

Rappels programmation orientée objet et Java : partie 2

Arnaud Labourel arnaud.labourel@univ-amu.fr



Section 1

Le penser objet

Analyse d'un problème en programmation objet

Penser objet : décomposer le programme en objets

- Quels sont les objets nécessaires à la résolution du problème ?
⇒ décomposition du problème en objets
- À quels modèles des objets correspondent-il ?
⇒ Quelles sont les classes ?
- Quels sont les fonctionnalités/opérations dont on doit/veut pouvoir disposer sur ces objets ?
⇒ Quelles sont les méthodes des classes ?
- Quelle est la structure des données de l'objet ?
⇒ Quelles sont les attributs des classes ?

Exemple de problème

- un catalogue regroupe des articles, il permet de trouver un article à partir de sa référence.
- un article est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- une commande est créée pour un client et un catalogue donnés, on peut ajouter des articles à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un client peut créer une commande pour un catalogue et commander dans cette commande des articles à partir de leur références.

Classes du problèmes

- un **catalogue** regroupe des **articles**, il permet de trouver un article à partir de sa **référence**.
- un **article** est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- une **commande** est créée pour un **client** et un **catalogue** donnés, on peut ajouter des **articles** à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un **client** peut créer une **commande** pour un catalogue et commander dans cette commande des **articles** à partir de leur références.

- un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.
- un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre
- une commande est créée pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leur références.

Méthodes de la classe Catalog

Description d'un catalogue

un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.

Méthodes :

- Item getItem(String reference)

Méthodes de la classe Item

Description d'un article

un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre

Méthodes :

- `double getPrice()`
- `String getReference()`
- `boolean isMoreExpensiveThan(Item other)`

Constructeurs et méthodes de la classe Order

Description d'une commande

une commande est **créée** pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.

Méthodes et constructeurs :

- `Order(Client client, Catalog catalog)` (constructeur)
- `void addItem(Item item)`
- `List<Item> allItems()`
- `double getTotalPriceOfItems()`
- `double getShippingCost()`
- `Client getClient()`
- `Catalog getCatalog()`

Constructeurs et méthodes de la classe Client

Description d'un client

un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leur références.

Méthodes :

- `Order createOrder(Catalog catalog)`
- `void orderItem(Order order, String reference)`

Section 2

Vocabulaire programmation objet

Un **objet** :

- peut être **construit**
- est **structuré** : il est constitué d'un ensemble d'**attributs** (données de l'objet)
- possède un **état** : la valeur de ses attributs
- possède une **interface** : les opérations applicables appelées **méthodes**

Objet et référence

- Chaque objet est identifié par une adresse mémoire appelée **référence**
- C'est cette référence qui permet d'accéder à l'objet

Syntaxe des objets

Construire un objet

```
Classe objet = new Classe(arguments);
```

- Après l'appel au constructeur et l'affectation, la variable contient la référence de l'objet
- Sans appel au constructeur, la variable contient `null` (adresse pointant vers rien)

Accéder à un attribut d'un objet

```
objet.attribut = ...;
```

Appeler une méthode d'un objet

```
objet.méthode(arguments)
```

Attention au null

Si vous n'avez pas construit l'objet, tout accès à :

- une de ses méthodes
- un de ses attributs

générera une `NullPointerException` (davantage de détails sur les exceptions dans un cours ultérieur).

Règles d'or

- Toujours penser à construire les objets qu'on a besoin.
- Éviter si possible d'utiliser `null` dans son code.

Une **classe** (d'objet) définit des :

- **constructeurs** : des façons de construire/instancier les objets (**instances** de la classe)
- **attributs** (champs, propriétés ou données membres) : la structure des objets de la classe
- **méthodes** : le comportement des objets de la classe

Exemple de syntaxe de définition d'une classe

```
public NameOfTheClass{  
    public final Type1 attribute1 = new Type1(arguments);  
    private Type2 attribute2;  
  
    public NameOfTheClass(arguments){  
        // constructor code  
    }  
  
    public Type2 getAttribute2(){  
        return this.attribute2;  
    }  
}
```


Déclaration d'attributs

Exemples **sans** initialisation

```
private int count;  
private final Color backgroundColor;  
private final String name;
```

Exemples **avec** initialisation

```
private int count = 0;  
private final Color backColor = new Color(0.5,0.2,0);  
private final String name = "John Doe";
```

⇒ objets créés avec des valeurs de bases pour les attributs.

Mot-clés

• `private` : accessible que dans la classe

Déclaration d'une méthode

```
visibilité + type de retour + nom +  
    (type paramètre 1 + nom paramètre 1, ...) + {  
    instructions  
}
```

Exemples

```
public int getCount() { ... }  
public void setCount(int newCount) { ... }  
public void launchMissile(GPSCoordinate coord) { ... }  
private Vector3D getAcceleration() { ... }
```

Corps d'une méthode

Le corps d'une méthode ou d'une classe est composé :

- d'*instructions*,
- de *structures de contrôle*, permettant de décider quelles instructions exécuter.

Remarques

- Sans structure de contrôles, les instructions sont exécutées une par une, dans le sens de lecture.
- On écrit généralement une instruction ou une structure de contrôle par ligne.
- On utilise l'*indentation* pour mettre en valeur les structures de contrôle.

Déclaration d'une variable

Une *variable* est un espace mémoire nommé pour stocker une valeur *momentanément*.

Portée : une variable (qui n'est pas un attribut) existe :

- depuis le moment où elle est déclarée
- jusqu'à la sortie du bloc { ... } où elle est définie.

Remarques

- chaque variable possède :
 - ▶ un type (qui peut correspondre à un type primitif, une classe ou une interface), et ne peut contenir que des valeurs de ce type.
 - ▶ une valeur (les variables non-affectées ont une valeur par défaut : 0, null, ...)

```
int count;
```

```
Vector2D position;
```

L'affectation

L'*affectation* est l'opération consistant à mettre une valeur dans un espace mémoire défini par une variable (**l-value**) :

L'affectation est soumise aux contraintes de types: la valeur stockée doit être d'un type correspondant à celui déclaré.

```
count = 12;  
count = count + 4;  
position = new Vector(3, -2.5);
```

La création d'un nouvel objet se fait avec le mot-clé `new`, qui déclenche :

- ❶ l'allocation d'un espace mémoire pour contenir le nouvel objet,
- ❷ l'initialisation des propriétés de cet objet avec les valeurs de bases,
- ❸ l'appel au constructeur de l'objet,
- ❹ le retour de la *référence* (adresse mémoire) de l'objet par l'instruction

```
new Vector(3,-2.5);  
new Spaceship();  
new ArrayList<Integer>();
```

Le mot-clé return

Une méthode peut retourner une valeur lorsqu'elle a terminé son travail, en utilisant le mot-clé `return`.

Lors de l'évaluation de l'instruction `return`, l'expression après le `return` (le résultat) est *d'abord* évaluée (si elle existe), *ensuite* l'exécution de la méthode est interrompue.

Le résultat est retourné au niveau de l'instruction ayant fait l'appel de méthode.

Exemples d'utilisation de return

```
return;  
return 42;  
return new Vector(-5, 2.333);
```

Les expressions

Les valeurs sont définies par des *expressions* :

- des littéraux : 0.01, -42, "toto", ...
- combinaison d'expressions par des opérateurs +, *, /, -
- variables et attributs,
- appels de méthode (ayant un type de retour autre que void) ou de constructeurs

Les expressions apparaissent :

- en membre droit d'une affectation,
- en paramètre d'un appel de méthode, d'un constructeur,
- après le mot-clé return

Section 3

À retenir absolument

Une **classe** (d'objet) définit des :

- **constructeurs** : des façons de construire/instancier les objets (**instances** de la classe)
- **attributs** (champs, propriétés ou données membres) : la structure des objets de la classe
- **méthodes** : le comportement des objets de la classe

Syntaxe de définition d'une classe

```
public NameOfTheClass{  
    public final Type1 attribute1;  
    private Type2 attribute2;  
  
    public NameOfTheClass(...){  
        // constructor code  
    }  
  
    public getAttribute2(){  
        return this.attribute2;  
    }  
}
```

Section 4

Classes utiles et types primitifs

Classe Object

Par défaut, les classes étendent la classe `Object` de Java et ont donc les méthodes suivantes que l'on peut redéfinir :

- `boolean equals(Object obj)`: Indicates whether some other object is “equal to” this one.
- `String toString()`: Returns a string representation of the object.
- ...

Test d'égalité

- `o1 == o2` : vrai si `o1` et `o2` sont le même objet et faux sinon
- `o1.equals(o2)` : vrai si `o1` et `o2` correspondent à deux objets considérés égaux (exemple : étudiant ayant le même id, chaîne de caractères ayant les mêmes caractères, ...)

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31} - 1$	0
long	entier	64 bits	-2^{63} à $2^{63} - 1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	false ou true	false

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Cela un impact différent suivant que l'argument soit un objet ou un type primitif :

- Pour les objets, cela signifie passer une copie de la référence : il est donc possible de modifier l'état de l'objet.
- Pour les types primitifs, cela signifie que l'argument est une copie uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Tableaux unidimensionnels

En Java, les tableaux sont des objets (et donc des références).

Déclaration d'une variable de type "référence vers un tableau" :

```
int[] arrayOfInt;  
double[] arrayOfDouble;
```

Construction d'un tableau :

```
arrayOfInt = new int[10]  
arrayOfDouble = new double[3];
```

Utilisation d'un tableau :

```
arrayOfInt[0] = 5;  
arrayOfInt[9] = 10;  
arrayOfDouble[2] = arrayOfInt[0] / arrayOfInt[9];
```


Tableaux multidimensionnels

Déclaration :

```
int[] [] matrixOfInt;
```

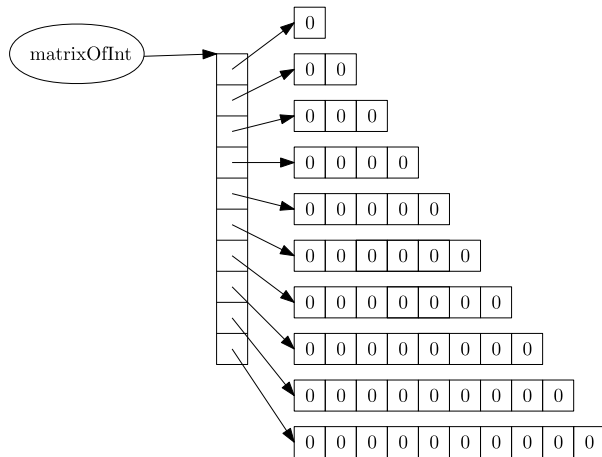
Construction :

```
matrixOfInt = new int[10] [];  
for(int row = 0; row < matrixOfInt.length; row++)  
    matrixOfInt[row] = new int[5];  
/* ou directement */  
matrix = new int[10][5];
```

Que produit le code suivant (réponse au transparent suivant) ?

```
matrixOfInt = new int[10] [];  
for(int row = 0; row < matrixOfInt.length; row++)  
    matrixOfInt[row] = new int[row + 1];
```

Tableau de tableaux



Règles pour les tableaux en Java

```
TypeOfTheElements [] [] [] [] array = new TypeOfTheElements [x] [y] [z] [] ;
```

Règles

- Le type des éléments d'un tableau peut être n'importe quel type (primitif ou objet).
- Le nombre de [] définit la dimension du tableau.
- On peut construire un tableau en définissant une taille (positive ou nulle) pour au moins une dimension.

Chaînes de caractères (1/2)

La classe `String` permet de définir des chaînes de caractères invariables (**immutable**)

Déclaration et création :

```
String hello = "Hello";
```

```
String world = "World";
```

Concaténation :

```
String helloWorld = hello + " " + world + " ! ";
```

```
int integer = 13;
```

```
String helloWorld1213 = hello + " " + world + " "  
                        + 12 + " " + integer;
```

Chaînes de caractères (2/2)

Affichage :

```
System.out.print(helloWorld);    // affiche "Hello World !"
System.out.println(helloWorld);  // affiche "Hello World !"
                                  // avec retour à la ligne
```

Comparaison :

```
String a1 = "a";
String a2 = "a";
String a3 = new String("a");
System.out.println(a1==a2);      // affiche "true"
System.out.println(a1==a3);      // affiche "false"
System.out.println(a1.equals(a3)); // affiche "true"
```

Les classes `ArrayList` et `LinkedList` : permettent de créer des listes en java.

```
List<String> strings = new ArrayList<>();  
string.add("first");  
string.add("second");  
System.out.println(strings);  
// affiche "[first, second]"
```

Section 5

Structures de contrôles

La structure conditionnelle : if

La **structure conditionnelle** permet d'exécuter un bloc d'instructions que si une condition est réunie.

```
if (people.isGoodGuy()) {  
    System.out.println("Greetings, " + people.getName() + "!");  
}
```

```
if (people.isGoodGuy()) {  
    System.out.println("Greetings, " + people.getName() + "!");  
} else {  
    System.out.println("You are not welcome here, "  
        + people.getName() + ".");  
}
```


La répétition “tant que” : while(condition)

Le while permet de répéter un bloc d'instructions tant que sa condition est vraie :

```
int count = 10;
while (count > 0) {
    System.out.println(count + "...");
    count = count - 1;
}
System.out.println("BOUM!!!");
```

Attention

Le test n'est effectué que juste avant d'effectuer la première instruction du bloc à chaque itération !

La répétition pour toujours (*forever*) : `for(;;)`

La boucle `for(;;)` permet de répéter un bloc d'instructions tant qu'on ne quitte pas la boucle avec les instructions `break` ou `return`.

```
int count = 10;
for(;;) {
    System.out.println(count + "...");
    if (count <= 0) break;
    count = count - 1;
}
System.out.println("BOUM!!!");
```

Note

on peut avoir plusieurs instructions `break` dans la même boucle.

La répétition bornée : for

Dans certains cas, il est plus direct d'utiliser une boucle for, qui isole :

- l'initialisation de la boucle,
- la condition d'arrêt,
- l'instruction de progression.

```
for (int count = 10; count > 0; count = count - 1) {  
    System.out.println(count + "...");  
}  
System.out.println("BOUM!!!");
```

On peut aussi utiliser break et return dans une boucle for.

L'itération pour chaque : for

Pour les objets qui sont des **collections**, implémentant l'interface `Iterable<Elt>`, on peut utiliser la boucle `for` ainsi:

```
for (Elt item : myCollection) {  
    item.doSomething();  
    // ...  
}
```

Exemple de collections

- `List<Elt>`
- les tableaux

Imbrication des structures de contrôle

Les structures de contrôles définissent des blocs d'instructions, entre { et } :

- possible d'utiliser d'autres structures de contrôle dans ces blocs, par **imbrication**,
- les blocs définissent les **portées** des variables, une variable ne vit que dans le bloc où elle est définie et les blocs imbriqués dedans.
- les structures de contrôle compliquent la lecture du code par un humain : il faut éviter l'**imbrication** !

Section 6

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(Object object) {  
        stack[size] = object; size++;  
    }  
  
    public Object pop() {  
        size--;  
        Object object = stack[size];  
        stack[size]=null; // Pour le Garbage Collector.  
        return object;  
    }  
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();  
String string = "truc";  
stack.push(string);  
string = (String)stack.pop();  
// Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();  
Integer intValue = new Integer(2);  
stack.push(intValue);  
String string = (String)stack.pop();  
// Erreur à l'exécution
```


La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();  
String string = "truc";  
stack.push(string); // Le paramètre doit être un String.  
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe Stack qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypes automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(T element) {  
        stack[size] = element;  
        size++;  
    }  
    public T pop() {  
        size--;  
        T element = (T)stack[size];  
        stack[size] = null;  
        return element;  
    }  
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (wrapper class) `Integer` :

Interdit : ~~`Stack<int> stack = new Stack<int>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<Integer>();  
int intValue = 2;  
Integer integer = new Integer(intValue);  
// → emballage du int dans un Integer.  
stack.push(integer);  
Integer otherInteger = stack.pop();  
int otherIntValue = otherInteger.intValue();  
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31} - 1$
long	Long	64 bits	-2^{63} à $2^{63} - 1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	false ou true

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Byte valueOf(byte b)`
- `public static char charValue()`
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();  
int intValue = 2;  
stack.push(intValue);  
// → emballage automatique du int dans un Integer.  
int otherIntValue = stack.pop();  
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des new soient présents dans le code.

Exemple : liste chaînée générique

On considère une liste chaînée de String

```
public class LinkedList {  
    private class Node {  
        private String data;  
        private Node next;  
        public Node(String data, Node next) {  
            this.data = data;  
            this.next = next;  
        }  
    }  
    private Node first = null;  
    public void add(String data) {  
        first = new Node(data, first);  
    }  
}
```

Exemple : liste chaînée générique

Nous la transformons en classe paramétrée de la façon suivante :

```
public class LinkedList<T> {  
    private class Node {  
        private T data;  
        private Node next;  
        public Node(T data, Node next) {  
            this.data = data;  
            this.next = next;  
        }  
    }  
    private Node first = null;  
    public void add(T data) {  
        first = new Node(data, first);  
    }  
}
```

Plusieurs paramètres de types

```
public class Pair<A, B> {  
    public A first;  
    public B second;  
    public Pair(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
    public static <A, B> Pair<A,B>  
        makePair(A first, B second) {  
        return new Pair<A,B>(first, second);  
    }  
}
```

Utilisation d'une classe avec plusieurs paramètres de types

```
public class Main {  
  
    public static void main(String[] args) {  
        Pair<String,Integer> pair =  
            Pair.<String,Integer>makePair("tot",12);  
        System.out.println(pair);  
    }  
}
```

Section 7

Une méthodologie pour bien nommer

Pourquoi bien nommer est important

Albert Camus (1944)

“Mal nommer un objet, c’est ajouter au malheur de ce monde”

Que veut dire le texte suivant ?

La L3 info : MIAGE ne dépend pas de la même UFR que la L2 info : MI, elle dépend de la FEG et non de la FS. Pour faire vos IA et IP, vous devez donc contacter la scolarité de Forbin et non celle de SCH.

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage (1/2)

En anglais

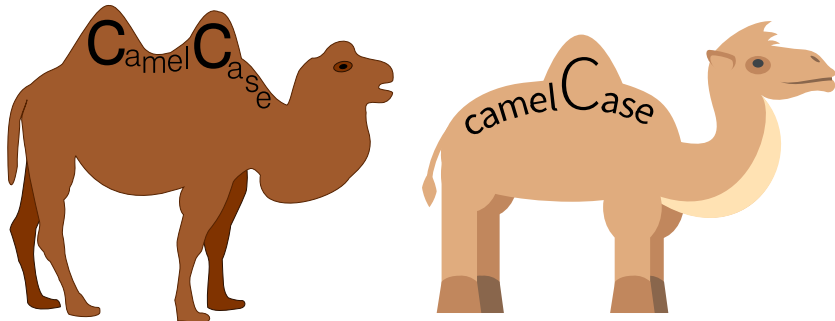
- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les bibliothèques standards).
⇒ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

En “camel case”

Un nom composé de plusieurs mots n'utilise ni espace ni ponctuation, et sépare les mots en mettant en capitale la première lettre de chaque mot.

Exemples: `getElementsByTagName`, `ListArray`, ...

Règles de nommage (2/2)



CamelCase vs camelCase

- Les noms qui définissent un type (classes, interfaces, enum, record, ...) commence par un majuscule.
- Les autres noms (méthodes, variables, attributs, ...) commencent par une minuscule.

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
⇒ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet $\Rightarrow$ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashCode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Méthodes de conversion $\Rightarrow$ utilisation du `to`

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les parties du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confié à d'autres méthodes

Ne pas mentir

```
class User {  
    private boolean authenticated;  
    private String password;  
  
    public boolean checkPassword(String password) {  
        if (password.equals(this.password)) {  
            authenticated = true;  
            return true;  
        }  
        return false;  
    }  
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après ses données.

Section 8

Des principes pour bien programmer

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
void cook(){  
    // On choisit la recette  
    Food wantToEat = thinkAboutFood();  
    Recipe recipe = lookOnMarmiton(wantToEat);  
    // On réunit les ingrédients  
    openFridge();  
    for(Ingredient ingredient :  
        recipe.getFreshIngredients()){  
        takeInFrige(ingredient);  
    }  
    closeFridge();  
    openCupboard();  
    ...  
    // On suit la recette
```

Bonne approche

```
void cook(){  
    Recipe recipe = chooseRecipe();  
    gatherIngredients(recipe);  
    followRecipe(recipe);  
}
```

```
Recipe chooseRecipe(){  
    Food wantToEat = thinkAboutFood();  
    Recipe recipe = lookOnMarmiton(wantToEat);  
    return recipe;  
}
```

...

Programme bien conçu

Un programme est “bien conçu” s’il permet de :

- Absorber les changements avec un minimum d’effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs

- Limiter les modules impactés
 - ▶ Simplifier les tests unitaires
 - ▶ Rester conforme à la partie des spécifications qui n’ont pas changé
 - ▶ Faciliter l’intégration
- Gagner du temps

Section 9

Tests

Différents types de de tests

Règle

Un code non testé n'a aucune valeur.

Corollaire

Tout code doit être testé

Différents type de tests

- **Test unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
- **Test d'intégration** : Tester des portions du programme combinant plusieurs classes.
- **Test de non régression** : Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
 - ▶ cas normaux
 - ▶ cas limites
 - ▶ cas anormaux

Tests unitaires en java : JUnit

- Un framework de test unitaire pour Java
- S'appuie sur des **assertions**

Utilisation de JUnit (1/2)

Règles

- 1 classe de test = un ensemble de méthodes de test
- 1 classe de test par classe à tester
- 1 méthode de test = 1 cas de test
- 1 cas de test = (description, données d'entrée, résultat attendu)

Structure d'une méthode de test de base

- méthode d'instance publique
- annotée avec `@Test` (à mettre avant la déclaration de la méthode)
- ne prend aucun paramètre
- ne renvoie rien
- lève une `AssertionError` en cas de test échoué

Utilisation de JUnit (2/2)

Conventions de nommage

- nom d'une classe de test : *NameTestedClassTest*
- nom d'une méthode de test : *testNameMethodTested*

Structure d'un projet avec tests

les tests sont séparés du code de production : répertoire `main` pour le code de production et répertoire `test` pour le code de tests.

⇒ nécessaire de séparer les test du code de production car :

- on ne donne pas l'accès au code de test au client par exemple
- les tests ont un rôle spécifique différent du code de production

Assertions JUnit (1/2)

- `assertTrue(boolean condition)` : vérifie que `condition` est vraie.
- `assertFalse(boolean condition)` : vérifie que `condition` est faux.
- `assertEquals(expected, actual)` : vérifie que `expected` est égal à `actual`
égal : `equals` pour les objets et `==` pour les types primitifs.
- `assertEquals(double expected, double actual, double delta)` :
vérifie que $|expected - actual| \leq delta$
- `assertNull(Object object)` : vérifie que la référence est `null`
- `assertNotNull(Object object)` : vérifie que la référence **n'est pas** `null`

Assertions JUnit (2/2)

- `assertSame(Object expected, Object actual)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertArrayEquals(Object[] expected, Object[] actual)` : vérifie si les deux tableaux contiennent les même éléments dans le même ordre.
- `fail()` : échoue toujours

Message

Pour toutes les méthodes `assert`, il est possible de mettre un message en dernier paramètre qui permet d'identifier l'assertion.

Exemple de classe à tester : RationalNumber

```
public class RationalNumber {  
    public final int numerator;  
    public final int denominator;  
  
    public RationalNumber(int numerator, int denominator) {  
        int gcd = gcd(numerator, denominator);  
        this.numerator = numerator / gcd;  
        this.denominator = denominator / gcd;  
    }  
  
    public RationalNumber add(RationalNumber val) {  
        int numerator    = (this.numerator * val.denominator) + (this.denominator * val.numerator);  
        int denominator = this.denominator * val.denominator;  
        return new RationalNumber(numerator, denominator);  
    }  
}
```

Exemple de classe de test

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class RationalNumberTest {

    @Test
    void testAdd(){
        RationalNumber one = new RationalNumber(1, 1);
        RationalNumber onePlusOne = one.add(one);
        assertEquals(2, onePlusOne.numerator,
            "Numerator of one plus one must be two.");
        assertEquals(1, onePlusOne.denominator,
            "Denominator of one plus one must be one.");
    }
}
```

Exemple de classe à tester (1/2) : Emails

```
public class Emails {  
    private String text;  
    public Emails(String text) { this.text = text; }  
    public List<String> userNames() {  
        int pos = 0;  
        List<String> users = new ArrayList<String>();  
        for(;;) {  
            int atIndex = text.indexOf('@', pos);  
            if (atIndex == -1) break;  
            String userName = userName(atIndex);  
            if (userName.length() > 0) users.add(userName);  
            pos = atIndex + 1;  
        }  
        return users;  
    }  
}
```

Exemple de classe à tester (2/2) : Emails

```
private String userName(int atIndex) {  
    int back = atIndex - 1;  
    while (back >= 0 &&  
           (Character.isLetterOrDigit(text.charAt(back))  
            || text.charAt(back) == '.')) {  
        back--;  
    }  
    return text.substring(back + 1, atIndex);  
}
```

Exemple de classe de test (1/2)

```
public class EmailsTest{
    @Test
    public void testUserNamesBasic() {
        Emails emails =
            new Emails("foo bart@cs.edu xyz marge@ms.com baz");
        assertEquals(Arrays.asList("bart", "marge"),
            emails.getUserNames());
    }
    @Test
    public void testUserNamesChars() {
        Emails emails =
            new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");
        assertEquals(Arrays.asList("f.ast", "a.2.c"),
            emails.getUserNames());
    }
}
```


Exemple de classe de test (2/2)

@Test

```
public void testUserNamesHard() {  
    Emails emails = new Emails("x y@cs 3@ @z@");  
    assertEquals(Arrays.asList("y", "3", "z"),  
        emails.getUserNames());  
    emails = new Emails("no emails here!");  
    assertEquals(Collections.emptyList(),  
        emails.getUserNames());  
    emails = new Emails("@@@");  
    assertEquals(Collections.emptyList(),  
        emails.getUserNames());  
    emails = new Emails("");  
    assertEquals(Collections.emptyList(),  
        emails.getUserNames());  
}
```

Test unitaires (à retenir)

- Il est essentiel de tester son code.
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important de tester tous les types de cas :
 - ▶ cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
 - ▶ cas limites (utilisation de la méthode sur une donnée "étrange")
 - ▶ cas anormaux (vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés)

La suite : le TDD (Test Driven Development)

- Écrire un test qui échoue avant de pouvoir écrire du code
- Écrire une assertion à la fois qui fait échouer un test
- Écrire le minimum de code pour que l'assertion soit satisfaite

Principe

- Le code d'un projet est stocké dans un serveur.
- Les développeurs soumettent des modifications avec des commentaires à chaque fois.
- Le serveur conserve l'historique des mises à jour

Pourquoi la gestion de version ?

- Pour travailler de manière harmonieuse en équipe sans se marcher dessus
- Pour revenir en arrière en cas de problèmes (chaque modification est générée une version du code sur laquelle il est possible de revenir)
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

- Logiciel de gestion de version le plus populaire
- Serveur gratuit : github
- Version libre de logiciel serveur : gitlab
- Gestion de version décentralisée : la gestion de version se fait aussi en local

Utilisation de git

- Via l'IDE : VCS (Version Control Systems) dans le menu d'IntelliJ
- En ligne de commande : commande git

Git (source: Randall Munroe (xkcd))



Fonctionnement basique de git

- À la première utilisation, on crée une copie locale du dépôt git (`clone` ou `init`)
- À chaque commit :
 - ▶ on récupère la version courante du dépôt sur le serveur (`pull`)
 - ▶ On ajoute les fichiers à modifier (`add`)
 - ▶ On finalise le commit en donnant un message résumant les modifications (`commit`)
- Après un ou plusieurs commits, on met à jour la version distante avec nos modifications (`push`)

Exemples de commandes git

```
git clone adresse_projet
```

⇒ Clone un projet en local depuis un serveur

```
git add nom_de_fichier
```

⇒ Ajoute un fichier à la prochaine mise à jour.

```
git commit -m"commentaire"
```

⇒ Fait une mise à jour en local

```
git push
```

⇒ Pousse les mises à jour locales sur le serveur

```
git pull
```

→ Récupère les mises à jour du serveur en local

- Importance d'avoir une gestion de version de son code
- Commandes de base : `clone`, `add`, `commit`, `push`, `pull`, ...

Git avancé (dans vos prochains cours)

- Notion de branche : on “fork” le dépôt pour créer une branche séparée sur laquelle on va travailler et on demande ensuite de l'intégrer dans la branche principale une fois les modifications effectuées.
- CI/CD (Intégration Continue/Déploiement (ou livraison) continue) : automatisation des tests et du déploiement (par exemple sur les serveurs qui utilise le code) ou de la livraison (publication du code compilé sur un serveur) à chaque push sur la branche principale.

Section 10

Interfaces

Problèmes de recyclage

- Papiers, bouteilles, piles électriques, cageots, ... sont des objets **différents** :
⇒ *déchirer* du papier, *remplir* un bouteille
- Mais ces objets partagent tous la propriété d'être **recyclable**
⇒ tous peuvent être *recyclés* (même si le processus peut varier)
On peut *recycler* tous les objets d'une poubelle.

En programmation objet

- Paper, Bottle, Battery, Crate, ... sont des classes d'objets **différentes**
- Toutes ont une méthode `recycle()` avec une implémentation **adaptée** à chacune.

Comment recycler tous les objets d'une poubelles

Avec le code suivant ?

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

Problème

Comment définir le tableau Trashcan ?

Quel est le type T de ces éléments ?

Remarques

- les objets de type T implémentent la méthode `recycle`
- Trashcan doit pouvoir contenir des objets de types différents

La solution objet

Conserver les classes différentes et créer un type commun.

- on doit conserver les classes différenciées : Paper, Bottle, ...
- on doit traiter les objets sans les différencier par leur classe.
- il faut pouvoir considérer les instances des classes comme des objets de type implémente la méthode `recycle`.

Projection

On va projeter les objets sur un type commun qui ne gardera que la partie commune des fonctionnalités. On ne considère qu'une facette de l'objet.

Solution java : interface

- En java, une interface est un ensemble de déclaration de signatures de méthodes et définit un type.
- Une classe peut **implémenter** une interface et doit dans ce cas définir le comportement (code) pour **chacune** des méthodes de l'interface.
- les instances d'une classe pourront être vues comme étant du **type de l'interface**, manipulées comme telles et avoir leur référence stockée dans une variable du type de l'interface.
- Un référence du type d'une interface accepte uniquement les appels de méthodes définies dans l'interface.

Exemple d'utilisation (1/2)

```
public interface Recyclable{
    public void recycle();
}

public class Paper implements Recyclable{
    // ...
    public void recycle(){
        System.out.println("Recycling paper");
    }
}

public class Bottle implements Recyclable{
    // ...
    public void recycle(){
        System.out.println("Recycling bottle");
    }
}
```

Exemple d'utilisation (2/2)

```
Recyclable[] trashcan = new Recyclable[2];
```

```
trashcan[0] = new Paper();
```

```
trashcan[1] = new Bottle();
```

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

Interface pour l'affichage

Supposons que des classes implémentent un service de façons différentes :

```
class SimplePrinter {  
    void print(String document){  
        System.out.println(document);  
    }  
}  
  
class BracePrinter {  
    void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Les instances de ces deux classes possèdent une méthode print avec la même signature (types des arguments et du retour)

Code facilement modifiable

Nous souhaiterions pouvoir facilement passer du code suivant :

```
Printer printer = new SimplePrinter();  
printer.print("truc"); // → truc
```

au code suivant :

```
Printer printer = new BracePrinter();  
printer.print("truc"); // → {truc}
```

Il nous faudrait définir un type `Printer` qui oblige la variable à contenir des références vers des objets qui implémentent la méthode `print`.

⇒ définition d'une interface `Printer`.

Abstraction

On peut vouloir traiter les objets en utilisant les services qu'ils partagent :

```
for (int index = 0; index < printers.length index++)  
    printers[index].print(document);
```

On peut aussi vouloir écrire un programme en supposant que les objets manipulés implémentent certains services (comme le fait de pouvoir les comparer) :

```
boolean isSorted(Comparable[] array) {  
    for (int i = 0; i < array.length - 1; i++)  
        if (array[i].compareTo(array[i+1]) > 0)  
            return false;  
    return true  
}
```

Description d'une interface

Description d'une interface en Java :

```
public interface Printer{  
    /**  
     * Affiche la chaîne de caractères document.  
     * @param document la chaîne à afficher  
     */  
    public void print(String document);  
}
```

Une interface :

- définit la signature (types des arguments et du retour) d'une ou plusieurs méthodes
- est un contrat
- définit un type : référence vers un objet implémentant les méthodes de l'interface

Implémentation d'une interface

Le mot-clé `implements` permet d'indiquer qu'une classe implémente un interface :

```
class SimplePrinter implements Printer {  
    void print(String document){  
        System.out.println(document);  
    }  
}  
  
class BracePrinter implements Printer {  
    void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Java vérifie à la compilation que toutes les méthodes de l'interface sont implémentées.

Références et interfaces

Déclaration d'une variable de type référence vers une instance d'une classe qui implémente l'interface `Printer` :

```
Printer printer;
```

Important

Une interface ne définit pas de constructeurs.

Interdit : ~~`printer = new Printer();`~~

Compatibilité classe et instance

Pour affecter une référence à une variable d'un type défini par une interface, on doit instancier une classe implémentant l'interface.

Références et interfaces

Il est donc possible d'instancier une classe implémentant l'interface,

```
SimplePrinter simplePrinter = new SimplePrinter();
```

puis de stocker la référence de l'objet dans une variable de type de l'interface :

```
Printer printer1 = simplePrinter;
```

On parle alors d'**upcasting** (transtypage vers le haut). On peut aussi directement mettre un tel objet sans passer par une variable intermédiaire :

```
Printer printer2 = new BracePrinter();
```

Par contre, cela ne fonctionne pas dans le cas où la classe n'implémente pas l'interface :

```
Printer printer3 = new String("Hello!"); //interdit
```

Références et interfaces

```
class Utils {  
    static void printString(Printer[] printers,  
                            String doc) {  
        for (int i = 0; i < printers.length; i++)  
            printers[i].print(doc);  
    }  
  
    static void printArray(String[] array,  
                           Printer printer) {  
        for (int i = 0; i < array.length; i++)  
            printer.print(array[i]);  
    }  
}
```

Polymorphisme

L'existence des méthodes est vérifiée à la compilation.

Le code suivant ne compilera pas car l'interface `Printer` n'a pas de méthode `println` :

```
Printer printer = new SimplePrinter();  
printer.println("Hello!"); // Impossible
```

Définition polymorphisme

Du grec ancien *polús* (plusieurs) et *morphê* (forme), concept consistant à fournir une interface unique à des entités pouvant avoir différents types.

Polymorphisme

Le choix de la méthode à exécuter ne peut être fait qu'à l'exécution :

```
Printer[] printers = new Printer[2];  
printers[0] = new SimplePrinter();  
printers[1] = new BracePrinter();  
Random random = new Random(); // générateur aléatoire  
int index = random.nextInt(2); // 0 et 1  
printers[index].print("mon message");
```

L'affichage dépend du tirage aléatoire pour index :

- index=0 → méthode de la classe SimplePrinter → mon message
- index=1 → méthode de la classe BracePrinter → {mon message}

Implémentations multiples

Il peut être utile d'avoir une classe implémentant plusieurs interfaces.

Par exemple, une classe Modem pourrait implémenter les deux interfaces suivantes :

```
public interface ConnexionManager
{
    public void dial(string phoneNumber);
    public void hangUp();
}

public interface TransmissionManager
{
    public void send(char c);
    public char receive();
}
```

Implémentations multiples

Une classe Printable avec une méthode print qui permet d'afficher l'objet.

```
interface Printable {  
    public void print();  
}
```

Une classe Stack avec deux méthodes :

- push qui permet d'empiler un entier.
- pop dépile et retourne l'entier en haut de la pile.

```
interface Stack {  
    public void push(int value);  
    public int pop();  
}
```

Implémentations multiples

Implémentation des deux interfaces précédentes :

```
public class PrintableArrayStack
    implements Stack, Printable {
    private int[] array; private int size;
    public PrintableArrayStack(int capacity) {
        array = new int[capacity]; size = 0;
    }
    public void push(int v) { array[size] = v; size++; }
    public int pop() { size--; return array[size]; }
    public void print() {
        for (int i = 0; i < size; i++)
            System.out.print(array[i]+" ");
        System.out.println();
    }
}
```

Implémentations multiples

Implémentation d'une des deux interfaces :

```
public class PrintableString implements Printable {  
    private String string;  
    public PrintableString(String string) {  
        this.string = string;  
    }  
    public void print() {  
        System.out.println(string);  
    }  
}
```

Implémentations multiples

Exemple d'utilisation des classes précédentes :

```
Printable[] printables = new Printable[3];
printables[0] = new PrintableString("bonjour");
PrintableArrayStack stack = new PrintableArrayStack(10);
printables[1] = stack;
printables[2] = new PrintableString("salut");
stack.push(10);
stack.push(30); System.out.println(stack.pop());
stack.push(12);
for (int i = 0; i < printables.length; i++)
    printables[i].print();
```

Qu'écrit ce programme sur la sortie standard ?

Vérification des types

Vérification des types à la compilation :

```
Stack[] arrayStack = new Stack[2];  
arrayStack[0] = new PrintableStack();  
arrayStack[1] = new PrintableString("t"); // Erreur !
```

PrintableString n'implémente pas Stack.

```
Stack stack = new PrintableStack();  
Printable printable = stack; // Erreur !
```

Le type Stack n'est pas compatible avec le type Printable.

Résumé des interfaces

- Une interface est un ensemble de signatures de méthodes.
- Une classe peut implémenter une ou plusieurs interfaces : elle doit préciser le comportement de chacune des méthodes de chaque interface qu'elle implémente.
- Il est possible de déclarer une variable pouvant contenir des références vers des instances de classes qui implémentent l'interface.
- Java vérifie à la compilation que toutes les affectations et les appels de méthodes sont corrects.
- Le choix du code qui va être exécuté est décidé à l'exécution (en fonction de l'instance pointée par la référence).

Section 11

Interfaces génériques

Exemple d'interfaces génériques en Java

- Comparable<T> : objets qu'on peut comparer à des objets de type T.
- List<T> : liste d'objets de type T.
- Stack<E> : pile d'objet de type T.
- Iterable<T> : collection d'objet de type T qu'on peut parcourir avec un boucle.

Rappel : types paramétrés/génériques

Types dont la définition contient un autre type.

Définition de Comparable<T>

```
public interface Comparable<T>{  
    /**  
     * Compares this object with the specified object for  
     * order. Returns a negative integer, zero, or a  
     * positive integer as this object is less than,  
     * equal to, or greater than the specified object.  
     * @param other the objet to be compared  
     * @return a negative integer, zero, or a positive  
     * integer as this object is less than, equal to, or  
     * greater than the specified object.  
     */  
    int compareTo(T other);  
}
```

Utilisation de Comparable<T>

// Utilisation typique

```
public class MyOrderedClass
    implements Comparable<MyOrderedClass>{

    int compareTo(MyOrderedClass other){
        // ...
    }
}
```

Sert à définir une relation d'ordre entre les objets d'une classe (par exemple pour les trier).

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return lastname.compareTo(other.lastName);
    }
}

List<Student> students; //...
Collections.sort(students);
// Tri par nom de famille
```

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return idNumber - other.idNumber;
    }
}
```

```
Student[] students; //...
Arrays.sort(students);
// Tri par numéro d'étudiant
```

Interface Iterable

```
public interface Iterable<T>{  
    Iterator<T> iterator();  
    void forEach(Consumer <? super T> action);  
    Spliterator<T> spliterator();  
}
```

- Définition complexe
- Utilisation facile et utile

Utilisation d'Iterable

Utilisation : si une classe implémente `Iterable<T>`, ces instances contiennent une collection d'objets de type `T`.

On peut parcourir les objets de la collection à l'aide d'une boucle `for`.

```
public class Order{
    Iterable<Items> items;
    //...
    public void printAllItems(){
        for (Item item : this.items){
            System.out.println(item.toString());
        }
    }
}
```


Interface Collection

```
public interface Collection<T> extends Iterable<T>{  
    boolean add(T element);  
    boolean contains(T element);  
    boolean isEmpty();  
    boolean remove(Object o);  
    //...  
}
```

Mot-clé extends

Quand une interface “fille” étend une interface “mère”, elle hérite de toutes les méthodes de sa “mère”.

Une classe implémentant la classe “fille” doit donc définir les méthodes des deux interfaces.

Interfaces étendant Collection

Il existent de nombreuses interfaces qui sont une extension de l'interface `Collection` :

- `Set` : collection d'objet dans laquelle chaque objet ne peut apparaitre qu'une fois. Implémenté par `HashSet` et `TreeSet`.
- `List` : séquence d'éléments. Implémenté par `ArrayList` et `LinkedList`.
- `Deque` (**d**ouble **e**nded **q**ueue) : séquence d'éléments avec accès qu'au début et à la fin (file d'attente). Implémenté par `ArrayDeque` et `LinkedList`.
- ...

Section 12

Composition et délégation

Exemple de classe

On considère la classe Point suivante :

```
public class Point {  
    public final double x, y;  
    public Point(double x, double y){  
        this.x = x;  
        this.y = y;  
    }  
    public double distanceTo(Point p){  
        double dx = this.x - p.x;  
        double dy = this.y - p.y;  
        return Math.hypot(dx, dy);  
    }  
}
```

Composition

Afin d'implémenter ses services, une instance peut créer des instances et conserver leurs références dans ses attributs.

```
public class Circle {  
    private Point center;  
    private double radius;  
  
    public Circle(double x, double y, double radius){  
        this.center = new Point(x, y);  
        this.radius = radius;  
    }  
  
    public double getX(){ return center.x; }  
    public double getY(){ return center.y; }  
    public double getRadius(){ return radius; }
```

Agrégation

Une instance peut simplement posséder des références vers des instances :

```
public class Circle {  
    private Point center, point;  
    public Circle(Point center, Point point){  
        this.center = center;  
        this.point = point;  
    }  
    public Point getCenter(){ return center; }  
    public double getRadius(){  
        double dx = center.x - point.x;  
        double dy = center.y - point.y;  
        return Math.hypot(dx, dy);  
    }  
}
```

Agrégation et délégation

Délégation du calcul de la distance à l'instance center de la classe Point :

```
public class Circle {  
    private Point center, point;  
  
    public Circle(Point center, Point point){  
        this.center = center;  
        this.point = point;  
    }  
    public Point getCenter(){ return center; }  
    public double getRadius(){  
        return center.distanceTo(point);  
    }  
}
```

Agrégation récursive

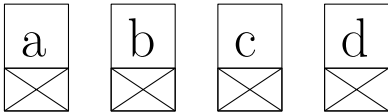
Il est possible de créer des structures récursives (les attributs de la classe contiennent des références vers une instance de la classe).

```
public class Node {  
    private Node[] nodes;  
    private String name;  
  
    public Node(String name, Node[] nodes){  
        this.name = name;  
        this.nodes = nodes;  
    }  
  
    public Node(String name){  
        this(name, new Node[0]);  
    }  
}
```


Utilisation d'une structure récursive

```
Node a = new Node("a");  
Node b = new Node("b");  
Node c = new Node("c");  
Node d = new Node("d");
```

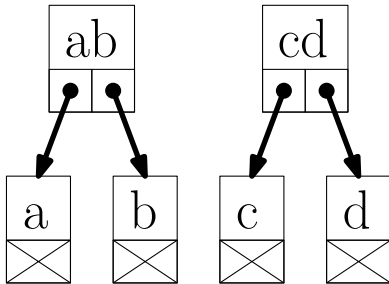
Utilisation d'une structure récursive



Utilisation d'une structure récursive

```
Node a = new Node("a");  
Node b = new Node("b");  
Node c = new Node("c");  
Node d = new Node("d");  
Node[] arrayAB = new Node[]{a,b};  
Node ab = new Node("ab", arrayAB);  
Node[] arrayCD = new Node[]{c,d};  
Node cd = new Node("cd", arrayCD);
```

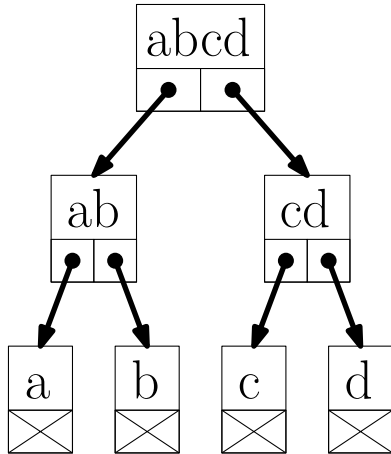
Utilisation d'une structure récursive



Utilisation d'une structure récursive

```
Node a = new Node("a");
Node b = new Node("b");
Node c = new Node("c");
Node d = new Node("d");
Node[] arrayAB = new Node[]{a,b};
Node ab = new Node("ab", arrayAB);
Node[] arrayCD = new Node[]{c,d};
Node cd = new Node(arrayCD, "cd");
Node cd = new Node("cd", arrayCD);
Node abcd = Node cd = new Node("abcd", arrayABCD);
```

Utilisation d'une structure récursive

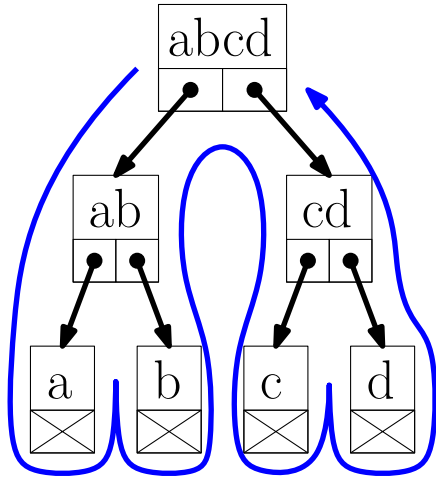


Agrégation récursive et méthodes

Il est ensuite possible d'implémenter des méthodes de façon récursive :

```
public class Node {  
    private Node[] nodes;  
    private String name;  
  
    // Constructeurs des transparents précédents.  
  
    public void print(){  
        System.out.print "[" + name + " ]");  
        for(int i = 0; i < nodes.length; i++){  
            nodes[i].print();  
        }  
    }  
}
```

Parcours d'une structure récursive



Section 13

Classes abstraites et extension

Classe ListSum

Supposons que nous ayons la classe suivante :

```
public class ListSum {  
    private int[] list = new int[10];  
    private int size = 0;  
    public void add(int value) {  
        list[size] = value;  
        size++;  
    }  
    public int eval() {  
        int result = 0;  
        for (int i = 0; i < size; i++)  
            result += list[i];  
        return result;  
    }  
}
```

Classe ListProduct

Supposons que nous ayons aussi la classe suivante (très similaire) :

```
public class ListProduct {  
    private int[] list = new int[10];  
    private int size = 0;  
    public void add(int value) {  
        list[size] = value;  
        size++;  
    }  
    public int eval() {  
        int result = 1;  
        for (int i = 0; i < size; i++)  
            result *= list[i];  
        return result;  
    }  
}
```

Refactoring pour isoler la répétition

Les deux classes sont très similaires et il y a de la répétition de code.

Il est possible de *refactorer* (réécrire le code) les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {  
    /* attributs, constructeur et méthode add. */  
    public int eval() {  
        int result = neutral();  
        for (int i = 0; i < size; i++)  
            result = compute(result, list[i]);  
        return result;  
    }  
    private int neutral() { return 0; }  
    private int compute(int a, int b) { return a+b; }  
}
```

Refactoring pour isoler la répétition

Il est possible de *refactorer* les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {  
    /* attributs, constructeur et méthode add. */  
    public int eval() {  
        int result = neutral();  
        for (int i = 0; i < size; i++)  
            result = compute(result, list[i]);  
        return result;  
    }  
    private int neutral() { return 1; }  
    private int compute(int a, int b) { return a*b; }  
}
```

Comment éviter la répétition ?

Après la *refactorisation* du code :

- seules les méthodes `neutral` et `compute` diffèrent
- il serait intéressant de pouvoir supprimer les duplications de code

Deux solutions :

- La délégation en utilisant une interface
- L'extension et les classes abstraites

Interface Operator

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {  
    public int neutral();  
    public int compute(int a, int b);  
}  
  
public class Sum implements Operator {  
    public int neutral() { return 0; }  
    public int compute(int a, int b) { return a+b; }  
}  
  
public class Product implements Operator {  
    public int neutral() { return 1; }  
    public int compute(int a, int b) { return a*b; }
```

Délégation

Les classes ListSum et ListProduct sont fusionnées dans une unique classe List qui délègue les calculs à un objet qui implémente l'interface Operator :

```
public class List {  
    /* attributs et méthode add */  
    private Operator operator;  
    public List(Operator operator){  
        this.operator = operator;  
    }  
    public int eval() {  
        int result = operator.neutral();  
        for (int i = 0; i < size; i++)  
            result = operator.compute(result, list[i]);  
        return result;  
    }  
}
```


Délégation

Utilisation des classes ListSum et ListProduct :

```
ListSum listSum = new ListSum();  
listSum.add(2); listSum.add(3);  
System.out.println(listSum.eval());  
ListProduct listProduct = new ListProduct();  
listProduct.add(2); listProduct.add(3);  
System.out.println(listProduct.eval());
```

Utilisation après la *refactorisation* du code :

```
List listSum = new List(new Sum());  
listSum.add(2); listSum.add(3);  
System.out.println(listSum.eval());  
List listProduct = new List(new Product());  
listProduct.add(2); listProduct.add(3);
```

Classes abstraites

```
public abstract class List {  
    private int[] list = new int[10];  
    private int size = 0;  
    public void add(int value) {  
        list[size] = value; size++;  
    }  
    public int eval() {  
        int result = neutral();  
        // utilisation d'une méthode abstraite  
        for (int i = 0; i < size; i++)  
            result = compute(result, list[i]); // idem  
        return result;  
    }  
    public abstract int neutral(); // méthode abstraite
```

Mot-clé abstract

Classe abstraite

- On peut mettre `abstract` devant le nom de la classe à sa définition pour signifier qu'une classe est abstraite.
- Une classe est abstraite si des méthodes ne sont pas implémentées.
⇒ Classe abstraite = classe avec des méthodes abstraites
- Tout comme pour un interface, une classe abstraite n'est pas instanciable.

Méthode abstraite

- `abstract` devant le nom de le type de retour de la méthode à sa définition pour signifier qu'une méthode est abstraite.
- Méthode abstraite = méthode sans code, juste la signature (type du retour et des paramètres) est définie

Classes abstraites et extension

Tout comme pour les interfaces, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur  
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe List afin de récupérer les attributs et les méthodes de List et définir le code des méthodes abstraites :

```
public class ListSum extends List {  
    public int neutral() { return 0; }  
    public int compute(int a, int b) { return a+b; }  
}
```

Classes abstraites et extension

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

- les méthodes `add` et `eval` sont définies dans `List` et `ListSum` hérite du code de ses méthodes.
- les méthodes `neutral` et `compute` qui étaient abstraites dans `List` sont définies dans `ListSum`.

On peut donc instancier la classe `ListSum` :

```
ListSum listSum = new ListSum();  
listSum.add(3);  
listSum.add(7);  
System.out.println(listSum.eval());
```

Classes abstraites et extension

On peut procéder de manière similaire pour créer une classe ListProduct

```
public class ListProduct extends List {  
    public int neutral() { return 1; }  
    public int compute(int a, int b) { return a*b; }  
}
```

La classe ListProduct n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();  
listProduct.add(3);  
listProduct.add(7);  
System.out.println(listProduct.eval());
```

Généralisation de l'extension aux classes non-abstraites

Plus généralement, l'extension permet de créer une classe en :

- conservant les services (attributs et méthodes) d'une autre classe;
- ajoutant de nouveaux services (attributs et méthodes);
- redéfinissant certains services (méthodes).

En Java :

- On utilise le mot-clé `extends` pour étendre une classe ;
- Une classe ne peut étendre directement qu'une seule classe.

Important

Il est toujours préférable de privilégier l'implémentation à l'extension.

Extension pour ajouter des nouveaux services

Supposons que nous ayons la classe Point suivante :

```
public class Point {  
    public int x, y;  
    public void setPosition(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Il est possible d'ajouter de nouveaux services en utilisant l'extension :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public void setColor(int r, int g, int b) {  
        this.r = r; this.g = g; this.b = b;  
    }  
}
```


Extension pour ajouter des nouveaux services

Les services (méthodes et attributs) de Point sont disponibles dans Pixel :

```
Pixel pixel = new Pixel();  
pixel.setPosition(4,8);  
System.out.println(pixel.x); // → 4  
System.out.println(pixel.y); // → 8  
pixel.setColor(200,200,120);
```

Évidemment, les services de Pixel ne sont pas disponibles dans Point :

```
Point point = new Point();  
point.setPosition(4,8);  
System.out.println(point.x); // → 4  
System.out.println(point.y); // → 8  
point.setColor(200,200,120); // impossible !
```

Redéfinition de méthode

Supposons que nous ayons la classe Point suivante :

```
public class Point {  
    public int x, y;  
    public void setPosition(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void clear() {  
        x = 0; y = 0;  
    }  
}
```

Redéfinition de méthode

Il est possible de redéfinir la méthode `clear` dans `Point` :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public void setColor(int r, int g, int b) {  
        this.r = r; this.g = g; this.b = b;  
    }  
    public void clear(){  
        x = 0; y = 0;  
        r = 0; g = 0; b = 0;  
    }  
}
```

Redéfinition avec super

```
public class Point {  
    public int x, y;  
    public void setPosition(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void clear() {  
        setPosition(0, 0);  
    }  
}
```

Le mot-clé `super` permet d'utiliser la méthode `clear` de `Point` :

```
public class Pixel extends Point { public int r, g, b;  
    public void setColor(int r, int g, int b) {  
        this.r = r; this.g = g; this.b = b; }  
}
```

Le mot-clé super

```
public class Point {  
    public int x, y;  
    public void setPosition(int x, int y) {  
        this.x = x; this.y = y; }  
}
```

Si la méthode n'a pas été redéfinie, le mot clé super est inutile :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public void setColor(int r, int g, int b) {  
        this.r = r; this.g = g; this.b = b;  
    }  
    public void clear() {  
        /*super.* /setPosition(0, 0);  
    }  
}
```

Les constructeurs et le mot-clé super

```
public class Point {  
    public int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

La classe Point n'a pas de constructeur sans paramètre, il faut donc indiquer comment initialiser la partie de Pixel issue de la classe Point :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public Pixel(int x, int y, int r, int g, int b) {  
        super(x, y); // appel du constructeur de Point  
        this.r = r; this.g = g; this.b = b;  
    }  
}
```

Les constructeurs

Supposons que nous ayons la classe Point suivante :

```
public class Point {  
    public int x, y;  
    public Point() { x = 0; y = 0; }  
    public Point(int x, int y) { this.x = x; this.y = y; }  
}
```

Par défaut, le constructeur sans paramètre est appelé :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public Pixel(int r, int g, int b) {  
        // appel du constructeur sans paramètre de Point  
        this.r = r; this.g = g; this.b = b;  
    }  
}
```

Les constructeurs

```
public class Point {  
    public int x, y;  
    //public Point() { x = 0; y = 0; }  
    public Point(int x, int y) { this.x = x; this.y = y; }  
}
```

Ici, vous devez préciser les paramètres du constructeur avec super :

```
public class Pixel extends Point {  
    public int r, g, b;  
    public Pixel(int r, int g, int b) {  
        // erreur de compilation  
        // (aucun constructeur sans paramètre)  
        this.r = r; this.g = g; this.b = b;  
    }  
}
```


Transtypes et polymorphisme

Aucune méthode ou attribut ne peut être supprimée lors d'une extension. Par exemple, Pixel possède toutes les attributs et méthodes de Point (même si certaines méthodes ont pu être redéfinies).

Par conséquent, l'**upcasting** est toujours autorisé :

```
Point point = new Pixel();  
point.setPosition(2,4);  
System.out.println(point.x + " " + point.y);  
point.clear();
```

Remarques

- Le code exécuté lors d'un appel de méthode est déterminé à l'exécution en fonction de la référence présente dans la variable.
- Le typage des variables permet de vérifier à la compilation l'existence des

Définition d'upcasting

Considérer une instance d'une classe comme une instance d'une de ses **super-classes**, c'est-à-dire :

- une classe étendue par la classe
- une interface implémentée par la classe

Remarques

- Après l'*upcasting*, les services supplémentaires de la classe (méthodes et attributs non-disponibles dans la super-classe) ne sont plus accessibles.
- Lors de l'appel des méthodes, si la classe a redéfini (*overrides*) la méthode, c'est le code de la classe qui est exécuté.

La classe Object

Par défaut, les classes étendent la classe Object de Java. Par conséquent, l'*upcasting* vers la classe Object est toujours possible :

```
Pixel pixel = new Pixel();  
Object object = pixel;  
Object[] array = new Object[10];  
for (int i = 0; i < t; i++) {  
    if (i%2==0) array[i] = new Point();  
    else array[i] = new Pixel();  
}
```

Notez que `object.setPosition(2,3)` n'est pas autorisé dans le code ci-dessus car la classe Object ne possède pas la méthode `setPosition` et seul le type de la variable compte pour déterminer si l'appel d'une méthode ou l'utilisation d'une attribut est autorisé.

Méthodes de la classe Object

- `protected Object clone()`: Creates and returns a copy of this object.
- `boolean equals(Object obj)`: Indicates whether some other object is “equal to” this one.
- `Class<?> getClass()`: Returns the runtime class of this Object.
- `int hashCode()`: Returns a hash code value for the object.
- `String toString()`: Returns a string representation of the object.

Il y a aussi des méthodes `wait` et `notify` pour attendre sur un objet et réveiller des *thread* en attentes sur un objet : cours de *Programmation objet concurrente* en Master 1 informatique d'AMU.

La méthode toString() de la classe Object

Par transitivité de l'extension, toutes les méthodes et attributs de la classe Object sont disponibles sur toutes les instances :

```
Object object = new Object(); Point point = new Point(2,3);
System.out.println(object.toString());
// → java.lang.Object@19189e1
System.out.println(point.toString());
// → test.Point@7c6768
```

La méthode toString est utilisée par Java pour convertir une référence en chaîne de caractères :

```
Object object = new Object();
Point point = new Point(2,3);
String string = object+";"+point;
System.out.println(string);
```

La méthode toString() et le polymorphisme (1/2)

Évidemment, il est possible de redéfinir la méthode toString :

```
public class Point {  
    public int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
  
    public String toString() {  
        return "("+x+", "+y+")";  
    }  
}
```

La méthode toString() et le polymorphisme (2/2)

Le polymorphisme fait que cette méthode est appelée si la variable contient une référence vers un Point :

```
Point point = new Point(2,3);  
Object object = point;  
System.out.println(point); // → (2,3)  
System.out.println(object); // → (2,3)
```

Rappel : extension d'interface

Supposons que nous ayons l'interface suivante :

```
public interface List<E> {  
    public int size();  
    public E get(int index);  
}
```

En Java, on peut aussi étendre une interface :

```
public interface ModifiableList<E> extends List<E> {  
    public void add(E value);  
    public void remove(int index);  
}
```

Une classe qui implémente l'interface `ModifiableList` doit implémenter les méthodes `size`, `get`, `add` et `remove`.

Extension d'interface

Supposons que la classe `ArrayModifiableList` implémente l'interface `ModifiableList`. Dans ce cas, nous pouvons écrire :

```
ModifiableList<Integer> modifiableList = new ArrayModifiableList<>();  
modifiableList.add(2);  
modifiableList.add(5);  
modifiableList.remove(0);  
List list = modifiableList;  
System.out.println(list.size());
```

En revanche, il n'est pas possible d'écrire :

```
list.remove(0);  
/* → Cette méthode n'existe pas */  
/* dans l'interface List. */
```

Extension de plusieurs interfaces

Supposons que nous avons l'interface suivante :

```
public interface Printable { public void print(); }
```

En Java, une interface peut étendre plusieurs interfaces :

```
public interface ModifiablePrintableList  
    extends ModifiableList, Printable {  
}
```

Remarques

- Nous ne définissons pas de nouvelles méthodes dans l'interface `ModifiablePrintableList`.
⇒ Cette interface ne représente que l'union des interfaces `ModifiableList` et `Printable`.
- De nouvelles méthodes auraient pu être définies dans

Section 14

Extension et accessibilité

Accessibilité : modificateur public et default

Une classe ou un membre (attribut ou méthode) est accessible :

- de n'importe où s'il est précédé du modificateur public ;
- des classes de son paquet si rien n'est précisé (default).

```
public class MyClass {  
    public int myPublicField;  
    int myPackageField;  
    public void doPublicAction() { }  
    void doPackageAction() { }  
}
```

Remarques :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

Accessibilité : modificateur private

Un membre (attribut ou méthode) est accessible :

- de n'importe où s'il est précédé du modificateur `public`
- des classes de son paquet si rien n'est précisé (default)
- des méthodes de la classe s'il est précédé de `private`

```
public class MyClass {  
    private int privateField;  
    private void doPrivateAction() { }  
}
```

Afin de limiter les conséquences d'une modification :

- Les méthodes ou attributs définies pour rendre lisible l'implémentation des fonctionnalités doivent être privées ;
- Seule l'interface de la classe doit être publique.

Accessibilité : modificateur protected

Un membre (attribut ou méthode) `protected` n'est accessible que par les méthodes des classes du paquet et par les classes qui l'étendent.

Le modificateur `protected` permet de protéger un membre :

```
public class MyClass {  
    protected int protectedField;  
    protected void doProtectedAction() { }  
}
```

Modificateur	Classe	Paquet	Extension	Extérieur
private	✓			
default	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

Section 15

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

Dans une classe, plusieurs méthodes peuvent avoir le même nom.

C'est ce qu'on appelle la surcharge de méthode.

Il est par contre nécessaire que la séquence dans l'ordre des types des arguments soit différente pour chaque méthode ayant le même nom.

La méthode est choisie par le compilateur de la façon suivante :

- Le nombre de paramètres doit correspondre
- Les affectations des paramètres doivent être valides
- Parmi ces méthodes, le compilateur choisit la plus spécialisée, c'est-à-dire celle entraînant le moins de changement de types (passage à une super-classe ou promotion pour les types primitifs)

Exemple de surcharge de méthode

```
public class DataArtist {  
  
    public void draw(String s) {  
        /* ... */  
    }  
    public void draw(int i) {  
        /* ... */  
    }  
    public void draw(double f) {  
        /* ... */  
    }  
    public void draw(int i, double f) {  
        /* ... */  
    }  
}
```

Exemple de surcharge de méthode

```
class Adder {  
    public static int add(int intVal1, int intVal2) {  
        System.out.println("integer");  
        return intVal1+intVal2;  
    }  
  
    public static double add(double doubleVal1,  
                             double doubleVal2) {  
        System.out.println("double");  
        return doubleVal1+doubleVal2;  
    }  
}
```

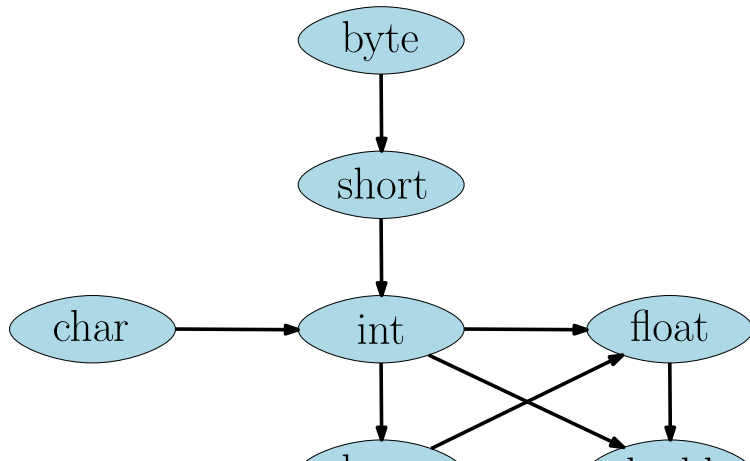
Exemple de surcharge de méthode

```
int intValue = 1;
double doubleValue = 2.2;
double result1 = Adder.add(doubleValue, doubleValue);
// → double

int result3 = Adder.add(intValue, intValue);
// → int
```

Promotion pour les types primitifs

Si les types des arguments ne correspondent pas aux types des paramètres, les types des arguments sont promus en suivant les flèches :



Exemple de surcharge de méthode

Pour les instances de classe, c'est le type du conteneur à la compilation qui compte.

```
class Printer {  
    static void print(Object object) {  
        System.out.println("Object : "+object);  
    }  
    static void print(String string) {  
        System.out.println("String : "+string);  
    }  
}  
  
String string = "message";  
Object object = string;  
Printer.print(string); // → String : message  
Printer.print(object); // → Object : message
```

Plusieurs constructeurs

C'est exactement les mêmes règles qui s'appliquent pour les constructeurs d'une classe.

```
public class Point{  
    public int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point() {  
        this(0, 0);  
    }  
  
    public Point(Point p) {  
        this(p.x, p.y);  
    }  
}
```

Section 16

Exceptions

Les exceptions

Un programme peut être confronté à une condition exceptionnelle (ou exception) durant son exécution.

Une exception est une situation qui empêche l'exécution normale du programme (elle ne doit pas toujours être considérée comme un bug). Quelques exemples de situations exceptionnelles :

- un fichier nécessaire à l'exécution du programme n'existe pas,
- une division par zéro,
- un débordement dans un tableau,
- un besoin de se connecter à un serveur et celui-ci est injoignable,
- un dépilement d'une pile vide,
- ...

Mécanisme de gestion des exceptions

Java propose un mécanisme de gestion des exceptions afin de distinguer l'exécution normale du traitement de celles-ci afin d'en faciliter leur gestion.

En Java, une exception est concrétisée par une instance d'une classe qui étend la classe `Exception`.

Pour lever (déclencher) une exception, on utilise le mot-clé `throw` :

```
if (problem) throw new MyException("error");
```

Pour capturer une exception, on utilise les mots-clés `try` et `catch` :

```
try { /* Problème possible */ }  
catch (MyException e) { /* traiter l'exception. */ }
```

Exemple d'exceptions existantes en java

- `ArithmeticException` : opération arithmétique impossible comme la division par 0.
- `IndexOutOfBoundsException` : dépassement d'indice dans un tableau, un vecteur, ...
- `NullPointerException` : accès à un attribut/méthodes/case pour les tableaux d'une référence valant `null`, argument `null` alors que ce n'est pas autorisé.
- `FileNotFoundException` : échec de l'ouverture d'un fichier à partir d'un chemin.
- `IllegalArgumentException` : argument incorrect (en dehors des valeurs autorisées) lors de l'appel d'une méthode.
- `NoSuchElementException` : next alors que l'itération est finie, dépilement d'une pile vide, ...
- ...

Définir son exception

Il suffit d'étendre la classe `Exception` (ou une classe qui l'étend) :

```
public class MyException extends Exception {  
    private int number;  
  
    public MyException(int number) {  
        this.number = number;  
    }  
  
    public String getMessage() {  
        return "Error " + number;  
    }  
}
```

La syntaxe try/catch

Pour capturer une exception, on utilise la syntaxe try/catch :

```
public static void test(int value) {  
    System.out.print("A ");  
    try {  
        System.out.println("B ");  
        if (value > 12) throw new MyException(value);  
        System.out.print("C ");  
    } catch (MyException e) { System.out.println(e); }  
    System.out.println("D");  
}
```

test(11)	test(13)
----------	----------

A B	A B
-----	-----

C D	MyException: Error 13
-----	-----------------------

Exceptions et signatures des méthodes

Une méthode doit préciser dans sa signature toutes les exceptions qu'elle peut lever et qu'elle n'a pas traitées avec un bloc try/catch :

```
public static void testValue(int value)
    throws MyException {
    if (value>12) throw new MyException(value);
}
public static void runTestValue(int value)
    throws MyException {
    testValue(value);
}
```

La méthode testValue peut lever une exception de type MyException.

runTestValue doit indiquer qu'elle peut lever une exception car elle ne gère pas l'exception provoquée par l'appel testValue(value).

Gestions des exceptions : règle

La méthode `runTestValue` peut lever une exception (de type `MyException`).

Lorsqu'on fait un appel à la méthode `runTestValue`, il est vérifié à la compilation que l'une des deux propriétés suivante est vraie :

- la méthode appelant `runTestValue` est indiquée comme pouvant lever l'exception `MyException` (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un bloc `try/catch`.

Si aucune des deux propriétés est vérifiée alors il y a une erreur à la compilation.

```
Error:(YY, XX) java: unreported exception MyException;  
      must be caught or declared to be thrown
```

Exceptions et signatures des méthodes

Une méthode doit donc préciser dans sa signature toutes les exceptions qu'elle peut lever et qu'elle n'a pas traitées avec un bloc try/catch

On doit donc écrire :

```
public class Main {  
    public static void main(String[] args) {  
        try { Test.runTestValue(13); }  
        catch (MyException e) { e.printStackTrace(); }  
    }  
}
```

Techniquement la méthode main pourrait indiquer qu'elle génère l'exception MyException mais cela n'aurait pas beaucoup de sens car cela voudrait dire qu'on ne gère pas vraiment l'exception.

Méthode printStackTrace

La méthode printStackTrace permet d'afficher la pile d'appels :

```
public class Main {  
    public static void main(String[] args) {  
        try { Test.runTestValue(13); }  
        catch (MyException e) { e.printStackTrace(); }  
    }  
}
```

```
MyException: Error 13  
    at Test.testValue(Test.java:23)  
    at Test.runTestValue(Test.java:20)  
    at Main.main(Main.java:6)
```


La classe RuntimeException

Une méthode doit indiquer toutes les exceptions qu'elle peut lever sauf si l'exception étend la classe RuntimeException. Bien évidemment, la classe RuntimeException étend Exception.

Quelques classes Java qui étendent RuntimeException :

- ArithmeticException
- ClassCastException
- IllegalArgumentException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException
- NoSuchElementException

Notez que ces exceptions s'apparentent le plus souvent à des bugs.

Description de la classe `RuntimeException` dans la documentation Java

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

`RuntimeException` and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Règle générale : ajout cas RuntimeException

Lorsqu'on fait un appel à une méthode `canThrow` pouvant lever une exception `MyException` qui n'étend pas `RuntimeException`, il est vérifié à la compilation que l'une des deux propriétés suivantes est vraie :

- la méthode appelant `canThrow` dans son code est indiquée comme pouvant lever une exception de type `MyException` ou une de ces super-classes (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un bloc `try/catch`.

Capturer une exception en fonction de son type

```
public static int divide(Integer a, Integer b) {  
    try { return a/b; }  
    catch (ArithmeticException exception) {  
        exception.printStackTrace();  
        return Integer.MAX_VALUE;  
    } catch (NullPointerException exception) {  
        exception.printStackTrace();  
        return 0;  
    }  
}
```

divide(12,0) :

```
java.lang.ArithmeticException: / by zero  
    at Test.diviser(Test.java:17)  
    at Test.main(Test.java:22)
```

Le mot-clé finally

On rajouter un bloc finally après des blocs try. Le bloc associé au mot-clé finally est toujours exécuté :

```
public static void readFile(String fileName) {  
    try {  
        FileReader fileReader = new FileReader(fileName);  
        /* peut déclencher une FileNotFoundException. */  
        try {  
            int character = fileReader.read(); // IOException ?  
            while (character != -1) {  
                System.out.println(character);  
                character = fileReader.read(); // IOException ?  
            }  
        } finally { fileReader.close(); /*dans tous les cas*/ }  
    } catch (IOException exception) {
```

Gestion de différents types d'exceptions

```
try { FileReader fileReader = new FileReader(fileName);  
    try {  
        int character = fileReader.read(); // IOException ?  
        while (character!=-1) {  
            System.out.println(character);  
            character = fileReader.read(); // IOException ?  
        }  
    } finally { /* à faire dans tous les cas. */  
        fileReader.close();  
    }  
} catch (FileNotFoundException exception) {  
    System.out.println("File "+fileName+" not found.");  
} catch (IOException exception) {  
    exception.printStackTrace();  
}
```

Exceptions pour des piles (1/2)

```
public class Stack<T> {  
    private Object[] stack;  
    private int size;  
  
    public Stack(int capacity) {  
        stack = new Object[capacity];  
        size = 0;  
    }  
}
```

Exceptions pour des piles (2/2)

```
public class Stack<T> {  
    public void push(T object) throws FullStackException {  
        if (size == stack.length)  
            throw new FullStackException();  
        stack[size] = object;  
        size++;  
    }  
    public T pop() throws EmptyStackException {  
        if (size == 0) throw new EmptyStackException();  
        size--;  
        T object = (T)stack[size];  
        stack[size]=null;  
        return object;  
    }  
}
```


Définition des exceptions pour les piles

```
public class StackException extends Exception {  
    public StackException(String msg) {  
        super(msg);  
    }  
}  
  
public class FullStackException extends StackException {  
    public FullStackException() {  
        super("Full stack.");  
    }  
}  
  
public class EmptyStackException extends StackException {  
    public EmptyStackException() {  
        super("Empty stack.");  
    }  
}
```

Exemple d'utilisation (1/2)

```
Stack<Integer> stack = new Stack<Integer>(2);
```

```
try {  
    stack.push(1);  
    stack.push(2);  
    stack.push(3);  
} catch (StackException e) {  
    e.printStackTrace();  
}
```

```
FullStackException: Full stack.  
    at Stack.push(Stack.java:13)  
    at Main.main(Main.java:10)
```

Exemple d'utilisation (2/2)

```
Stack<Integer> stack = new Stack<Integer>(2);
```

```
try {  
    stack.push(1);  
    stack.pop();  
    stack.pop();  
} catch (StackException e) {  
    e.printStackTrace();  
}
```

```
EmptyStackException: Empty stack.  
    at Stack.pop(Stack.java:18)  
    at Main.main(Main.java:10)
```

Section 17

Les énumérations

Énumérations en Java

En programmation, une *énumération* est un type spécial qui permet de définir des variables pouvant prendre des valeurs parmi un ensemble prédéfini de constantes.

Il est possible de définir des énumérations en Java grâce au mot-clé `enum`

```
enum Suit{  
    SPADES,  
    HEARTS,  
    DIAMONDS,  
    CLUBS;  
}
```

Énumérations en Java

```
enum Suit {  
    SPADES, HEARTS, DIAMONDS, CLUBS;  
}
```

Une énumération est une classe avec des éléments prédéfinis et statiques.

On peut donc tester directement l'égalité avec l'opérateur égal car il n'y qu'un objet et donc qu'une référence qui correspond à chaque valeur possible.

```
Suit suit = Suit.SPADES;  
/* ... */  
if (suit == Suit.SPADES)  
{  
    /* ..... */  
}
```

Définition de champs, de méthodes et d'un constructeur

```
public enum Suit {  
    SPADES("Pique", "Pi"), HEARTS("Coeur", "Co"),  
    DIAMONDS("Carreau", "Ca"), CLUBS("Trèfle", "Tr");  
  
    private final String frenchName;  
    private final String frenchSymbol;  
  
    Suit(String name, String symbol) {  
        this.frenchName = name;  
        this.frenchSymbol = symbol;  
    }  
  
    public String frenchName() { return frenchName; }  
    public String frenchSymbol() { return frenchSymbol; }  
}
```

Toutes les énumération de java étendent la classe Enum.

Quelques méthodes utiles :

- `String name()` : retourne le nom de la constante tel que déclaré dans l'enum.
- `int ordinal()` : retourne la position de la constante dans la déclaration de l'enum (commençant par 0).

Tout enum définit aussi une méthode statique `values()` renvoyant un tableau contenant les constantes dans l'ordre de leurs déclarations.

Exemple d'utilisation d'énumération (1/2)

```
public static void main(String[] args) {  
    Suit[] values = Suit.values();  
    for (Suit suit : values)  
        System.out.printf("Le symbole de %s est %s.\n",  
                           suit.frenchName(), suit.frenchSymbol());  
}
```

Affichage :

Le symbole de Pique est Pi.

Le symbole de Coeur est Co.

Le symbole de Carreau est Ca.

Le symbole de Trèfle est Tr.

Exemple d'utilisation d'énumération (2/2)

```
public static void main(String[] args) {  
    for (Suit suit : Suit.values())  
        System.out.printf("The position of %s is %d.\n",  
                           suit.name(), suit.ordinal());  
}
```

Affichage :

The position of SPADES is 0

The position of HEARTS is 1

The position of DIAMONDS is 2

The position of CLUBS is 3

Mot-clé switch et enum

Supposons qu'on est une enum pour les jour de la semaine :

```
enum Day
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY;
}
```

Mot-clé switch et enum

```
public void dayIsLike() {  
    switch (day) {  
        case MONDAY:  
            System.out.println("Mondays are bad.");  
            break;  
        case FRIDAY:  
            System.out.println("Fridays are better.");  
            break;  
        case SATURDAY:  
        case SUNDAY:  
            System.out.println("Weekends are best.");  
            break;  
        default:  
            System.out.println("Midweek days are so-so.");  
    }  
}
```

Section 18

Mot-clé final

Première utilisation du mot-clé `final`

Mot-clé `final` dans la déclaration d'un attribut : interdit la modification de la valeur de l'attribut après la construction de l'objet.

Exemple :

```
public class Integer {  
    private final int value;  
    public Integer(int value) {  
        this.value = value;  
    }  
}
```

- Un attribut `final` doit être initialisé après la construction de l'instance
- La valeur de l'attribut ne peut plus être modifiée ensuite

Deuxième utilisation du mot-clé final

Mot-clé final dans la déclaration d'une variable : interdit la modification de la valeur de la variable après la première affectation.

```
public class Stack<T> { /* ... */  
    public T pop() {  
        final T top = array[size-1];  
        array[size-1] = null;  
        size--;  
        return top;  
    }  
}
```

```
public final class Math {  
    public static final double PI = 3.14159265358979323846;  
}
```

Troisième utilisation du mot-clé final

Mot-clé final dans la déclaration d'une méthode : interdire la redéfinition d'une méthode dans une sous-classe

```
public class Integer {  
    /* ... */  
    final public Integer add(Integer val) {  
        return new Integer(this.value + val.value);  
    }  
}
```

- Une classe étendant Integer ne peut pas redéfinir (donner une nouvelle implémentation) la méthode add

Quatrième utilisation du mot-clé final

Mot-clé final dans la déclaration d'une classe : interdit l'extension de la classe

```
final public class Integer {  
    /* ... */  
}
```

- Il devient impossible de créer une classe étendant Integer
- On le fait souvent pour des raisons de sécurité et d'efficacité
- De nombreuses classes de la bibliothèque standard de Java sont final comme Math, String et System

Section 19

Commentaires et documentation

Commentaires inutiles

```
String get(String[] source, int index) {  
    // Teste si l'index est dans les limites du tableau.  
    if (index < 0 || index >= source.length)  
        return null;  
    return source[index];  
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```
boolean indexIsInBounds(String[] source, int index) {  
    return index >= 0 && index < source.length;  
}  
  
String get(String[] source, int index) {  
    if (!indexIsInBounds(source, index)) return null;  
    return source[index];  
}
```

Commentaires inutiles

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */  
boolean isAvailable() {  
    return true;  
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */  
boolean isAvailable() {  
    return false;  
}
```

Commentaires inutiles = répétition

Commentaires inutiles

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
    /* le résultat sera stocké dans cette liste : */
    List<Rectangle> list2 = new ArrayList<Rectangle>();
    for (Rectangle x : list)
        if (x.w == x.h /* un carré ? */)
            list2.add(x);
    return list2;
}

class Rectangle {
    public int w; /* largeur */
    public int h; /* hauteur */
}
```

Commentaires inutiles

On peut se passer de commentaire en rajoutant une méthode et en nommant correctement les éléments du code.

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {  
    List<Rectangle> squares = new ArrayList<Rectangle>();  
    for (Rectangle rectangle : rectangles)  
        if (rectangle.isSquare())  
            squares.add(rectangle);  
    return squares;  
}  
  
class Rectangle {  
    private int width, height;  
    boolean isSquare() {  
        return width == height;  
    }  
}
```

Commentaires utiles

Des commentaires pour décrire les tâches à réaliser peuvent être utiles

```
void processElement(Stack<Formula> stack,
                    String element) {
    // TODO : prendre en compte les signes '-' et '/'
    switch (element) {
        case "+": processSum(stack); break;
        case "*": processProduct(stack); break;
        default : processInteger(stack, element);
        break;
    }
}
```

Commentaires utiles

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns true if this list contains the
 * specified element. More formally, returns
 * true if and only if this list contains
 * at least one element e such that
 * (o==null ? e==null : o.equals(e)).
 *
 * @param o element whose presence in this list is
 * to be tested
 * @return true if this list
 * contains the specified element
 */
```

```
public boolean contains(Object o) {
```


JavaDoc permet de générer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, ...

La documentation contient :

- Une description des membres : attributs et méthodes (publics et protégés) des classes
- Une description des classes, interfaces, ...
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

Un bloc de commentaire Java commençant par `/**` deviendra un bloc de commentaire Javadoc qui sera inclus dans la documentation du code source.

Tag	Description
@author	pour préciser l'auteur de la fonctionnalité
@deprecated	indique que l'attribut, la méthode ou la classe est dépréciée
@return	pour décrire la valeur de retour
{@code literal}	Formate <code>literal</code> en code
{@link reference}	permet de créer un lien

Pour générer la javadoc en IntelliJ

Tools → generate Javadoc

```
/**
 * The {@code Byte} class wraps a value of primitive
 * type {@code byte} in an object.  An object of type
 * {@code Byte} contains a single field whose type is
 * {@code byte}.
 *
 * <p>In addition, this class provides several methods
 * for converting a {@code byte} to a {@code String}
 * and a {@code String} to a {@code byte}, as well as
 * other constants and methods useful when dealing
 * with a {@code byte}.
 *
 * @author Nakul Saraiya
 * @author Joseph D. Darcy
```

Section 20

Structure d'un projet et paquets

Structure d'un projet

En Java, un projet peut être découpé en paquets (package).

Les paquets permettent de :

- associer des classes afin de mieux organiser le code
- de créer des modules indépendants réutilisables
- d'avoir plusieurs classes qui possèdent le même nom (du moment qu'elles ne sont pas dans le même paquet)

Un paquet (package) :

- est une collection de classes
- peut contenir des sous-paquets

Lors de l'exécution...

Java utilise l'arborescence de fichier pour retrouver les fichiers `.class`

- Une classe (ou une interface) correspond à un fichier `.class`
- Un dossier correspond à un paquet

Les `.class` du paquet `com.univ_amu` doivent :

- être dans le sous-dossier `com/univ_amu`
- le dossier `com` doit être à la racine d'un des dossiers du `ClassPath`

Le `ClassPath` inclut :

- le répertoire courant
- les dossiers de la variable d'environnement `CLASSPATH`
- des archives `JAR`
- des dossiers précisés sur la ligne de commande de java (`-classpath path`)

Lors de la compilation... (1/2)

Le mot-clé `package` permet de préciser le paquet des classes ou interfaces définies dans le fichier :

```
package com.univ_amu;  
public class MyClass { /* ... */ }
```

Java utilise l'arborescence pour retrouver le code des classes ou interfaces :

- Une classe (ou une interface) `MyClass` est cherchée dans le fichier `MyClass.java`
- Le fichier `MyClass.java` est cherché dans le dossier associé au paquet de `MyClass`

Lors de la compilation... (2/2)

```
package com.univ_amu;  
public class MyClass { /* ... */ }
```

Dans l'exemple précédent, il est donc conseillé que le fichier :

- se nomme `MyClass.java`
- se trouve dans le dossier `com/univ_amu` (Par défaut, la compilation crée `MyClass.class` dans `com/univ_amu`)

Utilisation d'une classe à partir d'un autre paquet

Accessibilité :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

On peut désigner une classe qui se trouve dans un autre paquet :

```
package com.my_project;  
    public class Main {  
        public static void main(String[] args) {  
            com.univ_amu.MyClass myClass =  
                new com.univ_amu.MyClass();  
        }  
    }  
}
```

Importer une classe

Vous pouvez également importer une classe :

```
package com.my_project;
import com.univ_amu.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Deux classes de deux paquets différents peuvent avoir le même nom :

- Exemple : `java.util.List` et `java.awt.List`
- Attention de choisir le bon import

Importer un paquet

Vous pouvez également importer toutes les classes d'un paquet :

```
package com.my_project;  
import com.univ_amu.*;  
  
public class Main {  
    public static void main(String[] args) {  
        MyClass myClass = new MyClass();  
    }  
}
```

Remarques :

- Les classes des sous-paquets ne sont pas importées
- Le paquet `java.lang` est importé par défaut

Importer des membres statiques

Depuis Java 5, il est possible d'importer directement des méthodes ou attributs de classes (static).

La syntaxe est la suivante :

```
import static my_package.my_class.my_static_member;
```

Exemple :

```
import static java.lang.Math.PI;  
import static java.lang.Math.pow;
```

Exemple sans import statique

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
        System.out.println(  
            "A circle with a diameter of 5 cm has");  
        System.out.println("a circumference of "  
            + (Math.PI * 5) + " cm");  
        System.out.println("and an area of "  
            + (Math.PI * Math.pow(2.5, 2))  
            + " sq. cm");  
    }  
}
```

Exemple d'import statique

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println(
            "A circle with a diameter of 5 cm has");
        out.println("a circumference of "
            + (PI * 5) + " cm");
        out.println("and an area of "
            + (PI * pow(2.5, 2)) + " sq. cm");
    }
}
```

Un exemple

Le fichier `com/univ_amu/HelloWorld.java` :

```
package com.univ_amu;
    public class HelloWorld {
        public static void main(String[] arg) {
            System.out.println("Hello world ! ");
        }
    }
```

```
$ javac com/univ_amu/HelloWorld.java
```

```
$ ls com/univ_amu
```

```
HelloWorld.java HelloWorld.class
```

```
$ java com.univ_amu.HelloWorld
```

```
Hello world !
```

Quelques remarques

Nommage des paquets :

- Les noms de paquets sont écrits en minuscules
- Pour éviter les collisions, on utilise le nom du domaine à l'envers $\Rightarrow$ `com.google.gson`, `com.oracle.jdbc`
- Si le nom n'est pas valide, on utilise des underscores : $\Rightarrow$ `com.univ_amu`

Fichier JAR (Java Archive) :

- est une archive ZIP pour distribuer un ensemble de classes Java
- contient un *manifest* (qui peut préciser la classe qui contient le main)
- peut également faire partie du ClassPath
- peut être généré en ligne de commande (`jar`) ou avec un IDE

Section 21

Classe interne

Classe imbriquée statique

Il est possible de définir une classe à l'intérieur d'une autre (classe imbriquée ou *nested class*) :

```
public class LinkedList {  
    public static class Node {  
        private String data; private Node next;  
        public Node(String data, Node next) {  
            this.data = data; this.next = next;  
        }  
    }  
}
```

Il est possible d'instancier la classe interne sans qu'une instance de `LinkedList` existe car elle est statique :

```
LinkedList.Node node = new LinkedList.Node("hell", null);
```

Classe imbriquée statique

Rappel

Une classe non-imbriquée publique (`public`) doit être dans un fichier portant son nom.

Interdit !

Fichier `LinkedList.java`

```
public class LinkedList { /*...*/ }
```

```
public class Node { /*...*/ }
```

⇒ erreur à la compilation :

```
Error: (9, 8) java: class Node is public, should be  
declared in a file named Node.java
```

Remarque

Une classe imbriquée peut être publique et accessible depuis l'extérieur

Classe imbriquée statique

Il est également possible de la rendre privée à la classe `LinkedList` :

```
public class LinkedList {  
    private static class Node {  
        private String data;  
        private /*LinkedList.**/Node next;  
        public Node(String data, Node next) {  
            this.data = data;  
            this.next = next;  
        }  
    }  
}
```

Dans ce cas, seules les méthodes de `LinkedList` pourront l'utiliser. Notez que des méthodes statiques définies dans `LinkedList` peuvent également utiliser cette classe

Classe imbriquée statique

Exemple d'implémentation de méthodes dans la classe LinkedList :

```
public class LinkedList {  
    /* Code de la classe interne statique Node. */  
    private Node first = null;  
    public void add(String data) {  
        first = new Node(data, first);  
    }  
    public void print() {  
        Node node = first;  
        while (node!=null) {  
            System.out.print("["+node.data+"]");  
            node = node.next;  
        }  
    }  
}
```

Classe imbriquée statique

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList();  
list.add("a");  
list.add("b");  
list.add("c");  
list.print();
```

Cet exemple produit la sortie suivante :

[c] [b] [a]

Classe imbriquée statique

Une classe imbriquée statique ne peut accéder qu'aux attributs et méthodes statiques de la classe qui la contient :

```
public class LinkedList {  
    private static class Node {  
        /* Champs et méthodes de Node. */  
        boolean isFirst() {  
            return this==first; // interdit !  
        }  
    }  
    private Node first;  
    /* Autres champs et méthodes de LinkedList. */  
}
```

Classe interne

En revanche, si la classe interne n'est pas statique, elle peut accéder aux champs de classe qui la contient :

```
public class LinkedList {  
    private class Node {  
        /* Champs et méthodes de Node. */  
        private boolean isFirst() {  
            return this==first; // autorisé !  
        }  
    }  
    private Node first;  
    /* Autres champs et méthodes de LinkedList. */  
}
```


Classe interne

Java insère dans l'instance de Node une référence vers l'instance de LinkedList qui a permis de la créer :

```
public class LinkedList {  
    private class Node {  
        /* Champs et méthodes de Node. */  
        private boolean isFirst() {  
            return this==/*référenceVersLinkedList.*/first;  
        }  
    }  
  
    public void add(String data) {  
        first = new Node(data, first);  
    }  
  
    /* Autres champs et méthodes de la classe LinkedList. */  
}
```

Classe interne

Il est possible d'utiliser la méthode `isFirst` dans `LinkedList` :

```
public class LinkedList {  
    /* Code de la classe interne statique Node  
       et champs et méthodes de la classe LinkedList. */  
    public void print() {  
        Node node = first;  
        while (node!=null) {  
            System.out.print "["+node.data  
                +", "+node.isFirst()+" ]";  
            node = node.next;  
        }  
    }  
}
```

Classe interne

Exemple d'utilisation de la classe précédente :

```
LinkedList list = new LinkedList(); list.add("a");  
list.add("b");  
list.add("c");  
list.print();
```

Cet exemple produit la sortie suivante :

```
[c,true] [b,false] [a,false]
```

Section 22

Types paramétrés (notions avancées)

Condition sur les paramètres – Problématique

```
public interface Comparable<T> {  
    public int compareTo(T element);  
}  
  
class Greatest {  
    private String element;  
    public void add(String element) {  
        if (this.element==null ||  
            this.element.compareTo(element)<0)  
            this.element = element;  
    }  
    public String get() { return element; }  
}
```

Comment rendre la classe Greatest générique ?

Condition sur les paramètres

```
class Greatest<T extends Comparable<T>> {  
    private T element;  
    public void add(T element) {  
        if (this.element==null  
            || element.compareTo(this.element)>0)  
            this.element = element;  
    }  
    public T get() {  
        return element;  
    }  
}
```

Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public T get() { return element; }  
}  
  
class Card implements Comparable<Card> { /* ... */ }  
class PrettyCard extends Card { /* ... */ }
```

Il n'est pas possible d'écrire les lignes suivantes car PrettyCard n'implémente pas l'interface Comparable<PrettyCard> :

```
Greatest<PrettyCard> greatest =  
    new Greatest<PrettyCard>();  
greatest.add(new PrettyCard(Card.diamond, 7));
```

Syntaxe ? super

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public T get() { return element; }  
}  
  
class Card implements Comparable<Card> { /* ... */ }  
class PrettyCard extends Card { /* ... */ }
```

Il est possible d'écrire les lignes suivantes car PrettyCard implémente l'interface Comparable<Card> et Card super PrettyCard:

```
Greatest<PrettyCard> greatest =  
    new Greatest<PrettyCard>();
```


Wildcard ?

Dans un paramètre de généricité, le symbole ? (appelé *wildcard*) dénote une variable de type anonyme.

On peut la contraindre avec les mot-clés `super` et `extends`.

Exemples :

- `List<?>` : une liste de type quelconque.
- `List<? extends Shape>` : une liste d'instances d'une sous-classe de `Shape`.
- `List<? super Disc>` : une liste d'instances d'une classe ancêtre de `Disc`.
- `E extends Comparable<? super E>` : un type `E` implémentant l'interface `Comparable<P>` pour `P` ancêtre de `E`.

? extends – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public void addAll(List<T> list) {  
        for (T element : list) add(element);  
    }  
}
```

Il n'est pas possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();  
Greatest<Card> greatest = new Greatest<Card>();  
/* ... */  
list.addAll(list);
```

? extends

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {  
    /* ... */  
    public void add(T element) { /* ... */ }  
    public void addAll(List<? extends T> list) {  
        for (T element : list) add(element);  
    }  
}
```

Il est maintenant possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();  
Greatest<Card> greatest = new Greatest<Card>();  
/* ... */  
greatest.addAll(list);
```

Méthodes paramétrées et conditions sur les types

```
class Tools {  
    static <T extends Comparable<T>>  
    boolean isSorted(T[] array) {  
        for (int i = 0; i < array.length-1; i++)  
            if (array[i].compareTo(array[i+1]) > 0)  
                return false;  
        return true;  
    }  
}
```

Exemple :

```
String[] array = { "ezjf", "aaz", "zz" };  
System.out.println(Tools.isSorted(array));
```

Méthodes paramétrées et conditions sur les types

Méthode pour copier une liste src vers une autre liste dest :

```
static <T> void    copy(List<? super T> dest, List<? extends T>
src)
```

On suppose qu'on a une classe `MovingPixel` qui étend `Pixel` qui elle-même étend `Point`.

On peut écrire :

```
List<MovingPixel> src = new ArrayList<>();
List<Point> dest = new ArrayList<>();
Collections.<Pixel>copy(dest, src);
```

Utilisation extends et super

Lorsqu'on a une collection d'objets de type `T` :

- En entrée/écriture, on veut donner des objets qui ont au moins tous les services des objets de type `T`.

On doit donc donner des objets dont la classe étend `T : ? extends T`

- En sortie/lecture, on veut récupérer des objets qui ont au plus tous les services des objets de type `T`.

On doit donc récupérer des objets qui sont étendu par la classe `T : ? super T`

Section 23

Interfaces (notions avancées)

Les classes anonymes

Supposons que nous ayons l'interface suivante :

```
Interface ActionListener {  
    public void actionPerformed(ActionEvent event);  
}
```

Il est possible de :

- définir une classe anonyme qui implémente cette interface
- d'obtenir immédiatement une instance de cette classe

```
ActionListener listener = new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        counter++;  
    }  
});
```


Les classes anonymes

```
public class Window {  
    private int counter;  
    public Window() {  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter++;  
            }  
        });  
    }  
}
```

Les classes anonymes

Il est possible d'utiliser des attributs de la classe "externe" :

```
public class Window {  
    private Counter counter = new Counter();  
    public Window() {  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter.count();  
            }  
        });  
    }  
}
```

Les classes anonymes

Il est possible d'utiliser des variables finales de la méthode :

```
public class Window {  
    public Window() {  
        final Counter counter = new Counter();  
        Button button = new Button("count");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                counter.count();  
            }  
        });  
    }  
}
```

Java 8 : Lambda expressions

Avec Java 8, il est possible d'écrire directement :

```
public class Window {  
    public Window() {  
        Button button = new Button("button");  
        button.addActionListener(  
            event -> System.out.println(event)  
        );  
    }  
}
```

Explication : ActionListener possède une seule méthode donc on peut affecter une lambda expression à une variable de type ActionListener.

```
public interface ActionListener {  
    public void actionPerformed(ActionEvent event);  
}
```

Interfaces fonctionnelles

En Java 8, une interface n'ayant qu'une méthode abstraite est une interface fonctionnelle. Les quatre interfaces fonctionnelles suivantes (et plein d'autres) sont déjà définies :

```
public interface Predicate<T> {  
    public boolean test(T t);  
}  
  
public interface Function<T,R> {  
    public R apply(T t);  
}  
  
public interface Consumer<T> {  
    void accept(T t);  
}  
  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Syntaxe d'une lambda expression

Pour instancier une interface fonctionnelle, on peut utiliser une lambda expression :

L'interface suivante :

```
public interface MyFunctionalInterface{  
    public T myMethod(A arg1, B arg2, C arg2);  
}
```

peut être instancier par :

```
MyFunctionalInterface fonc =  
    (arg1, arg2, arg2)  
    -> /* expression définissant le résultat de myMethod */
```

Si T est void alors l'expression peut être void comme un println.

Exemples de lambda expression

On considère une classe `Person` avec deux attributs `name` et `age` et les getters et setters associés.

On a le droit d'écrire les lambda expressions suivantes en Java :

- `person -> person.getAge() >= 18` de type `Predicate<Person>`
- `person -> person.getName()` de type `Function<Person,String>`
- `name -> System.out.println(name)` de type `Consumer<Person>`

Remarques

- Il n'est pas nécessaire de mettre le type des paramètres.
- On peut omettre les parenthèses dans le cas où il n'y a qu'un seul paramètre

Référence de méthodes

Dans un certain nombre de cas, une lambda expression se contente d'appeler une méthode ou un constructeur.

Il est plus clair dans ce cas de se référer directement à la méthode ou au constructeur.

Lambda expression	référence de méthode
<code>x -> Math.sqrt(x)</code>	<code>Math::sqrt</code>
<code>name -> System.out.println(name)</code>	<code>System.out::println</code>
<code>person -> person.getName()</code>	<code>Person::getName</code>
<code>name -> new Person(name)</code>	<code>Person::new</code>

Stream = Abstraction d'un flux d'éléments sur lequel on veut faire des calculs

Ce n'est pas une Collection d'élément car un Stream ne contient pas d'élément

Création d'un Stream :

- À partir d'une collection comme une liste avec `list.stream()`
- À partir d'un fichier : `Files.lines(Path path)`
- À partir d'un intervalle : `IntStream.range(int start, int end)`

Exemples d'utilisation

```
persons
    .stream()
    .filter(person -> person.getAge() >= 18)
    .map(person -> person.getName())
    .forEach(name -> System.out.println(name));
```

Types des paramètres et retours des méthodes :

- `stream()` → `Stream<Person>`
- `filter(Predicate<Person>)` → `Stream<Person>`
- `map(Function<Person, String>)` → `Stream<String>`
- `forEach(Consumer<String>)`

Cycle de vie d'un Stream

Un Stream est toujours utilisé en trois phases :

- Création du Stream (à partir d'une collection, d'un fichier, ...),
- Opérations intermédiaires sur le Stream (suppression d'éléments, transformation de chaque élément, combinaison) qui prennent un Stream et renvoient un Stream,
- Une seule opération terminale du Stream (calcul de la somme, de la moyenne, application d'une fonction sans retour sur chaque élément, ...).

Opérations intermédiaires possibles sur un Stream

- `Stream<E> filter(Predicate<? super E>)` : sélectionne si un élément reste dans le Stream
- `<R> Stream<R> map(Function<? super E, ? extends R>)` : transforme les éléments du Stream en leur appliquant une fonction
- `Stream<E> sorted(Comparator<? super E>)` : trie les éléments

Opérations terminales possibles sur un Stream

- `long count()` : compte le nombre d'éléments
- `long sum()` : somme les éléments (entiers ou double)
- `Stream<E> forEach(Consumer<? super E>)` : Appele le consumer pour chaque élément
- `allMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour tous les éléments
- `anyMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour au moins un élément
- `collect(Collectors.toList())` : crée une liste avec les éléments du Stream

Section 24

La gestion de projet

- ➊ Rechercher et caractériser les fonctions qu'un logiciel devrait avoir pour satisfaire les besoins de son utilisateur.
- ➋ Hiérarchiser ces fonctions.

Utilisée pour **créer** mais aussi **améliorer** un logiciel (ou plus généralement un produit).

Identification des fonctions

- Fonction **principale** : la raison pour laquelle le logiciel est créé. Cette fonction peut être divisée en plusieurs fonctions simples.
- Fonction **contrainte** : conditions que le produit doit vérifier mais qui ne sont pas sa raison d'exister (par exemple la sécurité).
- Fonction **complémentaire** : ce qui facilite l'utilisation du logiciel, l'améliore ou le complète.

Cahier des charges

Un outil de l'analyse fonctionnelle : lister, décrire et hiérarchiser les fonctions.

Exemple de la tondeuse

Fonction principale :

- Couper l'herbe.

Fonctions contraintes :

- Respecter les normes de sécurité;
- Pouvoir être conservée à l'extérieur;
- S'adapter au terrain;
- Ne pas être trop bruyante;
- Ne pas être trop encombrante;
- etc,...

Fonctions complémentaires:

- Ramasser l'herbe;

Quelques notions de gestion de projet

- **Livrable** : produit destiné à la livraison : documentation, code, tests, etc. . .
- **Jalon** : fin d'une étape ou évènement important.

Le début d'un projet et sa fin sont des jalons. On fixe des jalons intermédiaires pour mesurer l'avancée du projet.

Un jalon peut être un livrable lié à une date.

La gestion de projet en informatique

Développer un logiciel implique de :

- comprendre le besoin du client;
- en tirer un cahier des charges;
- concevoir l'architecture du logiciel;
- développer le logiciel;
- tester s'il fonctionne comme prévu;
- le maintenir.

Deux types de management pour y parvenir :

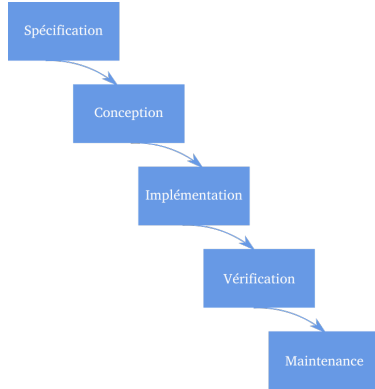
- méthodes traditionnelles;
- méthodes agiles.

Cascade (ou Waterfall)

Méthode traditionnelle, inspirée par le BTP.

Passage d'une phase à l'autre uniquement quand la précédente est terminée et vérifiée.

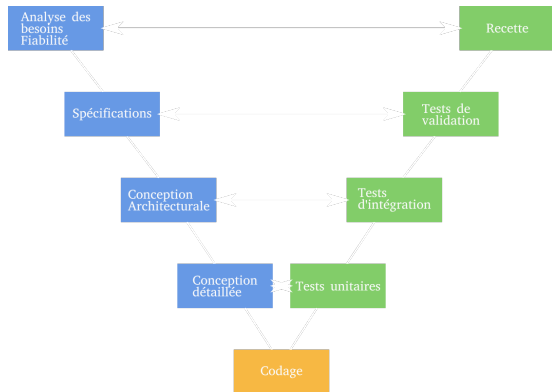
Cascade



Inconvénient : très peu réactif en cas d'erreur ou de modification nécessaire en cours de projet.

Cycle en V

Un amélioration du modèle en cascade : limiter le retour aux étapes précédentes.
Standard depuis 1980.



Pro et con du cycle en V

Phase ascendante : renvoie de l'information sur la phase correspondante pour améliorer le logiciel.

Phase descendante : anticipation des attendus des étapes montantes.

Inconvénient : détache complètement la conception de la réalisation.

Objectifs :

- plus pragmatique que les méthodes traditionnelles;
- impliquer le client pendant le développement;
- être réactif et s'adapter aux changements.

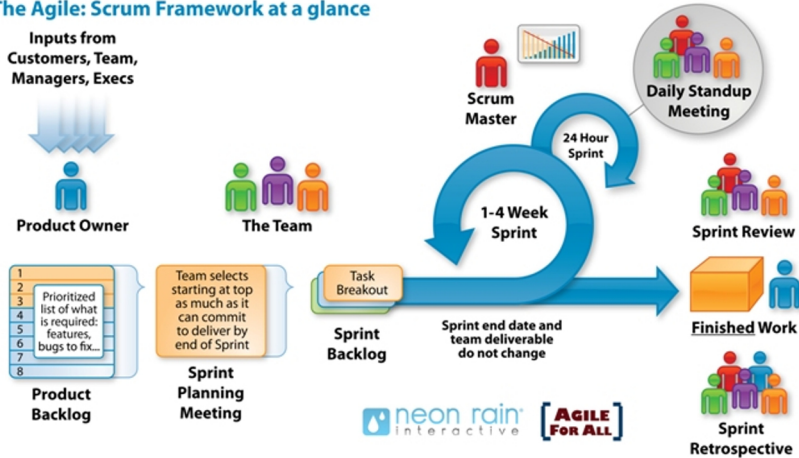
Définition par le manifeste Agile (2001).

Exemples :

- développement rapide d'applications (RAD);
- adaptative software development;
- extreme programming (XP);
- Scrum.

SCRUM

The Agile: Scrum Framework at a glance



Section 25

Tests unitaires et développement par les tests (TDD)

Pourquoi les tests unitaires ?

- assurer la correction du code;
- trouver rapidement les bugs pendant le développement;
- identifier des manques dans les spécifications;
- faciliter les modifications.

Cycle de développement du TDD

- ❶ Écrire un test : définit une fonction ou l'amélioration d'une fonction.
- ❷ Lancer le test : il doit échouer (pour montrer que le test fait référence à une fonctionnalité qui n'existe pas encore, et qu'il fonctionne bien).
- ❸ Écrire le code qui fait passer le test (et rien d'autre).
- ❹ Lancer les tests : si le test ne passe pas, retourner à l'étape 3.
- ❺ Refactorer : déplacer du code si besoin, supprimer la duplication, vérifier les noms. . .

Théorisé par Kent Beck

- réfléchir à ce que fait le code avant de coder;
- garantir que tout est testé (puisque rien n'est écrit sans test);
- réduire significativement le temps passé à debugger;
- avancer par petits pas;
- voir son avancée.

Section 26

Les principes solides

Les cinq principes pour créer du code SOLID

- **Single Responsibility Principle (SRP)** : Une classe ne doit avoir qu'une seule responsabilité
- **Open/Closed Principle (OCP)** : Programme ouvert pour l'extension, fermé à la modification
- **Liskov Substitution Principle (LSP)** : Les sous-types doivent être substituables par leurs types de base
- **Interface Segregation Principle (ISP)** : Éviter les interfaces qui contiennent beaucoup de méthodes
- **Dependency Inversion Principle (DIP)** : Les modules d'un programme doivent être indépendants. Les modules doivent dépendre d'abstractions.

Single Responsibility Principle (SRP)

Principe SRP

Une classe ne doit avoir qu'une **responsabilité = raison de changer**

Les effets néfastes des responsabilités multiples

- Difficulté à nommer car la classe est trop complexe
- Difficulté à réutiliser le code car seulement une des responsabilités est réutilisable
- Difficulté à mettre à jour le code car changer l'implémentation d'une partie impacte les autres parties

Pourquoi SRP ?

- Facilite le nommage et documentation
- Facilite l'écriture des tests
- Facilite la réutilisation des classes

Open/Closed Principle (OCP)

Principe OCP

Programme ouvert pour l'extension, fermé à la modification

Signification

Vous devez pouvoir ajouter une nouvelle fonctionnalité :

- en ajoutant des classes (Ouvert pour l'extension)
- sans modifier le code existant (fermé à la modification)

Avantages

- Le code existant n'est pas modifié $\Rightarrow$ augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Violation OCP (1/2)

```
public class Circle {  
    Point center;  
    int radius;  
}
```

```
public class Rectangle {  
    Point TopLeft, RightBottom;  
}
```

Violation OCP (2/2)

```
public class GraphicTools {  
    static void draw(Rectangle r){}  
    static void draw(Circle c){}  
    static void draw(Object[] shapes){  
        for(Object o : shapes){  
            if(o instanceof Rectangle){  
                draw((Rectangle) o);  
            }  
            if(o instanceof Circle){  
                draw((Circle) o);  
            }  
        }  
    }  
}
```

Programme propre (1/2)

```
interface Shape {  
    void draw();  
}
```

```
public class Circle implements Shape {  
    Point center;  
    int radius;  
    public void draw(){ ... }  
}
```

```
public class Rectangle implements Shape {  
    Point TopLeft, RightBottom;  
    public void draw(){ ... }  
}
```

Liskov Substitution Principle (LSP)

Principe

Les sous-types doivent être substituables par leurs types de base.

Signification

Si une classe **A** étend une classe **B** alors un programme **P** écrit pour manipuler des instances de type **B** doit avoir le même comportement s'il manipule des instances de la classe **A**.

Avantages

- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité

Liskov Substitution Principle (LSP)

Violation de LSP :

```
public void test(Rectangle r) {  
    r.setWidth (2);  
}  
r.setHeight(3);  
if (r.getArea() != 3*2)  
  
System.out.println("bizarre !");
```

La mauvaise question :

Un carré est-il un rectangle ?

La bonne question :