JAVA OBJET

OBJET

- Le langage JAVA est une programmation orienté **objet** (POO)
- La POO est une programmation qui se base sur l'utilisation exclusive des objets (classes).

- **OBJET:** c'est entité indépendante ayant un nom, des attributs et des méthodes

Ex: Personne, Livre, Voiture

- **JAVA POO** : Ensemble d'instruction écrites et adressées à une machine dans le but d'obtenir un résultat après exécution.

Déclaration d'un classe

SYNTAXE :

```
public class NomDeLaClasse
{
	// Variables d'instance
	porteV	type	attribut 1 ;
	porteV	type	attribut 2 ;

	// Constructeurs
	porteV	NomDeLaClasse()
	{
		//instructions
	}

	// Méthodes
	porteV	type	methode()
	{
		//instructions
	}
}
```

Encapsulation (portée de la variable)

L'encapsulation permet de masquer la visibilité d'un attribut ou d'une méthode à des degrés divers.

On distingue 3 niveaux de portée de variables en Java :

- ***public*** : accès publique. Dans le cas où l'attribut et/ou méthode sont précédés de + ou public, les attributs et méthodes sont accessibles depuis partout, par toutes les classes

- ***protected*** : accès protégé. Les attributs et les méthodes sont accessibles uniquement par la classe Fille et par la classe elle-même (classe mère)

  →Elle est aussi visible pour des classes du même package.

- ***private*** : accès privé. Les attributs et les méthodes sont accessibles qu'à l'intérieur d'une classe. Héritée, elle n'est pas visible.

  →On peut y accéder grâce au Getter et au Setter

Types

- boolean
- String
- int
- float / double

Attributs /Variables

- Les attributs sont les variables de la classe
- Les attribut d'une classe, s'ils ne sont initialisés, se voient affecter automatiquement une valeur par défaut,

Cette valeur vaut ***0*** pour les variables numériques, ***false*** pour les booléens et ***null*** pour les autres types

La portée de variables et attributs:

- Les variable ne sont connues qu'à l'intérieur du boc dans lequel elle sont déclarées
- En cas de conflits de nom, c'est toujours la variables la plus locale qui est considérée comme étant la variable désignée par cette partie du programme

Attributs /Variables

Exemple:

```
public class Personne
{
        public   String   nom;
        public   String   prenom;
        public   int        age;
}
```

Personne
nom prenom age

Les constructeurs

Pour initialiser un objet on utilise une méthode "spéciale" appelée constructeur.

Le constructeur est une méthode:

- **De même nom que la classe**
- **ne retourne rien** (Ne pas mettre *void* comme valeur de retour).
  - → **Un constructeur est une méthodes d'initialisation**

Toute classe possède au moins un constructeur. Si le programmeur ne l'écrit pas, il en existe un défaut, sans paramètre, le code vide.

Si le programmeur ne fournit pas de telle méthode, Java en fournit une sans argument et qui a un corps vide **automatiquement**.

→Ce constructeur alloue les attributs de l'objet et initialise par défaut les attributs avec les valeurs par défaut de Java.

Les constructeurs

Le constructeur est donc une méthode comme une autre,

- elle exploite tous les attributs de l'objet,
- elle peut déclarer des variables locales,
- elle peut appeler des méthodes privées de la classe et des méthodes d'autres classes

- On peut définir plusieurs constructeurs différents, du moment qu'ils se **différencient** par les paramètres de la méthode

  → qu'on appelle une **surcharge**

- Cette méthode est **public**

Les constructeurs

```
public class Personne
{
	public	String	nom;
	public	String	prenom;
	public	int	age;


	//Constructeur par défaut
	public Personne()
	{
	    nom = "Inconnu";
	    prenom = "Inconnu";
	    age = 0;
	}

}
```

Les constructeurs

```
public class Personne
{
      public   String   nom;
      public   String   prenom;
      public   int       age;


      // 2ème  constructeur
      //Constructeur avec paramètres
      //« p » en première lettre des paramètres pour différencier
      public Personne(String pNom, String pPrenom, int pAge)
      {
          nom  = pNom;
          prenom = pPrenom;
          age = pAge
      }
}
```

Méthodes

- Une méthodes est un ensemble d'instructions (codes) réutilisable qui effectue une action bien définie

- Les méthodes se définissent dans une classe et ne peuvent pas être imbriquées. Elles sont déclarées les unes après les autres.

- Une méthode peut être surchargée en modifiant le type de ses paramètres, leur nombre, ou les deux.
- Pour Java, le fait de surcharger une méthode lui indique qu'il s'agit de deux, trois ou X méthodes différentes, car les paramètres d'appel sont différents.

Par conséquent, Java ne se trompe jamais d'appel de méthode, puisqu'il se base sur les paramètres passés à cette dernière.

Méthodes

```
public class Personne
{
	// Variables d'instance

	// Constructeurs

	void afficherNom ( String nom)
	{
	   System.out.println(« Bonjour » + nom);
	}
}
```

Instancier: créer un objet

- La création d'un objet se fait en instanciant une classe.
- L'instruction **new permet de créer un objet** à partir d'une classe.

L'appel de new pour créer un nouvel objet déclenche, dans l'ordre:

- l'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ses attributs
- l'initialisation explicite des attributs , s'il y a lieu
- l'exécution d'un constructeur défini ou celui par défaut

L'objet qui résulte du processus d'instanciation contient toute **l'arborescence** de composition des attributs définition de la classe.

- Une fois instancié, l'accès aux attributs se fait par l'opérateur : **. (point)**.

Création d'un objet (instanciation)

```
public static void main (String argvs [])
{
	//création d'un objet de type Personne
	Personne pers = new Personne();
	Personne pers1 = new Personne ("DURAND", "Jean" , 20);

	// attribution d'une valeur
	pers.nom = "DUPOND";
	pers.prenom = "Jerome";
	pers.age = 18;

	// appel de la fonction
	pers.afficherNom(pers.nom);
	pers1.afficherNom(pers1.nom);
}
```

Résumons

- Créer une classe(Objet)
- Déclarer les variable d'instance
- Créer les méthodes et les différentes méthodes surchargé (avec un nbre différent ou des paramètres différents)
- Revenons sur le programme principal, créer une instance de l'objet créer (créer une image)
- Pour déclarer et initialiser un objet, il faudra respecter l'ordre des paramètres passé lors de la création des méthodes : sinon erreur de compilation
- Jusqu'à maintenant les variable d'instance sont accessible à partir du ***main (implique qu'on peut directement modifier les attributs de la classe)***.
- Pour protéger nos variables d'instance, nous allons les déclarer en ***private***.
- Désormais, ces attributs ne sont plus accessibles en dehors de la classe où ils sont déclarés

Surcharges

```
public Personne (String pNom, String pPrenom, int pAge)
{
      super (pNom, Pprenom, pAge);
}

public Personne (String pNom, String pPrenom)
{
      this(pNom, Pprenom, 0);
}

public Personne (String pNom)
{
       this(pNom, "Jerome" , 0);
}

public Personne ()
{
       this(" DURAND" , "Jerome" , 0);
}
```

- La surcharge d'une méthode ou d'un constructeur permet de définir plusieurs fois une même méthode/constructeur avec des arguments différents.
- **this**: permet d'appeler une variable d'instance courante d'une classe
- **super**: permet d'appeler un constructeur parents

Héritage

- L'héritage en Java permet de pratiquer de la réutilisabilité entre les objets.

- C'est une relation entre une classe mère et une classe fille (Généralisation/Spécialisation).

On parle de:

  - **Généralisation** quand on regroupe les éléments communs de plusieurs classes en une superclasse
  - **Spécialisation** quand on crée des sous-classes pour différencier les éléments particuliers d'une classe

- La **classe fille** qui **hérite** de tous les **attributs** et toutes les **méthodes** d'une seule et unique classe normal appelée **classe mère** en utilisant le mot clé **extends**.

Héritage

Remarques :

- La mère n'hérite jamais de la fille.
- On ne réécrit pas les méthodes de la classe mère dans la classe fille sinon on parlera alors de deux autres notions (**Surcharge** et **polymorphisme**)
- On peut rajouter dans la classe fille des nouvelles méthodes et des nouveaux attributs.
- Il faudra toujours penser à instancier la classe fille avant de faire appel à la richesse de la classe mère (= les méthodes et les attributs de la classe mère)

Et si on a envie d'hériter de plusieurs classes ?

- On va utiliser une classe **interface**.

Héritage

SYNTAXE:

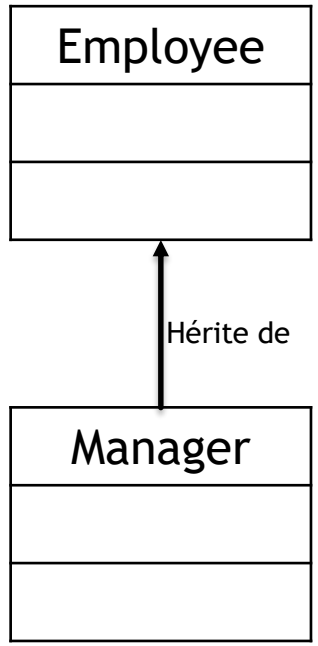
```
class ClasseFille extends ClasseMere
{

}
```

EXEMPLE:

```
Class Manager extends  Employee
{

}
```



Spécialisation

On considère 2 classes liées par l'héritage.

Par exemple :

```
class Employee { . . . }

class Manager extends  Employee { . . . }
```

Souvent une méthode est définie dans la classe de base, mais a besoin d'être "affinée" dans la classe héritière → On parle de la **spécialisation**

Par exemple :

```
class Employee
{
   void calculPrime()
      {
            // une prime pour un
            employe
      }
}
```

```
class Manager extends Employee
{
   void calculPrime()
      {
            // une prime plus
            importante pour un manager
       }
}
```

Polymorphisme

- Le nom de la méthode calculPrime() dénote alors 2 codes distincts.

Autrement dit l'identificateur calculPrime représente "plusieurs formes" de code distinct.

→ **On parle de polymorphisme**

On utilise le polymorphisme à l'aide d'une **référence d'objet** de classe de base.

Par exemple :

```
Employee e = new Employee();

e.calculPrime();
 // lance la méthode calculPrime() de la classe Employee
```

mais aussi

```
Employee e1 = new Manager();

e.calculPrime();
// lance la méthode calculPrime() de la classe Manager
```

Polymorphisme

- Un des avantages de ceci, c'est de pouvoir utiliser une seule référence de la classe *Employee* pour référencer un *Employee* ou un *Manager*.

C'est possible car un Manager (classe héritière de la classe Employee) est un Employee : **sémantique de l'héritage.**

En conclusion, il est fondamental de voir que pour utiliser le polymorphisme, il faut toutes les conditions suivantes :

- avoir une arborescence d'héritage sur les classes
- l'utiliser à l'aide de référence d'objet de la classe de base.
- avoir deux méthodes de même signature définies l'une dans la classe de base, l'autre dans la classe dérivée.

Remarque : on dit alors qu'on a @Override (supprimé) la méthode de la mère.

Si on veut quand même appelé la méthode de la mère, dans ce cas, au niveau de la fille on écrit :

```
super.methodeMere() ;
```

Surcharge VS Polymorphisme

	Surcharge	Polymorphisme
Héritage	nul besoin	nécessite une arborescence de classes
signature des méthodes	doivent différer	doivent être les mêmes
résolu à	la compilation	l'exécution

Classe Interface

- l'héritage multiple est interdit en Java, donc **si on a envie d'hériter de plusieurs classes**

→ **On utilise la Classe Interface**

- La classe interface permet de créer un nouveau **superclasse** : on peut même en ajouter autant que l'on le veut dans une seule classe

Comment faire?

- Étape 1: Déclarer la classe interface et y **déclarer** juste les méthodes (non définis)
- Étape 2: Déclarer une autre classe (classe fille) qui va hériter de la classe Interface
  - C'est dans cette classe fille qu'on va **définir** toutes les méthodes de la classe interface avant d'en utiliser une

Classe Interface

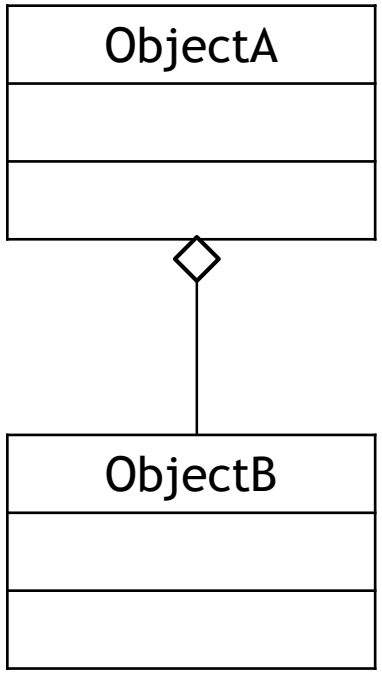
Pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé ***implements***

```
public interface Employe
{
	public void A();
	public String B();
}

public interface Personne
{
	public void C();
	public String D();
}
```

```
public class Manager  implements Employe , Personne
{
	public void A(){    //Définir la méthode A   }
	public String B({    //Définir la méthode B   }
	public void C(){    //Définir la méthode C  }
	public String D(){  //Définir la méthode D   }
}
```

Relation d'agrégation



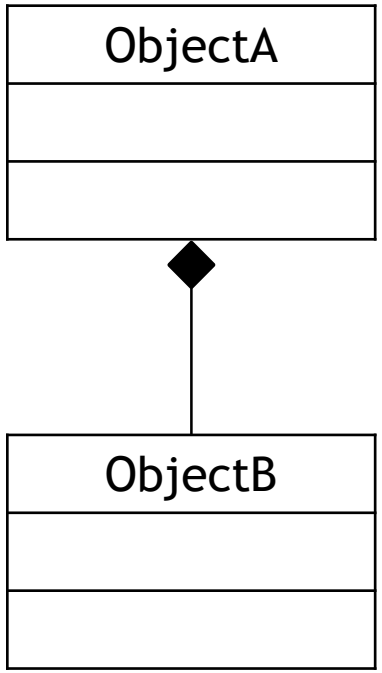

```
public class ObjectA
{     //variable d'instance
      private ObjectB nomB;

      //méthodes
      public void setNomB (ObjectB bNom)
      {
            this.nomB = bNom;
      }

      public ObjectB getNom ()
      {
            return nomB;
      }
}
```

```
public class ObjectB
{
}
```

Relation de composition




```
public class ObjectA
{     //variable d'instance
      private ObjectB b;
      private String nom;

      //Constructeur
      public ObjectA (String aNom)
      {
            this.nom = aNom;
            this.b = new ObjectB ();
      }
}
```

```
public class ObjectB
{


}
```

Classe Abstraites

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthodes sans la définir (c'est-à-dire sans en donner le code).

La définition du code est dans ce cas laissée aux classes fille.
→ **On parle de classe abstraite**

- Elle doit être marqué par le mot réservé **abstract**
- Toutes ses méthodes qui ne sont pas définies, doivent elle aussi être marquées **abstract**
- Une classe abstraite ne peut pas être instanciée, elle ne peut qu'être héritée (c'est sont but)
- Par contre, il est possible de déclarer et d'utiliser des variable du type de la classe abstraite
- Tant qu'une classe Fille d'une classe abstraite ne définit pas toutes les méthodes abstraites de ses superclasses, alors elle est abstraite elle aussi

Classe Abstraites

```
public abstract class Employee
{
	private String nom;
	private int prime;
	private int salaireBase;


	//méthode non définie
	public abstract void afficherNom(String nom);


	//méthode définie
	public int calculPrime()
	{
		int salaire = prime + salaireBase
		return salaire
	}
}
```

```
public class Manager extends Employee
{

	public void afficherNom(String nom)
	{
		System.out.println("Bonjour"+ nom);
	}

}
```