une tentative de lecture bloquante. Au moment de la lecture effective, il se peut qu'il n'y ait plus d'octets disponibles.
 - **long skip (long n)** qui permet d'ignorer un certain nombre d'octets en provenance du flux. Cette fonction renvoie le nombre d'octets effectivement ignorés.

La classe OutputStream (1)

- Un OutputStream est un flux d'écriture d'octets.
- La classe OutputStream est abstraite.
- Les méthodes principales qui peuvent être utilisées sur un OutputStream sont :
 - **public abstract void write (int) throws IOException** qui écrit l'octet passé en paramètre,
 - **void write (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
 - **void write (byte [] b, int off, int len)** qui écrit les octets lus depuis un tableau d'octets à partir d'une position donnée et sur une longueur donnée

La classe OutputStream (2)

- Les méthodes principales qui peuvent être utilisées sur un OutputStream sont (suite) :
 - **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
 - **flush ()** qui permet de purger le tampon en cas d'écritures bufférissées.

Les flux d'octets

- Classe DataInputStream
 - Sous classes de InputStream permettent de lire tous les types de base de Java.
- Classe DataOutputStream
 - Sous classes de OutputStream permettent d'écrire tous les types de base de Java.
- Classes ZipOutputStream et ZipInputStream
 - Permettent de lire et d'écrire des fichiers dans le format de compression zip.

Empilement de flux filtrés (1)

- En Java, chaque type de flux est destiné à réaliser une tâche.
- Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe :
 - « Empile », à la façon des poupées russes, plusieurs flux ayant des comportements plus élémentaires.
 - On parle de flux filtrés.
 - Concrètement, il s'agit de passer, dans le constructeur d'un flux, un autre flux déjà existant pour combiner leurs caractéristiques.

Empilement de flux filtrés (2)

- **FileInputStream**
 - Permet de lire depuis un fichier mais ne sait lire que des octets.
- **DataInputStream**
 - Permet de combiner les octets pour fournir des méthodes de lecture de plus haut niveau (pour lire un double par exemple), mais ne sait pas lire depuis un fichier.
- Une combinaison des deux permet de combiner leurs caractéristiques :

```
FileInputStream fic = new FileInputStream ("fichier");
DataInputStream din = new DataInputStream (fic);
double d = din.readDouble ();
```

Empilement de flux filtrés (3)

Lecture bufférisée de nombres depuis un fichier

```
DataInputStream din = new DataInputStream(new BufferedInputStream(  
    new FileInputStream ("monfichier")));
```

Lecture de nombres dans un fichier au format zip

```
ZipInputStream zin = new ZipInputStream (  
    new FileInputStream ("monfichier.zip"));  
DataInputStream din = new DataInputStream (zin);
```

Flux de fichiers à accès direct (1)

- La classe RandomAccessFile
 - Permet de lire ou d'écrire dans un fichier à n'importe quel emplacement (par opposition aux fichiers à accès séquentiels).
- Elle implémente les interfaces DataInput et DataOutput
 - Permettent de lire ou d'écrire tous les types Java de base, les lignes, les chaînes de caractères ascii ou unicode, etc ...

Flux de fichiers à accès direct (2)

- Un fichier à accès direct peut être :
 - Ouvert en lecture seule (option "r") ou
 - en lecture / écriture (option "rw").
- Ces fichiers possèdent un pointeur de fichier qui indique constamment la donnée suivante :
 - La position de ce pointeur est donnée par **long getFilePointer ()** et celui-ci peut être déplacé à une position donnée grâce à **seek (long off)**.

Les flux de caractères (1)

- Ce sont des sous-classes de Reader et Writer.
- Ces flux utilisent le codage de caractères Unicode.
- Exemples :
 - Conversion des caractères saisis au clavier en caractères dans le codage par défaut

```
InputStreamReader in = new InputStreamReader (System.in);
```

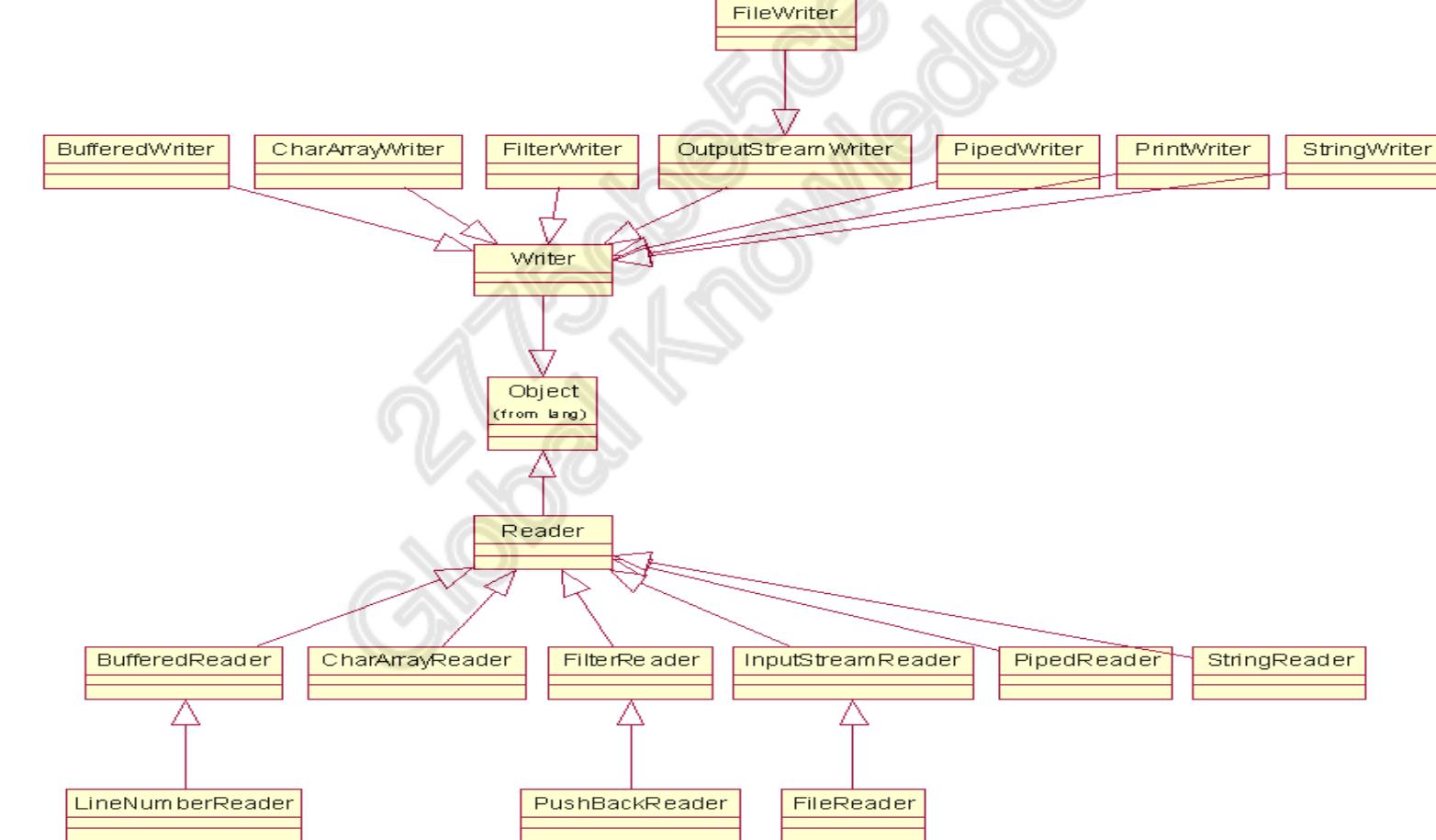
Conversion des caractères d'un fichier
avec un codage explicitement indiqué

```
InputStreamReader in = new InputStreamReader (  
    new FileInputStream ("chinois.txt"), "ISO2022CN");
```

Les flux de caractères (2)

- Pour écrire des chaînes de caractères et des nombres sous forme de texte :
 - On utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.
- Pour lire des chaînes de caractères sous forme de texte, il faut utiliser, par exemple :
 - **BufferedReader** qui possède une méthode **readLine()** .
 - Pour la lecture de nombres sous forme de texte, il n'existe pas de solution toute faite : il faut par exemple passer par des chaînes de caractères et les convertir en nombres.

La hiérarchie des flux de caractères



Les flux de données prédéfinis (1)

- Il existe 3 flux prédéfinis :
 - L'entrée standard System.in (instance de InputStream)
 - La sortie standard System.out (instance de PrintStream)
 - La sortie standard d'erreurs System.err (instance de PrintStream)

```
try {  
    int c;  
    while((c = System.in.read()) != -1) {  
        System.out.print(c);  
    }  
} catch(IOException e) {  
    System.out.print(e);  
}
```

Les flux de données prédéfinis (2)

- La classe `InputStream` ne propose que des méthodes élémentaires. Préférez la classe `BufferedReader`. qui permet de récupérer des chaînes de caractères.

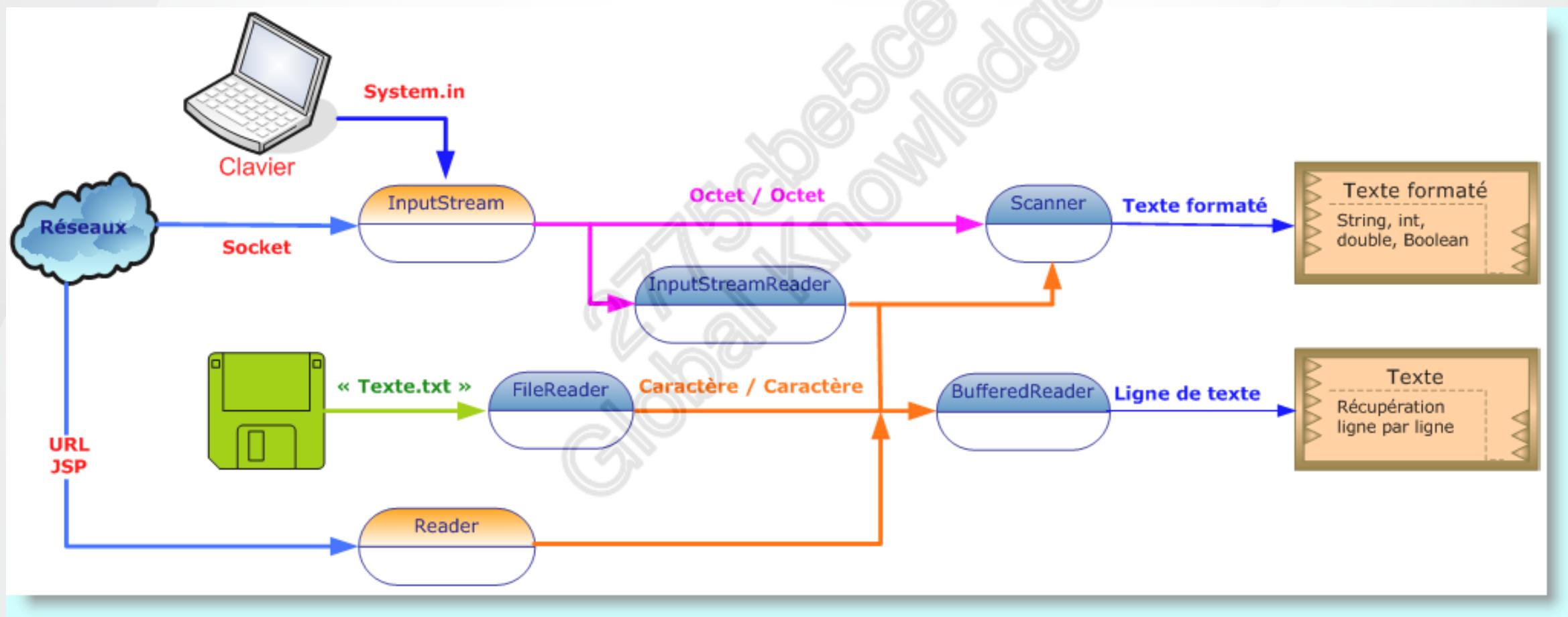
```
try {  
    Reader reader = new InputStreamReader(System.in);  
    BufferedReader keyboard = new BufferedReader(reader);  
  
    System.out.print("Entrez une ligne de texte : ");  
    String line = keyboard.readLine();  
    System.out.println("Vous avez saisi : " + line);  
} catch(IOException e) {  
    System.out.print(e);} 
```

Les flux de données prédéfinis (3)

- L'utilisation de flux "bufferisés" permet d'améliorer considérablement les performances

```
import java.io.*;  
  
class TestVitesseFlux {  
    public static void main(String[] args) {  
        FileInputStream fis; BufferedInputStream bis;  
        try {fis = new FileInputStream(new File("test.txt"));  
            bis = new BufferedInputStream(new FileInputStream(new  
File("test.txt")));  
            byte[] buf = new byte[8];  
            long startTime = System.currentTimeMillis();  
            while(fis.read(buf) != -1);  
            System.out.println("Temps de lecture avec FileInputStream : " +  
(System.currentTimeMillis() - startTime));  
            startTime = System.currentTimeMillis();  
            while(bis.read(buf) != -1);  
            System.out.println("Temps de lecture avec BufferedInputStream :  
" + (System.currentTimeMillis() - startTime));  
            fis.close(); bis.close(); }  
        catch (FileNotFoundException e) { e.printStackTrace(); } catch  
(IOException e) { e.printStackTrace(); } }  
}
```

Pour conclure



La sérialisation (1)

- La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).
- Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.
- La sérialisation peut donc être considérée comme une forme de persistance des données.

La sérialisation (2)

- 2 classes `ObjectInputStream` et `ObjectOutputStream` proposent, respectivement, les méthodes `readObject` et `writeObject`
- Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface `java.io.Serializable`.
- Il faut que la classe n'ait pas supprimé le constructeur par défaut

Exemple de sérialisation

- Soit un bean, implémentant `Serializable` et détenant ces 2 méthodes, où `s` représente le nom du fichier.
- La méthode `sauvegarde` a été exécutée.
- On exécuterait le code suivant pour restaurer le bean :

```
ClasseBean objRecup = (ClasseBean )  
ClasseBean.relecture(s);  
//La caste est obligatoire
```

```
void sauvegarde(String s) {  
    try {FileOutputStream f = new FileOutputStream(new File(s));  
        ObjectOutputStream oos = new ObjectOutputStream(f);  
        oos.writeObject(this); //objet à sauvegarder  
        oos.close();}  
    catch (Exception e)  
    { System.out.println("Erreur "+e);}  
}  
  
static Object relecture(String s) {  
    try {FileInputStream f = new FileInputStream(new File(s));  
        ObjectInputStream oos = new ObjectInputStream(f);  
        Object o=oos.readObject();  
        oos.close();  
        return o;}  
    catch (Exception e)  
    { System.out.println("Erreur "+e);  
        return null;}  
}
```

Class Scanner (1)

- Une classe injustement méconnue du JDK est Scanner (depuis la version 5 de Java)
- Fonctionnalités très intéressantes pour parser des chaînes de caractères, en extraire et convertir les composants.
- Un Scanner peut se brancher sur à peu près n'importe quelle source : InputStream, Readable (et donc Reader), File... et bien sûr une simple String.
- Ensuite utiliser les méthodes de type hasNext...() / next...(), ou alors les méthodes de type find...() / match() / group().

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();  
System.out.println("vous avez saisi " + i);
```

« Branchement » de la console système sur Scanner, pour lire la saisie utilisateur

Class Scanner (2)

- Méthode hasNext() / next()
 1. Découper la chaîne de caractères en *tokens* grâce à un délimiteur ; il s'agit par défaut d'un caractère "blanc" (espace, tabulation, retour à la ligne...), mais il est évidemment possible de fournir sa propre expression via la méthode useDelimiter (expression).
 2. Utiliser les méthodes de type hasNext...() et next...() pour parcourir, récupérer et convertir ces tokens.
- Les méthodes de type hasNext...() (hasNextInt(), hasNextFloat()...) fonctionnent sur le même principe qu'un Iterator.

Class Scanner (3)

- A l'aide de ces méthodes, il est très facile de parser une chaîne dont vous maîtrisez parfaitement le format, par exemple un fichier .csv :

```
String s =  
"Dalton;Joe;1.4\n" +  
"Dalton;Jack;1.6\n" +  
"Dalton;William;1.8\n" +  
"Dalton;Averell;2.0";  
  
Scanner scan = new Scanner(s);  
scan.useDelimiter(";|\n");  
scan.useLocale(Locale.US); // Pour les floats  
  
while(scan.hasNextLine()) {  
    System.out.printf("%2$s %1$s : %3$.1f m \n", scan.next(), scan.next(),  
        scan.nextFloat());  
}
```

Manipulation de fichier de propriétés

- Un fichier de propriétés est un fichier ayant une extension *properties*.
- Le contenu du fichier est structuré ainsi : **nom=valeur**
- Pour travailler avec ce fichier il faut :
 - Créer un objet Properties.
 - Créer une instance de FileInputStream sur ce fichier
 - Manipuler la méthode load de l'objet Properties avec l'instance FileInputStream.
 - Manipuler la méthode getProperty de l'objet Properties avec le nom de la propriété.

Manipulation de fichier de propriétés

- Soit le fichier database.properties :

```
jdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver  
jdbc.url=jdbc:odbc:nombase  
jdbc.username=java  
jdbc.password=java
```

- Nous pourrions extraire les informations ainsi →

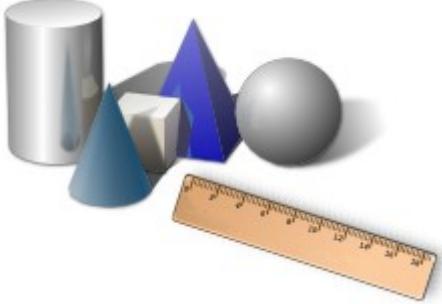
```
public static Connection getConnection() throws SQLException, IOException {  
  
    Properties props = new Properties();  
    FileInputStream in = new FileInputStream("database.properties");  
    props.load(in);  
    in.close();  
  
    String drivers = props.getProperty("jdbc.drivers");  
    if (drivers != null)  
        System.setProperty("jdbc.drivers", drivers);  
    String url = props.getProperty("jdbc.url");  
    String user = props.getProperty("jdbc.username");  
    String pwd = props.getProperty("jdbc.password");  
  
    return DriverManager.getConnection(url, user, pwd);  
}
```

Manipulation de fichier de propriétés

- Il existe des propriétés système, récupérables via la méthode `getProperty()` appliquée à la classe `System`. Cette méthode attend en argument la propriété souhaitée. *Un petit aperçu*
- Par exemple :

```
//constante séparateur de ligne  
String crtfs = System.getProperties().getProperty("line.separator");
```

Catégorie	Propriété
Propriétés du système :	os.name
	os.version
	os.arch
Propriétés de fichiers :	file.separator
	path.separator
	line.separator
Propriétés de la JVM :	java.version
	java.home
	java.class.path
	java.vendor
	java.vendor.url
Propriétés utilisateur :	user.name
	user.home
	user.dir
	user.country
	user.language



Exercice 11- Input / Output

➤ ENTREES/SORTIES

- Manipulation d'un fichier de propriétés
- Sauvegarde/Restauration de l'objet Compte

Sommaire

- **Chapitre 1 :** Introduction
- **Chapitre 2 :** Le langage JAVA
- **Chapitre 3 :** Compléments au langage Java
- **Chapitre 4 :** Classes Abstraites & Interfaces
- **Chapitre 5 :** Exceptions
- **Chapitre 6 :** Les génériques
- **Chapitre 7 :** Les expressions lambda
- **Chapitre 8 :** Les Streams
- **Chapitre 9 :** Thread
- **Chapitre 10 :** JavaBean
- **Chapitre 11 :** Accès au base de données
- **Chapitre 12 :** Entrées Sorties
- **Chapitre 13 :** JavaFX

Objectifs du chapitre

- Introduction
- Architecture JavaFX
- Cycle de vie d'une application JavaFX
- Les propriétés JavaFX
- Binding API
- High-level binding API
- Low-level binding API
- Composants graphiques
- Le langage FXML

Introduction



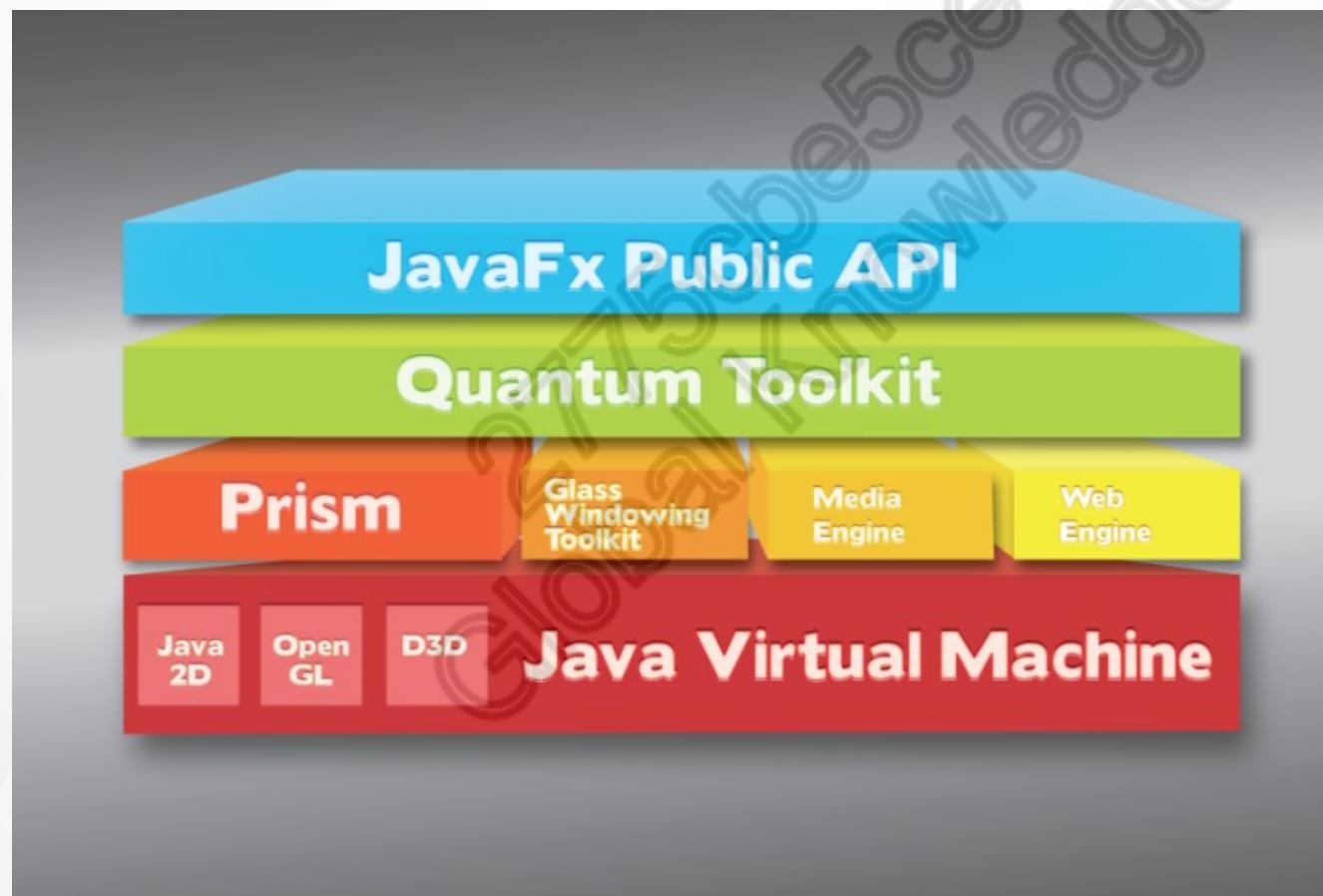
Introduction

- JavaFX est la nouvelle boîte à outils d'interface utilisateur destinée aux applications client basées sur Java s'exécutant sur des périphériques de bureau, intégrés et mobiles. JavaFX fait partie du JDK 8 et est fourni en tant qu'API Java pure.
- Entre autres, les fonctionnalités suivantes sont supportées :
 - Graphiques 2D et 3D accélérés
 - Contrôles d'interface utilisateur,
 - dispositions et graphiques
 - Support audio et vidéo
 - Effets et animations
 - Prise en charge de HTML5
 - Liaisons, CSS, FXML

Introduction

- Pour fournir des performances optimales, JavaFX utilise différents moteurs de rendu natifs en fonction de la plate-forme sur laquelle il s'exécute.
- Sous Windows, par exemple, Direct3D est utilisé, alors que sur la plupart des systèmes, il utilise OpenGL

Architecture JavaFX



Architecture JavaFX

- JavaFX fournit une API complète avec un ensemble de classes et d'interfaces pour créer des applications à interface graphique avec des graphiques riches.
- Les packages importants de cette API sont :
 - **javafx.animation** : Contient des classes pour ajouter des animations basées sur la transition, telles que le remplissage, le fondu, la rotation, la mise à l'échelle et la traduction, aux nœuds JavaFX.
 - **javafx.application** : Contient un ensemble de classes responsables du cycle de vie de l'application JavaFX.
 - **javafx.css** : Contient des classes pour ajouter un style similaire à CSS aux applications d'interface graphique JavaFX.

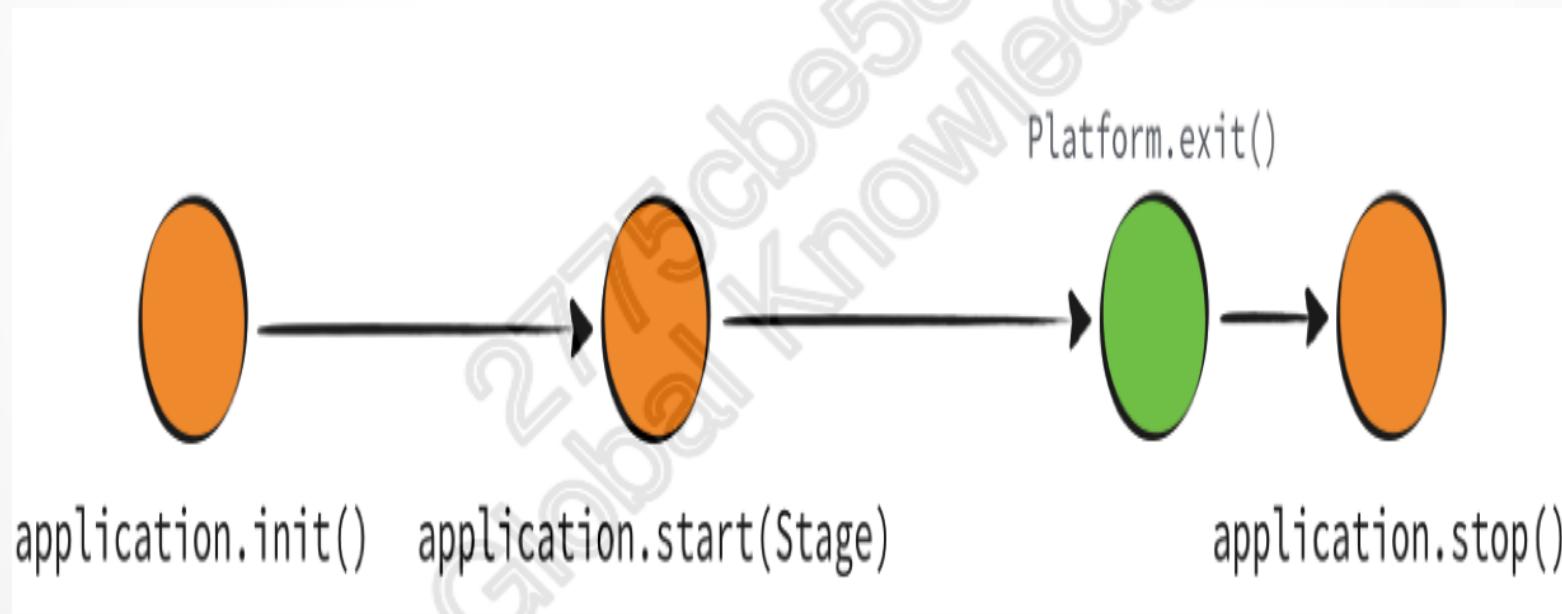
Architecture JavaFX

- **javafx.event** : Contient des classes et des interfaces pour livrer et gérer des événements JavaFX.
- **javafx.geometry** : Contient des classes pour définir des objets 2D et y effectuer des opérations.
- **javafx.stage** : Contient les classes de conteneur de niveau supérieur pour l'application JavaFX.
- **javafx.scene** : Fournit des classes et des interfaces permettant de prendre en charge le graphe de scène. En outre, il fournit également des sous-packages tels que canevas, graphique, contrôle, effet, image, entrée, mise en page, support, peinture, forme, texte, transformation, Web, etc.

Cycle de vie d'une application JavaFX

- Une application JavaFX s'exécute dans le cycle de vie JavaFX par défaut défini par la classe Application.
- Au lancement de l'application, la méthode launch() doit être appelée. Cette méthode appelle la méthode init().
- La méthode init () est appelée avant la création du thread d'application JavaFX; par conséquent, aucune opération spécifique à l'interface utilisateur n'est autorisée.
- Puis la méthode start() obligatoirement implémentée par le client.
- Une boucle est lancée pour gérer les évènements.
- Si un signal d'exit est lancé, la méthode stop() est appelée.

Cycle de vie d'une application JavaFX



Application « HelloWorldJavaFX »

```
Import javafx.application.Application;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.layout.StackPane;  
import javafx.stage.Stage;  
  
public class HelloWorldJavaFX extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        Button button = new Button("Hello World JAVAFX");  
        button.setOnAction (e -> System.out.println("Hello  
World"));  
  
        StackPane myPane = new StackPane();  
        myPane.getChildren().add(button);  
  
        Scene myScene = new Scene(myPane);
```

Hériter
d'Application

Redéfinir start(..)

Ajout bouton et
Gestion
événementielle

conteneur

Application « HelloWorldJavaFX »

```
        primaryStage.setScene(myScene);
        primaryStage.setWidth(400);
        primaryStage.setHeight(300);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Gestion de la fenêtre

Lancement de l'IHM

Application « HelloWorldJavaFX »

- La classe abstraite Application fournit des implémentations pour les méthodes init() et stop() qui ne sont pas à implémenter.
- Seule la méthode start() doit être redéfinie obligatoirement.
- La méthode static main(), point d'entrée de toute application Java, appelle la méthode launch() d'Application.
- La classe HelloWorldJavaFX implémente la méthode start() qui reçoit l'objet javafx.stage.Stage qui représente la fenêtre principale de l'application. On y ajoute la gestion des différents composants graphiques.

Application « HelloWorldJavaFX »



Les propriétés JavaFX

- Les propriétés JavaFX sont basées sur les propriétés JavaBeans standard.
- Le runtime JavaFX fournit des implémentations par défaut pour tous les types de propriétés, qui peuvent être utilisées dans une classe.

Les propriétés JavaFX

```
private final IntegerProperty size =  
new SimpleIntegerProperty(this, "size", 42);  
  
public int getSize() {  
    return size.get();  
}  
  
public void setSize(int newValue) {  
    size.set(newValue);  
}  
  
public IntegerProperty sizeProperty() {  
    return size;  
}
```

Binding API

- JavaFX offre la possibilité de créer des liaisons entre les propriétés.
- Les liaisons synchronisent automatiquement les valeurs de deux propriétés, évitant ainsi le code passe-partout sujet aux erreurs, qui est par ailleurs nécessaire.
- Les propriétés peuvent être liées de manière unidirectionnelle ou bidirectionnelle, implémentées respectivement par les méthodes bind () et bindBidirectional ().
- Si une propriété A est liée de manière unidirectionnelle à une propriété B, la propriété A aura toujours la même valeur que B.

Binding API

- La propriété A devient en lecture seule dans ce cas.
- Si les propriétés A et B sont liées dans les deux sens, les modifications sont propagées dans les deux sens.
- En plus de lier directement deux propriétés, il est également possible de lier une propriété à une expression.
- Si l'un des opérandes de l'expression change, l'expression est automatiquement réévaluée et le résultat est affecté à la propriété.

High-level binding API

- L'API de haut niveau est le moyen le plus rapide et le plus simple de commencer à utiliser des liaisons dans vos propres applications.
- Elle se compose de deux parties: l'API Fluent et la classe Bindings.
- L'API Fluent expose les méthodes sur les différents objets de dépendance, tandis que la classe Bindings fournit des méthodes de fabrique statique.

High-level binding API

OPERATIONS	EXAMPLES	TYPE
Arithmetic Operations	<code>num1.add(num2)</code> <code>Bindings.divide(num1, num2)</code> <code>num1.negate()</code>	Number
Boolean Operations	<code>Bindings.or(bool1, bool2)</code> <code>bool1.not()</code>	Boolean
Comparisons	<code>obj1.isEqualTo(obj2)</code> <code>Bindings.notEqual(obj1, obj2)</code> <code>num1.greaterThan(num2)</code> <code>num1.lessThanOrEqualTo(num2)</code>	All Number, String
Conversions	<code>Bindings.equalIgnoreCase(s1, s2)</code> <code>s1.isNotEqualIgnoreCase(s2)</code> <code>obj.asString()</code>	String All
	<code>num.asString(format)</code> <code>Bindings.format(format, val...)</code>	Number, Object
Null Check	<code>obj.isNull()</code> <code>Bindings.notNull(obj)</code>	Object, String
String Operations	<code>Bindings.concat(s1, s2)</code>	String
Min / Max	<code>Bindings.min(num1, num2)</code> <code>Bindings.max(num1, num2)</code>	Number
Collections	<code>Bindings.valueAt(list, index)</code> <code>Bindings.size(collection)</code>	Collection
Select Binding	<code>Bindings.select(root, properties...)</code>	
Ternary Expression	<code>Bindings.when(cond).then(val1).otherwise(val2)</code>	

High-level binding API

```
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;

public class HiightLevelBindingExample{

    public static void main(String[] args) {
        IntegerProperty num1 = new SimpleIntegerProperty(1);
        IntegerProperty num2 = new SimpleIntegerProperty(2);
        IntegerProperty num3 = new SimpleIntegerProperty(3);
        IntegerProperty num4 = new SimpleIntegerProperty(4);
        NumberBinding total =      Bindings.add(num1.multiply(num2),num3.multiply(num4));
        System.out.println(total.getValue());
        num1.setValue(2);
        System.err.println(total.getValue());
    ...
}
```

Low-level binding API

- Avec l'API de bas niveau, il est possible de définir des liaisons pour des expressions arbitraires.
- L'exemple de code suivant montre comment définir une liaison qui calcule la longueur d'un vecteur (x, y), où x et y sont deux propriétés DoubleProperties.

```
(() -> Math.sqrt(x.get() * x.get() + y.get() *  
y.get()));
```
- L'API de bas niveau est destinée aux développeurs qui ont besoin de plus de flexibilité ou de meilleures performances que celle offerte par l'API de haut niveau.

Low-level binding API

- L'utilisation de l'API de bas niveau implique l'extension d'une des classes de liaison et le remplacement de sa méthode `computeValue ()` pour renvoyer la valeur actuelle de la liaison.
- L'exemple suivant utilise une sous-classe personnalisée de `DoubleBinding`.
- L'appel de `super.bind ()` transmet les dépendances jusqu'à `DoubleBinding` afin que le comportement d'invalidation par défaut soit conservé.
- Il n'est généralement pas nécessaire de vérifier si la liaison est invalide. Ce comportement est fourni par la classe de base.

Low-level binding API

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.DoubleBinding;

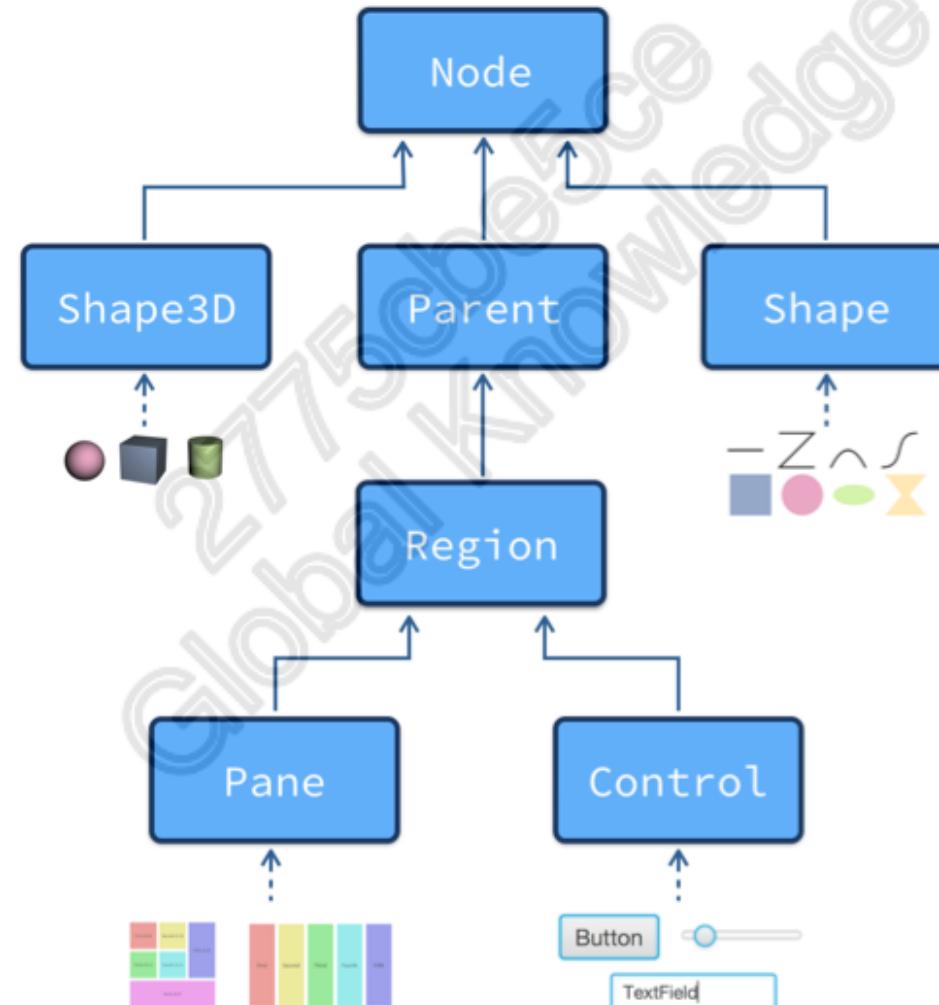
public class Main {

    public static void main(String[] args) {
        final DoubleProperty a = new SimpleDoubleProperty(1);
        final DoubleProperty b = new SimpleDoubleProperty(2);
        final DoubleProperty c = new SimpleDoubleProperty(3);
        final DoubleProperty d = new SimpleDoubleProperty(4);
```

Low-level binding API

```
DoubleBinding db = new DoubleBinding() {  
    {super.bind(a, b, c, d); }  
  
    @Override  
    protected double computeValue() {  
        return (a.get() * b.get()) + (c.get() * d.get());  
    }  
};  
  
System.out.println(db.get());  
b.set(3);  
System.out.println(db.get());  
}  
}
```

Composants graphiques

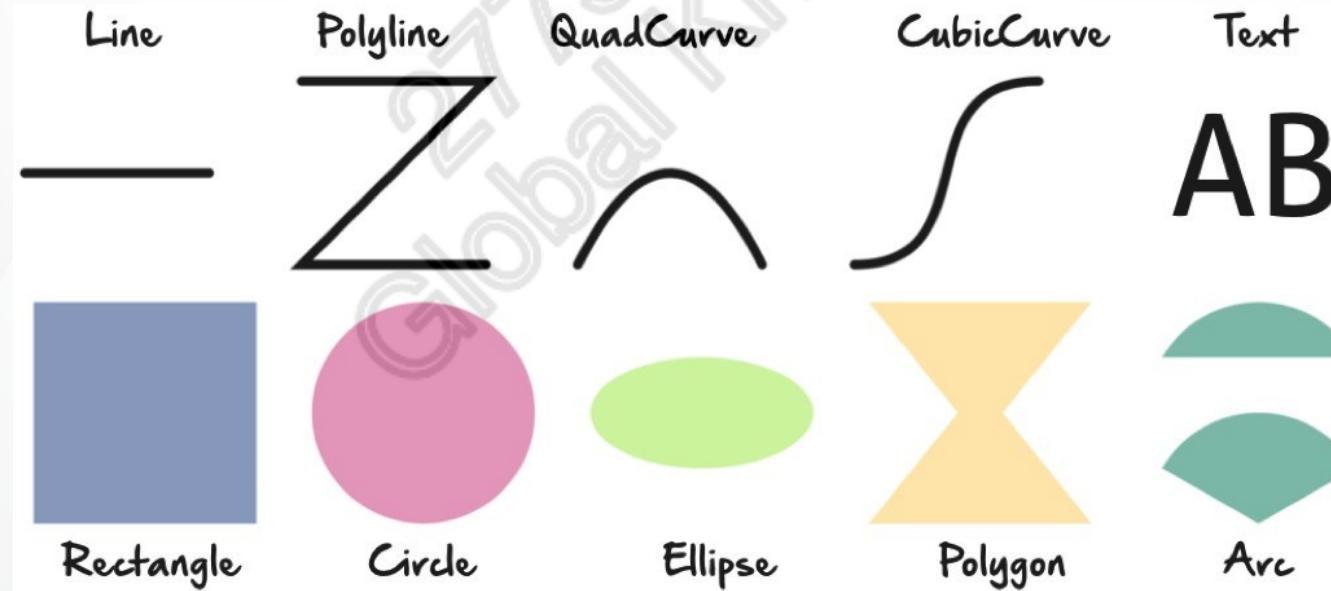


Composants graphiques

- Toutes les formes géométriques et le texte (qui est juste une forme très complexe) étendent la classe Shape ou Shape3D.
- Ce sont des nœuds de feuille dans le graphe de scène. La possibilité de contenir d'autres nœuds en tant qu'enfants est définie dans la classe abstraite Parent.
- Les contrôles d'interface utilisateur et les panneaux de présentation étendent cette classe.
- Outre les propriétés qui définissent le comportement spécifique d'un nœud, elles prennent toutes en charge la gestion des événements et le style CSS.

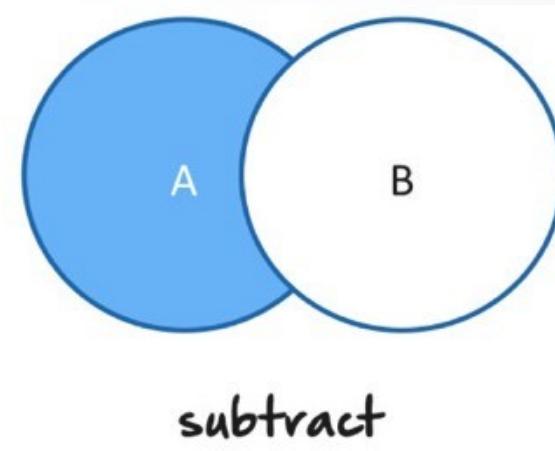
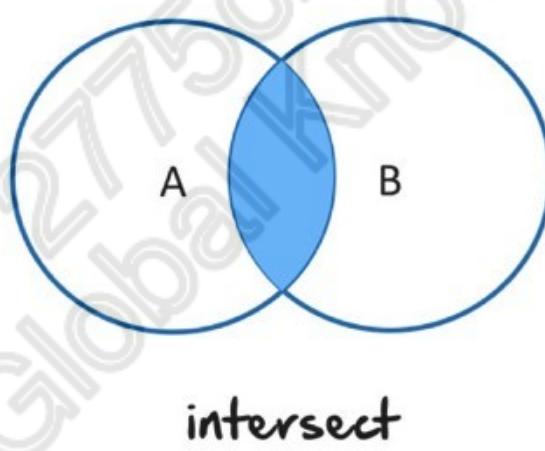
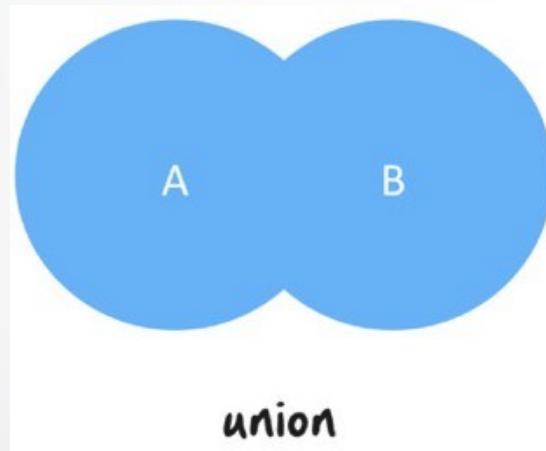
Les formes

- Les formes définissent les nœuds les plus élémentaires pouvant être affichés dans un graphique de scène JavaFX.
- La classe Shape est la super-classe de toutes les primitives géométriques et définit les caractéristiques de base suivantes:



Les formes

- De plus, cette classe fournit les opérations booléennes union, intersection et soustraction pour créer de nouvelles formes.



Les formes

- Au-delà de la définition du trait et du remplissage d'une forme sur une couleur, vous pouvez utiliser les quatre implémentations de la classe Paint:



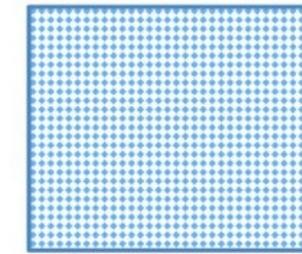
Color



LinearGradient



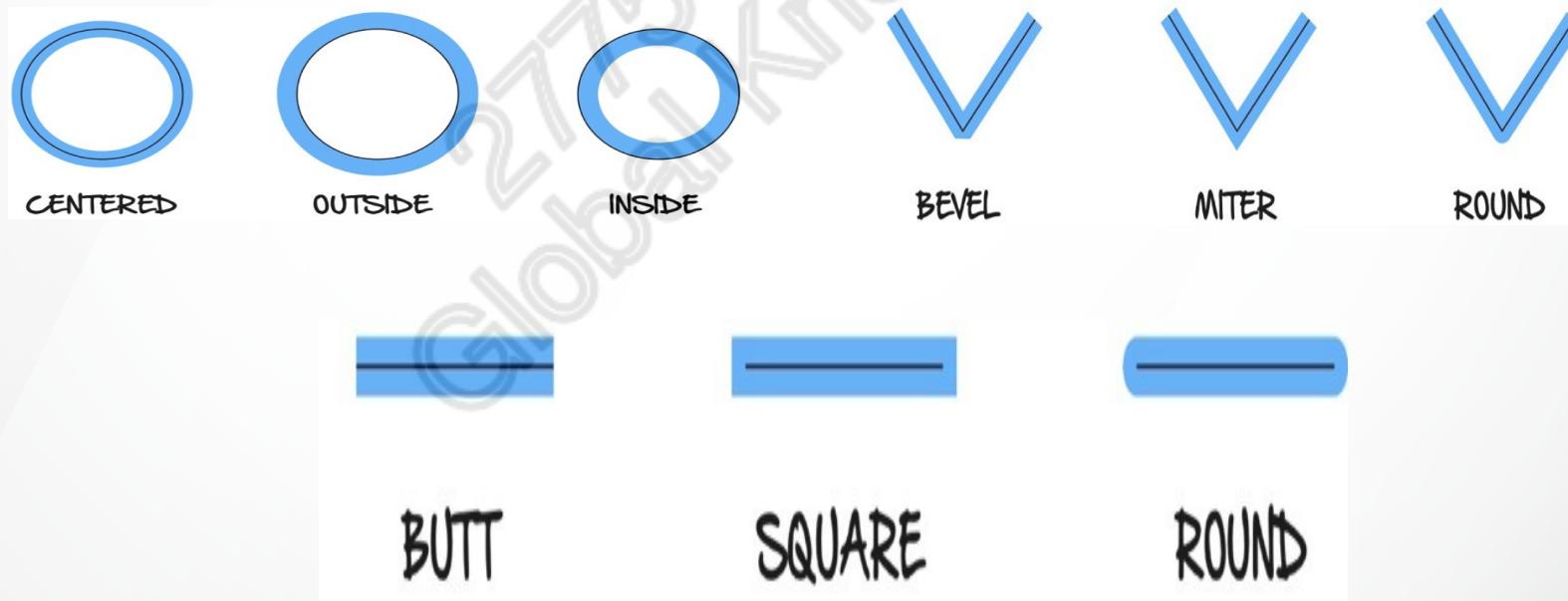
RadialGradient



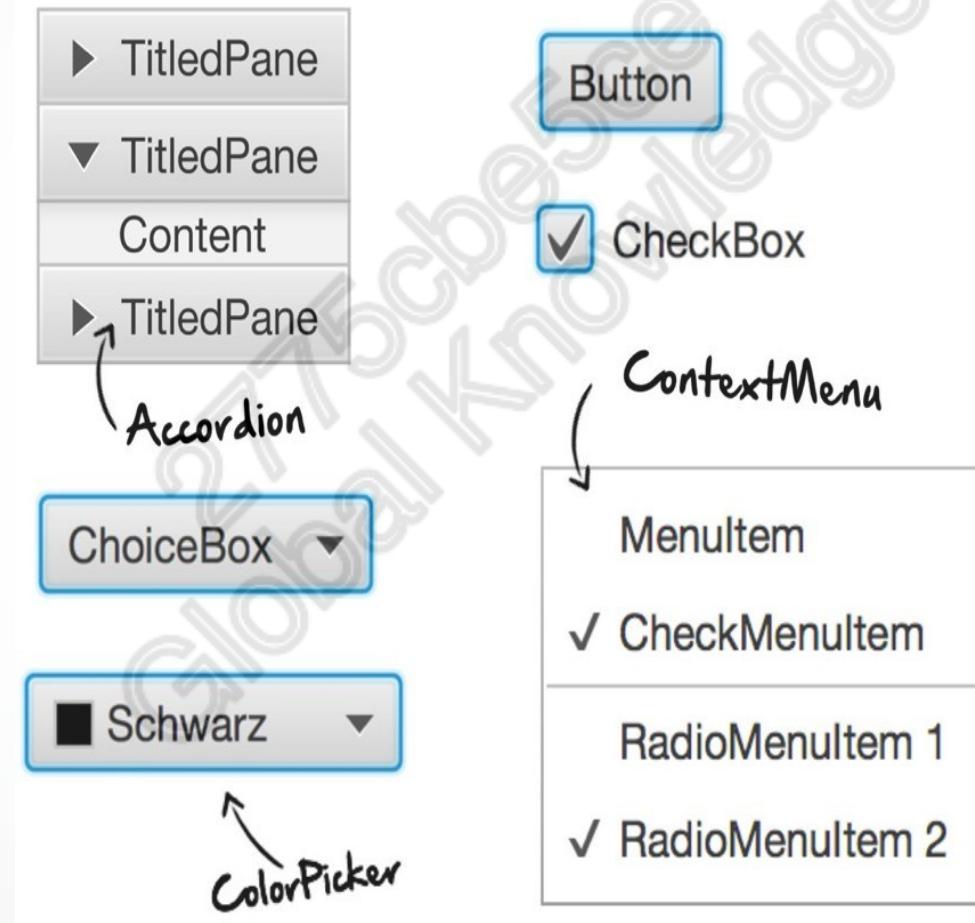
ImagePattern

Les formes

- Le trait peut également être configuré en modifiant les propriétés `strokeType`, `strokeLineJoin` et `strokeLineCap`.
- Il est également possible de définir un motif fringant.



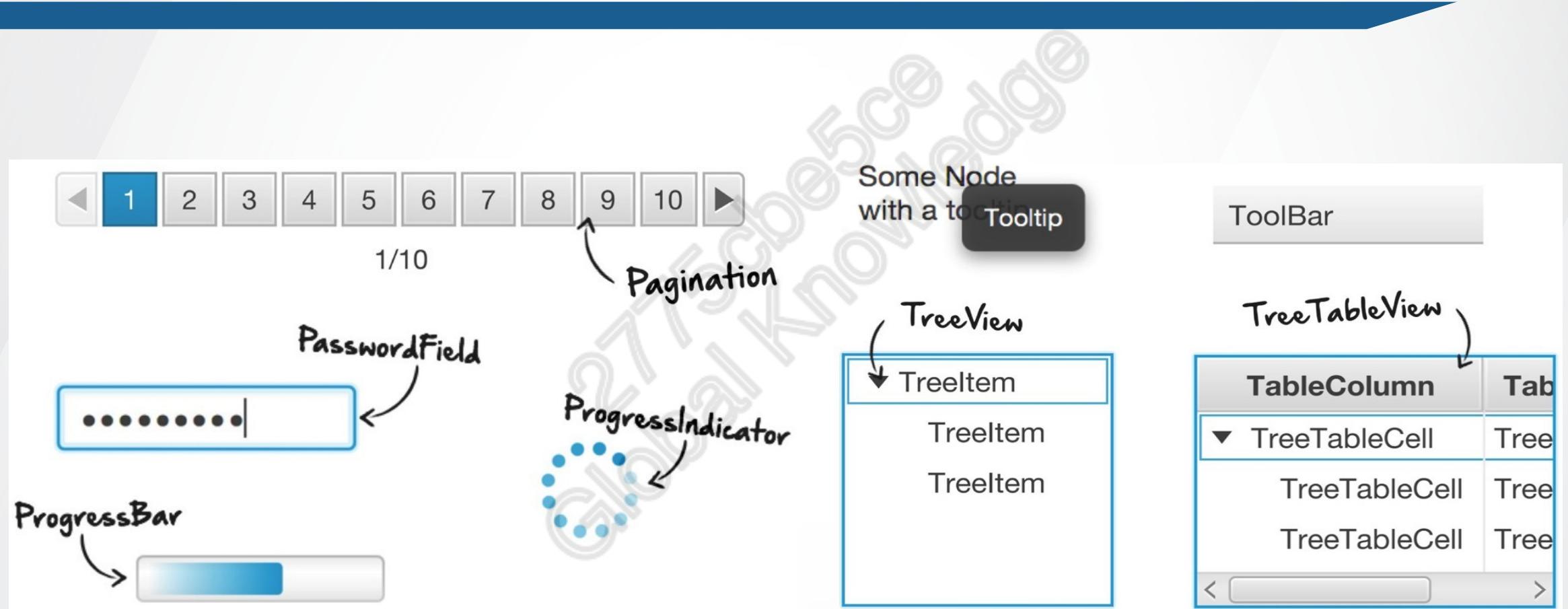
Les contrôles



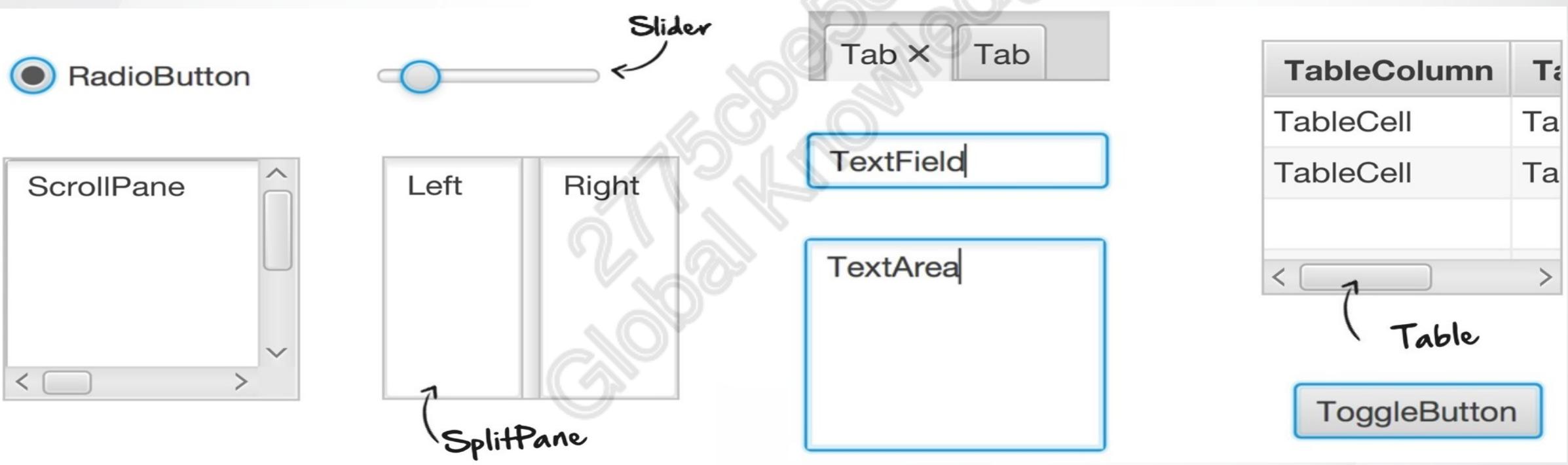
Les contrôles

- Les contrôles sont les nœuds d'interface utilisateur JavaFX qu'un développeur utilisera la plupart du temps.
- Tous les contrôles, tels que les boutons, les champs de texte ou les tableaux, étendent la classe Control.
- Par défaut, les contrôles JavaFX seront rendus dans un thème cohérent et indépendant du système.

Les contrôles



Les contrôles

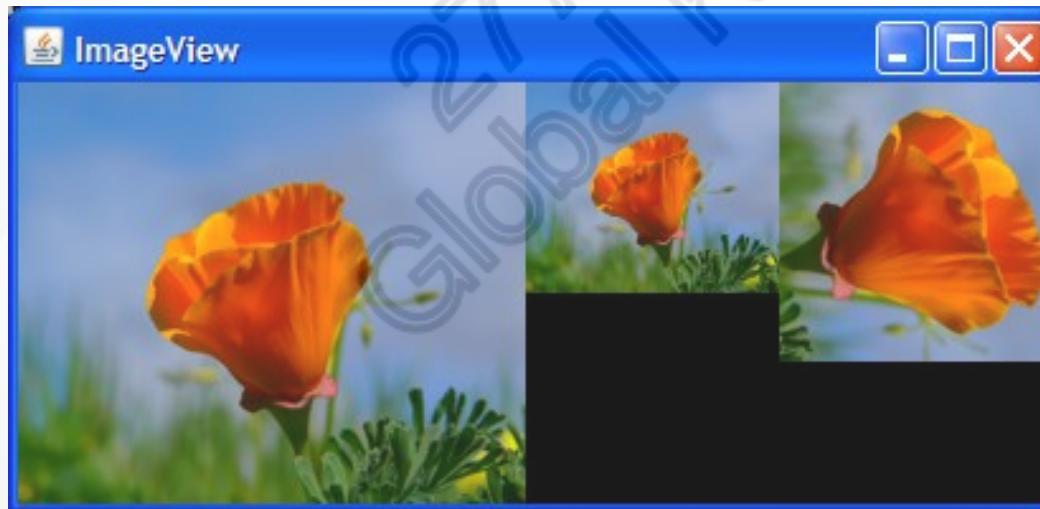


ImageView

- Deux classes sont nécessaires pour afficher une image.
- La classe `Image` encapsule les données brutes d'une image et ses propriétés.
- La classe `ImageView` est un nœud de graphe de scène et elle est chargée d'afficher l'image à l'écran.
- Cette division est nécessaire car elle permet à JavaFX d'afficher une image plusieurs fois à l'écran, tandis que les données ne sont conservées en mémoire qu'une seule fois.

ImageView

```
Image image = new Image("path/to/image");
ImageView imageView = new ImageView(image);
myPane.getChildren().add(imageView);
```



MediaView

- La classe MediaView peut être utilisée pour afficher une vidéo à l'écran ou lire un fichier audio.
- JavaFX prend en charge les codecs audio MP3, AIFF, WAV et MPEG-4 et les codecs vidéo FLV (Flash Video) ou MPEG-4 (H.264 / AVC).
- Deux classes sont nécessaires pour lire un flux audio:
 - la classe Media encapsule les données brutes,
 - tandis que MediaPlayer fournit une fonctionnalité permettant de contrôler la lecture et contient d'autres informations utiles sur le média.
- Pour lire une vidéo, une troisième classe est nécessaire: MediaView. C'est un nœud de graphe de scène classique, ce qui signifie que vous pouvez même appliquer des effets et animer des vidéos.

MediaView

```
Media media = new Media("path/to/media");
MediaPlayer player = new MediaPlayer(media);
MediaView mediaView = new MediaView(player);
player.setVolume(0.5);
player.play();
```



Charts

- JavaFX contient une API de graphiques à part entière qui lui permet de définir et de visualiser des graphiques.
- La figure suivante montre le type de graphique disponible dans le JDK standard:



Canvas

- Dans certaines situations, il est plus efficace de contrôler totalement le rendu et d'afficher directement l'écran.
- JavaFX fournit la classe Canvas pour ces scénarios.
- Il fournit à GraphicsContext plusieurs méthodes pour dessiner directement des figures géométriques ou des images.
- Le composant Canvas est comparable à HTML Canvas ou à la fonctionnalité Java2D Graphics2D.

WebView

- La classe JavaFX WebView peut être utilisée pour incorporer n'importe quel contenu Web dans votre application.
- WebView utilise WebKit en interne pour restituer le contenu Web et fournir une interaction avec ce contenu.
- Même les applications HTML5 riches peuvent être encapsulées dans une fenêtre ou un volet JavaFX.
- WebView fournit un objet WebEngine qui permet au développeur d'interagir directement avec le contenu HTML et, par exemple, d'injecter du JavaScript ou de manipuler le DOM.

WebView

```
WebView webView = new WebView();
WebEngine engine =
webView.getEngine();

//Load a web page
engine.load("http://www.google.com");

//Add the web view to the JavaFX view
myPane.getChildren().add(webView);

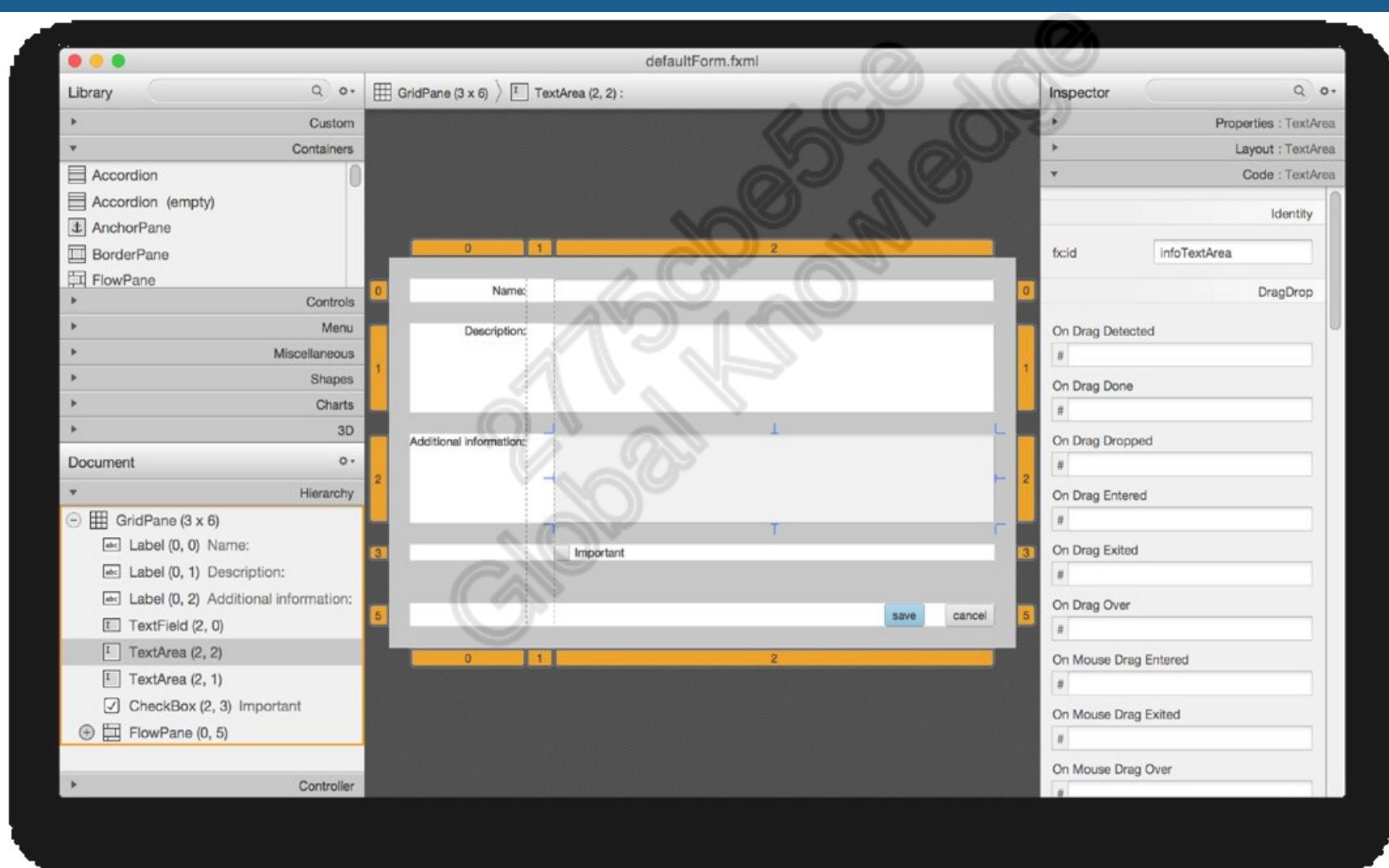
// Inject JavaScript
engine.executeScript("history.back()");
;
```



Le langage FXML

- FXML est un langage basé sur XML qui définit la structure et la disposition des interfaces utilisateur JavaFX.
- Il ne dépend pas des outils et peut être modifié avec n'importe quel éditeur, mais il est particulièrement utile d'utiliser SceneBuilder, un éditeur WYSIWYG pour FXML.

Le langage FXML



Le langage FXML

- FXML vous permet de créer une séparation claire entre la vue d'une application et la logique.
- JavaFX fournit une API permettant de définir des packages MVC basés sur FXML et un contrôleur Java.
- Voici une petite définition FXML pour une vue qui ne contient qu'un seul bouton dans un StackPane:

Le langage FXML

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<StackPane maxHeight="-Infinity" maxWidth="- Infinity" minHeight="-Infinity"
minWidth="-Infinity"
prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <Button mnemonicParsing="false" text="ButtonTitle"
/>
    </children>
</StackPane>
```

Le langage FXML

- Une fois la vue FXML créée, elle peut être chargée avec le Classe FXMLLoader:

```
FXMLLoader loader = new FXMLLoader(getClass(). getResource("demo.fxml"));
StackPane view = loader.load();
```

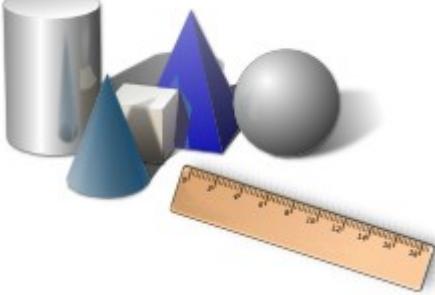
- Un contrôleur basé sur Java peut être lié au fichier FXML.
- L'annotation @FXML peut être utilisée pour injecter des nœuds de vue directement dans la classe du contrôleur.
- Un noeud injectable doit être marqué avec un fx: id unique dans FXML:

Le langage FXML

```
public class ViewController {  
    @FXML  
    private Button myButton;  
}
```

- Lors du chargement du flux FXML, le contrôleur peut être transmis au chargeur. Dans ce cas, FXMLLoader injectera automatiquement tous les champs annotés avec **@FXML**:

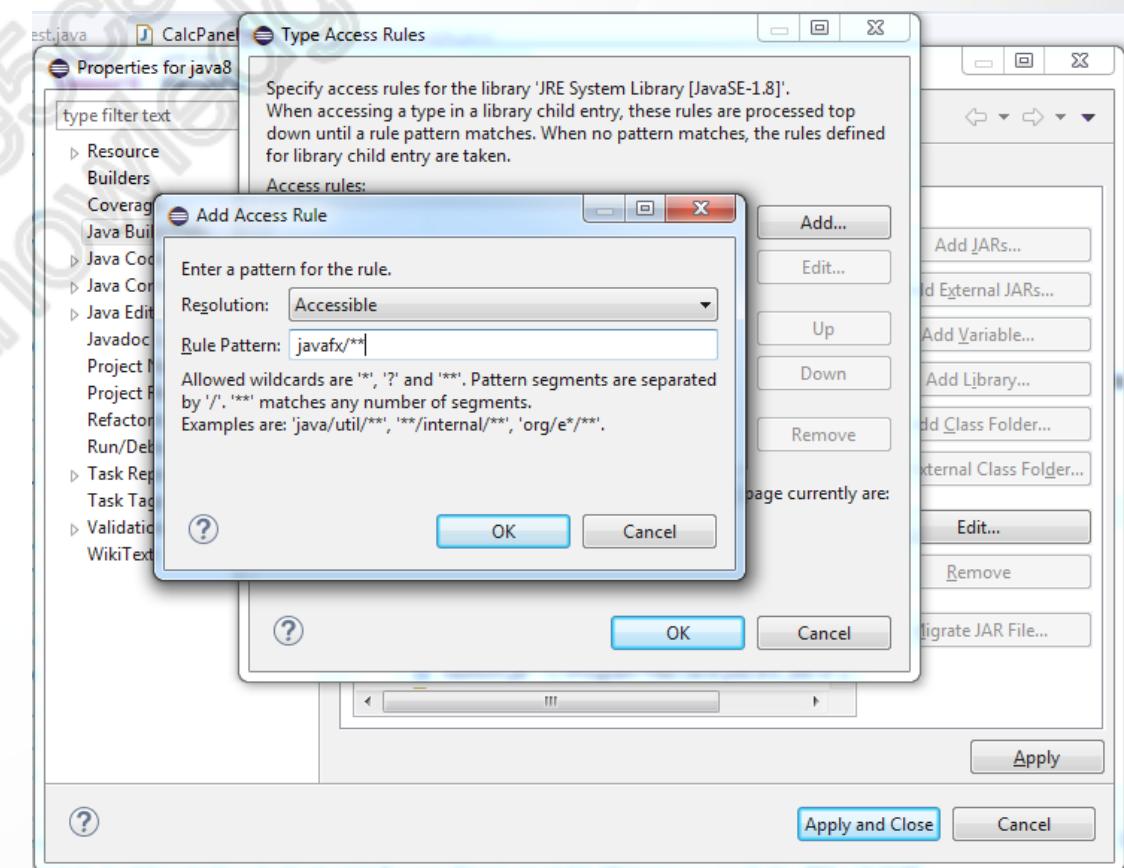
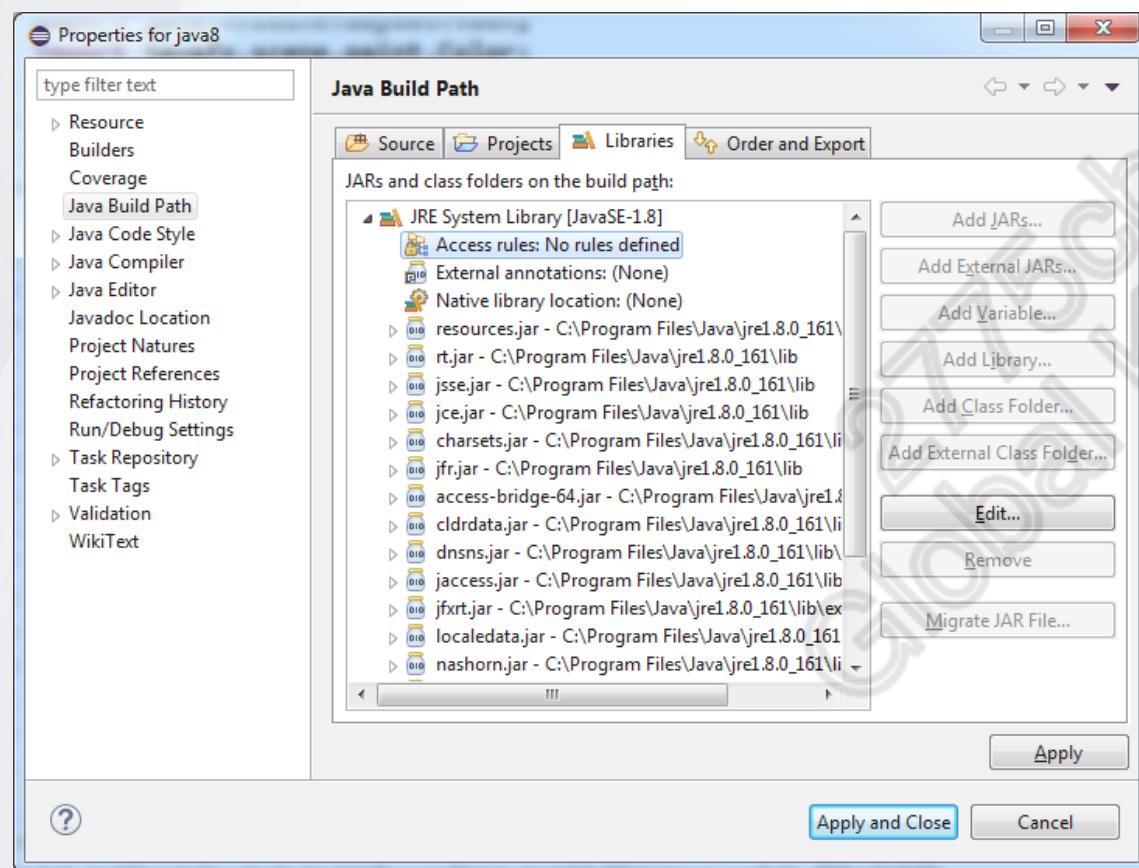
```
ViewController controller = new ViewController();  
FXMLLoader loader = new FXMLLoader(getClass().getResource("demo.fxml"));  
loader.setController(controller);  
StackPane view = loader.load();
```



Exercice 12 - Interface Graphique JavaFX

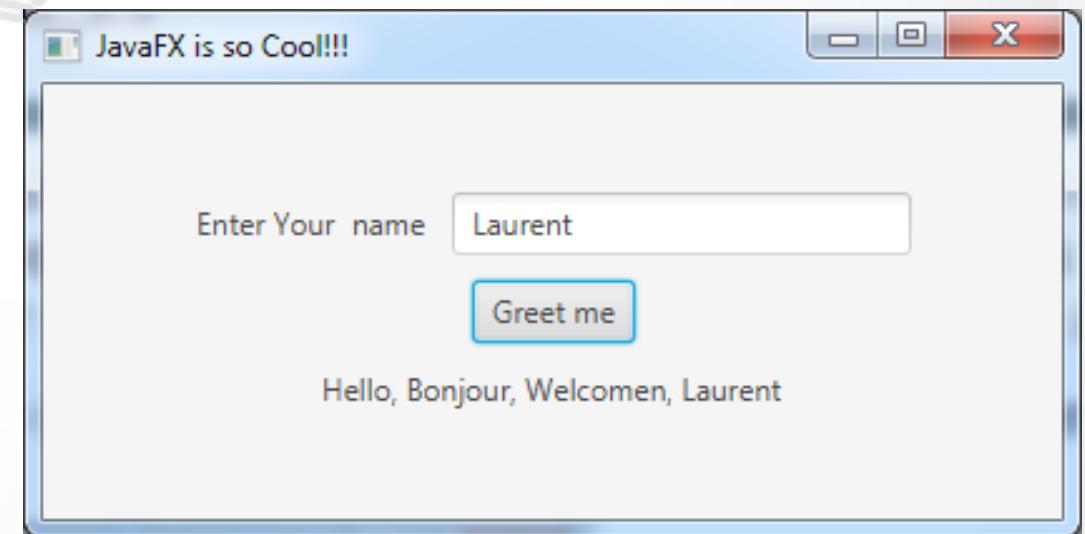
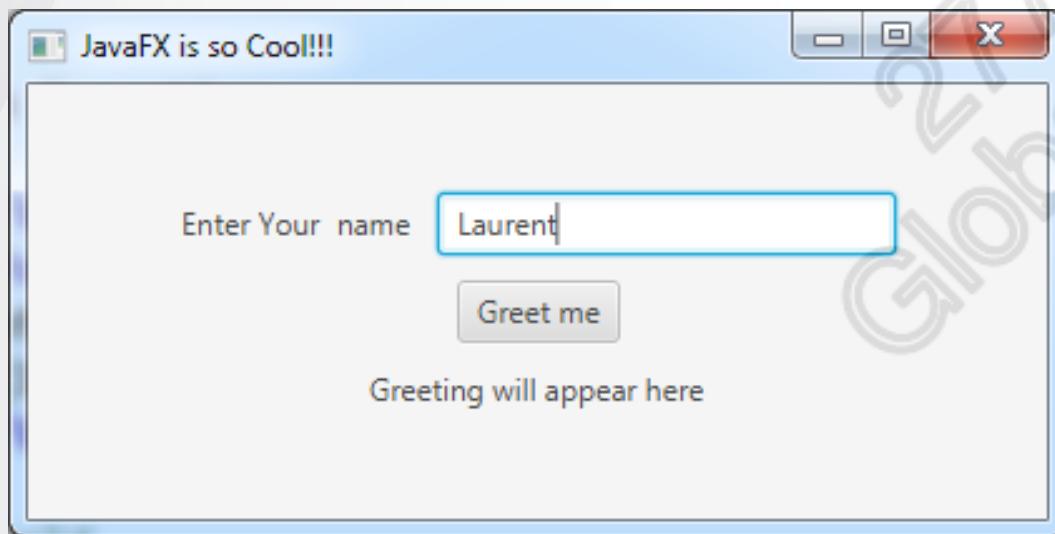
- Pour JavaFX activer sur votre projet Eclipse :
 - Allez dans Propriétés puis Chemin de construction Java.
 - Double-cliquez sur votre bibliothèque système JRE et sélectionnez Règles d'accès.
 - Cliquez sur Modifier dans le menu de droite.
 - Créer une nouvelle règle avec résolution accessible et le modèle de règle suivant: javafx / **. Ce paramètre activera les packages et les classes JavaFX pour votre application et préservera la règle du dossier ext pour toutes les autres classes.

Exercice JavaFX



Exercice JavaFX

- Dans cet atelier, vous allez utiliser JavaFX pour créer une application qui peut afficher un message de salutation destinée à une personne qui va renseigner son nom dans un champ texte en entrée.
- L'interface utilisateur de l'application devrait ressembler à ceci:



Exercice JavaFX

```
@Override  
public void start(Stage stage) throws Exception  
{  
    Button greetButton = new Button("Greet me");  
    Label label = new Label("Enter Your name");  
    TextField nameTextField = new TextField();  
    nameTextField.setPrefColumnCount(15);  
    Label greetingLabel = new Label("Greeting will appear here");  
  
    //Top HBox holds label and Textfield  
    HBox topHBox = new HBox(10, label, nameTextField);  
    topHBox.setAlignment(Pos.CENTER);
```

Exercice JavaFX

```
stage.setTitle("JavaFX is so Cool!!!");
VBox outerVBox = new VBox(10, topHBox, greetButton, greetingLabel);
outerVBox.setAlignment(Pos.CENTER);
// padding on vB0x
outerVBox.setPadding(new Insets(10, 10, 10, 10));
Scene scene = new Scene(outerVBox);

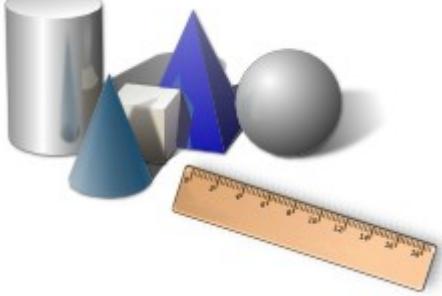
stage.setScene(scene);
stage.show();
```

Exercice JavaFX

```
// Event handler
EventHandler<ActionEvent> handler = evt ->
{
    String name = nameTextField.getText();
    greetingLabel.setText("Hello, Bonjour, Welcomen, " + name);
};
greetButton.setOnAction(handler);
}
```

Pour conclure





Exercice 13 – Crédit d'archive

- Dans le cahier d'exercices, on vous explique comment générer un .jar de votre projet. Si vous souhaitez vous l'envoyer par mail, sachez que gmail bloquera l'envoi pour des questions de sécurité. Dans ce cas, l'astuce consiste à changer l'extension en txt (par exemple) et lors de la récupération, basculer de nouveau en .jar
- Les corrections détaillées sont disponibles en format pdf, sur edoc, si ce n'est pas le cas prévenez le formateur.

Evaluation de la formation



Formulaire et billet d'humeur

- Renseigner le formulaire Global Knowledge
- Chacun choisit 3 mots qui décrivent le mieux ses sentiments sur la formation
- A l'aise, absorbé, abattu, ahuri, agacé, allégé, agité, amusé, animé, attentif, apathique, de bonne humeur, bloqué, calme, captivé, centré, charmé, concentré, concerné, confiant, confortable, content de soi, curieux, détaché, déconcerté, détendu, déçu, emballé, embrouillé, enchanté, encouragé, ennuyé, étonné, éveillé, éreinté, étourdi, fier, fatigué, gai, galvanisé, hilare, impatient, impliqué, informé, inquiet, insouciant, indifférent, intéressé, joyeux, libre, nourri, optimiste, paisible, rassuré, ravi, satisfait, sceptique, sensibilisé, soulagé, stimulé, vindicatif, valorisé, zen



Global Knowledge®

Votre partenaire formation

MERCI