

java done quick

boré & eliazZz & sttev

13 décembre 2014

no docs/no objects/fox only/final destination/mge me

1 Généricité

D'après ma source sûre (micko black) ce serait un type de polymorphisme¹.
Donc ce terme vient de générique. En effet on va pouvoir créer ce qu'on appelle des méthodes mais surtout des classes génériques, c'est-à-dire qui vont pouvoir faire des actions génériques à n'importe quelle types de données.

Une classe générique se définit de la manière suivante :

```
public class GenericQuadruplet<T1,T2,T3,T4> {  
    private T1 first;  
    private T2 second;  
    private T3 third;  
    private T4 fourth;  
}
```

T1, T2, T3 et T4 étant des paramètres génériques, des types d'objets. Dans cette classe générique, on définit une variable de chacun de ces types. On définit ensuite un ou plusieurs constructeur(s), getters, setters et fonctions qui feront des actions sur ces objets. (Se référer au cours pour plus de détails).
Pour instancier une classe générique avec les objets que l'on désire, on procède de la manière suivante en créant une autre classe par exemple :

```
public class GenericExample {  
    public static void main(String[] args)  
    {  
        GenericQuadruplet<Integer,String,Double,Byte> q1 =  
            new GenericQuadruplet<Integer,String,Double,Byte>  
                (new Integer(1), new String("str"), new Double(3.0), new Byte((byte)4));  
    }  
}
```

Ici ma classe *GenericExample* crée une instance de ma classe *GenericQuadruplet* et initialise mes objets avec leurs valeurs grâce au constructeur défini dans cette dernière.

En gros, je crée une classe générique avec 4 objets différents qui sont *Integer*, *String*, *Double* et *Byte*. J'aurais pu mettre, si je l'avais voulu, que des *Integer*. D'où l'utilité d'une classe générique.

Attention : Les paramètres d'une classe générique sont nécessairement des objets c'est pourquoi mettre *int* au lieu de *Integer* ou *double* au lieu de *Double* est faux.

1. Mis a part pour impressionner quelques rang d'hommes, c'est pas très important

En effet *int* et *double* sont des types primitifs tandis que *Integer* et *Double* sont types des types dit "enveloppes" correspondant à ces deux types primitifs. --En plus d'être un objet, un type enveloppe possède des fonctions utiles manipulant les types primitives auxquels il est affilié.-- (Se référer au cours pour les autres exceptions).

L'utilité principale ici est d'écrire un code unique pour plusieurs types d'objets. Prenons un exemple concret, celui du cours est plutôt #bogoss je trouve : On crée une classe générique Point :

```
public class Point<X,Y> {  
    protected X x;  
    protected Y y;  
    [...]  
}
```

X et Y sont mes objets que je définirai plus tard. Bien sûr, on part du principe qu'il y a aussi tout le reste - constructeur, getter, setters, etc -.

Je crée une autre classe pour faire appel à toutes mes jolies fonctions de ma classe générique :

```
public class GenericInheritance {  
    public static void main(String[] args) {  
        Point<Integer,Integer> p =  
            newPoint<Integer,Integer>(newInteger(1), newInteger(2));  
        [...]  
    }  
}
```

Donc là j'ai créé une instance de ma classe générique avec 2 objets Integer. Du coup je peux mettre ce que je veux et même foutre un objet String - pourquoi pas -.

D'ailleurs ça le ferait pas trop pour une histoire de points et de dimensions ("*Diagonaliser la matrice et calculer le nuage de points!*" # Rinkel).

Donc : comment être sûr d'utiliser uniquement des nombres ?? Et bien il suffit de restreindre la généricité pour cela rien de plus simple il suffit de rajouter un *extends* dans les paramètres de ma classe générique :

```
public class Point<X extends Number,Y extends Number> {  
    [...]  
}
```

Donc là pour chacun de mes objets je fais un *extends Number* pour indiquer que ces objets ne peuvent être que de type Integer, Double ..

Information : L'héritage fonctionne également avec les classes génériques.

2 Collections

Il s'agit d'une interface qui est implémentée par des classes afin de manipuler une liste d'objet dynamiquement. Une *ArrayList* est une collection, une *LinkedList* aussi, etc.

On peut parcourir une collection avec un itérateur, *Iterator*. Par exemple, cela permet à Java de faire des *foreach* avec une liste. Et pour nous programmeur de renau... renom, cela permet dans certains cas de parcourir une liste rapidement.

Les collections s'utilisent tous un peu de la même façon de fait qu'ils possèdent tous les fonctions de *Collection*, mais voyons les types intéressants :

- **LinkedList** et **ArrayList**. Ils héritent tout deux de *Collection* et de *Iterable*;
- **HashSet**. Il hérite de *Set*;
- **HashMap**. Il hérite de *Map*;

LinkedList et ArrayList Une LinkedList comme son nom l'indique est une liste dont chaque élément est "lié" à la suivante et à la précédente (ils possèdent des références). Alors qu'ArrayList est un tableau dynamique. Il se redimensionne lorsque l'on rajoute un élément (Il s'agissait anciennement de la collection Vector)

Par conséquent, si l'on veut récupérer l'élément n d'une LinkedList, la liste va partir de son premier élément et aller vers l'élément suivant, n fois. Contrairement à l'ArrayList qui est un tableau et qui récupère donc directement le $n^{ième}$ élément. De l'autre côté, si je veux insérer une nouvelle valeur dans l'ArrayList, la classe va initialiser un nouveau tableau dans laquelle il va rajouter les éléments existants ainsi que le petit nouveau, alors que le LinkedList rajoute juste une référence à son dernier élément vers la nouvelle. Bref, c'est une histoire de complexité et de temps d'accès.

```
LinkedList<String> linkedListString = new LinkedList<String>();
ArrayList<String> arrayListString = new ArrayList<String>();
//etc
```

HashSet : Une collection de type set, c'est une collection dans laquelle chaque élément ne peut s'y trouver **qu'une fois**. Ce type de Set utilise la fonction `hashCode()` pour vérifier si un élément existe déjà ou pas. Notez que pour parcourir un HashSet, il faut forcément utiliser un *Iterator*.

```
HashSet monHashSet<String> = new HashSet<String>(); // on crée notre Set
monHashSet.add(new String("1")); // on ajoute des string quelconques
monHashSet.add(new String("2"));
monHashSet.add(new String("3"));
i = monHashSet.iterator();
while(i.hasNext()) // tant qu'on a un suivant
{
    System.out.println(i.next()); // on affiche le suivant
}
/* Notez que l'ordre de l'itération se fait aléatoirement.*/
```

HashMap : Un HashMap, c'est une collection dans laquelle chaque valeur est associé à une clé. Il s'agit d'une classe générique qui prend deux types d'objets en paramètre. On lui définit alors le type des "clés" et celui des objets que l'on veut placer. Ainsi quand je souhaite ajouter un objet je dois lui spécifier une clé qui me permettra de le retrouver par la suite.

```
// on crée notre Map<Key, Object>
HashMap monHashMap<Character, String> = new HashMap<Character, String>();
// on ajoute des string quelconques
monHashMap.put('B', "Bruno");
monHashMap.put('S', "Steeve");
// Erreur le caractère 'B' est la clé associé à la chaine "Bruno"
monHashMap.put('B', "Borey");
// récupère la chaine associé à la clé 'S'
String p = monHashMap.get('S');
if(monHashMap.get('E')==null)
    System.out.println("Mon HashMap ne possède pas de chaine associé à la
        lettre E");

System.out.println("Liste des valeurs contenues dans mon HashMap")
// Affichage de chaque chaine de monHashMap
```

```
for (String s : monHashMap.values())  
    System.out.println("-" + s);
```

3 Flux de données : les fichiers

Comme en système, il existe des *InputStream* ou des *OutputStream*, qui permettent de gérer des flux de données. Pour l'instant, on va s'intéresser aux flux de données pour les fichiers, mais on verra après les flux de données destinés aux réseau, qui ne sont pas si différents.

Revenons à nos fichiers.

Tout comme en système, il faut d'abord ouvrir le fichier. Pour cela, on déclare une instance de *File* avec en paramètre le nom du fichier.

```
File fichierRandom = new File("td32.txt");
```

On peut effectuer deux actions sur un fichier : **écrire**, ou **lire**. Cela se représente donc par deux classes distinctes, **FileOutputStream** et **FileInputStream**.

Commençons par l'écriture.

Écriture : Pour écrire depuis le programme, il faut utiliser la classe **FileOutputStream**, dont le constructeur prend en paramètre le *File* créé auparavant. Après avoir créé le *FileOutputStream*, on utilise la fonction *.write*, qui prend en paramètre un tableau de bytes. Si on veut écrire un objet dans un fichier, on passe par *ObjectOutputStream*.

Imaginons que je souhaite écrire dans un fichier un objet de type "Random" dans un fichier "td32", on va donc avoir :

```
FileOutputStream fluxSortie = new FileOutputStream(new File("td32"));  
Random rangdhome = new Random();  
ObjectOutputStream oos = new ObjectOutputStream(fluxSortie);  
oos.writeObject(rangdhome);  
fluxSortie.close();
```

Penchons nous maintenant sur la sauvegarde : la *serialisation*. Vous vous souvenez de Lavengro ? Ben voilà. Il faut implémenter l'interface *Serializable*. Pour l'écriture après ça ne change rien, c'est pour la lecture qu'une subtilité va apparaître. D'ailleurs, en parlant de lecture, concentrons nous dessus.

Lecture : On utilise alors *FileInputStream*. Le principe est le même que pour l'écriture. On va reprendre notre fichier du haut, qui comporte un objet de type "Random".

```
FileInputStream fluxEntree = new FileInputStream(new File("td32"));  
ObjectInputStream ois = new ObjectInputStream(fluxEntree);  
Random recup = (Random) ois.readObject();
```

les pipes Ah ces bon vieux pipes. Deux classes, une qui représente l'entrée - *PipedInputStream* -, une autre qui représente la sortie - *PipedOutputStream* -. Ça n'a pas l'air très important.

4 Communication réseau : serveur et client

Toujours comme en Système, on va passer par des sockets ; mais ne vous inquiétez pas, c'est plus facile ! Intéressons nous d'abord à la partie serveur.

4.1 Le serveur

Un serveur va d'abord commencer par créer un socket d'écoute. C'est la classe `ServerSocket`. Ensuite, il a juste à accepter les connexions entrantes, à l'aide de la fonction de la classe socket `accept()`.

```
int port = 1337;
ServerSocket s = new ServerSocket(port);
boolean isRunning = true;
// On accepte les requetes tant que le serveur tourne
while(isRunning){
    try{
        Socket socketClient = s.accept();
        [...]
    }
    catch(IOException ex){
        System.out.println("dommage");
    }
}
```

4.2 Le client

Pour le client c'est pas plus compliqué : il a juste à créer un socket contenant l'adresse ip du serveur et son port.

```
try{
    Socket soc = new Socket(ipServeur, port);
    [...]
}
catch(IOException ex){
    [...]
}
```

Mais, vous devez vous dire, c'est bien joli mais j'aimerais communiquer ouam.
Et vous avez raison. Pour cela, on va utiliser nos bons vieux flux de données.
Les équivalents à nos flux de données pour nos fichiers sont ici `BufferedReader` et `PrintWriter`.

Lire des informations : Pour lire depuis un socket, on va donc utiliser, vous l'aurez deviné, `BufferedReader`.
Voyons comment ça marche :
(Pour cet exemple, on se met partie client, mais c'est la même chose pour le serveur.)

```
try
{
    BufferedReader lecture = new BufferedReader(new InputStreamReader(soc.
        getInputStream()));
    lecture.read();
    [...]
}
catch(IOException e)
{
    [...]
}
```

Écrire des informations : De même, on va utiliser *PrintWriter* pour écrire dans notre socket.

```
try
{
    PrintWriter ecriture = new PrintWriter(new OutputStreamWriter(soc.
        getOutputStream()));
    ecriture.write("merci");
}
catch(IOException e)
{
    [...]
}
```

5 Exceptions et loggers

5.1 Exceptions

Parce personne n'écrit un programme non-trivial sans erreur du premier coup (peut-être à part Pouly), et que la vie d'un programme n'est pas un long fleuve tranquille, on risque bien souvent de rencontrer au coin d'une ligne de code les **exceptions**.

Déjà, il faut savoir qu'il existe 3 grands types d'erreur en Java.

- **Error**, lorsque il se passe une erreur vraiment grave (mémoire qui fuit, jvm qui fait du caca, ...);
- **RuntimeException**, lorsque il se passe une erreur liée à la programmation (le fameux *NullPointerException*, *OutOfBounds* et j'en passe);
- **Exceptions**, qui correspondent en fait à des erreurs moins graves qui ne compromettent pas forcément totalement le fonctionnement du programme. C'est ce qui va nous intéresser dans cette partie.

Les *RuntimeException* sont dérivées des *Exceptions*. À la différence des *Exceptions* tout court, on est pas obligé de traiter - c'est à dire avec un bloc *try/catch/finally* - les *RuntimeException*.

Parlons maintenant du traitement en soi.

Il y a deux types d'exceptions, les exceptions lancées toutes seules - à l'exécution de la ligne de code - ou les exceptions jetées avec *throw*. Concentrons nous pour l'instant sur le premier cas.

Lorsque il existe une possibilité d'*Exception*, il faut enrober le code qui pourrait lever une exception de chocolat d'un bloc *try*. Ce bloc doit être suivi d'un autre bloc, *catch*, qui contiendra le code exécuté si jamais une exception est levée. Ce bloc doit prendre en paramètre le type d'*Exception* qu'il aura à traiter. Éclairissons tout ça avec un petit exemple pas piqué des hannetons :

```
try
{
    // commande qui lancera une exception de type IOException
    FileOutputStream reader = new FileInputStream(new File("un fichier inexistant
        mdr"));
}
catch (IOException e)
{
    System.out.println("Je suis exécuté si c'est une exception de type
        IOException");
}
catch (Exception e)
{
    System.out.println("Je suis exécuté pour tous les types d'exceptions");
}
```

```
System.out.println("Sauf pour IOException du coup.");  
}
```

Maintenant, regardons les exceptions lancées par *throw*.

Pour ce type d'exceptions, il faut d'abord spécifier dans la signature de la fonction le type d'exception lancée. Ensuite, dans le corps du programme, on lance - *throw* - une nouvelle instance de cette Exception. Regardons un exemple :

```
public void procedureCapableException() throws IOException  
{  
    //Condition qui vérifie si tout va bien ou pas  
    boolean cLaMerde = false;  
    [...] //reste du programme qui pourrait modifier notre boolean  
    if (cLaMerde)  
    {  
        throw new IOException();  
    }  
    [...]  
}
```

c tou

5.2 Loggers

Les loggers permettent de ... logger tous les événements d'un programme de façon précise : connexions, ouverture de fichier, et surtout erreurs. Si vous voulez, c'est une sorte de *System.out.println()*, mais en beaucoup mieux.

Ainsi, notre cher Blau2ez utilise en tant que logger *log4j*. Pas vraiment besoin de savoir ce que c'est réellement, contentez de vous de savoir comment le mettre en place :

```
import org.apache.log4j.Logger;  
public class Randon {  
    [attributs]  
    public static Logger logger = new Logger(Randon.class);  
    [...]  
}
```

Et voilà, le logger est mis en place. Maintenant, pour l'utiliser, c'est aussi simple.

```
// On se trouve dans notre classe Randon  
public static void main(String args[])  
{  
    logger.debug("msg de debogage");  
    logger.info("msg d'information");  
    logger.warn("msg d'avertissement");  
    logger.error("msg d'erreur");  
    logger.fatal("msg d'erreur fatale");  
}
```

À chaque fois qu'on appelle le logger, on écrit donc un message dans la console java. Super !!!!!!! Enfin, parlons un peu des paramètres du loggers ; c'est un peu spécial, puisque en fait on modifie le fichier .xml du logger :

A FAIRE !!!!!!!

6 Threads & Jamy

Ah, on se lassera jamais de cette *bonne boutade* datant de la fiche de Système. Enfin bref. Tout comme les pthreads de notre bon vieux Bouyou, les threads partagent un même espace mémoire et permettent en fait d'exécuter des tâches en parallèle. Une classe agissant en tant que thread doit implémenter l'interface *Runnable*, et doit ainsi donc surcharger la méthode *run()*. C'est dans cette méthode *run()* que se trouvera le code exécuté par le thread.

```
public class ThreadPerso implements Runnable
{
    [attributs]
    [constructeur]

    public void run()
    {
        // faire des trucs
    }
}
```

Et hop, ainsi, dans notre main, lorsque on voudra créer des threads qui exécuteront ça, on fera :

```
public static void main(String args[])
{
    Thread th1 = new Thread(new ThreadPerso());
    Thread th2 = new Thread(new ThreadPerso());

    th1.start(); th2.start();
    [...]
    th1.join(); th2.join()
}
```

Mais comme tout ce qui touche à l'accès parallèle à des données, il y a des risques de se *mélanger les données* entre Threads. Par conséquent, il faut protéger l'accès aux données pour chaque Thread. On peut par exemple rendre une procédure synchronisée, c'est à dire que lorsque un thread est occupé avec les données, les autres ne peuvent toucher aux données. Dans ce cas, on ajoute le mot-clé *synchronized* dans la signature de la fonction concernée.

```
public synchronized void modifierTruc(Truc modifier)
{
    [...]
}
```

L'autre moyen de protéger les données s'appelle *Lock*. En fait, utiliser un *Lock* permet de bloquer une partie des ressources utilisées entre le moment où on utilise le lock et le moment où on le "délock". Cela permet de ne pas avoir à bloquer tous les autres threads pendant toute une fonction.

```
Lock lock = new Lock();

public void modifierTruc(Truc modifier)
{
    lock.lock();
    [...]
    lock.unlock();
}
```


}

c tou pour se soir j'ai dodo