LU2IN002 - Introduction à la programmation orientée-objet

Christophe Marsala

SCIENCES SORBONNE UNIVERSITÉ

Cours 11 – 9 décembre 2022

Programme du jour

1. Les flux (suite)
2. Conteneurs génériques
3. Design Patterns

Plan du cours

Bilan : flux pour lire depuis un fichier (1)

```
1  FileInputStream in = new FileInputStream(new File("data.txt"));
```

Lecture du flux (toujours du début à la fin : flux séquentiel) :

- une lecture peut lever une IOException
- lecture octet par octet

```
2  int c = in.read(); // lecture d'un octet
3  // si c vaut -1 : fin de fichier atteinte
```

- lecture de haut niveau :
  - encapsulation du FileInputStream dans DataInputStream

```
4  DataInputStream istream = new DataInputStream(in);
```

  - méthodes disponibles :

```
5  istream.readChar();       // rend un char
6  istream.readInt());         // rend un int
7  istream.readDouble()); // rend un double
8  istream.readFloat());     // rend un float
```

  - la fermeture du flux peut se faire depuis DataInputStream

```
9  istream.close();
```


Bilan : flux pour lire depuis un fichier (2)

```
1  ...
2  istream.readChar();       // rend un char
3  istream.readInt());         // rend un int
4  istream.readDouble()); // rend un double
5  istream.readFloat());     // rend un float
6  ...
```

Exceptions avec un DataInputStream

- une lecture peut lever une IOException
- si on ne peut pas lire ce qui est demandé : EOFException
  - par exemple : un int correspond à 4 octets
  - utile pour détecter la fin du fichier

Ouverture en écriture...

Par défaut : remplacement des fichiers existants...

Gare aux catastrophes !

Il existe une option pour ajouter des choses dans un fichier **sans** écraser le contenu :

```
1  try {
2      ostream = new DataOutputStream(
3        new FileOutputStream(new File("toto.dat"), true) );
4                  // le boolean sert pour l'option append
5  ...
```

Classes pour les flux : premier bilan (1)

- Package java.io.*
- Classe File : créer un objet pour représenter un fichier
  - gérer un fichier ou un répertoire (créer, lire,...)
- Flux : connexion entre le programme et un support
  - flux entrant : input
    - classes abstraites : InputStream (général), Reader (caractères)
    - accès : FileInputStream (octet), FileReader (caractères)
  - flux sortant : output
    - classes abstraites : OutputStream (général), Writer (caract.)
    - accès : FileOutputStream (octet), FileWriter (caractères)
- Accès de haut niveau à un flux
  - lecture : DataInputStream
  - écriture : DataOutputStream

ASCII or not ASCII...

Dans les opérations précédentes, voici le fichier manipulé (ouvert avec emacs) :

```
^@r@^@^@^@^@^@^@^@^F^@^@^@t^@o^@t^@o
```

Ce qui n'est pas très lisible pour nous...

$\Longrightarrow$ Il y a des avantages et des inconvénients.

- gain de place, fichier plus compact,...
- lisibilité, portabilité,...

Gestion de l'ASCII

Problème de l'accès à des données brutes

- on peut pas lire directement les int/double sauvés par la méthode précédente...
- on peut éventuellement tricher pour l'écriture :

```
1 ostream.writeChars(((Double) 2.5).toString());
```

$\Rightarrow$ possible mais perte de précision

Codage ASCII

- codage sur 1 octet : table des caractères

Problème de lecture des String

- Lecture caractère par caractère fastidieuse
- Problème des sauts de ligne (codage différent selon le système)

$\Rightarrow$ Choisir en fonction des situations (possibilité de mélanger)

ASCII or not ASCII...

ASCII

- Fichiers de Textes
- Entêtes des fichiers
- XML, HTML...
- Parfois pour quelques chiffres
  - lorsque la précision n'est pas importante
  - pour les fichiers excels...

not ASCII

- Fichiers de chiffres
  - Volume
  - Précision

Gestion en lecture de fichier ASCII

- Encore une nouvelle classe : BufferedReader...
- qui introduit une nouvelle méthode :

public String readLine() throws IOException

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
Returns : A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
Throws : IOException - If an I/O error occurs

```
1  BufferedReader in = null;
2  try {
3      in = new BufferedReader( new FileReader("tatouine.txt") );
4
5      String buf = in.readLine();
6      while(buf != null){
7          System.out.println(buf);
8          buf = in.readLine();
9      }
10 }...
```


Stratégie de lecture des fichiers ASCII

Une fois la ligne lue, il est nécessaire de la traiter...

- Séparer les mots : classe StringTokenizer
  - choix des séparateurs (espace, tabulation, virgule...), accès aux sous-chaines
- Conversion avec la classe Double :

```
public static Double valueOf(String s) throws NumberFormatException

public static double parseDouble(String s) throws
        NumberFormatException
```

- Fonctions similaires avec la classe Integer,...

Classes pour les flux : premier bilan (2)

- Package java.io.*
- Classe File : créer un objet pour représenter un fichier
  - gérer un fichier ou un répertoire (créer, lire,...)
- Flux : connexion entre le programme et un support
  - flux entrant : input
    - classes abstraites : InputStream (général), Reader (caractères)
    - accès : FileInputStream (octet), FileReader (caractères)
  - flux sortant : output
    - classes abstraites : OutputStream (général), Writer (caract.)
    - accès : FileOutputStream (octet), FileWriter (caractères)
- Accès de haut niveau à un flux
  - lecture par type : DataInputStream
  - écriture par type : DataOutputStream
  - ASCII : accès par ligne : BufferedReader, BufferedWriter
    - combiner avec StringTokeniser
  - ASCII : encore plus souple : Scanner

Ecriture de fichiers ASCII

- Classe BufferedWriter
- Mêmes principes que BufferedReader mais pour écrire

```
1 BufferedWriter writer =
2     new BufferedWriter(
3         new FileWriter(new File("monFichierASCII.txt")));
4
5 writer.write("Vive la POO !");
6 writer.close();
```


Vers les bases de données...

Fichier structuré

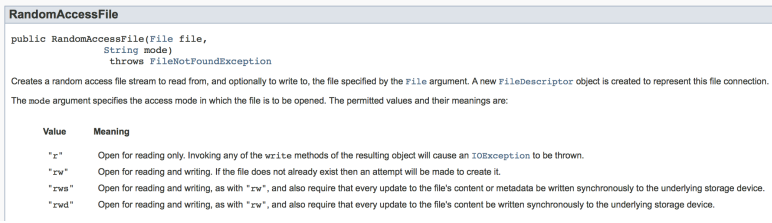
Dans certains fichiers, on souhaite stocker des données structurées, par exemple :

- des matrices de chiffres,
- des fichiers clients avec des colonnes structurées (eg : nom, prénom, ...)
- ...

En général, on souhaite faire des traitements associés...
Il faut aller assez vite

Classe RandomAccessFile

- Création/Ouverture assez classique



RandomAccessFile

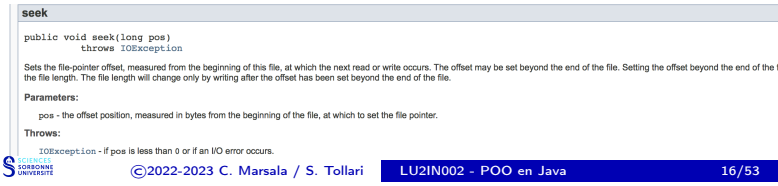
```
public RandomAccessFile(File file,
                        String mode)
                 throws FileNotFoundException
```

Creates a random access file stream to read from, and optionally to write to, the file specified by the `File` argument. A new `FileDescriptor` object is created to represent this file connection.

The `mode` argument specifies the access mode in which the file is to be opened. The permitted values and their meanings are:

Value	Meaning
`"r"`	Open for reading only. Invoking any of the `write` methods of the resulting object will cause an `IOException` to be thrown.
`"rw"`	Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
`"rws"`	Open for reading and writing, as with `"rw"`, and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.
`"rwd"`	Open for reading and writing, as with `"rw"`, and also require that every update to the file's content be written synchronously to the underlying storage device.

  - Option `"r", "w", "rw"`
- Fonction de déplacement du curseur dans le fichier



seek

```
public void seek(long pos)
          throws IOException
```

Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. The offset may be set beyond the end of the file. Setting the offset beyond the end of the the file length. The file length will change only by writing after the offset has been set beyond the end of the file.

Parameters:

`pos` - the offset position, measured in bytes from the beginning of the file, at which to set the file pointer.

Throws:

`IOException` - if `pos` is less than `0` or if an I/O error occurs.

RandomAccessFile : exemple d'écriture

```
1  import java.io.RandomAccessFile;
2  [...]
3  String fname = "monfichiertest.dat";
4  File f = new File(fname);
5  RandomAccessFile raf = new RandomAccessFile(f, "rw");
6
7  raf.writeChars(String.format("%4s", "titi"));
8  raf.writeChars(String.format("%10s", "toto"));
9  raf.writeDouble(Math.PI);
10 raf.writeChars(String.format("%10s", "tata"));
11 raf.close();
```

Fichier produit :

```
1 titi      toto          !ûTD      tata
```


RandomAccessFile : exemple de lecture

Fichier produit :

```
1 titi      toto          !ûTD      tata
```

```
1         RandomAccessFile raf2 = new RandomAccessFile(f, "rw");
2
3         for(int i=0; i<4; i++)
4             System.out.print(raf2.readChar());
5         System.out.println(); // titi
6
7         raf2.seek(28);
8         System.out.println(raf2.readDouble()); // 3.14...
9
10        raf2.seek(8);
11        for(int i=0; i<10; i++)
12            System.out.print(raf2.readChar()); //      toto
13        System.out.println();
14
15        raf2.seek(0);
16        System.out.println(raf2.readLine());
17        //titi      toto          !ûTD      tata
18
19        raf2.close();
```

Les flux : pas seulement pour les fichiers

- Lire une String comme un flux avec la classe StringReader

```
String s = "blablabla...";
// initialisation à partir d'une String
StringReader sread = new StringReader(s);
System.out.println(sread.read()); // -> 98 (code de 'b')
```

- Fonctionnalité très utile pour unifier une chaîne de traitements
  - tous les types d'entrées peuvent être des flux, on peut jouer avec l'héritage

La console est un flux (les E/S standards)

- Entrée console = un flux particulier (System.in)

```
// créer un Scanner, choisir un délimiteur
try {
    s = new Scanner(System.in);
    s.useDelimiter("␣");
    while (s.hasNext()) {
        System.out.println(s.next());
    }
} catch (...) {
} finally {
    if (s != null) {
        s.close();
    }
}
```

NB : les saisies consoles ne sont validées qu'après un retour chariot

1. Tapons le texte : Vendredi 9 décembre [ENTREE]
2. Après chaque espace, rien ne se passe
3. Après le retour, la séparation a été prise en compte :

```
Vendredi
9
décembre
```


Outil avancé pour la console

- Objet Console

```
Console c = System.console(); // Singleton ! renvoie la référence
if (c == null) {
    System.err.println("No␣console.");
    System.exit(1);
}

String login = c.readLine("Enter␣your␣login:␣");
char [] oldPassword = c.readPassword("Enter␣your␣old␣password:␣");
```

- Notons la forme particulière de la construction de la console
- La console est gérée comme un singleton

Lecture HTTP sur le web

- Lire sur internet est une simple formalité...
  - outil pour les URL → flux
  - ouverture / lecture classique !

```
URL url = new URL("http://www.yahoo.fr"); // outil ad'hoc
BufferedReader bf=new BufferedReader(
   new InputStreamReader(url.openStream(), "utf8"));
String buf = bf.readLine();
while(buf != null){
    buf = bf.readLine();
    System.out.println(buf);
}
bf.close();
```

Collecte de données sur le web

On peut mettre les données lues aux formats :

- String : utilisation des méthodes pour sélectionner les parties pertinentes
- XML : bibliothèque DOM
- brutes dans un fichier pour des traitements ultérieurs
- ...

Vers les BD, SQL

- Connection aux bases de données aisée

```
// le driver gère la connection (à MySQL par exemple)
Connection con = DriverManager.getConnection(
                        "jdbc:myDriver:myDatabase",
                        username,
                        password);

// instanciation de la connection
Statement stmt = con.createStatement();
// envoi d'une requête
ResultSet rs = stmt.executeQuery("SELECT␣a,␣b,␣c␣FROM␣Table1");
// traitement séquentiel des réponses
    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
```


Plan du cours

Problématique générale

- Construire une structure de données adaptée à différents types d'entrées
  - Listes : d'entier, de réels, de String...
  - Piles, Files, Tas...
  - ⇒ ne pas construire une classe par cas!!!
- Solution (avant Java 1.5) :

```java
public class ListeGenOld {
    private final static int TAILLE_MAX = 500;
    private Object[] liste;
    private int size;

    public ListeGenOld(){
        liste = new Object[TAILLE_MAX];
        size  = 0;
    }

    public void add(Object o){liste[size] = o; size ++;}
    public Object get(int i){return liste[i];}
    ... // remove, getSize...
}
```


Limites

- Obligation de faire des cast à chaque récupération d'objet
- Sécurisation *bof* : on peut mettre n'importe quoi dans la structure... + cast à la récupération
- Difficilement compatible avec des algorithmes génériques (tri, min, max...)

⟹ Dans les versions ultérieures : les conteneur génériques

Le cas ArrayList (usage)

La plupart des structures de données classiques existent déjà

- ArrayList, HashMap, HashSet, PriorityQueue

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Usage :

- ArrayList<E> = liste de E, est un type à part entière
  - déclaration, instanciation classique

```java
// sur un exemple avec des Integer
ArrayList<Integer> maListe = new ArrayList<Integer>();
```

- get : retourne des Integer (pas besoin de cast)
- on ne peut mettre que des Integer dedans

Création d'une classe générique...

Exemple (très) utile : la Paire

La plupart des langages modernes gèrent des N-uplet... Mais pas Java. On peut créer une classe Paire pour retourner facilement plusieurs valeurs depuis une méthode.

```java
public class Paire<A,B> {
    private A el1;
    private B el2;
    public Paire(A el1, B el2) {
        super();
        this.el1 = el1;
        this.el2 = el2;
    }
    public A getEl1() {
        return el1;
    }
    public B getEl2() {
        return el2;
    }
}
```


... Et usage client

La syntaxe est celle des ArrayList... Vous la connaissez déjà !

```java
public static void main(String[] args){
  // Un premier exemple de Paire:
  Paire<Integer, String> p = new Paire<Integer, String>(2, "toto");
  System.out.println(p.getEl1()+"␣"+p.getEl2());

  // Un autre exemple de Paire:
  Paire<Float, Float> pflo = new Paire<Float, Float>(4.2, 11.38);
  System.out.println(pflo.getEl1()+"␣"+pflo.getEl2());
}
```

- Le type est donc :
  - dans un cas Paire<Integer, String>
  - et dans l'autre cas Paire<Float, Float>
- Le type du contenant est passé en argument *spécial* entre <>

Extension d'une classe générique (1)

Besoin d'une liste avec des méthodes spécifiques... Mais toujours générique

exemple récupération de la case du milieu

```java
public class MaListeMilieu<E> extends ArrayList<E>{
  // constructeur sans argument par défaut
  // Les méthodes sont héritées: add, get, size...

  // Méthode spécifique
  public E getMilieu(){
      return super.get(super.size()/2); // division entière
  }
}
```

Usage client :

```java
public static void main(String[] args){
  MaListeMilieu<Double> li = new MaListeMilieu<Double>();
  li.add(2.0); li.add(1.4); li.add(3.7);
  System.out.println(li.getMilieu());
}
```

Extension d'une classe générique (2)

Besoin d'une liste de *quelque chose*

- créer un aquarium = liste de poissons
- Train = liste de wagons
- Population = liste de personnes
- ...

```
public class Aquarium extends ArrayList<Poisson>{
// bcp de méthodes héritées !!!
}
```

Usage client : la liste ne gère que des poissons

```
public static void main(String[] args){
    Aquarium aqua = new Aquarium();
    aqua.add(new Thon());
    aqua.add(new Requin());
    ...
}
```


Cast sur les objets génériques <E>

1. Coté *contenu* : très agréable (et classique)
   - objets de types E et descendants de E
   - récupération d'objets dans des variables E

2. Coté *contenant* : non flexible

```
ArrayList<Personne> pop = new ArrayList<Personne>(); // OK

// Etudiant extends Personne
ArrayList<Personne> promo = new ArrayList<Etudiant>(); // KO
```

⇒ Une seule issue (en cas de besoin) : la syntaxe *wildcard*

La syntaxe *wildcard*

Subsomption *contenu/contenant*

On a besoin de cette propriété pour définir des algorithmes génériques.
Syntaxe : `ArrayList<?>` ou `ArrayList<? extends Poisson>`
N'importe quelle liste, ou n'importe quelle liste d'objets dérivés de poissons :

```
ArrayList<? extends Poisson> li = new ArrayList<Thon>();
```

Exemple : comment proposer une technique de recherche de minimum dans une liste sans connaître le type de contenu ?

1. Définir une propriété (interface) : Comparable
2. Définir un algorithme acceptant n'importe quel conteneur d'objets comparables en utilisant la syntaxe *wildcard*

ATTENTION : ce type de syntaxe empêche toute modification sur l'objet passé

La syntaxe *wildcard* (préliminaires)

1. Définir une propriété : Comparable

```
public interface Comparable<E> {
    //retourne -1 si ref < obj, 0 si égalité, 1 sinon
    public int compareTo(E obj);
}
```

Avec par exemple un `Poisson` répondant à la spécification :

```
public class Poisson implements Comparable<Poisson>{
    private double taille;

    public Poisson(double taille) {
        super();
        this.taille = taille;
    }
    public double getTaille() {
        return taille;
    }
    public int compareTo(Poisson obj) {
        if(taille<obj.taille) return -1;
        else if(taille==obj.taille) return 0;
        return 1;
    }
}
```


La syntaxe *wildcard*

2. Utiliser la propriété dans un algorithme générique :

```
public class GenericTools<E extends Comparable<E>> {

    // constructeur par défaut
    public E getMinimum(ArrayList<? extends E> liste){
        E min = liste.get(0);
        for (int i=1; i<liste.size(); i++){
            // si : min > liste.get(i)
            if(min.compareTo(liste.get(i)) == 1)
                min = liste.get(i);
        }
        return min;
    }
}
```

Usage coté client :

```
public static void main(String[] args){
    ArrayList<Poisson> aquarium = new ArrayList<Poisson>();
    GenericTools<Poisson> tool = new GenericTools<Poisson>();
    Poisson lePlusPetit = tool.getMinimum(aquarium);
}
```


Algorithmes génériques

Classe algorithmique de gestion des listes générique

`Collections`

- min, max, sort, shuffle, indexOf, frequency...

Quelques exemples d'outils disponibles :

static int	frequency(Collection<?> c, Object o) Returns the number of elements in the specified collection equal to the specified object.
static int	indexOfSubList(List<?> source, List<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T extends Object & Comparable<? super T>> T	min(Collection<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static void	shuffle(List<?> list) Randomly permutes the specified list using a default source of randomness.
static <T extends Comparable<? super T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the natural ordering of its elements.
static <T> void	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.

Plan du cours

De la Bidouille OO à la Programmation OO

Principes Objets :

- Encapsulation
- Héritage & abstraction
- Polymorphisme

Question (difficile)

Comment combiner ces différents outils pour écrire des programmes ou boîtes-à-outils flexibles, réutilisables, efficaces, etc... ?

Patrons de conception (design patterns)

Définition : design pattern

Élément de conception réutilisable permettant de résoudre un problème récurrent en programmation orientée objet.

- Historique :
  - analogie avec une méthode de conception d'immeuble en architecture [Alexander, 77]
  - introduites par le "GOF" dans le livre Design Patterns en 1999
    - dans le "GOF" 23 patterns standards

Pourquoi les patterns ?

Des recettes d'expert ayant fait leurs preuves.
Un vocabulaire commun pour les architectes logiciels.
Approche incontournable dans le monde de la P.O.O.

Remarque : certains patterns sont déjà inclus dans le langage...

Idiomes de la programmation

Portion de code Java que l'on utilise lorsque l'on a à résoudre un problème récurrent

- Parcours d'un tableau

```
for(int i=0;i<tableau.length;i++) {
   tableau[i] = ...
}
```

- Gestion d'exception

```
try {
  XYZ(...);
} catch(XYZException e) {
      e.printStackTrace(System.err);
}
```


Conception vs. Programmation

Un design pattern est un élément de conception (objet)
Remarque : ce n'est pas au niveau du langage de programmation (donc pas spécifique à Java, un pattern est aussi valable en C++, en Python...)

Citations

"Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière" Christopher Alexander - 1977.

"Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" [Buschmann] - 1996.

Coté implémentation : grands principes

Principe 1

Favoriser la composition (lien dynamique, flexible) sur l'héritage (lien statique, peu flexible) La délégation est un exemple d'outil pour la composition

Attention : favoriser ne veut pas dire remplacer systématiquement, l'héritage est largement utilisé aussi !

Principe 2

Les clients programment en priorité pour des interfaces (ou abstractions, classes abstraites, etc) plutôt qu'en lien direct avec les implémentations (classes concrètes)

Différents types de patterns

- Pattern de création : utilisé pour la création d'objets
  - par exemple : singleton, prototype, factory,...
- Pattern structurel : utilisé pour rajouter des fonctionnalités à une classe
  - par exemple : decorator, composite,...
- Pattern comportemental : utilisé pour mettre en œuvre des moyens de communication entre objets
  - par exemple : iterator, strategy,...

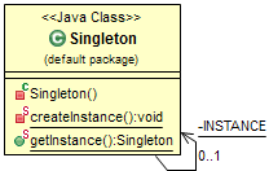
Description standard

Nom (classification)	singleton (création)
Intention	description générale et succinte
Alias	autres noms connus pour le pattern
Motivation	au moins 2 exemples/scénarios
Indications d'utilisation	liste des situations qui justifient de l'utilisation du pattern
Structure	diagramme de classe UML indépendant du langage
Constituants	explication des différentes classes intervenant dans le pattern
Implémentation	principes, pièges, astuces, techniques pour implanter le pattern dans un langage objet donné
Utilisations remarquables	programmes réels qui l'utilisent
Limites	limites concernant son utilisation

Singleton (Construction)

Garantir l'unicité d'une instance

- A utiliser quand il ne peut y avoir qu'une instance (eg Console)
- Fournir un accès à cette instance



Singleton (Construction)

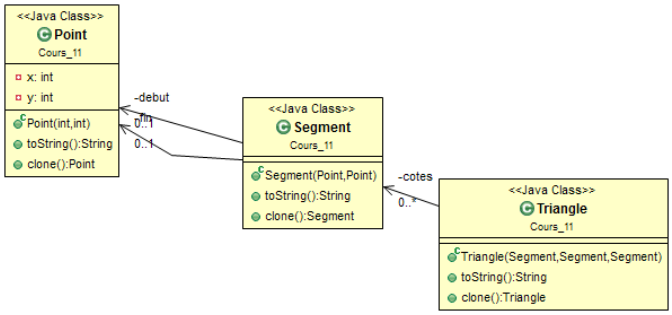
```
1  public class Singleton {
2      private static Singleton INSTANCE = null;
3
4      private Singleton() {}
5
6      private static void createInstance() {
7          if (INSTANCE == null) {
8              INSTANCE = new Singleton();
9          }
10     }
11     public static Singleton getInstance() {
12         if (INSTANCE == null) createInstance();
13         return INSTANCE;
14     }
15 }
```


Prototype (Construction)

Créer un nouvel objet à partir d'un objet existant

- Fournir une copie exacte de l'objet existant
- Fournir une interface commune aux objets

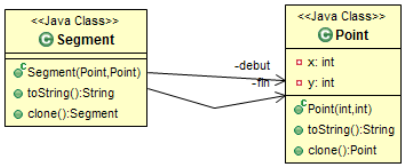
$\Longrightarrow$ clonage ou constructeur par recopie



Composite (Structure)

Un objet E, composé d'objets E

- une sous-figure, composée de figure
- une stratégie, composée de sous-stratégie




```
1  public class Segment {
2          private Point debut, fin;
3          public Segment(Point p1, Point p2) {
4                  debut = p1; fin = p2;
5          }
6
7          public String toString() {
8                  return "Segment [debut=" + debut + ", fin=" + fin
9          }
10         public Segment clone() {
11                 return new Segment(debut.clone(), fin.clone());
12         }
13 }
```

Iterator (comportement)

Objectif

- Découpler le choix des structures de données des implémentations d'algorithmes
- Proposer une interface de parcours d'un ensemble d'objets quelle que soit la structure associée (liste, pile, arbre,...)

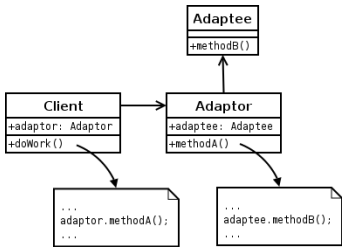
```
// code d'utilisation spécifique
Iterator<MyType> iter = list.iterator();

// code commun à toute structure de données
while (iter.hasNext()) {
    System.out.print(iter.next());
    if (iter.hasNext())
        System.out.print(",␣");
}
```


Adapter (Structurel)

Idée : réutiliser une fonction déjà implémentée...

... Mais dans une architecture contrainte
= Adapter la classe
Repose sur le principe de la délégation



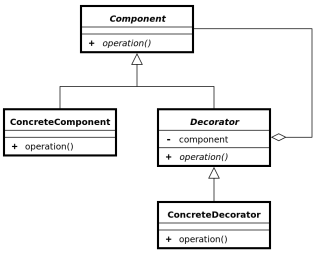
- Très très fréquent : réutiliser une classe sans la modifier... Alors que le client est déjà spécifié
  - on a un objet d'une classe Client qui veut utiliser un objet de classe B mais le client ne sait manipuler des objets de classe A. Il faut donc adapter la classe B à l'interface de A pour que le client puisse finalement utiliser B.
- Technique d'optimisation, calcul de graphes...
  - Adapter les objets à passer en paramètres... Ou adapter les algorithmes.

Decorator (Structure)

Ajouter une fonctionnalité...

... sur n'importe quel objet d'une arborescence de classe

- Permettre de lire des informations de haut niveau (String, double...) dans n'importe quel flux
- Rendre n'importe quelle stratégie prudente



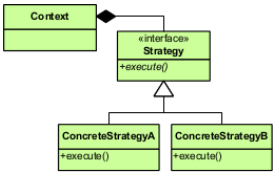

```
public MonObjetDecore extends MonObjet{
  private MonObjet obj;
  public MonObjetDecore(MonObjet obj){ this.obj = obj;}

  public void mafonction(){
    if(cas 1) return obj.mafonction()
    else // code spécifique
  }
```


Strategy (Comportement)

Gérer le comportement distinctement de l'objet

- Robot... Qui marche, rampe, vole... Ou une combinaison des 3
- Le client gère la stratégie
- On peut créer de nouvelles stratégies avec des robots existants




```
public class Robot{
  private Strategy str;
  public Robot(Strategy str){this.str = str;}

  public void action(){
    str.action(); // ou str.action(this);
  }
```

⇒ Une alternative (beaucoup plus flexible) à l'héritage sur les robots
Note : DP compatible avec `Composite`

Et plein d'autres encore

- Visitor
  - vient éxecuter un algorithme dans un objet
- MVC : model view controler
  - pour les interfaces graphiques, séparation des éléments clés
- ...

Comme un second niveau de programmation (de la conception en fait !)
A continuer avec de l'UML et d'autres UE de génie logiciel

- UE L3 LU3IN002 "Programmation par objets"