Introduction programmation Java

Cours 9

Stéphane Airiau

Université Paris-Dauphine

Performances sur des opérations

Pour de larges volume de données, les méthodes usuelles (add, remove, contains, size) devraient être rapides.

Implémentation de l'interface `Set`

	`add`	`remove`	`contains`
`HashSet`	temps constant	temps constant	temps constant
`TreeSet`	$log(n)$	$log(n)$	$log(n)$

Implémentation de l'interface `List`

	`get`	`set`	autre
`LinkedList`	n	n	
`ArrayList`	temps constant*	temps constant	n

Performances empirique

Je veux comparer les performances de la méthode`contains` pour les classes `ArrayList`, `LinkedList`, `HashSet` et `TreeSet`.

> J'ai un dictionnaire d'environs 25,000 mots à ma disposition
> Pour 100 exécutions
>
> 1- je tire au sort environ 10,000 mots du dictionnaire et je les places dans la structure
>
> 2- je tire au sort 1000 mots du dictionnaire et je les place dans un tableau de `String`
>
> 3- je mesure le temps qu'il faut pour savoir si chaque mot du tableau de `String` se trouve dans ma structure
>
> Je calcule le temps total pour chaque structure

structure	`ArrayList`	`LinkedList`	`HashSet`	`TreeSet`
`contains`	3.245 s	5.286 s	0.004 s	0.024 s
`remove`	3.155 s	5.378 s	0.003 s	0.041 s

Temps total

Un peu d'algorithmique

- On veut stocker un ensemble de k éléments qui ont une clé unique.
- Les clés peuvent prendre valeur dans un univers U.
- Si U est grand, il n'est pas pratique, voir impossible, de faire un tableau de taille U pour stocker un élément qui a pour clé n dans la case $n^{ième}$ case du tableau.
- Si $k << |U|$, on peut utiliser une **fonction de hachage** h et l'élément qui a pour clé i sera stocké dans la case $h(i)$.
  On dit aussi que $h(i)$ est la valeur de hachage de i.

⇨ on cherche à minimiser la taille du tableau qui stocke les éléments.

- **problème :** deux clés peuvent avoir la même valeur de hachage (inévitable)
  on a alors une "collision"

Résoudre les collisions avec chainage

idée : on stocke tous les éléments qui ont une même valeur de hash dans une liste chaînée.
⇨la case i contient une référence vers la tête de la liste.
Supposons qu'on ait k valeurs de hachage et n éléments

- si les n éléments ont la même valeur de hachage, on aura simplement une longue liste chainée ⇨ **aucun** intérêt !
- la situation *idéalle* est une répartition uniforme des n éléments sur les k valeurs de hachage.

- l'insertion d'un élément se fait donc en $\mathcal{O}(1)$
- enlever un élément se fait en $\mathcal{O}(1)$ si la liste est doublement chaînée
- cherche un élément se fait en $\Theta(1+\alpha)$ où α est le nombre moyen d'éléments stockés dans une chaîne.

Fonction de Hachage

Supposons qu'on ait k valeurs de hachage.
On veut une fonction qui distribue de manière uniforme toutes les clés dans chacun des k.
Le problème, c'est qu'on ne connait que rarement la probabilité qu'on ait un élément avec une certaine clé.
Il existe des techniques pour générer de bonnes fonctions de hachage, mais ce n'est pas le problème dans ce cours.

Retour sur `HashCode()` de la classe `Object`

Pour rappel, la classe `Object` possède une méthode `hashCode()` qui retourne un `int`.
⇨ la méthode `hashCode` est la fonction de hachage pour la classe !

⇨ **Attention !** si vous utilisez un `HashSet` ou un `HashMap`, l'implémentation de `hashCode()` est **très** important.

- si vous pouvez vous ramener à des fonctions de hachage codé dans `Java`, cela pourra avoir de bonnes performances (ex `String`).
- si votre fonction de hachage est constante, vous perdez tout bénéfice d'utiliser ces structures
- si votre fonction de hachage n'est pas cohérente par rapport à votre méthode `equals`, vous risquez de ne pas trouver un élément pourtant présent !
  - si `this.equals(obj)` retourne vrai, on doit aussi avoir `this.hashCode()==obj.hashCode()`.

Retour sur `HashCode()` de la classe `Object`

Il existe une méthode `Objects.hash(varargs)` qui calcule la valeur de hachage des ses arguments et les combine automatiquement.

La classe `Object` possède une méthode `hashCode`, mais qui utilise des méthodes qui dépendent de l'implémentation locale. La valeur peut être dérivée de l'adresse mémoire, d'un nombre aléatoire, ou d'une combinaison des deux.

Un mot sur la boucle `for each`

```
for (Personnage p : villageois)
      System.out.println(p);
```

- Facilité syntaxique pour écrire plus rapidement une boucle.
- Lors de la compilation, `Javava` traduire cette ligne en utilisant un `Iterator`.

Attention A l'intérieur de la boucle, on **ne** peut **pas** modifier la collection (par exemple avec `remove()`).
⇨ sinon, cela cause une `ConcurrentModificationException`.

Classes internes

Cours 9

Stéphane Airiau

Université Paris-Dauphine

Classes Internes

Exemple classe interne `static` et `public`

```
1  public class Facture{
2    public static class Item {
3      private String description;
4      private int quantite;
5      private double prixUnitaire;
6      public Item(String desc, int q, double pU){
7        this.description = desc;
8        this.quantite = q;
9        this.prixUnitaire = pU;
10     }
11     double montant(){return quantite*prixUnitaire;}
12   }
13
14   private List<Item> liste = new ArrayList<>();
15
16   public void addItem(Item it){
17     liste.add(it);
18   }
19
20 }
```

Utilisation de l'encapsulation (comme d'habitude).

classes internes `static` et `public`

N'importe qui peut alors construire un objet de la classe interne en utilisant le nom "complet", ici `Facture.Item`.

```
Facture.Item newItem = new Facture.Item("Moka Yrgacheffe", 2, 5.5);
```

Le compilateur génère deux classes différentes :

- `Facture.class`
- `Facture$Item.class`

Ce qui montre bien qu'il n'y a pas trop de différences entre une classe `static` et une classe "classique"

⇨ juste un moyen pour cacher une classe !

Exemple classe interne `static` et `private`

```
1  public class Facture{
2     private static class Item {
3       String description;
4       int quantite;
5       double prixUnitaire;
6
7       double montant(){return quantite*prixUnitaire;}
8     }
9
10    private List<Item> liste = new ArrayList<>();
11    ...
12  }
```

Ici, la classe interne est privée (pas besoin d'utiliser `private` pour les attributs).

⇨ seulement des méthodes de la classe englobante (ici `Facture`) ont accès à la classe interne.

Exemple classe interne `static` et `private`

```
1  public class Facture{
2     private static class Item {
3       String description;
4       int quantite;
5       double prixUnitaire;
6
7       double montant(){return quantite*prixUnitaire;}
8     }
9
10    private List<Item> liste = new ArrayList<>();
11    ...
12    public void addItem(String des, int q, double pU){
13      Item newitem = new Item();
14      newItem.description = des;
15      newItem.quantite = q;
16      newItem.prixUnitaire = pU;
17      liste.add(newItem);
18      }
19  }
```

Exemple de construction d'une instance de la classe interne.
Tout se passe comme si la classe interne est une classe normale.

Classes Internes

Il est possible de définir une classe à l'intérieur d'une classe !
⇨ de telles classes peuvent simplifier le développement

- cacher des détails d'implémentation (visibilité restreinte)
- avoir des packages avec moins de classes

Il y a deux grands types de classes internes, qui sont assez différents

- *classe interne d'instance*
  - cette classe va donc accéder à tous les champs de sa classe (même les privés)
  - on ne peut créer une instance d'une classe interne que depuis une méthode d'instance de sa classe « englobante »
- *classe interne statique*
  - pas de référence à des membres d'instance de sa classe « englobante »
  - on peut instancier cette classe (elle est `static` vis à vis de sa classe englobante).
  - ⇨ une classe interne static est vraiment comme une classe classique (sauf qu'elle est codée à l'intérieur d'une autre classe)

Classes internes

Deux grosses différences :

- une classe interne **a accès** aux éléments de sa classe englobante !
- il faut donc un objet de la classe englobante pour qu'un objet de la classe interne puisse exister !

```
1  public class ReseauSocial {
2
3    public class Membre {
4      private String nom;
5      private List<Membre> amis;
6
7      public Membre(String nom){
8        this.nom=nom;
9        amis = new LinkedList<>;
10     }
11     ...
12   }
13
14   private List<Member> membres;
15   ...
16 }
```

Classes internes

```
1  public class ReseauSocial {
2
3    public class Membre {
4      private String nom;
5      private List<Membre> amis;
6
7      public Membre(String nom){
8        this.nom=nom;
9        amis = new LinkedList<>;
10     }
11     ...
12   }
13
14   private List<Member> membres;
15   ...
15   public void accepte(String nom){
16     membres.add(new Membre(nom));
17   }
18 }
```

Classes internes : utilisation

```
ReseauSocial faceBouc = new ReseauSocial();
ReseauSocial.Membre chevre = faceBook.accepte("chevre");
```

Si la `chevre` veut quitter le réseau, elle peut appeler la méthode pour partir. On peut implémenter la méthode comme suit :

```
1  public class ReseauSocial {
2
3     public class Membre {
4        ...
5       public void quitter(){
6          membres.remove(this);
7       }
8     }
9
10    List<Membre> membres;
11 }
```

La classe interne peut accéder aux attributs de la classe englobante !

Classes internes : utilisation

```
ReseauSocial faceBouc = new ReseauSocial();
ReseauSocial.Membre chevre = faceBook.accepte("chevre");
```

```
1  public class ReseauSocial {
2
3     public class Membre {
4        ...
5       public void quitter(){
6          efface(this);
7       }
8     }
9
10    List<Membre> membres;
11    public void accepte(String nom){ ... }
12    public void efface(Membre m){ ... }
13 }
```

La classe interne peut accéder aux méthodes de la classe englobante !

Classes internes : accès à la classe englobante

Imaginons que l'on veuille vérifier si un membre appartient au bon réseau.

```
1  public class ReseauSocial {
2
3     public class Membre {
4        ...
5       public void appartient(ReseauSocial rs){
6          return ? == rs;
7       }
8     }
9  }
```

Classes internes : accès à la classe englobante

Imaginons que l'on veuille vérifier si un membre appartient au bon réseau.

```
1  public class ReseauSocial {
2
3     public class Membre {
4        ...
5       public void appartient(ReseauSocial rs){
6          return ReseauSocial.this == rs;
7       }
8     }
9  }
```

Classes internes : création d'une instance

```
 1  public class ReseauSocial {
 2
...
13
14     private List<Member> membres;
15     ...
15     public void accepte(String nom){
16       membres.add(new Membre(nom));
17     }
18   }
```

Ici `new Membre(nom)` est un raccourci pour `this.new Membre(nom)`.

En dehors de la classe englobante, on peut créer un objet de la classe interne à partir d'une instance de la classe englobante :

```
ReseauSocial.Membre mark = faceBouc.new Membre("Mark");
```

Petit détail

- on ne peut déclarer dans une classe interne que des variables `static` qui soient des constantes
  (sinon, il y aurait une ambiguité)

Exemples

Classe interne `static`

```
1 public class Externe {
2    int nbEtudiants;
3    public static class Interne {
4      public int nbEtudiantsMax=25;
5    }
6 }
```

```
1 Externe a = new Externe();
2 Externe.Interne b=
3    new Externe.Interne();
```

Classe interne `non static`

```
1 public class Externe {
2    int nbEtudiants;
3    public class Interne {
4      public int nbEtudiantsMax=25;
5    }
6 }
```

```
1 Externe a = new Externe();
2 Externe.Interne b=
3   a.new Interne();
```

Classes Locales

ou comment définir de manière concise des classes qui implémentent une interface
On peut définir une classe à l'intérieur d'une **méthode** !
Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.

Classes Locales

ou comment définir de manière concise des classes qui implémentent une interface
On peut définir une classe à l'intérieur d'une **méthode** !
Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.

Classes Locales

ou comment définir de manière concise des classes qui implémentent une interface
On peut définir une classe à l'intérieur d'une **méthode** !
Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
- la méthode doit retourner un objet qui implémente l'interface `IntSequence`

Classes Locales

ou comment définir de manière concise des classes qui implémentent une interface
On peut définir une classe à l'intérieur d'une **méthode** !
Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
- la méthode doit retourner un objet qui implémente l'interface `IntSequence`
- l'appelant n'est pas intéressé par le type de l'objet
  Tout ce qu'il l'intéresse, c'est que l'objet implémente l'interface

Classes Locales

ou comment définir de manière concise des classes qui implémentent une interface
On peut définir une classe à l'intérieur d'une **méthode** !
Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
- la méthode doit retourner un objet qui implémente l'interface `IntSequence`
- l'appelant n'est pas intéressé par le type de l'objet
  Tout ce qu'il l'intéresse, c'est que l'objet implémente l'interface

⇨ on va faire une classe had hoc à l'intérieur de la méthode !

Exemple Classes Locales

```
1  private static Random gen = new Random();
2
3  public static IntSequence randomInts(int low, int high) {
4     class RandomSequence implements IntSequence {
5       public int next(){
6         return low + gen.nextInt(high-low+1);
7       }
8       public boolean hasNext(){return true;}
9     }
10    return new RandomSequence();
11  }
```

- pas besoin de spécifier si la classe locale est `public` ou `private` puisqu'elle ne sera jamais accessible à l'extérieur de la méthode !
- la classe a accès aux variables accessible à l'intérieur de son scope. Dans l'exemple, elle a **directement** accès à
  - `low` et `high` qui sont des paramètres de la méthode
  - `gen` qui est une variable `static` de la classe

Classe Anonyme

```
1  private static Random gen = new Random();
2
3  public static IntSequence randomInts(int low, int high) {
4     class RandomSequence implements IntSequence {
5       public int next(){
6         return low + gen.nextInt(high-low+1);
7       }
8       public boolean hasNext(){return true;}
9     }
10    return new RandomSequence();
11  }
```

Dans l'exemple, on a utilisé le nom `RandomSequence` une fois pour créer l'objet.

Dans ce cas, on pourrait se passer d'utiliser un nom ⇨ classe **anonyme**.

Classe Anonyme

```
1  private static Random gen = new Random();
2
3  public static IntSequence randomInts(int low, int high) {
4     return new IntSequence() {
5       public int next(){
6         return low + gen.nextInt(high-low+1);
7         }
8       public boolean hasNext(){return true;}
9     }
10
11  }
```

C'est assez pratique et concis.
Très utilisé pour les interfaces graphiques et les threads.

Maintenant, `Java` propose un autre moyen pour faire cela : l'utilisation de "lambda expressions"
Mais on garde cela pour le cours de `Java` avancé.

λ expression

```
(String first, String second) -> first.length()-second.lenght()
```

```
Runnable task () -> {for (int i=0;i<1000;i++) doWork();}
```

Une interface avec une seule méthode à implémenter est appelée une interface fonctionnelle
ex : `Runnable` et `Comparator`.

Au lieu de créer un objet qui implémente l'interface `Comparator`, on peut fournir une λ expression

```
Arrays.sort(words,
   (String first, String second) -> first.length()-second.lenght())
```

On peut même omettre les paramètres `String` car `Java`pourra les inférer car `Java` sait qu'il va attendre un `Comparator<String>` !

```
Arrays.sort(words,
    (first, second) -> first.length()-second.lenght())
```