Année 2008-2009

esil

Constructeurs, objets et tas

Nicolas Baudru
mél : nicolas.baudru@esil.univmed.fr
page web : nicolas.baudru.perso.esil.univmed.fr

Nous avons déja vu :

- comment créer des objets ;
- lors de leur création, les objets résident sur le tas ;
- un objet reste accessible tant qu'une variable référence pointe sur cet objet ;
- un objet non référencé ne peut plus être utilisé ;
- un objet non référencé est un candidat au ramasse-miettes.

Où vis-tu ?

esil

En programmation, deux zones mémoires sont importantes :

- le tas : c'est là où les objets vivent ;
- la pile : les appels de méthodes et les variables locales y vivent.

Où vivent les variables locales références ?

Où vivent les variables d'instance primitives ?

Où vivent les variables d'instance références ?

Où vivent les tableaux ?

Comment sont représentés les objets issus de l'héritage ?

Fonctionnement de la pile

- Lors de l'appel d'une méthode f(), f() est empilée avec ses variables locales.
- A la fin du traitement de f(), f() est dépilée, ainsi que ses variables locales.
- La méthode au sommet de la pile est toujours celle en cours d'exécution.

Exemple :

```
public void toDo() {
  boolean b = true;
  go(4);
}

public void go (int x) {
  int z = x + 24;
  next();
}

public void next() {
  char c = 'a';
}
```

Les variables d'instance vivent sur le tas

esil

- Lorsqu'un "new MyClass()" est exécuté, la JVM fait de la place sur le tas pour le classe MyClass, en tenant compte de ses variables d'instance.
- Les variables d'instance d'un objet sont dans cet objet. Java leur alloue de la mémoire en fonction de leur type. Ex : 32 bits pour un int, 64 pour un long.

Que se passe-t-il si la variable d'instance est une référence ?

```
public class PC {
  int poids = 3;
  Clavier c;
  Ecran e = new Ecran();
  ...
}
```

Représentez l'état du tas juste après la création d'un objet PC ?

esil

```
Chien monChien = new Chien();
```

1. Déclarer une variable référence : Chien monChien = new Chien();
2. Créer un objet : Chien monChien = new Chien();
3. Lier l'objet et la référence : Chien monChien = new Chien();

Qu'est que Chien() ?

Chien() est le constructeur de Chien. Un constructeur ressemble à une méthode portant le nom de la classe mais ce n'est pas une méthode. Il contient le code qui va s'exécuté lors de son invocation à l'aide du mot clé new.

- La seule façon d'invoquer un constructeur est l'utilisation du mot clé new suivi du nom de la classe. (En fait ce n'est pas tout à fait exact.)
- La JVM trouve cette classe et appelle le constructeur contenu dans cette classe.
- Vous pouvez écrire un constructeur pour votre classe, mais si vous ne le faites pas, le compilateur l'écrit à votre place.
- Le constructeur par défaut ressemble à :

```
public Chien()  {   }
```

Que remarquez-vous ?

Créer son propre constructeur

L'intérêt d'un constructeur est qu'il s'exécute juste après la création de l'objet, et avant que cet objet puisse être affecté à une référence. Cela vous permet de «préparer» l'objet à sa future utilisation. Vous pouvez par exemple initialiser des variables d'instance.

```
public class Canard {
  int poids;

  public Canard() {
    poids = 3; // poids par défaut
    System.out.println("Coin");
  }
}

public class TestCanard {
  public static void main (String[] args) {
    Canard c = new Canard();
  }
}
```

Initialiser votre Canard

Le meilleur endroit pour effectuer l'initialisation d'un objet est le constructeur. Il suffit de créer un constructeur avec des arguments. Comme pour les méthodes, vous pouvez surcharger les constructeurs.

```
public class Canard {
  int poids;

  public Canard() {
    poids = 3;  // poids par défaut
  }

  public Canard(int poidsCanard) {
    poids = poidsCanard;
  }
}

public class TestCanard {
  public static void main (String[] args) {
    Canard c1 = new Canard();
    Canard c2 = new Canard(3);
  }
}
```

Surcharge des constucteurs

- Si vous écrivez votre propre constructeur, le compilateur n'en crée pas.
- Si une classe contient plusieurs constructeurs, leurs listes d'arguments doivent être différentes (ordre ou type). C'est la surcharge de constructeur.

```
public class Canard {
  int poids;

  public Canard(int poidsCanard) {
    poids = poidsCanard;
  }
}

public class TestCanard {
  public static void main (String[] args) {
    Canard c1 = new Canard();
    Canard c2 = new Canard(3);
  }
}
```

Cela compile-t-il ?

Un constructeur doit-il être public ?

Si non, à quoi peut servir un constructeur privé ?

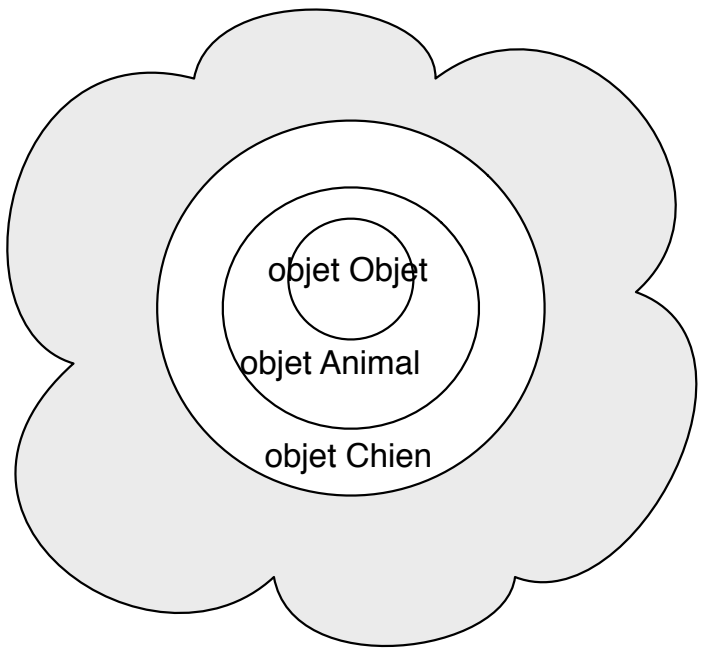
Et que se passe-t-il si votre classe hérite d'une superclasse ?

Une classe abstraite doit-elle avoir un constructeur ?

Et une interface ?

Rappel : tout objet contient ses variables et ses méthodes ainsi que celles des classes dont il hérite.

Quand le constructeur d'une classe est appelé, un processus de chaînage des constructeurs est réalisé : le constructeur appelé invoque le constructeur de la superclasse (appelé le superconstructeur), qui invoque lui aussi son superconstructeur, et ainsi de suite jusqu'à la superclasse Objet.



```
public class Animal {
  public Animal () { System.out.println("Animal");}
}

public class Chien extends Animal {
  public Chien() { System.out.println("Chien");}
}

public class TestChien {
  public static void main(String[] args){
    Chien c = new Chien();
  }
}
```

Quel est le résultat de ce programme ?

Appeler le superconstructeur de la superclasse

- Le superconstructeur d'une superclasse A ne peut être appelé que par le constructeur d'une classe qui hérite de A.
- Pour appeler le superconstructeur de A, il suffit alors d'utiliser l'instruction super().
- super() doit être appelé avant toute autre instruction !
- Si on omet l'appel à super() dans un constructeur, le compilateur se charge de l'ajouter au tout début du code de ce constructeur.
- De manière récurrente, le superconstructeur commencera par appeler son superconstructeur, jusqu'à atteindre l'objet Objet.

Appeler le superconstructeur de la superclasse

esil

```
public class Animal {
  public Animal () { System.out.println("Animal");}
}

public class Chien extends Animal{
  public Chien() {
    super();  // Attention, Animal() ne fonctionne pas !
    System.out.println("Chien");
  }
}

public class TestChien {
  public static void main(String[] args){
    Chien c = new Chien();
  }
}
```

super() doit être appelé avant toute autre instruction !

Appeler le superconstructeur de la superclasse

esil

```
public class Animal {
  public Animal() { System.out.print("Animal");}
  public Animal (int i){ System.out.print( i ); }
}

public class Chien extends Animal{
  public Chien() {
    -----------
    System.out.println("Chien");
  }
}

public class TestChien {
   public static void main(String[] args){
     Chien c = new Chien();
   }
 }
```

Complétez le programme afin d'obtenir "Animal Chien" ?

Complétez le programme afin d'obtenir "3 Chien" ?

Superconstructeur avec argument

```
public abstract class Animal {
  private String nom;
  public Animal (String leNom){nom = leNom; }
}

public class Chien extends Animal{
  public Chien (String nomDuChien) {
    super(nomDuChien);
  }
}

public class TestChien {
  public static void main(String[] args){
    Chien c = new Chien("Fido");
  }
}
```

Invoquer un constructeur surchargé

- Utiliser this() pour appeler un constructeur depuis un constructeur surchargé dans la même classe.
- L'appel de this() ne peut se trouver que dans un constructeur.
- L'appel de this() doit être la première instruction du constructeur.
- Un constructeur peut appeler this() ou super(), mais pas les deux.

```
public class Chien extends Animal{
  public Chien() {
    this("Fido", 20)
  }

  public Chien (String nomDuChien, tailleDuChien) {
    super(nomDuChien, tailleDuChien);
    // reste de l'initialisation
  }
}
```

Durée de vie d'une variable

La durée de vie d'une variable dépend de sa nature (locale ou d'instance).

- Une variable locale n'a d'existence que dans la méthode qui l'a déclarée. Tant que la méthode réside sur la pile, ses variables locales vivent.
- Une variable d'instance vit aussi longtemps que l'objet qui la contient.

Différence entre portée et durée de vie des variables locales :

- Durée de vie : Une variable locale est en vie tant que son bloc est sur la pile, i.e. tant que l'exécution de la méthode n'est pas terminée.
- Portée : Une variable locale n'est visible que dans la méthode M1 qui la déclare. Si une M1 appelle une autre méthode M2, la variable locale n'est plus dans la portée du programme (i.e. elle ne peut plus être utilisée) et ce jusqu'à la reprise de l'exécution de la méthode M1.

Durée de vie d'un objet

La durée de vie d'un objet dépend de la durée de vie des variables références qui lui sont liées. Un objet est vivant tant qu'il a des références vivantes.

Si un objet n'a plus de référence, il est alors perdu et candidat au ramasse-miettes. Un objet est candidat au ramasse-miettes quand sa dernière référence vivante disparaît.

Trois façons de perdre une référence :

- Elle meurt à la fin de l'exécution d'une méthode

```
void go() {Chien c = new Chien();}
```

- Elle est affectée à un autre objet :

```
Chien c = new Chien();
c = new Chien();
```

- Elle est positionnée à null :

```
Chien c = new Chien();
c = null;
```

Ce que nous avons vu