Programmation orientée objet

Eléments complémentaires...

Plan du cours

Plan du cours

- Introduction aux collections

Plan du cours

- Introduction aux collections
- Les exceptions

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers
- Présentation des threads

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers
- Présentation des threads
- Présentation des paquets

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers
- Présentation des threads
- Présentation des paquets
- Génération automatique de documentation

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers
- Présentation des threads
- Présentation des paquets
- Génération automatique de documentation
- La documentation officielle

Plan du cours

- Introduction aux collections
- Les exceptions
- Manipulation des fichiers
- Présentation des threads
- Présentation des paquets
- Génération automatique de documentation
- La documentation officielle

  Ce cours est un ensemble de notions diverses pouvant être utilisées en TP. Elles sont juste introduites, en cas de besoin précis une recherche sur Internet (par exemple http://www.developpez.com) sera nécessaire...

Les collections

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

- Depuis la version 1.5, Java propose un ensemble de classes (et d'interfaces) pour faciliter l'écriture de ces ensembles (avant la version 1.5, c'était possible mais plus compliqué...)

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

- Depuis la version 1.5, Java propose un ensemble de classes (et d'interfaces) pour faciliter l'écriture de ces ensembles (avant la version 1.5, c'était possible mais plus compliqué...)

- Les collections utilisent la généricité, un concept de la programmation objets qui n'a pas été vu en cours. Il n'est pas nécessaire de connaître ce concept pour pouvoir utiliser les collections.

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

- Depuis la version 1.5, Java propose un ensemble de classes (et d'interfaces) pour faciliter l'écriture de ces ensembles (avant la version 1.5, c'était possible mais plus compliqué...)

- Les collections utilisent la généricité, un concept de la programmation objets qui n'a pas été vu en cours. Il n'est pas nécessaire de connaître ce concept pour pouvoir utiliser les collections.

- La généricité fait appel à la notation < > lors des déclarations. Les quelques exemples présentés seront suffisant pour la suite du module.

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

- Depuis la version 1.5, Java propose un ensemble de classes (et d'interfaces) pour faciliter l'écriture de ces ensembles (avant la version 1.5, c'était possible mais plus compliqué...)

- Les collections utilisent la généricité, un concept de la programmation objets qui n'a pas été vu en cours. Il n'est pas nécessaire de connaître ce concept pour pouvoir utiliser les collections.

- La généricité fait appel à la notation < > lors des déclarations. Les quelques exemples présentés seront suffisant pour la suite du module.

- Les collections peuvent être parcourues comme les tableaux avec la notation `for` simplifiée :

Les collections

- Les tableaux sont souvent limités pour regrouper des objets et ils ne permettent pas de représenter des ensembles de données comme les piles, les files, les dictionnaires, ... Ces ensembles de données sont souvent regroupés sous le nom de **collections**.

- Les collections existent dans de nombreux langages, les concepts sont les mêmes, seule la mise en oeuvre diffère.

- Depuis la version 1.5, Java propose un ensemble de classes (et d'interfaces) pour faciliter l'écriture de ces ensembles (avant la version 1.5, c'était possible mais plus compliqué...)

- Les collections utilisent la généricité, un concept de la programmation objets qui n'a pas été vu en cours. Il n'est pas nécessaire de connaître ce concept pour pouvoir utiliser les collections.

- La généricité fait appel à la notation < > lors des déclarations. Les quelques exemples présentés seront suffisant pour la suite du module.

- Les collections peuvent être parcourues comme les tableaux avec la notation `for` simplifiée :

```
for (Type elt : maCollection) {
...
}
```

Les principales collections

Les principales collections

- Les collections de type `Set` (c'est une interface) assurent qu'aucun élément ne sera stocké en double. Plusieurs classes implémentent cette interface, la manière dont les données sont stockée est différentes dans chaque classe.

Les principales collections

- Les collections de type `Set` (c'est une interface) assurent qu'aucun élément ne sera stocké en double. Plusieurs classes implémentent cette interface, la manière dont les données sont stockée est différentes dans chaque classe.

- L'interface `Queue` permet de représenter des collections dans lesquelles l'ordre des éléments est important (pile LIFO, file FIFO,...).

Les principales collections

- Les collections de type `Set` (c'est une interface) assurent qu'aucun élément ne sera stocké en double. Plusieurs classes implémentent cette interface, la manière dont les données sont stockée est différentes dans chaque classe.

- L'interface `Queue` permet de représenter des collections dans lesquelles l'ordre des éléments est important (pile LIFO, file FIFO,...).

- Les classes implémentant l'interface `List` permettent de stocker des éléments en prenant en compte leur place (généralement un indice).

Les principales collections

- Les collections de type `Set` (c'est une interface) assurent qu'aucun élément ne sera stocké en double. Plusieurs classes implémentent cette interface, la manière dont les données sont stockée est différentes dans chaque classe.

- L'interface `Queue` permet de représenter des collections dans lesquelles l'ordre des éléments est important (pile LIFO, file FIFO,...).

- Les classes implémentant l'interface `List` permettent de stocker des éléments en prenant en compte leur place (généralement un indice).

- Les tables de hachage (ou tableaux associatifs) sont construit à partir de l'interface `Map`. Dans un tableau associatif chaque donnée stockée (*Value*) et représentée par un objet (*Key*). La clé (*Key*) doit être unique, par contre les données stockées (*Value*) peuvent être multiple (mais à des clés différentes).

Les principales collections

- Les collections de type `Set` (c'est une interface) assurent qu'aucun élément ne sera stocké en double. Plusieurs classes implémentent cette interface, la manière dont les données sont stockée est différentes dans chaque classe.

- L'interface `Queue` permet de représenter des collections dans lesquelles l'ordre des éléments est important (pile LIFO, file FIFO,...).

- Les classes implémentant l'interface `List` permettent de stocker des éléments en prenant en compte leur place (généralement un indice).

- Les tables de hachage (ou tableaux associatifs) sont construit à partir de l'interface `Map`. Dans un tableau associatif chaque donnée stockée (*Value*) et représentée par un objet (*Key*). La clé (*Key*) doit être unique, par contre les données stockées (*Value*) peuvent être multiple (mais à des clés différentes).

- Dans le cadre de ce cours nous ne travaillerons que sur les `List` et les `Map`.

L’interface `List` et la classe `ArrayList`

L’interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n’est pas forcement possible.

L’interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n’est pas forcement possible.

- Les différentes classes implémentant l’interface `List` permettent de gérer ce type d’ensemble de données.

L’interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n’est pas forcement possible.

- Les différentes classes implémentant l’interface `List` permettent de gérer ce type d’ensemble de données.

- L’interface impose de nombreuses méthodes pour gérer ces collections :

L'interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n'est pas forcement possible.

- Les différentes classes implémentant l'interface `List` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

    - *add* pour ajouter un élément,

L'interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n'est pas forcement possible.

- Les différentes classes implémentant l'interface `List` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

  - `add` pour ajouter un élément,

  - `contains` pour recherche un élément dans la `List`

L'interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n'est pas forcement possible.

- Les différentes classes implémentant l'interface `List` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

  - `add` pour ajouter un élément,

  - `contains` pour recherche un élément dans la `List`

  - `get` pour accéder à une élément à une position donnée

L'interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n'est pas forcement possible.

- Les différentes classes implémentant l'interface `List` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

  - `add` pour ajouter un élément,

  - `contains` pour recherche un élément dans la `List`

  - `get` pour accéder à une élément à une position donnée

  - `remove` pour enlever un élément de la `List`,...

L'interface `List` et la classe `ArrayList`

- Pour certaines applications les tableaux ne sont pas utilisables ; pour créer un tableau il est nécessaire de connaître sa taille à la création ce qui n'est pas forcement possible.

- Les différentes classes implémentant l'interface `List` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :
  - `add` pour ajouter un élément,
  - `contains` pour recherche un élément dans la `List`
  - `get` pour accéder à une élément à une position donnée
  - `remove` pour enlever un élément de la `List`,...

- Dans le cadre de ce cours nous n'utiliserons que la classe `ArrayList`

L’interface `List` et la classe `ArrayList`

L'interface `List` et la classe `ArrayList`

- Lors de la déclaration d'une collection, on déclare l'interface voulue (ici `List`) suivie d'une classe (ou d'un type) entre les symboles `< >` ; cette notation fait appel à la généricité.

L'interface `List` et la classe `ArrayList`

- Lors de la déclaration d'une collection, on déclare l'interface voulue (ici `List`) suivie d'une classe (ou d'un type) entre les symboles `< >` ; cette notation fait appel à la généricité.

```java
List<String> maListe;
maListe = new ArrayList<String>();
maListe.add("Premier element");
maListe.add("Deuxieme element");
//...
System.out.println("Le permier element est : " + maListe.get(0));
//...
for (String elt : maListe) {
    System.out.println(elt);
}
```

L'interface `List` et la classe `ArrayList`

- Lors de la déclaration d'une collection, on déclare l'interface voulue (ici `List`) suivie d'une classe (ou d'un type) entre les symboles `< >` ; cette notation fait appel à la généricité.

- La classe choisie pour l'implémentation est spécifiée lors de la construction (ici une `ArrayList`) qui reprend la même notation.

```java
List<String> maListe;
maListe = new ArrayList<String>();
maListe.add("Premier element");
maListe.add("Deuxieme element");
//...
System.out.println("Le permier element est : " + maListe.get(0));
//...
for (String elt : maListe) {
    System.out.println(elt);
}
```

L'interface `List` et la classe `ArrayList`

- Lors de la déclaration d'une collection, on déclare l'interface voulue (ici `List`) suivie d'une classe (ou d'un type) entre les symboles `< >` ; cette notation fait appel à la généricité.

- La classe choisie pour l'implémentation est spécifiée lors de la construction (ici une `ArrayList`) qui reprend la même notation.

- En utilisant l'interface pour la déclaration (y compris dans les méthodes) on peut modifier le choix de l'implémentation (le type de `List`) sans modifications conséquentes du programme.

```
List<String> maListe;
maListe = new ArrayList<String>();
maListe.add("Premier element");
maListe.add("Deuxieme element");
//...
System.out.println("Le permier element est : " + maListe.get(0));
//...
for (String elt : maListe) {
    System.out.println(elt);
}
```

L'interface `Map` et la classe `HashMap`

L’interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d’utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

L’interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d’utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l’interface `Map` permettent de gérer ce type d’ensemble de données.

L’interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d’utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l’interface `Map` permettent de gérer ce type d’ensemble de données.

- L’interface impose de nombreuses méthodes pour gérer ces collections :

L'interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d'utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l'interface `Map` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

    - `put` pour ajouter un élément,

L'interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d'utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l'interface `Map` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

    - `put` pour ajouter un élément,

    - `containsKey` pour recherche une clé (un mot dans l'exemple)

L'interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d'utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l'interface `Map` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :

    - `put` pour ajouter un élément,

    - `containsKey` pour recherche une clé (un mot dans l'exemple)

    - `containsValue` pour recherche une valeur (une définition dans l'exemple)

L'interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d'utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l'interface `Map` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :
  - `put` pour ajouter un élément,
  - `containsKey` pour recherche une clé (un mot dans l'exemple)
  - `containsValue` pour recherche une valeur (une définition dans l'exemple)
  - `remove` pour enlever un élément de la `Map`,...

L'interface `Map` et la classe `HashMap`

- Les tableaux (et les listes) utilisent un entier comme identifiant pour chaque élément. Il est souvent plus pratique de pouvoir associer un objet à un autre (cela évite d'utiliser deux tableaux partageant les mêmes indices). Par exemple, pour un dictionnaire on associe une chaîne (le mot) à une autre chaîne (la définition).

- Les différentes classes implémentant l'interface `Map` permettent de gérer ce type d'ensemble de données.

- L'interface impose de nombreuses méthodes pour gérer ces collections :
    - `put` pour ajouter un élément,
    - `containsKey` pour recherche une clé (un mot dans l'exemple)
    - `containsValue` pour recherche une valeur (une définition dans l'exemple)
    - `remove` pour enlever un élément de la `Map`,...
- Dans le cadre de ce cours nous n'utiliserons que la classe `HashMap`

L'interface `Map` et la classe `HashMap`

L'interface `Map` et la classe `HashMap`

- L'interface `Map` et les classes associées utilisent une écriture proche de celle des `List`.

L'interface `Map` et la classe `HashMap`

- L'interface `Map` et les classes associées utilisent une écriture proche de celle des `List`.

- Lors de la déclaration l'interface est suivie des types de la clé et de la valeur séparés par une virgule placés dans les signes < >

L'interface `Map` et la classe `HashMap`

- L'interface `Map` et les classes associées utilisent une écriture proche de celle des `List`.

- Lors de la déclaration l'interface est suivie des types de la clé et de la valeur séparés par une virgule placés dans les signes < >

```
Map<String,Date> anniversaire;
anniversaire = new HashMap<String,Date>();
//...
String nom = "Paul";
anniversaire.put("Pierre", new Date(1981,12,1));
anniversaire.put(nom, new Date(1983,11,13));
// Résultat incohérent car création d'un nouvel objet String pour "Pierre"
System.out.println("Pierre est né le : " + anniversaire.get("Pierre"));
// Résultat cohérent car on recherche le même objet
System.out.println("Paul est né le : " + anniversaire.get(nom));
```

L'interface `Map` et la classe `HashMap`

- L'interface `Map` et les classes associées utilisent une écriture proche de celle des `List`.

- Lors de la déclaration l'interface est suivie des types de la clé et de la valeur séparés par une virgule placés dans les signes < >

- Attention, la recherche de clé/valeur se fait sur les **références** et non pas sur la valeur des objets.

```
Map<String,Date> anniversaire;
anniversaire = new HashMap<String,Date>();
//...
String nom = "Paul";
anniversaire.put("Pierre", new Date(1981,12,1));
anniversaire.put(nom, new Date(1983,11,13));
// Résultat incohérent car création d'un nouvel objet String pour "Pierre"
System.out.println("Pierre est né le : " + anniversaire.get("Pierre"));
// Résultat cohérent car on recherche le même objet
System.out.println("Paul est né le : " + anniversaire.get(nom));
```

Principe des exceptions

Principe des exceptions

- Un programme peut provoquer des erreurs à l'exécution (qui ne sont pas des erreurs d'algorithme) non prévisibles comme par exemple ouvrir un fichier (qui n'existe pas...) accéder à un port réseau...

Principe des exceptions

- Un programme peut provoquer des erreurs à l'exécution (qui ne sont pas des erreurs d'algorithme) non prévisibles comme par exemple ouvrir un fichier (qui n'existe pas...) accéder à un port réseau...

- La gestion de ce type d'erreur est assuré par les exceptions. Les exceptions sont des objets crées pour représenter l'erreur qui vient de se produire.

Principe des exceptions

- Un programme peut provoquer des erreurs à l'exécution (qui ne sont pas des erreurs d'algorithme) non prévisibles comme par exemple ouvrir un fichier (qui n'existe pas...) accéder à un port réseau...

- La gestion de ce type d'erreur est assuré par les exceptions. Les exceptions sont des objets crées pour représenter l'erreur qui vient de se produire.

- On dit qu'une classe peut *lever* (throw) une exception quand quelque chose risque de "mal" se passer...

Principe des exceptions

- Un programme peut provoquer des erreurs à l'exécution (qui ne sont pas des erreurs d'algorithme) non prévisibles comme par exemple ouvrir un fichier (qui n'existe pas...) accéder à un port réseau...

- La gestion de ce type d'erreur est assuré par les exceptions. Les exceptions sont des objets crées pour représenter l'erreur qui vient de se produire.

- On dit qu'une classe peut *lever* (throw) une exception quand quelque chose risque de "mal" se passer...

- Toutes les exceptions sont des classes filles de la classe `Exception`. Cette classe contient quelques méthodes, notamment pour avoir du détails sur "le problème".

Principe des exceptions

Principe des exceptions

- Les méthodes pouvant provoquer des exceptions sont signalées dans leurs déclarations. Le mot clef `throw` suivie d'un (ou plusieurs) type(s) d'exception(s) est inséré entre la parenthèse fermante et l'accolade ouvrante.

Principe des exceptions

- Les méthodes pouvant provoquer des exceptions sont signalées dans leurs déclarations. Le mot clef `throw` suivie d'un (ou plusieurs) type(s) d'exception(s) est inséré entre la parenthèse fermante et l'accolade ouvrante.

- Il est possible de créer de nouvelles exception (en héritant de `Exception`) et d'en lever dans vos propres classes.

Principe des exceptions

- Les méthodes pouvant provoquer des exceptions sont signalées dans leurs déclarations. Le mot clef `throw` suivie d'un (ou plusieurs) type(s) d'exception(s) est inséré entre la parenthèse fermante et l'accolade ouvrante.

- Il est possible de créer de nouvelles exception (en héritant de `Exception`) et d'en lever dans vos propres classes.

- La syntaxe de gestion des exceptions peut paraître lourde mais elle est parfaitement gérée (et automatiquement écrite) par les environnements de développement modernes.

Gérer les exceptions

Gérer les exceptions

- Deux solutions sont possibles pour prendre en compte une exception : l'intercepter et la traiter ou l'ignorer et la renvoyer "plus haut".

Gérer les exceptions

- Deux solutions sont possibles pour prendre en compte une exception : l'intercepter et la traiter ou l'ignorer et la renvoyer "plus haut".

- La première méthode utilise une construction `try{...}catch(...){...}`

Gérer les exceptions

- Deux solutions sont possibles pour prendre en compte une exception : l'intercepter et la traiter ou l'ignorer et la renvoyer "plus haut".

- La première méthode utilise une construction `try{...}catch(...){...}`

- Le code pouvant provoquer une erreur est placé dans les accolades entre `try` et `catch`. Dans les parenthèses du `catch` on déclare le type d'exception que l'on souhaite intercepter suivi d'un nom (pour pouvoir manipuler l'objet en cas de besoin).

Gérer les exceptions

- Deux solutions sont possibles pour prendre en compte une exception : l'intercepter et la traiter ou l'ignorer et la renvoyer "plus haut".

- La première méthode utilise une construction `try{...}catch(...){...}`

- Le code pouvant provoquer une erreur est placé dans les accolades entre `try` et `catch`. Dans les parenthèses du `catch` on déclare le type d'exception que l'on souhaite intercepter suivi d'un nom (pour pouvoir manipuler l'objet en cas de besoin).

- A la suite on place le code qui sera exécuté si et seulement si une exception a eu lieu.

Gérer les exceptions

- Deux solutions sont possibles pour prendre en compte une exception : l'intercepter et la traiter ou l'ignorer et la renvoyer "plus haut".

- La première méthode utilise une construction `try{...}catch(...){...}`

- Le code pouvant provoquer une erreur est placé dans les accolades entre `try` et `catch`. Dans les parenthèses du `catch` on déclare le type d'exception que l'on souhaite intercepter suivi d'un nom (pour pouvoir manipuler l'objet en cas de besoin).

- A la suite on place le code qui sera exécuté si et seulement si une exception a eu lieu.

- Plusieurs clauses `catch` peuvent être mise en cascades pour choisir un traitement différent en fonction de l'exception.

Exemple de traitement

Exemple de traitement

- La classe `Socket` permet d'ouvrir une connexion réseau TCP vers un hôte sur un port donné.

Exemple de traitement

- La classe `Socket` permet d'ouvrir une connexion réseau TCP vers un hôte sur un port donné.

- Lors de la construction d'un objet de cette classe, plusieurs exceptions peuvent se produire (`UnknownHostException` si l'hôte est inconnu, ou `IOException` pour une erreur plus générale d'entrées/sorties).

Exemple de traitement

- La classe `Socket` permet d'ouvrir une connexion réseau TCP vers un hôte sur un port donné.

- Lors de la construction d'un objet de cette classe, plusieurs exceptions peuvent se produire (`UnknownHostException` si l'hôte est inconnu, ou `IOException` pour une erreur plus générale d'entrées/sorties).

```
public class TestReseau {
    public static void main(String[] args) {
        Socket monSocket;
        try {
            monSocket = new Socket("www.google.fr",80);
        } catch (UnknownHostException excp) {
            System.out.println("Hôte inconnu !");
        } catch (IOException excp) {
            System.out.println("Erreur d'entrées/sorties ...");
        }
    }
}
```

Ignorer une exception

Ignorer une exception

- Une exception peut être ignorée et transmise à l'élément appelant (elle peut ainsi remonter toute la hiérarchie des méthodes jusqu'à la méthode `main` et la machine virtuelle).

Ignorer une exception

- Une exception peut être ignorée et transmise à l'élément appelant (elle peut ainsi remonter toute la hiérarchie des méthodes jusqu'à la méthode `main` et la machine virtuelle).

- Il suffit simplement d'ajouter `throws` suivit du type de (ou des) exception(s) à la déclaration de méthode. En fait la méthode déclare ainsi la possibilité de lever une exception qui est celle qu'elle "laisse" passer.

Ignorer une exception

- Une exception peut être ignorée et transmise à l'élément appelant (elle peut ainsi remonter toute la hiérarchie des méthodes jusqu'à la méthode `main` et la machine virtuelle).

- Il suffit simplement d'ajouter `throws` suivit du type de (ou des) exception(s) à la déclaration de méthode. En fait la méthode déclare ainsi la possibilité de lever une exception qui est celle qu'elle "laisse" passer.

- Cette méthode est déconseillée car elle ne fait que déplacer un problème...

Ignorer une exception

- Une exception peut être ignorée et transmise à l'élément appelant (elle peut ainsi remonter toute la hiérarchie des méthodes jusqu'à la méthode `main` et la machine virtuelle).

- Il suffit simplement d'ajouter `throws` suivit du type de (ou des) exception(s) à la déclaration de méthode. En fait la méthode déclare ainsi la possibilité de lever une exception qui est celle qu'elle "laisse" passer.

- Cette méthode est déconseillée car elle ne fait que déplacer un problème...

```
public class TestReseau {
        public static void main(String[] args) throws UnknownHostException, IOException {
        Socket monSocket;
        monSocket = new Socket("www.google.fr",80);
    }
}
```

Les exceptions `RuntimeException`

Les exceptions `RuntimeException`

- Le compilateur signale certaines exceptions non traitées ; celle qui ne sont pas signalée sont des exceptions qui proviennent de la logique du code (indice qui déborde dans un tableau,...) et non d'un problème **imprévisible**.

Les exceptions `RuntimeException`

- Le compilateur signale certaines exceptions non traitées ; celle qui ne sont pas signalée sont des exceptions qui proviennent de la logique du code (indice qui déborde dans un tableau,...) et non d'un problème **imprévisible**.

- Elles peuvent toutefois être gérées comme les autres exceptions.

Les exceptions `RuntimeException`

- Le compilateur signale certaines exceptions non traitées ; celle qui ne sont pas signalée sont des exceptions qui proviennent de la logique du code (indice qui déborde dans un tableau,...) et non d'un problème **imprévisible**.

- Elles peuvent toutefois être gérées comme les autres exceptions.

- Attention toutefois à ne pas remplacer la recherche d'erreurs (*bugs*) par des blocs `try...catch`

Les exceptions `RuntimeException`

- Le compilateur signale certaines exceptions non traitées ; celle qui ne sont pas signalée sont des exceptions qui proviennent de la logique du code (indice qui déborde dans un tableau,...) et non d'un problème **imprévisible**.

- Elles peuvent toutefois être gérées comme les autres exceptions.

- Attention toutefois à ne pas remplacer la recherche d'erreurs (*bugs*) par des blocs `try...catch`

```
int[] tab = new int[10];
try{
    for (int i = 0; i < 11; i++) {
        tab[i] = i;
    }
}catch (ArrayIndexOutOfBoundsException exp){
    System.out.println("Débordement de l'index");
}
```

Manipulation des fichiers

Manipulation des fichiers

- La manipulation (lecture/écriture) est une tâche usuelle du programmeur.

Manipulation des fichiers

- La manipulation (lecture/écriture) est une tâche usuelle du programmeur.

- La classe `File` permet de d'accéder à un fichier en Java. Un des constructeurs permet de donner le nom de fichier sous la forme d'une chaîne de caractères.

Manipulation des fichiers

- La manipulation (lecture/écriture) est une tâche usuelle du programmeur.

- La classe `File` permet de d'accéder à un fichier en Java. Un des constructeurs permet de donner le nom de fichier sous la forme d'une chaîne de caractères.

- Cette classe propose de nombreuses méthodes pour accéder aux droits (lecture ou écriture) du fichier, pour créer des répertoires, effacer un fichier,...

Manipulation des fichiers

- La manipulation (lecture/écriture) est une tâche usuelle du programmeur.

- La classe `File` permet de d'accéder à un fichier en Java. Un des constructeurs permet de donner le nom de fichier sous la forme d'une chaîne de caractères.

- Cette classe propose de nombreuses méthodes pour accéder aux droits (lecture ou écriture) du fichier, pour créer des répertoires, effacer un fichier,...

- Pour lire et/ou écrire dans des fichiers on utilise les classe `Scanner` et `PrintWriter`.

La classe Scanner

La classe Scanner

- Depuis la version 1.5, la lecture de données est assurée par la classe Scanner beaucoup plus simple que les solutions précédentes.

La classe `Scanner`

- Depuis la version 1.5, la lecture de données est assurée par la classe `Scanner` beaucoup plus simple que les solutions précédentes.

- Plusieurs constructeurs sont possibles permettant de lire des données depuis un objet de type `File`, un objet de type `InputStream` (comme `System.in` ou une connexion réseaux), ...

La classe `Scanner`

- Depuis la version 1.5, la lecture de données est assurée par la classe `Scanner` beaucoup plus simple que les solutions précédentes.

- Plusieurs constructeurs sont possibles permettant de lire des données depuis un objet de type `File`, un objet de type `InputStream` (comme `System.in` ou une connexion réseaux), ...

- Les méthodes d'accès permettent de lire les types (comme `float`, `double`, `int`, ...) ou des chaînes de caractères (`String`).

La classe `Scanner`

- Depuis la version 1.5, la lecture de données est assurée par la classe `Scanner` beaucoup plus simple que les solutions précédentes.

- Plusieurs constructeurs sont possibles permettant de lire des données depuis un objet de type `File`, un objet de type `InputStream` (comme `System.in` ou une connexion réseaux), ...

- Les méthodes d'accès permettent de lire les types (comme `float`, `double`, `int`, ...) ou des chaînes de caractères (`String`).

- Pour les chaînes, la méthode `nextLine()` permet de lire une nouvelle ligne, la méthode `hasNextLine()` permet de savoir s'il reste une ligne à lire.

La classe `Scanner`

- Depuis la version 1.5, la lecture de données est assurée par la classe `Scanner` beaucoup plus simple que les solutions précédentes.

- Plusieurs constructeurs sont possibles permettant de lire des données depuis un objet de type `File`, un objet de type `InputStream` (comme `System.in` ou une connexion réseaux), ...

- Les méthodes d'accès permettent de lire les types (comme `float`, `double`, `int`, ...) ou des chaînes de caractères (`String`).

- Pour les chaînes, la méthode `nextLine()` permet de lire une nouvelle ligne, la méthode `hasNextLine()` permet de savoir s'il reste une ligne à lire.

- Pour tous les autres types, on trouve `nextDouble()` et `hasNextDouble()`, ...

La classe `PrintWriter`

La classe `PrintWriter`

- La classe `PrintWriter` permet d'écrire des types ou des chaînes de caractères facilement.

La classe `PrintWriter`

- La classe `PrintWriter` permet d'écrire des types ou des chaînes de caractères facilement.

- Plusieurs constructeurs sont possibles permettant d'écrire des données depuis un objet de type `File`, un objet de type `OutputStream` (comme `System.out` ou une connexion réseaux), ...

La classe `PrintWriter`

- La classe `PrintWriter` permet d'écrire des types ou des chaînes de caractères facilement.

- Plusieurs constructeurs sont possibles permettant d'écrire des données depuis un objet de type `File`, un objet de type `OutputStream` (comme `System.out` ou une connexion réseaux), ...

- Les deux méthodes les plus utilisées sont `println()` et `print()`. Elles permettent d'envoyer le type, la chaîne de caractères ou l'objet (la méthode `toString()` est alors utilisée) passé en paramètre dans le flux sortant.

La classe `PrintWriter`

- La classe `PrintWriter` permet d'écrire des types ou des chaînes de caractères facilement.

- Plusieurs constructeurs sont possibles permettant d'écrire des données depuis un objet de type `File`, un objet de type `OutputStream` (comme `System.out` ou une connexion réseaux), ...

- Les deux méthodes les plus utilisées sont `println()` et `print()`. Elles permettent d'envoyer le type, la chaîne de caractères ou l'objet (la méthode `toString()` est alors utilisée) passé en paramètre dans le flux sortant.

- La méthode `println()` ajoute un saut de ligne à la fin de la chaîne contrairement à `print()`.

La classe `PrintWriter`

- La classe `PrintWriter` permet d'écrire des types ou des chaînes de caractères facilement.

- Plusieurs constructeurs sont possibles permettant d'écrire des données depuis un objet de type `File`, un objet de type `OutputStream` (comme `System.out` ou une connexion réseaux), ...

- Les deux méthodes les plus utilisées sont `println()` et `print()`. Elles permettent d'envoyer le type, la chaîne de caractères ou l'objet (la méthode `toString()` est alors utilisée) passé en paramètre dans le flux sortant.

- La méthode `println()` ajoute un saut de ligne à la fin de la chaîne contrairement à `print()`.

- Un flux doit être fermé en utilisant la méthode `close()`.

Exemple de lecture/écriture de fichiers

Exemple de lecture/écriture de fichiers

```
import java.io.*;
import java.util.Scanner;

public class LectureCopie {
    public static void main(String[] args) throws FileNotFoundException {
        File src,dest;
        Scanner sc;
        PrintWriter pw;
        String ligneLue;

        src = new File("/Users/jb/RevisionsC/Bonjour.c");
        dest = new File("/Users/jb/CopieBonjour.c");

        sc = new Scanner(src);
        pw = new PrintWriter(dest);

        while(sc.hasNextLine()){
            ligneLue = sc.nextLine();
            System.out.println("La ligne lue : " + ligneLue);
            pw.println(ligneLue);
        }
        pw.close();
    }
}
```

Les Threads

Les Threads

- Des programmes complets sont souvent composés de petits programmes s’exécutant en parallèle (pas tout à fait en même temps) pour éviter des lenteurs lors de l’exécution.

Les Threads

- Des programmes complets sont souvent composés de petits programmes s'exécutant en parallèle (pas tout à fait en même temps) pour éviter des lenteurs lors de l'exécution.

- Ces petits programmes peuvent être des processus (plutôt gérés par le système d'exploitation) et les threads (processus léger gérés par une seule application). Java facilite la programmation de threads avec la classe `Thread`.

Les Threads

- Des programmes complets sont souvent composés de petits programmes s'exécutant en parallèle (pas tout à fait en même temps) pour éviter des lenteurs lors de l'exécution.

- Ces petits programmes peuvent être des processus (plutôt gérés par le système d'exploitation) et les threads (processus léger gérés par une seule application). Java facilite la programmation de threads avec la classe `Thread`.

- Un `Thread` est un petit programme que l'on pourra exécuter en parallèle du programme principal.

Les Threads

- Des programmes complets sont souvent composés de petits programmes s'exécutant en parallèle (pas tout à fait en même temps) pour éviter des lenteurs lors de l'exécution.

- Ces petits programmes peuvent être des processus (plutôt gérés par le système d'exploitation) et les threads (processus léger gérés par une seule application). Java facilite la programmation de threads avec la classe `Thread`.

- Un `Thread` est un petit programme que l'on pourra exécuter en parallèle du programme principal.

- Il est possible de choisir le moment de l'exécution, de mettre le thread en pause,...

Les Threads

- Des programmes complets sont souvent composés de petits programmes s'exécutant en parallèle (pas tout à fait en même temps) pour éviter des lenteurs lors de l'exécution.

- Ces petits programmes peuvent être des processus (plutôt gérés par le système d'exploitation) et les threads (processus léger gérés par une seule application). Java facilite la programmation de threads avec la classe `Thread`.

- Un `Thread` est un petit programme que l'on pourra exécuter en parallèle du programme principal.

- Il est possible de choisir le moment de l'exécution, de mettre le thread en pause,...

- Seuls quelques éléments de bases des threads seront présentés, la programmation multitâches étant un sujet très vaste (échange de données, ...)

La classe `Thread` et le programme principal

La classe `Thread` et le programme principal

- La méthode statique `sleep(long milli)` de la classe `Thread` permet de mettre en veille le thread courant en pause pendant milli millisecondes. Cette méthode peut provoquer une `InterruptedException`.

La classe `Thread` et le programme principal

- La méthode statique `sleep(long milli)` de la classe `Thread` permet de mettre en veille le thread courant en pause pendant milli millisecondes. Cette méthode peut provoquer une `InterruptedException`.

- Chaque programme Java comporte au moins un thread qui correspond à la méthode `main`. Ce thread peut être mis en pause en utilisant la méthode ci dessus.

La classe `Thread` et le programme principal

- La méthode statique `sleep(long milli)` de la classe `Thread` permet de mettre en veille le thread courant en pause pendant milli millisecondes. Cette méthode peut provoquer une `InterruptedException`.

- Chaque programme Java comporte au moins un thread qui correspond à la méthode `main`. Ce thread peut être mis en pause en utilisant la méthode ci dessus.

```java
public class TestPause {
    public static void main(String[] args){
        System.out.println("Début");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Le thread à été interrompu");
        }
        System.out.println("Fin 5 seconde plus tard");
    }
}
```

Création d'un nouveau `Thread`

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

- Cette classe doit surcharger la méthode `run()` de la classe mère. Dans cette méthode on place tout le code à exécuter. Si il est nécessaire de recevoir des arguments, on les passera par le constructeur ou par une autre méthode.

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

- Cette classe doit surcharger la méthode `run()` de la classe mère. Dans cette méthode on place tout le code à exécuter. Si il est nécessaire de recevoir des arguments, on les passera par le constructeur ou par une autre méthode.

- Le thread est crée comme n'importe quel autre objet. Son exécution est déclenchée par l'appel de la méthode `start()` (et pas de la méthode `run()`).

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

- Cette classe doit surcharger la méthode `run()` de la classe mère. Dans cette méthode on place tout le code à exécuter. Si il est nécessaire de recevoir des arguments, on les passera par le constructeur ou par une autre méthode.

- Le thread est crée comme n'importe quel autre objet. Son exécution est déclenchée par l'appel de la méthode `start()` (et pas de la méthode `run()`).

```
public class MonThread extends Thread {
    public void run() {
        System.out.println("Debut du thread...");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Thread interrompu !");
        }
        System.out.println("... Fin du thread");
    }
}
```

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

- Cette classe doit surcharger la méthode `run()` de la classe mère. Dans cette méthode on place tout le code à exécuter. Si il est nécessaire de recevoir des arguments, on les passera par le constructeur ou par une autre méthode.

- Le thread est crée comme n'importe quel autre objet. Son exécution est déclenchée par l'appel de la méthode `start()` (et pas de la méthode `run()`).

```
public class TestThread {
    public static void main(String[] args) {
        MonThread t = new MonThread();
        System.out.println("Lancement du thread");
        t.start();
        System.out.println("Après lancement”);
    }
}
```

Création d'un nouveau `Thread`

- Pour créer un nouveau thread, on doit créer un classe qui hérite de `Thread`.

- Cette classe doit surcharger la méthode `run()` de la classe mère. Dans cette méthode on place tout le code à exécuter. Si il est nécessaire de recevoir des arguments, on les passera par le constructeur ou par une autre méthode.

- Le thread est crée comme n'importe quel autre objet. Son exécution est déclenchée par l'appel de la méthode `start()` (et pas de la méthode `run()`).

```
Lancement du thread
Début du thread...
Après lancement
... Fin du thread
```

Utilisation des threads

Utilisation des threads

- Quand un programme doit interagir avec le matériel informatique (accès réseaux, lecture de fichiers, impressions, ...) les threads sont utilisés pour éviter de bloquer le programme principal.

Utilisation des threads

- Quand un programme doit interagir avec le matériel informatique (accès réseaux, lecture de fichiers, impressions, ...) les threads sont utilisés pour éviter de bloquer le programme principal.

- Les programmes basés sur une interface graphique (GUI) utilisent des threads pour éviter de “geler” l’interface pendant la réalisation d’une tâche.

Utilisation des threads

- Quand un programme doit interagir avec le matériel informatique (accès réseaux, lecture de fichiers, impressions, ...) les threads sont utilisés pour éviter de bloquer le programme principal.

- Les programmes basés sur une interface graphique (GUI) utilisent des threads pour éviter de “geler” l’interface pendant la réalisation d’une tâche.

- Les threads permettent de simplifier un traitement complexe en petits traitements plus simple et plus économe en temps (souvent, ...).

Utilisation des threads

- Quand un programme doit interagir avec le matériel informatique (accès réseaux, lecture de fichiers, impressions, ...) les threads sont utilisés pour éviter de bloquer le programme principal.

- Les programmes basés sur une interface graphique (GUI) utilisent des threads pour éviter de “geler” l’interface pendant la réalisation d’une tâche.

- Les threads permettent de simplifier un traitement complexe en petits traitements plus simple et plus économe en temps (souvent, ...).

- Les threads sont utilisés dans la programmation multi-agents pour la recherche, les jeux vidéos, ...

Présentation des paquets

Présentation des paquets

- Pour organiser un projet complexe, Java introduit la notion de paquets (*package*). Les classes peuvent ainsi être regroupées par affinités.

Présentation des paquets

- Pour organiser un projet complexe, Java introduit la notion de paquets (*package*). Les classes peuvent ainsi être regroupées par affinités.

- Pour placer une classe dans un paquet, il suffit d'ajouter `package` suivie du nom du paquet sur la première ligne du fichier (donc avant `class`, `import`, ...).

Présentation des paquets

- Pour organiser un projet complexe, Java introduit la notion de paquets (*package*). Les classes peuvent ainsi être regroupées par affinités.

- Pour placer une classe dans un paquet, il suffit d'ajouter `package` suivie du nom du paquet sur la première ligne du fichier (donc avant `class`, `import`, ...).

- L'importation d'une classe appartenant à un paquet différent (il n'est pas nécessaire d'importer les classes appartenant au même package) se fait à l'aide de la clause `import` suivit du nom du paquet et du nom de la classe.

Présentation des paquets

- Pour organiser un projet complexe, Java introduit la notion de paquets (*package*). Les classes peuvent ainsi être regroupées par affinités.

- Pour placer une classe dans un paquet, il suffit d'ajouter `package` suivie du nom du paquet sur la première ligne du fichier (donc avant `class`, `import`, ...).

- L'importation d'une classe appartenant à un paquet différent (il n'est pas nécessaire d'importer les classes appartenant au même package) se fait à l'aide de la clause `import` suivit du nom du paquet et du nom de la classe.

- Les paquets ont aussi une conséquence sur la visibilité (notamment si aucun attribut (donc *friendly*) n'est spécifié).

Hiérarchie dans les paquets

Hiérarchie dans les paquets

- Les classes sont stockées dans des répertoires ayant le même nom que les paquets. Par exemple une classe déclarée avec `package monpaquet.mesclasses` sera stockée dans le répertoire `mesclasses`, lui même étant dans le répertoire `monpaquet`.

Hiérarchie dans les paquets

- Les classes sont stockées dans des répertoires ayant le même nom que les paquets. Par exemple une classe déclarée avec `package monpaquet.mesclasses` sera stockée dans le répertoire `mesclasses`, lui même étant dans le répertoire `monpaquet`.
- Bien que les packages soient organisés comme des fichiers, l'importation ne suit pas les règles des fichiers. Le joker (*) permet d'importer toutes les classes d'un paquets mais n'importe pas toutes les classes comprises dans les paquets inclus.

Hiérarchie dans les paquets

- Les classes sont stockées dans des répertoires ayant le même nom que les paquets. Par exemple une classe déclarée avec `package monpaquet.mesclasses` sera stockée dans le répertoire `mesclasses`, lui même étant dans le répertoire `monpaquet`.

- Bien que les packages soient organisés comme des fichiers, l'importation ne suit pas les règles des fichiers. Le joker (*) permet d'importer toutes les classes d'un paquets mais n'importe pas toutes les classes comprises dans les paquets inclus.

- Les environnements de développement modernes facilitent l'importation de classe en proposant d'ajouter automatiquement l'importation des classes. Il faut quand même attention à choisir la bonne classe dans le cas de classes ayant le même nom mais pas dans le même paquet.

Génération automatique de documentation

Génération automatique de documentation

- Lors du développement d'un programme il est utile de tenir à jour une documentation complète des différents éléments.

Génération automatique de documentation

- Lors du développement d'un programme il est utile de tenir à jour une documentation complète des différents éléments.

- Java fournit un outil automatique de génération de la documentation à partir des commentaires présents dans le code source.

Génération automatique de documentation

- Lors du développement d'un programme il est utile de tenir à jour une documentation complète des différents éléments.

- Java fournit un outil automatique de génération de la documentation à partir des commentaires présents dans le code source.

- Le programmeur n'a qu'à ajouter des balises particulières à ses commentaires pour qu'ils soient ajoutés à la documentation. Les commentaires devant être intégrés à la documentation doivent être ouvert avec `/**` et fermés avec `*/`

Génération automatique de documentation

- Lors du développement d'un programme il est utile de tenir à jour une documentation complète des différents éléments.

- Java fournit un outil automatique de génération de la documentation à partir des commentaires présents dans le code source.

- Le programmeur n'a qu'à ajouter des balises particulières à ses commentaires pour qu'ils soient ajoutés à la documentation. Les commentaires devant être intégrés à la documentation doivent être ouvert avec `/**` et fermés avec `*/`

- La documentation est ensuite construite par l'exécutable `javadoc` qui construit un ensemble de fichier HTML pouvant être consulté par n'importe quel navigateur web.

Génération automatique de documentation

Génération automatique de documentation

- Les balises sont des mots clés précédés du caractère @. Certaines balises ne sont applicables qu'à certains éléments (classes, méthodes, ...)

Génération automatique de documentation

- Les balises sont des mots clés précédés du caractère @. Certaines balises ne sont applicables qu'à certains éléments (classes, méthodes, ...)

- D'autres balises sont générales comme par exemple :

Génération automatique de documentation

- Les balises sont des mots clés précédés du caractère @. Certaines balises ne sont applicables qu'à certains éléments (classes, méthodes, ...)

- D'autres balises sont générales comme par exemple :

  - `@see` qui permet de construire un lien vers une explication complémentaire (site web) ou vers un autre élément du programme.

Génération automatique de documentation

- Les balises sont des mots clés précédés du caractère @. Certaines balises ne sont applicables qu'à certains éléments (classes, méthodes, ...)

- D'autres balises sont générales comme par exemple :

  - `@see` qui permet de construire un lien vers une explication complémentaire (site web) ou vers un autre élément du programme.

  - `@since` définit depuis quelle version du projet cet élément est présent, très utile pour les suivis de version.

Génération automatique de documentation

- Les balises sont des mots clés précédés du caractère @. Certaines balises ne sont applicables qu'à certains éléments (classes, méthodes, ...)

- D'autres balises sont générales comme par exemple :
  - `@see` qui permet de construire un lien vers une explication complémentaire (site web) ou vers un autre élément du programme.
  - `@since` définit depuis quelle version du projet cet élément est présent, très utile pour les suivis de version.

- `javadoc` retrouve automatiquement les notions d'héritages, d'implémentation d'interface, de visibilité... Il n'est donc pas utile de les ajouter dans les commentaires.

Documentation d’une classe

Documentation d’une classe

- En plus des balises générales, les balises suivantes sont souvent utilisées pour commenter une classe :

Documentation d’une classe

- En plus des balises générales, les balises suivantes sont souvent utilisées pour commenter une classe :
    - `@version` : est utilisé pour donner un numéro de version à la classe, souvent utiliser pour vérifier les évolutions d’un gros projet.

Documentation d’une classe

- En plus des balises générales, les balises suivantes sont souvent utilisées pour commenter une classe :
  - `@version` : est utilisé pour donner un numéro de version à la classe, souvent utiliser pour vérifier les évolutions d’un gros projet.
  - `@author` : permet de signaler l’auteur de la classe notamment pour les projets développés par une équipe

Documentation d’une classe

- En plus des balises générales, les balises suivantes sont souvent utilisées pour commenter une classe :
  - `@version` : est utilisé pour donner un numéro de version à la classe, souvent utiliser pour vérifier les évolutions d’un gros projet.
  - `@author` : permet de signaler l’auteur de la classe notamment pour les projets développés par une équipe

```java
/**
 * Représente une porte ET à plusieurs entrées
 *
 * @author Jean-Baptiste Vioix
 * @version 1.0
 */
public class PorteET extends Porte{
   ...
}
```

Documentation d’une classe

- En plus des balises générales, les balises suivantes sont souvent utilisées pour commenter une classe :
  - `@version` : est utilisé pour donner un numéro de version à la classe, souvent utiliser pour vérifier les évolutions d’un gros projet.
  - `@author` : permet de signaler l’auteur de la classe notamment pour les projets développés par une équipe

Class PorteET

```
java.lang.Object
  └Porte
      └PorteET
```

```
public class PorteET
extends Porte
```

Représente une porte ET à plusieurs entrées

Version:
1.0
Author:
Jean-Baptiste Vioix

Documentation des méthodes et des attributs

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :
    - `@return` : permet de détailler le type de retour de la méthode.

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :
  - `@return` : permet de détailler le type de retour de la méthode.
  - `@param` : est utiliser pour présenter les paramètres ainsi que leurs types. On utilise une balise `@param` par paramètre.

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :
  - `@return` : permet de détailler le type de retour de la méthode.
  - `@param` : est utiliser pour présenter les paramètres ainsi que leurs types. On utilise une balise `@param` par paramètre.
- Pour les attributs il n'y a pas de balises particulière, on place juste un commentaire `javadoc` au dessus avec quelques explications.

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :
  - `@return` : permet de détailler le type de retour de la méthode.
  - `@param` : est utiliser pour présenter les paramètres ainsi que leurs types. On utilise une balise `@param` par paramètre.
- Pour les attributs il n'y a pas de balises particulière, on place juste un commentaire `javadoc` au dessus avec quelques explications.

```
/**
* Cette méthode calcule le produit de 2 entiers
*
* @param a un entier qui représente le multiplicante
* @param b un entier qui représente le multiplicateur
* @return le produit
*/
public int produit(int a,int b){
    ...
}
```

Documentation des méthodes et des attributs

- Les principales balises associées aux méthodes sont :
  - `@return` : permet de détailler le type de retour de la méthode.
  - `@param` : est utiliser pour présenter les paramètres ainsi que leurs types. On utilise une balise `@param` par paramètre.
- Pour les attributs il n'y a pas de balises particulière, on place juste un commentaire `javadoc` au dessus avec quelques explications.

produit

```
public int produit(int a,
                   int b)
```

Cette méthode calcule le produit de 2 entiers

Parameters:

`a` - un entier qui représente le multiplicante

`b` - un entier qui représente le multiplicateur

Returns:

le produit

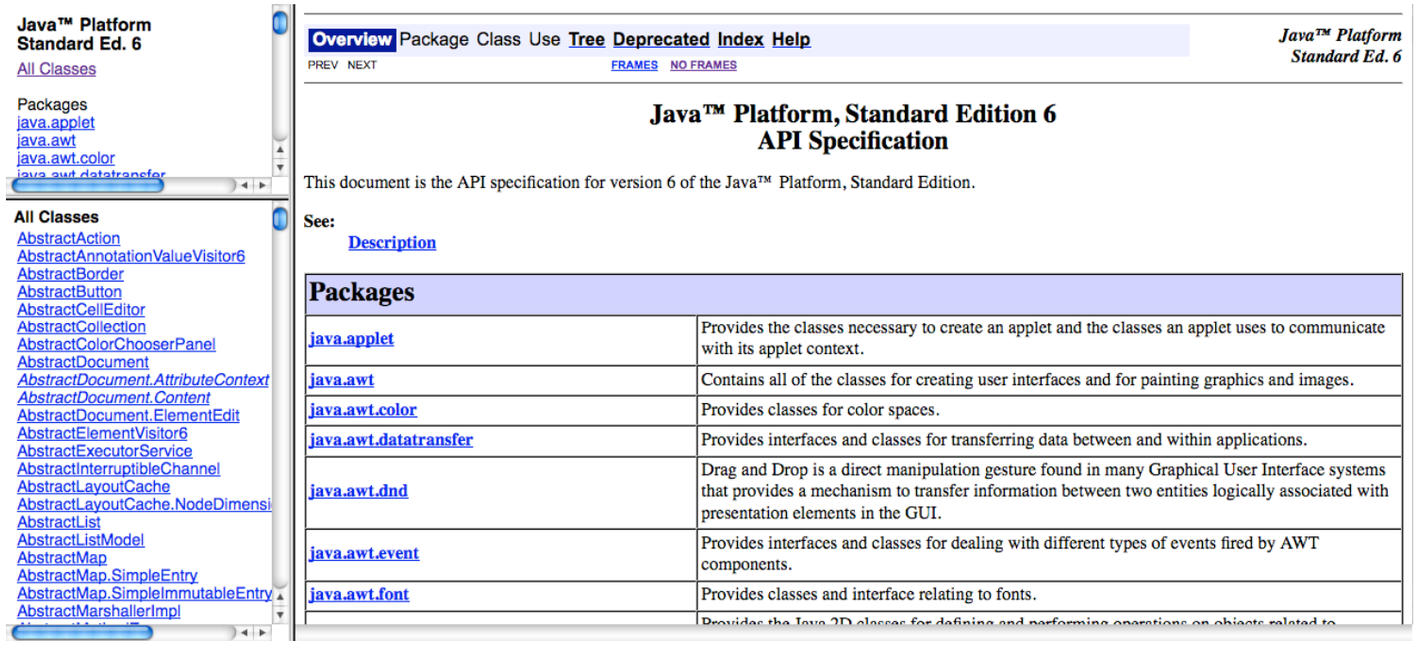
La documentation officielle

La documentation officielle

- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/

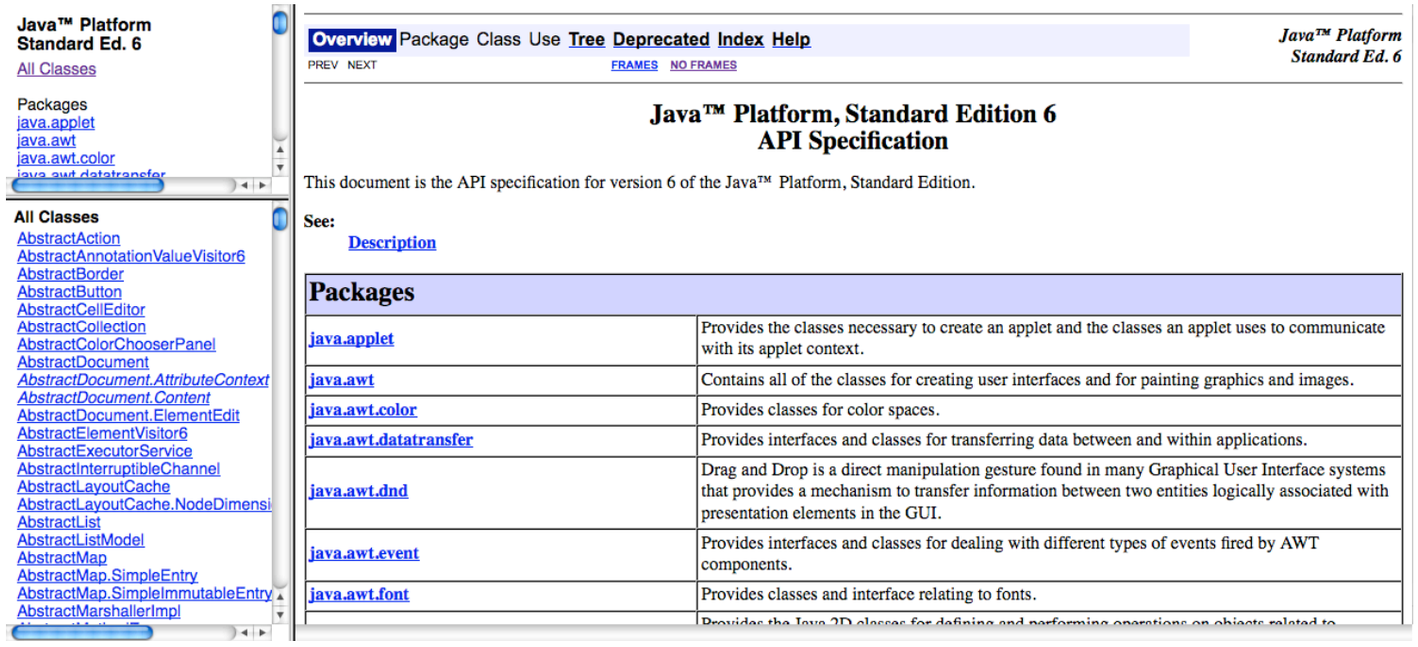
La documentation officielle

- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



La documentation officielle

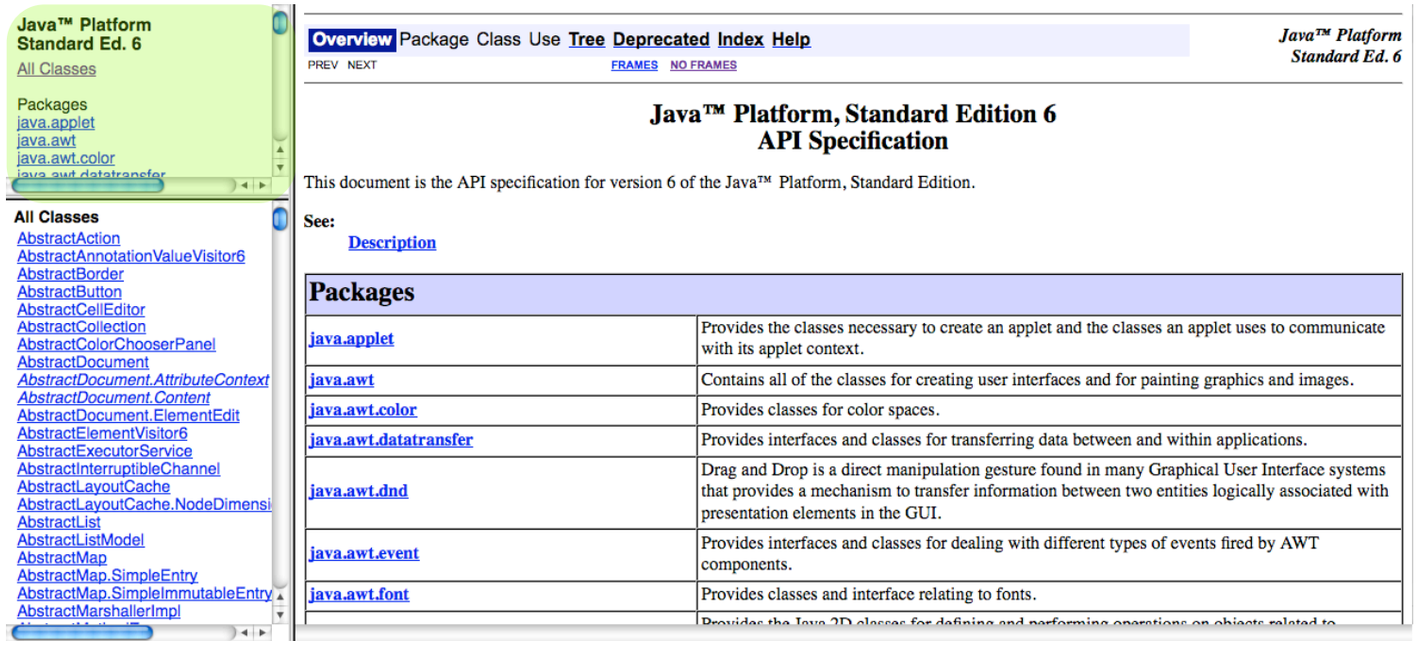
- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



- Ce site web est composé d'un **explorateur de paquets**, d'un **explorateur de classes** et de l'**affichage de l'élément sélectionné**.

La documentation officielle

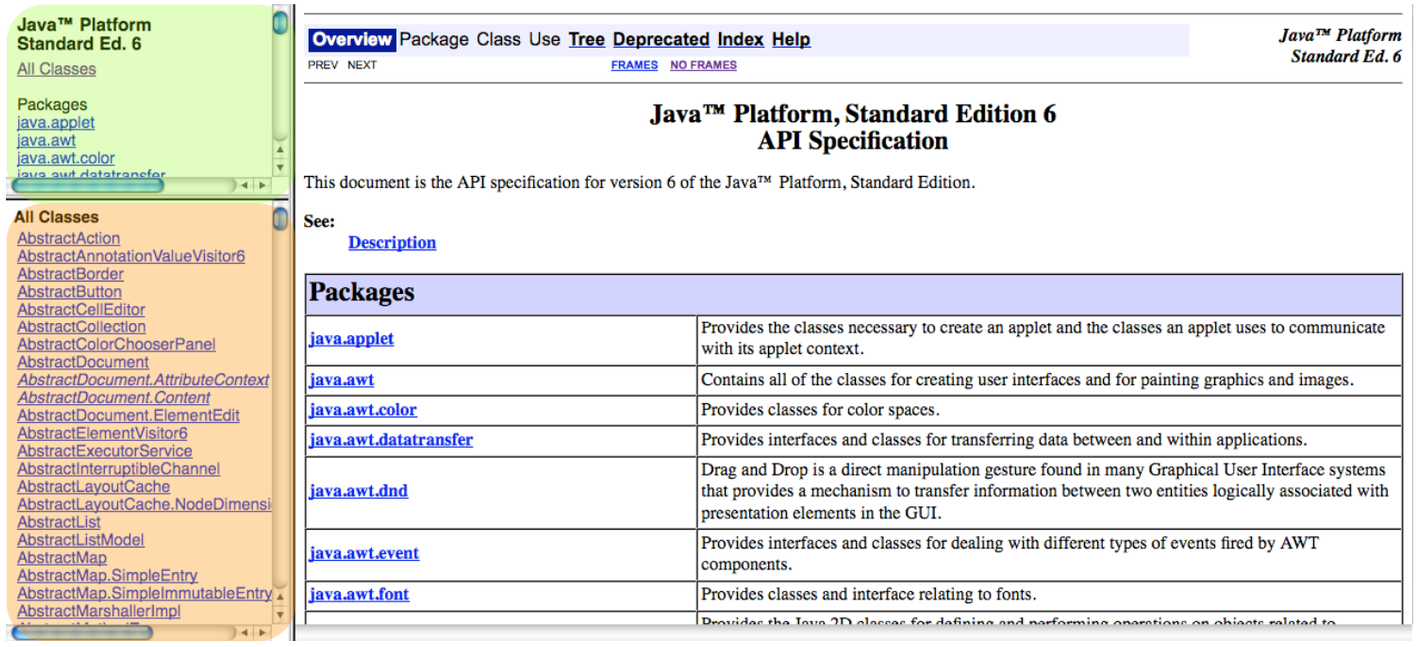
- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



- Ce site web est composé d'un **explorateur de paquets**, d'un **explorateur de classes** et de l'**affichage de l'élément sélectionné**.

La documentation officielle

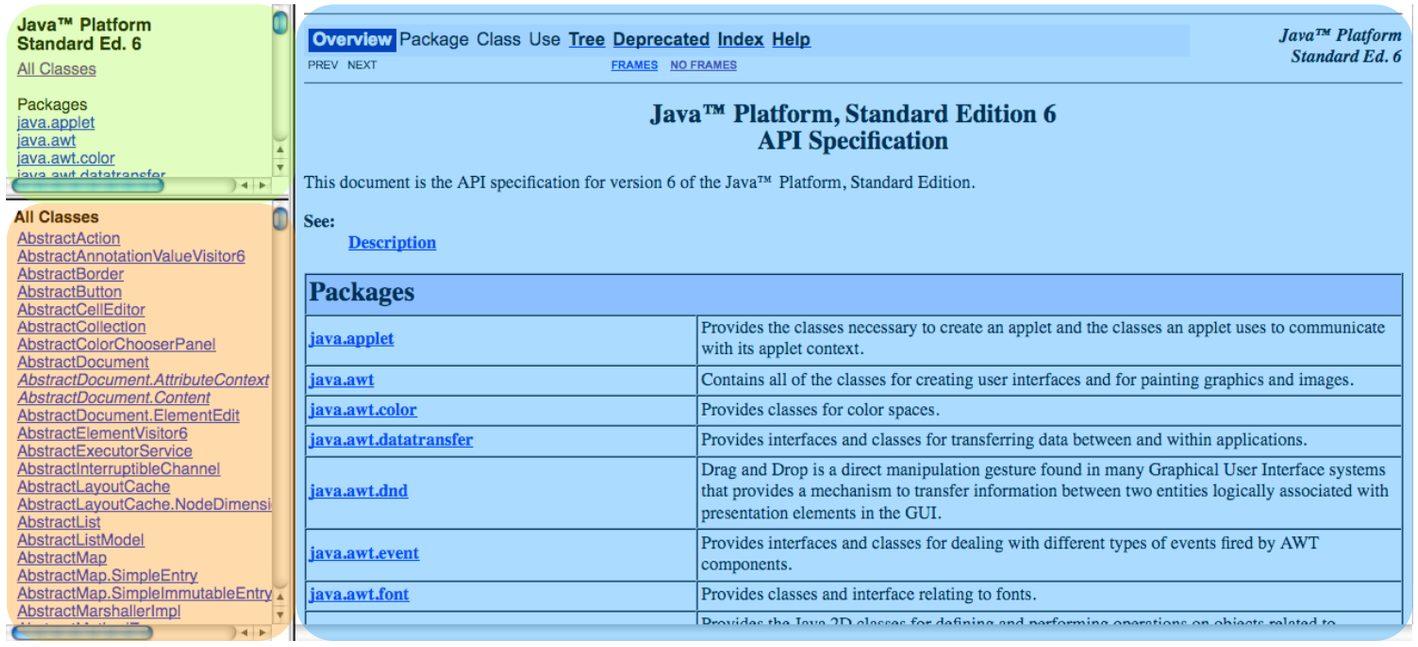
- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



- Ce site web est composé d'un **explorateur de paquets**, d'un **explorateur de classes** et de l'**affichage de l'élément sélectionné**.

La documentation officielle

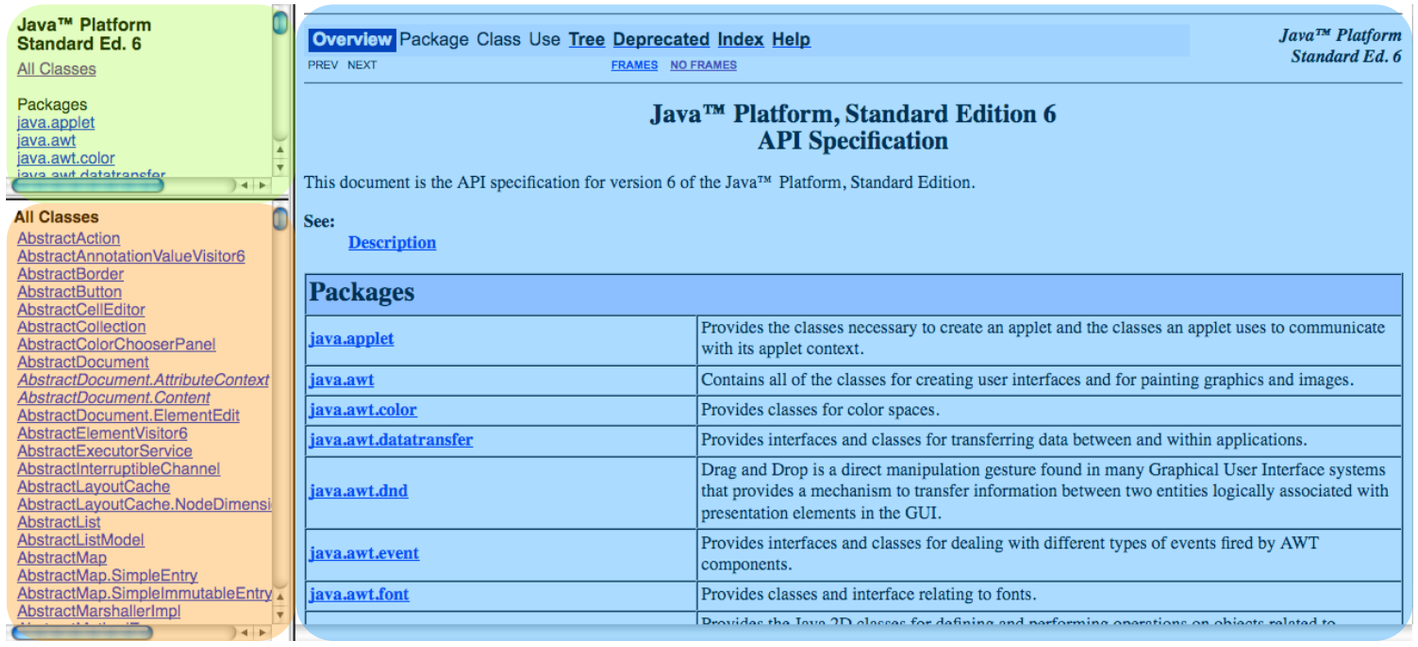
- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



- Ce site web est composé d'un **explorateur de paquets**, d'un **explorateur de classes** et de l'**affichage de l'élément sélectionné**.

La documentation officielle

- Toutes les classes présentes dans Java sont documentées sur http://java.sun.com/javase/6/docs/api/



- Ce site web est composé d'un **explorateur de paquets**, d'un **explorateur de classes** et de l'**affichage de l'élément sélectionné**.

- Cette documentation est très utilisée pour connaître les méthodes et les attributs des différentes classes.

La documentation officielle : détails d'une classe

---

java.lang

Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

La documentation officielle : détails d’une classe

- Dans l’explorateur de classes, on sélectionne la classe `String`.

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

java.lang

Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

Field Summary	
`static` Comparator<String>	**`CASE_INSENSITIVE_ORDER`** A Comparator that orders `String` objects as by `compareToIgnoreCase`.

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

Constructor Summary

String()
Initializes a newly created String object so that it represents an empty character sequence.

String(byte[] bytes)
Constructs a new String by decoding the specified array of bytes using the platform's default charset.

String(byte[] bytes, Charset charset)
Constructs a new String by decoding the specified array of bytes using the specified charset.

String(byte[] ascii, int hibyte)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*

String(byte[] bytes, int offset, int length)
Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

String(byte[] bytes, int offset, int length, Charset charset)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] ascii, int hibyte, int offset, int count)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*

String(byte[] bytes, int offset, int length, String charsetName)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] bytes, String charsetName)

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

Method Summary

char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.

La documentation officielle : détails d'une classe

- Dans l'explorateur de classes, on sélectionne la classe `String`.

- L'aide de la classe se compose de trois parties : une présentation générale (avec quelques exemples d'utilisation dans certains cas), les attributs, les constructeurs, le résumé des méthodes puis le détails de tous ces éléments.

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:
`charAt` in interface `CharSequence`
Parameters:
`index` - the index of the `char` value.
Returns:
the `char` value at the specified index of this string. The first `char` value is at index `0`.
Throws:
`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.