Encapsulation et interfaces

Cours 5 / GL / LP DA2I

Cédric Lhoussaine

2019-2020

Outline

Rappels sur les variables

Portée des variables

Les données ne sont pas connues partout

- variables locales : dans le bloc $\{\ldots\}$ où elles sont déclarées
- variables d'instance : propres à chaque instance
- variables de classe : partagées par toutes les instances d'une même classe
- processus; mémoire partagée
- var. d'environnement; fichiers

  Règle générale: en Java, une variable locale masque toujours les éventuelles variables globales de même nom.

Le problème du masquage des attributs

```
public class Date {
    private int jour, mois, annee;
    public Date(int j, int m, int a) {
        int jour = j;
        int mois = m;
        int annee = a;
    }
}
```

Catastrophe: les variables locales du constructeur masquent les attributs!

Solution

lever le masquage avec `this`:

```
public class Date {
    private int jour, mois, annee ;
    public Date(int jour, int mois, int annee) {
        this.jour = jour ;
        this.mois = mois ;
        this.annee = annee ;
    }
}
```

Exemples

Quels affichages réalisent les classes C1 et C2 ?

```
public class C1 {
    static int i=100 ;
    public static void main (String [] arg) {
        System.out.print(i+" ");
        for (int i=0; i<10; i++)
            System.out.print(i+" ");
        System.out.println(i);
    }
}
```

```
public class C2 {
    static int i=100 ;
    public static void main (String [] arg) {
        int i = 0;
        System.out.print(i+" ");
        for (i=0; i <10; i++)
            System.out.print(i+" ");
        System.out.println(i);
    }
}
```

Exemples

Résultat pour `C1`:

```
100 0 1 2 3 4 5 6 7 8 9 100
```

Résultat pour `C2`:

```
0 0 1 2 3 4 5 6 7 8 9 10
```

Persistance des variables

Les données peuvent persister en mémoire + ou - longtemps:

- résultats temporaires d'évaluation d'une expression
- variables locales d'un bloc, d'une fonction
- ...

Un objet est une entité autonome dans la mémoire $\rightarrow$ il ne "disparaît" pas lorsqu'il a terminé une méthode!

Un objet est persistant

Persistance des variables

Les données peuvent persister en mémoire + ou - longtemps:

- résultats temporaires d'évaluation d'une expression
- variables locales d'un bloc, d'une fonction
- ...

Un objet est une entité autonome dans la mémoire → il ne "disparaît" pas lorsqu'il a terminé une méthode!

Un objet est persistant

Deux mécanismes permettent de s'en débarrasser:

- les destructeurs (C++) : désallocation explicite
- le *garbage collector* ou ramasse-miettes (Java, Smalltalk) → nettoyage implicite (absence de référencement)

L'encapsulation

Principes

Un utilisateur n'a pas besoin de connaître les détails de l'implémentation d'un objet pour l'utiliser.

- on utilise un objet par envoi de messages
- la spécification des méthodes doit donc suffire
- la manipulation directe d'attributs peut nuire à la généralité du code

. . . d'ailleurs dans certains langages (Smalltalk, Ruby) on ne peut pas les manipuler !

Principes

Un utilisateur n'a pas besoin de connaître les détails de l'implémentation d'un objet pour l'utiliser.

- on utilise un objet par envoi de messages
- la spécification des méthodes doit donc suffire
- la manipulation directe d'attributs peut nuire à la généralité du code

. . . d'ailleurs dans certains langages (Smalltalk, Ruby) on ne peut pas les manipuler !

La plupart des attributs, des méthodes, ou même des classes peuvent et doivent être masqués. Objectifs:

- sécuriser l'accès aux données
- faciliter les changements d'implémentation
- faciliter la réutilisation

Réalisation de l'encapsulation

On peut restreindre l'accessibilité des attributs et des méthodes d'une classe.

Plusieurs niveaux de protection:

- `private` : les attributs ou méthodes masqués ne sont manipulables que par les méthodes de la *même classe*;
- sans indication: que par les classes d'un *même package*;
- `protected`: que par les classes d'une *même famille*;
- `public` : accessibles par *n'importe quelle classe*.

Protection d'accès

Accès aux attributs* et de méthodes:

	classe	package	sous-classes	partout
`private`	X			
(rien)	X	X		
`protected`	X	X	X	
`public`	X	X	X	X

Les classes ne peuvent être modifiées que par `public` ou rien.

La notion de signature

Signature d'une méthode =

- ses modificateurs (accès / classe / constante)
- son type de retour
- son nom
- ses paramètres et leur type
- exceptions éventuelles (`throws`)

modificateurs	type	nom	paramètres
`public static`	`void`	`main`	`()`
`private`	`int`	`pgcd`	`(int a,int b)`

L'utilisateur n'a pas besoin de savoir *comment est réalisé* un objet (ou un package). Il doit juste savoir *comment l'utiliser*.

Principe d'encapsulation

On ne laisse voir d'un composant que ce qui est indispensable à l'utilisateur:

- les signatures publiques d'une classe
- les classes publiques d'un package

Notion d'interface

Interface vs. implémentation

Définition
L'ensemble des signatures publiques d'une classe constitue son interface

- concept général: ce qu'on peut voir de la classe (ce qui apparaît dans la javadoc)
- s'oppose à son implémentation qui est cachée

De même l'ensemble des classes publiques d'un package = l'interface du package

L'abstraction

Ne retenir que les caractéristiques essentielles d'un objet du point de vue d'un observateur donné.

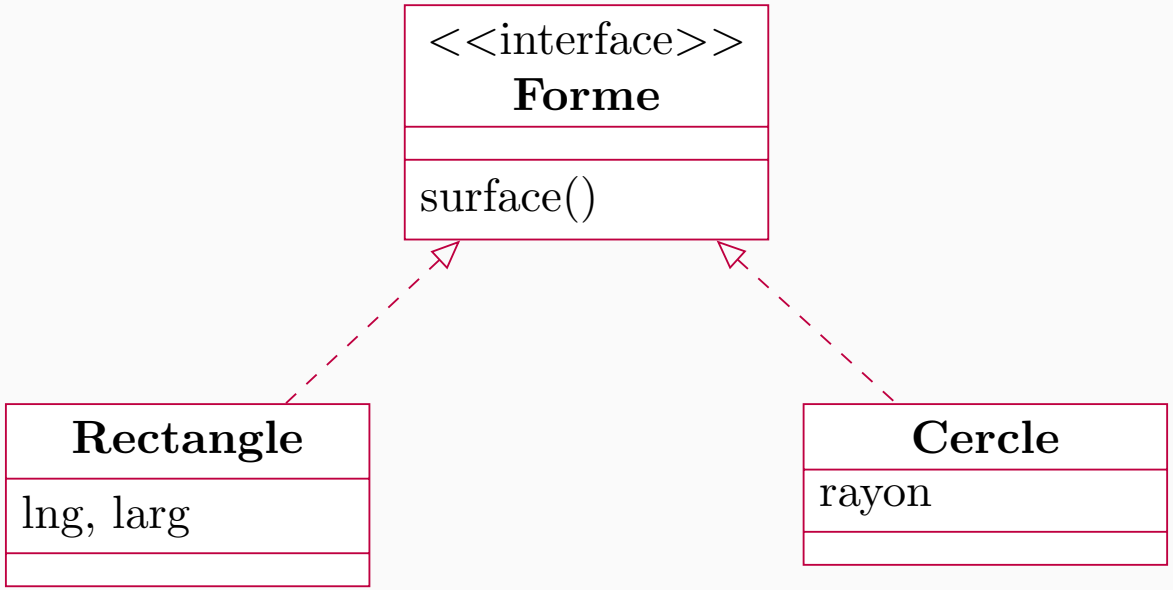
L'abstraction

Ne retenir que les caractéristiques essentielles d'un objet du point de vue d'un observateur donné.

Les interfaces d'un objet
Il faut donner à l'utilisateur d'un objet un sous-ensemble du comportement complet, qui correspond à ses attentes (à son point de vue).

Plusieurs utilisateurs / plusieurs points de vue / plusieurs interfaces.

Implémenter des interfaces



Réalise la relation sorte-de.

Le langage Java permet de définir des interfaces:

- ensembles de signatures de méthodes (pas de code)
- implémentées par une classe (qui définit le code)
  Une interface est en quelque sorte un contrat garantissant qu'une classe fournit un comportement donné.

Exemple d'interface

```
public interface Forme {
    double surface() ;
    double PI = 3.14159 ;
}

public class Rectangle implements Forme {
    private double lng, larg ;
    public Rectangle(double a, double b) {
        lng = a; larg = b ;
    }
    public double surface() { return lng*larg; }
}

public class Cercle implements Forme {
    private double rayon ;
    public Cercle(double r){ rayon = r ; }
    public double surface() {
        return PI * rayon * rayon ;
    }
}
```

L'interface

- ne doit avoir que des méthodes abstraites = signature seule sans code (ou presque...)
- ces méthodes sont tacitement public
- les seuls attributs autorisés sont des constantes de classe → tout attribut est supposé tacitement `public static final`
- une classe qui implémente une interface doit définir le code de toutes les méthodes de cette interface → notion de CONTRAT
- une classe peut implémenter un nombre quelconque d'interfaces

```
class Chat implements Compagnon, Mammifere
```

Une interface ne peut pas être instanciée!

Inclusion de types
Si `C` implémente les interfaces `I` et `J`, alors les instances de `C` peuvent être référencées avec des variables de type `C`, `I` ou `J`.

```
Rectangle r = new Rectangle(5,3);
Cercle c = new Cercle(1);
Forme f1 = r, f2 = c;

f1.surface() // => 15.0
f2.surface() // => 3.14159
```

Le code à exécuter pour surface dépend de la classe de l'objet, pas de son type.

Interface et typage

Attention au sens!

- le transtypage est toujours possible et implicite dans le sens *particulier* $\rightarrow$ *général*
- ce n'est pas toujours possible dans l'autre sens
- quand ça l'est, c'est explicite

Interface et typage

```
Forme f1 = new Rectangle(5, 3);
Forme f2 = new Cercle(1);
r = (Rectangle) f1;
c = (Cercle) f1;
```

Résultat:

- accepté à la compilation (vérification des types) comment savoir *a priori* la classe de f1 ?
- erreur à l'exécution (vérification des classes) `Exception in thread "main" java.lang.ClassCastException`

Interfaces et généricité

Les interfaces peuvent utiliser des types génériques

```
public interface Couple<A,B> {
    // A et B sont deux types génériques
    A premier() ;
    B second() ;
}
```

Les classes qui l'implémentent peuvent rester génériques

```
    public class Paire<T> implements Couple<T,T> {
        // on a en fait A = B (= T)
        T un, deux ;
        public Paire(T premier, T second) {
            un = premier ; deux = second ;
        }
        public T premier() { return un ; }
        public T second() { return deux ; }
    }

    //...

Paire<String> binome = new Paire<String>("Dupont", "Dupond");
Paire<Integer> coords = new Paire<Integer>(14, 18) ;
```

. . . ou au contraire n'utiliser que des types concrets

```
public class CoordEchecs implements Couple<Character,Integer> {
    Character colonne;
    Integer ligne;
    public CoordEchecs(char c, int i) {
        colonne = c ; ligne = i ;
    }
    public Character premier() { return colonne ; }
    public Integer second() { return ligne ; }
}

//...

CoordEchecs coup = new CoordEchecs('B', 7) ;
```

Comparable<E>

```java
// java.long.Comparable<E>
public interface Comparable<E> {
    public int compareTo(E other) ;
}
```

- l'interface `Comparable<E>` permet de généraliser les opérateurs de comparaison (<, ==, >)
- comme pour l'égalité logique, la nature de la comparaison est à la charge de l'utilisateur
- retourne un entier négatif si `this` est "plus petit" que `other` positif si c'est l'inverse, zéro en cas d'égalité
- implémentée par la plupart des classes usuelles

Exemple

```
public class Date implements Comparable<Date> {
    ...
    public int compareTo(Date other) {
        if (this.annee != other.annee)
            return this.annee - other.annee ;
        if (this.mois != other.mois)
            return this.mois - other.mois ;
        return this.jour - other.jour ;
    }
    ... }

if (d1.compareTo(d2) < 0)
    System.out.println(d1 + " antérieure à " + d2) ;
```

Emacs 26.3 (Org mode 9.1.9)