Chapitre 4: Classes

Présentation pour

Java Software Solutions

Foundations of Program Design

Deuxième Edition

par John Lewis et William Loftus

Java Software Solutions est publié par Addison-Wesley

Presentation slides are copyright 2000 by John Lewis and William Loftus. All rights reserved.
Instructors using the textbook may use and modify these slides for pedagogical purposes.

Écrire des classes

- On a utilisé des classes prédéfinies. On apprendra à écrire nos propres classes pour définir de nouveaux objets
- Chapitre 4 se concentre sur :
  - Déclarations de classes
  - Déclarations de méthodes
  - Variables instanciées
  - Encapsulation
  - Overloading de méthodes
  - Objets graphiques

Objets

- Un objet possède :
  - *état* - caractéristiques qui le décrivent
  - *comportement* - ce qu'il peut faire (ou ce qu'on peut lui faire)
- Par exemple, soit une pièce de monnaie qui peut être "pile" ou "face"
- L'état de la pièce est son côté courant (pile ou face)
- Le comportement de la pièce est qu'elle peut être lancée (jouer à pile ou face)
- Ici le comportement de la pièce peut changer son état

Classes

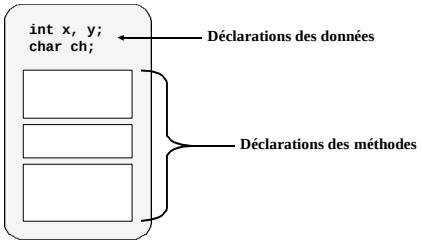
- Une *classe* est comme un "moule" pour des objets
- C'est le modèle d'après lequel les objets sont créés
- Par exemple, la classe `String` est utilisée pour définir des objets de type `String`
- Chaque objet `String` contient des caractères spécifiques (son état)
- Chaque objet `String` peut fournir des services (comportements) tel `toUpperCase`

Classes

- La classe `String` est fournie par la librairie standard de classes de Java
- On peut aussi écrire nos propres classes qui définiront des objets spécifiques selon nos besoins
- Par exemple, supposons qu'on désire écrire un programme pour simuler un jeu de pile ou face
- On peut écrire une classe `Coin` pour représenter un objet `coin`

Classes

- Une classe contient des déclarations de données et des déclarations de méthodes



Portée des données

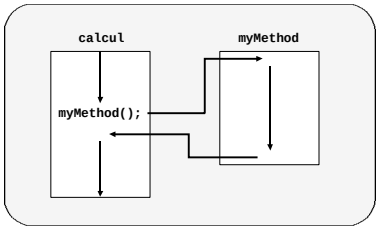
- La *portée* des données est la partie d'un programme dans laquelle ces données peuvent être utilisées (référencées)
- Les données déclarées au niveau de la classe peuvent être utilisées par toutes les méthodes dans cette classe
- Les données déclarées à l'intérieur d'une méthode peuvent seulement être utilisées dans cette méthode
- Les données déclarées à l'intérieur d'une méthode sont appelées des *données locales*

Méthodes

- Une *déclaration de méthode* spécifie le code qui sera exécuté lorsque la méthode est appelée
- Lorsqu'une méthode est appelée, le contrôle de flux se rend à la méthode et exécute son code
- Lorsque terminé, le flux retourne à l'endroit d'où la méthode a été appelée, et continue
- L'appel peut retourner une valeur ou non, dépendant de la définition de la méthode

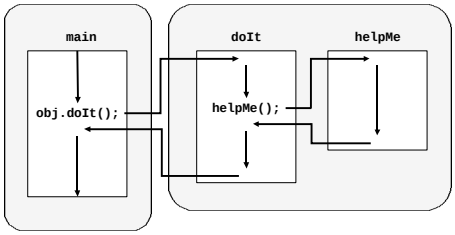
Contrôle de flux des méthodes

- La méthode appelée peut être dans la même classe. Dans ce cas, seulement le nom de la méthode est nécessaire



Contrôle de flux des méthodes

- La méthode appelée peut faire partie d'une autre classe ou d'un autre objet



La classe Coin

- Dans notre classe `Coin` on peut définir les données suivantes :
  - `face`, un entier qui représente la face courante
  - `FACE` et `PILE`, des constantes entières qui représentent les deux états possibles
- On peut définir les méthodes suivantes :
  - Un constructeur `Coin`, pour initialiser l'objet
  - Une méthode `flip`, pour lancer la pièce
  - Une méthode `getFace`, pour retourner la face courante
  - Une méthode `toString`, pour retourner une chaîne de caractères à imprimer

La classe Coin

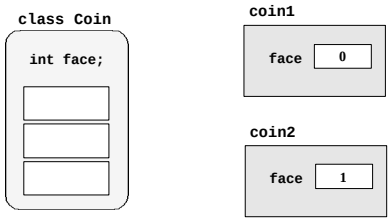
- Voir CountFlips.java (page 179)
- Voir Coin.java (page 180)
- Une fois la classe `Coin` définie, on peut la réutiliser dans d'autres programmes
- Notez que le programme `CountFlips` n'utilisait pas la méthode `toString`
- Un programme n'utilisera pas nécessairement tous les services offerts par un objet

Données instanciées

- La variable `face` dans la classe `Coin` est appelée une donnée instanciée parce que chaque *instance* (objet) de la classe `Coin` a sa propre variable
- Une classe déclare le type des données, mais elle ne réserve pas d'espace mémoire pour elles
- Chaque fois qu'un objet `Coin` est créé, une nouvelle variable `face` est aussi créée
- Les objets d'une classe partagent les définitions des méthodes, mais ils ont leur espace unique pour les données
- C'est la seule façon pour que deux objets peuvent avoir des états différents

Données instanciées

- Voir FlipRace.java (page 182)



Encapsulation

- Un objet peut être vu de deux façons :
  - interne - ses données et les algorithmes utilisés par ses méthodes
  - externe - l'interaction de l'objet avec les autres objets dans le programme
- Du point de vue externe, l'objet est une entité *encapsulée*, fournissant un ensemble de services spécifiques
- Ces services définissent *l'interface* à l'objet
- Rappel: au Chapitre 2, on a défini un objet comme une abstraction, qui cache les détails du reste du système

15

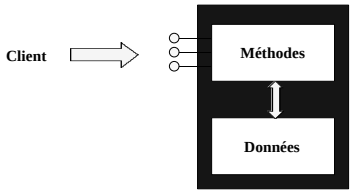
Encapsulation

- Un objet devrait être *self-governing*
- Tout changement à l'état de l'objet (ses variables) devrait être fait par les méthodes de cet objet
- On devrait rendre difficile, et même impossible, pour un objet de modifier l'état d'un autre objet
- L'usager, ou *client*, d'un objet peut appeler ses services, mais il ne devrait pas être au courant de comment ces services sont réalisés

16

Encapsulation

- Un objet encapsulé peut être interprété comme une *boîte noire*
- Son fonctionnement interne est caché au client, qui ne peut accéder qu'aux méthodes par leur interface



17

Modificateurs de visibilité

- En Java, l'encapsulation est réalisée par l'utilisation appropriée des *modificateurs de visibilité*
- Un *modificateur* est un mot réservé en Java qui spécifie les caractéristiques particulières d'une méthode ou d'une donnée
- On a déjà utilisé le modificateur `final` pour définir une constante
- Java a trois modificateurs de visibilité : `public`, `private`, et `protected`
- On discutera plus tard du modificateur `protected`

18

Modificateurs de visibilité

- **Les membres d'une classe déclarés de *visibilité publique* peuvent être accédés de n'importe où**
- **Les membres déclarés de *visibilité privée* peuvent seulement être accédés de l'intérieur de la classe**
- **Les membres déclarés sans modificateur de visibilité ont la *visibilité de défaut* et peuvent être accédés par toute classe dans le même package**
- **Les modificateurs en Java sont discutés en détail à l'Appendice F**

19

Modificateurs de visibilité

- **Comme règle générale, aucune donnée d'un objet ne devrait être déclarée de visibilité publique**
- **Les méthodes qui fournissent les services de l'objet sont habituellement déclarées de visibilité publique pour qu'elles puissent être appelées par les clients**
- **Les méthodes publiques sont aussi appelées des *méthodes de service***
- **Une méthode créée simplement pour aider une méthode de service est appelée une *méthode de support***
- **Puisqu'une méthode de support n'est pas supposée être appelée par un client, elle ne devrait pas être déclarée de visibilité publique**

20

Les déclarations de méthodes

- **Une déclaration de méthode commence avec un *entête* de méthode**

```
char calc (int num1, int num2, String message)
```

nom de méthode

liste de paramètres

type de retour

La liste de paramètres spécifie le type et le nom de chaque paramètre

Le nom d'un paramètre dans la déclaration de la méthode est appelé un *argument formel*

Les déclarations de méthodes

- **L'entête d'une méthode est suivi par le *corps de la méthode***

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

L'expression retournée doit être compatible avec le type de retour

`sum` et `result` sont des *données locales*

Elles sont créées chaque fois que la méthode est appelée, et sont détruites lorsque la méthode termine son exécution

L'énoncé de retour

- **Le *type de retour* d'une méthode indique le type de la valeur que la méthode retourne à l'appelant**
- **Une méthode qui ne retourne pas une valeur a un type de retour `void`**
- ***L'énoncé de retour* `return` spécifie la valeur qui sera retournée**
- **Son expression doit être conforme avec le type de retour**

23

Paramètres

- **Chaque fois qu'une méthode est appelée, les *arguments actuels* dans l'appel sont *copiés* dans les *arguments formels***

```
ch = obj.calc (25, count, "Hello");

char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

Les constructeurs

- **Rappel: un constructeur est une méthode spéciale utilisée pour initialiser un objet nouvellement créé**
- **Un constructeur :**
  - **A le même nom que sa classe**
  - **Ne retourne pas de valeur**
  - **N'a pas de type de retour, même pas `void`**
  - **Souvent initialise les valeurs des variables d'instance**
- **Le programmeur n'est pas obligé de définir un constructeur pour une classe**

25

Ecrire des classes

- **Voir BankAccounts.java (page 188)**
- **Voir Account.java (page 189)**
- **Un objet composé *(aggregate)* est un objet qui contient des références à d'autres objets**
- **Un objet `Account` est un objet composé parce qu'il contient une référence à un objet `String` (qui contient le nom du propriétaire)**
- **Un objet composé représente une relation *a-un***
- **Un compte bancaire *a un* nom**

Ecrire des classes

- **Parfois un objet doit interagir avec d'autres objets du même type**
- **Par exemple, on peut additionner deux objets nombres rationnels (`Rational`) comme suit :**

```
r3 = r1.add(r2);
```

- **Un objet (`r1`) exécute la méthode et l'autre objet (`r2`) est passé en paramètre**
- **Voir RationalNumbers.java (page 196)**
- **Voir Rational.java (page 197)**

Surcharger une méthode

- ***Surcharger une méthode (method overloading)* utilise le même nom de méthode pour plusieurs méthodes**
- **La *signature* de chaque méthode surchargée doit être unique**
- **La signature inclut le nombre, type et ordre des paramètres**
- **Le compilateur doit être capable de déterminer quelle version de la méthode est appelée en analysant les paramètres**
- **Le type de retour de la méthode *ne fait pas partie* de la signature**

28

Surcharger une méthode

Version 1

```
float tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Sélectionne

```
result = tryMe (25, 4.32)
```

Surcharger une méthode

- **La méthode `println` est surchargée :**

```
println (String s)
println (int i)
println (double d)
      etc.
```

- **Les lignes suivantes appellent différentes versions de la méthode `println` :**

```
System.out.println ("The total is:");
System.out.println (total);
```

30

Surcharger une méthode

- **Les constructeurs peuvent être surchargés**
- **Un constructeur surchargés permet plusieurs façons d'initialiser un nouvel objet**
- **Voir SnakeEyes.java (page 203)**
- **Voir Die.java (page 204)**

31

La classe StringTokenizer

- **L'exemple suivant utilise la classe `StringTokenizer`, définie dans le package `java.util`**
- **Un objet `StringTokenizer` sépare un chaîne de caractères en des sous-chaînes plus petites (*tokens*)**
- **Par défaut, le tokenizer sépare la chaîne aux espaces blancs**
- **Le constructeur `StringTokenizer` prend en paramètre la chaîne originale à séparer**
- **Chaque appel à la méthode `nextToken` retourne le prochain élément de la chaîne**

Décomposition de méthodes

- **Une méthode devrait être relativement petite, pour être facilement compréhensible comme une seule entité**
- **Une méthode plus grosse devrait être décomposée en plusieurs méthodes plus petites nécessaires pour la compréhension**
- **Ainsi une méthode de service d'un objet peut appeler une ou plusieurs méthodes de support pour accomplir son but**
- **Voir PigLatin.java (page 207)**
- **Voir PigLatinTranslator.java (page 208)**

Les méthodes Applet

- **Dans les exemples précédents, on a utilisé la méthode `paint` de la classe `Applet` pour dessiner dans un applet**
- **La classe `Applet` a plusieurs méthodes qui sont appelées automatiquement à certains moments durant l'activité d'un applet**
- **La méthode `init` est exécutée seulement une fois lorsque l'applet est chargé**
- **La classe `Applet` contient aussi d'autres méthodes qui aident généralement au traitement dans l'applet**

Objets graphiques

- **Tout objet qu'on définit en écrivant une classe peut avoir des éléments graphiques**
- **L'objet doit simplement obtenir un contexte graphique (un objet `Graphics`) dans lequel dessiner**
- **Un applet peut passer son contexte graphique à un autre objet comme pour tout autre paramètre**
- **Voir LineUp.java (page 212)**
- **Voir StickFigure.java (page 215)**