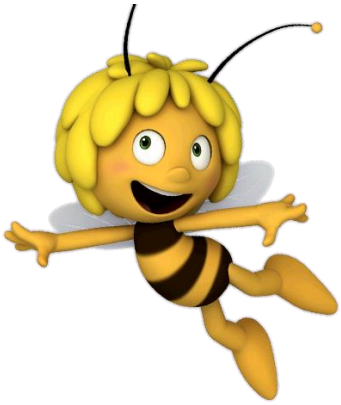
TD : Héritage et polymorphisme au pays des abeilles



Vous venez d'arriver sur le projet **honeybee** qui possède actuellement les classes suivantes :

```
public class HoneyBee {

	public String doYourJob() {
		return "I'm a HoneyBee!";
	}

	public String fly() {
		return "I believe, I can fly.";
	}
}
```

```
public class Queen extends HoneyBee{

    public String doYourJob(){
        return "I'm a Queen, any questions?";
    }
}
```

```
public class Worker extends HoneyBee{

    public String doYourJob(){
        return "I'm a worker, I work all day!";
    }
}
```

```
public class Drone extends HoneyBee {

	public String doYourJob() {
		return "I'm a drone, I'm going to date our Queen!";
	}

}
```

---

1. **Rétro-conception** :
   Pour bien entrer dans le projet, commencer par modéliser le **diagramme de classes** correspondant au code précédent

2. Ce projet dispose également d'une classe **`Main`** contenant le jeu d'essai suivant :

```
1   public class Main {
2
3         public static void main(String[] args) {
4
5               HoneyBee queen = new Queen();
6               HoneyBee firstWorker = new HoneyBee();
7               HoneyBee secondWorker = new Worker();
8               HoneyBee thirdWorker = new Worker();
9               HoneyBee firstDrone = new HoneyBee();
10              HoneyBee secondDrone = new Drone();
11
12              System.out.println(queen.doYourJob());
13              System.out.println(firstWorker.doYourJob());
14              System.out.println(secondWorker.doYourJob());
15              System.out.println(thirdWorker.doYourJob());
16              System.out.println(firstDrone.doYourJob());
17              System.out.println(secondDrone.doYourJob());
18
19              System.out.println("------------------");
20
21              System.out.println(queen.fly());
22              System.out.println(firstWorker.fly());
23              System.out.println(secondWorker.fly());
24              System.out.println(thirdWorker.fly());
25              System.out.println(firstDrone.fly());
26              System.out.println(secondDrone.fly());
27        }
28
29  }
```

Quel **affichage devrait-on visualiser sur la console** à l'exécution de ce **`main`** ?

3. Pour l'instant, considérons que **seule** la classe **`Worker`** est modifiée de la manière suivante :

```
public String doYourJob() {
      return super.doYourJob() +  " I'm a worker, I work all day!";
}
```

a. Quel **affichage devrait-on visualiser sur la console** pour les lignes 12 à 17 de ce projet ?

b. Pour la suite, on suppose que le `super.doYourJob()` a également été ajouté dans les classes **Queen** et **Drone**. Qu'en est-il de la classe **HoneyBee**, est-il pertinent d'ajouter un `super.doYourJob()` ? Justifiez votre réponse.

4. Tout est bien impersonnel et notre client souhaite maintenant pouvoir **nommer ses abeilles** avec **`Mellifera, Maya, Marguerite, Propolis, Willy`** et **`Didier`**.

   a. Où l'attribut **`name`** doit-il être ajouté ? Bien sûr, le nom ne pourra être attribué qu'une seule fois.

   b. En évitant au maximum la duplication de code, écrire les constructeurs qui permettront de créer les différentes sortes d'abeille avec leur nom, de manière à ce que les lignes 5 à 10 de la méthode **`main`** deviennent :

```
HoneyBee queen = new Queen("Mellifera");
HoneyBee firstWorker = new HoneyBee("Maya");
HoneyBee secondWorker = new Worker("Marguerite");
HoneyBee thirdWorker = new Worker("Propolis");
HoneyBee firstDrone = new HoneyBee("Willy");
HoneyBee secondDrone = new Drone("Didier");
```

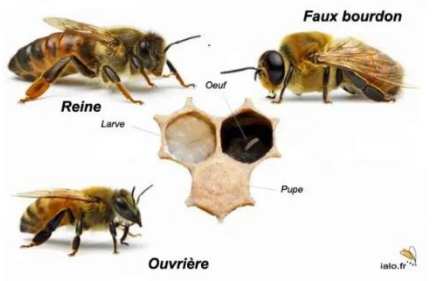
   ce qui provoquera l'affichage suivant à l'exécution des lignes 12 à 17 :

```
Mellifera I'm a HoneyBee! I'm a Queen, any questions?
Maya I'm a HoneyBee!
Marguerite I'm a HoneyBee! I'm a worker, I work all day!
Propolis I'm a HoneyBee! I'm a worker, I work all day!
Willy I'm a HoneyBee!
Didier I'm a HoneyBee! I'm a drone, I'm going to date our Queen!
```

5. **Qualité de code :**

   a. Extrait de https://www.apiculture.net/blog/organisation-du-travail-dans-la-ruche-n144

« *Au sein d'une ruche, des milliers d'abeilles cohabitent et s'entraident pour un seul but commun : la survie de la colonie. La reine, les ouvrières et les faux-bourdons possèdent chacun leurs particularités et doivent alors assumer des tâches bien distinctes.* »



   ⇨ A la lecture de ce texte, quels sont les acteurs de la ruche dans le monde réel ?
   Quels seront donc les ***objets*** qui interagiront entre eux **dans notre application** ?

   ⇨ Comment pouvez-vous donc améliorer votre *design* (code et diagramme de classes) pour représenter au mieux la réalité de la ruche ?

   ⇨ Quel impact cette modification aura-t-elle sur le code actuel de la classe **`main`** ? Faîtes les modifications nécessaires pour pouvoir toujours manipuler, une reine, trois ouvrières et deux bourdons.

b. Rappelez-vous de la méthode **`toString`** de la classe **`Object`**, classe mère de toutes les classes Java, qui est habituellement redéfinie dans les classes filles.

⇨ Y-a-t-il dans cette conception (design) **une/des méthode(s) redéfinie(s)?**
Si oui, laquelle/lesquelles et quel comportement a/ont-elles ?

⇨ Quelle **annotation** pouvez-vous ajouter dans le code afin de faciliter la lecture et la compréhension de ce point de design indiquant qu'un ***polymorphisme*** est potentiellement mis en œuvre. Ajouter cette annotation au(x) bon(s) endroits 😉

6. Une colonie d'abeilles dans un tableau …

Extrait de https://fr.wikipedia.org/wiki/Polymorphisme_(informatique) et plus particulièrement de la rubrique **Intérêt du polymorphisme** :
*En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, **le polymorphisme permet une programmation beaucoup plus générique**. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.*

a. Pour permettre cette programmation plus générique, nous devons tout d'abord regrouper toutes nos abeilles (jusque-là isolées dans les lignes 5 à 10) dans un même tableau.
Déclarez et instanciez un tableau d'abeilles qui regroupera (contiendra) les 6 abeilles :
Mellifera, Maya, Marguerite, Propolis, Willy et Didier.
Pour savoir comment déclarer en Java, consultez l'annexe un peu de documentation sur les tableaux en Java 😉
⇨ **Convention de nommage :** vous nommerez ce tableau **`honeyBees`** car en principe, lorsqu'on manipule un groupe d'objets d'une classe **`Item`**, on reprend le nom de la classe et on ajoute un **`s`** pour nommer le groupe d'objets de cette classe, un groupe d'objets de type **`Item`** devrait donc être nommés **`items`**.

b. Pour illustrer le côté « générique » de la programmation, vous allez écrire en 4 lignes maximum les instructions, qui à partir du tableau d'abeille, vous permettront d'obtenir l'affichage suivant :

```
Mellifera I'm a HoneyBee! I'm a Queen, any questions?
I believe, I can fly.
-----------------------------------------
Maya I'm a HoneyBee! I'm a worker, I work all day!
I believe, I can fly.
-----------------------------------------
Marguerite I'm a HoneyBee! I'm a worker, I work all day!
I believe, I can fly.
-----------------------------------------
Propolis I'm a HoneyBee! I'm a worker, I work all day!
I believe, I can fly.
-----------------------------------------
Willy I'm a HoneyBee! I'm a drone, I'm going to date our Queen!
I believe, I can fly.
-----------------------------------------
Didier I'm a HoneyBee! I'm a drone, I'm going to date our Queen!
I believe, I can fly.
```

c. Pour finir, il est intéressant de savoir de quelle sorte d'abeilles la colonie que l'on vient de créer est constituer. A la fin de la méthode main, écrivez le code qui permet de dénombrer le nombre de reine(s), d'ouvrière(s) et de faux-bourdon(s) dans la colonie et d'afficher ce résultat sous la forme :

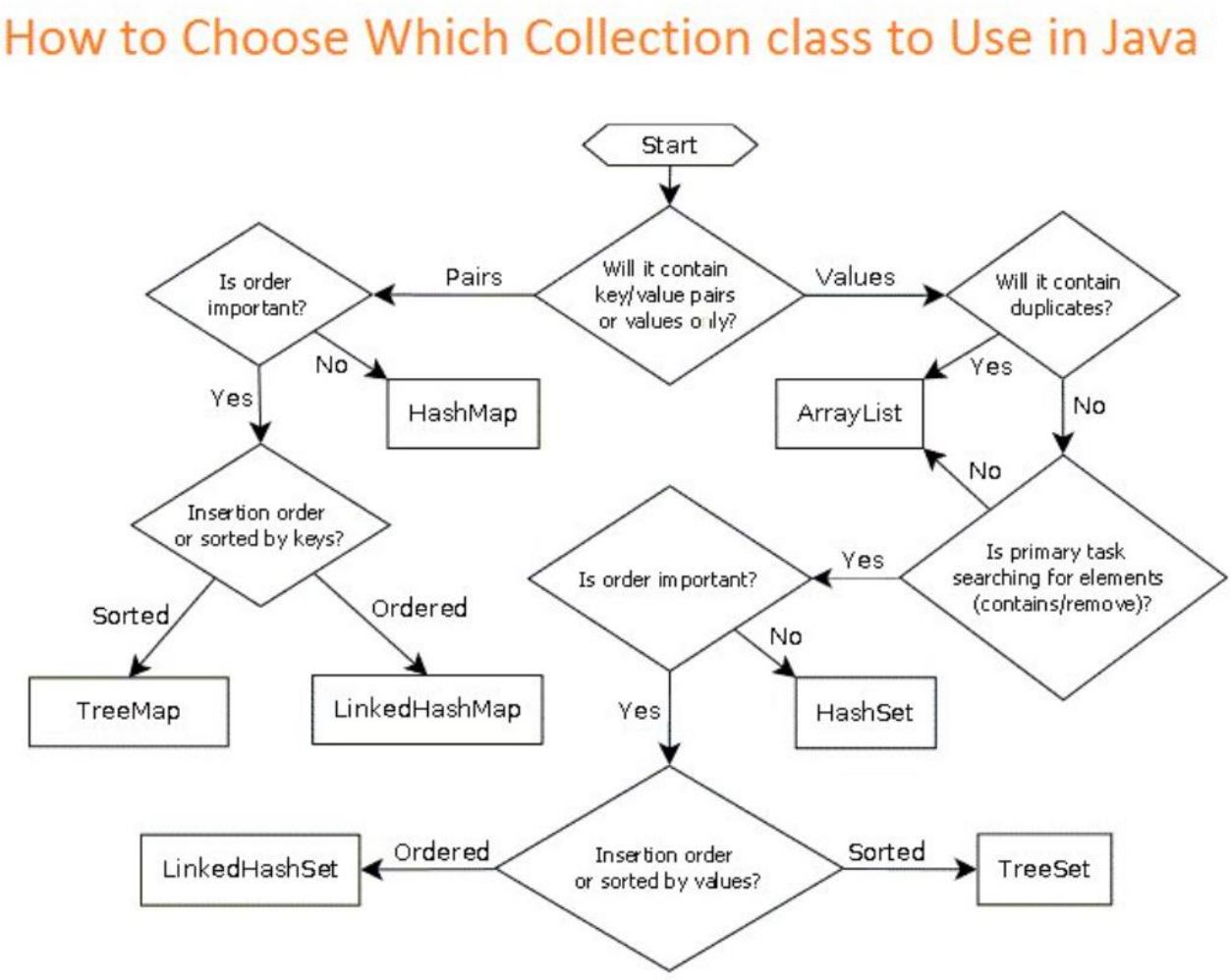
```
My colony has 6 honeybees
 -> 1 queen(s)
 -> 3 worker(s)
 -> 2 drone(s)
```

Pour les plus rapides …

7. Une colonie d'abeilles dans une collection …

L'ajout de nouvelles abeilles dans une structure de données de types tableau n'est donc pas « facile » à mettre en œuvre (« facile » employée ici dans le sens où l'ajout ne se fait pas en une instruction 😉). Dans Java, dès lors que nous souhaitons **manipuler des groupes d'objets**, nous manipulerons ces objets comme une **`Collection`**. Toutes collections disposant déjà d'une méthode **add** implémentée et prête à être utilisée : l'ajout d'une abeille se réduisant ainsi à une seule instruction !

En java, il existe plusieurs classes qui permettent de stocker une collection d'objets.
Le diagramme suivant peut vous aider choisir la *bonne* collection en tenant compte des contraintes du problème que vous avez à résoudre (stockage comme simple valeur ou comme une paire clé-valeur, etc.…), unicité des objets dans la collection, l'ordre des objets a-t-il de l'importance, …)



a. En vous aidant de ce diagramme entourez dans la liste suivante de collections, la collection *idéale*, celle qui pourrait répondre le plus simplement au besoin de notre colonie d'abeilles :

`ArrayList, HashSet, LinkedHashSet, TreeSet, HashMap, TreeMap, LinkedHasMap, LinkedHashSet`

Indice :

Notre collection d'abeilles ne stockera que des abeilles c-a-d que des valeurs, pas de paires clé-valeurs 😉

b. Procédez à l'instanciation de votre collection d'abeilles que vous nommerez **honeybees**.
Ajoutez dans cette collection **6 abeilles (1 reine, 2 faux-bourdons et 3 ouvrières).**
Vous êtes libre sur le nom de vos abeilles 😉

Le site https://www.baeldung.com/java-collections propose un ensemble de guides et d'exemples pour explique comment instancier et manipuler les diverses collections du JDK… à l'image de la documentation suivante sur la classe `ArrayList` (https://www.baeldung.com/java-arraylist)

ArrayList resides within Java Core Libraries, so you don't need any additional libraries. In order to use it just add the following import statement:

```
import java.util.ArrayList;
```

2. Create an *ArrayList*

ArrayList has several constructors and we will present them all in this section.

First, notice that *ArrayList* is a generic class, so you can parameterize it with any type you want and the compiler will ensure that, for example, you will not be able to put *Integer* values inside a collection of *Strings*. Also, you don't need to cast elements when retrieving them from a collection.

Secondly, it is good practice to use generic interface *List* as a variable type, because it decouples it from a particular implementation.

2.1. Default No-Arg Constructor

```
List<String> list = new ArrayList<>();
assertTrue(list.isEmpty());
```

We're simply creating an empty *ArrayList* instance.

2.2. Constructor Accepting Initial Capacity

```
List<String> list = new ArrayList<>(20);
```

Here you specify the initial length of an underlying array. This may help you avoid unnecessary resizing while adding new items.

3. Add Elements to the *ArrayList*

You may insert an element either at the end or at the specific position:

```
List<Long> list = new ArrayList<>();

list.add(1L);
list.add(2L);
list.add(1, 3L);

assertThat(Arrays.asList(1L, 3L, 2L), equalTo(list));
```

Annexe : Un peu de documentation sur les tableaux en Java

Extraits du site https://www.baeldung.com/java-arrays-guide

📣 Retenez bien cette adresse, le site *baeldung* est très utile lorsqu'on code en Java pour retrouver rapidement une syntaxe ou avoir un rapide coup d'œil sur une notion 😉

2. What's an Array?

First things first, we need to define what's an array? According to the Java documentation, an array is **an object containing a fixed number of values of the same type**. The elements of an array are indexed, which means we can access them with numbers (called *indices*).

We can consider an array as a numbered list of cells, each cell being a variable holding a value. In Java, the numbering starts at 0.

There are primitive type arrays and object type arrays. This means we can use arrays of *int, float, boolean, ...* But also arrays of *String, Object* and custom types as well.

3.1. Declaration

We'll begin with the declaration. There are two ways to declare an array in Java:

```
int[] anArray;
```

or:

```
int anOtherArray[];
```

The former is more widely used than the latter.

3.2. Initialization

Now that it's time to see how to initialize arrays. Again there are multiple ways to initialize an array. We'll see the main ones here, but this article covers arrays initialization in detail.

Let's begin with a simple way:

```
int[] anArray = new int[10];
```

By using this method, we initialized an array of ten *int* elements. Note that we need to specify the size of the array.

When using this method, **we initialize each element to its default value**, here 0. When initializing an array of *Object*, elements are *null* by default.

We'll now see another way giving us the possibility to set values to the array directly when creating it:

```
int[] anArray = new int[] {1, 2, 3, 4, 5};
```

Here, we initialized a five element array containing numbers 1 to 5. When using this method we don't need to specify the length of the array, it's the number of elements then declared between the braces.

4. Accessing Elements

Let's now see how to access the elements of an array. We can achieve this by requiring an array cell position.

For example, this little code snippet will print 10 to the console:

```
anArray[0] = 10;
System.out.println(anArray[0]);
```

Note how we are using indices to access the array cells. **The number between the brackets is the specific position of the array we want to access.**

When accessing a cell, if the passed index is negative or goes beyond the last cell, Java will throw an *ArrayIndexOutOfBoundException*.

We should be careful then **not to use a negative index, or an index greater than or equal to the array size**.

5. Iterating Over an Array

Accessing elements one by one can be useful, but we might want to iterate through an array. Let's see how we can achieve this.

The first way is to use the *for* loop:

```
int[] anArray = new int[] {1, 2, 3, 4, 5};
for (int i = 0; i < anArray.length; i++) {
    System.out.println(anArray[i]);
}
```

This should print numbers 1 to 5 to the console. **As we can see we made use of the *length* property. This is a public property giving us the size of the array.**

Of course, it's possible to use other loop mechanisms such as *while* or *do while*. But, as for Java collections, it's possible to loop over arrays using the *foreach* loop:

```
int[] anArray = new int[] {1, 2, 3, 4, 5};
for (int element : anArray) {
    System.out.println(element);
}
```

This example is equivalent to the previous one, but we got rid of the indices boilerplate code. **The *foreach* loop is an option when:**

- we don't need to modify the array (putting another value in an element won't modify the element in the array)
- we don't need the indices to do something else