EXAMEN, PROGRAMMATION ORIENTEE OBJET M2103 (POO)

Nom..... Prénom..... Groupe.....

<u>Consignes</u> : vous avez 2 heures pour répondre aux questions ci-dessous.

Vous avez le droit aux imprimés des CMs (avec des notes pertinentes aux cours) de ce module et le bouquin Java Récapitulation, mais rien d'autre en tant que matériel supplémentaire. Ceci veut dire en particulier : pas de sac-à-dos, pas de portable, pas de calculatrice, pas de téléphone portable. Il n'est pas permis non plus d'avoir écrit des problèmes et solutions dans leur entier comme « note de cours ».

Lisez attentivement chaque question avant d'y répondre et faites attention à tout détail qui peut vous donner un indice sur la bonne réponse.

Attention aux consignes et ne faites que ce qui est demandé par le texte. Vous n'aurez pas le temps de finir sinon.

Finalement, faites attention au code que vous allez écrire. Un code négligent sera mal-noté. Respectez également les bonnes pratiques de la programmation.

NOTES : Cet examen contient des questions courtes et des questions plus longues. Les énoncés sont, dans la mesure possible, indépendantes, donc si vous n'arrivez pas répondre à une question, vous aurez la possibilité de passer à la prochaine.

Les points indiqués à côté des questions donnent un bilan sur 26 points. À la fin de l'examen vous avez aussi une question BONUS sur 6 points. Si votre score (avec bonus) dépassera 26 points pour l'examen le score additionnel sera pris en compte pour les scores de vos TPs et TDs notés.

L'annexe contient des extraits de Javadoc qui peuvent vous être utiles.

NOTE / 26

NOTE EXAMEN (70%) /20

MOYENNE TP (15%) /20

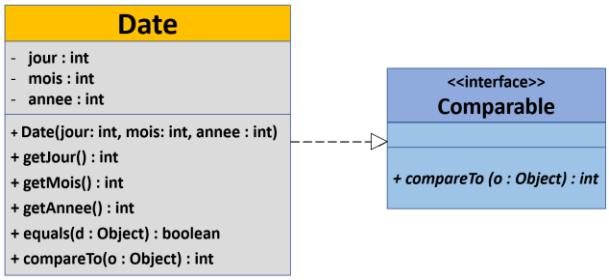
MOYENNE TD (15%) /20

NOTE MODULE /20

Question I (6 points)

Nous allons partir sur un environnement où des patients peuvent chercher et recevoir des traitements pour des diverses maladies. Dans cette première question nous allons nous concentrer sur quelques outils de base : une classe abstraite, une classe concrète et quelques interfaces.

Une première classe, Date, est donnée ci-dessous. Pour l'instant on suppose que toutes les méthodes de cette classe ont été implémentées. L'interface `Comparable` existe déjà en Java.

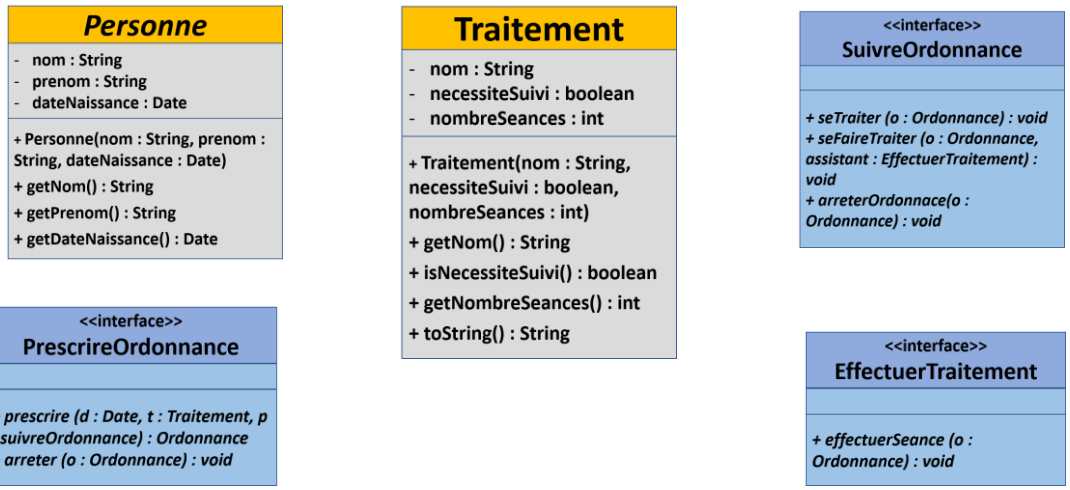


Répondez aux questions suivantes :

1. **(1.5 points)** Expliquez, dans vos propres mots, quelle est la différence entre ces concepts de programmation orientée objet (et Java) : une classe, un objet, une instance de classe.
2. **(1 point)** Une classe abstraite (plusieurs réponses possibles) :

    a. Doit contenir au moins une méthode abstraite
    b. Ne peut pas contenir ni des attributs, ni des méthodes statiques
    c. N'est jamais instanciable
    d. Peut servir de superclasse à un nombre quelconque de sousclasses
    e. Peut servir d'interface à un nombre quelconque de classes

3. Voici les diagrammes de classe d'une classe abstraite `Personne`, une classe concrète `Traitement` et de trois interfaces `PrescrireOrdonnance`, `SuivreOrdonnance` et `EffectuerTraitement`. ***Vous pouvez supposer que tous les types de variable utilisés dans le programme existent et sont bien définis.***



La classe `Personne` représente une personne : avec son nom, prénom et date de naissance. En dehors du constructeur elle contient trois méthodes, qui rendent les valeurs des trois attributs de la classe.
La classe `Traitement` représente la notion d'un traitement médical. La classe a trois attributs : un nom, une valeur booléenne `necessiteSuivi` et un nombre de séances. La valeur `necessiteSuivi` indique si un traitement peut être suivi par un patient seul (il s'agit par exemple de prendre une gélule) ou non (il s'agit d'une injection, un examen, etc.). Le nombre de séances se réfère à combien de fois le traitement doit être répété.
En dehors des constructeur et des méthodes rendant la valeur des attributs de cette classe, la classe traitement a également un méthode `toString()` qui rend une valeur `String`.

L'interface `SuivreOrdonnance` décrit le fonctionnement de suivre une `Ordonnance` (qui sera faite pour un traitement, à voir l'exercice suivant). Elle contient trois méthodes : une méthode `seTraiter(Ordonnance o)` qui spécifie l'action d'effectuer une séance de traitement sans assistance ; une méthode `seFaireTraiter(Ordonnance o, EffectuerTraitement assistant)` qui spécifie l'action d'effectuer une séance de traitement avec l'aide d'un assistant agréé ; et une méthode `arreter(Ordonnance o)` qui spécifie l'action d'arrêter de suivre une ordonnance.

L'interface `PrescrireOrdonnance` décrit le fonctionnement de prescrire une ordonnance (à une date donnée, pour un traitement donné et pour une entité qui peut suivre une ordonnance) et de la faire arrêter. Elle contient deux méthodes, chacune décrivant une de ces actions.

L'interface `EffectuerTraitement` décrit le fonctionnement d'assister une entité qui suit une Ordonnance dans le traitement. Cette interface contient une seule méthode qui spécifie l'action d'assister une séance pour une ordonnance donnée.

a. **(1 point)** Est-ce que la ligne de code ci-dessous compile ? Quel est le résultat ? (pour rappel, le diagramme de la classe `Date` se trouve dans la question I)

```
Personne personne = new Personne("Dupont", "Jean", new Date(1, 5, 1977));
```

b. **(1 point)** Détaillez la méthode `toString()` de la classe `Traitement`. Cette méthode doit retourner :
"`Traitement <nom du traitement>, <nombre seances> a effectuer.`" Si `necessiteSuivi` est mis à `true`, alors il faut ajouter : "`A effectuer avec l'assistance d'un assistant agree.`"

c. **(1.5 points)** Détaillez toute l'interface `SuivreOrdonnance`.

Question II (10.5 points)

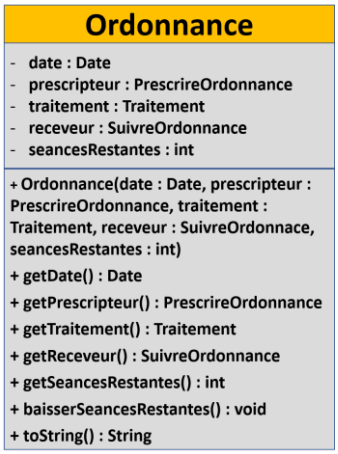
Dans cette question nous allons traiter la classe `Ordonnance`, ainsi que trois sousclasses de la classe `Personne`.

1. Voici le diagramme de classe pour la classe `Ordonnance`. Dans cette classe nous avons cinq attributs : la date quand l'ordonnance est faite, son prescripteur, le traitement qu'il faut suivre, le receveur et un attribut qui stocke de façon dynamique combien de séances il reste à réaliser d'un traitement. En dehors des méthodes de type `get` pour chaque attribut, cette classe a également :

   Un constructeur, qui met la valeur des premiers quatre attributs aux valeurs correspondantes données en paramètre, et qui initialise la valeur du nombre de séances restantes au nombre total de séances du traitement (revoir la `classe Traitement`, Question I.2)

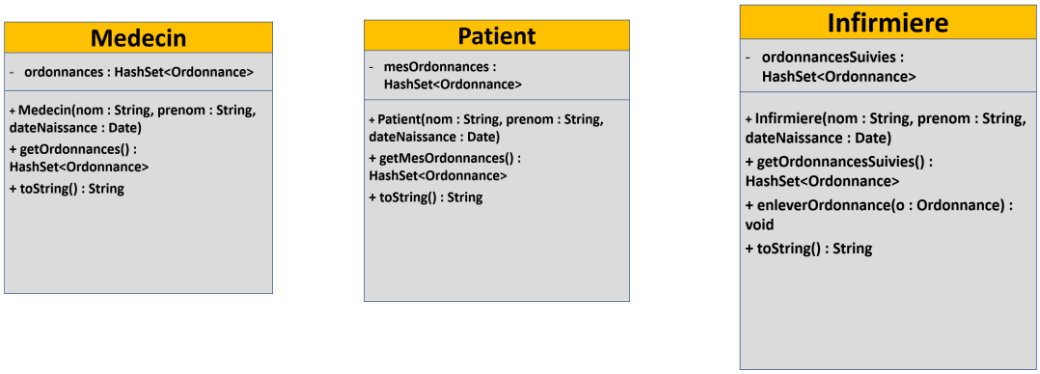
   Une méthode `baisseNombreSeances()`, qui baisse le nombre de séances restantes par 1 (toutefois, sans baisser cette valeur sous 0)

   Une méthode `toString()` qui retourne la concaténation de la valeur des cinq attributs.



(1 point) Détaillez le constructeur de la classe `Ordonnance`.

2. Ci-dessous vous avez les diagrammes de classe de trois sousclasses de la classe `Personne` : `Medecin, Patient` et `Infirmiere`. Voici des diagrammes de classe incomplètes pour ces trois classes.



La classe `Medecin` est une sousclasse de la classe `Personne` qui implémente l'interface `PrescrireOrdonnance` (voir question I). On détaille dans le diagramme de classe une collection de type `HashSet<Ordonnance>` en tant qu'attribut.

Répondez aux questions suivantes concernant la classe `Medecin` :

a. **(0.75 points)** Quels attributs un objet de la classe `Medecin` a-t-il (type et nom) ?
b. **(0.75 point)** Quelles méthodes la classe `Medecin` hérite-t-elle ?
c. **(0.75 point)** Quelles méthodes DOIT encore implémenter la classe `Medecin` ?
d. **(1.25 point)** Dans la classe `Medecin`, détaillez le constructeur donné dans le diagramme de classe, sachant que celui-ci appellera le constructeur de la classe `Personne`, puis instanciera `mesOrdonnances` en tant que nouvel `HashSet`.

La classe `Patient` est une sousclasse de `Personne`, qui implémente l'interface `SuivreOrdonnance`.
La classe `Infirmiere` est également une sousclasse de `Personne`, mais qui implémente l'interface `EffectuerTraitement`.

e. **(1.5 point)** Pour chacune de ces deux classes, indiquez quelles méthodes elles doivent implémenter (en dehors des méthodes déjà indiquées dans leurs diagrammes de classe).
f. **(1 point)** La méthode `ajouterOrdonnance(Ordonnance o)` de la classe `Patient` vérifie si le paramètre en entrée n'est pas `null` : le cas échéant, l'ordonnance o sera ajoutée aux valeurs actuellement stockées par `mesOrdonnances`. Détaillez cette méthode.

3. **(1.5 point)** Dans la classe Medecin : **détaillez la méthode** `Ordonnance prescrire (Date d, Traitement t, Patient p)` . Cette méthode devrait faire les choses suivantes : créer une nouvelle `Ordonnance` pour les paramètres en entrée (on se rappelle : on est dans la classe `Medecin` !) ; ajouter l'ordonnance à l'attribut ordonnances stocké par la classe `Medecin` ; faire le patient ajouter cette ordonnance également (voire les diagrammes de classe de Patient ainsi que de l'interface qu'elle implémente !); et retourner l'ordonnance en tant que sortie de la méthode.

4. Voici le code de la méthode `arreter(Ordonnance o)` de la classe `Medecin` :

```
@Override
public void arreter(Ordonnance ordonnance) {
	SuivreOrdonnance receveur = ordonnance.getReceveur();
	receveur.arreterOrdonnance(ordonnance);
	this.ordonnances.remove(ordonnance);
}
```

(2 points) Dans la classe `Patient` **écrivez la méthode** `seTraiter(Ordonnance o),` qui aura le fonctionnement suivant : si l'ordonnance en paramètre est bien dans le `HashSet mesOrdonnances` du patient, et si cette ordonnance est pour un traitement qui ne nécessite pas de suivi, alors on baisse le nombre de séances restantes sur cette ordonnance par 1. Si le nombre de séances restantes est maintenant à 0, on fait le prescripteur de cette ordonnance l'arrêter (voire la bonne méthode de la classe `Medecin` et de l'interface que cette dernière implémente). Si le traitement ciblé par l'ordonnance nécessite suivi, faites afficher le message suivant : « `Vous ne pouvez pas effectuer ce traitement seul. Veuillez contacter un assistant agréé.` » Si l'ordonnance donnée en paramètre n'est pas dans le `HashSet` du patient, alors faites afficher le message suivant « `Vous n'avez pas une ordonnance pour ce traitement.` »

Question III (4.5 points)

A partir de maintenant nous allons supposer que notre programme a tout le code spécifié par les questions antérieures (ceci inclura donc les classes complétées de la Question II, et non pas seulement les méthodes données par leurs diagrammes de classe).

Dans cet exercice nous allons travailler sur une classe `ActiviteMedicale`, qui aura une méthode `public static void main`.

1. **(3 points)** Dans la méthode `public static void main`, écrivez le code qui crée les objets suivants :
   - Un patient jeanDupont avec prénom et nom Jean Dupont (avec la date de naissance que vous voulez)
   - Un médecin drLaRoche avec prénom et nom Guillaume LaRoche (avec la date de naissance que vous voulez)
   - Un traitement vaccination qui s'appelle « vaccination », qui nécessite un suivi et qui a une séance seulement
   - Un traitement antibiotique qui s'appelle « antibiotique », qui ne nécessite pas de suivi, et qui part sur 15 séances
   - Une ordonnance faite par dr. LaRoche pour Jean Dupont pour le traitement de vaccination (pour la date de votre choix)
   - Une ordonnance faite par le même médecin pour Jean Dupont pour le traitement d'antibiotiques (encore une fois pour la date de votre choix)

2. **(1.5 points)** Etant donné le fonctionnement de la méthode seTraiter de la classe Patient, telle que décrite dans la Question II.4, quel est l'effet des instructions ci-dessous ? Justifiez vos réponses.

```
jeanDupont.seTraiter(vaccination) ;
System.out.println(drLaRoche.getOrdonnances()) ; // a savoir que la class
HashSet hérite une méthode toString qui affiche les éléments dans le format
[premierElement, deuxiemeElement, …]

jeanDupont.seTraiter(antibiotiques) ;
System.out.println(drLaRoche.getOrdonnances()) ;
```

Question IV (5 points)

Finalement, cette question concerne les exceptions et les tests unitaires.

1. **(1 point)** Quelle est la différence entre une exception contrôlée et une exception non-contrôlée ?
2. **(2 points)** On a le fichier suivant appelé `input.txt` :

```
Jean Dupont 3 7 1986
Marie Vauclair 25 10 1987
Suzanne Fontaine 10 3 1985
```

   Ce fichier stocke une liste de 3 patients. Ecrivez du code (qui rentrera dans la méthode `public static void main`) qui, à partir de ce fichier (dont le chemin est "`C:/input.txt`"), crée une `ArrayList<Patient>` dont les éléments seront justement les trois patients ci-dessus.

3. **(1 point)** Revenez à la classe `Date` (Question I). Sa méthode `compareTo` lève une exception lorsque l'objet donné en comparaison est `null`. Sachant que NullPointerException est une exception non-contrôlée, ecrivez une nouvelle méthode de test qui vérifie si l'exception se lève correctement.
4. **(1 point)** Quel est le rôle des méthodes setUp et tearDown dans une classe de test de Java ?

Question BONUS (6 points)

Finalement cette question concerne les collections de type TreeSet. Répondez aux questions suivantes :

1. Regardez le diagramme de la classe Date telle que donnée dans la Question I. Supposez que le constructeur et les méthodes `getJour()`, `getMois()`, `getAnnee()` et `equals(Object d)` sont déjà écrites.

(1 point) Détaillez dans cette classe la méthode `int compareTo(Object date)`. Cette méthode doit être implémentée une `Override` de la méthode `int compareTo(Object o)` de l'interface `Comparable`. Voici ce que cette méthode doit faire :

- Si l'objet en paramètre est `null`, alors il faut envoyer une exception de type `NullPointerException` (exception qui hérite de `RuntimeException`).
- Si la date courante (`this`) tombe avant la date en paramètre, alors il faut renvoyer -1.
- Si la date courante (`this`) tombe après la date en paramètre, alors il faut renvoyer 1.
- Si les deux dates sont égales, alors il faut renvoyer 0

Attention : l'objet date en paramètre est déclaré comme étant de type `Object`. Que faut-il faire pour qu'on puisse utiliser les attributs et méthodes de la classe `Date` pour l'objet en paramètre ?

2. Maintenant passons à la classe `ActiviteMedicale`. Disons qu'on veut créer dans la méthode `public static void main` un objet de type `TreeSet<Ordonnance>`. Regardez l'extrait suivant de la JavaDoc de l'interface `SortedSet`, que `TreeSet` implémente :

```
public interface SortedSet<E>
extends Set<E>
```

A `Set` that further provides a *total ordering* on its elements. The elements are ordered using their natural ordering, or by a `Comparator` typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of `SortedMap`.)

All elements inserted into a sorted set must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such elements must be *mutually comparable*: `e1.compareTo(e2)` (or `comparator.compare(e1, e2)`) must not throw a `ClassCastException` for any elements `e1` and `e2` in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a`ClassCastException`.

a. **(1 point)** Est-ce qu'on peut créer ce `TreeSet` dans la classe `ActiviteMedicale` sans modifier la classe `Ordonnance` ? Pourquoi (pourquoi pas) ?
b. **(2 points)** Modifiez la classe `Ordonnance` telle qu'elle implémente l'interface `Comparable`. Ceci veut dire qu'on doit ajouter dans la classe Ordonnance deux méthodes : `boolean equals(Object ordonnance)` et `int compareTo(Object object)`.

   Nous allons comparer les ordonnances seulement par date. Donc on dira que deux ordonnances o1 et o2 sont égales si elles ont été faites à la même date. De plus `o1 < o2` lorsque la date de o1 < la date de o2 (et pareil pour >).
   Utilisez la classe Date de la Question I, avec les méthodes `equals` et `compareTo` (la dernière spécifiée dans la question précédente) pour **détailler la méthode `compareTo` pour la classe Ordonnance**.

c. **(2 points)** Maintenant supposez que les méthodes `equals` et `compareTo` de la classe Ordonnance sont bien implémentées (selon les spécifications de la question antérieure). Qu'est-ce qui se passe si on met le code suivant dans la `méthode public static void main` ?

```
final Date hier = new Date(12, 6, 2018) ;
final Date aujourdhui = new Date(13,6, 2018) ;
TreeSet<Ordonnance> ensembleOrdonnances = new TreeSet<Ordonnace>() ;
final Ordonnance o1 = drLaRoche.prescrire(hier, vaccination,
jeanDupont) ;
final Ordonnance o2 = drLaRoche.prescrire(aujourdhui, vaccination,
jeanDupont) ;
final Ordonnance o3 = drLaRoche.prescrire(hier, antibiotiques,
jeanDupont) ;

ensembleOrdonnances.add(o1) ;
ensembleOrdonnances.add(o2) ;
ensembleOrdonnances.add(o3) ;

System.out.println(ensembleOrdonnances.size()) ;
System.out.println(ensembleOrdonnances) ;
```

L'interface Comparable

`public interface` **`Comparable<T>`**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals.

[...]

Méthode compareTo()

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) ==`

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Throws:

`NullPointerException - if the specified object is null`

`ClassCastException - if the specified object's type prevents it from being compared to this object.`

Méthodes de la classe HashSet

Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Adds the specified element to this set if it is not already present.
void	**clear()** Removes all of the elements from this set.
Object	**clone()** Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	**contains(Object o)** Returns true if this set contains the specified element.
boolean	**isEmpty()** Returns true if this set contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this set.
boolean	**remove(Object o)** Removes the specified element from this set if it is present.
int	**size()** Returns the number of elements in this set (its cardinality).

Methods inherited from class java.util.AbstractSet

equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection

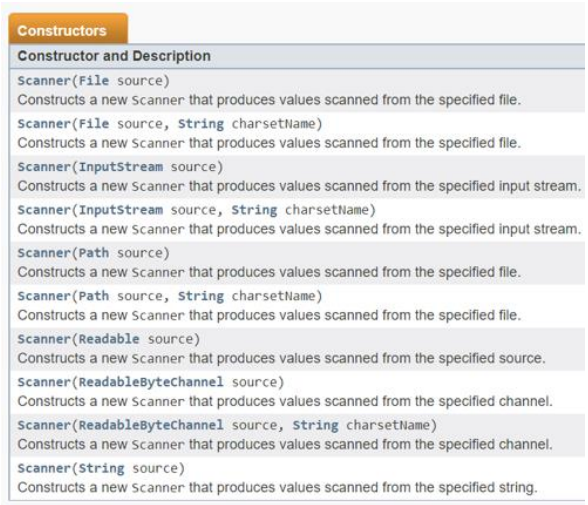
addAll, containsAll, retainAll, toArray, toArray, toString

Méthodes de la classe ArrayList

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void	**trimToSize**() Trims the capacity of this ArrayList instance to be the list's current size.

La classe Scanner



Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(InputStream source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(Path source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new Scanner that produces values scanned from the specified source.
`Scanner(ReadableByteChannel source)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(String source)` Constructs a new Scanner that produces values scanned from the specified string.

hasNext

`public boolean hasNext()`
Returns true if this scanner has another token in its input. This method may block while waiting for input to scan. The scanner does not advance past any input.

Specified by:

hasNext in interface Iterator<String>

Returns:

true if and only if this scanner has another token

Throws:

IllegalStateException - if this scanner is closed

See Also:

`Iterator`

next

`public String next()`
Finds and returns the next complete token from this scanner. A complete token is preceded and followed by input that matches the delimiter pattern. This method may block while waiting for input to scan, even if a previous invocation of `hasNext()` returned `true`.

Specified by:

next in interface Iterator<String>

Returns:

the next token

Throws:

NoSuchElementException - if no more tokens are available

IllegalStateException - if this scanner is closed

See Also:

`Iterator`

nextInt

`public int nextInt()`
Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

InputMismatchException - if the next token does not match the *Integer* regular expression, or is out of range

NoSuchElementException - if input is exhausted

IllegalStateException - if this scanner is closed