Fiche 4
Une classe sans objet : notion de classe utilitaire
(quand une classe se prend pour un objet)

Dans certains cas, des calculs et autres constantes sont indépendants d'un contexte d'objet.

Par exemple la constante PI, le taux de TVA en vigueur, l'algorithme de recherche dans un tableau, dans une liste, …

Les propriétés (attributs ou opérations) ne dépendant pas d'une instance d'un objet sont dites de portée de classe (opposé à portée d'instance) et sont notifiées **static** dans la plupart des langages de programmation.

Une classe peut ne comporter que des propriétés statiques (**static**), comme Math, Arrays en java, dans ce cas elle est qualifiée d'**utilitaire**. (terminologie UML), elle sera chargée en mémoire, comme toute classe utilisée par le projet. Attention, l'inverse n'est pas vrai !

Par exemple, la classe Arrays contient une grande variété de méthodes pour manipuler des tableaux (tris, recherche, initialisation, ...) ainsi que des fonctions pour représenter le contenu d'un tableau sous forme d'une chaîne de caractères (à l'image de print_r en PHP).

Pour se prémunir de toute instanciation, ce type de classe (utilitaire), redéfinit le constructeur par défaut (celui sans argument) en le rendant privé.

Extrait (http://download.oracle.com/javase/6/docs/api/java/util/Arrays.html) :

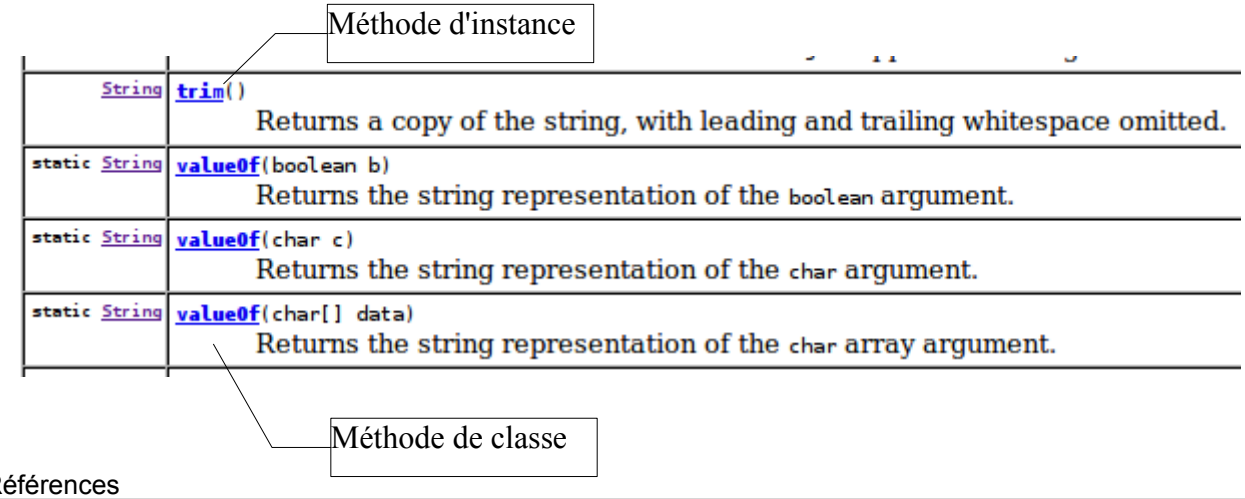
```
public class Arrays {

    // Suppresses default constructor, ensuring non-instantiability.
    private Arrays() {    }

  ... la suite est composée de déclarations et définitions de propriétés static
    (attributs et méthodes)
}
```

Attention : Rien n'interdit à une classe de comporter des propriétés statiques **et** d'instances. Elle est alors en mesure de jouer les deux rôles (utilitaire et type d'objet)

Exemple, la classe String en java dispose de méthodes **valueOf** (surchargées = plusieurs versions qui ne varient que par la signature de ses paramètres) qui retournent la représentation textuelle de son argument :



Références

Understanding Instance and Class Members :

http://download.oracle.com/javase/tutorial/java/javaOO/classvars.html

Glossaire lié à Java : http://java.sun.com/docs/glossary.html

EXERCICES

(extrait de l'API de la classe String)

java.lang

Class String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

Method Summary

String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this `String` to upper case using the rules of the default locale.
static String	**valueOf**(float f) Returns the string representation of the `float` argument.
static String	**valueOf**(int i) Returns the string representation of the `int` argument.
static String	**valueOf**(long l) Returns the string representation of the `long` argument.
static String	**valueOf**(Object obj) Returns the string representation of the `Object` argument.

Vérifier la validité des instructions de la deuxième colonne du tableau à partir de l'extrait de la classe String ci-dessus et de la déclaration et initialisation de la variable prenom (de type String) ci-dessous.

```
String prenom = "Salima";
```

	Instruction	Résultat (valeur de x) et Commentaire
1	`String x = prenom.toString();`	
2	`String x = String.toUpperCase(prenom);`	
3	`String x = String.toUpperCase();`	
4	`String x = String.valueOf(prenom);`	
5	`String x= String.valueOf(prenom.toString());`	
6	`String x= prenom.toUpperCase();`	
7	`String x = prenom.valueOf(prenom);`	
8	`String x = toLowerCase(prenom);`	
9	`Object x = prenom;`	
10	`String x = String.valueOf(prenom.length());`	

Fiche 5
Héritage d'implémentation

Le terme **Héritage**, lorsqu'il s'applique à des classes, fait référence à de la **réutilisation** d'une **implémentation**.

Exemple : MaFenetre **extends** JFrame {…}

La classe MaFenetre hérite de l'implémentation de la classe JFrame. Elle se comportera en tout point donc comme une JFrame si elle ne redéfinie pas certains de ces (ou ses) comportements.

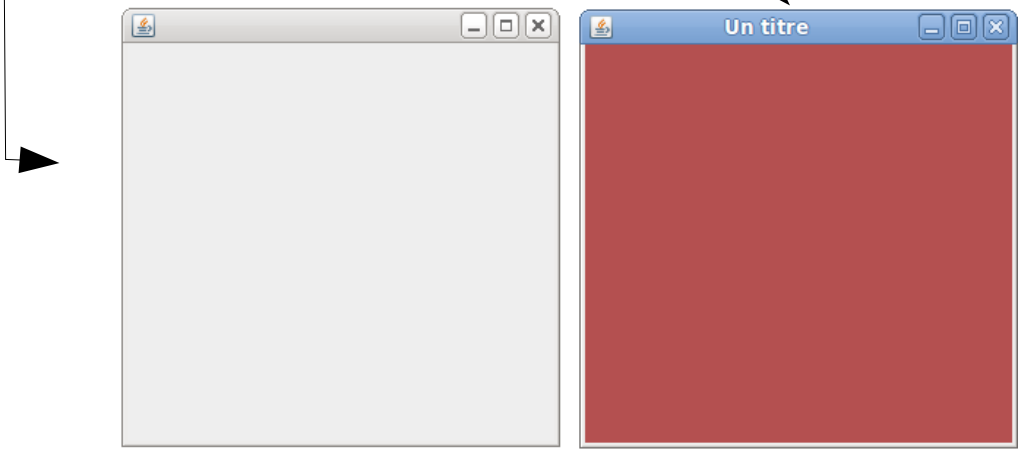
Exemple :

```
public class Fenetre extends JFrame {
       public Fenetre(String titre) {
        super(titre);
        this.setBackground(new Color(180,80,80));
      }
  }
```

Ci-dessous, le résultat de l'exécution du programme Application qui instancie un JFrame de base (à gauche) et Fenetre (à droite).

```
public class Application {
  public static void main(String[] args) {
        JFrame f = new JFrame();
        f.setSize(300,300);
        f.setVisible(true);

        f = new Fenetre("Un titre");
        f.setSize(300,300);
        f.setVisible(true);
  }
}
```



On remarquera que la classe Fenetre redéfinit simplement l'**état** de base, et pas son **comportement** (quoique que la couleur soit le résultat d'un comportement)

Pour exploiter les parties héritées, les langages objet définissent des mots clés spécifiques. En voici quelques uns définis par le langage java :

extends : relation déclarative entre une classe mère et une classe enfant
super() : première instruction dans le cors d'un constructeur (appel le constructeur hérité)
super.méthode() : dans le cors d'une méthode redéfinie, appel de la version héritée.
***InterfaceName*.super**.*méthode() : nouveau depuis Java8, les interfaces peuvent avoir des implémentations par défaut (méthodes)*
final : en caractéristique d'une classe, interdit tout héritage. (*leaf* en UML)
abstract : en caractéristique d'une classe, interdit toute instanciation.

Fiche 5 Bis

Que faire de son héritage ? Les 5 comportements clés.

Répartis en 3 catégories : Conservateur, Réformateur et Nihiliste.

Conservateur (fidèle) Respecte de la tradition. Maintient la tradition en conservant l'héritage.

1. Conserver l'existant sans **rien changer**.
   => On reprend à la lettre les caractéristiques héritées. C'est le comportement par défaut.

2. Conserver l'existant en **ajoutant une touche personnelle**.
   => On hérite puis on ajoute du code.

Réformateur Renouvelle la tradition : Va de la simple transformation à la métamorphose.

3. On **tente de conserver le sens**
   => mais on l'exprime par du **code entièrement nouveau**.
4. On **ne conserve pas le sens**
   => et on raconte autre chose. Attention le problème de sens peut révéler une mauvaise analyse/conception.

Nihiliste (du latin *nihil* <<rien>>) Gomme la tradition. Supprime toutes traces des caractéristiques comportementales héritées.

5. On remplace la partie héritée par une **expression du vide**.
   => Même remarque que 4.

A titre d'illustration : la classe Joueur.

```
class Personne {
  private String nom;
  ...
  public String toString() {
    return this.nom ;
 }
}
```

```
class Joueur extends Personne {
  private int niveau;

  @Override
  public String toString() {. . . }
}
```

et appliquons les 5 comportements d'héritage sur la méthode héritée.

appel de la méthode héritée

```
1     public String toString() { return super.toString(); }
      // Simple appel à la méthode héritée. Ne change donc rien.
      // C'est le comportement par défaut (en absence de redéfinition).
```

```
2     public String toString() { return super.toString() + " : "+this.niveau;}
      // Appel la méthode héritée puis ajoute une touche personnelle.
```

```
3     public String toString() {
         return getClass().getName() + ": niveau = " this.niveau +}
      // Redéfinit la méthode héritée dans l'esprit attendu.
```

```
4     public String toString() {
            return "Je passe incognito et je le dis..."; }
      // Redéfinit la méthode héritée dans un autre esprit.
```

```
5     public String toString() { /* je me tais */ return ""; }
      // Décide de ne rien faire et rend donc une chaîne vide.
```

Fiche 6
classe abstraite

Une interface est une structure définissant un type. Elle est peut être vide ou composée de déclarations d'opération (et de constantes statiques). Voir par exemple l'interface *Shape* ou *List* en Java.

Ces structures ne peuvent pas être instanciées car elle sont incomplètement définies : les méthodes n'ont pas de corps (on parle alors d'opérations). Une interface est une abstraction. Depuis Java8, les interface peuvent déclarer des opérations avec une implémentation par défaut.

Dans le but toujours de factoriser du code, il peut arriver que l'on soit amené à placer de l'implémentation (des attributs) dans ce type de structure, or cela ne peut se faire que dans une classe. Ces classes là sont souvent incomplète, car certaines méthodes ne sont pas implémentées. Ce type de classe, non destiné à l'instanciation directe, est qualifié d'**abstrait** (***abstract***). On parle alors de classes abstraites.

Exemple :

```
abstract public class Personne {
	private String nom;
	private String prenom;
	private String login;

	public Personne(String nom, String prenom) {
		super();
		this.nom = nom;
		this.prenom = prenom ;
		this.makeLogin();
	}

	@Override
	public String toString() {
		return this.nom + " " + this.prenom + " " + this.login;
	}


	abstract protected String makeLogin();
}


public class UtilisateurSysteme extends Personne {
	public Utilisateur(String nom, String prenom) {
		super(nom, prenom);
	}

	protected void makeLogin() {
	  // TODO algorithme make this.login
	}
}
public class UtilisateurComptable extends Personne {
	public Utilisateur(String nom, String prenom) {
		super(nom, prenom);
	}

	protected void makeLogin() {
	  // TODO algorithme make this.login
	}
}
```

Classe abstraite

Une conséquence de

La force du modèle est ici

Méthode abstraite (elle n'a pas de corps)

Implémentation attendue de la méthode ici

Implémentation attendue de la méthode ici