**Qu'est-ce qu'une annotation Java ?**

Une annotation est un marqueur particulier (introduit par un caractère @) permettant d'ajouter une méta-données à un élément de votre programme. On peut ajouter une annotation à un type (classe, interface, type énuméré, ...), un attribut, une méthode ou encore un constructeur. En soi, une annotation ne fait pas grand-chose : il faut qu'un framework (ou une librairie) recherche ces annotations pour les traiter.

la différence entre une librairie ou un framework réside dans le fait qu'un framework (littéralement, un cadre de travail) impose une manière structurante de développer. Ce n'est pas le cas d'une librairie. En termes de frameworks Java, on peut notamment parler de Spring Framework, d'Hibernate, de JUnit ou encore de JSF : dans ces trois cas, ces frameworks utilisent des annotations.

Les annotations Java étant des méta-données, pour les rechercher et les manipuler, on doit utiliser le moteur de réflexion Java. Il existe cependant quelques annotations qui n'ont de sens que pour la compréhension du code source, par exemple l'annotation @Override. Ce type d'annotations est non manipulable par le moteur de réflexion.

**Annotations relatives à la compréhension du code**

On dit d'une telle annotation qu'elle a une rétention de type « Source ». Dit autrement, l'annotation n'est pas stockée dans le .class produit par le compilateur.

Il existe quelques annotations de ce type. La plus connue est certainement java.lang.@Override. Elle permet d'indiquer au développeur relisant le code qu'une méthode réalise une redéfinition : il existe donc une méthode de signature équivalente sur un type parent (classe héritée ou interface implémentée). Le cas le plus classique d'utilisation de cette annotation consiste en la redéfinition de la méthode toString. En voici un exemple, pour rappel.

The**@Override** annotation is one of a default Java annotation and it can be introduced in **Java 1.5** Version. The **@Override** annotation indicates that the **child class method is over-writing its base class method**.

The **@Override**annotation can be useful for two reasons

* It extracts a warning from the compiler if the annotated method doesn't actually override anything.
* It can improve the readability of the source code.

n'oubliez pas qu'il est possible de générer automatiquement une version initiale du toString en utilisant les assistants de votre IDE. Sous Eclipse, placez-vous à l'endroit ou vous souhaitez générer la méthode et saisissez-y le mot toString. Appuyez ensuite sur la séquence de touche CTRL+ESPACE. La méthode doit être générée automatiquement et vous devriez y retrouver l'annotation @Override.

Une autre annotation n'ayant du sens qu'au sein du code source est l'annotation java.lang.SuppressWarnings. Elle permet d'indiquer au compilateur ou à votre IDE de ne pas tenir compte de certains avertissements (warnings). Voici un exemple d'utilisation de cette annotation pour demander à ne pas signaler les éléments de code non utilisés.

## Annotations propagées dans le code binaire

Pour les autres annotations, il faut savoir qu'elles sont poussées dans le fichier .class produit par le compilateur. Il est donc possible de les manipuler au « runtime » (durant l'exécution de votre programme) via le moteur de réflexion Java. C'est le cas de l'annotation java.lang.Deprecated. Voici un exemple de code listant l'ensemble de méthodes dépréciées portées par la classe.

## La JSR 250 et le package javax.annotations

La [JSR 250](https://jcp.org/en/jsr/detail?id=250) définit un ensemble d'annotations communes (Common Annotations) pour la plate-forme Java. Voici un descriptif de ces annotations.

* **@Generated** : cette annotation est utilisée pour indiquer qu'un code a été produit par un générateur. Bien entendu, elle est de rétention source et peut être utilisées sur tous les éléments de code (classes, interfaces, méthodes, constructeurs, ...). Malgré son existence, rares sont les générateurs ajoutant cette annotation dans le code qu'ils produisent. Voici un exemple d'utilisation.
* **@PostConstruct** : permet d'indiquer qu'une méthode est à exécuter après sa phase initialisation (constructeur + setters). Cette annotation doit, bien entendu, être accessible au runtime pour être trouvée par le système qui va coordonner les appels aux méthodes associées. Cette annotation est intimement liée à l'annotation @PreDestroy.
* Attention le simple fait d'apposer cette annotation sur une méthode ne forcera pas un appel implicite. Il faut absolument qu'un framework associé à ces annotations les recherche et réalise les appels : c'est notamment de cas de la plate-forme Jakarta EE (ex Java EE) ou de Spring Framework.
* **@PreDestroy** : permet d'indiquer qu'une méthode est à exécuter avant la libération de l'objet. Cette annotation doit, bien entendu, être accessible au runtime pour être trouvée par le système qui va coordonner les appels aux méthodes associées. Cette annotation est intimement liée à l'annotation @PostConstruct. Consultez l'extrait de code précédent pour en voir un exemple d'utilisation.
* Attention : le simple fait d'apposer cette annotation sur une méthode ne forcera pas un appel implicite. Il faut absolument qu'un framework associé à ces annotations les recherche et réalise les appels : c'est notamment de cas de la plate-forme Jakarta EE (ex Java EE) ou de Spring Framework.
* **@Resource** : permet d'associer à un attribut ou à un setter une JNDI. JNDI (Java Naming and Directory Interface) est une API permettant d'accéder à des ressources stockées dans une base de données hiérarchique. Les serveurs d'applications Java EE propose, notamment, une base de données de ce type : dans ce cas, le serveur aura la responsabilité de retrouver ces annotations et d'injecter les instances de ressources dans les beans considérés. Voici un exemple d'utilisation.