# Packages

## Usage

Il est vivement conseillé de regrouper ses classes par thème. Un tel **regroupement** sera appelé package. Pour ce faire, plusieurs règles sont à respecter :

- La première ligne de votre programme (hormis des lignes de commentaires) doit être la déclaration de package : exemple

  ```
  package monpack;
  ```

- Le fichier source devra être placé dans une directory de même nom que le package.

## Sous-package

Les packages peuvent eux-mêmes contenir des sous-packages.Dans ce cas la première ligne sera du type :

```
package monpack.monsouspack;
```

Le fichier source devra être placé dans une sous directory nommée `monsouspack` de la directory `monpack`.

Ce n'est qu'à ces conditions que Java s'y retrouvera et pourra compiler votre source. Si par exemple, on fait :

```
javac –d classes Source.java
```

dans la directory `monsouspack` (sous directory de `monpack`) et pour autant que la première ligne de `Source.java` soit `package monpack.monsouspack;`, le compilateur créera la sous directory `monpack` de `classes` et la sous directory `monsouspack` de `monpack` et y placera `Source.class`.

## Utilité

Les packages fournissent un moyen de définir des **espaces de noms**. Il arrive en effet souvent que l'on veuille donner à plusieurs classes le même nom : une classe `Personne` peut intervenir dans plusieurs cadres, par exemple. L'utilisation de package résout ce problème. Toutefois, rien n'empêche de donner le même nom à plusieurs packages. Seule une convention permet de remédier à ce problème : Sun demande que les noms de packages commencent par le nom de domaine inversé de la société développant le package. Ainsi un package développé à l'IPL devrait commencer par :

`be.ipl.`*`nom_du_package`*

Les packages sont aussi un moyen de **protection** : ils permettent de limiter les accès aux classes, méthodes et attributs. En effet si devant le nom d'une classe, d'une méthode ou d'un attribut, on n'indique aucun modificateur d'accès (`public`, `private`,...) cette classe, cette méthode ou ce champ a un accès package. Cela signifie qu'il sera accessible dans les classes situées dans le même package.

## La clause import

Si vous désirez employer des classes situées dans un package autre que le package `java.lang` et celui où est placée votre classe en cours, vous devez importer ce package ou du moins les classes que vous employez :

```
package monpack;

import monpack.monsouspack.*;

// ou import monpack.monsouspack.Source

public class TestSource {

      ...
```

Si vous n'avez pas mis de déclaration de package dans votre classe, elle sera considérée comme faisant partie du package `<default>`. Il est impossible d'importer ce package `<default>`.

Il y a deux sortes d'imports : les imports simples :

```
import java.util.Arrays;
```

et les imports à la demande :

```
import java.util.*;
```

Dans ce dernier cas, le compilateur n'importera de ce package que les classes nécessaires.

Toutes les classes à retrouver doivent être dans le CLASSPATH sauf les classes situées dans les API fournies avec Java. Les clauses `import` servent à compléter les noms : normalement, si je ne mets pas de clause `import`, je pourrai utiliser la classe `Arrays` en indiquant chaque fois le nom complet, par exemple :

```
java.util.Arrays.sort(table);
```

Si deux classes portent le même nom dans deux packages différents et que le programmeur désire les utiliser toutes deux, le plus simple est d'utiliser les noms complets afin d'éviter toute ambiguïté.

## La clause import static

Il est possible de préciser que l'import effectué doit être `static`. Un `import static` est similaire à un `import` classique. Effectivement, l'`import` classique importe les classes d'un package en permettant l'usage sans précision du package tandis que l'`import static` importe les membres `static` des classes en permettant leur utilisation sans préciser la classe dans laquelle ils sont définis.

```
import static java.util.Collections.*;

import java.util.ArrayList;
import java.util.List;

public final class StaticImporter {

      public static void main(String... aArgs) {
```

```
            List<String> strings = new ArrayList<>();
            strings.add("bla");
            strings.add("bluu");
            strings.add("blo");

            replaceAll(strings, "bla", "ble");

      }
}
```

La méthode `replaceAll` est une propriété de `Collections`. Elle est utilisable sans précision du package lorsque l'`import static` de `Collections` est effectué.

Il faut toutefois utiliser cet import avec parcimonie. L'exemple suivant effectue deux `import static` (`Collections` et `Comparator`). Les méthodes `sort` et `replaceAll` appartiennent à `Collections` tandis que `comparing` fait partie de `Comparator`. Par contre, l'appel à la méthode `reverseOrder` n'est pas possible car cette méthode est définie dans les deux classes importées.

```
import java.util.*;
import static java.util.Collections.*;
import static java.util.Comparator.*;

public final class StaticImporter {

      public static void main(String... aArgs) {
            List<String> strings = new ArrayList<>();
            strings.add("bla");
            strings.add("bluu");
            strings.add("blo");

            replaceAll(strings, "bla", "ble");
            sort(strings, comparing(s -> s));

            /*
             * reverseOrder(); appel ambigu car cette méthode est définie dans
             * Collections et Comparator
             */

      }
}
```

L'usage abusif d'`import static` peut rendre le code illisible et non maintenable, polluant son espace de noms avec tous les membres statiques importés.

Utilisé de façon appropriée, l'`import static` peut rendre votre programme plus lisible, en supprimant de multiples répétitions de noms de classe.

Mais cet import peut également s'avérer particulièrement nuisible à la lisibilité; si vous avez besoin d'un ou deux membres, importez-les individuellement.