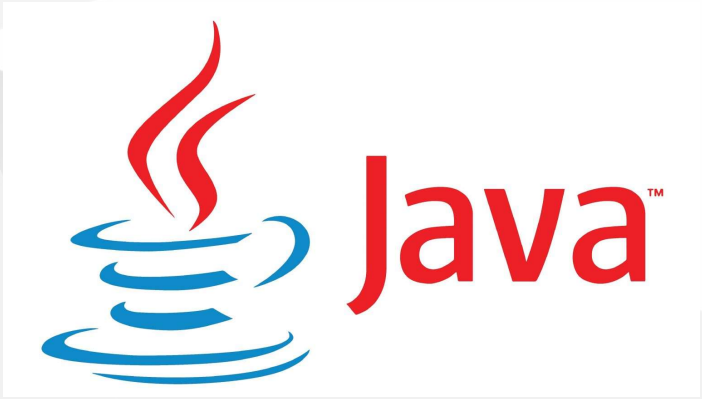
Afpa
se former, avancer



POO / OPP : suite des concepts

LA CLASSE OBJECT
LA CLASSE À L'ORIGINE DE TOUS

LA CLASSE OBJECT

Java est un langage qui ne supporte que l'héritage simple. L'arborescence d'héritage est un arbre dont la racine est la classe Object.

Si le développeur ne précise pas de classe parente dans la déclaration d'une classe, alors la classe hérite implicitement de Object.

- La classe Object fournit des méthodes communes à toutes les classes.
- Certaines de ces méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement.

La méthode equals

- L'implémentation par défaut de equals fournie par Object compare les références entre elles. Si la simple égalité de référence ne suffit pas, il faut alors redéfinir la méthode.
- *Dans certaines classes, cette méthode est déjà redéfinie : pour la classe String par exemple.*

La méthode hashCode

- La méthode hashCode est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation de table de hachage. (Utilisation très technique)

La méthode toString

- C'est une méthode très utile, notamment pour le débugage et la production de log. Elle permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Elle est implicitement appelée par le compilateur lorsqu'on concatène une chaîne de caractères avec un objet. Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.

REDÉFINITION DE LA MÉTHODE EQUALS

L'implémentation par défaut de equals fournie par Object compare les références entre elles. L'implémentation par défaut est donc simplement :

```
public boolean equals(Object obj) {  new *
    return (this == obj);
}
```

Parfois, l'implémentation par défaut peut suffire mais si on souhaite tester de l'égalité d'un objet au-delà de la notion de référence, il faut redéfinir la méthode equals.

L'implémentation de equals doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement :

- Son implémentation doit être réflexive :
  - Pour x non nul, x.equals(x) doit être vrai
- Son implémentation doit être symétrique :
  - Si x.equals(y) est vrai alors y.equals(x) doit être vrai
- Son implémentation doit être transitive :
  - Pour x, y et z non nuls
    - Si x.equals(y) est vrai
    - Et si y.equals(z) est vrai
    - Alors x.equals(z) doit être vrai
- Son implémentation doit être consistante
  - Pour x et y non nuls
  - Si x.equals(y) est vrai alors il doit rester vrai tant que l'état de x et de y est inchangé.
- Si x est non nul alors x.equals(null) doit être faux.

LA CLASSE OBJECT

La méthode finalize

La méthode finalize est appelée par le ramasse-miettes avant que l'objet ne soit supprimé et la mémoire récupérée.

Redéfinir cette méthode donne donc l'opportunité au développeur de déclencher un traitement avant que l'objet ne disparaisse.

La méthode clone

La méthode clone est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet.

Par défaut, elle est déclarée protected car toutes les classes ne désirent pas permettre de cloner une instance.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur Cloneable.

ATTENTION : il effectue un clonage simple et non en profondeur donc si l'objet à cloner contient des références à des objets alors les attributs de ces objets ne seront pas clonés.

LA CLASSE OBJECT

La méthode getClass

La méthode getClass permet d'accéder à l'objet représentant la classe de l'instance.

Cela signifie qu'un programme Java peut accéder par programmation à la définition de la classe d'une instance.

Cette méthode est notamment très utilisée dans des usages avancés impliquant la réflexivité.

Les méthodes de concurrence

La classe Object fournit un ensemble de méthodes qui sont utilisées pour l'échange de signaux (thread) dans la programmation concurrente.

Il s'agit des méthodes notify, notifyAll et wait.

Afpa

LES TYPES DE COMPARAISONS

==, EQUALS, COMPARATOR

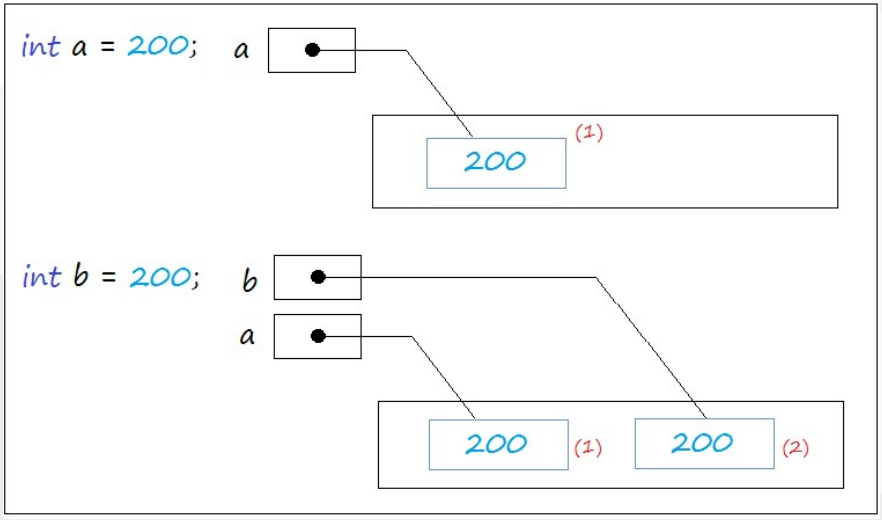
LES TYPES DE COMPARAISON EN JAVA

Comparaison des types primitifs

Avec le type primitif, nous avons seulement une façon de comparer par l'utilisation l'opérateur ==

les types primitifs comparent entre eux via ses valeurs.

Exemple



COMPARAISON LES TYPES DE RÉFÉRENCE AVEC ==

L'utilisation l'opérateur == pour comparer des types de référence

l'opérateur == compare les références de l'objet pour voir si elles font référence à la même instance String.

- Si la valeur des deux références d'objet fait référence à la même instance String
  - le résultat de l'expression booléenne serait True.
- Si, en revanche, la valeur des deux références d'objet fait référence à différentes instances *même si les deux occurrences de String ont la même valeur*.
  - Le résultat de l'expression booléenne serait False.

```java
public class ReferenceEeDemo {
    public static void main(String[] args) {
    // REMARQUE: pour String, deux façons d'initialiser l'objet ci-dessous ne sont pas les mêmes:
        String str1 = "String 1";
        String str2 = new String("String 1");
    // L'opérateur 'new' crée la zone de mémoire (1)
    // contient la chaîne (String) "This is text"
    // Et 's1' est une référence qui pointe vers la zone (1).
        String s1 = new String("This is text");
    // L'opérateur 'new' crée la zone de mémoire (2)
    // Contient la chaîne (String) "This is text"
    // Et 's2' est une référence qui pointe vers la zone (2)
        String s2 = new String("This is text");
    // Utilisez l'opérateur == pour comparer 's1' et 's2'.
    // Les résultats sont faux (false).
    // Il est évidemment différent de ce que vous pensez.
    // La raison en est le type de référence
    // L'opérateur == compare les positions auxquelles ils indiquent.
        boolean e1 = (s1 == s2); // false
    System.out.println("s1 == s2 ? " + e1);

            // Il n'y a aucun opérateur 'new'.
            // Java crée une référence appelée 'obj'
            // Et pointant vers une zone de mémoire sur laquelle 's1' indique.
            Object obj = s1;

            // 2 référence 'obj' et 's1'  indiquent toutes une zone de mémoire.
            // Le résultat renvoie vrai (true).
            boolean e2 = (obj == s1); // true

            System.out.println("obj == s1 ? " + e2);
        }
    }
```

```
s1 == s2 ? false
obj == s1 ? true
```

COMPARAISON LES TYPES DE RÉFÉRENCE AVEC EQUALS

L'utilisation l'opérateur equals(..) pour comparer des types de référence.

La méthode equals() compare la valeur des objets, indépendamment du fait que les deux références d'objet se réfèrent ou non à la même instance si celle-ci a été redéfinie dans la classe (la classe String par exemple).

- Si les deux références d'objet de type String font référence à la même instance String, alors c'est parfait!
- Si les deux références d'objet font référence à deux occurrences String différentes, cela ne fait aucune différence. C'est la « valeur » (c'est-à-dire: le contenu de la chaîne de caractères).
- Si ce n'est pas le cas alors il faut redéfinir la méthode equals

```java
public class StringComparationDemo {

    Run | Debug
    public static void main(String[] args) {

        String s1 = new String(original: "This is text");

        String s2 = new String(original: "This is text");

        // s1 and s2 Comparison, use equals(..)
        boolean e1 = s1.equals(s2);

        // The result is true
        System.out.println("first comparation: s1 equals s2 ? " + e1);

        s2 = new String(original: "New s2 text");

        boolean e2 = s1.equals(s2);

        // The result is false
        System.out.println("second comparation: s1 equals s2 ? " + e2);
    }
}
```

```
first comparation: s1 equals s2 ? true
second comparation: s1 equals s2 ? false
```

REMPLACER LA MÉTHODE EQUALS(OBJECT)

La méthode equals(Object) est donc une méthode disponible dans la classe Object.

Dans certaines situations, vous pouvez remplacer (polymorphisme) cette méthode dans les sous-classes.

```java
public class NumberOfMedalsComparationDemo {

    Run | Debug
    public static void main(String[] args) {

        // La réussite de l'équipe américaine.
        NumberOfMedals american = new NumberOfMedals(goldCount: 40, silverCount: 15, bronzeCount: 15);

        // La réussite de l'équipe japonaise.
        NumberOfMedals japan = new NumberOfMedals(goldCount: 10, silverCount: 5, bronzeCount: 20);

        // La réussite de l'équipe sub- coréenne
        NumberOfMedals korea = new NumberOfMedals(goldCount: 10, silverCount: 5, bronzeCount: 20);

        System.out.println("Medals of American equals Japan ? " + american.equals(japan));

        System.out.println("Medals of Korea equals Japan ? " + korea.equals(japan));
    }
}
```

```
Medals of American equals Japan ? false
Medals of Korea equals Japan ? true
```

```java
public class NumberOfMedals {

    // Le nombre de médailles d'or
    private int goldCount;

    // Le nombre de médailles d'argent.
    private int silverCount;

    // Le nombre de médailles de bronze
    private int bronzeCount;

    public NumberOfMedals(int goldCount, int silverCount, int bronzeCount) {
        this.goldCount = goldCount;
        this.silverCount = silverCount;
        this.bronzeCount = bronzeCount;
    }

    public int getGoldCount() {
        return goldCount;
    }

    public int getSilverCount() {
        return silverCount;
    }

    public int getBronzeCount() {
        return bronzeCount;
    }

    // Remplacez la méthode equals(Object) de la classe Objet
    @Override
    public boolean equals(Object other) {
        // Si other = null, le résultat renvoie faux (false).
        if (other == null) {
            return false;
        }
        // Si 'other' n'est pas le type de NumberOfMedals
        // le résultat renvoie faux (false).
        if (!(other instanceof NumberOfMedals)) {
            return false;
        }

        NumberOfMedals otherNoM = (NumberOfMedals) other;

        if (this.goldCount == otherNoM.goldCount && this.silverCount == otherNoM.silverCount
                && this.bronzeCount == otherNoM.bronzeCount) {
            return true;
        }
        return false;
    }
}
```

Afpa

COMPARER DEUX OBJETS
INTERFACE COMPARABLE

INTERFACE COMPARABLE

L'interface Comparable est utilisée pour comparer un objet de la même classe avec une instance de cette classe, elle fournit l'ordre des données pour les objets de la classe définie par l'utilisateur.

La classe doit implémenter l'interface java.lang.Comparable pour comparer son instance, elle fournit la méthode compareTo qui prend un paramètre de l'objet de cette classe.

DES OBJETS PEUVENT ÊTRE COMPARÉS ENTRE EUX (COMPARABLE)

Vous écrivez une classe qui simule un acteur (Actor).

Vous souhaitez organiser l'ordre des acteurs selon le principe de comparer le nom de famille (lastName) d'abord, comparez le prénom (firstName)

Pour cela, vous devez implémenter l'interface Comparable dans votre classe Actor et surcharger la méthode compareTo

```java
// Pour comparer les uns avec les autres,
// la classe d'Actor doit implémenter une interface comparable.
public class Actor implements Comparable<Actor> {

    private String firstName;
    private String lastName;

    public Actor(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    // Comparez cet Actor avec un autre actor.
    // Selon le principe de comparaison du dernier nom d'abord,
    // puis comparez FirstName.
    @Override
    public int compareTo(Actor other) {

        // Comparez deux chaines.
        int value = this.lastName.compareTo(other.lastName);

        // Si lastName de deux objets ne sont pas égaux
        if (value != 0) {
            return value;
        }
        // Si lastName de deux objets sont le même.
        // Ensuite, comparez fistName.
        value = this.firstName.compareTo(other.firstName);
        return value;
    }

}
```

COLLECTIONS
COLLECTIONNEZ-LES TOUS !!

LES COLLECTIONS

On a découvert les tableaux et on peut vite se rendre compte de leurs limites avec l'utilisation des objets.

- Leur taille fixe par exemple.

Les collections vont apporter une souplesse d'utilisation.

Parmi les collections, on trouve :

- Les listes (lists)
- Les ensembles (sets)
- Les tableaux associatifs (maps)

Toutes ces collections sont génériques. On peut donc créer que des collections d'objets.

Si on veut créer une collection d'un type primitif. Il faut utiliser la classe enveloppe du type :

- Integer pour int par exemple

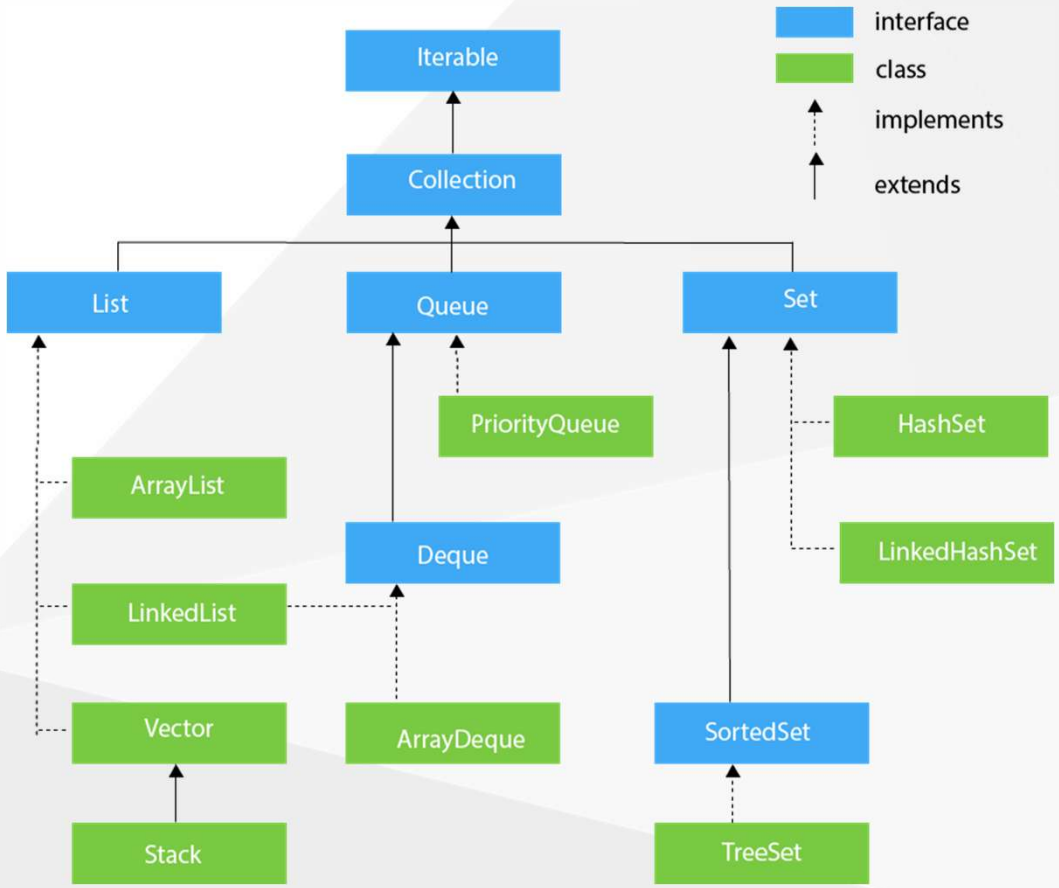
LISTS

Une liste est une collection ordonnée d'éléments.

Il existe différentes façons d'implémenter des listes dont les performances sont optimisées soit pour les accès aléatoires aux éléments soit pour les opérations d'insertion et de suppression d'éléments.

L'interface Collection dont hérite toutes les autres interfaces pour les listes, hérite elle-même de Iterable.

- Cela signifie que toutes les classes et toutes les interfaces servant à représenter des listes dans le Java Collections Framework peuvent être parcourues avec une structure de for amélioré (foreach).



LISTS

La classe ArrayList

La classe java.util.ArrayList est une implémentation de l'interface List.

Elle stocke les éléments de la liste sous la forme de blocs en mémoire.

- Cela signifie que la classe ArrayList est très performante pour les accès aléatoires en lecture aux éléments de la liste.
- Par contre, les opérations d'ajout et de suppression d'un élement se font en temps linéaire.
- Elle est donc moins performante que la classe LinkedList sur ce point.

La classe LinkedList

La classe java.util.LinkedList est une implémentation de l'interface List.

Sa représentation interne est une liste doublement chaînée.

- Cela signifie que la classe LinkedList est très performante pour les opérations d'insertion et de suppression d'éléments.
- Par contre, l'accès aléatoire en lecture aux éléments se fait en temps linéaire.
- Elle est donc moins performante que la classe ArrayList sur ce point.

JAVA.UTIL.ARRAY LIST

La classe ArrayList peut être utilisée pour créer des listes d'objets.

- redimensionnable dynamiquement, ce qui signifie que sa taille peut changer pendant l'exécution du programme.

```
ArrayList< String> names = new ArrayList< String>();
```

Une ArrayList peut contenir n'importe quel type d'objet

- L'ajout d'un type de données entre crochets fait que Java vérifie que vous affectez les types appropriés à une liste.

Le constructeur par défaut crée une **ArrayList** avec une capacitée de 10 éléments mais :

```
ArrayList< String> noms = new ArrayList< String>(20);
```

la méthode **Collections.sort(collection)** permet de trier une ArrayList

Méthode	Description
public void add(Object) public void add(int, Object)	Ajouter un élément à une ArrayList; la version par défaut ajoute un élément au prochain emplacement disponible; une version surchargée vous permet de spécifier une position à laquelle nous voulons ajouter l'élément
public void remove(int)	Supprimer un élément d'une ArrayList à un emplacement spécifié
public void set(int, Object)	Modifier un élément à un emplacement spécifié dans une ArrayList
Object get(int)	Récupérer un élément d'un emplacement spécifié dans une ArrayList
public int size()	Renvoyer la taille actuelle de ArrayList

PARCOURS D'UNE ARRAYLIST

Création et parcours d'un ArrayList et d'une List

```java
/**
 * Création d'une ArrayList et manipulation.
 */
List<String> liste = new ArrayList<>();

ArrayList<String> couleurs = new ArrayList<>();
couleurs.add("Red");
couleurs.add("Blue");
couleurs.add("Yellow");
couleurs.add("Green");

// ajout d'element dans la liste
liste.add("une première chaîne");
liste.add("une troisième chaîne");

// demande de la taille de la liste
System.out.println(liste.size()); // 2

// insertion d'un élément
liste.add( index: 1,  element: "une seconde chaîne");

// affichage de la taille
System.out.println(liste.size()); // 3

// Parcours de la liste
for (String s : liste) {
    System.out.println(s);
}

for(String s : couleurs) {
    System.out.println(s);
}
```

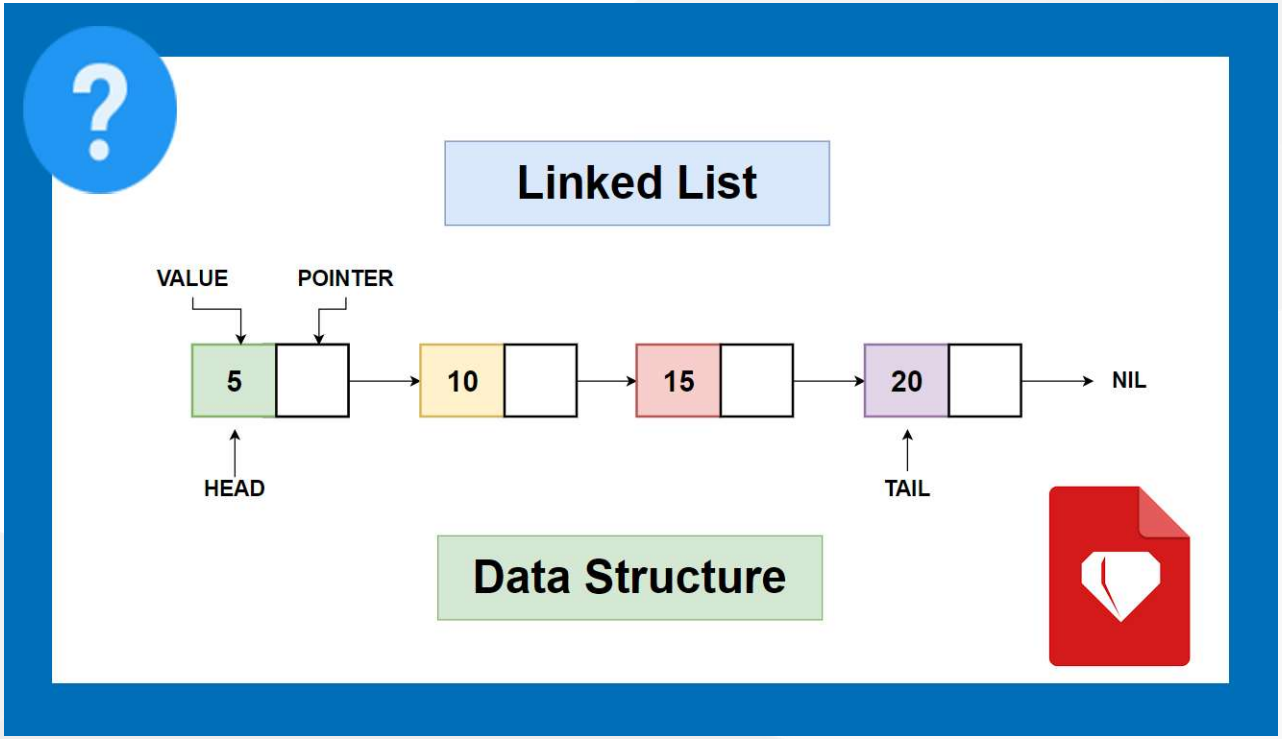
JAVA.UTIL.LINKEDLIST

LinkedList sont des structures de données linéaires où les éléments ne sont pas stockés dans des emplacements contigus et chaque élément est un objet séparé avec une partie de données et une partie d'adresse.

Les éléments sont liés à l'aide de pointeurs et d'adresses.

Chaque élément est appelé un nœud.

La classe **LinkedList** hérite de **AbstractSequentialList** et implémente l'interface **List**.



JAVA.UTIL.LINKEDLIST

Méthode	Description
void add(int index, Object element)	Cette méthode insère l'élément spécifié à la position spécifiée dans cette liste.
boolean add(Object o)	Cette méthode ajoute l'élément spécifié à la fin de cette liste.
boolean addAll(Collection c)	Cette méthode ajoute tous les éléments de la collection spécifiée à la fin de cette liste, dans l'ordre dans lequel ils sont renvoyés par l'itérateur de la collection spécifiée.
boolean addAll(int index, Collection c)	Cette méthode Insère tous les éléments de la collection spécifiée dans cette liste, en commençant à la position spécifiée
void addLast(Object o)	Cette méthode ajoute l'élément spécifié à la fin de cette liste.
Object remove()	Cette méthode récupère et supprime la tête (premier élément) de la liste.
int size()	Cette méthode renvoie le nombre d'éléments dans cette liste.
Etc…	

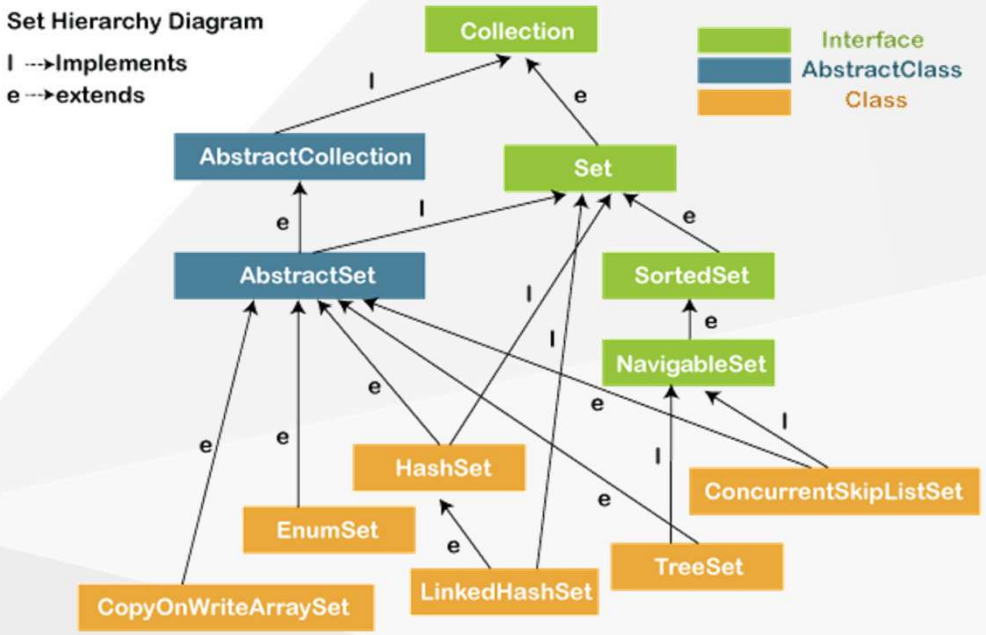
SET

Les ensembles (set) sont des collections qui ne contiennent aucun doublon. Deux éléments e1 et e2 sont des doublons si :

- e1.equals(e2) == true ou si e1 vaut null et e2 vaut null.

Pour contrôler l'unicité, 3 implémentations :

- TreeSet
  - Particularité : conserve toujours ses éléments triés.
- HashSet
  - utilise un code de hachage (hash code) pour contrôler l'unicité de ces éléments. Un code de hachage est une valeur associée à objet.
- LinkedHashSet.
  - utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des éléments sera le même que l'ordre d'insertion.



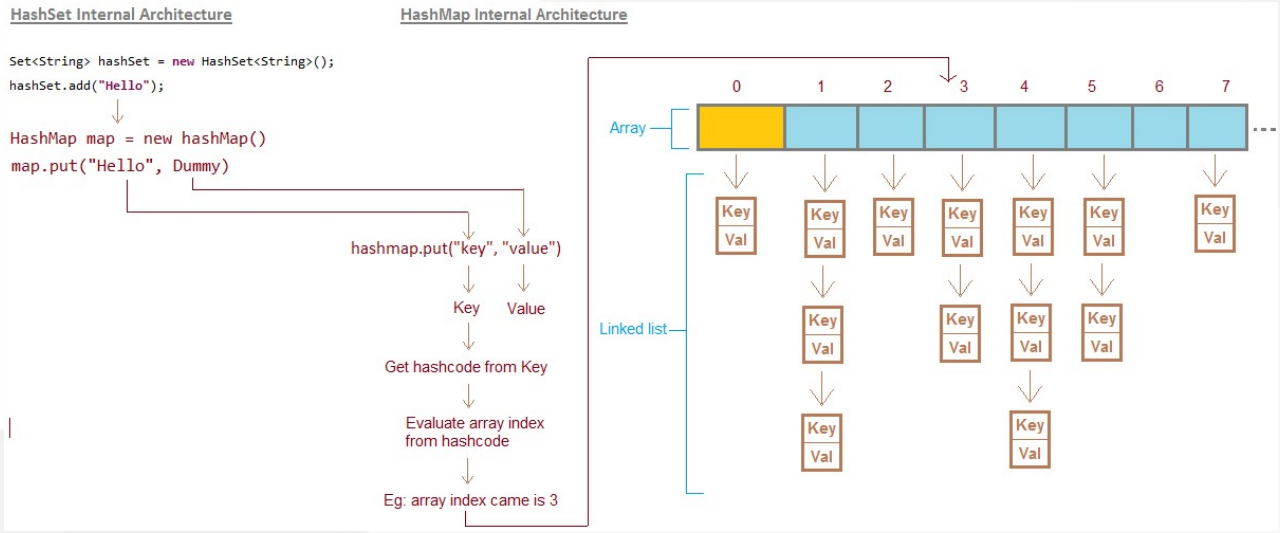
JAVA.UTIL.HASHSET

Une structure de données bien connue pour trouver des objets rapidement est la table de hachage (**HashTable**).

Une table de hachage calcule un entier, appelé le code de hachage pour chaque objet.

Un code de hachage est en quelque sorte dérivé des champs d'instance d'un objet, de préférence de telle sorte que les objets avec des données différentes génèrent des codes différents.

En Java, les tables de hachage sont implémentées en tant que tableaux de listes chaînées (**LinkedList**). Chaque liste s'appelle une alvéole (en anglais, buckets ou slots).



JAVA.UTIL.HASHSET

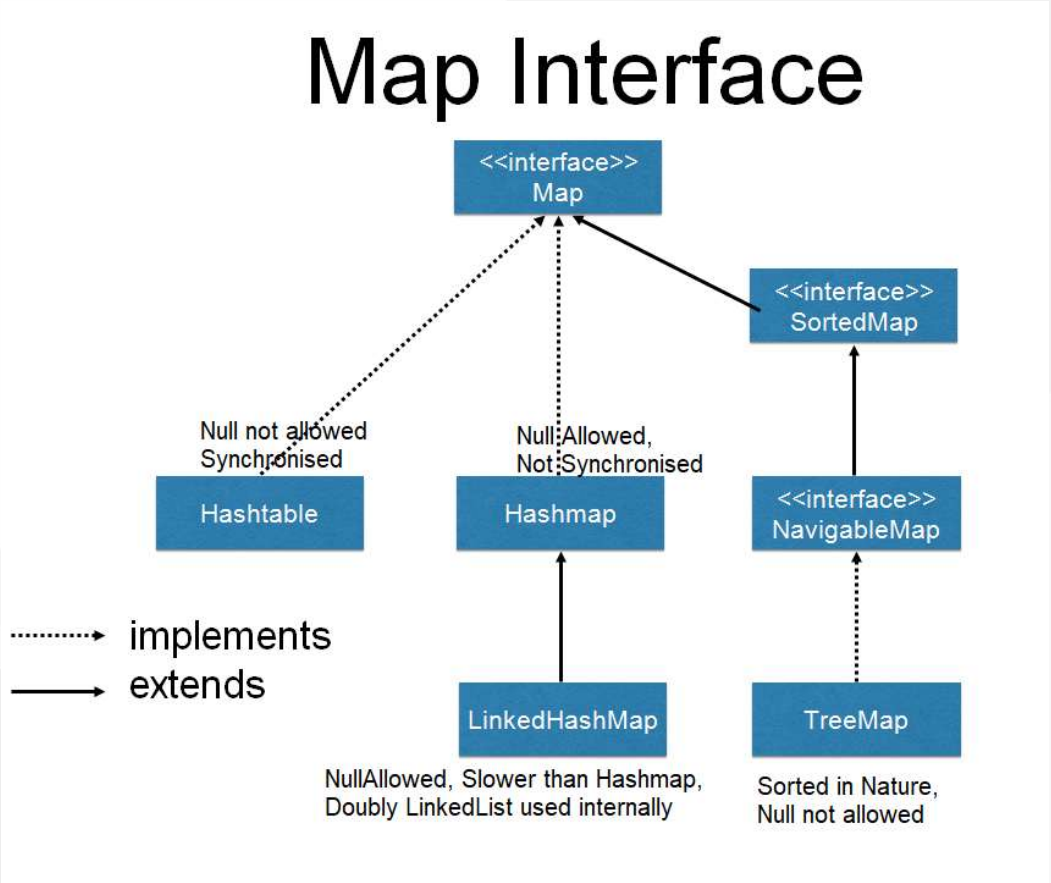
Méthode	Description
HashSet()	Ce constructeur construit un HashSet par défaut.
HashSet(Collection c)	Ce constructeur initialise le hash set en utilisant les éléments de la collection c.
boolean add(Object o)	Ajouter l'élément spécifié à HashSet s'il n'est pas déjà présent.
boolean isEmpty()	Retourner true si cet HashSet ne contient aucun élément.
Iterator iterator()	Retourner un itérateur sur les éléments de cet HashSet.
boolean remove(Object o)	Supprimer l'élément spécifié de cet HashSet s'il est présent.
int size()	Retourner le nombre d'éléments.
void clear()	Supprimer tous les éléments.

LES MAPS

Les *map* permet d'associer une clé à une valeur. Elle ne peut pas contenir de doublon de clés.

Le Java Collections Framework fournit plusieurs implémentations de tableaux associatifs :

- TreeMap,
- HashMap,
- LinkedHashMap.



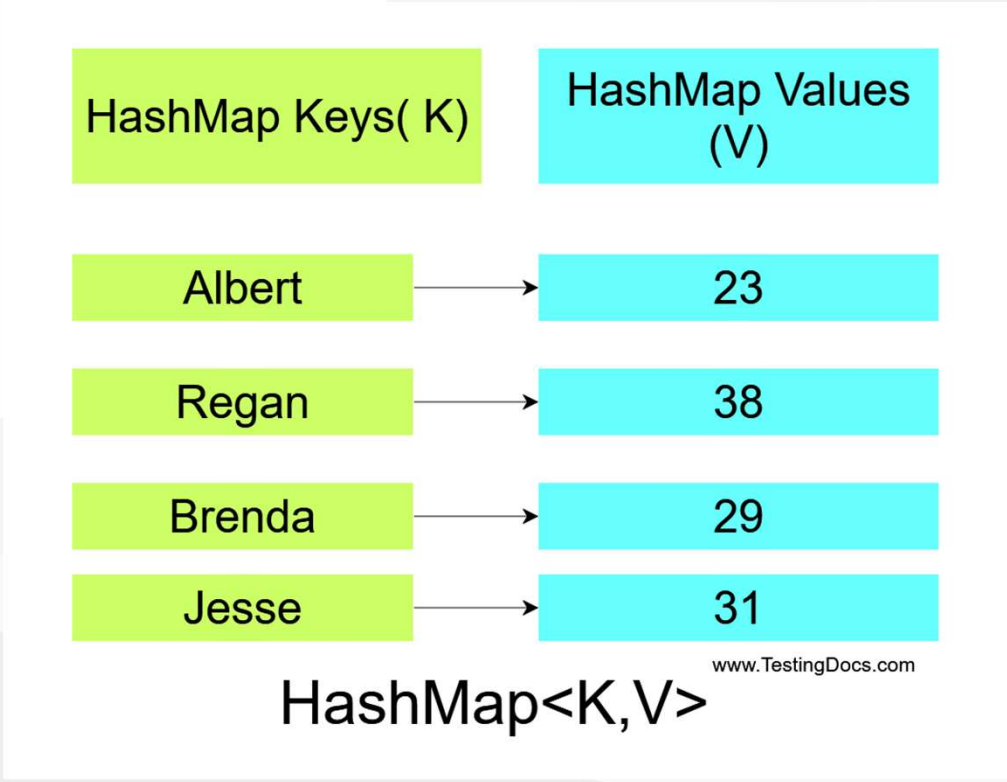
JAVA.UTIL.HASHMAP

En général, vous avez des informations clés et vous souhaitez rechercher l'élément associé.

La structure des données HashMap sert cet objectif.

Un HashMap stocke des paires clé/valeur.

Vous pouvez trouver une valeur si vous fournissez la clé.



JAVA.UTIL.HASHMAP

Méthode	Description
HashMap<K, V>()	Ce constructeur construit un HashMap par défaut.
HashMap<K, V>(Map m)	Ce constructeur initialise un HashMap en utilisant les éléments de l'objet Map donné m.
HashMap<K, V>(int capacity)	Ce constructeur initialise la capacité de HAshMap sur la valeur entière donnée, capacity.
void clear()	Supprimer tous les mappages de cette map.
boolean containsKey(Object key)	Renvoyer true si cette map contient un mappage pour la clé spécifiée.
boolean containsValue(Object value)	Renvoyer true si cette map mappe une ou plusieurs clés sur la valeur spécifiée.
Object get(Object key)	Renvoyer la valeur à laquelle la clé spécifiée est mappée dans cette map, ou null si la carte ne contient aucune correspondance pour cette clé.
Object put(Object key, Object value)	Associer la valeur spécifiée à la clé spécifiée dans cette map.
boolean isEmpty()	Renvoyer true si cette map ne contient aucune correspondance clé-valeur.
Object remove(Object key)	Supprimer le mappage de cette clé de cette map si elle est présente..
int size()	Renvoyer le nombre de mappages clé-valeur dans cette map
Etc…	

LA CLASSE OUTIL COLLECTIONS

La classe java.util.Collections est une classe outils qui contient de nombreuses méthodes pour les listes, les ensembles et les tableaux associatifs.

Elle contient également des attributs de classes correspondant à une liste, un ensemble et un tableau associatif vides et immutables.

```java
List<String> liste = new ArrayList<>();
Collections.addAll(liste, "un", "deux", "trois", "quatre");

// La chaîne a plus grande dans la liste : "un"
String max = Collections.max(liste);
System.out.println(max);

// Inverse l'ordre de la liste
Collections.reverse(liste);
// [quatre, trois, deux, un]
System.out.println(liste);

// Trie la liste
Collections.sort(liste);
// [deux, quatre, trois, un]
System.out.println(liste);

// Recherche de l'index de la chaîne "deux" dans la liste triée : 0
int index = Collections.binarySearch(liste, "deux");
System.out.println(index);

// Remplace tous les éléments par la même chaîne
Collections.fill(liste, "même chaîne partout");
// [même chaîne partout, même chaîne partout, même chaîne partout, même chaîne
// partout]
System.out.println(liste);

// Enveloppe la liste dans une liste qui n'autorise plus a modifier son contenu
liste = Collections.unmodifiableList(liste);

// On tente de modifier une liste qui n'est plus modifiable
liste.add("Test"); // ERREUR à l'exécution : UnsupportedOperationException
```

Afpa

LA CLASSE DATE

LES DATES EN JAVA

LES DATES

Java.util.Date

Elle est la première classe à être apparue pour représenter une date. Elle comporte de nombreuses limitations :

- Il n'est pas possible de représenter des dates antérieures à 1900
- Elle ne supporte pas les fuseaux horaires
- Elle ne supporte que les dates pour le calendrier grégorien
- Elle ne permet pas d'effectuer des opérations (ajout d'un jour, d'une année…)

L'API Date/Time

Depuis Java 8, une nouvelle API a été introduite pour représenter les dates, le temps et la durée. Toutes ces classes ont été regroupées dans la package java.time.

Pour les dates, les classes LocalDate, LocalTime et LocalDateTime permettent de représenter respectivement une date, une heure et une date et heure

On peut facilement passer d'un type à une autre.

- Par exemple la méthode LocalDate.atTime permet d'ajouter une heure à une date, créant ainsi une instance de LocalDateTime. Toutes les instances de ces classes sont immutables.

Si on veut avoir l'information de la date ou de l'heure d'aujourd'hui, on peut créer une instance grâce à la méthode now.

L'API DATE/TIME

Les classes Year et YearMonth

Les classe Year et YearMonth permettent de manipuler les dates et d'obtenir des informations intéressantes à partir de l'année ou du mois et de l'année.

```java
Year year = Year.of( isoYear: 2004);

// année bissextile ?
boolean isLeap = year.isLeap();

// 08/2004
YearMonth yearMonth = year.atMonth(Month.AUGUST);

// 31/08/2004
LocalDate localDate = yearMonth.atEndOfMonth();
```

La classe Instant

La classe Instant représente un point dans le temps.

Contrairement aux classes précédentes qui permettent de représenter les dates pour les humains, la classe Instant est adaptée pour réaliser des traitements de données temporelles.

```java
Instant maintenant = Instant.now();
Instant epoch = Instant.ofEpochSecond(0); // 01/01/1970 00:00:00.000

Instant uneMinuteDansLeFuture = maintenant.plusSeconds( secondsToAdd: 60);

long unixTimestamp = uneMinuteDansLeFuture.getEpochSecond();
```

L'API DATE / TIME

La Période

Il est possible de définir des périodes grâce à des instances de la classe Period.

Une période peut être construite directement ou à partir de la différence entre deux instances de type Temporal. Il est ensuite possible de modifier une date en ajoutant ou soustrayant une période.

```
YearMonth moisAnnee = Year.of( isoYear: 2000).atMonth(Month.APRIL); // 04/2000

// période de 1 an et deux mois
Period periode = Period.ofYears(1).plusMonths( monthsToAdd: 2);

YearMonth moisAnneePlusTard = moisAnnee.plus(periode); // 06/2001

Period periode65Jours = Period.between(LocalDate.now(),
        LocalDate.now().plusDays( daysToAdd: 65));
```

La durée

La durée est représentée par une instance de la classe Duration. Elle peut être obtenue à partir de deux instances de Instant.

```
Instant debut = Instant.now();

// ... traitement à mesurer

Duration duree = Duration.between(debut, Instant.now());
System.out.println(duree.toMillis());
```

FORMATAGE DES DATES

Pour formater une date pour l'affichage, il est possible d'utiliser la méthode format déclarée dans les classes LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Year et YearMonth.

Le format de représentation d'une date et/ou du temps est défini par la classe DateTimeFormatter.

```
// 01/09/2010 16:30
LocalDateTime dateTimeExemple = LocalDateTime.of( year: 2010,
        Month.SEPTEMBER, dayOfMonth: 1, hour: 16, minute: 30);

// En utilisant des formats ISO de dates
System.out.println(dateTimeExemple.format(DateTimeFormatter.BASIC_ISO_DATE));
System.out.println(dateTimeExemple.format(DateTimeFormatter.ISO_WEEK_DATE));
System.out.println(dateTimeExemple.format(DateTimeFormatter.ISO_DATE_TIME));

DateTimeFormatter datePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy");
// 01/09/2010
System.out.println(dateTimeExemple.format(datePattern));

DateTimeFormatter dateTimePattern = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
// 01/09/2010 16:30
System.out.println(dateTimeExemple.format(dateTimePattern));

// 1 septembre 2010
DateTimeFormatter frenchDatePattern = DateTimeFormatter
        .ofPattern( pattern: "d MMMM yyyy", Locale.FRANCE);
System.out.println(dateTimeExemple.format(frenchDatePattern));
```

LECTURES DES DATES

Pour transformer une chaîne de caractères en date (notamment pour obtenir une date à partir de la saisie d'un utilisateur ou de la lecture d'un fichier), il est possible d'utiliser la méthode parse déclarée dans les classes LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Year et YearMonth.

Le format d'une date et/ou du temps est défini par la classe DateTimeFormatter.

```
String exemple = "02/06/2010 12:35";

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");

Year yearExemple = Year.parse(exemple, dtf);
System.out.println(yearExemple);

YearMonth yearMonthExemple = YearMonth.parse(exemple, dtf);
System.out.println(yearMonthExemple);

LocalDate localDateExemple = LocalDate.parse(exemple, dtf);
System.out.println(localDateExemple);

LocalDateTime localDateTimeExemple = LocalDateTime.parse(exemple, dtf);
System.out.println(localDateTimeExemple);
```