EPSI I4 Web Services / **David Gayerie** - 2013-2014

NOTIONS COMPLÉMENTAIRES EN JAVA

COMPILATION

La compilation du code source ne produit pas un binaire mais du *bytecode*

Le fichier source Hello.java

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("hello the world!");
  }
}
```

Le fichier Hello.class désassemblé avec javap

```
public class Hello {
  public static void main(java.lang.String[]);
    Signature: ([Ljava/lang/String;)V
    Code:
       0: getstatic     #16
       3: ldc           #22
       5: invokevirtual #24
       8: return
}
```

Le bytecode est exécutable dans une machine virtuelle Java (JVM)

Le fichier source Hello.java

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("hello the world!");
  }
}
```

Compilation du fichier Hello.java

```
$ javac Hello.java
```

Exécution de la classe Hello

```
$ java Hello
hello the world!
```

Conséquence : Un programme est portable sur un système disposant d'une machine virtuelle Java.

> *Compile once, run anywhere*

ÉDITION DE LIEN

Java utilise un système de **lien dynamique à l'exécution** (dynamic link).

Un programme Java est une collection de fichiers .class parfois rassemblés dans une archive (i.e. fichier .jar).

L'édition de lien se fait au chargement d'un fichier .class dans la JVM. Le ***ClassLoader*** a la responsabilité de charger les classes manquantes.

Le *ClassLoader* est accessible programmatiquement.

Étant donné deux classes `Spaceship` et `Engine`

Spaceship.java

```
public class Spaceship {
  private Engine engine = new Engine();

  public void takeoff() {
    engine.startup();
  }
}
```

Engine.java

```
public class Engine {
  public void startup() {
    // ...
  }
}
```

Une référence à la classe `Spaceship` dans un programme entraînera son chargement par le *ClassLoader*. La classe `Engine` sera également chargée car elle est nécessaire au fonctionnement de `Spaceship`.

Si une classe ne peut pas être trouvée, on obtient l'erreur `NoClassDefFoundError` **à l'exécution**.

Erreur à l'exécution lorsque la classe `Engine` n'est pas trouvable

```
Exception in thread "main" java.lang.NoClassDefFoundError: Engine
        at Spaceship.<init>(Spaceship.java:2)
Caused by: java.lang.ClassNotFoundException: Engine
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        ... 2 more
```

Le *ClassLoader* par défaut de la JVM charge les classes à partir des fichiers .class.

L'ensemble des répertoires et des archives contenant les *packages* et les fichiers .class est appelé le **classpath**.

*Lancement d'un programme sous *NIX en spécifiant le classpath*

```
java -classpath .:spaceship/bin:lib/galaxy.jar SpaceBattle
```

Lancement d'un programme sous Windows en spécifiant le classpath

```
java -classpath .;spaceship/bin;lib/galaxy.jar SpaceBattle
```

A noter : le séparateur pour le paramètre classpath est différent selon les OS ':' sous *NIX et ';' sous Windows.

LES PAQUETS (PACKAGES)

Afin d'éviter la collision de nom (des classes, interfaces, énumérations ou annotations portant le même nom), Java supporte la notion de paquets (packages).

Un package est simplement un répertoire dans lequel sont placés les fichiers Java.

Un package définit un espace de nom commun à l'ensemble des éléments de ce package.

Le package d'un fichier source doit être la première instruction du fichier. Il est spécifié avec le mot-clé **package**.

```
package com.battleship;

public class Spaceship {

  // ...

}
```

Le fichier source doit se trouver dans le répertoire

```
com/battleship
```

Pour référencer la classe `Spaceship` dans un autre fichier source, on peut :

- Soit utiliser son nom complet (c'est-à-dire en incluant le package) :

```
com.battleship.Spaceship spaceship = new com.battleship.Spaceship();
```

- Soit inclure la classe dans l'espace de noms courant avec l'instruction **import** en début de fichier :

```
import com.battleship.Spaceship;
```

  On peut alors écrire

```
Spaceship spaceship = new Spaceship();
```

La mot-clé **import** permet aussi d'inclure la totalité des éléments d'un package en utilisant * :

```
import com.battleship.*;
```

Il n'est pas nécessaire d'importer un élément qui se trouve dans le même package.

Les éléments appartenant au package **java.lang** (comme la classe `String` par exemple) sont implicitement importés.

Par convention, le nom des packages s'écrit en minuscules.

Les packages commençant par **java** ou **javax** sont réservés.

Il est également possible d'importer les éléments **static** d'une classe.

Cela permet d'invoquer des méthodes ou de référencer les attributs sans avoir à les préfixer par le nom de la classe.

```
class Angle {
  private double radianAngle;

  public Angle(double radianAngle) {
    this.radianAngle = Math.max(0d, radianAngle);
    this.radianAngle = Math.min(2 * Math.PI, this.radianAngle);
  }
}
```

```
import static java.lang.Math.*;

class Angle {
  private double radianAngle;

  public Angle(double radianAngle) {
    this.radianAngle = max(0d, radianAngle);
    this.radianAngle = min(2 * PI, this.radianAngle);
  }
}
```

Un package peut contenir un fichier spécial : **package-info.java**

Ce fichier contient obligatoirement l'instruction **package** comme pour un fichier normal.

De plus, il peut contenir la documentation du package mais également des annotations de niveau package.

Exemple d'un fichier package-info.java

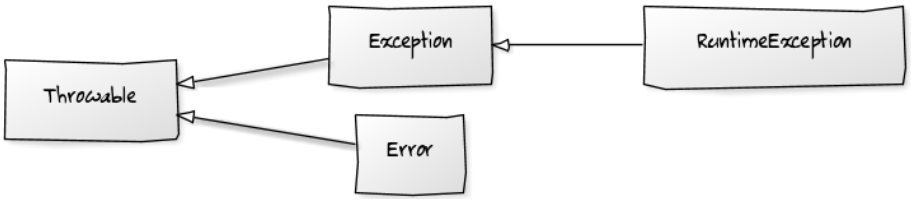
```
/**
 * La javadoc du package vient ici...
 */
@javax.xml.bind.annotation.XmlSchema
(namespace = "http://www.epsi.fr/i4-web-services/myfirstwebservice")
package fr.epsi.i4_web_services.myfirstwebservice;
```

LES EXCEPTIONS

Une exception permet d'interrompre un traitement lorsqu'une situation exceptionnelle se produit.
Une exception est propagée dans la pile des appels de méthodes. Elle peut être interceptée progammatiquement.
Si elle n'est interceptée par aucune méthode, elle remonte la pile d'appel et interrompt le thread d'exécution.
S'il s'agit du thread principal, le programme s'interrompt en erreur.

Les exceptions sont représentées par des classes qui ont comme ancêtre **java.lang.Throwable**

La hiérarchie de base des exceptions



Les classes héritant de **java.lang.Error** sont réservées pour la JVM afin de signaler des erreurs systèmes :

- NoClassDefFoundError
- OutOfMemoryError
- StackOverflowError
- ...

Pour lancer une exception on utilise le mot-clé **throw**.

```
// La classe hérite de exception qui hérite elle-même de throwable.
// Par convention le nom de la classe se termine par Exception.
public class EndOfTheWorldException extends Exception {
}
```

On peut maintenant signaler l'événement exceptionnel de la fin du monde en lançant l'exception :

```
throw new EndOfTheWorldException();
```

On distingue deux catégories d'exception :

- Les *checked exceptions* apparaissent dans la signature des méthodes qui peuvent les lancer ou les propager lors d'un appel grâce au mot-clé **throws**.

```
public void pressTheRedButton() throws EndOfTheWorldException {
  if(this.isTheEndOfTheWorld()) {
    throw new EndOfTheWorldException();
  }
  // ...
}

public void doSomethingSilly() throws EndOfTheWorldException {
  pressTheRedButton();
}
```

  Plusieurs *checked exceptions* peuvent être spécifiées en les séparant par une virgule :

```
public Stuff buy(long amount, Currency currency)
  throws NotEnoughMoneyException, NotAcceptedCurrencyException,
         StuffUnavailableException {
  // ...
}
```

- Les *unchecked exceptions* héritent directement ou indirectement de **RuntimeException** ou de **Error**.
  La déclaration de ces exceptions est **optionnelle** dans la signature des méthodes pouvant les lancer ou les propager.

```
public class EntityNotFoundException extends RuntimeException {
  public EntityNotFoundException(String message) {
    super(message);
  }
}
```

```
// Il n'est pas nécessaire d'ajouter :
//       throws EntityNotFoundException
// car EntityNotFoundException hérite de RuntimeException.
public Entity get(String id) {
  Entity entity = database.load(id);
  if (entity == null) {
    throw new EntityNotFoundException("Cannot find entity " + id);
  }
  return entity;
}
```

Une exception peut être interceptée par une méthode dans la pile d'appel :

```
try {
  evilMind.pressTheRedButton();
}
catch (EndOfTheWorldException ex) {
  hero.saveTheWorld();
}
```

Le block défini par le mot-clé **catch** est exécuté **uniquement** si une exception de type **EndOfTheWorldException** est lancée lors de l'exécution du block défini par le mot-clé **try**.

Un block défini par le mot-clé **finally** est exécuté systématiquement après un block **try** et un block **catch** (si une exception a été attrapée).

```
try {
  evilMind.pressTheRedButton();
}
catch (EndOfTheWorldException ex) {
  hero.saveTheWorld();
}
finally {
  // Finalement, le héros arrête l'esprit du mal qu'il ait eu
  // ou non à sauver le monde.
  hero.arrest(evilMind);
}
```

Un block **finally** est souvent utilisé en Java pour libérer des ressources système à la fin d'un traitement.

```
public String getFileContent(String filename) throws java.io.IOException {
  java.io.FileReader reader = new java.io.FileReader(filename);
  try {
    int nbCharRead = 0;
    char[] buffer = new char[1024];
    StringBuilder builder = new StringBuilder();
    // L'appel à reader.read peut lancer une java.io.IOException
    while ((nbCharRead = reader.read(buffer)) >= 0) {
      builder.append(buffer, 0, nbCharRead);
    }
    // le retour explicite n'empêche pas l'exécution du block finally.
    return builder.toString();
  }
  // Ce block est obligatoirement exécuté après le block try.
  // Ainsi le flux de lecture sur le fichier est fermé
  // avant le retour de la méthode.
  finally {
    reader.close();
  }
}
```

Depuis Java 7, on peut utiliser une expression ***try-with-resources*** ***(http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html)*** . Pour gérer le cas particulier des resources que l'on souhaite fermer automatiquement.

```
public String getFileContent(String filename) throws java.io.IOException {
  // La resource que l'on souhaite fermer automatiquement et
  // déclarer après le mot-clé try entre parenthèses.
  try (java.io.FileReader reader = new java.io.FileReader(filename)) {
    int nbCharRead = 0;
    char[] buffer = new char[1024];
    StringBuilder builder = new StringBuilder();
    while ((nbCharRead = reader.read(buffer)) >= 0) {
      builder.append(buffer, 0, nbCharRead);
    }
    return builder.toString();
  }
  // On évite l'utilisation du bloc finally de l'exemple
  // précédent. Le try-with-resource garantit que reader.close()
  // est appelé.
}
```

Seules les classes implémentant l'interface **java.lang.AutoCloseable (http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html)** ou **java.io.Closeable (http://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html)** peuvent être utilisées dans une expression *try-with-resources*.

LES INTERFACES

Une interface permet d'établir un contrat implémenté par des classes. L'interface ne fournit pas d'implémentation mais uniquement la signature des méthodes à implémenter.

Une interface peut déclarer des constantes souvent utilisées comme valeurs remarquables de retour ou de paramètres.

Un exemple d'interface

```
public interface Playable {
  int FIRST_STREAM = 0;
  int LAST_STREAM = Integer.MAX_VALUE;

  void play() throws PlayerNotReadyException, StreamInterruptedException;
  void stop();
  // on peut passer le numéro du flux à jouer
  // ou FIRST_STREAM ou LAST_STREAM
  boolean selectStream(int streamNum);
}
```

Les attributs déclarés dans une interface sont implicitement **public static final**. Ce sont donc bien des constantes.

Les méthodes déclarées dans une interface sont implicitement **public abstract**.

Une interface explicitant tous les mots-clés

```
public interface Playable {

  // les mots-clés public static final peuvent être explicités.
  public static final int FIRST_STREAM = 0;
  public static final int LAST_STREAM = Integer.MAX_VALUE;

  // les mots-clés public abstract peuvent être explicités.
  public abstract void play()
           throws PlayerNotReadyException, StreamInterruptedException;
  public abstract void stop();
  public abstract boolean selectStream(int streamNum);
}
```

Une classe **concrète** héritant d'une interface doit fournir une implémentation de toutes les méthodes de l'interface.

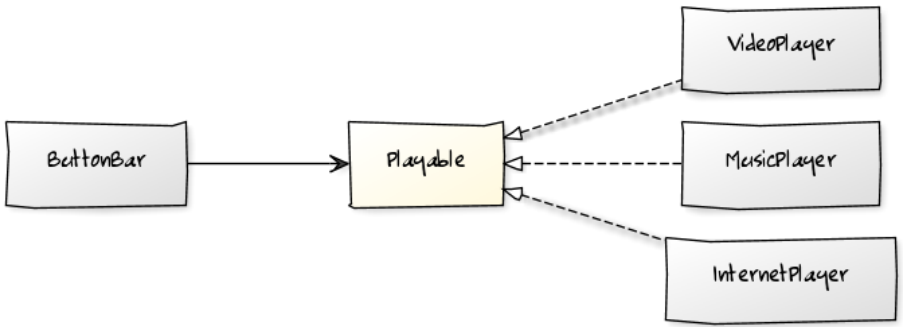
Une classe implémentant l'interface Playable

```
public class VideoPlayer implements Playable {

  public void play() {
    // implementation here
  }

  public void stop() {
    // implementation here
  }

  public boolean selectStream(int streamNum) {
    // implementation here
  }
}
```

Les règles à respecter dans la signature des méthodes de la classe sont les mêmes que pour la redéfinition de méthode (overriding).

Une interface permet d'introduire un *couplage faible* entre les éléments d'un système.

Un exemple de l'utilisation de l'interface Playable



Si la classe `ButtonBar` permet d'afficher graphiquement des boutons "play" et "stop", elle permet à l'utilisateur de contrôler n'importe quelle classe implémentant `Playable`.

Une interface peut hériter **d'une ou de plusieurs** autres interfaces.

L'héritage d'interface

```
public interface Navigable {
  void next();
  void previous();
}
```

```
public interface Playable {
  void play() throws PlayerNotReadyException, StreamInterruptedException;
  void stop();
}
```

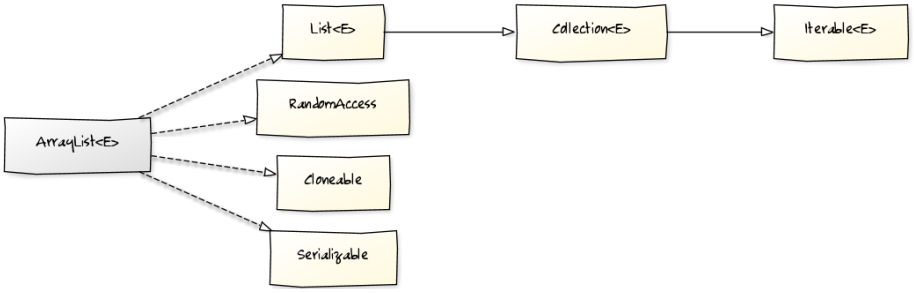
```
// Une classe implémentant l'interface Player doit
// fournir une implémentation des méthodes déclarées
// par Player, Navigable et Playable.
public interface Player extends Navigable, Playable {
  int getSelectedStream();
}
```

Certaines interfaces ne définissent aucune méthode. On parle alors d'**interfaces marqueur**. Elles indiquent que les classes qui les implémentent ont *certaines propriétés*.

L'API Java standard propose des interfaces marqueurs :

- **java.lang.Cloneable (http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)**
- **java.io.Serializable (http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)** (nous y reviendrons)
- ...

Les interfaces implémentées par la classe java.util.ArrayList



LA SÉRIALISATION

La sérialisation est un mécanisme permettant d'écrire ou de lire dans un flux les données binaires d'un objet ou d'une hiérarchie d'objets.

La sérialisation permet, par exemple, de sauvegarder dans un fichier ou de transmettre à travers un réseau, la représentation d'un objet.

Une classe est sérialisable si elle-même ou une classe parente implémente l'interface marqueur **java.io.Serializable (http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)** .

Une classe sérialisable

```
package epsi;

import java.io.Serializable;

public class Student implements Serializable {

  private static final long serialVersionUID = 1L;

  private final String name;
  private final int age;

  public Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String toString() {
    return "Hello, my name is " + name +
           " and I am " + age + " years old.";
  }
}
```

Si un objet contient un attribut non nul pointant sur un objet non sérialisable, la sérialisation échoue avec l'exception *java.io.NotSerializableException*.

Cependant, la serialisation ignore :

- Les attributs de classe (**static**)
- Les attributs déclarés avec le mot-clé **transient**
- Les attributs des classes parentes n'implémentant pas java.io.Serializable

Il existe un attribut jouant un rôle spécial :
serialVersionUID.

```
private static final long serialVersionUID = -2939239776154334018L;
```

Cet attribut permet de versionner la définition d'une classe lors de la sérialisation. Pendant la désérialisation, la version des données sérialisées est comparée à la version de la classe Java chargée par la JVM. Si les version diffèrent, la désérialisation se termine avec l'exception
java.io.InvalidClassException (http://docs.oracle.com/javase/7/docs/api/java/io/InvalidClassException.html) .

Pour sérialiser un objet, on utilise la classe **java.io.ObjectOutputStream (http://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html)** qui propose la méthode writeObject(Object).

```
Student student = new Student("John Doe", 24);
try (ObjectOutputStream oos = new ObjectOutputStream(outputStream)) {
  oos.writeObject(student);
}
```

Pour désérialiser un objet, on utilise la classe **java.io.ObjectInputStream (http://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html)** qui propose la méthode readObject().

```
Student student = null;
try (ObjectInputStream ois = new ObjectInputStream(inputStream)) {
  student = Student.class.cast(ois.readObject());
}
```

LA SYNCHRONISATION

Lorsqu'un programme s'exécute dans un environnement multi-thread, l'accès à des ressources partagées (mémoire, fichier, ...) devient problématique.

Comment garantir que les ressources demeurent cohérentes si plusieurs threads y accèdent en lecture et en écriture ?

Une classe "sensible" dans un environnement multi-thread

```
/**
 * Un modèle de parking de voitures.
 */
public class CarPark {

  /**
   * @Return un ticket de parking pour chaque véhicule entrant
   */
  public Ticket park() throws NoPlaceAvailableException {
    // ...
  }

  /**
   * Enregistre la sortie de parking d'un véhicule selon son ticket
   */
  public void unpark(Ticket ticket) {
    // ...
  }

  /**
   * @Return le nombre de places disponibles
   */
  public int getAvailablePlaces() {
    // ...
  }
}
```

Dans un environnement serveur, l'accès à des ressources partagées est une problématique permanente car un serveur "évolué" s'exécute toujours dans un environnement multi-thread.

Il est possible en Java de synchroniser l'exécution des méthodes, c'est-à-dire de garantir qu'un seul thread à la fois peut exécuter du code synchronisé. On utilise pour cela le mot-clé **synchronized**.

Un exemple de méthodes synchronisées

```
public class CarPark {

  public synchronized Ticket park() throws NoPlaceAvailableException {
    // ...
  }

  public synchronized void unpark(Ticket ticket) {
    // ...
  }

  public synchronized int getAvailablePlaces() {
    // ...
  }
}
```

Il est également possible de ne synchroniser qu'une portion de code. On fournit alors une instance d'objet non nulle qui sert de verrou.

Un exemple de bloc de code synchronisé avec un verrou

```
  public void doSomethingInConcurrency(Resource sharedResource)
    synchronized(sharedResource) {
      sharedResource.setModificationDate(Calendar.getInstance());
      // ...
    }
  }
```

A l'entrée du block synchronisé, un thread acquière le verrou (dans l'exemple : *sharedResource*). Tout autre thread voulant acquérir le verrou est bloqué jusqu'à ce que le verrou soit relâché automatiquement par la sortie du bloc d'exécution synchronisé.

Ainsi une méthode synchronisée est implicitement un bloc synchronisé sur l'instance de la classe (this).

```
public class CarPark {

  public Ticket park() throws NoPlaceAvailableException {
    synchronized(this) {
      // ...
    }
  }

  public void unpark(Ticket ticket) {
    synchronized(this) {
      // ...
    }
  }

  public int getAvailablePlaces() {
    synchronized(this) {
      // ...
    }
  }
}
```

La synchronisation est une des solutions proposées par Java pour le partage de ressources dans la programmation concurrente.

Cette solution a plusieurs défauts majeurs : création de goulets d'étranglement, surcoût de cycle CPU, mauvaise scalabilité...

La gestion de ressources partagées dans un environnement concurrent est **extrêmement** délicate !

La meilleure solution consiste à **limiter** les ressources partagées entre threads et à en laisser la responsabilité à des composants logiciel fiables (comme les gestionnaires de transaction, les conteneurs Java EE,...).

LES ANNOTATIONS

Les annotations en Java sont des marqueurs qui permettent d'ajouter des méta-données aux classes, aux méthodes, aux attributs, aux paramètres, aux variables, aux paquets ou aux annotations elles-mêmes.

Les annotations sont utilisées dans des domaines divers. Leur intérêt principal est de fournir une *méta-information* qui pourra être exploitée par un programme.

Une annotation est un type (comme une classe ou une interface) du langage Java : elle peut être référencée par son nom complet ou importée depuis un autre paquet grâce au mot-clé **import**.

Une annotation n'est pas instanciée, elle est simplement accolée à l'élément qu'elle vient enrichir :

```
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlID;
import javax.xml.bind.annotation.XmlElement;

// Une annotation de classe
public @XmlRootElement class Spaceship {

  // Plusieurs annotations sur un attribut
  private @XmlAttribute @XmlID String id;

  // Une autre annotation sur un attribut
  private @XmlElement Engine engine;

  // ...
}
```

Certaines annotations déclarent des attributs (par exemple l'annotation **javax.xml.bind.annotation.XmlElement (http://docs.oracle.com/javaee/7/api/javax/xml/bind/annotation/XmlElement.html)**)

```
  @XmlElement(name="moteur", required=true)
  private Engine engine;
```

Par convention, si un attribut de l'annotation s'appelle **value** et qu'il est le seul paramètre spécifié, alors son nom peut être omis pour plus de lisibilité.

```
  @SuppressWarnings("deprecation")
  public void myMethod() {
    //...
  }

  @SuppressWarnings(value="deprecation")
  public void myOtherMethod() {
    //...
  }
```

Un nombre très limité d'annotations sont exploitées directement par le compilateur.

On trouve les annotations déclarées dans le paquet **java.lang** :

- **Deprecated (http://docs.oracle.com/javase/7/docs/api/java/lang/Deprecated.html)**
- **Override (http://docs.oracle.com/javase/7/docs/api/java/lang/Override.html)**
- **SuppressWarnings (http://docs.oracle.com/javase/7/docs/api/java/lang/SuppressWarnings.html)**
- **SafeVarargs (http://docs.oracle.com/javase/7/docs/api/java/lang/SafeVarargs.html)** (introduite en Java 7)

Une annotation est définie par sa **rétention (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Retention.html)** , c'est-à-dire la façon dont une annotation sera conservée. Les différentes rétentions d'annotation sont :

- **source** : l'annotation est accessible durant la compilation mais n'est pas intégrée dans le fichier class généré.
- **class** : l'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré mais elle n'est pas chargée dans la JVM à l'exécution.
- **runtime** : l'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré et elle est chargée dans la JVM à l'exécution. Elle est accessible par introspection.

Pour les annotations de rétention **source**, la JVM prévoit un mécanisme pour inclure lors de la compilation des **processeurs d'annotations (http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html)** . Par exemple, ces processeurs peuvent servir à vérifier le code ou générer du code supplémentaire.

Lombok (http://projectlombok.org/) est un exemple de projet open-source fournissant des annotations permettant de générer du code au moment de la compilation *via* un processeur d'annotations.

Une déclaration d'annotation

```
public @interface MyAnnotation {
}
```

Une déclaration d'annotation avec attributs

```
public @interface MyAnnotation {
  String  name();
  boolean isOk();
  int[]   range() default {1, 2, 3};
}
```

Une annotation implémente implicitement **Annotation (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Annotation.html)** et rien d'autre !

Les attributs d'une annotation peuvent être uniquement :

- Un type primitif,
- une chaîne de caractères (java.lang.String),
- une référence de classe (java.lang.Class),
- une Annotation (java.lang.annotation.Annotation),
- un type énuméré (enum),
- un tableau à une dimension d'un de ces types.

Le mot-clé **default** permet de spécifier une valeur d'attribut par défaut.

Une annotation peut elle-même être annotée. Les annotations d'annotation fournies par Java 7 sont :

- **Documented (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Documented.html)**
- **Inherited (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Inherited.html)**
- **Retention (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Retention.html)**
- **Target (http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Target.html)**

Un exemple d'annotation pour un framework xUnit

```
package epsi.test;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Désigne une méthode comme étant un test unitaire.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
public @interface Test {
  String description();
}
```