Pour aller plus loin : idiomes

Denis Conan

avec Christian Bac

CSC4102

Télécom SudParis

Janvier 2024

Table des matières

2

Sommaire

Dans les sections qui suivent, nous parcourons un ensemble de bonnes façons de programmer, c'est-à-dire idiomes, en JAVA. La plupart des éléments sont extraits du livre de référence de l'un des membres de l'équipe de conception du langage JAVA : J. Bloch, « JAVA efficace », 2nd Edition, 2008. Nous ne pouvons que conseiller la lecture de cet ouvrage. Nous insérons ici des éléments directement en lien avec le module CSC4102.

Une différence importante entre l'étude que nous proposons dans ces quelques pages et le livre de J.Bloch se situe entre le fait que nous concevons une application et le fait que le livre discute aussi de la conception de bibliothèques : par exemple, J. Bloch est un des principaux contributeurs de la bibliothèque des collections JAVA. Dans la conception d'une bibliothèque, la programmation doit faciliter les évolutions futures de la bibliothèque ainsi que l'extension par des tiers de la bibliothèque. Ces compétences ne sont clairement pas dans le périmètre du module CSC4102.

Nous commençons par étudier la création et la destruction d'objets. Puis, nous revenons sur certaines des méthodes communes à tous les objets.

1 Création et destruction d'objets

 # 3

Dans cette section, nous présentons quelques idiomes très utilisés dans la conception de bibliothèque. La conception de bibliothèque met en œuvre des idiomes plus complexes ou plus subtiles.

4

1.1 Intérêt des méthodes de classe pour la création

- Repris de l'*Item 1* de [Bloch, 2008]
- Intérêts des méthodes de classe :
  - Nom, plutôt que `new` : p.ex. `Boolean.valueOf`
  - Peut ne pas créer un nouvel objet, mais retourner la référence d'une instance existante : p.ex. gestion d'un cache
  - Peut retourner un objet d'une classe enfant : p.ex. la classe `Collections` contient plusieurs dizaines de méthodes de classe pour construire tous les types de collections
    - `Collections.singletonList(o)` retourne une liste non modifiable contenant uniquement o
  - Permet des constructions moins verbeuses : p.ex. pour les classes paramétrées
    - `HashMap<String, String> dictionnaire = newInstance();`

Exemple de méthode de classe dans la bibliothèque des collections :

Classe `seance7.creationdestructiondesobjets.CreationParMethodeDeClasse`

```
1  public class CreationParMethodeDeClasse {
       @SuppressWarnings("unused") // pour eviter le warning dans Eclipse
       public static void main(String[] args) {
           List<String> l = Collections.singletonList("un");
5          System.out.println("l instanceof List = " + (l instanceof List)
                   + "\nl de type \"" + l.getClass() + "\"");
           // construction moins verbeuse : d2 moins verbeux que d1
           HashMap<String, String> d1 = new HashMap<String, String>();
9          HashMap<String, String> d2 = newInstance();
       }
       private static <C, V> HashMap<C, V> newInstance() {
           return new HashMap<>();
13     }
   }
```

Résultat de l'exécution :

```
l instanceof List = true
l de type "class java.util.Collections$SingletonList"
```

1.2 Problème du constructeur dépendant d'une méthode rédéfinie

5

- Repris de l'*Item 17* de [Bloch, 2008]
- Concevoir une classe qui peut être étendue
  - Un constructeur ne doit pas appeler une autre méthode de la classe si cette méthode peut être redéfinie
    - Sous risque de surprise

Classe `seance7.creationdestructiondesobjets.ParentConstructeurDependantMethodeRedefinie`

```
public class ParentConstructeurDependantMethodeRedefinie {
  public ParentConstructeurDependantMethodeRedefinie() { methodeRedefinie(); }
  public void methodeRedefinie() { } }
```

Classe `seance7.creationdestructiondesobjets.EnfantConstructeurDependantMethodeRedefinie`

```
public class EnfantConstructeurDependantMethodeRedefinie
                extends ParentConstructeurDependantMethodeRedefinie {
  public EnfantConstructeurDependantMethodeRedefinie() { super(); }
  @Override public void methodeRedefinie() { System.out.println("surprise"); } }
```

Voici un exemple d'utilisation qui suprendra le programmeur qui a étendu la classe `ParentConstructeurDependantMethodeRedefinie`.

Classe `seance7.creationdestructiondesobjets.ExempleConstructeurDependantMethodeRedefinie`

```
package eu.telecomsudparis.csc4102.cours.seance7.creationdestructiondesobjets;

public class ExempleConstructeurDependantMethodeRedefinie {
  public static void main(String[] args) {
    new ParentConstructeurDependantMethodeRedefinie();
    new EnfantConstructeurDependantMethodeRedefinie();
  }
}
```

1.3 Création d'objet avec beaucoup d'attributs, certains optionnels

6

- Repris de l'*Item 2* de [Bloch, 2008]
- Nous aimons dans des langages comme Python écrire

```
def ClasseAvecBeaucoupDAttributs(a, b=1.0, c="valeurpardefaut"): ...

ClasseAvecBeaucoupDAttributs(1)
ClasseAvecBeaucoupDAttributs(1, 2.0)
ClasseAvecBeaucoupDAttributs(1, b=2.0)
```

- Utilisation de l'idiome « *Builder* »
  - Il simule les paramètres optionnels

Classe `seance6.creationdestructiondesobjets.ClasseAvecBeaucoupDAttributs`

```
public class ClasseAvecBeaucoupDAttributs {
        private int a;
        private double b = 1.0;
        private String c = "valeur par defaut";
        public ClasseAvecBeaucoupDAttributs(final int a) {
                this.a = a;
        }
        public ClasseAvecBeaucoupDAttributs b(final double b) {
                this.b = b;
                return this;
        }
        public ClasseAvecBeaucoupDAttributs c(final String c) {
                this.c = c;
                return this;
        }
        @Override
        public String toString() {
                return "ClasseAvecBeaucoupDAttributs [a=" + a + ", b=" + b + ", c=" + c
                                + "]";
        }
}
```

Classe `seance6.creationdestructiondesobjets.CreationAvecIdiomeBuilder`

```
public class CreationAvecIdiomeBuilder {
        public static void main(String[] args) {
                ClasseAvecBeaucoupDAttributs o = (new ClasseAvecBeaucoupDAttributs(1))
                                .b(2.0).c("autre");
                System.out.println(o);
        }
}
```

Résultat de l'exécution :

```
ClasseAvecBeaucoupDAttributs [a=1, b=2.0, c=autre]
```

1.4 Non-instanciabilité d'une classe

7

- Repris de l'*Item 4* de [Bloch, 2008]
- Laisser la classe concrète et mettre le constructeur par défaut privé

Classe `seance6.creationdestructiondesobjets.ClasseNonInstanciable`

```
public class ClasseNonInstanciable {
        private ClasseNonInstanciable() {
3               throw new AssertionError();
        }
}
```

- Contrainte gardée dans les classes enfants

```
public class ClasseEnfantNonInstanciable extends ClasseNonInstanciable {
    private ClasseEnfantNonInstanciable(final String a) {...
    }
}
```

  - Donne le message d'erreur suivant à la compilation :
    - *Implicit super constructor ClasseNonInstanciable() is not visible. Must explicitly invoke another constructor*

8

1.5 Éviter la création inutile d'objet

- Repris de l'*Item 5* de [Bloch, 2008]
- Les classes de base proposent souvent des méthodes de classe ou des moyens pour réutiliser des objets créés plutôt que d'en instancier de nouveaux
  - P.ex. la classe `String` : « *A string literal always refers to the same instance of class String. This is because string literals —or, more generally, strings that are the values of constant expressions— are "interned" so as to share unique instances, using the method String.intern* » [Gosling et al., 2015, page 36]

Classe `seance6.creationdestructiondesobjets.EviterLaCreationInutileDObjet`

```
1                   for (int i = 0; i < 1000000000; i++) {
                            s = "un";
                    }
                    for (int i = 0; i < 1000000000; i++) {
5                           s = "un";
                    }
```

- Résultat sur ma machine : 33ms contre 2ms

1.6 Éviter le concept de *finalizer* des classes

9

- Repris de l'*Item 7* de [Bloch, 2008]
- « *If an object declares a finalizer, the finalizer is executed before the object is reclaimed to give the object a last chance to clean up resources that would not otherwise be released* » [Gosling et al., 2015, page 4]
- Mais :
  - « *The Java Virtual Machine may eventually automatically invoke its finalizer* » [Gosling et al., 2015, page 375]
    - Donc, aucune garantie que la méthode `finalize` soit appelée
      - ★ Par conséquent, ne pas mettre de code important dans un *finalizer*

Classe `seance6.creationdestructiondesobjets.ClasseAvecFinalizer`

```
1  public class ClasseAvecFinalizer {
       private int a;
       public ClasseAvecFinalizer(final int a) {
           this.a = a;
5      }
       @Override
       protected void finalize() {
           System.out.println("execution finalize de " + this);
9      }
       @Override
       public String toString() {
           return "ClasseAvecFinalizer [a=" + a + "]";
13     }
   }
```

Classe `seance6.creationdestructiondesobjets.ExempleClasseAvecFinalize`

```
   public class ExempleClasseAvecFinalize {
       @SuppressWarnings("unused")
       public static void main(String[] args) {
4          ClasseAvecFinalizer o = new ClasseAvecFinalizer(1);
       }
   }
```

L'exécution dans mon Eclipse ne donne aucun affichage.

2 Idiomes JAVA, exemples, méthodes de la classe `Object` communes à tous les objets

10

Nous commençons par revenir sur la méthode `equals` en donnant cette fois-ci des exemples de programmation de la méthode qui ne respectent pas les propriétés de symétrie et de transitivité ; ces exemples sont non triviaux mais importants : nous pensons que vous pouvez les comprendre aujourd'hui. Ensuite, nous introduisons une interface très utilisée pour cloner des objets, avec les concepts importants de copie légère et copie profonde.

11

2.1 Méthode `equals`

- Repris de l'*Item 8* de [Bloch, 2008]
- Contrat de la méthode `equals` [Class Object, JSE8]
  - Prend en argument un `Object` : utiliser `@Override` pour éviter une erreur
  - *Reflexive : for any non-null reference value `x`, `x.equals(x)` should return `true`*
  - *Symmetric : for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`*
  - *Transitive : for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`*
  - *Consistent : for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified*
  - *For any non-null reference value `x`, `x.equals(null)` should return `false`*

Dans les diapositives qui suivent, nous jouons aux plus malins en ne nous aidant pas de la génération des méthodes `equals` et `hashCode` avec notre IDE, mais en les programmant nous-mêmes, et de manière originale. Nous tombons alors dans des erreurs.

2.1.1 Exemple d'erreur sur la propriété de symétrie

12

Classe `seance7.methodescommunesatouslesobjets.EqualsErreurSymetrie`

```
    public EqualsErreurSymetrie(final String s) { this.s = s; }
    @Override public boolean equals(Object obj) {
      if (this == obj) {return true;} if (obj == null) {return false;}
4     EqualsErreurSymetrie other = (EqualsErreurSymetrie) obj;
      if (s == null) { if (other.s != null) return false; }
      else if (!s.equalsIgnoreCase(other.s)) return false; return true; } }
```

Classe `seance7.methodescommunesatouslesobjets.EqualsErreurSymetrieEnfant`

```
    public EqualsErreurSymetrieEnfant(final String s) { super(s); }
2   @Override public boolean equals(Object obj) {
      if (this == obj) {return true;} if (obj == null) {return false;}
      EqualsErreurSymetrie other = (EqualsErreurSymetrie) obj;
      if (s == null) { if (other.s != null) return false; }
6     else if (!s.equals(other.s)) return false; return true; } }
```

Classe `seance7.methodescommunesatouslesobjets.ExemplesMethodeEqualsErreurSymetrie`

```
    EqualsErreurSymetrie o1 = new EqualsErreurSymetrie("Avec");
    EqualsErreurSymetrieEnfant o2 = new EqualsErreurSymetrieEnfant("avec");
3   List<EqualsErreurSymetrie> l = new ArrayList<>(); l.add(o1);
    System.out.println(o2.equals(o1) + ", " + o1.equals(o2) + ", "+ l.contains(o2));
```

Le résultat de l'exécution est « `false, true, false` ».
Voici les codes complets.

Classe `seance7.methodescommunesatouslesobjets.EqualsErreurSymetrie`

```
1 package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

  public class EqualsErreurSymetrie {
    String s;
5   public EqualsErreurSymetrie(final String s) { this.s = s; }
    @Override public boolean equals(Object obj) {
      if (this == obj) {return true;} if (obj == null) {return false;}
      EqualsErreurSymetrie other = (EqualsErreurSymetrie) obj;
9     if (s == null) { if (other.s != null) return false; }
      else if (!s.equalsIgnoreCase(other.s)) return false; return true; } }
```

Classe `seance7.methodescommunesatouslesobjets.ExemplesMethodeEqualsErreurSymetrie`

```
  package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;
2
  import java.util.ArrayList;
  import java.util.List;

6 public class ExemplesMethodeEqualsErreurSymetrie {
    public static void main(String[] args) {
    EqualsErreurSymetrie o1 = new EqualsErreurSymetrie("Avec");
    EqualsErreurSymetrieEnfant o2 = new EqualsErreurSymetrieEnfant("avec");
10  List<EqualsErreurSymetrie> l = new ArrayList<>(); l.add(o1);
    System.out.println(o2.equals(o1) + ", " + o1.equals(o2) + ", "+ l.contains(o2));
    }
  }
```

La méthode `EqualsErreurSymetrie::equals` utilise la méthode `String::equalsIgnoreCase` alors que la méthode redéfinie dans la classe enfant `EqualsErreurSymetrieEnfant` utilise la méthode `String::equals`. Aussi, « `o2.equals(o1)` » utilise la méthode `EqualsErreurSymetrieEnfant::equals`, donc `String::equals`, et est évalué à « `false` ». Ensuite, « `o1.equals(o2)` » utilise la méthode `EqualsErreurSymetrie::equals`, donc `String::equalsIgnoreCase`, et est évaluée à « `true` ». Enfin, « `l.contains(o2)` » utilise la méthode `EqualsErreurSymetrieEnfant::equals`, donc `String::equals`, et est évalué à « `false` ».

2.1.2 Exemple d'erreur sur la propriété de transitivité

13

Classe `seance7.methodescommunesatouslesobjets.Parent`

```
public class Parent { private int a; // ...
```

Classe `seance7.methodescommunesatouslesobjets.EqualsErreurTransitivite`

```
public class EqualsErreurTransitivite extends Parent {
  private int b;
  public EqualsErreurTransitivite(final int a, final int b) {super(a);this.b = b;}
  @Override public boolean equals(Object obj) {
    if (this == obj) {return true;} if (obj == null) {return false;}
    if (obj.getClass().equals(Parent.class)) return super.equals(obj);
    EqualsErreurTransitivite other = (EqualsErreurTransitivite) obj;
    return super.equals(other) && b == other.b; } }
```

Classe `seance7.methodescommunesatouslesobjets.ExemplesMethodeEqualsErreurTransitivite`

```
  Parent p = new Parent(1);
  EqualsErreurTransitivite e1 = new EqualsErreurTransitivite(1, 2);
  EqualsErreurTransitivite e2 = new EqualsErreurTransitivite(1, 3);
  System.out.println(e1.equals(p) + ", " + p.equals(e2) + ", " + e1.equals(e2));
  }
```

■ Résultat de l'exécution : `true, true, false`

Le résultat de l'exécution est « `true, true, false` ».
Voici les codes complets.

Classe `seance7.methodescommunesatouslesobjets.Parent`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

public class Parent { private int a; // ...
  public Parent(final int a) { this.a = a;}
  @Override public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (!(obj instanceof Parent other)) return false;
    return a == other.a;} }
```

Classe `seance7.methodescommunesatouslesobjets.EqualsErreurTransitivite`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

public class EqualsErreurTransitivite extends Parent {
  private int b;
  public EqualsErreurTransitivite(final int a, final int b) {super(a);this.b = b;}
  @Override public boolean equals(Object obj) {
    if (this == obj) {return true;} if (obj == null) {return false;}
    if (obj.getClass().equals(Parent.class)) return super.equals(obj);
    EqualsErreurTransitivite other = (EqualsErreurTransitivite) obj;
    return super.equals(other) && b == other.b; } }
```

Classe `seance7.methodescommunesatouslesobjets.ExemplesMethodeEqualsErreurTransitivite`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

public class ExemplesMethodeEqualsErreurTransitivite {
  public static void main(String[] args) {
  Parent p = new Parent(1);
  EqualsErreurTransitivite e1 = new EqualsErreurTransitivite(1, 2);
  EqualsErreurTransitivite e2 = new EqualsErreurTransitivite(1, 3);
  System.out.println(e1.equals(p) + ", " + p.equals(e2) + ", " + e1.equals(e2));
  }
}
```

La classe `EqualsErreurTransitivite` étend la classe `Parent` et ajoute l'attribut « `b` » à l'attribut « `a` ». La méthode `EqualsErreurTransitivite::equals` se limite à la comparaison de l'attribut « `a` » si l'argument est un objet de type `Parent`, et compare les attributs « `a` » et « `b` » sinon. Si l'idée peut paraître séduisante de prime abord, l'exemple montre trivialement que la transitivité n'est pas respectée.

2.1.3 Leçons à retenir suite à l'étude des 2 exemples précédents

14

- Utiliser la génération Eclipse
  - Pose de l'annotation `@Override`
  - Évite l'utilisation de plusieurs transtypages (comme dans les exemples montrés ci-avant)
- Faire attention aux attributs pris en compte dans `equals` et qui changent de valeur
  - Commencer par les éviter
  - En cas de changement, que deviennent les collections les contenant ?
    - Étudier l'interaction avec `hashCode` (cf. diapositives à venir)
- Faire attention à la redéfinition dans les classes enfants
  - Si collection « sur la totalité de l'arbre d'héritage » alors commencer par redéfinir `equals` dans la classe parente sans redéfinition dans les classes enfants et utiliser l'opérateur `instanceof` au lieu de la méthode `getClass`
- Dans le cas exceptionnel et vraiment déconseillé où vous décidiez d'écrire vous-mêmes vos méthodes `equals` et `hashCode` faites des vérifications dans votre projet (Cf. diapositive qui suit)

2.1.4 Tests des propriétés de `equals`

15

Classe `seance7.methodescommunesatouslesobjets.Parent`

```
public class Parent { private int a; // ...
```

Classe `seance7.methodescommunesatouslesobjets.Enfant`

```
public class Enfant extends Parent { private int b;
  public Enfant(final int a, final int b) { super(a);this.b = b;} }
```

Classe `seance7.methodescommunesatouslesobjets.TestEquals`

```
public class TestEquals {
  private Parent p; private Enfant e, e2;
  @BeforeEach public void setUp() throws Exception {
    p = new Parent(1); e = new Enfant(1, 2); e2 = new Enfant(1,2); }
  @AfterEach public void tearDown() throws Exception {
    p = null; e = null; e2 = null; }
  @Test public void testReflexivite() {
    Assertions.assertTrue(p.equals(p) && e.equals(e) && e2.equals(e2)); }
  @Test public void testSymetrie() {
    Assertions.assertTrue((!p.equals(e)||e.equals(p))&&(!e.equals(p)||p.equals(e))); }
  @Test public void testTransitivity() {
    Assertions.assertTrue(e2.equals(p) && p.equals(e) && e2.equals(e)); } }
```

Voici les codes complets.

Classe `seance7.methodescommunesatouslesobjets.Parent`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

public class Parent { private int a; // ...
  public Parent(final int a) { this.a = a;}
  @Override public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (!(obj instanceof Parent other)) return false;
    return a == other.a;} }
```

Classe `seance7.methodescommunesatouslesobjets.Enfant`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

@SuppressWarnings("unused")
public class Enfant extends Parent { private int b;
  public Enfant(final int a, final int b) { super(a);this.b = b;} }
```

Classe `seance7.methodescommunesatouslesobjets.TestEquals`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class TestEquals {
  private Parent p; private Enfant e, e2;
  @BeforeEach public void setUp() throws Exception {
    p = new Parent(1); e = new Enfant(1, 2); e2 = new Enfant(1,2); }
  @AfterEach public void tearDown() throws Exception {
    p = null; e = null; e2 = null; }
  @Test public void testReflexivite() {
    Assertions.assertTrue(p.equals(p) && e.equals(e) && e2.equals(e2)); }
  @Test public void testSymetrie() {
    Assertions.assertTrue((!p.equals(e)||e.equals(p))&&(!e.equals(p)||p.equals(e))); }
  @Test public void testTransitivity() {
    Assertions.assertTrue(e2.equals(p) && p.equals(e) && e2.equals(e)); } }
```

2.2 Clonage d'objet, méthode `clone`

16

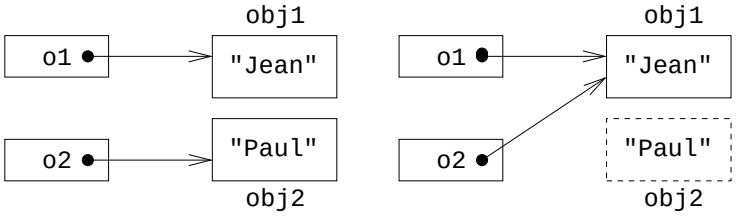
- Repris de l'*Item 11* de [Bloch, 2008]
- Particularité de cette méthode
  - Utilisation par une interface `Cloneable`[a]
- Avec la déclaration de mise en œuvre de l'interface
  - `clone` retourne un objet qui est une copie attribut par attribut de l'objet sur lequel elle est appelée
    - Cf. diapositives suivantes sur les concepts « clonage » et « copies légère et profonde »
- Sans la déclaration de mise en œuvre de l'interface
  - Levée de l'exception `CloneNotSupportedException`

a. La méthode `Clone` de `Object` est `protected`

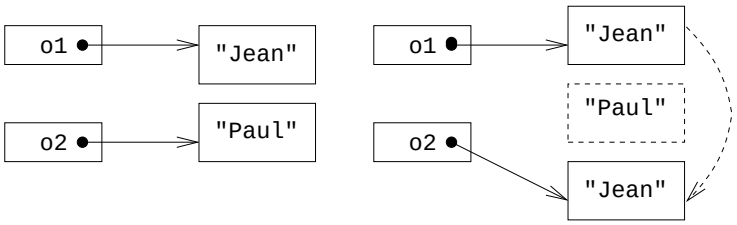
17

2.2.1 Concept du clonage

- Les objets sont manipulés par des références ⇒
  - `o2 = o1` : copie la référence `o1` dans `o2`



  - `o2 = o1.clone()` : `o2` référence un nouvel objet (copie de `o1`)



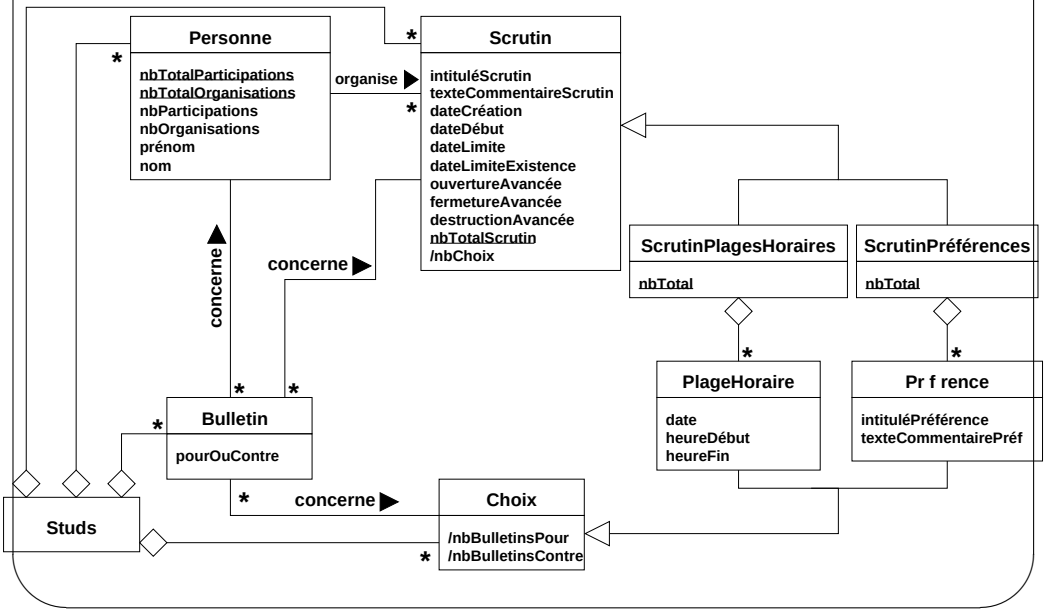
La classe `Object` propose une copie légère par un clonage basé sur la copie champ par champ réalisée par affectation : elle n'appelle pas récursivement la méthode `clone()` (pour cette raison, la copie est appelée légère). Cette opération de clonage par défaut n'est possible que si la classe réalise (*implements*) l'interface `java.lang.Cloneable`. Si ce n'est pas le cas, l'exception `CloneNotSupportedException` est levée. Cette copie champ par champ est suffisante pour avoir deux objets indépendants lorsque les objets copiés ne contiennent pas de référence.

Lorsque les objets contiennent des références, il faut définir la méthode `clone()` pour cette classe, mais aussi pour toutes les classes référencées par celle-ci, afin de réaliser une copie récursive dite « copie profonde ».

La méthode `clone` est `protected` dans la classe `Object` afin de restreindre l'utilisation de cette méthode aux classes et à leurs classes filles. Si le clonage est garanti (par la redéfinition de la méthode `clone`), celle-ci doit devenir publique.

2.2.2 Diagramme de classes exemple pour présenter le clonage

18

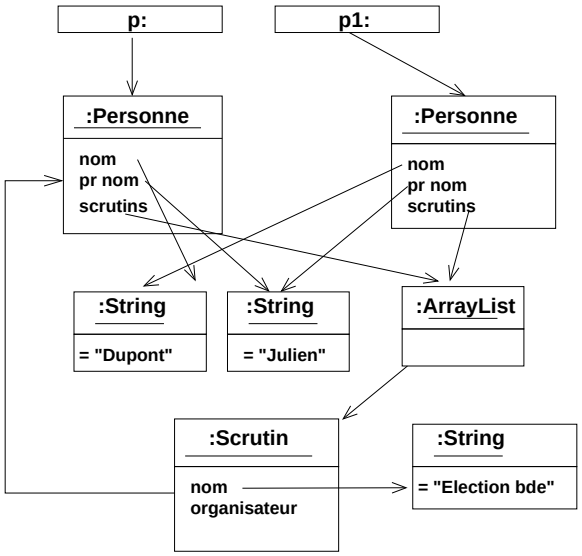


Nous voulons réaliser une méthode `clone` qui fonctionne pour la paire `Scrutin` et `Personne`.

La difficulté de ce clonage tient au fait que la relation est bidirectionnelle. Il faut donc être capable de dupliquer les deux objets et de ne pas avoir d'incohérence d'association entre ces objets.

2.2.3 Représentation graphique d'une copie légère

19



Classe `seance7.methodescommunesatouslesobjets.clonageleger.Personne`

```
1  package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageleger;
   import java.util.ArrayList;
   import java.util.List;

5  public class Personne implements Cloneable {
       private String nom, prenom;
       private int nbOrg = 0;
       private ArrayList<Scrutin> scrutins ;
9      public Personne(String nom, String prenom){
           this.nom = nom; this.prenom = prenom; scrutins = new ArrayList<Scrutin>(); }
       public Scrutin organiserScrutin(String nom) {
           scrutins.add(new Scrutin(nom,this)); return scrutins.get(nbOrg++); }
13     @Override
       public Personne clone() throws CloneNotSupportedException {
           return (Personne) super.clone();
       }
17     public String getNom(){return nom;}
       public String getPrenom(){return prenom;}
       public List<Scrutin> getScrutins(){return scrutins;}
       public void voter(Bulletin b){ }
21     public void consulterResultat(Scrutin s) { }
       public void seRetirerDUnScrutin(Scrutin s) { }
   }
```

Classe `seance7.methodescommunesatouslesobjets.clonageleger.ExempleClonageLeger`

```
1  package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageleger;

   import java.util.List;

5  public class ExempleClonageLeger {
       public static void main(final String[] args)
                   throws CloneNotSupportedException {
               Personne p = new Personne("Dupont", "Julien");
9              p.organiserScrutin("Election bde");
               Personne p1 = p.clone();
               if (p1 == p) {
                   System.out.println("p == p1");
13             }
               if (p.getNom().equals(p1.getNom())) {
                   System.out.println("p et p1 noms ==");
               }
17             if (p.getPrenom().equals(p1.getPrenom())) {
                   System.out.println("p et p1 prenoms ==");
               }
               List<Scrutin> a1, a2;
21             a1 = p.getScrutins();
               a2 = p1.getScrutins();
               if (a1 == a2) {
                   System.out.println("p et p1 scrutins ==");
25             } else {
```

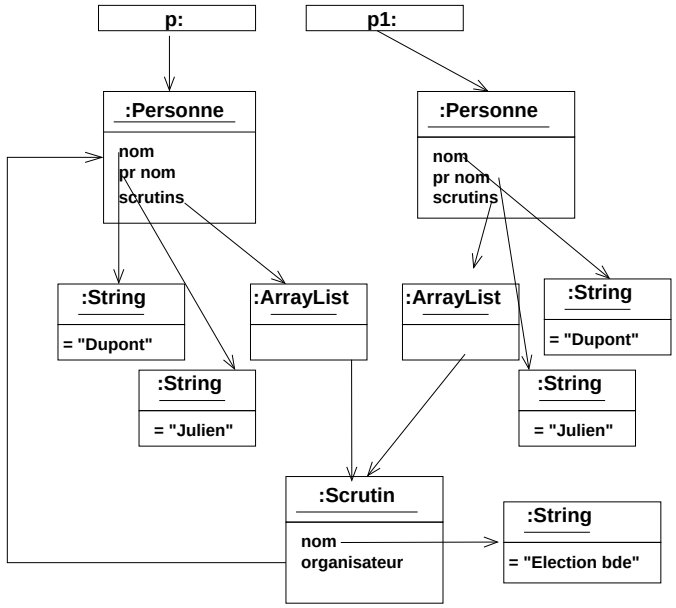
```
                        for (int i = 0; i < a1.size(); i++) {
                            if (a1.get(i) == a2.get(i)) {
                                System.out.println("Scrutin de rang " + i + " ==");
29                          }
                        }
                    }
                }
33  }
```

```
p et p1 noms ==
p et p1 prenoms ==
p et p1 scrutins ==
```

Dans cet exemple, la classe `Personne` implémente l'interface `Clonable` en réalisant le clonage par l'utilisation de la méthode `clone` héritée de la classe `Object`. L'objet obtenu est bien distinct de l'objet initial. Par contre, les objets référencés par les deux objets, à savoir le nom, le prénom et le tableau de `Scrutin` n'ont pas été dupliqués et donc ils sont partagés par les deux objets de type `Personne`.

2.2.4 Représentation graphique d'une copie plus profonde

20



Classe `seance7.methodescommunesatouslesobjets.clonageplusprofond.Personne`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageplusprofond;
import java.util.ArrayList;
import java.util.List;

public class Personne implements Cloneable {
    private String nom;
    private String prenom;
    private int nbParticipations = 0;
    private int nbOrg = 0;
    private ArrayList<Scrutin> scrutins;
    public Personne(final String n, final String p) {
        this.nom = n;
        this.prenom = p;
        scrutins = new ArrayList<>();
    }
    public Scrutin organiserScrutin(final String nom) {
        Scrutin s = new Scrutin(nom,this);
        scrutins.add(s);
        nbOrg ++;
        return s;
    }
    /** le clonage est profond, il duplique les objets references */
    @SuppressWarnings("unchecked")
    @Override
    public Personne clone() throws CloneNotSupportedException {
        Personne clone = new Personne(nom, prenom);
        clone.nbParticipations = nbParticipations;
        clone.nbOrg = nbOrg;
        clone.scrutins = (ArrayList<Scrutin>) scrutins.clone();
        return clone;
    }
    public String getNom() { return nom; }
    public String getPrenom() {return prenom; }
    public List<Scrutin> getScrutins() {return scrutins; }
    public void voter(final Bulletin b){ }
    public void consulterResultat(Scrutin s) { }
    public void seRetirerDUnScrutin(Scrutin s) { }
}
```

Classe `seance7.methodescommunesatouslesobjets.clonageplusprofond.ExempleClonagePlusProfond`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageplusprofond;
import java.util.List;
public class ExempleClonagePlusProfond {
    public static void main(final String[] args)
            throws CloneNotSupportedException {
        Personne p = new Personne("Dupont", "Julien");
        p.organiserScrutin("Election␣bde");
        Personne p1 = p.clone();
        if (p1 == p) {
            System.out.println("p␣==␣p1");
```

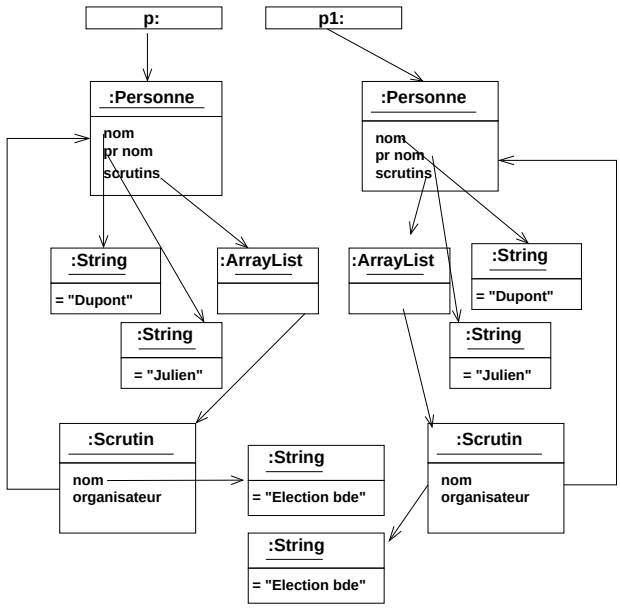
```
            }
            if (p.getNom() == p1.getNom()) {
                System.out.println("p et p1 noms ==");
            }
            if (p.getPrenom() == p1.getPrenom()) {
                System.out.println("p et p1 prenoms ==");
            }
            List<Scrutin> a1, a2;
            a1 = p.getScrutins();
            a2 = p1.getScrutins();
            if (a1 == a2) {
                System.out.println("p et p1 scrutins ==");
            } else {
                for(int i = 0; i < a1.size(); i++) {
                    if(a1.get(i) == a2.get(i)) {
                        System.out.println("Scrutins de rang " + i +" ==");
                    } else {
                        if(a1.get(i).getOrganisateur() == a2.get(i).getOrganisateur()) {
                            System.out.println("Oraganisateur des scrutins de rang" + i +"==");
                        }
                    }
                }
            }
        }
    }
```

Résultat de l'exécution :

```
p et p1 noms ==
p et p1 prenoms ==
Scrutins de rang 0 ==
```

2.2.5 Représentation graphique d'une copie profonde

21



Classe `seance7.methodescommunesatouslesobjets.clonageprofond.Scruptin`

```
1  package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageprofond;

   public class Scrutin implements Cloneable {
       private Personne organisateur;
5      private String nomScrutin;
       public Scrutin(final String nom, final Personne organisateur) {
           nomScrutin = nom;
           this.organisateur = organisateur;
9      }
       /** le clonage duplique le nom mais ne peut pas modifier l'organisateur.
        * En effet le clonage de l'organisation clone les scrutins qu'il a
        * organise et nous entrerions dans une cascade d'appels infinie
13      **/
       @Override
       public Scrutin clone() throws CloneNotSupportedException {
           return new Scrutin(nomScrutin, organisateur);
17     }
       /** Methode pour modifier l'organisateur en cas de clonage. */
       void setOrganisateur(final Personne norg) {
           organisateur = norg;
21     }
       public Personne getOrganisateur() {
           return organisateur;
       }
25 }
```

`Scrutin` contient deux attributs, une chaîne de caractères qui est clonable, et une référence sur l'organisateur du `Scrutin`. La méthode clone est définie entre les lignes 11 et 13. Elle fait appel à la méthode `clone` de la classe `Object` qui duplique les attributs de l'objet dans le clone.

Nous voulons aussi que le clone du `Scrutin` puisse être associé avec la `Personne` lorsque celui-ci aura été clonée. Pour cela, nous introduisons la méthode `setOrganisateur()`.

Classe `seance7.methodescommunesatouslesobjets.clonageprofond.Personne`

```
   package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageprofond;

3  import java.util.ArrayList;
   import java.util.List;

   public class Personne implements Cloneable {
7          private String nom;
           private String prenom;
           private int nbParticipations = 0;
           private int nbOrg = 0;
11         private ArrayList<Scrutin> scrutins;

           public Personne(final String n, final String p) {
                   this.nom = n;
```

```
15                  this.prenom = p;
                    scrutins = new ArrayList<>();
            }

19          public Scrutin organiserScrutin(final String nom) {
                    Scrutin s = new Scrutin(nom, this);
                    scrutins.add(s);
                    nbOrg++;
23                  return s;
            }

            /**
27           * le clonage est profond, il duplique les objets references. Ce n'est pas
             * necessaire pour les objets du type String du fait qu'un String est non
             * modifiable.
             */
31          @Override
            public Personne clone() throws CloneNotSupportedException {
                    Personne cl = new Personne(nom, prenom);
                    cl.nbParticipations = nbParticipations;
35                  cl.nbOrg = nbOrg;
                    for (Scrutin scr : scrutins) {
                            Scrutin sclone = scr.clone();
                            sclone.setOrganisateur(cl);
39                          cl.scrutins.add(sclone);
                    }
                    return cl;
            }
43
            public String getNom() {
                    return nom;
            }
47
            public String getPrenom() {
                    return prenom;
            }
51
            public List<Scrutin> getScrutins() {
                    return scrutins;
            }
55
            public void voter(final Bulletin b) {
            }

59          public void consulterResultat(Scrutin s) {
            }

            public void seRetirerDUnScrutin(Scrutin s) {
63          }
    }
```

Pour ce qui est de la personne, nous voulons la cloner et elle contient une liste des scrutins qu'elle a organisés. Il faut donc que nous dupliquions cette liste, ce qui est possible puisqu'elle implémente l'interface `Cloneable`. Il faut, cependant que la duplication soit profonde, c'est-à-dire que nous dupliquions les entrées de la liste.

Cette duplication profonde commence par cloner l'objet courant. Pour cela, elle crée un objet copie par appel d'un constructeur à la ligne 23, auquel elle passe les paramètres `nom` et `prenom`. Elle modifie ensuite les attributs `nbOrg` et `nbParticipations` en y mettant les valeurs des attributs correspondants dans l'objet courant. Ensuite, à la ligne 25, la méthode parcourt les scrutins un à un. Pour chaque scrutin, un clone du scrutin est construit à la ligne 26. L'organisateur de ce scrutin cloné est associé avec la personne en cours de clonage par la méthode `setOrganisateur`.

Classe `seance7.methodescommunesatouslesobjets.clonageprofond.ExempleClonageProfond`

```
package eu.telecomsudparis.csc4102.cours.seance7.methodescommunesatouslesobjets.clonageprofond;
import java.util.List;
public class ExempleClonageProfond {
4       public static void main(final String[] args)
                throws CloneNotSupportedException {
            Personne p = new Personne("Dupont", "Julien");
            p.organiserScrutin("Election bde 2010");
8           Personne p1 = p.clone();
            if (p1 == p) {
                System.out.println("p == p1");
            }
12          if (p.getNom() == p1.getNom()) {
                System.out.println("p et p1 noms ==");
            }
            if (p.getPrenom() == p1.getPrenom()) {
16              System.out.println("p et p1 prenoms ==");
```

```
            }
            List<Scrutin> a1, a2;
            a1 = p.getScrutins();
            a2 = p1.getScrutins();
            if (a1 == a2) {
                System.out.println("p et p1 scrutins ==");
            } else {
                for(int i = 0; i < a1.size(); i++) {
                    if(a1.get(i) == a2.get(i)) {
                        System.out.println("Scrutins de rang " + i +" ==");
                    } else {
                        if(a1.get(i).getOrganisateur() == a2.get(i).getOrganisateur()) {
                            System.out.println("Oraganisateur des scrutins de rang" + i +"==");
                        }
                    }
                }
            }
        }
    }
```

Résultat de l'exécution :

```
p et p1 noms ==
p et p1 prenoms ==
```

3 Classes immuables/inaltérables

- Repris de l'*Item 15* de [Bloch, 2008]
- Immuabilité/inaltérabilité (en anglais, `immutability`, `immutable objects`)
  - Une classe immuable est une classe dont les instances ne peuvent pas être modifiées
  - Que faire ?
    - Pas de méthode qui modifie l'état
      - ★ Pas d'accesseur
      - ★ Retourner une nouvelle instance lors des modifications (voir exemple)
    - Classe `final` ou constructeur privé avec méthode `static` pour construire les instances (voir exemple)
    - Tous les attributs `private`
    - Si des attributs sont des références
      - ★ Cloner les valeurs avant de les retourner
- Exemple prototypique = la classe `String`

C'est une bonne idée à tester dans vos codes que d'essayer d'écrire des classes immuables. Vous vous préparerez à la programmation concurrente, car une instance qui ne change pas de valeur est facilement partageable entre *threads*.

Étudiez l'exemple suivant.

Classe `seance7.classesditesimmuables.NombreComplexe`

```
1  package eu.telecomsudparis.csc4102.cours.seance7.classesditesimmuables;

   public class NombreComplexe {
           private final double partieReelle;
5          private final double partieImaginaire;
           public static final NombreComplexe ZERO = new NombreComplexe(0, 0);
           public static final NombreComplexe ONE = new NombreComplexe(1, 0);
           public static final NombreComplexe I = new NombreComplexe(0, 1);
9
           private NombreComplexe(final double partieReelle,
                           final double partieImaginaire) {
                   this.partieReelle = partieReelle;
13                 this.partieImaginaire = partieImaginaire;
           }

           public static NombreComplexe valueOf(final double partieReelle,
17                         final double partieImaginaire) {
                   return new NombreComplexe(partieReelle, partieImaginaire);
           }

21         public static NombreComplexe valueOfPolar(final double r,
                           final double theta) {
                   return new NombreComplexe(r * Math.cos(theta), r * Math.sin(theta));
           }
25
           public double partieReelle() {
                   return partieReelle;
           }
29
           public double partieImaginaire() {
                   return partieImaginaire;
           }
33
           public NombreComplexe ajouter(NombreComplexe c) {
                   return new NombreComplexe(partieReelle + c.partieReelle,
                                   partieImaginaire + c.partieImaginaire);
37         }

           public NombreComplexe soustraire(NombreComplexe c) {
                   return new NombreComplexe(partieReelle - c.partieReelle,
41                                 partieImaginaire - c.partieImaginaire);
           }
```

```
        public NombreComplexe multiplier(NombreComplexe c) {
45              return new NombreComplexe(
                                partieReelle * c.partieReelle
                                                - partieImaginaire * c.partieImaginaire,
                                partieReelle * c.partieImaginaire
49                                              + partieImaginaire * c.partieReelle);
        }

        public NombreComplexe diviser(NombreComplexe c) {
53              double tmp = c.partieReelle * c.partieReelle
                                + c.partieImaginaire * c.partieImaginaire;
                return new NombreComplexe(
                                (partieReelle * c.partieReelle
57                                              + partieImaginaire * c.partieImaginaire) / tmp,
                                (partieImaginaire * c.partieReelle
                                                - partieReelle * c.partieImaginaire) / tmp);
        }
61
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
65                      return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
69                      return false;
                NombreComplexe other = (NombreComplexe) obj;
                if (Double.doubleToLongBits(partieImaginaire) != Double
                                .doubleToLongBits(other.partieImaginaire))
73                      return false;
                return Double.doubleToLongBits(partieReelle) == Double
                                .doubleToLongBits(other.partieReelle);
        }
77
        @Override
        public int hashCode() {
                final int prime = 31;
81              int result = 1;
                long temp;
                temp = Double.doubleToLongBits(partieImaginaire);
                result = prime * result + (int) (temp ^ (temp >>> 32));
85              temp = Double.doubleToLongBits(partieReelle);
                result = prime * result + (int) (temp ^ (temp >>> 32));
                return result;
        }
89
        @Override
        public String toString() {
                return "(" + partieReelle + "␣+␣" + partieImaginaire + "i)";
93      }
    }
```

23

3.1 Cas des collections immuables

■ Lorsqu'une classe immuable contient un attribut de type collection

Classe seance7.classesditesimmuables.MauvaisFinalStaticTableau

```
protected static final String[] TABLEAU = {"un", "deux", "trois"};
```

Classe seance7.classesditesimmuables.MeilleursFinalStaticTableau

```
private static final String[] TABLEAU = { "un", "deux", "trois" };
public static final List<String> TAB = Collections.unmodifiableList(Arrays.asList(TABLEAU));
public String[] autreSolutionAvecGetterEtClone() { return TABLEAU.clone(); }
```

Classe seance7.classesditesimmuables.ExempleModificationTableauFinalStatic

```
    System.out.println(new MauvaisFinalStaticTableau());
2   MauvaisFinalStaticTableau.TABLEAU[0] = "deux";
    System.out.println(new MauvaisFinalStaticTableau());
    try { MeilleursFinalStaticTableau.TAB.remove(0); }
    catch (UnsupportedOperationException e) { System.out.println(e); }
6   MeilleursFinalStaticTableau m = new MeilleursFinalStaticTableau();
    String[] tab = m.autreSolutionAvecGetterEtClone(); tab[0] = "deux";
    System.out.println("tab[0] = " + tab[0]); System.out.println(m);
```

Voici ce que donne l'exécution :

```
MauvaisFinalStaticTableau [ un deux trois ]
MauvaisFinalStaticTableau [ deux deux trois ]
java.lang.UnsupportedOperationException
tab[0] = deux
MeilleurFinalStaticTableau [ un deux trois ]
```

Dans cet exemple, le programmeur souhaite mettre à disposition un tableau (une collection) constante : il faut que l'attribut soit `final` et que le contenu de la collection référencée soit lui-aussi non modifiable. L'exemple d'exécution montre que mettre l'attribut `final` n'est pas suffisant (classe `MauvaisFinalStaticTableau`). Il montre aussi qu'essayer de modifier une collection immuable génère une exception (classe `MeilleursFinalStaticTableau` et exception `UnsupportedOperationException`).

4 Pourquoi des *lambda expressions* JAVA ?*

Voici un scénario extrait du livre de M. Naftalin, « *Mastering Lambdas : Java Programming in a Multicore World* », Mc Graw Hill, Oracle Press, 2015, pour expliquer l'intérêt des *lambda expressions*

1. Commençons avec un code qui itère sur une collection d'objets modifiables

```
List<Point> pointList = Arrays.asList(new Point(1,2), new Point(2,4));

for(Point p : pointList) { p.translate(1,1); }
```

 # 24

La seconde ligne de ce code est traduite par le compilateur comme suit :

```
Iterator pointItr = pointList.iterator();

while (pointItr.hasNext()) { ((Point) pointItr.next()).translate(1,1); }
```

Problème : le compilateur génère du code qui sera toujours séquentiel

Nous souhaiterions une exécution qui puisse être parallèle à la demande, c'est-à-dire selon la mise en œuvre de la méthode `forEach` (qui suit)

D'où, nous souhaitons quelque chose comme ceci :

```
List<Point> pointList = Arrays.asList(new Point(1,2), new Point(2,4));

pointList.forEach(/*translation d'un point selon le vecteur (1,1)*/);
```

2. Solution (avec le patron de conception « Commande »)

```
// cette interface est définie dans java.lang (présentation simplifiée ici)

public interface Iterable<T> { ...

  default void forEach(Consumer<? super T> action) {

    for (T t : this) { action.accept(t); }

}}
```

 # 25

```
// cette interface est définie dans java.util.function

public interface Consumer<T> { void accept(T t); }
```

Avec la nouvelle interface `Consumer<T>`, le code précédent devient :

```
List<Point> pointList = Arrays.asList(new Point(1,2), new Point(2,4));

pointList.forEach(new Consumer<Point>() {

   public void accept(Point p) { p.translate(1,1); }

});
```

26

- Une curiosité = méthode `default` dans une interface
  - `https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`
- Une nouveauté : `Consumer<T>` est une interface qui possède une seule méthode : `void accept(T t)`
- Une difficulté de lecture = classe anonyme
  - `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`

27

3. **Remplacement de la classe anonyme par une *lambda expression***
   (a) Le compilateur devrait pouvoir inférer l'interface attendue par `forEach`

```
pointList.forEach(new Consumer<Point>() {

   public void accept(Point p) { p.translate(1,1); }

});
```

   (b) Avec le nom de l'interface attendue, le compilateur devrait pouvoir inférer le nom de la méthode

```
pointList.forEach(            Point

   public void accept(Point p) { p.translate(1,1); }

 );
```

   (c) Le type de l'argument peut alors être inféré du type appelant (`pointList`)

```
pointList.forEach(            Point

                     Point p    p.translate(1,1)

 );
```

   (d) **Avec une syntaxe particulière (->), cela donne une *lambda expression***

```
pointList.forEach(p -> p.translate(1, 1));
```

28

5 Différents types de références de méthodes

	Forme « *lambda expression* »	Forme « référence de méthode »	Type de méthode de référence
1	`str -> Integer.parseInt(str)`	`Integer::parseInt`	Statique
2	`Instant then = Instant.now(); t -> then.isAfter(t)`	`Instant.now()::isAfter`	Lié (*bound*)
3	`str -> str.toLowerCase()`	`String::toLowerCase`	Libre (*unbound*)
4	`() -> new TreeMap<K,V>`	`TreeMap<K,V>::new`	Constructeur de classe
5	`len -> int[len]`	`int[]::new`	Constructeur de tableau

- Les références de méthodes sont encore plus succinctes que les *lamddas*
  - Des outils comme `SpotBugs`[1] et `SonarLint`[2] dans `Eclipse` proposent systématiquement les remplacements
  - Lors du remplacement de la *lambda* par la référence, attention aux références de méthodes avec variables liées (#2) : variable « `then` » définie avant la définition de la *lambda expression Versus* méthode « `Instant.now()` » appelée dans la définition de la référence de méthode (cf. un exemple dans la diapositive qui suit)
- Exemples de références de méthodes

Classe `seance7/LambdaVersusReferenceDeMethode`

```
ToIntFunction<String> f1L = str -> Integer.parseInt(str);
ToIntFunction<String> f1RM = Integer::parseInt;
// essai de f1L et f1RM
System.out.println(f1L.applyAsInt("1") + "␣" + f1RM.applyAsInt("1"));
Instant then = Instant.now();
Predicate<Instant> f2L = t -> then.isAfter(t);
Predicate<Instant> f2RM = Instant.now()::isAfter;
// essai de f2L et f2RM + idem en lambdas
System.out.println(f2L.test(then) + "␣" + f2RM.test(then));
System.out.println(then.isAfter(then) + "␣" + Instant.now().isAfter(then));
UnaryOperator<String> f3L = str -> str.toLowerCase();
UnaryOperator<String> f3RM = String::toLowerCase;
// essai f3L et f3RM
System.out.println(f3L.apply("ABC") + "␣" + f3RM.apply("ABC"));
Supplier<TreeMap<String,String>> f4L = () -> new TreeMap<String,String>();
Supplier<TreeMap<String,String>> f4RM = TreeMap::new;
// essai f4L et f4RM
System.out.println(f4L.get() + "␣" + f4RM.get());
IntFunction<int[]> f5L = len -> new int[len];
IntFunction<int[]> f5RM = int[]::new;
// essaie f5L et f5RM
System.out.println(f5L.apply(4).length + "␣" + f5RM.apply(4).length);
```

Affichage : « `1 1` », puis « `false true` », « `false true` », « `abc abc` », « `{} {}` », et enfin « `4 4` »

1. `https://spotbugs.github.io/`
2. `https://www.sonarlint.org/eclipse/`

6 Pipeline et *stream*

29

6.1 Début du pipeline

30

- `IntStream`, etc. pour éviter les opérations de conversion entre le type conteneur `Integer` et le type primitif `int`
  + méthodes `generate`, `iterate`, `range`, `of`, etc. pour démarrer le pipeline
  - P.ex., `IntStream.iterate(1, i -> i * 2)`, `IntStream.range(7, 42)`
- Interface `Collection` maintenant avec les méthodes `stream` et `parallelStream` qui créent un *stream*
  - P.ex., `maListe.stream()`
  - Dans CSC4102, nous n'utiliserons pas de parallélisation
- Plus rarement, classe `Stream` avec les méthodes de classe `of`, `empty`, etc.
  - P.ex., `Stream.of(new Point(1, 2), new Point(2, 4))`

6.2 Traitements intermédiaires du pipeline

31

- Transformation = retourne un *stream* d'un autre type d'éléments
  - `Stream<R> map(Function<T,R> mapper)` et `flatMap`
  - `DoubleStream mapToDouble(ToDoubleFunction<T> mapper)`, etc.
- Filtrage = `Stream<T> filter(Predicate<T> predicate)`, `dropWhile`
- Combinaison de *streams* : `concat`
- Tri (`sorted`), suppression des doublons (`distinct`), troncature (`limit` et `skip`)
- Introspection pour le déverminage = même séquence en sortie + instertion d'un traitement avec `peek`

```
IntStream.iterate(1, i -> i*2).limit(10).peek(System.out::println).anyMatch(v -> v==128);
```

6.3 Terminaison d'un pipeline

32

- Recherche d'un élément et test de présence :
  - `Optional<T> findAny`[a], `findFirst`
  - `boolean anyMatch(Predicate<T>)`, `allMatch`, et `noneMatch`
- Réduction :
  - Calcul : `int sum()`, `Optional<T> min()`, `Optional<T> max()`, `long count()`, `Optional<T> average()`, `IntSummaryStatistics summaryStatistics`
  - Collecte : `R collect(Collector<T,A,R)` avec `T` le type des valeurs du *stream*, `R` le type de la valeur de retour, et `A` le type intermédiaire lors d'une exécution en parallèle
    - Bibliothèque d'algorithmes de `Collectors` :
      - `toSet`, `toList`, `toMap`, `toCollection`
      - `joining`, `groupingBy`, `partitioningBy`
- Application d'un effet de bord : `void forEach(Consumer<T>)`, `forEachOrdered`

a. Cf. section sur la classe `Optional` des diapositives

Pour les algorithmes de la classe `Collectors`, veuillez vous reporter à la page `https://docs.oracle.com/javase/9/docs/api/java/util/stream/Collectors.html`, qui est très bien faite avec des exemples clairs

Bibliographie

[Bloch, 2008] Bloch, J. (2008). *Effective Java, 2nd Edition*. Addison-Wesley.

[Class Collectors, JSE8] Class Collectors (JSE8). `Javadoc` of the class `java.util.stream.Collectors` of JAVA 9. `https://docs.oracle.com/javase/9/docs/api/java/util/stream/Collectors.html`.

[Class Object, JSE8] Class Object (JSE8). `Javadoc` of the class `Object` of JAVA SE 8. `https://docs.oracle.com/javase/9/docs/api/index.html?java/lang/Object.html`.

[Gosling et al., 2015] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2015). The Java Language Specification, Java SE 8 Edition. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.