Gestion des références

Antoine Nongaillard

BUT Info – IUT – Université de Lille

R2.01 – Séance 02

Plan

Les références

Java traite :

- des types **primitifs** pour les données simples : `char`, `int`, `double`...
- des **classes** pour les données structurées : `String`, `Array`, `StringBuilder`...

L'objet est un espace mémoire où sont stockées les valeurs des attributs. La **référence** de l'objet est l'adresse mémoire de cet espace mémoire.

Pourquoi ? Pour la portabilité des applications : la compilation génère du code binaire qui est interprété par une machine virtuelle et non pas un exécutable.

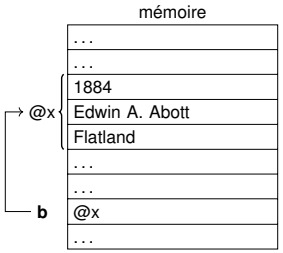
⇒ pas d'adressage direct à l'espace mémoire

En Java, la mémoire est allouée au travers de l'opérateur **new**. La libération de la mémoire est automatique via le **Garbage Collector**. Quand la référence d'un objet n'est plus conservée dans aucune variable, l'objet devient inaccessible et est supprimé automatiquement.

Instanciation et référence

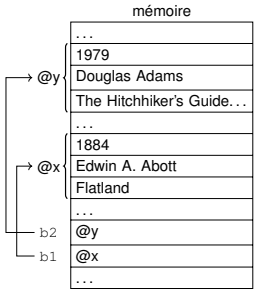
L'opérateur **new** crée un objet à partir d'une classe et retourne sa **référence** ⇒ l'invocation d'un constructeur réserve l'espace mémoire et initialise les attributs.

```
Book b = new Book("Flatland", "Edwin A. Abbott", 1884);
```



Instanciation et référence

```
Book b1;
Book b2;
b1 = new Book("Flatland", "Edwin A. Abbott", 1884);
b2 = new Book("The Hitchhiker's Guide...", "Douglas Adams", 1979);
```

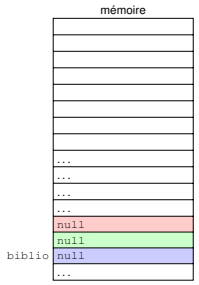


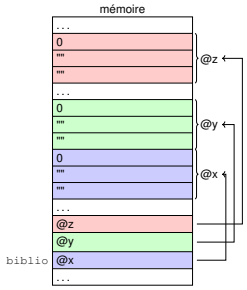
L'instruction `Book b1;` ne créé pas d'objet, mais une variable stockant une référence.

Tableaux d'objets

Les tableaux sont des objets, il faut donc les instancier. Avant l'initialisation individuelle de chaque case, le tableau contient des références à **`null`**.

```
Book[] biblio = new Book[3];
for(int i=0; i<biblio.length; i++)
    biblio[i] = new Book();
```





Plan

Passage par valeur/référence

Au sein d'une classe `PassByCopy` :

```
void methodWithInt(int i) {
  i=5;
  System.out.println("inside: "+i);
}
```

Le programme principal est le suivant :

```
PassByCopy test = new PassByCopy();
int value = 3;
System.out.println("before: "+value);
test.methodWithInt(value);
System.out.println("after: "+value);
```

Trace d'exécution : ?

```
before: 3
inside: 5
after: 3
```

Passage par valeur/référence

Au sein d'une classe `PassByCopy` :

```
void methodWithItem(Item i) {
  i = new Item(5,"anotherBook");
  System.out.println("inside: "+i);
}
```

Le programme principal est le suivant :

```
PassByCopy test = new PassByCopy();
Item value = new Item(2,"aBook");
System.out.println("before: "+value);
test.methodWithItem(value);
System.out.println("after: "+value);
```

Trace d'exécution : ?

```
before: aBook: 2.0
inside: anotherBook: 5.0
after: aBook: 2.0}
```


Passage par valeur/référence

Au sein d'une classe `PassByCopy` :

```
void changeItemPrice(Item i) {
  i.setPrice(5.0);
  System.out.println("inside: "+i);
}
```

Le programme principal est le suivant :

```
PassByCopy test = new PassByCopy();
Item value = new Item(3.0,"aBook");
System.out.println("before: "+value);
test.changeItemPrice(value);
System.out.println("after: "+value);
```

Trace d'exécution : ?

```
before: aBook: 3.0
inside: aBook: 5.0
after: aBook: 5.0
```


Plan

Auto-référence

En java, **`this`** référence l'objet qui invoque la méthode. Il est toujours défini dans le contexte d'exécution d'une méthode.

Il peut être utilisé pour lever des ambiguïtés. Sans le `this`, c'est le paramètre qui serait modifié.

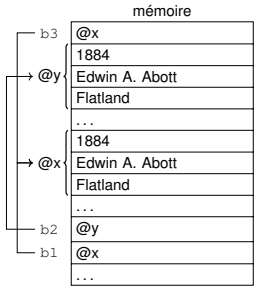
```
public Book(String author, String title, int publicationYear, String text) {
  this.author = author;
  this.title = title;
  this.publicationYear = publicationYear;
  this.text = text;
}
```

Le **`this`** peut être utile pour retourner sa propre référence dans une méthode. Il se manipule comme n'importe quelle autre référence.

```
Book theThickest(Book aBook) {
      if(this.text.length() > aBook.text.length()) return this;
      else return aBook;
}
```

Comparaisons : == et `equals`

```
Book b1 = new Book("Flatland", "Edwin A. Abbott", 1884);
Book b2 = new Book("Flatland", "Edwin A. Abbott", 1884);
Book b3 = b1;
```



== compare les références et non les attributs d'objets.

`if(b1 == b2) ...` ⇒ `false`
`if(b1 == b3) ...` ⇒ `true`

`equals()` compare les valeurs des attributs et doit être défini pour chaque classe.

`b1.equals(b2) ...` ⇒ `true`
`b2.equals(b3) ...` ⇒ `true`

Implémentation d'une méthode `equals`

Une méthode **`equals`** est toujours invocable. Si elle n'est pas définie, == sera invoqué à la place.

On peut redéfinir cette méthode basée sur des champs choisis.

Cette méthode doit respecter la structure suivante :

```
public boolean equals(Book other) {
  if (this == other) return true;
  if (other == null) return false;
  // primitive attribute
  if (this.publicationYear != other.publicationYear) return false;
  // object attribute
  if (this.author == null) {
    if (other.author != null) return false;
  } else if (!this.author.equals(other.author)) return false;
  // on more attributes if required
  // ...
  return true;
}
```


Les objets *immutable*

Un objet **immutable** est un objet qui **ne peut plus** être modifié après instanciation.

Vous en manipulez déjà : `String`...

Démonstration :

```
public String immutable() {
  String s = "";
  for(int i=0; i<10; i++) {
    s+=i;
  }
  return s;
}
```

Il en existe des versions **mutables** : `StringBuffer` et `StringBuilder`.

```
public String mutable() {
  StringBuilder sb = new StringBuilder();
  for(int i=0; i<20; i++)
    sb.append(i);
  return sb.toString();
}
```


Plan

Manipulation des attributs

On souhaite créer un objet `Employee`, caractérisé par un nom et un salaire.

Employee
name : String wages : double
Employee(String, double) payRise(double) : void toString() : String

On souhaite ne pouvoir modifier le salaire que si le nouveau montant est plus important que l'ancien.

Manipulation des attributs

Le code est facile à écrire...

```
class Employee {
  String name;
  double wages;

  Employee(String name, double wages) {
    this.name = name;
    this.wages = wages;
  }
  void payRise(double newWages) {
    if(newWages >= this.wages) {
      this.wages = newWages;
    }
  }
  public String toString() {
    return this.name + "=" + this.wages;
  }
}
```


Manipulation des attributs

Le code est également facile à utiliser **depuis une autre classe**...

```
public class UseEmployee {
  public static void main(String[] args) {
    Employee bruno = new Employee("Bruno", 4000.0);
    System.out.println(bruno);
    bruno.payRise(4200.0);
    System.out.println(bruno);
    bruno.payRise(42.0);
    System.out.println(bruno);
```

On obtient logiquement :

```
Bruno=4000.0
Bruno=4200.0
Bruno=4200.0
```


Manipulation des attributs

Pas besoin d'être un pirate informatique pour contourner cette "*sécurité* "...

```
public class UseEmployee {
  public static void main(String[] args) {
    Employee bruno = new Employee("Bruno", 4000.0);
    System.out.println(bruno);
    bruno.payRise(4200.0);
    System.out.println(bruno);
    bruno.payRise(42.0);
    System.out.println(bruno);
    bruno.wages = 42.0;
    System.out.println(bruno);
  }
}
```

On obtient :

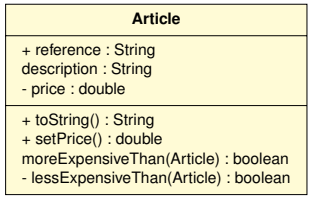
```
Bruno=4000.0
Bruno=4200.0
Bruno=4200.0
Bruno=42.0
```

Contrôle d'accès

L'objectif est de **restreindre la visibilité** des attributs ou des méthodes d'une classe.

En Java, les **restrictions d'accès** sont précisés via un mot-clé lors de la définition d'attributs, de méthodes ou de classes.

- `private` accessible uniquement aux instances de la classe : **-**
- `default` accessible uniquement depuis le même paquetage : ∼ ou **rien**
- `public` accessible pour tout le monde : **+**



Contrôle d'accès

Un exemple

Considérons la méthode principale suivante :

```
public static void main(String[] args) {
  Article a1 = new Article("BQ45A21", "chair", 75.25);
  a1.price = 25.0;
  a1.setPrice(25.0);
  System.out.println(a1);
  Article a2 = new Article("TQD124A", "table", 125.00);
  System.out.println(a1.lessExpensiveThan(a2));
  System.out.println(a1.moreExpensiveThan(a2));
}
```

Selon la classe où est définie cette méthode, quelles sont les instructions incorrectes ?

	$\in$ class	$\notin$ class, $\in$ pkg	$\notin$ pkg
ligne 3	✓	×	×
ligne 4	✓	✓	✓
ligne 7	✓	×	×
ligne 8	✓	✓	×

Plan

Visibilité, accesseurs et mutateurs

L'**interface publique** d'une classe est la liste des attributs et des méthodes accessibles à tous depuis l'extérieur de la classe.

On peut jouer sur la visibilité des attributs en déclarant l'attribut `private` et définissant des méthodes d'accès **si nécessaire** permettant la lecture ou la modification de ces attributs.

Les méthodes d'accès en lecture sont appelées *accesseur* (*getter*) alors que les méthodes d'accès en écriture sont appelées *mutateur* (*setter*).

Pour un attribut `reference` de type `String` :

- `String getReference()` : accesseur
- `void setReference(String s)` : mutateur

Visibilité des attributs

Par exemple, si la classe `Article` est déclarée de la manière suivante :

```
public class Article {
  public String reference;
  public String description;
  public double price;
```

Et que le code manipulant un article à partir d'une autre classe soit :

```
Article a1 = new Article("BQ45A21", "chair", 75.25);
System.out.println("This article is worth : " + a1.price);
```

Modifier la structure ou le comportement (prix TTC par exemple) ⇒ adaptation de chaque classe où l'on utilise directement l'attribut.

Visibilité des attributs

En revanche, si la classe `Article` est déclarée de la manière suivante :

```
public class Article {
  private String reference;
  private String description;
  private double price;

  public double getPrice() {
    return this.price;
  }
```

Et que le code manipulant un article à partir d'une autre classe soit :

```
Article a1 = new Article("BQ45A21", "chair", 75.25);
System.out.println("This article is worth : " + a1.getPrice());
```

La modification de la classe `Article` elle-même reste possible sans pour autant nécessiter d'adaptation ailleurs.

Visibilité des attributs

La modification pour travailler uniquement avec des prix TTC n'impacte aucune autre classe.

```
public class Article {
    private String reference;
    private String description;
    private double tva = 19.6;
    private double price;

    public double getPrice() {
        return this.price + (this.tva * this.price);
    }
```

Et que le code manipulant un article à partir d'une autre classe n'a pas besoin d'être modifié et reste fonctionnel.

```
Article a1 = new Article("BQ45A21", "chair", 75.25);
System.out.println("This article is worth : " + a1.getPrice());
```


Visibilité des méthodes

Le contrôle d'accès aux méthodes permet la définition de **l'interface publique** de l'objet proprement dit : l'ensemble des méthodes utilisables pour le manipuler.

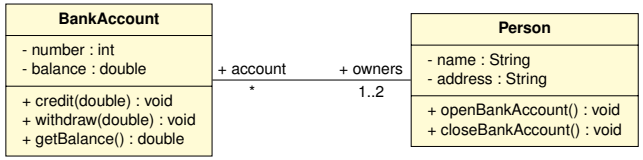
Le contrôle d'accès aux méthodes permet la décomposition des traitements sans pour autant changer l'interface publique de la classe.

```
  private boolean lessExpensiveThan(Article other) {
    return this.price < other.price;
  }
  public boolean theCheapest(Article[] toCompare) {
    for(int i=0; i<toCompare.length; i++) {
      if(! this.lessExpensiveThan(toCompare[i])) return false;
    }
    return true;
  }
```

Seule la méthode `theCheapest()` apparaît dans l'interface publique.

Notations UML

Les associations d'un **diagramme de classes** doivent être interprétées pour savoir quoi implémenter : le **diagramme d'objets**.



Un compte a 1 ou 2 propriétaires qui sont des personnes. Une personne peut avoir aucun ou plusieurs comptes bancaires.

En termes d'implémentation, on observe que :

- `owners` est un attribut par association de la classe `BankAccount` de type `Person` : un compte bancaire peut avoir un ou deux propriétaires.
- `accounts` est un attribut de la classe `Person` permettant de regrouper des objets de type `BankAccount` (comme un tableau par exemple).

⇒ **Il faut choisir une représentation cohérente avec les cardinalités**.

Multiplicité d'une association

Une association est caractérisée par une ou deux multiplicités et un ou deux arguments précisant les attributs concernés, selon la navigabilité de l'association.

Multiplicités possibles :

- **1** : exactement un, la multiplicité par défaut
- **0..1** : l'attribut est optionnel
- **1..*** : au moins un
- ***** : 0 ou plus
- **n..m** : entre n et m éléments

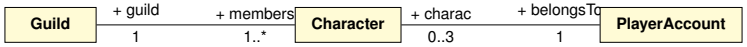
Navigabilité d'une association

Les associations ne sont pas forcément bidirectionnelles.

En termes d'implémentation :

- une salle a accès aux films qui y sont programmés, mais pas l'inverse ;
- on retrouve un attribut `prog` qui permet le regroupement de `Film` au sein de la classe `Theater` ;
- aucun attribut n'est ajouté à la classe `Film` : à partir d'un film, il n'est pas possible de retrouver dans quelles salles il est projeté.

Un exemple complet



Comment interprétez-vous ce diagramme de classes ? Quelles caractéristiques ces objets doivent au moins avoir ?

- la classe `Guild` est caractérisé par plusieurs `members` de type `Character` ;
- un personnage ne peut appartenir qu'à une seule guilde ;
- un compte de joueur peut contenir jusqu'à 3 personnages ;
- un personnage ne peut appartenir qu'à un seul compte de joueur.