Programmation 2 : quatrième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

8 ou 10 octobre 2018

Composition et délégation

Interaction entre les classes/instances

Une méthode peut utiliser les propriétés et méthodes d'un autre instance/classe :

```
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 leur 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 double 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 double getCenter(){ return center; }
    public double getRadius(){
        return center.distanceTo(point);
    }
}
```

Agrégation récursive

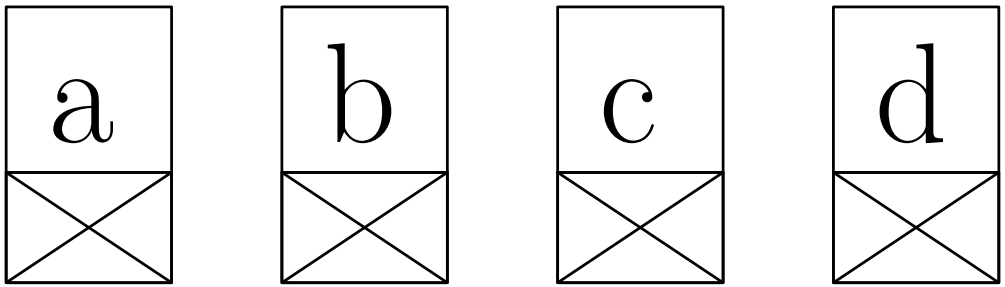
Il est possible de créer des structures récursives (les attributs de la classe contiennent des références vers une instance de la classe).

```
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");
```

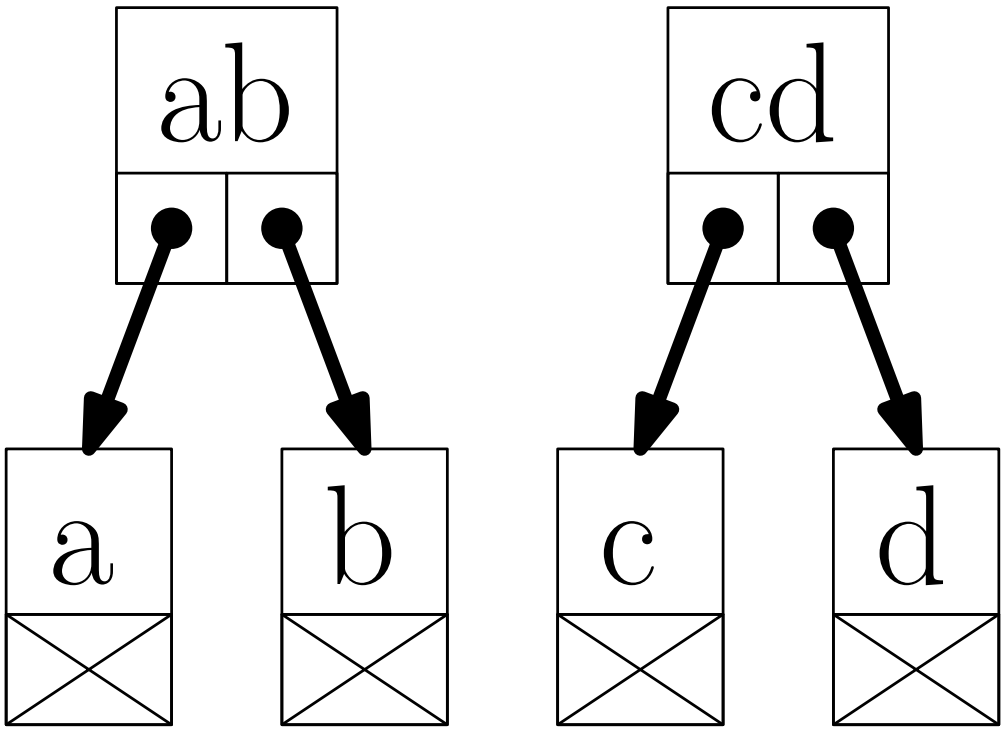
Utilisation d'une structure récursive



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);
```

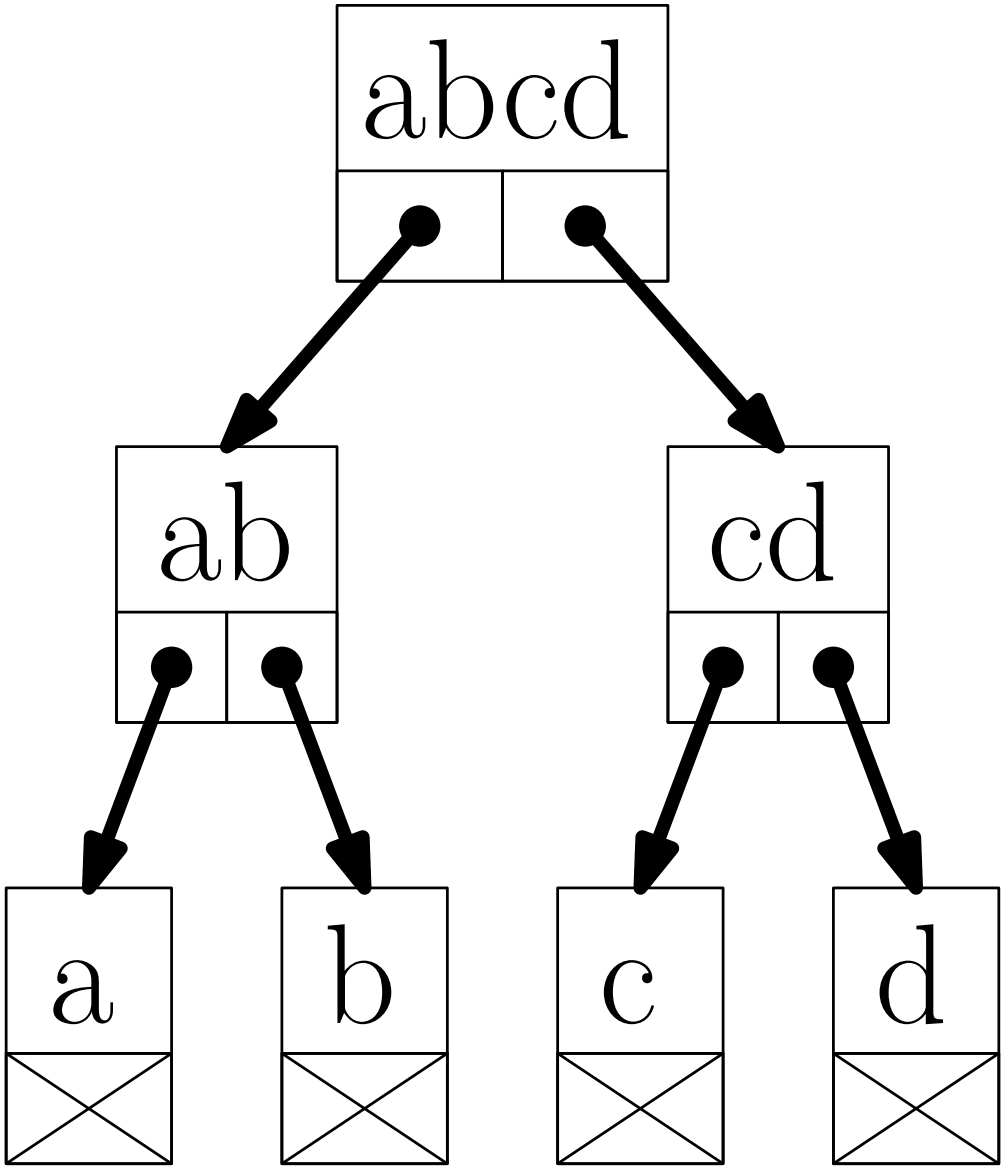
Utilisation d'une structure récursive



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

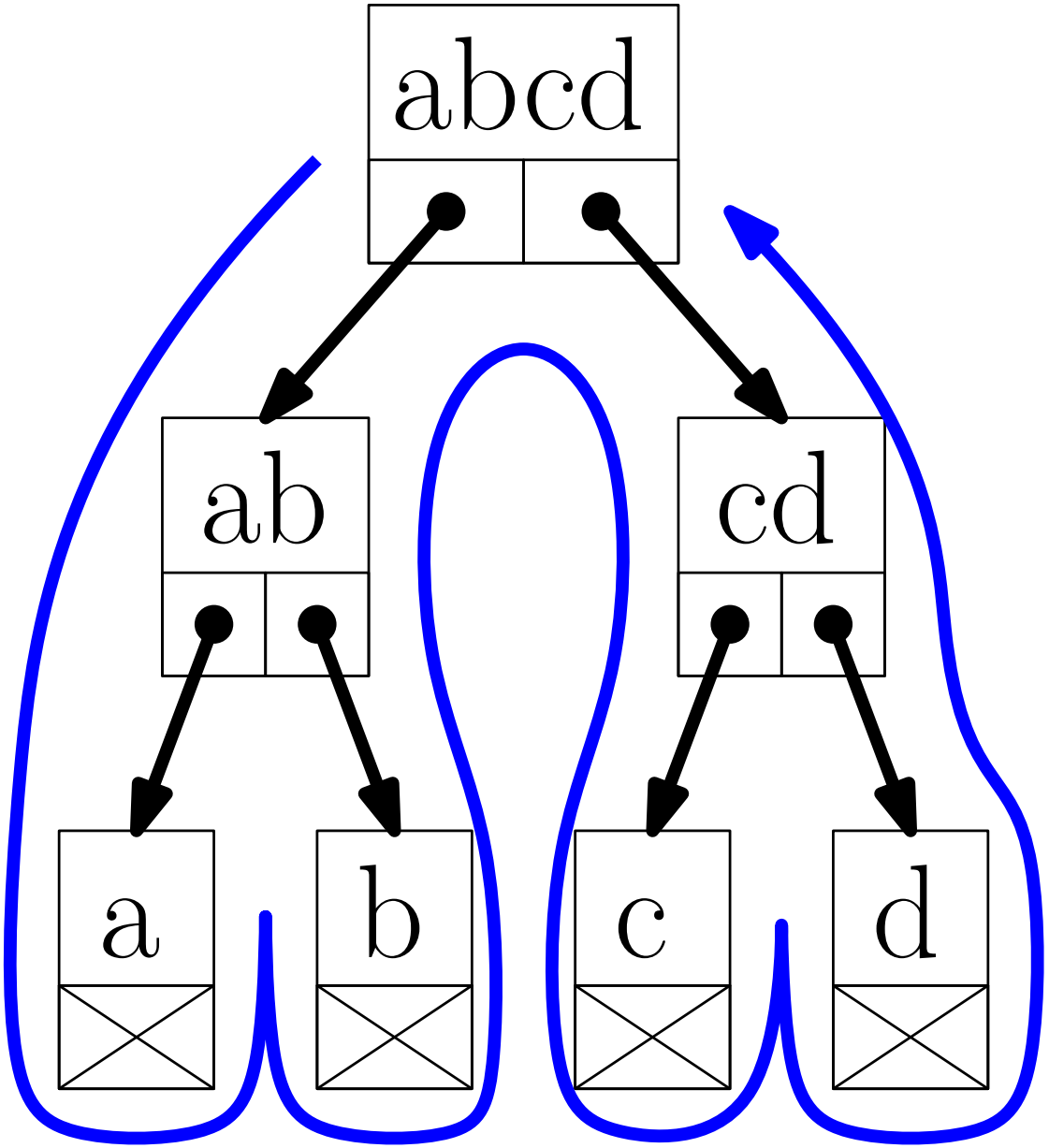


Agrégation récursive et méthodes

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



Tests et développement

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 d'un programme indépendamment les unes des autres.
- **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**

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 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 premier paramètre qui permet d'identifier l'assertion.

Gestion de version

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
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

Git

- 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

Exemple 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

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 transtypages 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 ohterIntValue = 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);
    }
}
```