

2 séances de TP : Découverte de l'héritage & du polymorphisme avec les abeilles

Introduction à la qualité de code

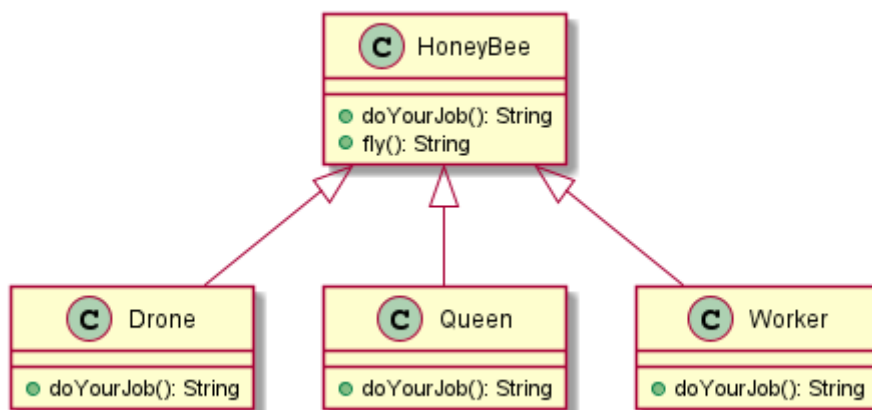
Exercice 1 : Un premier héritage autour des abeilles



1. Créez un projet **honeybee**, comme d'habitude sans module 😊
2. Dans ce projet, créez une première classe **HoneyBee** et implémentez la de la manière suivante :

```
public class HoneyBee {  
  
    public String doYourJob() {  
        return "I'm a HoneyBee!";  
    }  
  
    public String fly() {  
        return "I believe, I can fly.";  
    }  
}
```

3. Implémentez ensuite les classes **Queen**, **Worker** et **Drone** afin que votre code respecte la conception (diagramme de classes) suivante :



Sachant que :

- ➔ L'implémentation de la méthode **doYourJob()** de la classe **Drone** devra retourner :
I'm a drone, I'm going to date our Queen!
- ➔ L'implémentation de la méthode **doYourJob()** de la classe **Queen** devra retourner :
I'm a Queen, any questions?
- ➔ L'implémentation de la méthode **doYourJob()** de la classe **Worker** devra retourner :
I'm a worker, I work all day!

4. Vérifier le **bon comportement de votre implémentation** à l'aide de **jeux d'essai** :

- a. Créez une classe **HoneyBeeMain** , pour l'instant vierge de toute implémentation.
- b. Connaissez-vous **gist** ? C'est un outil de partage de morceaux de code (**snippets**) très pratique proposé par **Github**.

Utilisez l'adresse suivante : <https://unil.im/honeybee>

(eh oui, il existe un raccourcisseur d'URL sur l'ENT de l'Université 😊) pour vous rendre sur le *gist* suivant :

<https://gist.github.com/iblasquez/ab9a9201e40e6e1cdcfa16f78e9b75f3> qui contient le code que vous devez copier et coller dans votre classe **HoneyBeeMain**.

***Remarque :** Si vous avez un compte **github**, vous pouvez rapidement partager vos *snippets* avec **gist** en vous rendant sur : <https://gist.github.com/>*

*Pour en savoir plus sur **gist**, consultez sa documentation sur :*

<https://docs.github.com/en/get-started/writing-on-github/editing-and-sharing-content-with-gists/creating-gists>

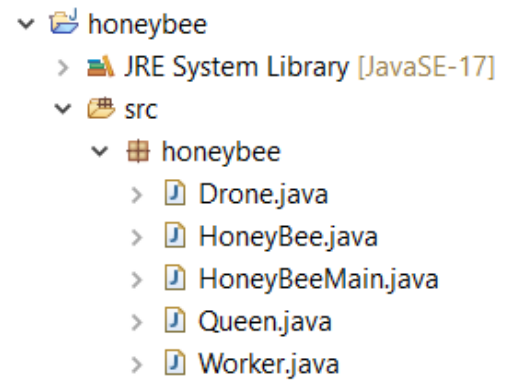
- c. Exécutez votre jeu d'essai. Vous devriez voir s'afficher sur la console le jeu d'essai suivant :

```
-----  
Some honeybees  
-----  
I'm a Queen, any questions?  
I'm a HoneyBee!  
I'm a worker, I work all day!  
I'm a worker, I work all day!  
I'm a HoneyBee!  
I'm a drone, I'm going to date our Queen!  
-----  
I believe, I can fly.  
I believe, I can fly.  
I believe, I can fly.  
I believe, I can fly.  
I believe, I can fly.  
I believe, I can fly.
```

5. Qualité du code : Cohérence : Mise en place de paquetages pour mieux structurer le projet

Actuellement, la vue **Package Explorer** vous indique que votre projet s'organise de la manière ci-contre, c.-à-d. que toutes ces classes appartiennent au même paquetage **honeybee**.

D'ailleurs, si vous consultez le code des classes **Drone**, **HoneyBee**, **HoneyBeeMain**, **Queen** et **Worker**, vous constaterez qu'ils commencent tous par l'instruction **package honeybee;**



Pour donner **plus de cohérence au code**, il serait intéressant de regrouper les classes par *thématique*.

Actuellement, deux thématiques pourraient être identifiées :

→ Une thématique autour du **cœur de métier de l'application** : **HoneyBee**, **Drone**, **Queen**, **Worker**

→ Une thématique autour des **jeux d'essais** pour tester le **comportement** de l'application : **HoneyBeeMain**

Mieux structurer son projet consiste à **regrouper au sein d'un même paquetage des classes cohérentes** nous allons donc créer deux paquetages :

→ Un paquetage **model** regroupant les classes métiers : **HoneyBee**, **Drone**, **Queen**, **Worker**

→ Un paquetage **application** regroupant les jeux d'essais : **HoneyBeeMain**

a. Commençons par **créer le paquetage application depuis le fichier source directement**

Ouvrir le fichier **HoneyBeeMain.java** et remplacer la première instruction **package honeybee;** par **package honeybee.application;**

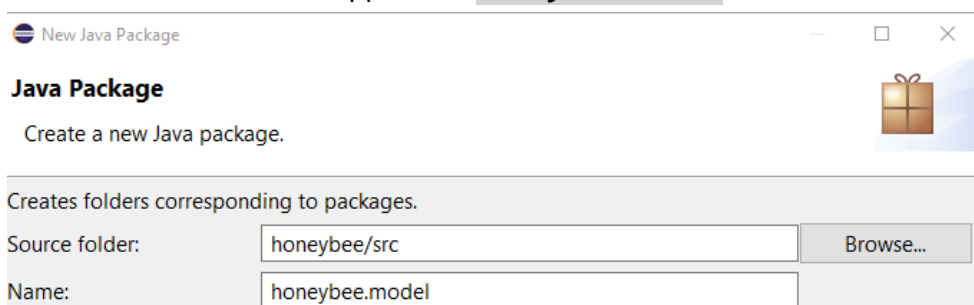
Aidez-vous ensuite de l'IDE pour créer ce paquetage : une croix rouge apparaît dans la marge, cliquez dessus et sélectionnez : **Move 'HoneyBeeMain.java' to package 'honeybee.application'**
N'oubliez pas de sauvegarder votre fichier HoneyBeeMain.java après cette modification.

*Si vous consultez la vue **Package Explorer**, vous constaterez que l'IDE a bien créé le nouveau paquetage. Des erreurs de compilation apparaissent désormais dans la classe, nous y reviendrons plus tard, une fois que les classes métiers auront été déplacées dans leur nouveau paquetage.*

b. Pour créer le **paquetage model**, nous allons procéder différemment et utiliser la **vue Package Explorer**. Dans la vue Package Explorer, placez-vous sur le **paquetage honeybee**.

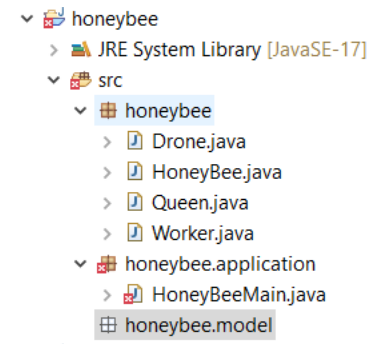
A l'aide d'un clic droit, choisissez **New→Package** (à la place du clic droit, vous pouvez aussi passer par le menu **File**) :

- Vérifiez que le **source folder** est bien **honeybee/src**
- Complétez le **name** de manière à voir apparaître **honeybee.model** avant de sélectionner **Finish**



Dans la vue package Explorer le **paquetage honeybee.model** apparaît : il est vide (icône blanche).

Depuis cette vue, faites glisser les classes métiers (**Drone, HoneyBee, Queen, Worker**) dans ce paquetage (cliquez sur **OK** et/ou **Continue** si des boîtes de dialogue se présentent)

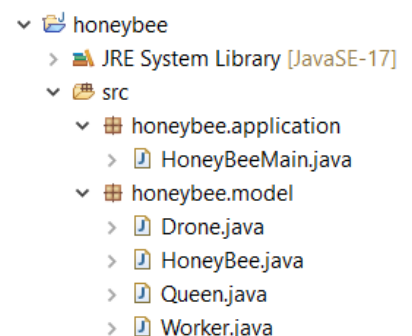


Les classes apparaissent maintenant sous **honeybee.model** dans la vue Package explorer
Si vous ouvrez le code, la première ligne de ces classes est devenue : **package honeybee.model;**

c. Il ne reste plus qu'à corriger les erreurs de compilation dans **HoneyBeeMain** dues au fait que les classes métiers ont changées de paquetages. Deux possibilités pour corriger ces erreurs :

- Cliquez sur chaque croix rouge dans la marge et sélectionnez **Import '...' (honeybee.model)**
- Ou rendez-vous dans le menu **Source → Organize Import** pour ajouter en une seule fois tous les imports manquants. Cette option vous sera très souvent utile, retenez-la bien et retenez surtout son raccourci clavier **Ctrl + Shift + O** qui vous fera gagner du temps 😊

Quelle que soit la solution choisie,
n'oubliez pas de **sauvegarder votre fichier** et **d'exécuter HoneyBeeMain** pour vérifier que les jeux d'essais sont toujours conforme au comportement attendu (celui donné en page 2) !



d. Rétro-conception sur le paquetage model ⇒ Diagramme de classes métier 😊

Pour visualiser le diagramme de classes métier, procédez à une petite rétro-conception (reverse-engineering) de votre code via **PlantUML** en sélectionnant **uniquement le package model c-a-d uniquement les classes métiers** !

Le diagramme de classes que vous visualisez devrait être similaire au diagramme de classes de la première page de cet énoncé 😊

6. Utilisation du mot clé **super** pour appeler une méthode de la classe mère

- a. Dans les classes filles, ajoutez comme première instruction de la méthode **doYourJob**, un appel à la méthode **doYourJob** de la classe mère, de manière à ce que l'exécution du main provoque maintenant l'affichage suivant sur la console :

```
-----  
Some honeybees  
-----  
I'm a HoneyBee! I'm a Queen, any questions?  
I'm a HoneyBee!  
I'm a HoneyBee! I'm a worker, I work all day!  
I'm a HoneyBee! I'm a worker, I work all day!  
I'm a HoneyBee!  
I'm a HoneyBee! I'm a drone, I'm going to date our Queen!  
-----
```

- b. Ajoutez **un attribut name** dans la classe **HoneyBee**. Quelle doit-être la **visibilité** de cet attribut pour continuer à respecter le **principe d'encapsulation**, mais pour qu'il reste **visible/accessible dans toutes les classes de la hiérarchie d'héritage**.

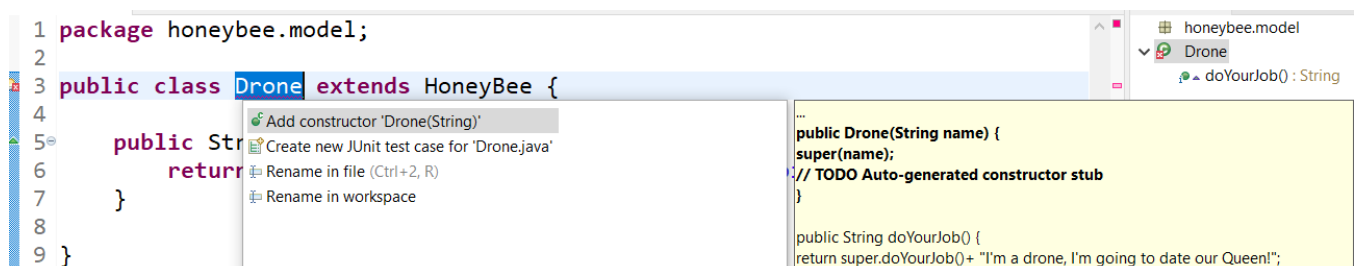
☐ **Getteur/Setteur** : Bien sûr, le nom ne pourra être attribué qu'une seule fois, **ajoutez donc seulement un getteur** et n'oubliez pas d'indiquer dans sa déclaration que l'attribut doit être

☐ **Constructeur primaire** : Aidez-vous de l'IDE pour générer le constructeur primaire qui prend en paramètre l'unique attribut de la classe (**Source** → **Generate Constructor using Fields**)

... De nombreuses erreurs apparaissent dans votre projet, nous allons les traiter classe après classe en nous aidant de l'IDE ...

☐ Rendez-vous dans la **classe Drone** : l'IDE vous indique une erreur de compilation. *En Java, lorsqu'il y a un héritage, la classe fille hérite de tous les attributs et de toutes les méthodes de la classe mère. Un Drone a maintenant un nom et pour instancier un Drone, il va falloir écrire un constructeur avec le paramètre **name**.*

Dans la classe **Drone**, cliquez sur la croix rouge, vous propose de générer le constructeur **Public Drone(String name)** qui nous manque. Cliquez une fois sur **Add constructor Drone(String)** vous propose dans le cadre jaune, le code que l'IDE pourrait générer pour corriger cette erreur de compilation c-a-d un **appel au constructeur de la classe mère en utilisant simplement le mot clé super**



```
1 package honeybee.model;  
2  
3 public class Drone extends HoneyBee {  
4  
5     public String doYourJob() {  
6         return super.doYourJob();  
7     }  
8  
9 }
```

...
public Drone(String name) {
 super(name);
 // TODO Auto-generated constructor stub
}

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

Pour corriger l'erreur de compilation, il ne vous reste plus qu'à écrire directement ce code, ou à le générer automatiquement en cliquant une nouvelle fois sur l'option **Add constructor Drone(String)**. N'oubliez pas de sauvegarder la classe **Drone** !... et de supprimer le commentaire `//TODO` dans le constructeur que vous venez de générer 😊

❑ Faites en sorte de corriger également les erreurs de compilation des classes **Queen** et **Worker**.

❑ Que se passe-t-il dans la classe **HoneyBeeMain** ? L'ajout du constructeur à un paramètre a provoqué des erreurs de compilation dans cette classe pourquoi ? Modifiez le code de la classe **HoneyBeeMain** de manière à instancier correctement les abeilles en utilisant les noms suivants : **Mellifera**, **Maya**, **Marguerite**, **Propolis**, **Willy** et **Didier** pour obtenir, à l'exécution, un affichage similaire à :

```
-----  
Some honeybees  
-----  
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!
```

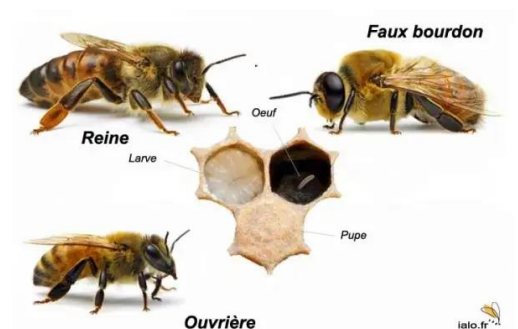
Remarque : A noter qu'une autre méthode nécessite une petite modification pour pouvoir visualiser exactement cet affichage 😊

Exercice 2 (relatif à R2.03): Introduction à la qualité de code : Bon sens, aide d'un linter (SonarLint), respect des règles de simplicité, cohérence du code

1. Bon sens suite à l'observation du monde réel:
Concret vs abstrait ? (instanciable vs non instanciable)
Utilisation du mot clé **abstract** pour rendre une
classe abstraite

Dans le monde réel, les acteurs de la ruche sont :
la reine, les ouvrières et les faux-bourdon

Dans notre application, **Queen**, **Worker** et **Drone** devront donc être des classes concrètes c.-à-d. instanciables afin qu'elles puissent permettre de créer des objets de type **Queen**, **Worker** et **Drone** qui pourront interagir entre eux et faire vivre la ruche.



Par contre, **aucun objet de type HoneyBee** ne sera directement instancié dans notre application donc la classe **HoneyBee doit devenir abstraite** (pour éviter d'écrire par inadvertance un **new HoneyBee** qui n'aurait pas de sens....)

- a. Ajoutez le mot clé **abstract** au bon endroit dans votre code afin de rendre la classe **HoneyBee** abstraite, ce qui provoque instantanément des erreurs de compilation dans la classe **HoneyBeeMain**. Pourquoi ?
- b. Faites en sorte de supprimer les erreurs de compilation de la classe **HoneyBee** afin que l'instanciation des différentes abeilles compile correctement et provoque toujours l'affichage suivant :

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

2. Amélioration de la qualité de code à l'aide d'un linter : SonarLint par exemple

a. Installation du plug-in SonarLint sous Eclipse (via Eclipse Market)

SonarLint est un plug-in proposé par **Sonar Source** (<http://www.sonarsource.com>) destiné à votre IDE préféré (Eclipse, IntelliJ) et qui va vous donner des indications sur la qualité de votre code :

Sur le site <http://www.sonarlint.org>, il est présenté de la manière suivante :

SonarLint is a **Free** and **Open Source** IDE extension that identifies and helps you fix quality and security issues as you code. Like a spell checker, SonarLint squiggles flaws and provides real-time feedback and clear remediation guidance to deliver clean code from the get-go.

Il y a deux manières d'installer un plug-in sous Eclipse :

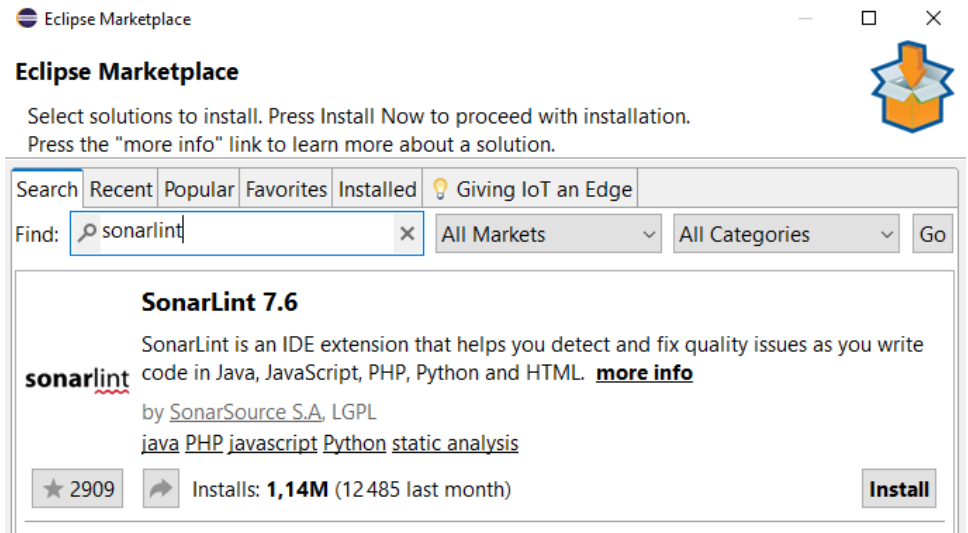
- soit à partir du menu **Help** → **Install New Software...**
- soit à partir du menu **Help** → **Eclipse Market**



⇒ Le plus simple pour installer plug-in **SonarLint** sous Eclipse est de passer par l'**Eclipse Market** comme nous allons le détailler 😊

⇒ Rendez-vous dans **Help → Eclipse Marketplace...**

Dans la barre de recherche, tapez **sonarlint** et cliquez sur **Go** pour rechercher le plug-in **SonarLint**.



⇒ Cliquez ensuite sur le bouton **Install**.

⇒ **Acceptez** les termes de la licence pour pouvoir cliquer sur **Finish**.

.... Le plug-in est en train de s'installer, il faut attendre un peu (vous pouvez suivre la barre de progression de l'installation en bas à droite de votre IDE) jusqu'à ce que vous voyez un message qui vous demande de redémarrer l'IDE

⇒ Cliquez sur **Restart Now** pour que l'IDE puisse redémarrer et prendre en compte ce plug-in.

🔊 *N'oubliez pas qu'après l'installation d'un plug-in, un redémarrage d'Eclipse est toujours nécessaire pour que ce plug-in soit bien pris en compte par l'IDE* 🔊

⇒ Une fois l'IDE redémarré, vérifiez que vous vous trouvez toujours sur la perspective Java (en haut à droite de votre IDE)



b. Lancement d'une analyse SonarLint

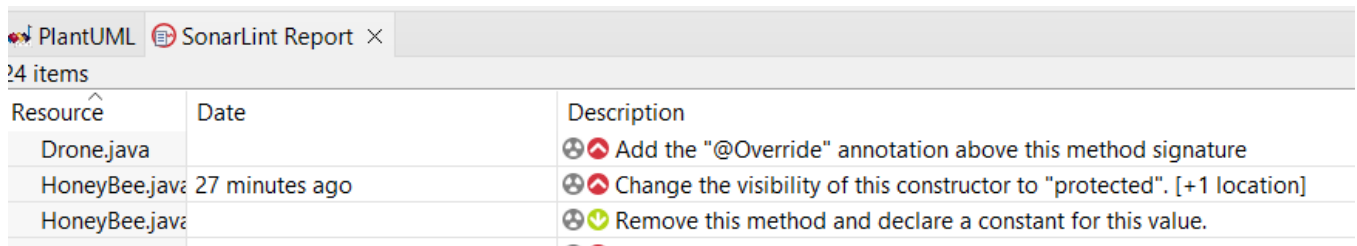
Pour lancer votre première analyse SonarLint, placez-vous dans la vue Package Explorer sur le projet **honeyBee**, puis via un clic droit choisir **'SonarLint -> Analyze...'**

Cliquez sur **OK** sur la fenêtre de dialogue qui s'ouvre à vous....

Si vous cochez *Always proceed without asking*, vous ne verrez plus ce message 😊

c. Analyse du code par SonarLint : Mise en évidence d'issues

Une fois, l'analyse terminée, une nouvelle **vue SonarLintReport** s'ouvre au bas de votre IDE.



Resource	Date	Description
Drone.java		⊕ Add the "@Override" annotation above this method signature
HoneyBee.java	27 minutes ago	⊕ Change the visibility of this constructor to "protected". [+1 location]
HoneyBee.java		⊕ Remove this method and declare a constant for this value.

→ Dans l'onglet **Resource**, une **classe du projet** est affichée.

→ Dans l'onglet **Description**, un **problème de qualité de code** dans cette classe est indiquée.

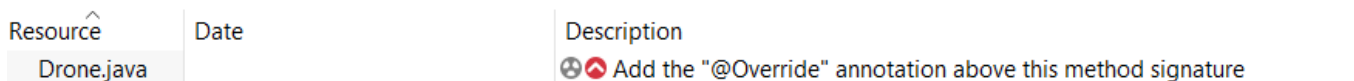
SonarLint appelle ces *problèmes autour de la qualité de code* : des **issues**.

Une issue ne provoque pas une erreur de compilation : le code compile, le projet s'exécute.

Une issue indique que la qualité de code peut être améliorée et qu'il pourrait y avoir danger !

- Les issues annotées en **rouge** sont des **issues majeures**.
- Les issues annotées en **vert** sont des **issues mineures**.

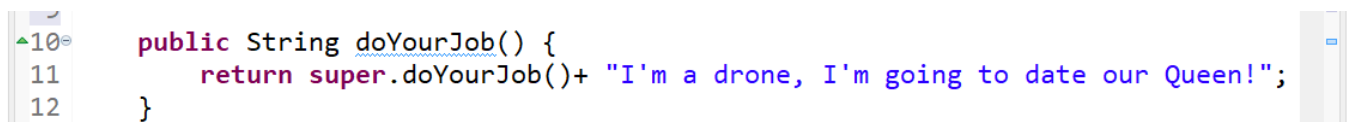
d. Zoom sur la première issue et amélioration de la qualité de code



Resource	Date	Description
Drone.java		⊕ Add the "@Override" annotation above this method signature

Si vous double-cliquez sur le première issue **Add the @Override annotation ...**

Vous vous retrouvez dans la classe **Drone**, à l'endroit où le problème de qualité (l'issue) a été détecté.



```
10 public String doYourJob() {
11     return super.doYourJob()+ "I'm a drone, I'm going to date our Queen!";
12 }
```

Il n'est pas nécessaire de passer par la vue **SonarLintReport** pour prendre conscience que le code que l'on est en train d'écrire a un potentiel problème de qualité.

Sur la copie d'écran ci-dessus, comme dans votre IDE, vous voyez qu'un petit rectangle bleu clair dans la marge de droite : c'est le signe que **SonarLint** a détecté une issue. Si vous cliquez dessus, vous retrouverez le même message que dans la vue **SonarLint** qui vous a mené ici.

☐ **Corrigez donc cette issue** en annotant la méthode **doYourJob()** de la classe **Drone** avec un **@Override**

☐ Lorsque vous sauvegardez votre fichier, le carré bleu disparaît puisque l'issue vient d'être corrigé.

☐ Revenez jeter un petit coup d'œil dans la vue **SonarLint Report**.

L'issue a-t-elle disparu ? si oui, laissez tel quel. Si non, vous pouvez relancer une analyse pour vérifier que l'issue a bien disparu.

e. Zoom sur l'issue suivante et amélioration de la qualité de code :

Resource	Date	Description
HoneyBee.java	47 minutes ago	Change the visibility of this constructor to "protected". [+1 location]

Cette fois, ci avant de double-cliquez pour vous rendre dans la classe **HoneyBee**.
Placez-vous simplement sur l'issue **et via un clic droit** sélectionnez **Rule Description**
pour ouvrir la vue **SonarLint Rule Description**

SonarLint Report SonarLint Rule Description ×

Constructors of an "abstract" class should not be declared "public" (java:S5993)

Code smell Major

Abstract classes should not have public constructors. Constructors of abstract classes can only be called in constructors of their subclasses. So there is no point in making them public. The `protected` modifier should be enough.

Noncompliant Code Example

```
public abstract class AbstractClass1 {
    public AbstractClass1 () { // Noncompliant, has public modifier
        // do something here
    }
}
```

Compliant Solution

```
public abstract class AbstractClass2 {
    protected AbstractClass2 () {
        // do something here
    }
}
```

La vue **SonarLint Rule Description** explique pourquoi l'issue a été détectée et comment la corriger.

→ Tout d'abord, en rappelant en quelques phrases la *bonne pratique* à respecter pour écrire un code de qualité.

→ en illustrant **le problème de qualité qui a été détecté dans votre code** qui est un problème similaire au code proposé dans la rubrique **Noncompliant Code Example**.

→ **en indiquant comment corriger cette issue** dans votre code en vous proposant une **Compliant Solution**.

Depuis la vue SonarLint Report, double-cliquez maintenant sur cette issue pour vous rendre dans la classe **HoneyBee**.

- ☐ Vous constaterez en effet que la classe est **abstract** et le constructeur est **public**.
- ☐ Vous n'avez plus qu'à changer la visibilité de ce constructeur et le rendre **protected** pour faire disparaître l'issue et sauvegarder votre modification.

Quelques mots supplémentaires pour vous convaincre que le constructeur d'une classe abstraite doit être protected ...

Si la classe est abstraite, le constructeur ne doit jamais être appelé en dehors de la classe, puisqu'aucune instanciation n'est possible. Pour éviter de l'appeler par inadvertance et écrire du code plus « sûr », il est recommandé de passer ce constructeur en **protected** `HoneyBee(String name)` : ainsi seules les classes filles pourront l'appeler et c'est ce qu'elles font avec l'instruction **super(name)**; 😊

f. Zoom sur l'issue suivante :

SonarLint Report × SonarLint Rule Description		
2 items		
Resource	Date	Description
HoneyBee.java		Remove this method and declare a constant for this value.

❑ Consultez la **Rule Description** de cette issue.

Les exemples qui vous sont donnés dans la description vous indiquent que l'issue a été levée car **SonarLint** a identifié un **code smell** (mauvaise odeur) qui n'est autre qu'un **nombre magique** (**magic number**).

❑ **Rendez-vous dans le code** en double cliquant sur l'issue.

Est-il pertinent d'extraire cette chaîne dans une constante ?

C'est à vous de décider, cette chaîne n'étant utilisée qu'une seule fois dans le code. Faites confiance à votre libre arbitre 😊

Mise en garde sur les outils : Un linter, comme *SonarLint*, vous aide à **détecter de potentiels problèmes de qualité** dans votre code, mais la décision de corriger ou non ces problèmes immédiatement ou d'attendre vous revient.

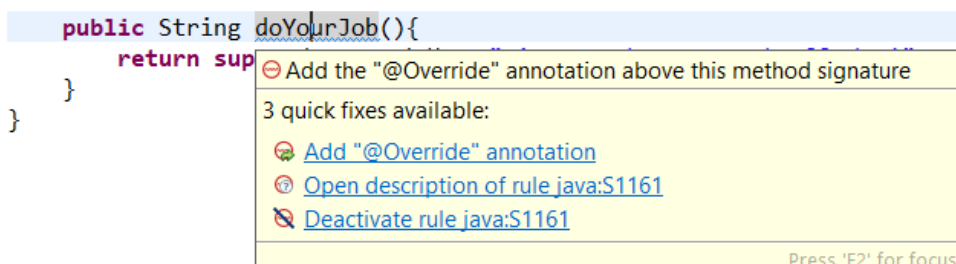
Attention, il n'est pas toujours pertinent de suivre les indications de SonarLint à la lettre et de corriger immédiatement toutes les issues, vous devez faire la part des choses en tenant compte de votre bon sens et de votre expérience 😊

g. Zoom sur les issues des classes Queen et Worker et amélioration de la qualité de code

Queen.java	Add the "@Override" annotation above this method signature
Worker.java	Add the "@Override" annotation above this method signature

Comme pour la classe **Drone**, améliorer la qualité de votre code en corrigeant ces deux issues par l'ajout de l'annotation **@Override** sur les méthodes **doYourJob()** pour montrer que ces méthodes **redéfinissent** la méthodes **doYourJob()** de la classe mère **HoneyBee**.

A noter que si vous placez votre souris sur le code souligné par une petite vaguelette bleue (ajoutée par le plug-in SonarLint) et si vous cliquez, le plug-in SonarLint vous fera directement des propositions pour corriger votre code ou en savoir plus sur cette issue ... 😊



L'utilisation de l'annotation **@Override** dans le code indique explicitement la **redéfinition d'une méthode** et la possibilité de mettre en place d'un éventuel polymorphisme (voir annexe pour un peu plus de détail)

h. Quid de la dernière issue ?

HoneyBeeMai

⚠️ Replace this use of System.out or System.err by a logger.

Cette issue apparaît pour l'instant un peu moins de 20 fois dans la vue **SonarLint Report**. Nous allons factoriser le code pour qu'elle n'apparaisse plus qu'une seule fois.

❑ Extraire une méthode ayant comme responsabilité l'affichage d'un message

Dans la classe **HoneyBeeMain**, créez une méthode **message** implémentée de la manière suivante :

```
private static void message(String text) {  
    System.out.println(text);  
}
```

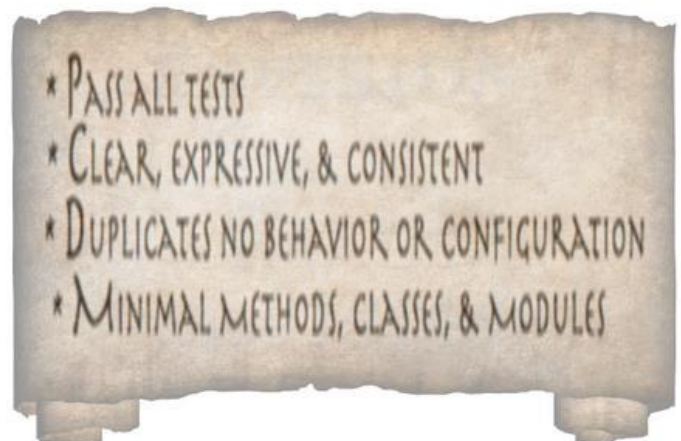
Et remplacez tous les `System.out.println` de la méthode **main** par des appels à **message**. Sauvegardez et exécutez pour vérifier que votre application continue à fonctionner correctement.

La méthode **message** extraite de la méthode **main** doit absolument être **static** pour que l'application puisse fonctionner correctement. **Pourquoi ?** Si vous ne connaissez pas la réponse discutez-en avec vos enseignants 😊

Relancez une analyse **SonarLint**, vous constatez que nous n'avons pas supprimé toutes les issues, mais uniquement celles que nous jugeons pertinentes par rapport à l'état actuel de notre développement. Après cette analyse *automatique* du code par SonarLint, nous allons maintenant procéder à une analyse *manuelle* en relisant le code et examinant une à une les 4 règles de simplicité (présentées en cours).

3. Focus sur les 4 règles de simplicité

Peut-on aller plus loin concernant l'amélioration de notre code ? Une fois, les issues pertinentes de SonarLint corrigées, nous pouvons continuer par une relecture de code (appelé aussi **revue de code**) en nous interrogeant, par exemple, sur le **respect des règles de simplicité**, règles que nous avons vues dans le cours sur la qualité logicielle.



a. Règle n°1 : Tous les tests passent

Même si les tests automatisés sous **JUnit** n'ont pas encore été implémentés, nous disposons actuellement de **jeux d'essais qui testent (vérifient & valident) le comportement de l'application**.

Le code est fonctionnel et répond à nos attentes en terme de comportement donc nous pouvons considérer que cette règle est, pour le moment, bien respectée dans l'état actuel de nos connaissances et que pour l'instant *les jeux d'essais* font office de *tests*.

b. Règle n°2 : Un code clair, expressif et consistant

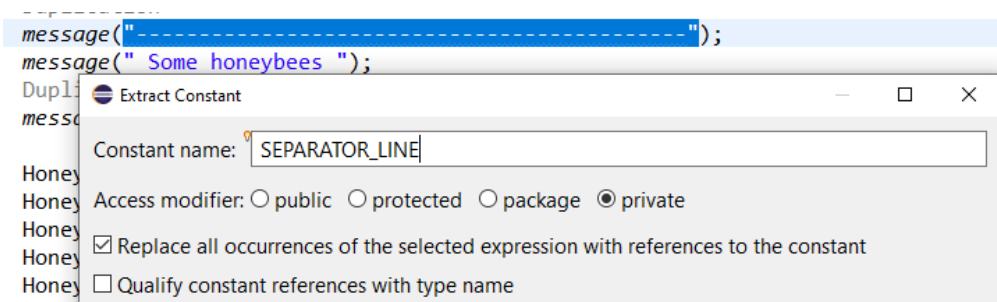
Pour cette règle, vous devez penser à vérifier que les **standards de codage** sont bien respectés (camelCase, convention classes, méthodes, attributs, etc, ...) et que le nommage du code est explicite c.-à-d. qu'il reflète bien la **terminologie métier** et **l'intention métier**. A priori, ce devrait être le cas 😊

c. Règle n°3 : Eviter la duplication !

Voyez-vous une duplication dans ce code ?

A priori, oui ! Dans la classe **HoneyBeeMain**, la chaîne `"-----"` qui se répète trois fois .

Cette chaîne peut alors être considérée comme une **mauvaise odeur (code smell)** apparentée à un **Nombre Magique (Magic Number)**



Pour supprimer cette mauvaise odeur et nettoyer le code (**clean code**), il suffit de sélectionner la chaîne, de faire un clic droit puis **Refactor → Extract Constant**. Dans **Constant Name**, écrire **SEPARATOR_LINE**, puis cliquez sur **OK**.

d. Règle n°4 : Les méthodes et classes doivent avoir un nombre de lignes de code minimal

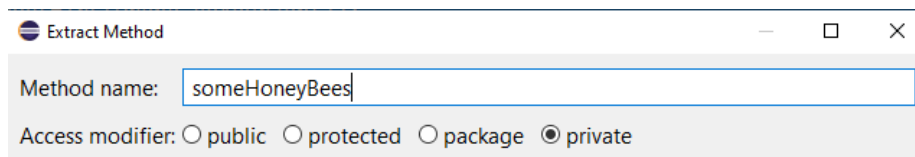
☐ **Retravailler le code des méthodes pour réduire le nombre d'instructions par méthode (nombre minimal de lignes) => afin de faciliter la lecture du code**

Pour illustrer ce point, nous allons sélectionner tout le code actuellement contenu dans la méthode **main** et l'extraire via l'IDE dans une méthode que nous appellerons **someHoneyBees**.

Vous comprendrez l'utilité de cette méthode par la suite,

lorsque nous écrirons de nouveaux jeux d'essais, mais cette fois-ci pour des colonies d'abeilles 😊

Comment procéder ? L'IDE vous aide à extraire rapidement et sûrement des lignes de code une méthode à l'aide de l'IDE : sélectionner l'ensemble des lignes à extraire, **clic droit** puis **Refactor** → **Extract Method**.



Après avoir sélectionné toutes les instructions contenues dans le **main**, procédez à un **Extract Method** à l'aide de l'IDE de manière à réduire la méthode **main** au nombre minimal de ligne possible à savoir une seule ligne 😊

```
public static void main(String[] args) {  
    someHoneyBees();  
}
```

Astuce IDE pour consulter rapidement le code d'une méthode : Placez votre curseur sur `someHoneyBees()` de la méthode `main`, puis faites un **CTRL + clic gauche**, l'IDE vous déplace automatiquement à l'emplacement de l'implémentation de la méthode `someHoneyBees`

Une bonne pratique du développement logiciel est le **diviser pour régner** (*divide and conquer*). Il est conseillé de **décomposer un problème complexe en petits problèmes** plus simple à résoudre. Les **Extract Method** sont de bonnes pratiques pour **découper en éléments algorithmique simples** (Apprentissage Critique 1 de la compétence 2) et améliorer ainsi la lisibilité du code 😊

- ❑ S'interroger sur la présence pertinente d'une méthode dans une classe par rapport au rôle que joue cette classe dans le programme
 - ⇒ permet de **réduire la taille des classes**, de faire apparaître **plus de classes (de petite taille)** afin de **mieux répartir les responsabilités**.

Dans la classe **HoneyBeeMain**, on pourrait se poser la question de la pertinence de la méthode **message**. Est-ce que tout ce qui concerne l'interface graphique en mode console, ne pourrait pas se trouver dans une classe **Console** ?

✓ Créez une nouvelle classe **Console** et placez-la dans un nouveau paquetage **honeybee.gui**. Déplacez la méthode **message** dans la classe **Console**.

Vous pouvez déplacer cette méthode directement à l'IDE. Dans la classe **HoneyBeeMain**, placez-vous sur la signature de la méthode **message**, puis **clic droit** suivi de **Refactor** → **Move...** Cliquez sur le bouton **Browse** et tapez dans **Choose a type** : **Console** pour faire apparaître la classe **Console**. Sélectionnez-la et cliquez sur **OK**, puis encore sur **OK**, puis sur **Continue** lorsque l'IDE vous demande de changer la visibilité de la méthode en **public**.

- ➔ Ouvrir la classe **Console** pour constater que la méthode **message** s'y trouve désormais avec la visibilité **public**. Nous décidons de laisser cette méthode en méthode statique pour l'instant, mais rien ne nous empêcherait d'en faire une méthode d'instance 😊
- ➔ Ouvrir la classe **HoneyBeeMain** pour constater que l'IDE a ajouté un **import** `honeybee.gui.Console`; et que l'appel à la méthode se fait dorénavant sous la forme : `Console.message(...)`;

Sauvegardez et exécutez l'application pour vérifier que tout fonctionne encore correctement après ce remaniement de code.

✓ Relisez le code de la classe `HoneyBeeMain`, y-a-t-il autre chose qui pourrait être de la responsabilité de la **Console** ?

⇒ Avec l'aide de l'IDE (**Refactor** → **Move**), placez-vous sur **SEPARATOR_LINE** (sur la ligne de son instantiation) et déplacez cette instantiation de **SEPARATOR_LINE** dans la classe **Console**. Sauvegardez et exécutez l'application pour vérifier que tout fonctionne encore correctement après ce remaniement de code.

✓ Ne pouvez-vous pas encore améliorer la répartition des responsabilités et la lisibilité de la classe ?

⇒ Sélectionnez par exemple les trois instructions suivantes :

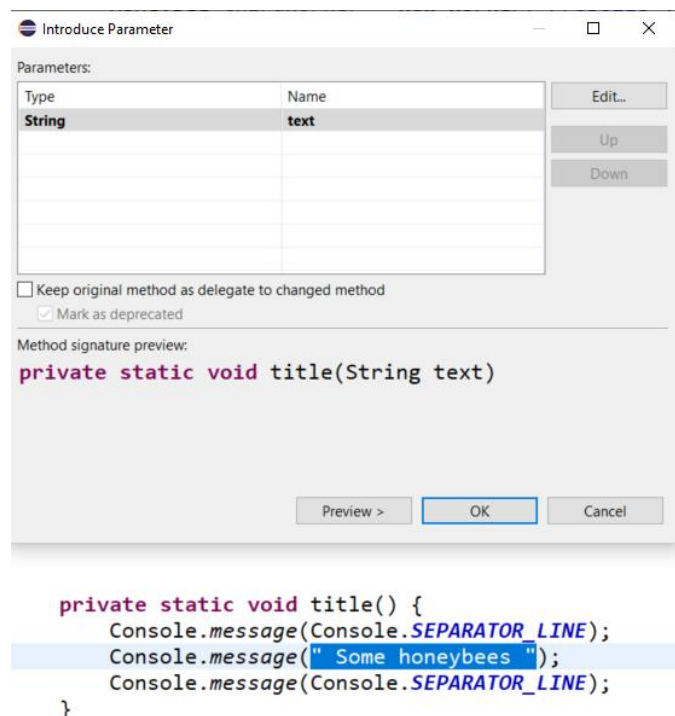
```
Console.message(Console.SEPARATOR_LINE);  
Console.message(" Some honeybees ");  
Console.message(Console.SEPARATOR_LINE);
```

⇒ A l'aide de l'IDE (**Refactor** → **Extract Methode**), regroupez ces trois instructions dans une méthode **title** dans la classe **HoneyBeeMain**.

```
private static void someHoneyBees() {  
    title();  
  
    HoneyBee queen = new Queen("Mellifera");  
    HoneyBee firstWorker = new Worker("Maya");  
}
```

Pour l'instant, vous disposez d'une méthode `title()` sans paramètre d'entrée.

Comme il est préférable **d'écrire du code ré-utilisable pour éviter la duplication** vous préféreriez que votre méthode dispose d'un paramètre d'entrée qui pourrait contenir le texte du titre à afficher.



⇒ L'IDE peut encore une fois vous aider dans cette tâche.

Dans la méthode **title**, sélectionnez `" Some honeybees "`.

A l'aide d'un **clic droit** suivi de **Refactor** → **Introduce Parameter**, modifiez le string en **text** dans la colonne **Name** comme le montre la copie d'écran précédente, afin qu'après avoir cliqué sur **OK**, vous obteniez :

```
private static void title(String text) {  
    Console.message(Console.SEPARATOR_LINE);  
    Console.message(text);  
    Console.message(Console.SEPARATOR_LINE);  
}
```

⇒ Il ne vous reste plus qu'à déplacer la méthode **title** dans la classe **Console** à l'aide d'un **Refactor** → **Move**

✓ De même, on pourrait aussi s'amuser à créer une méthode (avec un nom explicite) pour l'instruction `Console.message(Console.SEPARATOR_LINE);` et la déplacer dans la méthode **Console**. Est-ce nécessaire pour le moment ? A vous de voir, la qualité de code est subjective, on s'arrête quand on est satisfait du résultat, quitte à retravailler le code un peu plus tard 😊

En tous cas, la classe **Console** est maintenant une classe que vous pourrez, si vous le souhaitez, **réutiliser, voir même enrichir** en fonction de vos besoins dans vos prochains projets 😊

🔊 Pour continuer sur de bonnes bases :

Votre programme doit être fonctionnel et votre classe **Console** devrait ressembler à :

```
package honeybee.gui;
```

```
public class Console {
```

```
    public static final String SEPARATOR_LINE = "-----  
-----";
```

```
    public static void message(String text) {  
        System.out.println(text);  
    }
```

```
    public static void title(String text) {  
        message(SEPARATOR_LINE);  
        message(text);  
        message(SEPARATOR_LINE);  
    }  
}
```

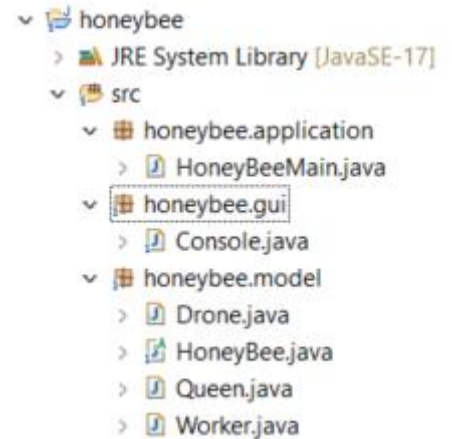
4. Vérifier la « forte » cohérence du code

Grâce à la **mise en place de paquetages**, les classes qui s'articulent autour d'une même thématique sont regroupées.

Pour assurer la **cohérence du code**, il est donc important **d'organiser (architecturer) son projet en paquetages** de la manière suivante.

Par convention, les paquetages sont souvent nommés de la manière suivante :

- **model** pour les classes métier
- **gui** ou **view** pour les IHM
- **controller** pour les contrôleurs
- **util** pour les utilitaires
- **application** pour les classes contenant un main
- etc ...



Il est important de vérifier régulièrement si les classes dans un même paquetage sont bien cohérentes entre elles, sinon il ne faut pas hésiter à créer de nouveaux paquetages ou sous-paquetages pour bien structurer votre projet et pouvoir y replonger plus facilement à n'importe quel moment par la suite.



La **qualité de code est subjective**, il faut savoir s'arrêter à moment donné lorsque nous pensons avoir le **bon compromis entre lisibilité et responsabilités** 😊

... quitte à reprendre un travail autour de la qualité lorsque de nouvelles fonctionnalités auront été ajoutées et les lignes de code augmentées ...

Qualité de code : Dorénavant quand vous écrirez du code :

Ayez le réflexe **SonarLint** !!!

Et

Interrogez-vous sur le **respect des quatre règles de simplicité** (tests, consistance, duplication et code minimal !)

Pour information :

Quand on essaye **d'améliorer la qualité de code** en s'aidant d'**outil tel qu'un linter** ou en procédant à **une simple relecture du code** (en se focalisant sur les règles de simplicité par exemple), on dit en génie logiciel qu'on pratique **une analyse statique du code**

« **statique** » car cette analyse permet d'obtenir des informations sur le code et le comportement d'un programme lors de son exécution sans réellement l'exécuter.

A contrario , utiliser **un debugger** sera une **analyse dynamique** du code car un debugger (qui fera l'objet d'un futur TP) suit **l'exécution d'un programme**.

... Retournons à nos abeilles ...

Exercice 3: Un tableau d'abeilles

Dans la nature, les abeilles ne vivent pas de manière isolées, mais elles vivent en **groupe** (appelé aussi colonie).

La première idée qui vous vient à l'esprit pour implémenter le plus simplement possible un groupe d'abeilles est la **structure de données : tableau**



1. Mise en place d'une méthode prête à accueillir nos jeux d'essais autour d'un tableau d'abeilles :

- ➔ Vous décidez donc d'ajouter dans la classe **HoneyBeeMain** une **méthode privée statique `arrayOfHoneyBees`** qui vous permettra d'écrire quelques jeux d'essais autour du polymorphisme en manipulant votre colonie d'abeille au travers d'un tableau.
Commencez cette méthode par un petit message du style :
`Console.title(" Array of honeybees ");`
- ➔ Lancez l'exécution pour voir s'afficher le message dans votre console. N'oubliez pas d'appeler la méthode **`arrayOfHoneyBees`** dans la méthode **`main`**, sinon elle ne s'exécutera pas 😊

L'implémentation de méthode `arrayOfHoneyBees` va être composée des trois mêmes parties que celles de la méthode `someHoneyBees`, à savoir :

- ✓ L'**instanciation** de l'objet manipulé dans ce jeu d'essais, à savoir le tableau d'abeilles
- ✓ L'appel et l'affichage du **`doYourJob`** de chaque abeille de la colonie
- ✓ L'appel et l'affichage du **`fly`** de chaque abeille de la colonie

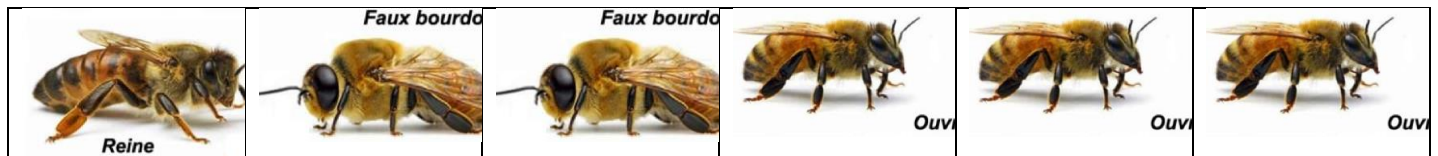
2. Mise en place d'un tableau d'abeilles composée initialement de 6 abeilles

- ➔ Pour commencer, vous devez déclarer un **tableau d'abeilles** : `HoneyBee[] honeyBees`

Bonne pratique Convention de nommage : 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`**.

Remarque : ici la colonie d'abeille est un objet qui contient plusieurs objets (*abeilles*) (c-a-d de type **`HoneyBee`**, c'est pour cela que nous avons choisi d'appeler cette colonie : **`honeyBees`**.

➔ Nous souhaitons que ce tableau d'abeilles regroupe (contienne) les 6 abeilles suivantes :



En vous de la documentation autour des tableaux en Java, **instanciez en une ligne la colonie d'abeille en respectant la disposition précédente dans le tableau d'abeilles (1 reine, 2 drones, 3 ouvrières).** Pour le nom des abeilles, vous êtes libres, de reprendre les noms précédents ou de laisser libre cours à votre imagination 😊

3. Appel et affichage du **doYourJob** de chaque abeille de la colonie

Extrait de [https://fr.wikipedia.org/wiki/Polymorphisme_\(informatique\)](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.*

Pour illustrer le côté « générique » de la programmation, **possible grâce à l'héritage et au polymorphisme**, vous allez écrire à l'aide d'une boucle pour classique :

```
for (int i = ...; i < ...; i++) {  
    ...  
}
```

une instruction unique, qui à partir du tableau d'abeille, vous permettra d'obtenir, à l'exécution de la méthode **arrayOfHoneyBees**, un affichage similaire au suivant (seuls les noms de vos abeilles peuvent changer suivant votre créativité 😊)

Array of honeybees

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

4. Appel et affichage du **fly** de chaque abeille de la colonie

De la même manière, mettre en œuvre le polymorphisme autour du **fly** pour vous permettre d'obtenir, à l'exécution de la méthode **arrayOfHoneyBees**, un affichage similaire au suivant (aux noms près des abeilles selon votre créativité 😊)

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

5. Savez-vous qu'il existe plusieurs syntaxes pour une boucle **for** en java ?

En java, il existe deux types de boucle « **pour** » :

- ✓ **Le **for** dit « classique »** (appelé ***for** -use index on array* par l'IDE Eclipse)

```
for (int i = 0; i < honeyBees.length; i++) {  
    ...  
}
```

- ✓ **Le **foreach**** (appelé ***for** -iterate over an array or an Iterable* par l'IDE Eclipse)

```
for (HoneyBee honeyBee : honeyBees) {  
    ...  
}
```

➔ Implémentez rapidement un **for** avec l'aide de votre IDE :

Placez-vous à la fin de la méthode **arrayOfHoneyBees** .

Ecrivez **for** puis utilisez l'auto-complétion de votre IDE : **CTRL+Espace** sous Eclipse

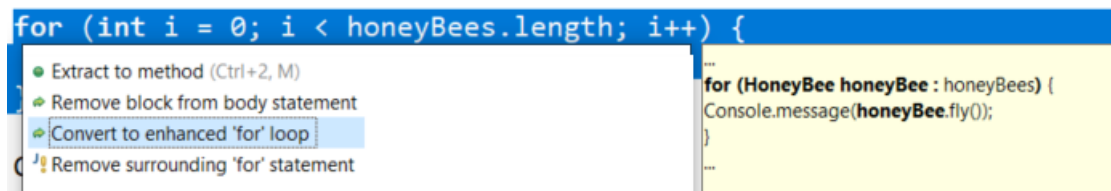
- ✓ Si vous choisissez **for-use index on array**, l'IDE vous écrira directement l'implémentation du **for** « classique » sur le tableau **honeybees**
- ✓ Si vous choisissez **for-iterate over an array or an Iterable**, l'IDE vous écrira directement l'implémentation du **for each** sur le tableau **honeybees**

Astuce : Utilisez l'auto-complétion de votre IDE pour écrire rapidement vos boucles **for** 😊

.... Avant de continuer, assurez-vous d'avoir effacer le code test généré précédemment et que le code contenu dans la méthode **arrayOfHoneyBees** soit uniquement celui produit pour répondre au point **4. Appel et affichage du fly de chaque abeille de la colonie** c-a-d uniquement composé de deux **for**...

➔ Transformez rapidement un **for** classique en **foreach** avec l'aide de votre IDE :

- Sélectionnez tout le code du **for** « classique » permettant l'appel et l'affichage du fly de chaque abeille de la colonie. Cliquez alors à droite pour pouvoir choisir **Quick Fix**
- Choisir l'option **Convert to enhanced for loop**, puis **double cliquez** sur cette option pour remplacer le code que vous venez de sélectionner par le code montré dans le cadre jaune, comme l'indique la copie d'écran ci-dessous :



- Sauvegardez et exécutez afin de vous vérifier que vous obtenez toujours le même affichage 😊

6. Ajout d'une abeille dans la colonie sous forme de tableau ?

Et si on vous demande maintenant d'ajouter une nouvelle ouvrière dans votre tableau, comment devriez-vous procéder ? Est-ce facile ? Puis un nouveau faux-bourdon ?

On ne vous demande pas ici d'implémenter l'ajout de ces abeilles dans le tableau (vous avez sûrement dû déjà faire ce genre de petit algorithme au semestre précédent), mais juste de vous rappeler de l'algo et de l'expliquer à votre enseignant de TP 😊

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 !

Exercice 4 : Une collection d'abeilles



1. Mise en place d'une méthode prête à accueillir nos jeux d'essais autour d'une collection d'abeilles

- ➔ Ajoutez dans la classe **HoneyBeeMain** une méthode **privée statique** `collectionOfHoneyBees` qui vous permettra d'écrire quelques jeux d'essais autour du polymorphisme en manipulant votre colonie d'abeille au travers d'une collection.

Commencez cette méthode par un petit message du style :

```
Console.title(" Collection of honeybees ");
```

- ➔ Lancez l'exécution pour voir s'afficher le message dans votre console.

2. Mise en place d'une collection d'abeilles composée initialement de 6 abeilles

- ➔ **Choix de conception** : Pour commencer, vous devez déterminer quelle collection concrète vous allez utilisée 😊

En vous aidant de l'*Annexe : Un peu de documentation à propos des Collections*, choisissez la collection qui vous semble la plus pertinente pour cette application.

Pour effectuer ce choix, focalisez-vous sur le diagramme de la dernière partie de cette documentation : **5. Pour savoir quelle collection concrète choisir dans votre application.**

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 😊

- ➔ **Implémentation** : Procédez ensuite à l'instanciation d'une collection d'abeilles. Veillez bien sûr à mettre en application la bonne pratique de nommage présentée précédemment, c.-à-d. à nommer votre collection d'abeilles : **honeybees**
🔊 Si vous ne savez pas comment instancier une collection, aidez-vous du site <https://www.baeldung.com/java-collections> qui propose de nombreux guides pour chaque collection du JDK 😊

- ➔ Ajoutez dans cette collection **6 abeilles (1 reine, 2 faux-bourçons et 3 ouvrières)**, à l'aide de la méthode **add** (disponible pour toutes les collections comme indiqué dans la partie 3. de l'annexe : Un peu de documentation à propos des Collections)
Comme précédemment, vous êtes libre sur le nom de vos abeilles 😊

3. Appel et affichage du **doYourJob** et du **fly** de chaque abeille de la collection : Polymorphisme à la rescousse !!!

Dans **la même boucle for**, mettre en œuvre le polymorphisme autour du service **doYourJob** et du service **fly** afin d'obtenir à l'exécution de la méthode **collectionOfHoneyBees** un affichage similaire au suivant (aux noms près des abeilles selon votre créativité 😊)

```
-----  
Collection of honeybees  
-----  
Mellifera I'm a HoneyBee!I'm a Queen, any questions?  
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.  
-----  
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.  
-----
```

4. Simplification de la mise en place de la collection

- ➔ Commentez, à l'aide de `/*...*/`, tout le code écrit précédemment pour l'instanciation de la collection et l'ajout des 6 abeilles.
- ➔ Savez-vous que vous pouvez remplacer tout ce code par une seule instruction ?
Pour cela, cherchez de la documentation du côté de la méthode statique **asList** de la classe **Arrays**
Implémentez cette instruction de manière à ce que l'exécution de méthode **collectionOfHoneyBees** donne le même affichage que précédemment.

Ce type d'implémentation pourrait vous être utile très prochainement pour implémenter l'étape Arrange des tests unitaires. Gardez cette remarque dans un coin de votre esprit 😊

Exercice 5 : Si on jouait un peu dans le bac à sable (sandbox)



Connaissez-vous la signification du terme **sandbox** en informatique ?

Extrait de <https://www.it-connect.fr/informatique-quest-ce-quune-sandbox/>

Une sandbox, que l'on peut traduire en français par "bac à sable", est un terme qui désigne un mécanisme de sécurité dont l'objectif est de créer un environnement temporaire et isolé pour exécuter un programme ou ouvrir un fichier. Grâce à cette sandbox et à l'isolation, on élimine les risques d'infection du système principal. En complément, les sandbox sont utiles aux développeurs pour tester un programme au fur et à mesure de son développement.
[...]

Le concept de sandbox n'est pas nouveau : il existe depuis les années 1970. Les développeurs utilisaient ce concept pour effectuer des tests et des simulations sur leur programme. Aujourd'hui, il y a différents types de sandbox et les acteurs de la sécurité l'utilisent pour de l'analyse de fichiers, parfois sans que vous vous en rendiez compte, car tout se joue dans le Cloud.

Pour ce TP, nous utiliserons le terme **sandbox**, au sens des années 1970 (car nous ne sommes pas dans une ressource de sécurité), c'est-à-dire qu'un bac à sable va nous permettre de faire des essais, de tester des choses, d'expérimenter des méthodes, ...

1. Créez une classe **SandBox** avec une méthode **main** dans le paquetage **honeybee.application**

Remarque Bonne Pratique : Lorsque vous saurez utilisé les gestionnaires de version (git en l'occurrence), il sera préférable de faire une branche **sandbox** pour faire vos expérimentations, plutôt que de créer une classe **SandBox**, car en principe, nous ne devrions trouver qu'un seul **main** par projet !

... Le but de ce bac à sable est d'expérimenter les méthodes **getClass** et **getName**, ainsi que l'opérateur **instance of**

2. Expérimentation des méthodes `getClass` de la classe `Object` `getName` de la classe `Class`

Un site écrit en français, très connu de tout développeur Java depuis les débuts du langage, est le site de **Jean Michel Doudoux : Développons en Java** dont l'adresse est :

https://www.jmdoudoux.fr/accueil_java.htm

→ Rendez-vous dans le **chapitre 23. La gestion dynamique des objets et l'introspection** avec le lien suivant <https://www.jmdoudoux.fr/java/dej/chap-introspection.htm>

Nous nous focaliserons uniquement sur la **partie 23.1.1. L'obtention d'un objet de type `Class`** qui, cela tombe bien, se trouve en début de document 😊.

→ Le premier point **23.1.1.1. La détermination de la classe d'un objet** fait référence à la méthode `getClass` de la classe `Object`.

a. Commencez par lire cette partie

b. Implémentez au début de la méthode `main` de la classe `SandBox`, le code suivant qui est une adaptation du code de JM Doudoux à notre projet actuel.

```
HoneyBee melli = new Queen("Mellifera");
HoneyBee may = new Worker("Maya");
HoneyBee will = new Drone("Willy");

Console.title("Appel à getClass de la classe Object ");
Console.message("Classe de l'objet melli : " + melli.getClass());
Console.message("Classe de l'objet may : " + may.getClass());
Console.message("Classe de l'objet will : " + will.getClass());
```

c. Exécutez pour illustrer et bien comprendre l'utilité d'un appel à la méthode `getClass` 😊

→ Le second point **23.1.1.2. L'obtention d'un objet `Class` à partir d'un nom de classe** fait référence aux méthodes `forName` et `getName`

Passer ce point, car il ne nous intéresse pas pour les expérimentations que nous souhaitons faire dans ce TP 😊

➔ Le troisième point **23.1.1.3 Une troisième façon d'obtenir un objet Class** fait référence à l'utilisation du mot clé **class** dans un programme pour avoir un objet de type **Class** pour pouvoir ensuite appliquer sur cet objet, la méthode **getName** de la classe **Class**.

a. Commencez par lire cette partie

b. Implémentez à la fin de méthode **main** de la classe **SandBox**, le code suivant qui est une adaptation du code de JM Doudoux à notre projet actuel.

```
Console.title("Appel à getName de la classe Class via le mot clé class ");
Console.message("Classe de d'un objet Class Queen.class : " +
               (Queen.class).getName());
Console.message("Classe de d'un objet Class Worker.class : " +
               (Worker.class).getName());
Console.message("Classe de d'un objet Class Drone.class : " +
               (Drone.class).getName());
```

c. Exécutez pour illustrer et bien comprendre l'utilité d'un appel à la méthode **getName** de la classe **Class** 😊

d. Ensuite, implémentez et exécutez le code suivant en complétant les "<...à vous de compléter...>" par ce qu'il vous semble correct 😊

```
Console.title("Appel à getName de la classe <...à vous de compléter...> ");
Console.message("<...à vous de compléter...> de l'objet melli : " +
               melli.getName());
Console.message("<...à vous de compléter...> de l'objet may : " +
               may.getName());
Console.message("<...à vous de compléter...> de l'objet will : " +
               will.getName());
```

⇒ Dans ce dernier jeu d'essai, la méthode **getName** exécutée est celle de la classe

➔ Si vous voulez en savoir plus ..., rendez-vous dans la javadoc officielle de votre JDK :

- Recherchez la classe **Object** et consultez le début de sa documentation, ainsi que les rubriques **Method Summary** et **Method Detail** relatives à la documentation de sa méthode **getClass**
- Recherchez la classe **Class** et consultez le début de sa documentation, ainsi que les rubriques **Method Summary** et **Method Detail** relatives à la documentation de sa méthode **getName**

3. Expérimentation de l'opérateur **instanceof**

- a. Implémentez à la fin de la méthode **main** de la classe **SandBox**, le code suivant :

```
Console.title("Opérateur instanceof ");
if (melli instanceof HoneyBee)
    Console.message("melli est une instance de HoneyBee");
if (melli instanceof Queen)
    Console.message("melli est une instance de Queen");
if (melli instanceof Worker)
    Console.message("melli est une instance de Worker");
if (melli instanceof Drone)
    Console.message("melli est une instance de Drone");
```

- b. Exécutez ce code et expliquez ce résultat 😊
- c. D'après ce résultat les instructions compilent-elles ?

Queen firstQueen = (Queen) melli ;

☐ Compile ☐ Ne compile pas

Queen nextQueen = (Worker) may ;

☐ Compile ☐ Ne compile pas

Pour vérifier vos réponses, pas besoin de votre enseignant de TP, il vous suffit d'implémenter chaque instruction dans l'IDE !

... Pour continuer, veuillez bien à avoir un programme qui compile ...

- d. Terminez par jeter un petit coup d'œil à la javadoc.
L'opérateur **instanceof** est présenté dans la rubrique **Pattern Matching for instanceof** à laquelle vous pouvez accéder via le lien suivant
<https://docs.oracle.com/en/java/javase/17/language/pattern-matching-instanceof-operator.html>

Maintenant que vous avez bien compris l'utilité de la méthode **getClass**,
il n'y a plus qu'à mettre en pratique l'opérateur **instanceof** dans le dernier exercice 😊

Exercice 6 : Dénombrer une collection d'abeilles

Revenir dans la méthode **collectionOfHoneyBees** de la classe HoneyBeeMain.

A la fin de cette méthode, implémentez un petit bout de code qui vous permettra de savoir de quelle sorte d'abeilles votre collection est constituée. Pour cela, il vous suffit juste de dénombrer le nombre de reine(s), d'ouvrière(s) et de faux-bourdon(s) dans la collection afin d'obtenir à l'exécution de la méthode **collectionOfHoneyBees** un affichage similaire au suivant (aux noms près des abeilles selon votre créativité 😊)

```
-----  
Collection of honeybees  
-----  
Mellifera I'm a HoneyBee!I'm a Queen, any questions?  
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.  
-----  
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.  
-----  
-----  
My collection has 6 honeybees  
-----  
-> 1 queen(s)  
-> 3 worker(s)  
-> 2 drone(s)
```

Pour terminer, si vous ne l'avez pas fait régulièrement,
passez tout le code de votre projet à **SonarLint** et corrigez les issues proposées
uniquement si vous jugez que c'est nécessaire 😊

Pour les plus rapides (facultatif) :

Si vous le souhaitez, vous pouvez continuer à travailler sur des problèmes de Conception et Programmation Objet autour des abeilles. Rendez-vous sur le dépôt github

(<https://github.com/iblasquez/enseignement-but1-developpement>) , puis allez dans la rubrique

Enoncés de TP puis **pour les plus rapides (facultatif)** : vous trouverez un nouvel énoncé **Simulation de la vie d'une ruche** à télécharger.

Annexe : Un peu de documentation à propos des Collections

❑ 1. Le JDK propose d'ailleurs dès sa version Standard : **The Collections Framework**

donc la javadoc est consultable en ligne à l'adresse suivante (par exemple) :

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/doc-files/coll-index.html>

⇒ Depuis cette adresse, pour avoir une présentation rapide de ce framework, il faut cliquer sur **Overview** 🗉

❑ 2. Vous devez retenir que **Collection** est l'interface *racine* de toutes les collections java (comme indiqué dans la javadoc dont est tiré l'extrait suivant

(<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html>)

```
public interface Collection<E>
    extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like *Set* and *List*. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

❑ 3. Cet extrait indique également que toutes les collections proposent (et donc implémentent) au minimum les méthodes présentes dans cette interface (dont la fameuse méthode **add**)

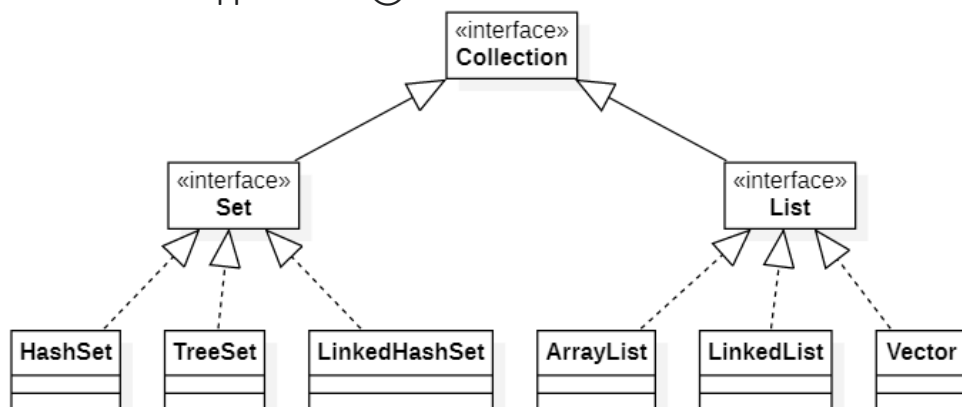
All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method	Description	
boolean	add (E e)	Ensures that this collection contains the specified element (optional operation).	
boolean	addAll (Collection<? extends E> c)	Adds all of the elements in the specified collection to this collection (optional operation).	
void	clear ()	Removes all of the elements from this collection (optional operation).	
boolean	contains (Object o)	Returns true if this collection contains the specified element.	
boolean	containsAll (Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.	
boolean	equals (Object o)	Compares the specified object with this collection for equality.	
int	hashCode ()	Returns the hash code value for this collection.	
boolean	isEmpty ()	Returns true if this collection contains no elements.	
Iterator<E>	iterator ()	Returns an iterator over the elements in this collection.	
default Stream<E>	parallelStream ()	Returns a possibly parallel Stream with this collection as its source.	
boolean	remove (Object o)	Removes a single instance of the specified element from this collection, if it is present (optional operation).	
boolean	removeAll (Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).	
default boolean	removeIf (Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.	
boolean	retainAll (Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).	
int	size ()	Returns the number of elements in this collection.	
default Spliterator<E>	spliterator ()	Creates a Spliterator over the elements in this collection.	
default Stream<E>	stream ()	Returns a sequential Stream with this collection as its source.	
Object[]	toArray ()	Returns an array containing all of the elements in this collection.	
default <T> T[]	toArray (IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.	
<T> T[]	toArray (T[] a)	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.	
Methods declared in interface java.lang.Iterable			
forEach			

❑ 4. La hiérarchie des collections proposées par le **Collections Framework** sont présentées dans la rubrique **Overview** de la javadoc via le tableau suivant :

General purpose implementations					
Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue, Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Dans cette hiérarchie, on trouve des **interfaces** et des **classes concrètes** (les classes cliquables en bleu dans le tableau) que nous utiliserons dans nos applications 😊.

Le diagramme de classes ci-contre représente
un tout petit extrait des nombreuses interfaces et classes disponibles dans le **framework Collections** du JDK.

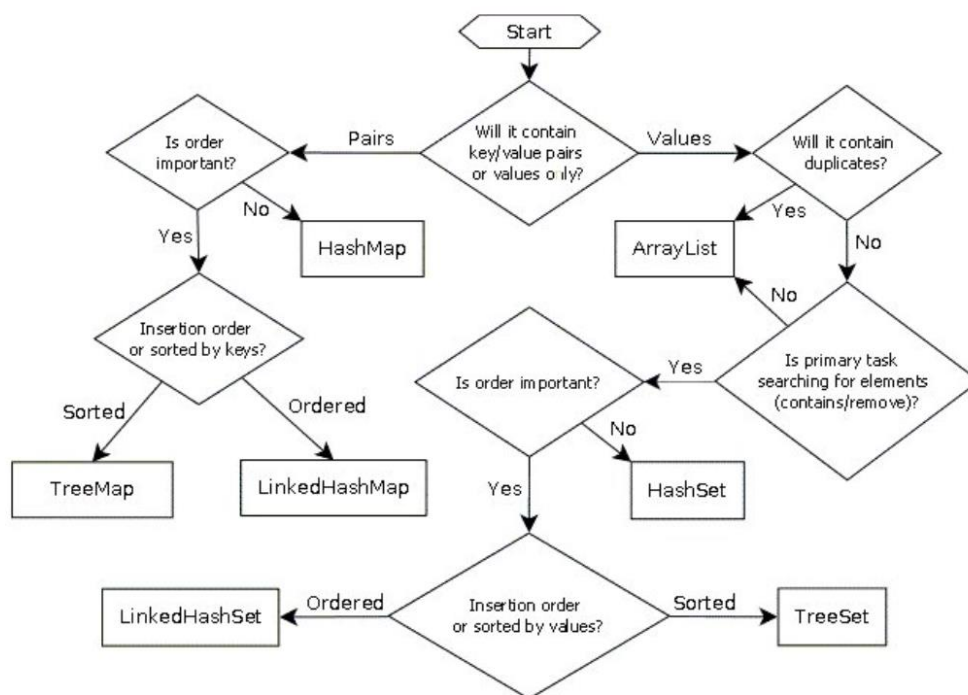


❑ 5. Pour savoir quelle collection concrète choisir dans votre application...

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, ...)

How to Choose Which Collection class to Use in Java



(Extrait de https://twitter.com/javarevisited/status/1495989911744573443?s=20&t=eTN0SRC808bqWU1nN_XVg)

❏ 6. Comment instancier et manipuler une collection ?

Le site <https://www.baeldung.com/java-collections> propose un ensemble de guides et d'exemples pour expliquer 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));
```

