Java et la Programmation Orientée Objet

Classes et Objets

- Une **classe** est un modèle définissant les caractéristiques et comportements d'un objet.
- Un **objet** est une instance d'une classe.

Exemple *minimal*

```
class Voiture {
    String marque;
    int vitesse;
}

public class Main {
    public static void main(String[] args) {
        Voiture v1 = new Voiture();
        v1.marque = "Toyota";
        v1.vitesse = 120;
    }
}
```

Encapsulation

- Protection des données via les modificateurs d'accès (`private` , `public` , `protected`).
- On accède aux données via des méthodes (`getters` et `setters`).

```
class Personne {
    private String nom;

    public String getNom() {
        return nom;
    }

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

Constructeurs et surcharge

- Un **constructeur** initialise un objet lors de sa création.
- Un constructeur a le nom de la classe ; plusieurs constructeurs possibles.
- Un constructeur peut en appeler un autre avec `this()`.

```
class Animal {
    private String nom;

    public Animal() {
        this("Chat");
    }

    public Animal(String nom) {
        this.nom = nom;
    }
}
```

Héritage

- Mot-clé `extends`, utilisation de `super` pour appeler constructeur et méthodes

```
class Animal {
    protected String nom;

    public Animal(String nom) { this.nom = nom; }

    @Override
    public String toString() {
        return nom;
    }
}
class Chien extends Animal {
    public Chien(String nom) { super(nom); }

    @Override
    public String toString() {
        return "Chien: " + super.toString();
    }
}
```

Abstraction

- Une **classe abstraite** ne peut pas être instanciée et peut contenir des méthodes abstraites.

```
abstract class Animal {
    abstract void faireDuBruit();
}

class Chat extends Animal {
    @Override
    public void faireDuBruit() {
        System.out.println("Miaou");
    }
}
```

Interfaces

- Une interface définit un contrat que les classes doivent respecter (`implements`).

```
interface Volant {
    void voler();
}

class Oiseau implements Volant {
    @Override
    public void voler() {
        System.out.println("Je vole !");
    }
}
```

La classe Object : equals et toString

- Toutes les classes en Java héritent implicitement de `Object`.
- La méthode `equals(Object o)` compare les adresses des objets
- La méthode `toString()` par défaut affiche l'adresse mémoire de l'objet.

```
class Animal {
    protected String nom;

    public Animal(String nom) { this.nom = nom; }

    /*@Override public String toString() {return nom;}*/
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Animal("Lion");
        Animal a2 = new Animal("Lion");
        System.out.println(a1.toString()); // Affiche: class Animal @f7b324
        System.out.println(a1.equals(a2)); // false (comparaison d'adresses mémoire)
    }
}
```

Importance de equals pour ArrayList

- `ArrayList` utilise `equals()` pour la recherche.
- Si `equals()` n'est pas redéfini, la recherche se fait par adresse mémoire.

```
import java.util.ArrayList;

class Animal {
    protected String nom;

    public Animal(String nom) { this.nom = nom; }

    @Override
    public boolean equals(Object obj) {
        if (obj == null || getClass() != obj.getClass()) return false;
        Animal animal = (Animal) obj;
        return nom.equals(animal.nom);
    }
}

public class Main {
    public static void main(String[] args) {
        ArrayList<Animal> animaux = new ArrayList<>();
        Animal a1 = new Animal("Lion");
        animaux.add(a1);
        System.out.println(animaux.indexOf(new Animal("Lion"))); // Avant surcharge de equals: -1, après: 0
    }
}
```

Conclusion

- Très proche de TypeScript mais beaucoup moins permissif
- Les surcharges de `toString` et `equals` sont importantes