Les classes anonymes

Algorithmique et langage de programmation

Gaël Thomas

Le problème

- La syntaxe utilisée pour l'héritage de classe/mise en œuvre d'interface est trop verbeuse pour des codes simples

```
interface Bird { void fly(); }

class MyBird implements Bird {
  void fly() {
    System.out.println("fly!");
  } }

class Test {
  void f() {
    Bird bird = new MyBird();
  } }
```

Nécessite une nouvelle classe,
donc un fichier,
le tout pour allouer
une **unique instance** et
peu de lignes de code

Les classes anonymes

- But : simplifier la syntaxe utilisée pour l'héritage de classe ou la mise en œuvre d'interface pour les codes simples
  - Allocation d'une **unique instance** de la classe dans le code
  - Peu de méthodes et de lignes de code dans la classe

- Principes :
  - Omettre le nom de la classe
  - Donner le code de la mise en œuvre au moment de l'allocation

Les classes anonymes par l'exemple

```
interface Bird { void fly(); }

class MyBird implements Bird {
  void fly() {
    System.out...;
  } }

class Test {
  void f() {
    Bird bird = new MyBird();
  } }
```

Mise en œuvre
au moment de l'allocation

Définit une nouvelle
classe qui hérite de Bird
et qui n'a pas de nom

```
class Test {
  void f() {
    Bird bird = new Bird() {
      void fly() {
        System.out...
      }
    }}
}
```

Les expressions lambda (1/2)

- Pour les cas les plus simple, le code reste verbeux, même avec les classes anonymes

- But des expressions lambda : simplifier la mise en œuvre d'une interface **fonctionnelle**
  - Interface **fonctionnelle** = interface avec une unique méthode

- Expression lambda = expression courte d'une méthode
  - `(arguments sans type) -> corps de la méthode;`
  - Exemple : `(x, y) -> x + y;`
  - Remarque : il existe des variations, non étudiée dans le cours, dans la façon de déclarer une expression lambda

Les expressions lambda par l'exemple

- Remplacement de la mise en œuvre de l'interface fonctionnelle par une expression lambda

```
interface Bird { void fly(); }
```

```
Bird bird = () -> { System.out.println("fly!"); };
```

⇔

```
Bird bird = new Bird() {
  void fly() {
    System.out.println("fly!");
  }
};
```

On ne garde que la partie « intéressante » de la déclaration

Portée de variables

- Accès aux champs de la classe anonyme, mais aussi de la classe englobante et aux variables de la méthode englobante
  - Accès en lecture seule aux variables de la méthode

```
class Nid {
  int x;      /* accès en lecture/écriture */

  void f() {
    int y;    /* en lecture seule */
    Bird bird = new Bird() {
      int z; /* accès en lecture/écriture */
      void fly() { z = x + y; }
    };
  } }
```

Notions clés

- Classe anonyme
  - Simplifie l'héritage de classe et la mise en œuvre d'interface dans les cas simples
  - `Bird bird = new Bird() { void fly() { … } };`

- Expressions lambda :
  - Simplifie encore le code pour les interfaces fonctionnelles
  - Interface fonctionnelles = interface avec une unique méthode
  - `Bird bird = () -> { System.out.println( "fly!"); }`

Pour approfondir

Le but de ce sous-cours optionnel est de comprendre comment sont construites les classes anonymes et les lambda de façon à comprendre la portée des variables et des champs

Plan du sous-cours

- Classe anonyme = classe interne de méthode sans nom

Plan du cours

1. Les classes internes
2. Les classes internes de méthodes
3. Les classes anonymes

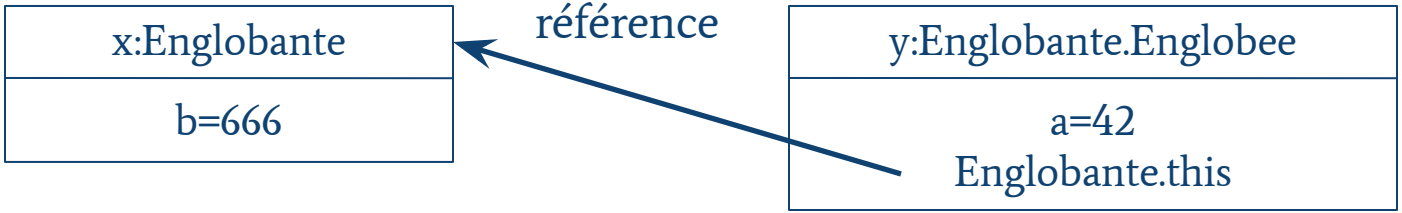
Classes internes

- Une classe interne est une classe définie dans une autre classe
  - Permet de **coupler** fortement deux classes
    - (la classe interne a accès aux champs de la classe externe)

```
class Englobante {
  class Englobee {
    int a;
    void f() { a = 42; b = 666; }
  }
  private int b;
  private Englobee englobee;
}
```

Mise en œuvre d'une classe interne

- Une classe interne possède un champ ajouté automatiquement permettant d'accéder à la classe externe




```
void f() { a = 42; b = 666; }
```

⇔

```
void f() { this.a = 42; Englobante.this.b = 666; }
```

Allocation d'une classe interne

- À partir de la classe externe, allocation de façon normale

```
class Englobante {
  class Englobee { … }
  Englobante() {
    englobee = new Englobee();
  }
}
```

- À partir de l'extérieur de la classe englobante :

```
Englobante x = new Englobante();
Englobante.Englobee y = x.new Englobee();
x.englobee = y; /* à faire à la main */
```

Classe interne et visibilité

- Les champs de la classe englobée sont toujours accessibles à partir de la classe englobante et vice-versa
- Composition des opérateurs de visibilités pour l'extérieur

```
public class Englobante { /* accessible partout */
  class Englobee {  /* accessible du package */
    private int x;  /* accessible de Englobante */
  }
  private int b;    /* accessible de Englobee */
  private Englobee englobee; /* idem */
}
```

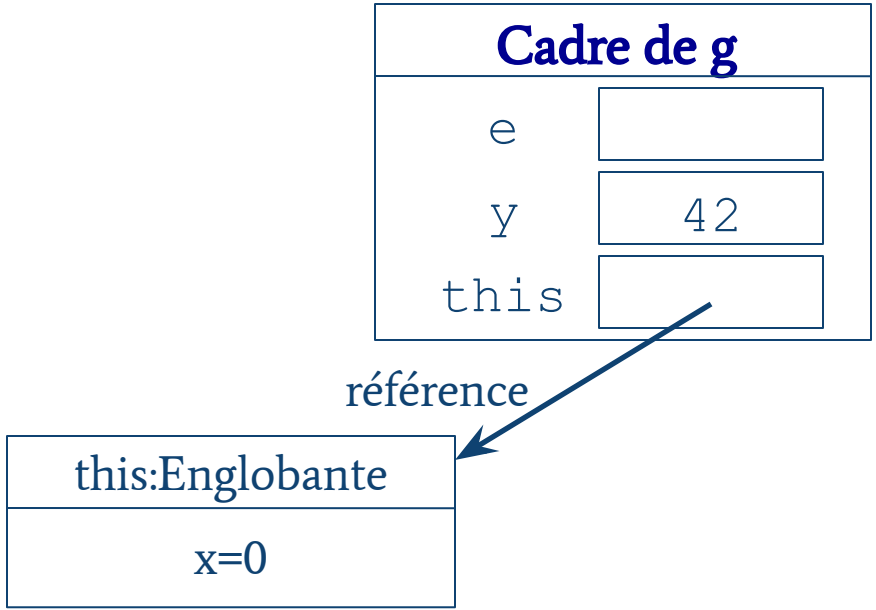
Classe interne de méthode

- Classe interne de méthode = classe définie dans une méthode
- La classe interne peut aussi accéder aux variables locales de la méthode (si var. que accédées en **lecture** après la déclaration)

```
class Englobante {
  int x;
  void f() {
    int y = 42; /* y en lecture seule dans Englobee */
    class Englobee { void g() { x = y; } };
    Englobee e = new Englobee();
    e.g();
  } }
```

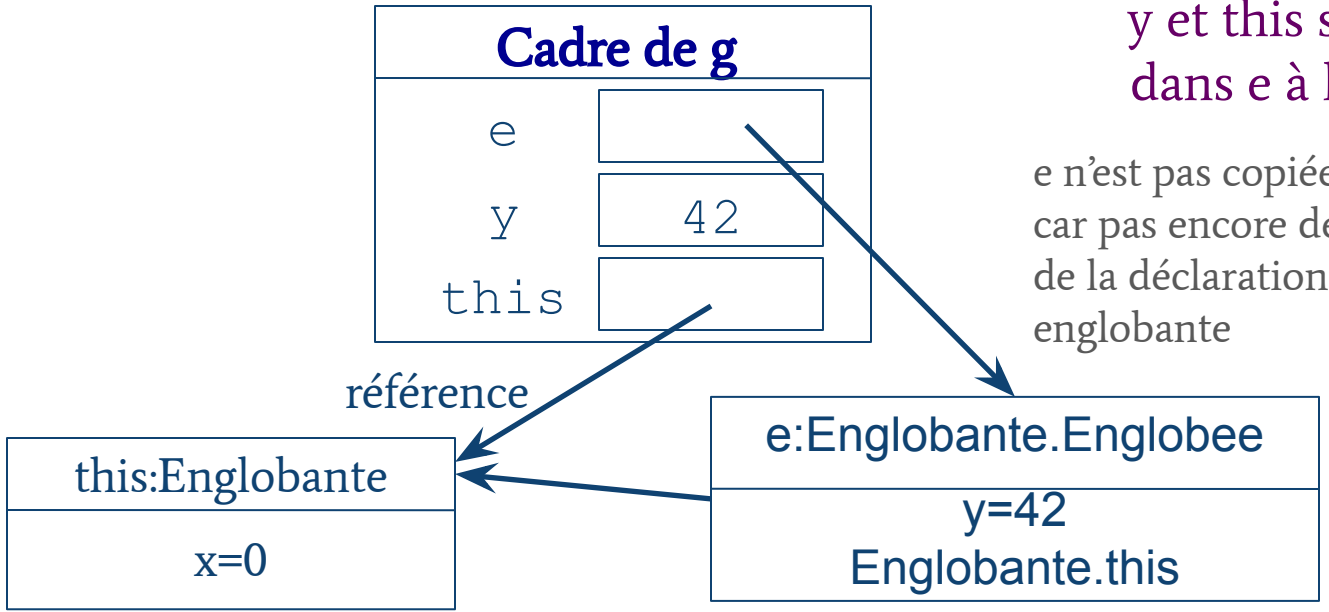
Mise en œuvre

- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



Mise en œuvre

- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



y et this sont copiés dans e à l'allocation

e n'est pas copiée car pas encore définie lors de la déclaration de la classe englobante

Classes anonymes (1/2)

- Classe anonyme = classe interne de méthode sans nom

```
void f() {                    /* en version nommée */
  class AvecNom extends Bird {
    void fly() { System.out.println("fly!"); }
  };
  Bird bird = new AvecNom();
}
```

⇔

```
void f() {                   /* en version anonyme */
  Bird bird = new Bird() {
    void fly() { System.out.println("fly!"); }
  };
}
```

Classes anonymes (2/2)

- Une classe anonyme est une classe interne de méthode
  - Si méthode d'instance, peut accéder aux champs de l'instance
  - Peut accéder aux variables locales de la méthode
    - *à condition que ces variables ne soient accédées qu'en lecture à partir de la déclaration de la classe anonyme*

```
class Test {
  int x;  /* lecture/écriture à partir de cl. anon. */
  void f() {
    int a = 42;    /* lecture à partir de cl. anon. */
    Bird bird = new Bird() { void fly() { x = a; } };
    bird.fly();
  } }
```

Et avec des expressions lambda

```
interface Bird { void fly(); }


   class Test {
     int x;      /* lecture/écriture à partir de lambda */
     void f() {
       int a = 42; /* lecture seule à partir de lambda */
       Bird bird = () -> x = a;
       bird.fly();
     }
   }
```

Notions clés (du sous-cours optionnel)

- Classe anonyme = classe interne de méthode sans nom
- Classe interne de méthode = classe interne définie dans une méthode

  Peut accéder aux variables de la méthode (si variables que lues)
- Classe interne = classe définit dans une autre classe

  Possède une référence vers une instance de la classe englobante

Pour votre culture : classe interne statique

- Si la classe interne est marquée `static`
  - pas liée à une instance de la classe englobante
  - pas d'accès aux champs d'instance d'une instance englobante
  - allocation sans instance d'une classe Englobante

```
class A {
  static class B {
    int a;
    void f() { a = 42; c = 666; } /* pas d'accès à b */
  }
  private int b;
  private static int c;
} /* allocation avec A.B x = new A.B(); */
```