Programmation Objet

Florian Boudin

Révision 3 du 25 juillet 2014

Classe 3 : *Classes et objets*

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

Implémentation de Bicycle

```
public class Bicycle {
  public int cadence; // the Bicycle class has
  public int gear; // three fields
  public int speed;

  public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence; // the Bicycle class
    speed = startSpeed; // has one constructor
  }

  public void setCadence(int newValue) { // the Bicycle class
    cadence = newValue; // has four methods
  }

  public void setGear(int newValue) {
    gear = newValue;
  }

  public void applyBrake(int decrement) {
    speed -= decrement;
  }

  public void speedUp(int increment) {
    speed += increment;
  }
}
```

Implémentation de MountainBike

```
public class MountainBike extends Bicycle {
  // the MountainBike subclass has one field
  public int seatHeight;

  // the MountainBike subclass has one constructor
  public MountainBike(int startHeight, int startCadence,
                      int startSpeed, int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
  }

  // the MountainBike subclass has one method
  public void setHeight(int newValue) {
    seatHeight = newValue;
  }
}
```

Déclaration d'une classe

- déclaration de la classe MyClass ...

```
class MyClass {
  // field, constructor, and
  // method declarations
}
```

- ... sous-classe de MySuperClass et qui implémente YourInterface

```
class MyClass extends MySuperClass implements YourInterface {
  // field, constructor, and
  // method declarations
}
```

Déclaration d'une classe

- une déclaration de classe peut inclure, dans l'ordre :
  1. les modificateurs tels que `public`, `private`, etc.
  2. le nom de la classe, avec la première lettre en majuscule
  3. le nom de la super-classe précédé par le mot-clé `extends`
  4. une liste des interfaces implémentées, séparée par des virgules, précédée par le mot-clé `implements`
  5. le corps de la classe, entouré par des accolades `{ }`

Déclaration des membres de la classe

- plusieurs types de variables :
  - variables membres de la classe (champs)
  - variables de blocs ou méthodes (variables locales)
  - variables dans les déclarations de méthodes (paramètres)
- `Bicycle` utilise les lignes de code suivantes :

```
public int cadence;
public int gear;
public int speed;
```

Déclaration des membres de la classe

- déclaration d'un champ :
  - `(public|private|...) type nom;`
- `public` : toutes les classes ont accès au champ
- `private` : le champ n'est accessible que dans la classe où il est déclaré
- *encapsulation* → champs privés
  - comment accéder quand-même aux valeurs ?
  - méthodes publiques pour accéder aux valeurs des champs

Déclaration des membres de la classe

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }
...
```

Définition des méthodes

- exemple de méthode

```
public double product( double length, int factor ) {
  // do the calculation here
}
```

- la *signature* d'une méthode

```
product(double, int)
```

Définition des méthodes

- une déclaration de méthode contient, dans l'ordre :
  1. les modificateurs tels que `public`, `private`, etc.
  2. le type de retour, type de données ou `void`
  3. le nom de la méthode
  4. la liste de paramètres entre parenthèses, délimités par des virgules
  5. le corps de la méthode entre accolades `{ }`

Surcharger des méthodes

- il est possible de *surcharger* des méthodes

```
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) {
    ...
  }
}
```

Surcharger des méthodes

- différentier les méthodes grace aux signatures
  - vous ne pouvez pas déclarer plus d'une méthode avec les mêmes paramètres
  - la surcharge de méthodes rends le code *moins lisible*

Retourner une valeur

- une méthode retourne au code qui l'a invoquée lorsque :
  - toutes les déclarations ont été exécutées
  - une instruction `return` est atteinte
  - une exception est lancée (traité plus tard)
- la déclaration contient le type de la valeur de retour
  - une méthode déclarée avec `void` ne retourne rien
  - le mot-clé `return` est utilisé pour spécifier la valeur de retour

Retourner une valeur

```java
public int multiply(int a, int b) {
  int m = a * b;
  return m;
}

public void multiPrint(String m, int n) {
  for (int i = 0; i < n; i++) {
    System.out.println(m);
  }
}
```

Les constructeurs

- une classe contient des *constructeurs* qui sont invoqués pour créer des objets
- les déclarations de constructeurs ressemblent aux déclarations de méthodes, sauf qu'elles utilisent le nom de la classe et n'ont pas de type de retour

```
public Bicycle( int startCadence, int startSpeed, int startGear ) {
  gear = startGear;
  cadence = startCadence;
  speed = startSpeed;
}
```

Les constructeurs

- créer un nouvel objet, un constructeur est appelé avec l'opérateur `new`

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

- il est possible de créer plusieurs constructeurs
- les constructeurs sont différentiés par leurs signatures

Les constructeurs

```
public Bicycle( int startCadence, int startSpeed, int startGear ) {
  gear = startGear;
  cadence = startCadence;
  speed = startSpeed;
}

public Bicycle() {
  gear = 1;
  cadence = 10;
  speed = 0;
}
```

- créer un objet avec le constructeur sans arguments

```
Bicycle myBike2 = new Bicycle();
```

Passage de paramètres

- la déclaration d'une méthode ou d'un constructeur déclare le nombre et le type d'arguments
- par exemple, la fonction ci-dessous calcule l'aire d'un losange en se basant sur la longueur de la petite diagonale et celle de la grande diagonale

```
public double aireLosange( double pDiag, double gDiag ) {
  double aire = (pDiag * gDiag) / 2.0;
  return aire;
}
```

Passage de paramètres

- le type des paramètres doit être spécifié :
  - types primitifs, e.g. `int`, `double`
  - tableaux, e.g. `int[]`
  - objets, e.g. `Bicycle[] bikes`
- la structure `varargs` permet de passer un nombre variable de paramètres aux méthodes
- `methode(type... nomParametre)`
  - raccourci pour créer un tableau de type

Passage de paramètres

```
public static void hello(String... names) {

  // names.length contient la taille du
  // tableau de String

  for (String n : names) {
    System.out.println("Hello "+n+".");
  }
}

hello("Paul", "Sue");

// Hello Paul.
// Hello Sue.
```

Passage de paramètres

- le nom d'un paramètre est utilisé dans le corps de la méthode est doit être unique dans sa portée

```
public double aireLosange( double pDiag, double gDiag ) {

  double pDiag = 0.1; // erreur
  ...
}
```

Passage de paramètres

- un paramètre peut avoir le même nom qu'un champ de la classe, il masque alors le champ

```
public class Circle {

  private int x, y, radius;

  public void setOrigin(int x, int y) {
  // x et y contiennent les
  // valeurs des parametres
  }
}
```

Passage de paramètres

- paramètres (types primitifs) passés par *valeurs*
  - toute modification de la valeur d'un paramètre n'existe que dans le cadre de la méthode

```
public class PassPrimitiveByValue {
  public static void main(String[] args) {
    int x = 3;
    passMethod(x);
    System.out.println("x = " + x); // x = 3
  }

  public static void passMethod(int p) {
    p = 10;
  }
}
```

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

Objets

- une application Java créée de nombreux objets qui interagissent en invoquant des méthodes
- grâce à ces interactions, une application peut effectuer diverses tâches comme implémenter une GUI ou envoyer des informations sur un réseau
- une fois qu'un objet a terminé le travail pour lequel il a été créé, ces ressources sont recyclées pour être utilisées par d'autres objets

CreateObjectDemo

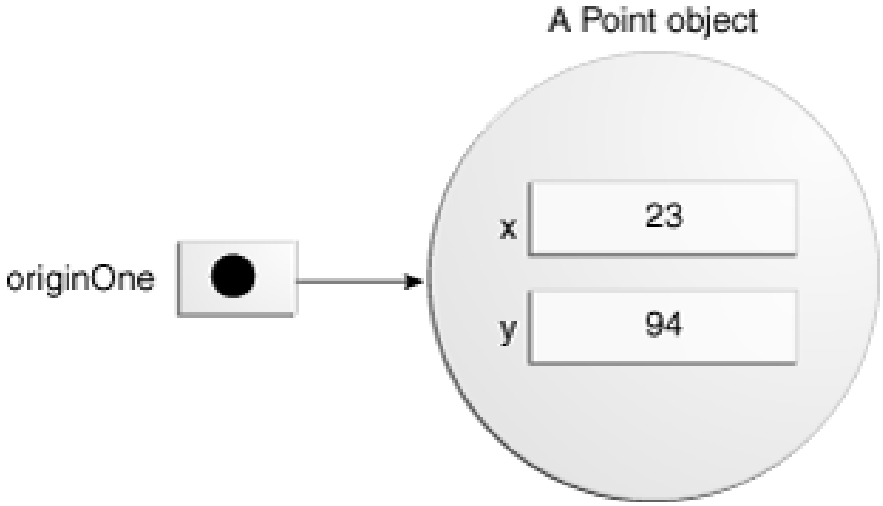
```
public class CreateObjectDemo {
  public static void main(String[] args) {
    // Declare and create a point object and two rectangle objects
    Point originOne = new Point(23, 94);
    Rectangle rectOne = new Rectangle(originOne, 100, 200);
    Rectangle rectTwo = new Rectangle(50, 100);

    // display rectOne's width, height, and area
    System.out.println("Width of rectOne: "+rectOne.width);
    System.out.println("Height of rectOne: "+rectOne.height);
    System.out.println("Area of rectOne: "+rectOne.getArea());

    // set rectTwo's position
    rectTwo.origin = originOne;

    // display rectTwo's position
    System.out.println("X Position of rectTwo: "+rectTwo.origin.x);
    System.out.println("Y Position of rectTwo: "+rectTwo.origin.y);

    // move rectTwo and display its new position
    rectTwo.move(40, 72);
    System.out.println("X Position of rectTwo: "+rectTwo.origin.x);
    System.out.println("Y Position of rectTwo: "+rectTwo.origin.y);
  }
}
```

Créer des objets

- exemple : la classe `Point`

```
public class Point {
  public int x = 0;
  public int y = 0;

  // constructor
  public Point(int a, int b) {
    x = a;
    y = b;
  }
}
```

Créer des objets



- création d'un objet Point

```
Point originOne = new Point(23, 94);
```

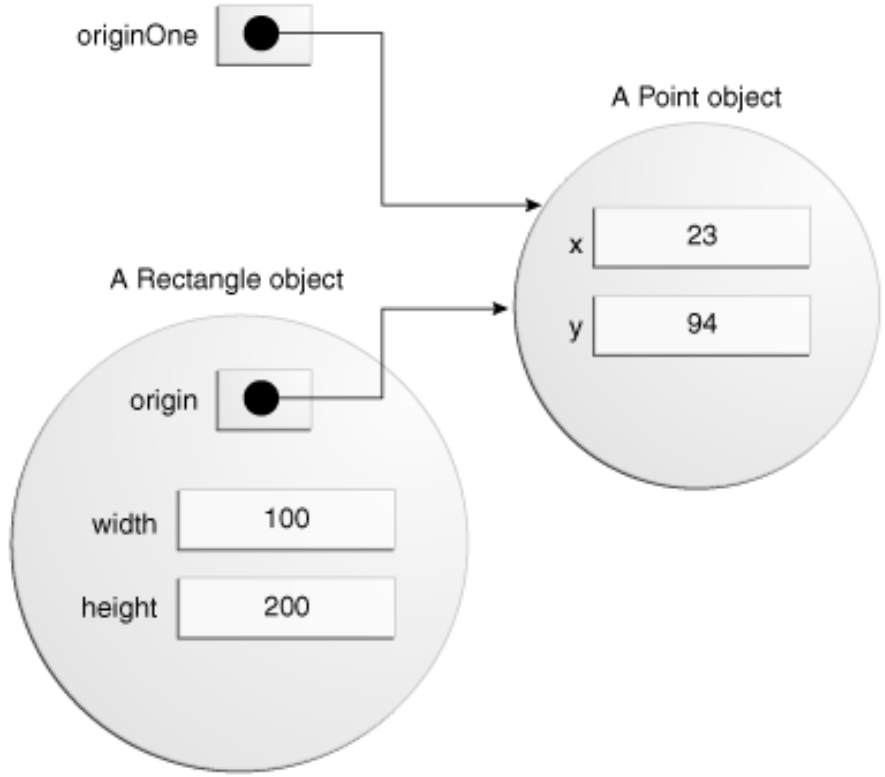
Constructeurs

```
public class Rectangle {

  public int width = 0;
  public int height = 0;
  public Point origin;

  public Rectangle() { // constructor 1
    origin = new Point(0, 0);
  }

  public Rectangle(Point p) { // constructor 2
    origin = p;
  }

  public Rectangle(int w, int h) { // constructor 3
    origin = new Point(0, 0);
    width = w;
    height = h;
  }

  public Rectangle(Point p, int w, int h) { // constructor 4
    origin = p;
    width = w;
    height = h;
  }
...
```

Créer des objets

- création d'un rectangle à partir de `originOne`

```
Rectangle rect = new Rectangle( originOne, 100, 200 );
```



Créer des objets

- création d'un second rectangle

```
Rectangle rectTwo = new Rectangle(50, 100);
// creates a new Point object whose x and y
// values are initialized to 0
```

- création d'un rectangle avec le constructeur vide

```
Rectangle rect = new Rectangle();
```

- toutes les classes doivent avoir un constructeur
  - par défaut, le compilateur fournit un constructeur sans arguments

Accéder aux variables membres

- *intérieur de la classe* : utiliser leurs noms

```
public class Rectangle {

  public int width = 0;
  public int height = 0;

  public printSize() {
    System.out.println("Height: "+height);
    System.out.println("Width: "+width);
  }
}
```

Accéder aux variables membres

- *extérieur de la classe* : utiliser l'operateur dot (.)
  - ne fonctionne que si les variables sont publiques

```
Rectangle rect = new Rectangle( originOne, 100, 200 );
System.out.println("Width: "+rect.width);
System.out.println("Height: "+rect.height);
```

Appeler les méthodes d'un objet

```
public class Rectangle {

  public int width = 0;
  public int height = 0;

  ...

  public int getArea() {
    return width * height;
  }
}
```

- appeler la méthode `getArea()`

```
Rectangle rect = new Rectangle();
System.out.println("Area: "+rect.getArea());
```

Le ramasse-miettes (*garbage collector*)

- pour certains langages, il est nécessaire de garder une trace de tous les objets créés et de les détruire explicitement lorsqu'ils ne sont plus utilisés
  - gérer la mémoire est fastidieux et source d'erreurs
- la plateforme Java permet de créer autant d'objets que nécessaire sans se soucier de les détruire
- l'environnement d'exécution Java supprime les objets quand il détermine qu'ils ne sont plus utilisés
- affecter la valeur `null` pour supprimer un objet

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

Le mot-clé `this`

- utilisé pour accéder aux champs masqués par les paramètres d'une méthode / constructeur

```
public class Point {

  public int x = 0;
  public int y = 0;

  public Point(int a, int b) {
    x = a;
    y = b;
  }
}
```

```
public class Point {

  public int x = 0;
  public int y = 0;

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

Le mot-clé this (cont.)

- utilisé pour appeler un constructeur de la classe

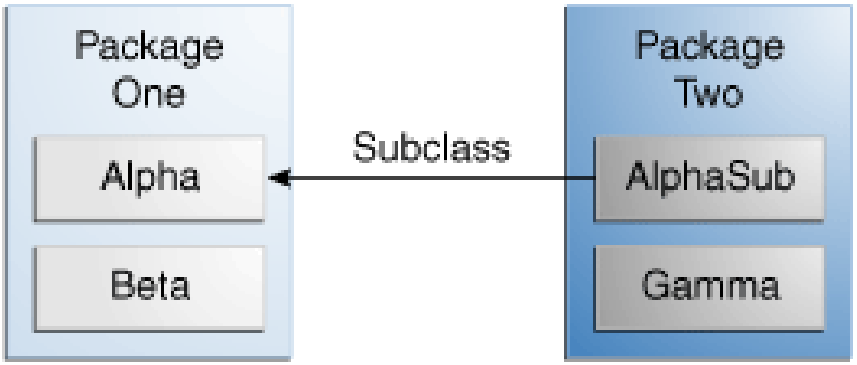
```
public class Rectangle {

  private int x, y;
  private int width, height;

  public Rectangle() {
    this(0, 0, 0, 0);
  }

  public Rectangle(int width, int height) {
    this(0, 0, width, height);
  }

  public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }
...
```

Les modificateurs d'accès

- les modificateurs d'accès déterminent si les autres classes peuvent utiliser une méthode ou un champ
- deux niveaux de contrôle d'accès :
  - au niveau de la classe, `public` ou sans modificateur
  - au niveau des membres, `public`, `private`, `protected`, sans modificateur

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`no modifier`	Y	Y	N	N
`private`	Y	N	N	N

Les modificateurs d'accès (cont.)



- le tableau montre où les membres de alpha sont visibles pour chaque modificateur

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`no modifier`	Y	Y	N	N
`private`	Y	N	N	N

Les modificateurs d'accès (cont.)

```
// les gens peuvent utiliser cette classe
public class Laundromat {

  // les gens ne peuvent pas utiliser
  // ce champ interne,
  private Laundry[] dirty;

  // mais ils peuvent utiliser ces
  // methodes publiques afin de manipuler
  // le champ interne
  public void wash() {...}
  public void dry() {...}

  // une sous-classe peut modifier ce champ
  protected int temperature;
}
```

Les variables de classe

- quand un certain nombre d'objets sont créés à partir d'une même classe, ils ont chacun leurs propres copies distinctes des variables d'instance
- dans le cas de la classe `Bicycle`, les variables d'instance sont `cadence`, `gear` et `speed`
  - chaque instance de `Bicycle` a ses propres valeurs pour ces variables, stockées dans des emplacements mémoire différents

Les variables de classe (cont.)

- le modificateur `static` permet de créer des variables partagées par tous les objets
- pour accéder à la variable `var` d'un objet `myObject`, instance la classe `MyClass`
  - `MyClass.var` (recommandé)
  - `myObject.var`

Exemple de variable de classe

- on veut créer plusieurs objets `Bicycle` et leur assigner à chacun un identifiant `id`, commençant à 1
  - `id` est unique à chaque objet → une variable d'instance
- on veut également un attribut qui permet de garder une trace du nombre d'objets `Bicycle` créés afin de savoir quel `id` sera attribué au prochain objet créé
  - ce nombre n'est pas lié à un objet mais à la classe, c'est donc une variable de classe

Exemple de variable de classe (cont.)

```
public class Bicycle{

  private int cadence;
  private int gear;
  private int speed;

  private int id;
  private static int numberOfBicycles = 0;

  public Bicycle( int startCadence, int startSpeed, int startGear ) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
    // increment number of Bicycles and assign ID number
    id = ++numberOfBicycles;
  }

  // new method to return the ID instance variable
  public int getID() {
    return id;
  }
}
```

Les méthodes de classe

- `static` permet de créer des méthodes de classe
  - `ClassName.methodName(args)` (recommandé)
  - `instanceName.methodName(args)`
- les méthodes statiques peuvent être appelées sans avoir a créer une instance de la classe
- les méthodes statiques sont le plus souvent utilisées pour accéder aux variables de classe

Les méthodes de classe (cont.)

- exemple avec l'ajout de la méthode ci-dessous :

```
public static int getNumberOfBicycles() {
  return numberOfBicycles;
}
```

- pour accéder au nombre d'objets `Bicycle` créés :

```
Bicycle.getNumberOfBicycles()
```

Les constantes

- le modificateur `static`, combiné au modificateur `final` sert à définir des variables constantes
- `final` indique que la valeur de la variable ne peut pas changer
- par exemple, la constance `PI` peut être définie par

```
static final double PI = 3.1415926535897793;
```

Questions : classes

- considérons la classe suivante :

```
public class IdentifyMyParts {
  public static int x = 7;
  public int y = 3;
}
```

1. quelles sont les variables de classe ?
2. quelles sont les variables d'instance ?
3. quelle sera la sortie du code suivant ?

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Réponses : classes

- considérons la classe suivante :

```
public class IdentifyMyParts {
  public static int x = 7;
  public int y = 3;
}
```

1. quelles sont les variables de classe ? **`x`**
2. quelles sont les variables d'instance ? **`y`**
3. quelle sera la sortie du code suivant ?

```
a.y = 5
b.y = 6
a.x = 2
b.x = 2
IdentifyMyParts.x = 2
```

Questions : objets

1. qui a-t-il de faux dans ce programme ?

```
public class SomethingIsWrong {
  public static void main(String[] args) {
    Rectangle myRect;
    myRect.width = 40;
    myRect.height = 50;
    System.out.println("myRect's area is " + myRect.area());
  }
}
```

2. combien de références aux objets existent après exécution du code ci-dessous ? les objets sont-ils détruits par le ramasses miettes ?

```
String[] students = new String[10];
String studentName = "Peter Parker";
students[0] = studentName;
studentName = null;
```

3. comment un programme détruit-il un objet ?

Réponses : objets

1. le compilateur génère une erreur car l'objet Rectangle n'a pas été créé

```
Rectangle myRect = new Rectangle();
```

2. une référence au tableau `students` et ce tableau a une référence à la chaine `Peter Smith`, aucun objet n'est admissible pour le ramasses miettes
3. un programme ne mentionne pas explicitement la destruction des objets, il peut mettre la référence d'un objet à `null` afin qu'il devienne admissible pour le ramasses miettes

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

Les classes imbriquées

- il est possible de définir une classe à l'intérieur d'une autre classe : on parle alors de classe imbriquée
- deux catégories de classes imbriquées :
  - les *classes imbriquées statiques*, déclarées static
  - les classes non-statiques appelées *classes internes*

```
class OuterClass {
  ...
  static class StaticNestedClass {
    ...
  }
  class InnerClass {
    ...
  }
}
```

Pourquoi utiliser des classes imbriquées ?

- *regroupement logique de classes*
  - si une classe n'est utile que pour une autre classe, il est alors logique de l'intégrer dans cette classe
- *augmentation de l'encapsulation*
  - considérons deux classes A et B, où B a besoin d'accéder aux membres de A qui devraient être privés. En imbriquant la classe B dans A, les membres de A déclarés comme privés sont accessibles par B
- du code plus lisible et plus facile à maintenir
  - imbriquer les petites classes dans des classes de niveau supérieur

Exemple de classes imbriquées

```
public class OuterClass {
  private int privateMemberVariable = 100;

  public class InnerClass {
    public void printPrivateVariable() {
      System.out.println(privateMemberVariable);
    }
  }

  public void callInnerClassMethod() {
    InnerClass innerClass = new InnerClass();
    innerClass.printPrivateVariable();
  }

  public static void main(String args[]) {
    OuterClass outerClass = new OuterClass();
    outerClass.callInnerClassMethod();
  }
}
```

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

Le type énumération

- le type `enum` est un type dont les champs se composent d'un ensemble fixe de constantes
  - champs en majuscules car se sont des constantes

```
enum Direction {
  NORTH, SOUTH, EAST, WEST
};
Direction.NORTH // access to NORTH

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

Le type énumération (cont.)

- une énumération est une classe spéciale qui fournit un type sécurisé de données constantes

```
enum HandSign {
  SCISSOR, PAPER, STONE
};
...

HandSign playerMove;
HandSign computerMove;

playerMove = HandSign.SCISSOR;
computerMove = HandSign.PAPER;

// playerMove = 0; // Compilation error
```

EnumTest

```
public class EnumTest {
  Day day;

  public EnumTest(Day day) {
    this.day = day;
  }

  public void tellItLikeItIs() {
    switch (day) {
      case FRIDAY:
        System.out.println("Fridays are better.");
        break;

      default:
        System.out.println("Other days are so-so.");
        break;
    }
  }

  public static void main(String[] args) {
    EnumTest firstDay = new EnumTest(Day.MONDAY);
    firstDay.tellItLikeItIs();
    EnumTest fifthDay = new EnumTest(Day.FRIDAY);
    fifthDay.tellItLikeItIs();
  }
}
```

Parcourir une énumération

```
enum Direction {
  NORTH, SOUTH, EAST, WEST
};

public class EnumLoop {
  public static void main(String[] args) {
    // loop using for
    for (Direction d : Direction.values()) {
      System.out.println(d);
    }
  }
}
```

Plan

- Classes et objets
  - Classes
  - Objets
  - Plus sur les classes
  - Les classes imbriquées
  - Le type énumération
  - La documentation du code

La documentation du code

- Java permet de documenter les classes et leurs membres en utilisant des commentaires

```
/**
  Une classe pour donner un
  <b>exemple</b> de
  documentation HTML.
*/
public class Exemple
{
  /**
    Documentation du membre de
    type entier nomme exemple...
  */
  public int exemple;
}
```

La documentation du code (cont.)

- un commentaire de documentation est placé juste avant l'entité commentée (classe, constructeur, etc.)
- des attributs spéciaux peuvent être définis avec @

```
/**
  Obtenir la somme de deux entiers.
  @param a Le premier nombre entier.
  @param b Le deuxieme nombre entier.
  @return La valeur de la somme des deux entiers.
*/
public int somme(int a, int b) {
  return a+b;
}
```

La documentation du code (cont.)

Attribut	Description
`@author`	nom de l'auteur de la classe
`@version`	version de la classe
`@deprecated`	marquer l'entité comme obsolète, décrire pourquoi et par quoi la remplacer
`@see`	ajouter un lien dans la section "Voir aussi"
`@param`	décrire un paramètre de méthode
`@return`	décrire la valeur retournée par une méthode

La documentation du code (cont.)

```
/**
  Une classe d'exemple.
*/
public class Exemple {
  /**
    Obtenir le carre d'un entier.
    @param a Le nombre entier.
    @return La valeur de du carre de l'entier.
  */
  public int carre(int a) {
    return a*a;
  }
}
```

- commande pour générer la javadoc

```
javadoc -d src/doc/ src/Exemple.java
```

Exemple de javadoc

Résumé des notions abordées

- Classes : déclaration, constructeurs, surcharge de méthodes, passage de paramètres, modifieurs d'accès, classes imbriquées
- Objets : instanciation, accès aux membres
- Enumérations, documentation du code