

Langage Java Chapitre 5

Exemple de collection: la classe Vector.

Exemple de dictionnaire: la classe HashMap.

Les itérateurs.

Les exceptions.

Java propose de nombreuses classes pour implémenter les collections, on peut citer les classes **ArrayList**, **LinkedList** et **Vector**.

1. La classe Vector

La classe **Vector** fait partie du package **java.util**. Cette classe est faite pour gérer une **collection d'objets** sous la forme d'un tableau dynamique, c'est-à-dire un tableau dont la taille peut s'accroître (ou diminuer) en fonction des besoins, contrairement à celle des tableaux standards dont la taille est définie statiquement lors de l'instanciation. Chaque objet du vecteur est repéré par un indice.

1.1 Extraits de la documentation :

Les champs :

Field Summary	
protected int	capacityIncrement The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
protected int	elementCount The number of valid components in this Vector object.
protected Object[]	elementData The array buffer into which the components of the vector are stored.

Les constructeurs :

Constructor Summary	
Vector ()	Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
Vector (Collection c)	Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
Vector (int initialCapacity)	Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector (int initialCapacity, int capacityIncrement)	Constructs an empty vector with the specified initial capacity and capacity increment.

Les méthodes :

Method Summary	
void	<code>add</code> (int index, <code>Object</code> element) Inserts the specified element at the specified position in this Vector.
boolean	<code>add</code> (<code>Object</code> o) Appends the specified element to the end of this Vector.
boolean	<code>addAll</code> (<code>Collection</code> c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	<code>addAll</code> (int index, <code>Collection</code> c) Inserts all of the elements in in the specified Collection into this Vector at the specified position.
void	<code>addElement</code> (<code>Object</code> obj) Adds the specified component to the end of this vector, increasing its size by one.
int	<code>capacity</code> () Returns the current capacity of this vector.
void	<code>clear</code> () Removes all of the elements from this Vector.
<code>Object</code>	<code>clone</code> () Returns a clone of this vector.
boolean	<code>contains</code> (<code>Object</code> elem) Tests if the specified object is a component in this vector.
boolean	<code>containsAll</code> (<code>Collection</code> c)
void	<code>copyInto</code> (<code>Object</code> [] anArray) Copies the components of this vector into the specified array.
<code>Object</code>	<code>elementAt</code> (int index) Returns the component at the specified index.
<code>Enumeration</code>	<code>elements</code> () Returns an enumeration of the components of this vector.
void	<code>ensureCapacity</code> (int minCapacity) Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean	<code>equals</code> (<code>Object</code> o) Compares the specified Object with this Vector for equality.
<code>Object</code>	<code>firstElement</code> () Returns the first component (the item at index 0) of this vector.
<code>Object</code>	<code>get</code> (int index) Returns the element at the specified position in this Vector.
int	<code>hashCode</code> () Returns the hash code value for this Vector.
int	<code>indexOf</code> (<code>Object</code> elem) Searches for the first occurrence of the given argument, testing for equality using the <code>equals</code> method.
int	<code>indexOf</code> (<code>Object</code> elem, int index) Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the <code>equals</code> method.

void	<code>insertElementAt(Object obj, int index)</code> Inserts the specified object as a component in this vector at the specified index.
boolean	<code>isEmpty()</code> Tests if this vector has no components.
<code>Object</code>	<code>lastElement()</code> Returns the last component of the vector.
int	<code>lastIndexOf(Object elem)</code> Returns the index of the last occurrence of the specified object in this vector.
int	<code>lastIndexOf(Object elem, int index)</code> Searches backwards for the specified object, starting from the specified index, and returns an index to it.
<code>Object</code>	<code>remove(int index)</code> Removes the element at the specified position in this Vector.
boolean	<code>remove(Object o)</code> Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged.
boolean	<code>removeAll(Collection c)</code> Removes from this Vector all of its elements that are contained in the specified Collection.
void	<code>removeAllElements()</code> Removes all components from this vector and sets its size to zero.
boolean	<code>removeElement(Object obj)</code> Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	<code>removeElementAt(int index)</code> Deletes the component at the specified index.
protected void	<code>removeRange(int fromIndex, int toIndex)</code> Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
boolean	<code>retainAll(Collection c)</code> Retains only the elements in this Vector that are contained in the specified Collection.
<code>Object</code>	<code>set(int index, Object element)</code> Replaces the element at the specified position in this Vector with the specified element.
void	<code>setElementAt(Object obj, int index)</code> Sets the component at the specified index of this vector to be the specified object.
void	<code>setSize(int newSize)</code> Sets the size of this vector.
int	<code>size()</code> Returns the number of components in this vector.
<code>List</code>	<code>subList(int fromIndex, int toIndex)</code> Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
<code>Object[]</code>	<code>toArray()</code> Returns an array containing all of the elements in this Vector in the correct order.
<code>Object[]</code>	<code>toArray(Object[] a)</code> Returns an array containing all of the elements in this Vector in the correct order.
<code>String</code>	<code>toString()</code> Returns a string representation of this Vector, containing the String representation of each element.
void	<code>trimToSize()</code> Trims the capacity of this vector to be the vector's current size.

1.2 Exploitation de la documentation

- **Les champs :**

Le champ `elementCount` contient le nombre d'objets dans le Vector.

Ce champ est **protected** et il doit être lu par la méthode `size()`.

- **Les constructeurs :**

Si nous choisissons le premier constructeur, une ligne de code telle que :

```
Vector LesAbonnes = new Vector();
```

instancie un objet LesAbonnes sur la classe Vector, sans paramètre avec le premier constructeur.

LesAbonnes est un objet Vector qui peut gérer une liste d'objets. Les objets gérés peuvent être de tous types prédéfinis de java : Integer, String ou des objets plus complexes comme abonne que nous avons créé dans le chapitre précédent.

On peut ajouter des objets dans le Vector, en supprimer, relire la liste, etc

La taille (capacité) d'un Vector s'adapte automatiquement au nombre d'objets qu'il contient : il grandit lorsqu'on ajoute un nouvel objet, il diminue lorsqu'on en enlève un.

Attention :

A partir de Java 1.5 il faut préciser le type des objets contenus dans le Vector lors de la déclaration et de l'instanciation. Par exemple, pour un Vector de String :

```
Vector<String> LesNoms;
```

```
LesNoms = new Vector< String >();
```

- Les méthodes

AddElement(Object obj)

La ligne de code **LesAbonnes.addElement(abon1);**

ajoute un objet abon1 dans le Vector, abon1 est par exemple un objet Abonne.

On peut ajouter autant d'éléments que l'on veut tant que la capacité du Vector n'est pas atteinte. Le Vector s'agrandit alors.

get(int index)

La ligne de code **Abonne CestLui= LesAbonnes.get(n1);**

Relit un objet à l'indice n1 dans le Vector. Avec java inférieur à 1.5 un transtypage (ou cast)

Abonne CestLui= (Abonne)LesAbonnes.get(n1); est nécessaire pour relire correctement l'objet.

1.3 Exercice : étudier les méthodes suivantes de la documentation :

size(), add(), clear(), contains(), equals(), isEmpty(), remove(), toString()

1.4 Exercice : Création d'un annuaire en utilisant la classe Vector.

Reprendre les classe individu et abonne du chapitre 4.

```
package agenda;
import java.io.*;
```

```
public class Individu {
```

```

private String Nom;
private String Prenom;
private int Age ;
public Individu(String nom,String prenom, int age ){
    Nom = nom;
    Prenom = prenom;
    Age = age;
}
public void afficheIndividu(){
    System.out.println("Nom : "+Nom+ " Prénom : "+Prenom);
}
public String getNom() {
    return Nom ;
}
public String getPrenom() {
    return Prenom ;
}
public int getAge() {
    return Age ;
}
}

```

```

package agenda;
import java.io.*;
public class Abonne extends Individu {
    private String Telephone;
    public Abonne(String nom, String prenom, int age, String telephone){
        super(nom, prenom, age); // constructeur de la super classe
        Telephone= telephone;
    }
    public void afficheAbonne(){
        afficheIndividu() ; //appel de la méthode afficheIndividu() de la classe mère
        System.out.println("Téléphone : "+telephone);
    }
}

```

Ecrire une classe Annuaire.java en complétant le code suivant :

```

package agenda;
import java.util.*;
import java.io.*;

public class Annuaire {

    // Déclarer un objet sur la classe Vector
    private Vector<Abonne> LesAbonnes;
    // Le constructeur d'Annuaire crée le Vector, écrire ici le constructeur d'Annuaire
    .....

    // ajouter un abonne dans l'annuaire

```

```

public void Ajouter(Abonne abo1)
{
    .....
}

// relire l'annuaire avec affichage sur l'écran
public void Relire()
{
    .....
}

// Pour rechercher un abonné par son nom
public String RechercherParLeNom(String CeNom)
{
    .....
}

// Pour rechercher un abonné par son numéro
public String RechercherParLeNumero(String CeNumero)
{
    .....
}
}

```

Le constructeur de la classe Annuaire crée un Vector pour stocker des objets abonne.


Les méthodes de la classe sont :

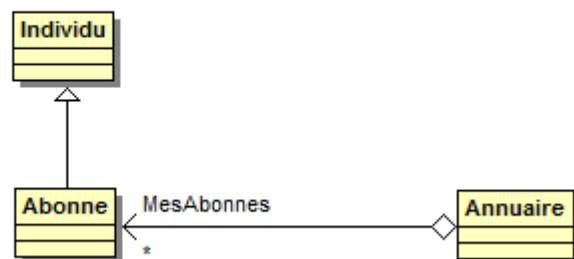
Ajouter un abonné
 Relire la liste des abonnés
 RechercherParLeNom
 RechercherParLeNumero


Ecrire un programme de test avec une méthode main : testAnnuaire.java

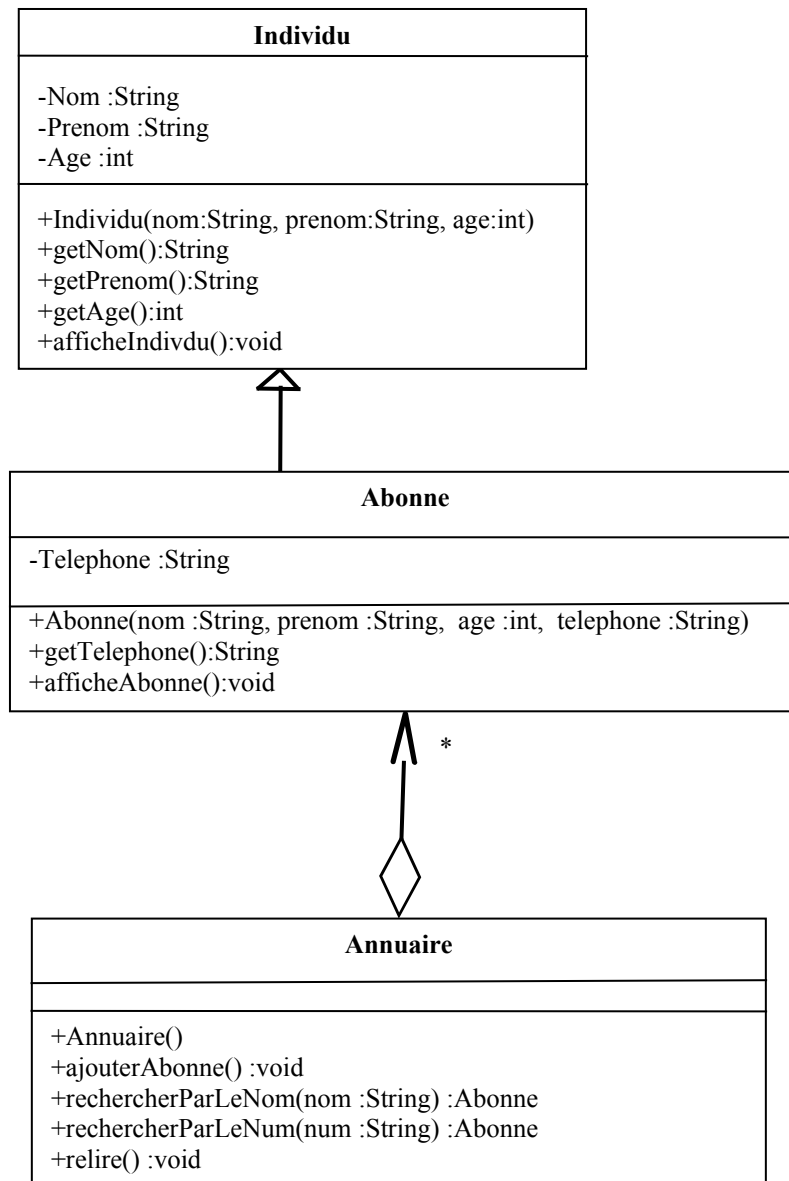
Diagramme de classes UML simplifié

Les flèches entre les classes indiquent des associations ou liens entre-elles.

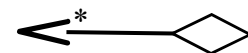
- La flèche  indique que la classe Abonne **hérite** de la classe Individu
 On dit aussi que :
- La classe Abonne est la spécialisation de la classe Individu.
 - La classe Abonne est la sous-classe de la super classe Individu.
 - La classe Abonne est la classe fille, Individu la classe mère.
 - La classe Abonne est la classe étendue, Individu la classe de base.



La flèche  indique une association unidirectionnelle, ici 1 annuaire peut contenir un nombre quelconque (*) d'abonnés. L'annuaire est personnel et le détenteur y mémorise les abonnés qu'il souhaite. Le losange précise une agrégation: «un annuaire contient des abonnés».



Par convention, on n'indique pas l'implémentation de l'association



L'implémentation de cette association -une agrégation- est réalisée ici à l'aide d'un vecteur.
La valeur * représente la cardinalité de l'association, * indique un nombre quelconque : 1 annuaire contient un certain nombre d'abonnés.

2. Les itérateurs

Un itérateur permet de parcourir une collection d'objets.

Un itérateur est une instance de la classe **Iterator**, il est instancié en appelant la méthode iterator() de la collection à traiter.

Exemple

```

Vector<Abonne> LesAbonnes;
Iterator<Abonne> it = LesAbonnes.iterator() ;

while ( it.hasNext() ) {
    Abonne abonne = it.next() ;
    // On traite maintenant l'objet abonne
    ...
}

```

Travail

- a) Repérer l'implémentation de l'interface **Iterable** dans la classe **Vector**. Etudier la documentation de la méthode **iterator()** ainsi que celles des méthodes **hasNext()** et **next()** de l'interface **Iterator**.
- b) Modifier le programme 1.4 afin d'utiliser un itérateur pour balayer les objets du vecteur lorsque c'est nécessaire.

3. La classe HashMap

La classe **HashMap** fait partie du package **java.util**. Cette classe est faite pour gérer une **collection d'objets**, un peu comme la classe **Vector**. A la différence de la classe **Vector**, où les objets gérés dans le **Vector** sont repérés par des indices, les objets gérés dans le **HashMap** le sont par une **clé**. En algorithmie la classe **HashMap** est assimilée à un dictionnaire : les clés sont les mots, les objets sont les définitions: on recherche une définition à partir d'un mot.

Les lignes de code suivantes montrent l'instanciation d'objets sur la classe **HashMap** , ainsi que la mise en œuvre de certaines méthodes :

```
HashMap<String, Abonne> Hashannu1 = new HashMap<String, Abonne> () ;
```

Un tel **HashMap** permet d'associer le nom de l'abonné (le **String**) comme clef pour chaque Abonné stocké.

A chaque objet inséré dans un **HashMap** doit correspondre une clef qui lui est propre.

On peut associer le couple { **String**, **Abonne**) où l'objet **String** – le nom de l'abonné – représente la clef et l'objet **Abonne** l'objet à stocker.

// pour ajouter un nouvel abonné

```
Hashannu1.put(abon1.NomAbonne,abon1);
```

Syntaxe: **Object put(Object key, Object value)**

value désigne l'objet à stocker, **key** l'objet clé associé.

La recherche d'un abonné à partir d'un nom s'effectue d'une manière très simple.

// pour rechercher un N° à partir du nom **String** leNom :

```
Abonne ab = Hashannu1.get( leNom);
```

Il n'est plus nécessaire de balayer un hashmap comme on peut le faire pour un tableau ou un vector: la méthode **get** retourne l'objet souhaité grâce à la clé donnée en argument.

Pour effectuer la recherche d'un abonné à partir d'un numéro, on peut associer le couple {numéro, Abonné} pour créer un 2^{ème} HashMap où l'objet numéro représente la clé et l'objet Abonné l'objet à stocker.

```
Hashannu1.put(abon1.NomAbonne, abon1);
Hashannu2.put(abon1.NumeroDappel), abon1);

Abonne obj = Hashannu1.get( leNom);
```

Constructor Summary	
Constructors	
Constructor and Description	
<code>HashMap ()</code>	Constructs an empty <code>HashMap</code> with the default initial capacity (16) and the default load factor (0.75).
<code>HashMap (int initialCapacity)</code>	Constructs an empty <code>HashMap</code> with the specified initial capacity and the default load factor (0.75).
<code>HashMap (int initialCapacity, float loadFactor)</code>	Constructs an empty <code>HashMap</code> with the specified initial capacity and load factor.
<code>HashMap (Map<? extends K, ? extends V> m)</code>	Constructs a new <code>HashMap</code> with the same mappings as the specified <code>Map</code> .
Method Summary	
Methods	
Modifier and Type	Method and Description
<code>void</code>	<code>clear ()</code> Removes all of the mappings from this map.
<code>Object</code>	<code>clone ()</code> Returns a shallow copy of this <code>HashMap</code> instance: the keys and values themselves are not cloned.
<code>boolean</code>	<code>containsKey (Object key)</code> Returns <code>true</code> if this map contains a mapping for the specified key.
<code>boolean</code>	<code>containsValue (Object value)</code> Returns <code>true</code> if this map maps one or more keys to the specified value.
<code>Set<Map.Entry<K, V>></code>	<code>entrySet ()</code> Returns a <code>Set</code> view of the mappings contained in this map.
<code>V</code>	<code>get (Object key)</code> Returns the value to which the specified key is mapped, or <code>null</code> if this map contains no mapping for the key.
<code>boolean</code>	<code>isEmpty ()</code> Returns <code>true</code> if this map contains no key-value mappings.
<code>Set<K></code>	<code>keySet ()</code> Returns a <code>Set</code> view of the keys contained in this map.
<code>V</code>	<code>put (K key, V value)</code> Associates the specified value with the specified key in this map.
<code>void</code>	<code>putAll (Map<? extends K, ? extends V> m)</code> Copies all of the mappings from the specified map to this map.
<code>V</code>	<code>remove (Object key)</code> Removes the mapping for the specified key from this map if present.
<code>int</code>	<code>size ()</code> Returns the number of key-value mappings in this map.
<code>Collection<V></code>	<code>values ()</code> Returns a <code>Collection</code> view of the values contained in this map.

Travaux pratiques appliqués :

1° Création d'un annuaire en utilisant la classe HashMap.

On reprend l'exercice précédent, réalisé avec la classe `Vector`, en utilisant cette fois la classe `HashMap`.

Il faut utiliser 2 objets `HashMap`, l'un pour stocker les objets abonnés avec comme clé les noms des abonnés, l'autre pour stocker également les objets abonnés mais avec comme clé les numéros de téléphone.

L'affichage de tous les abonnés nécessitent de récupérer tous les objets contenus dans le `HashMap`. On obtient cela en utilisant la méthode **values()** de l'objet `HashMap` considéré. Cette méthode retourne tous les objets dans un objet **Collection** qu'on peut balayer à l'aide

d'un **itérateur** (voir le paragraphe 2 précédent et la classe Collection).

Ecrire une classe `AnnuaireMap.java` ayant les mêmes fonctionnalités que la classe `Annuaire` de l'exercice précédent, et mettant en œuvre la classe `HashMap` et non plus la classe `Vector`.

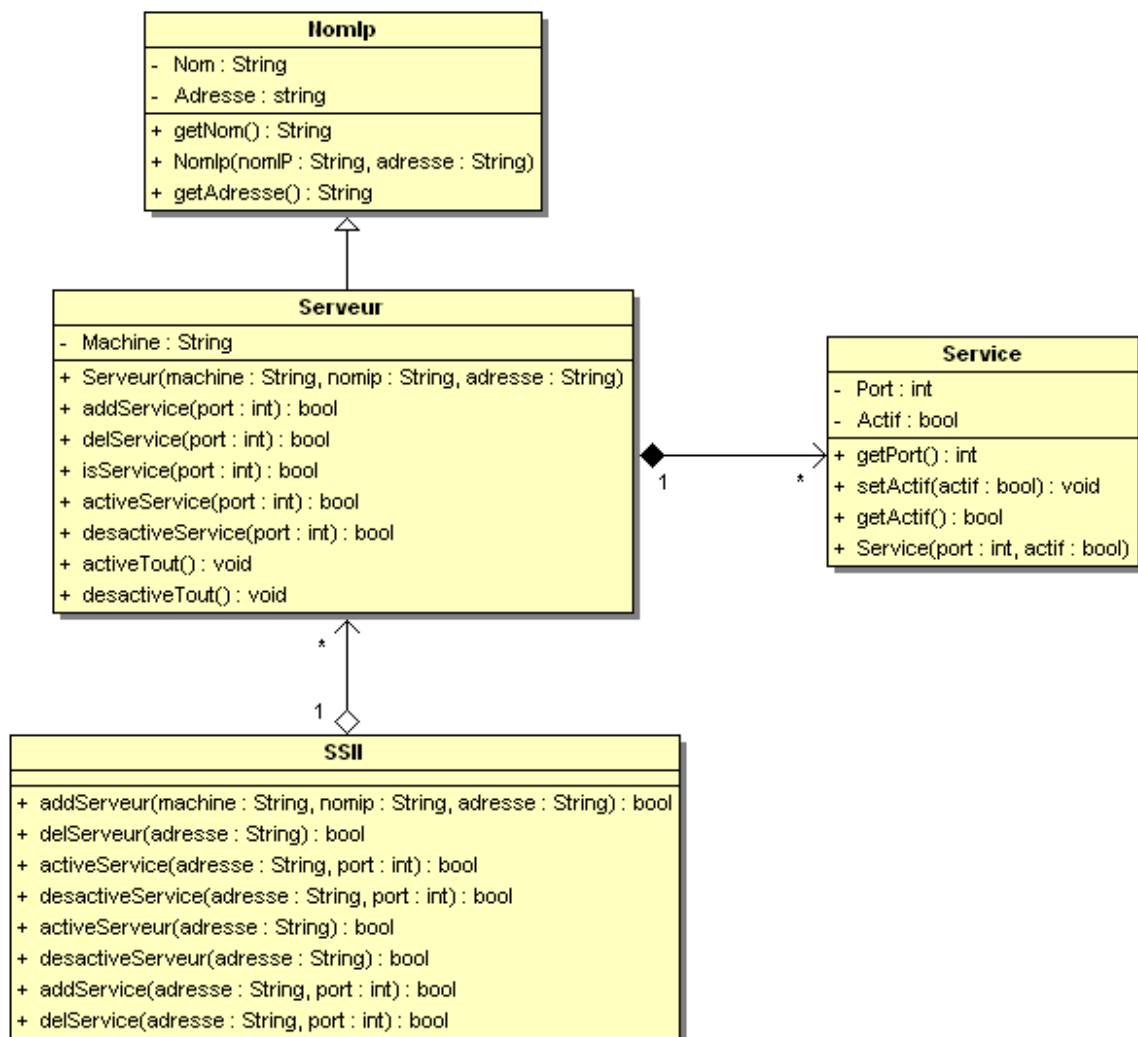
Ecrire un programme de test avec une méthode `main` : `testAnnuaireMap.java`

Tester la méthode `toString()` de la classe `HashMap`.

2° Modélisation d'une SSII

La SSII gère plusieurs serveurs Internet. Chaque serveur dispose de sa propre adresse IP et peut fournir plusieurs services.

Voici le diagramme des classes qui modélise cette SSII.



Le losange grisé symbolise une composition : un serveur "contient" des services, si le serveur tombe en panne, tous les services sont arrêtés.

Le losange clair symbolise une agrégation : la SSII gère des serveurs, on peut supposer que la durée de vie des serveurs est étroitement liée à la durée de vie de la SSII, mais que la faillite de la SSII n'implique pas la destruction des machines...

Chaque serveur possède sa propre référence -l'attribut Machine- et sa propre adresse IP. Le nom correspond au nom Internet de la machine.

Un serveur peut héberger plusieurs services, chaque service est caractérisé par un entier stocké dans l'attribut Port (correspond au n° de port d'un service TCP/UDP). Un serveur ne peut pas exécuté 2 services identiques.

La SSII gère plusieurs serveurs.

L'association Serveur – Service peut être assurée par un vecteur ou un hashmap. La clé du hashmap est alors le Port (entier à convertir en objet pour assurer la clé).

La classe Serveur

La méthode addService(port:int) ajoute le service correspondant, le service peut-être créé dans la méthode et mis inactif par défaut.

La méthode isService(port:int) teste si le service correspondant est géré par le serveur sans se préoccuper si ce service est actif ou pas..

La méthode activeService(port:int) active le service correspondant du serveur.

La méthode activeTout() active tous les services présents sur le serveur.

- Coder les classes NomIp, Service et Serveur.
- Ecrire une classe Main qui crée un serveur, des services, active et désactive ses services. Vérifier et tester avec soin le bon fonctionnement des divers objets créés.
- Ecrire la classe SSII. L'association SSII - Serveur peut être assurée par un vecteur ou un hashmap. La clé du hashmap peut être l'adresse IP.
- Compléter la classe Main pour créer un objet SSII, des serveurs, des services, active et désactive ses services. Vérifier et tester avec soin le bon fonctionnement des divers objets créés.

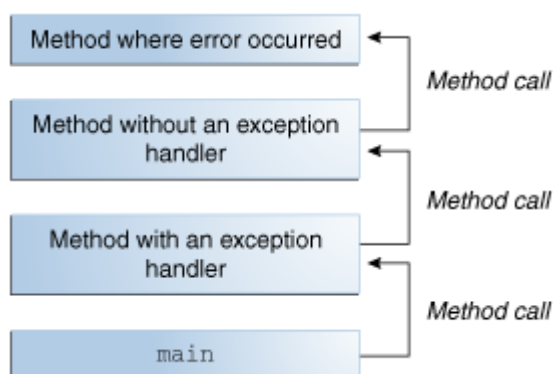
4. Les exceptions

4.1 Généralités sur les exceptions .

Une exception correspond à un événement anormal ou inattendu qui interrompt le fonctionnement normal d'un programme.

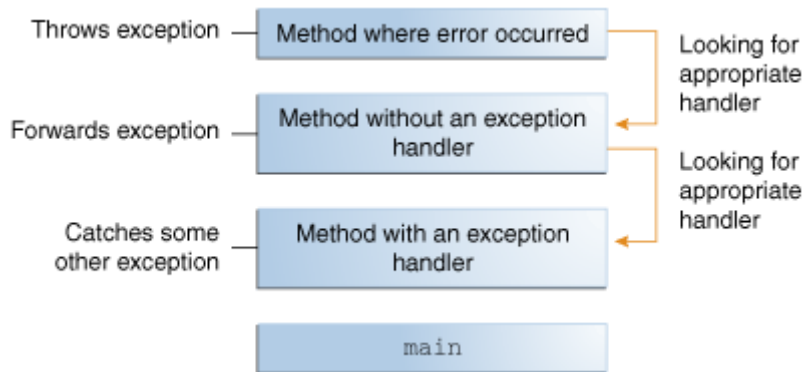
Une exception est déclenchée -lancée- par une méthode quand une erreur survient lors de son exécution. Un objet de type exception contenant des informations sur l'erreur survenue est alors créé.

Dès qu'une exception est survenue, le système recherche un bloc de code pour traiter l'exception.



La figure ci-contre montre le détail des appels des méthodes (call stack), ces appels se lisent de bas (du main) vers le haut.

La figure suivante montre le traitement de l'exception, la méthode en haut qui a lancée l'exception n'a pas de traitement prévu, celle qui l'a appelée non plus, en revanche la 3^{ème} dispose d'un gestionnaire de traitement de cette exception.



Il y a 2 possibilités pour traiter les exceptions :

1. Une méthode susceptible de déclencher une exception peut placer les instructions sensibles dans un bloc **try { }**. Le gestionnaire d'exception est alors constitué d'instructions placées dans un bloc **catch(..) { }**.
2. Une méthode susceptible de lancer **une exception sans l'attraper** (c'est-à-dire contenant une instruction susceptible de lancer une exception sans que celle-ci soit attrapée à l'intérieur de la méthode) **doit l'indiquer dans son en-tête** avec le mot réservé **throws**.

Il y a 3 types d'exception :

1. **Les exceptions vérifiées (checked) systématiquement.** Elles doivent obligatoirement être traitées comme il est indiqué ci-dessus, sinon le programme ne compilera pas. Les « checked » exceptions sont toutes du type **java.lang.Exception** et de ses classes dérivées hormis sa sous classe **java.lang.RuntimeException**.
2. Les exceptions de type **java.lang.Error**. Leur traitement est facultatif et c'est au programmeur de choisir le comportement du programme si une exception de ce type est déclenchée. Par exemple, une méthode a ouvert correctement un fichier mais une erreur se produit lors de sa lecture, une exception de type **java.io.IOException** est alors lancée, c'est au programmeur de choisir de la gérer avec les blocs try-catch ou de laisser le fonctionnement par défaut provoquer l'arrêt du programme en affichant la pile des méthodes traversées par l'exception.
3. Les exceptions de type **java.lang.RuntimeException**. Leur traitement est facultatif et c'est au programmeur de choisir le comportement du programme si une exception de ce type est déclenchée. Par exemple, une exception du type **java.lang.ArithmeticException** est lancée lors d'une division par zéro, ou bien une exception du type **java.lang.IndexOutOfBoundsException** est lancée lorsqu'on tente d'accéder à une valeur à l'extérieur d'un tableau ou d'un vecteur.

4.2 Exception « checked »

1^{er} exemple : exception « checked » attrapée

On reprend l'exemple du 1^{er} cours java.

```

import java.io.*;
public class Bonjour2
{
    public static void main(String[] args)
    {
        BufferedReader clavier= new BufferedReader(new InputStreamReader(System.in));
        String Saisies;
        int n,fois=0;
        System.out.println("Combien de fois par jour dites-vous Bonjour ?");
        try
        {
            Saisies=clavier.readLine();
            fois= Integer.parseInt(Saisies);
        }
        catch(IOException e)
        {
            System.out.println("Erreur IO");
            System.exit(0);
        }
        for(n=0;n<fois;n++)
        {
            System.out.print("Bonjour numero ");
            System.out.println(n+1);
        }
    }
}

```

La méthode `readLine()` est susceptible de lancer une exception du type `IOException`. Elle sera attrapée dans le bloc `catch`.

2^{ème} exemple : exception « **checked** » non attrapée, on l'indique dans son entête avec **throws**

```

import java.io.*;
public class Bonjour2
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader clavier= new BufferedReader(new InputStreamReader(System.in));
        String Saisies;
        int n,fois=0;
        System.out.println("Combien de fois par jour dites-vous Bonjour ?");

        Saisies=clavier.readLine();
        fois= Integer.parseInt(Saisies);

        for(n=0;n<fois;n++)
        {
            System.out.print("Bonjour numero ");
            System.out.println(n+1);
        }
    }
}

```

4.3 Exemple de cheminement d'une exception

Lorsqu'une exception est lancée, elle va se propager de la façon décrite ci-dessous. Nous supposons qu'une exception de type **NonException** est lancée par une certaine instruction **i** d'une méthode de nom `uneMethode`. L'explication que nous donnons est récursive.

- Soit l'instruction **I** se trouve dans un bloc de `uneMethode` avec les conditions suivantes :

- précédé du mot réservé **try**, bloc que nous appelons "bloc **try**"
- et suivi d'un bloc précédé du mot réservé **catch**, bloc que nous appelons "bloc **catch**", le mot **catch** étant assorti d'un **argument**, cet argument étant un objet instance de la classe ou d'une super classe de l'exception lancée.

```
public void uneMethode {
    try { ...
        instruction H ;
        instruction I ;
        instruction J ;
        instruction K ;
        ...
    } catch (NomException e) {
        // Bloc d'insrcutions du catch
        .....
    }
    Instruction X ;
    ...
}
```

L'instruction I déclenche l'exception de type `NomException` !!!

Alors :

- les instructions qui suivent le lancement de l'exception et intérieures au bloc **try** sont ignorées (instruction J et au-delà).
 - les instructions du bloc **catch** sont exécutées.
 - le programme reprend normalement avec l'instruction qui suit le bloc **catch** (instruction X).
- Soit l'instruction **I** n'est pas située comme indiqué ci-dessus.
Alors, la méthode `uneMethode` se termine. Si `uneMethode` est la méthode `main`, le programme se termine et l'exception n'a pas été attrapée.
Si `uneMethode` est appelée par une `uneAutreMethode` : soit `uneAutreMethode` appelle la méthode `uneMethode` dans un bloc **try-catch** et le bloc **catch** est exécuté, soit l'exception remonte dans la méthode qui a appelé `uneAutreMethode` et ainsi de suite... Voir la figure page 12.

4.4 Exception non « checked » Utilisation de try-catch, 1^{er} exemple

On prend l'exemple du 1^{er} cours Java, on intercepte l'exception **NumberFormatException**, exception déclenchée (`RuntimeException`) par la méthode de conversion `Integer.parseInt()` si l'utilisateur entre au clavier autre chose qu'un entier.

```
import java.io.*;
public class Bonjour4
{
    int Nombre ;
    BufferedReader clavier= new BufferedReader(new InputStreamReader(System.in));

    public void Lire() {
```

```

boolean saisie ;
String Saisies;
int fois=0;
do
{
    saisie = true ;           //saisie passe à false si l'utilisateur ne tape pas un entier
    System.out.println("Combien de fois par jour dites-vous bonjour ?");
    try
    {
        Saisies=clavier.readLine();
        fois= Integer.parseInt(Saisies);
        System.out.println("Saisie correcte");
    }
    catch (NumberFormatException e){
        System.out.println("Erreur de saisie");
        System.out.println(e);
        saisie = false ;
    }
    catch(IOException e)
    {
        System.out.println("Erreur IO");
        System.exit(0);
    }
    System.out.println("Avant fin du test sur saisie");
} while (saisie == false);

Nombre = fois ;
}

public void afficher() {
    for(int n=0;n<Nombre;n++)
    {
        System.out.print("bonjour n° ");
        System.out.println(n+1);
    }
}

public static void main(String[] args) {
    Bonjour4 b = new Bonjour4();
    b.Lire();
    b.afficher();
}
}

```

Cet exemple montre bien que l'instruction **while(saisie==false)** ; est exécutée, que l'exception soit déclenchée ou pas.

Le **booléen** **saisie** est mis à **true** au début de la boucle **do { } while(saisie==false)** ;

Le traitement de l'exception met **saisie** à **false** et force à rester dans la boucle pour effectuer une nouvelle saisie.

Exercice:

α) Tester cet exemple dans les 2 cas de fonctionnement.

β) Mettre en commentaire le bloc catch NumberFormatException comme ci-dessous :

```
/* catch (NumberFormatException e){
    System.out.println("Erreur de saisie");
    System.out.println(e);
    saisie = false ;
} */
```

Tester en faisant une erreur de saisie, observer les N° des lignes indiquées dans le message d'erreur, justifier ce message.

4.5 Utilisation de try-catch, 2^{ème} exemple (exemple tiré du cours d'Irène Charon)

Pour cet exemple, l'objectif est de calculer une moyenne de notes entières envoyées en arguments par la ligne de commande.

```
class ExceptionCatch
{
    static int moyenne(String[] liste)
    {
        int somme = 0, entier, nbNotes = 0;

        for (int i = 0; i < liste.length; i++)
        {
            try
            {
                entier = Integer.parseInt(liste[i]);
                somme += entier;
                nbNotes++;
            }
            catch (NumberFormatException e)
            {
                System.out.println("La " + (i+1) + " eme note n'est "+
                                   "pas entiere");
            }
        }
        return somme/nbNotes;
    }

    public static void main(String[] argv)
    {
        System.out. println("La moyenne est "+moyenne(argv));
    }
}
```

Après la compilation,

1^{ère} exécution de cet exemple :


```
java ExceptionCatch 14 12
```

```
La moyenne est 13
```

2^{ème} exécution de cet exemple, avec une erreur sur la 1ère note:

```
java ExceptionCatch ha 14 12
```

```
La 1 eme note n'est pas entiere
La moyenne est 13
```

On constate le déclenchement de l'exception `NumberFormatException` pour le faux nombre « ha », mais le programme continue.

3^{ème} exécution de cet exemple, avec une erreur sur les 2 notes:

```
java ExceptionCatch ha 15.5
```

```
La 1 eme note n'est pas entiere
La 2 eme note n'est pas entiere
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionCatch.moyenne(ExceptionCatch.java:21)
    at ExceptionCatch.main(ExceptionCatch.java:26)
```

Le compteur `nbNotes` n'a pas été incrémenté, `nbnotes` vaut 0, la division par 0 déclenche une exception `java.lang.ArithmeticException: / by zero` qui n'est pas capturée et qui force la fin du programme. La pile des méthodes stoppées est indiquée dans le message d'erreur.

Afficher la pile de méthodes traversées par l'exception

```
try
{
    entier = Integer.parseInt(liste[i]);
    somme += entier;
    nbNotes++;
}
catch (NumberFormatException e)
{
    System.out.println("La " + (i+1) + " eme note n'est "+
                      "pas entiere");
    e.printStackTrace() ;
}
}
return somme/nbNotes;
```

La méthode `printStackTrace()` de l'objet instance de `NumberFormatException` affiche la liste des méthodes traversées par l'exception.

Exercice:

- a) Tester cet exemple dans les 2 cas de fonctionnement.

- b) Améliorer le programme de calcul de la moyenne afin qu'il fonctionne même si on entre des notes incorrectes. Les notes correctes sont des valeurs comprises dans l'intervalle [0 20]. On peut créer une exception qui sera déclenchée lorsque les notes ne seront pas dans l'intervalle [0 20], les notes incorrectes seront exclues du calcul grâce aux exceptions.

4.6 Utilisation de throws – throw, création de sa classe d'exception, exemple

Si on veut signaler un événement exceptionnel d'un type non prévu par l'API, il faut étendre la classe **java.lang.Exception**; la classe étendue ne contient en général pas d'autre champ qu'un (ou plusieurs) constructeur(s) et éventuellement une redéfinition de la méthode `toString`.

La méthode susceptible de déclencher cette exception doit le signaler à l'aide du mot clé **throws** :

```
type nomMethodeX (type arg1,...) throws ExceptionX
```

Tout appel à cette méthode doit être encapsulé dans un bloc `try ... catch()` :

```
try {
    ...
    nomMethodeX(...);
    ...
}
catch (ExceptionX e) { ... }
```

On doit trouver dans le code de la méthode **nomMethodeX()** la clause du lancement de l'exception à l'aide du mot réservé **throw** comme nous le verrons dans le prochain exemple. Le lancement de l'exception consiste à créer une instance de la classe d'exception définie.

Voilà une classe héritant de la classe **Exception**.

```
public class ExceptionRien extends Exception
{
    public ExceptionRien() {
        super();
        System.out.println("Constructeur de ExceptionRien");
    }
    public String toString()
    {
        return("Aucune note n'est valide");
    }
}
```

Pour cet exemple, l'objectif est de calculer une moyenne de notes entières envoyées en arguments par la ligne de commande. Les arguments non entiers (et donc erronés) doivent être éliminés.

On utilise l'exception `ExceptionRien`.

```
public class ExceptionThrow
{
```

```

static int moyenne(String[] liste) throws ExceptionRien
{
    int somme=0,entier, nbNotes=0;
    int i;

    for (i=0;i < liste.length;i++)
    {
        try
        {
            entier=Integer.parseInt(liste[i]);
            somme+=entier;
            nbNotes++;
        }
        catch (NumberFormatException e)
        {
            System.out.println("La "+(i+1)+" eme note n'est "+
                               "pas entiere");
        }
    }
    if (nbNotes==0) throw new ExceptionRien() ;
    return somme/nbNotes;
}

public static void main(String[] argv)
{
    try
    {
        System.out.println("La moyenne est "+moyenne(argv));
    }
    catch (ExceptionRien e)
    {
        System.out.println(e);
    }
}

```

Exécution, pour :
 java ExceptionThrows ha 15.5
 on obtient :

```

La 1 eme note n'est pas entiere
La 2 eme note n'est pas entiere
Constructeur de ExceptionRien
Aucune note n'est valide

```

static int moyenne(String[] liste) throws ExceptionRien : la méthode moyenne indique ainsi qu'il est possible qu'une de ses instructions envoie une exception de type ExceptionRien sans que celle-ci soit attrapée par un mécanisme try-catch. Il est obligatoire d'indiquer ainsi un éventuel lancement d'exception non attrapée, sauf pour les exceptions les plus courantes de l'API. Si vous oubliez de signaler par la clause **throws** l'éventualité d'un tel lancement d'exception, le compilateur vous le rappellera.

if (nbNotes==0) throw new ExceptionRien() ; : on demande ainsi le lancement d'une instance de **ExceptionRien**. Une fois lancée, l'exception se propagera comme expliqué dans l'exemple précédent.

Dans cet exemple d'exécution, l'exception **NumberFormatException** est appelée 2 fois dans la boucle, puis il y a sortie de la boucle, et l'exception **ExceptionRien** est créée. Cela consiste à exécuter le constructeur d' **ExceptionRien** puis la méthode **toString()** de l'instance est exécutée grâce à l'instruction **System.out.println(e)** placée dans le bloc **catch**. Le programme se termine ensuite.

Exercice 4.4 : tester cet exemple avec aucune note valide.

4.7 La clause finally

S'il y a du code important (par exemple du code effectuant un nettoyage) que vous voulez exécuter même si une exception est déclenchée et que le programme s'arrête, placez ce code à la fin, dans un bloc **finally**. Voici un exemple de ce fonctionnement :

```
try {
    ... // Insérer ici du code pouvant déclencher une exception.
}
catch( Exception e ) {
    ... // Insérer ici le code de gestion de l'exception.
}
finally {
    ... // Le code inséré ici sera toujours exécuté,
        // que l'exception ait été déclenchée dans le bloc try ou non.
}
```

Exemple précédent modifié

```
static int moyenne(String[] liste) throws ExceptionRien
{
    int somme=0,entier, nbNotes=0;
    int i;

    for (i=0;i < liste.length;i++)
    {
        try
        {
            entier=Integer.parseInt(liste[i]);
            somme+=entier;
            nbNotes++;
        }
        catch (NumberFormatException e)
        {
            System.out.println("La "+(i+1)+" eme note n'est "+
                                "pas entiere");
        }
        finally {
            System.out.println("Affiché par la clause finally");
        }
    }
}
```

```

    }
}
if (nbNotes==0) throw new ExceptionRien();
return somme/nbNotes;
}

```

4.8 Mise en œuvre des exceptions pour la recherche d'un abonné dans l'annuaire .

Les méthodes **RechercherParLeNom** et **RechercherParLeNumero** de la classe Annuaire , étudiée dans les exercices précédents, pourraient déclencher une exception dans le cas où un objet recherché n'est pas trouvé .

- On va écrire une classe **RechercheException** pour notre propre exception. On utilise ici la classe RuntimeException qui est spécialisée pour toutes les exceptions lancées par la JVM, elle dérive de la classe Exception) :

```

public class RechercheException extends RuntimeException {
    public RechercheException() {
        super( "La recherche a échoué \n " );
    }
}

```

La classe RechercheException est dérivée de la classe RuntimeException , qui peut être déclenchée quand un programme est en exécution.

Notre classe dérivée affiche un message en utilisant la super classe.

- Dans le fichier Annuaire.java , la méthode RechercherParLeNumero devient :

```

// Pour rechercher un abonné par son numéro
public String RechercherParLeNumero(String CeNumero) throws RechercheException
{
    int n1 ;
    int etat=0;
    for(n1=0 ; n1 < LesAbonnes.size() ; n1 ++ )
    {
        Abonne CestLui= (Abonne)LesAbonnes.get(n1);
        if( CestLui.NumeroDappel.compareTo( CeNumero ) == 0 )
        {
            etat=1;
            return CestLui.NomAbonne;
        }
    }
    if(etat == 0) throw new RechercheException();

    return "";
}

```

throws RechercheException car cette méthode va pouvoir déclencher une exception .
throw new RechercheException(); c'est là qu'est déclenchée l'exception .

- Dans l'application TestAnnuaire.java lorsqu'on utilise la méthode RechercherParLeNumero , il faut un bloc try qui peut lever l'exception et un bloc catch qui la gère , le bloc finally n'étant pas nécessaire .

// méthode de recherche gérée par exception

```
try
{
    String cetAbo= CetAnnuaire.RechercherParLeNumero(RechercheNumero);
    System.out.println("Le numéro " + RechercheNumero + " est dans l'annuaire");
    System.out.println("Son abonné est "+ cetAbo );
}
catch(RechercheException e)
{
    System.out.println(e.toString());
    System.out.println("L'abonné " + Integer.toString(RechercheNumero) + " n'est pas dans l'annuaire");
}
}
```

4.9 Travaux pratiques appliqués :

a) Tester et observer le fonctionnement des exemples restants du paragraphe 4.

b)

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE+1; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }

    public static void main(String[] args)
```

```
{  
    ListOfNumbers l = new ListOfNumbers();  
    l.writeList();  
}  
}
```

Ce programme ne compile pas (volontairement).

1. Trouver pourquoi.
2. Corriger ce problème afin qu'il compile.
3. Exécuter ce programme. Que se passe-t-il ?
4. Corriger ce problème exclusivement en traitant l'exception.

b) Reprendre le TP n°1 par exemple, et travailler avec les exceptions pour gérer la recherche des objets dans le Vector .