Génie du Logiciel et des Systèmes

—

Introspection en Java

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Sciences du Numérique

Préambule

Exercice 1 Dessiner un diagramme de classe qui représente les principaux concepts présents dans le langage Java.

Sommaire

Sommaire

- JUnit 3 et sa classe TestRunner
- Lister les méthodes d'une classe
- Utilisations possibles
- Définitions

Exemple de classe de test avec JUnit 3

```
1  public class StringBufferTest extends junit.framework.TestCase {
2      private StringBuffer s1;                          // acteur
3
4      protected void setUp() {    // initialiser les données (acteurs)
5          s1 = new StringBuffer("Le texte");
6      }
7
8      public void testReverse() { // un premier test
9          s1.reverse();                                 // action
10         assertEquals("etxet eL", s1.toString());      // assertion
11     }
12
13     public void testDelete() {  // un deuxième test
14         s1.delete(2, 7);                              // action
15         assertEquals("Lee", s1.toString());           // assertion
16     }
17
18     public void testDernier() {
19         assertEquals('e', s1.charAt(-1));
20     }
21
22     public void testInitiale() {
23         assertEquals('L', s1.charAt(1));
24 }   }
```

Exécution de la classe de Test

```
java junit.textui.TestRunner StringBufferTest
```

```
1  ..F.E.
2  Time: 0,002
3  There was 1 error:
4  1) testDernier(StringBufferTest)java.lang.StringIndexOutOfBoundsException: index -1,length 8
5          at java.base/java.lang.String.checkIndex(String.java:3278)
6          at java.base/java.lang.AbstractStringBuilder.charAt(AbstractStringBuilder.java:307)
7          at java.base/java.lang.StringBuffer.charAt(StringBuffer.java:242)
8          at StringBufferTest.testDernier(StringBufferTest.java:19)
9          at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
10         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
11         at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
12 There was 1 failure:
13 1) testInitiale(StringBufferTest)junit.framework.AssertionFailedError: expected:<L> but was:<e>
14         at StringBufferTest.testInitiale(StringBufferTest.java:23)
15         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
16         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
17         at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
18
19 FAILURES!!!
20 Tests run: 4,  Failures: 1,  Errors: 1
```

Remarque : Les problèmes sont dans la classe de test et pas dans la classe String de Java !

Comment fonctionne la classe TestRunner de JUnit

Rappel du fonctionnement

Fonctionnement de la méthode principale de la classe junit.textui.TestRunner (de JUnit 3.8.1) :

- Elle prend en paramètre le nom d'une classe (StringBufferTest).
- Elle lance autant de tests (tests élémentaires) que StringBufferTest contient de méthodes qui commencent par test (`testReplace`, `testDelete`, etc.) en les comptabilisant en réussi (.), échec (F, Failure) ou erreur (E, Error).
- Elle exécute la méthode `setUp()` avant d'exécuter une méthode de test.
- Elle exécute la méthode `tearDown()` après avoir exécuté une méthode de test.
- Exemples d'utilisation :

```
junit.textui.TestRunner StringBufferTest
junit.textui.TestRunner PointTest
junit.textui.TestRunner PointNommeTest
```

Comment écrire la classe junit.textui.TestRunner de JUnit 3.8.1 ?

Exemple : classe TestRunner de JUnit

Principe de la solution

Il faut être capable de :

- **trouver et charger une classe à partir de son nom**
  par exemple, la classe StringBufferTest de nom "StringBufferTest"
- **accéder à ses méthodes** pour :
  - identifier celle qui s'appelle `setUp`, sans paramètre
  - identifier celle qui s'appelle `tearDown`, sans paramètre
  - identifier toutes les méthodes `test*`, sans paramètre
- **construire un objet de cette classe** de test (StringBufferTest)
- **appeler les méthodes identifiées sur cette instance**
  - dans l'ordre, `setUp`, une méthode de test puis `tearDown`

C'est ce que permettent :

- **introspection** : obtenir à l'exécution des informations sur la structure de l'application (classes, attributs, méthodes, etc.)
- et **intercession** : agir sur l'application grâce aux informations récupérées par introspection.

Découvrir les méthodes d'une classe

Objectif : Afficher les méthodes publiques des classes dont le nom est passé en argument de la ligne de commande, donc pas connues à la compilation.

```
1  public class AfficherMethodes {
2      /** Afficher les méthodes des classes dont les noms sont donnés.
3      * @param noms noms des classes */
4      public static void main(String[] noms) {
5          AfficherMethodes afficheur = new AfficherMethodes();
6          for (String nom: noms) {
7              afficheur.listerMethodes(nom);
8      }   }
9
10     /** Afficher la signature des méthodes de la classe nommée nomClasse.
11     * @param nomClasse le nom de la classe */
12     public void listerMethodes(String nomClasse) {
13         try {
14             Class<?> laClasse = Class.forName(nomClasse);
15             System.out.println("Méthodes de " + laClasse.getName() + " :");
16             for (java.lang.reflect.Method m : laClasse.getMethods()) {
17                 System.out.println(" - " + m);
18         }   }
19         catch (ClassNotFoundException e) {
20             System.out.println("!!! Classe inconnue : " + nomClasse);
21 }   }   }
```

Nouveau : La classe `Class` et ses méthodes forName, getName, getMethods.

Charger une classe et afficher ses méthodes

Utilisation : java AfficherMethodes PointNomme

```
1  Méthodes de PointNomme :
2   - public java.lang.String PointNomme.getNom()
3   - public void PointNomme.setNom(java.lang.String)
4   - public void PointNomme.afficher()
5   - public double Point.getX()
6   - public double Point.getY()
7   - public void Point.setX(double)
8   - public void Point.setY(double)
9   - public void Point.translater(double,double)
10  - public java.awt.Color Point.getCouleur()
11  - public void Point.setCouleur(java.awt.Color)
12  - public java.lang.String Point.toString()
13  - public double Point.distance(Point)
14  - public final native void java.lang.Object.wait(long) throws java.lang.InterruptedExceptio
15  - public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
16  - public final void java.lang.Object.wait() throws java.lang.InterruptedException
17  - public boolean java.lang.Object.equals(java.lang.Object)
18  - public native int java.lang.Object.hashCode()
19  - public final native java.lang.Class java.lang.Object.getClass()
20  - public final native void java.lang.Object.notify()
21  - public final native void java.lang.Object.notifyAll()
```

Quelques utilisations possibles

- **Navigateur de classes :**
  doit fonctionner avec les classes des nouvelles applications !
- **Débogueur :**
  afficher à l'utilisateur l'état du programme en fonction des éléments écrits dans le programme (classes, méthodes, etc.)
- **Environnement de développement d'interface graphique :**
  avec la possibilité d'ajouter son jeu de composants graphiques.
- **Outil de développement**, par exemple Bluej (www.bluej.org).
- Tout programme qui doit manipuler les caractéristiques spécifiques de classes qui ne sont pas encore connues au moment de son exécution (JUnit, etc.)

Définitions

Introspection : Interroger dynamiquement les objets d'une application pour retrouver la structure de l'application :

- les classes,
- les attributs,
- les méthodes,
- les constructeurs
- ...

Intercession : Modifier l'état d'une application en s'appuyant sur les informations obtenues par introspection.

En anglais : *introspection* et *reflection*

Références

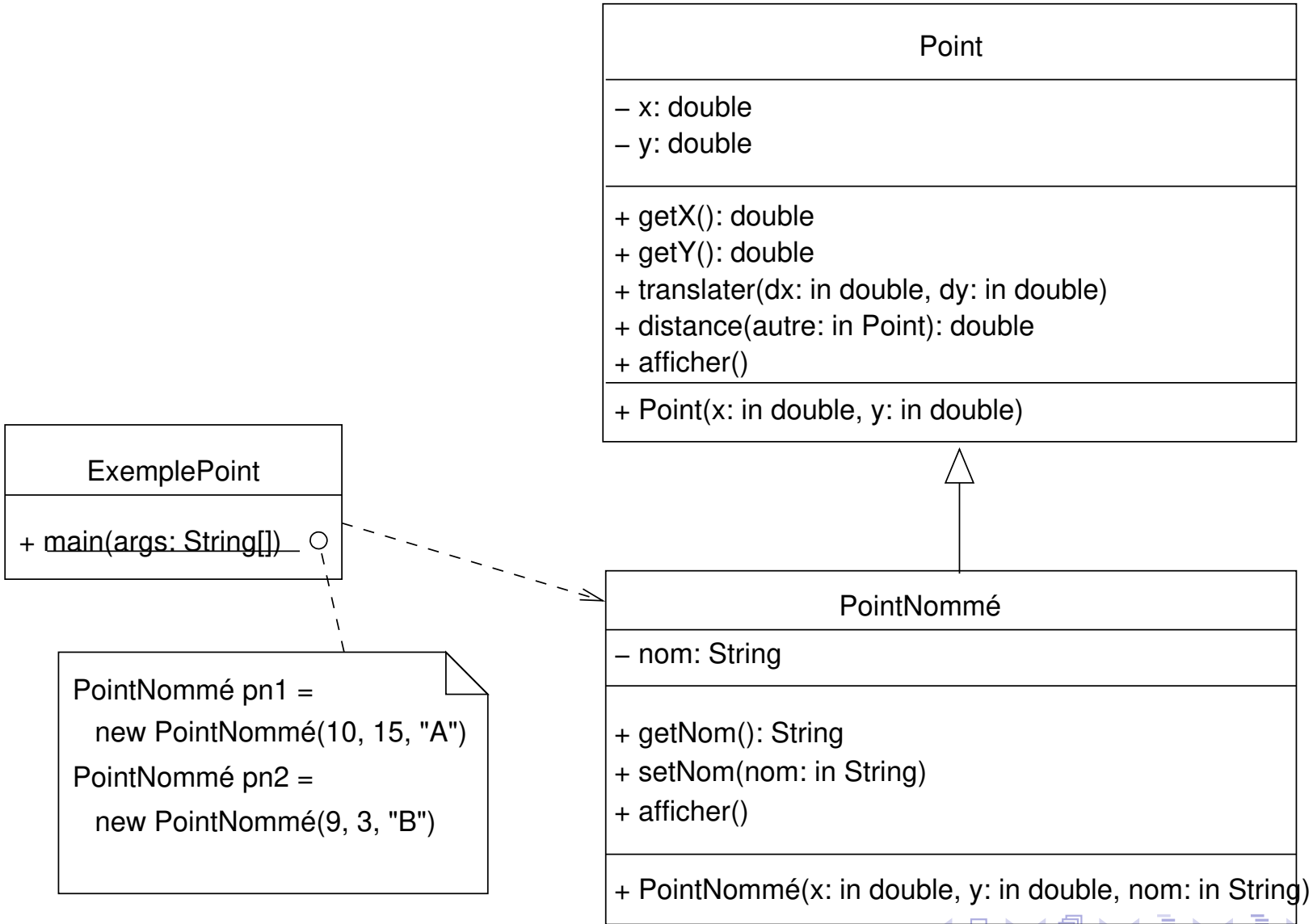
[1] C. NFP121, "Réflexivité." `http://jfod.cnam.fr/NFP121/Intro/`.

[2] C. S. Horstmann and G. Cornell, *Au cœur de Java 2*, vol. 1 Notions fondamentales. Campus Press, 8 ed., 2008.

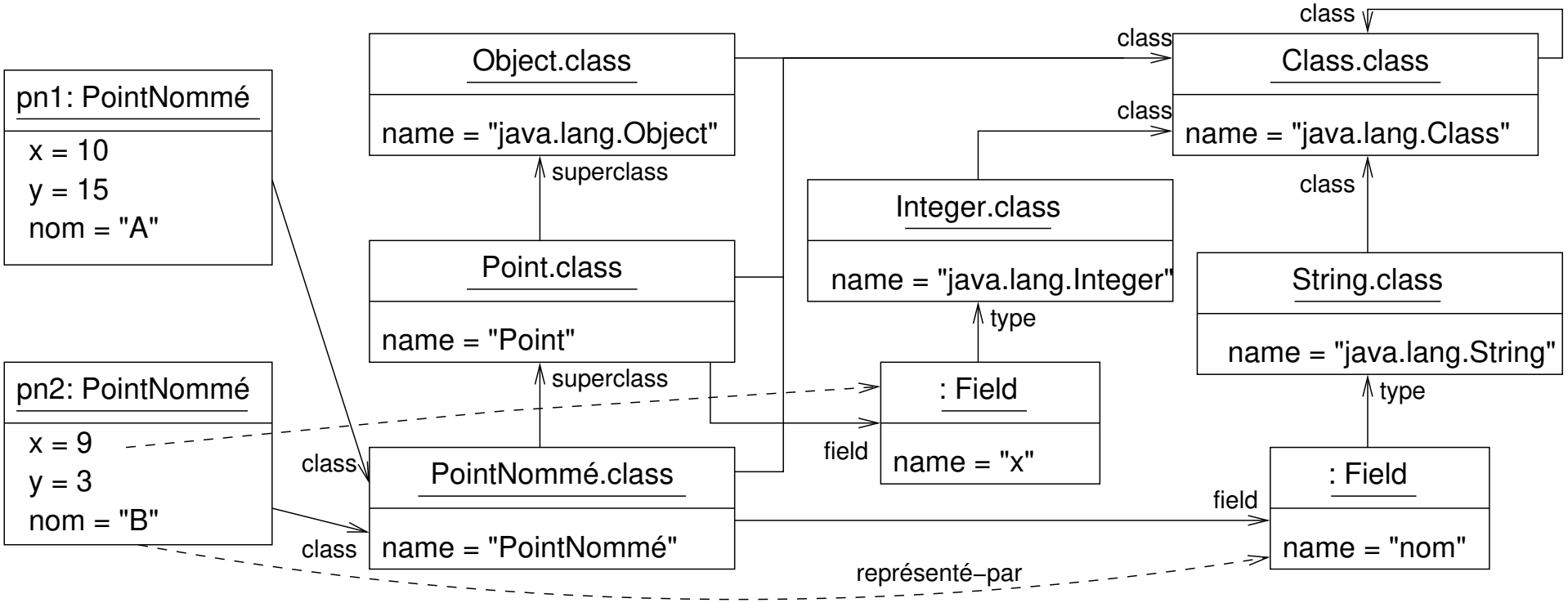
Sommaire

- Exemple d'application
- Architecture du paquetage `java.lang.Reflect`
- Obtenir un objet de type Class
- Charger de nouvelles classes
- Informations sur le type à l'exécution
- Informations liées aux supertypes
- Récupérer les autres membres de la classes
- Accessibilité

Les classes Point et PointNommé

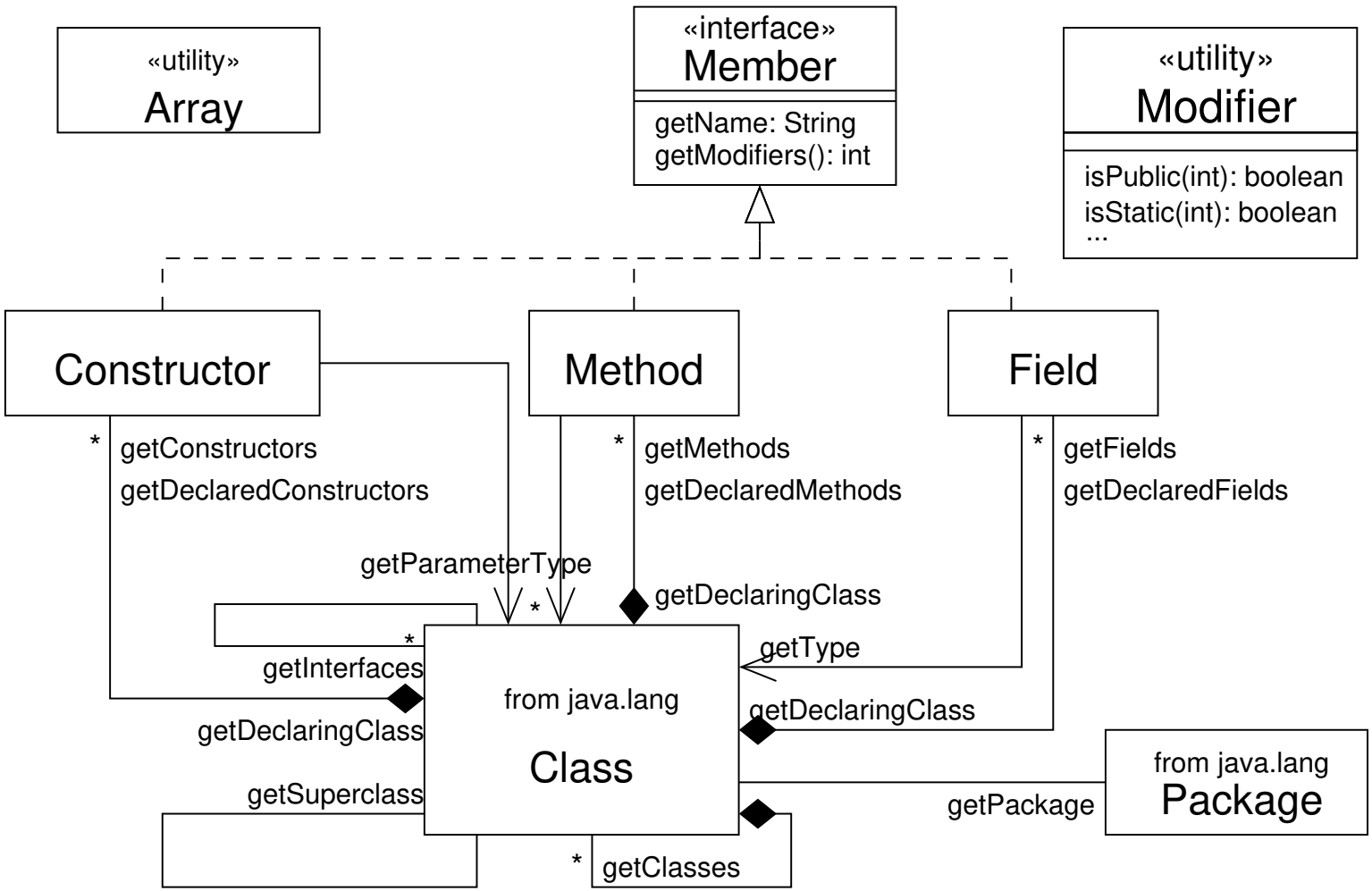


Les objets à l'exécution



- deux objets pn1 et pn2
- avec leurs propres états
- dont la classe est PointNommé
- ayant un attribut appelé "nom"
- de type String
- ... et des méthodes (non représentées) ...
- PointNommé a pour superclasse Point
- Point a pour superclasse Object

Diagramme de classe (simplifié) de java.lang.Reflect



Remarque : La classe Class est centrale !

Obtenir un objet de type Class

```
1  /** Illustrer différents moyens d'obtenir un objet Class */
2  public class ObtenirUnObjetClass {
3      public static void main(String[] args) throws ClassNotFoundException {
4
5          // à partir d'une instance
6          Point p1 = new Point(1, 2);
7          Class<?> c1 = p1.getClass();
8          System.out.println("c1 = " + c1);
9          System.out.println("\"chaine\".getClass() = " + "chaine".getClass());
10
11         // à partir d'une classe
12         Class<PointNomme> c2 = PointNomme.class;
13         System.out.println("c2 = " + c2);
14         System.out.println("int.class = " + int.class);
15         System.out.println("int[].class = " + int[].class);
16         System.out.println("double[].class = " + double[].class);
17         System.out.println("Integer.class = " + Integer.class);
18         System.out.println("Integer[].class = " + Integer[].class);
19         System.out.println("java.util.Iterator.class = " + java.util.Iterator.class);
20
21         // à partir du chargeur de classe (donc pas connu à la compilation !)
22         Class<?> c3 = Class.forName("java.lang.Float");
23         System.out.println("c3 = " + c3);
24
25         // attention à donner le nom qualifié !
26         Class<?> c4 = Class.forName("Float");
27 }   }
```

Obtenir un objet de type Class

Résultat de l'exécution

```
1  c1 = class Point
2  "chaine".getClass() = class java.lang.String
3  c2 = class PointNomme
4  int.class = int
5  int[].class = class [I
6  double[].class = class [D
7  Integer.class = class java.lang.Integer
8  Integer[].class = class [Ljava.lang.Integer;
9  java.util.Iterator.class = interface java.util.Iterator
10 c3 = class java.lang.Float
11 Exception in thread "main" java.lang.ClassNotFoundException: Float
12         at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.
               java:581)
13         at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(
               ClassLoaders.java:178)
14         at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
15         at java.base/java.lang.Class.forName0(Native Method)
16         at java.base/java.lang.Class.forName(Class.java:315)
17         at ObtenirUnObjetClass.main(ObtenirUnObjetClass.java:26)
```

Synthèse

Un objet de type `Class` représente un type du langage Java.

Class est générique : Class<T> où T est le type que représente cet objet `Class` !

On peut obtenir un objet de type `Class` à partir :

1. d'un objet et utilisant la méthode `getClass()` définie dans `Object`

   ```
   p1.getClass()
   ```

2. d'un type en utilisant l'attribut **`class`**

   ```
   Point.class     // la classe Point doit être connue à la compilation !
   ```

3. du nom du type (String) grâce à la méthode de classe `forName` de `Class`

   ```
   Class.forName("Point")  // éventuellement ClassNotFoundException !
   ```

Remarque : Le code suivant ne compile pas... car p1 pourrait être un point nommé !

```
Point p1 = new Point(1, 2);
Class<Point> c1 = p1.getClass();                 // ERREUR
Class<? extends Point> c1 = p1.getClass();       // OK
```

Charger de nouvelles classes

```
static Class<?> forName(String nomQualifié) throws ClassNotFoundException
        // charger une classe à partir de son nom (qualifié) en utilisant
        // le chargeur courant (getClassLoader())
```

Intérêt : Charger dynamiquement des classes inconnues à la compilation.

Remarque : On ne peut pas connaître la valeur du paramètre de généricité donc `Class<?>`

Exemples :

- JUnit pour charger la classe de Test
- ...
- Une application graphique qui permet à l'utilisateur de donner le nom d'une classe (par exemple Math), propose ensuite dans une liste déroulante les méthodes de classe à deux paramètres réels de cette classe, et deux champs de saisie pour les deux opérandes effectifs.

Informations sur le type à l'exécution : isInstance et cast

```
1  public class TesterLeType {
2      public static void main(String[] args) {
3          Point p = new PointNomme(1, 2, "A");
4
5          // Classique : avec l'opérateur instanceof
6          if (p instanceof PointNomme) {
7              PointNomme pn = (PointNomme) p;
8              System.out.println("nom = " + pn.getNom());
9          }
10
11         // en utilisant isInstance() et cast
12         if (PointNomme.class.isInstance(p)) {
13             PointNomme pn = PointNomme.class.cast(p);
14             System.out.println("nom = " + pn.getNom());
15         }
16
17         // en utilisant getClass() (Attention : égalité stricte du type)
18         if (p.getClass() == PointNomme.class) {
19             // ...
20 }   }   }
```

Obtenir les supertypes : super classe et interfaces réalisées

- getSuperclass() : obtenir la superclasse
- getInterfaces() : obtenir le tableau des interfaces directement réalisées

Exercice : Lister les super-types directs d'une classe (sans tenir compte de la transitivité de la relation de sous-typage).

```
1  public class ListerSuperTypesDirects {
2      static void listerSuperTypes(Class c) {
3          System.out.println();
4          System.out.println(c);
5          System.out.println("␣-␣Super␣classe␣:␣" + c.getSuperclass());
6          System.out.println("␣-␣Interfaces␣:␣");
7          for (Class<?> type : c.getInterfaces()) {
8              System.out.println("␣␣␣␣␣-␣" + type);
9      }   }
10
11     public static void main(String[] args) {
12         listerSuperTypes(PointNomme.class);
13         listerSuperTypes(java.util.List.class);
14         listerSuperTypes(java.util.ArrayList.class);
15         listerSuperTypes(Object.class);
16         listerSuperTypes(int.class);
17 }   }
```

Obtenir les supertypes : superclasse et interfaces réalisées

Résultat de l'exécution

```
1  class PointNomme
2   - Super classe : class Point
3   - Interfaces :
4
5  interface java.util.List
6   - Super classe : null
7   - Interfaces :
8       - interface java.util.Collection
9
10 class java.util.ArrayList
11  - Super classe : class java.util.AbstractList
12  - Interfaces :
13      - interface java.util.List
14      - interface java.util.RandomAccess
15      - interface java.lang.Cloneable
16      - interface java.io.Serializable
17
18 class java.lang.Object
19  - Super classe : null
20  - Interfaces :
21
22 int
23  - Super classe : null
24  - Interfaces :
```

Obtenir les méthodes de la classe

Récupérer toutes les méthodes :

- getMethods : toutes les méthodes publiques de la classe (y compris héritées)
- getDeclaredMethods : toutes les méthodes déclarées dans la classe (quelque soit le droit d'accès). Donc pas les méthodes héritées.

Récupérer une seule méthode : getMethod et getDeclaredMethod
Il faut préciser la signature de la méthode :

- le nom de la méthode
- le type des paramètres
  - soit un tableau de Class
  - soit en énumérant les types (java 1.5)

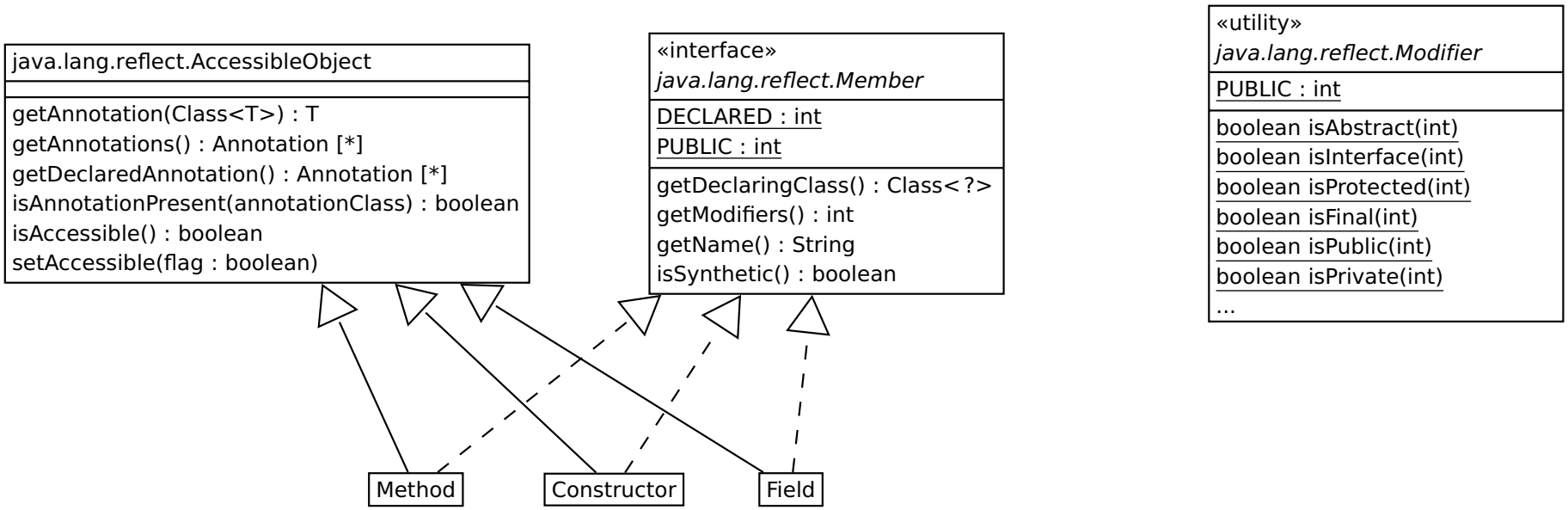
Même principe pour obtenir les **constructeurs** ou les **attributs** d'une classe (Method est remplacé par Constructor ou Field).

Exemple d'utilisation

```
1  public class GetMethodes {
2      public static void main(String[] args) throws NoSuchMethodException {
3          Point p = new PointNomme(1, 2, "A");
4          Class<?> cp = p.getClass(); // l'objet Class de p
5          // afficher toutes les méthodes déclarées dans cp
6          for (java.lang.reflect.Method m : cp.getDeclaredMethods())
7              System.out.println("␣-␣" + m);
8
9          // récupérer une méthode particulière (java >= 1.5)
10         java.lang.reflect.Method tr =
11         cp.getMethod("translater", double.class, double.class);
12         System.out.println("tr␣=␣" + tr);
13
14         // ou en construisant explicitement le tableau des paramètres
15         Class<?>[] parametres = { double.class, double.class };
16         assert tr.equals( cp.getMethod("translater", parametres) );
17
18         // et pour une méthode sans paramètre
19         java.lang.reflect.Method aff = cp.getDeclaredMethod("afficher");
20         System.out.println("aff␣=␣" + aff);
21         assert aff.equals( cp.getDeclaredMethod("afficher", new Class<?>[] {}) );
22 }   }
```

```
1  - public java.lang.String PointNomme.getNom()
2  - public void PointNomme.setNom(java.lang.String)
3  - public void PointNomme.afficher()
4  tr = public void Point.translater(double,double)
5  aff = public void PointNomme.afficher()
```

AccessibleObject et Member



- `getModifiers()` :
  - retrouver les modifieurs d'un membre (droit d'accès, **`final`**, **`static`**)
  - Voir `java.lang.reflect.Modifier`.
- `setAccessible()` :
  - rendre des membres privés accessibles (par introspection !)
  - Utilisé par exemple pour la sérialisation

Sommaire

Principe

Définition : L'intercession est le fait d'exploiter les informations récupérées par introspection pour agir sur l'état du programme.

Opérations possibles :

- Créer un objet : **`newInstance`** (à partir d'un constructeur, `Constructor`)
- Manipuler un attribut (`Field`) : le modifier (**`set`**) et accéder à sa valeur (**`get`**)
- Appeler une méthode (`Method`) : **`invoke`**

Remarque : Jusqu'à Java8, il était possible de créer un objet à partir d'une classe mais la méthode newInstance de Class est marquée obsolète depuis Java9.

Exemple d'intercession

```
1  import java.lang.reflect.*;
2
3  class A {
4      int n;
5      public void ajouter(int increment) {
6          n = n + increment;
7  }   }
8
9  public class ExempleIntercession {
10     public static void main(String... args) throws NoSuchMethodException, NoSuchFieldException,
11             IllegalAccessException, InstantiationException, InvocationTargetException
12     {
13         //---- classique ----------     ------ par introspection/intercession ----------
14                                         Class<A> laClasse = A.class;
15         // créer un objet ---------     ------------------------------------------------------------
16                                         Constructor<A> cnstrDefaut = laClasse.getDeclaredConstructor();
17         A a1 = new A();                 A a2 = cnstrDefaut.newInstance();
18         // modifier un attribut --      ------------------------------------------------------------
19                                         Field fieldN = laClasse.getDeclaredField("n");
20         a1.n = 5;                       fieldN.setInt(a2, 5);
21         // appeler un méthode ----      ------------------------------------------------------------
22                                         Method mInc = laClasse.getMethod("ajouter", int.class);
23         a1.ajouter(2);                  mInc.invoke(a2, 2);
24         // accéder à un attribut -      ------------------------------------------------------------
25         int v1 = a1.n;                  int v2 = fieldN.getInt(a2);
26
27         System.out.println(v1);         System.out.println(v2);
28 }   }
```

Sommaire

- La classe `java.lang.reflect.Method`
- La classe `java.lang.reflect.Constructor`
- La classe `java.lang.reflect.Field`
- La classe `java.lang.Class`
- La classe `java.lang.reflect.Array`

La classe java.lang.reflect.Method

```
1  Class<?> getDeclaringClass()    // la classe déclarant la méthode
2
3  // signature de la méthode
4  String getName()                // le nom de la méthode
5  Class<?>[] getParameterTypes()  // les types des paramètres
6  Class<?> getReturnType()        // le type de retour
7  Class<?>[] getExceptionTypes()  // les exceptions propagées
8  boolean isVarArgs()             // à nombre variable d'argument ?
9  int getModifiers()              // Modifieurs : droit d'accès, final, abstraite...
10
11 boolean isSynthetic()           // engendrée par le compilateur ?
12 boolean isBridge()              // lié à la généricité
13
14 Object invoke(Object recepteur,Object[] paramètres)
15     throws IllegalAccessException, IllegalArgumentException,
16                                 InvocationTargetException
17
18 // d'autres liées à la généricité, à la sécurité...
```

La classe java.lang.reflect.Constructor<T>

```
1  Class<T> getDeclaringClass()    // la classe déclarant le constructeur
2
3  // signature du constructeur
4  Class<?>[] getParameterTypes()  // les types des paramètres
5  Class<?>[] getExceptionTypes()  // les exceptions propagées
6  boolean isVarArgs()             // à nombre variable d'argument ?
7  int getModifiers()              // Modifiers : droit d'accès, final, abstraite...
8
9  boolean isSynthetic()           // engendré par le compilateur ?
10 boolean isBridge()              // lié à la généricité
11
12 Object newInstance(Object...) throws InstantiationException,
13     IllegalAccessException, IllegalArgumentException, InvocationTargetException
14
15 // d'autres liées à la généricité, à la sécurité...
```

La classe java.lang.reflect.Field

```
1  int getModifiers()              // Modifieurs : droit d'accès, final...
2  String getName()                // le nom de l'attribut
3  Class<?> getType()              // le type de l'attribut
4  boolean isEnumConstant()        // constante d'un type énuméré ?
5  Class<?> getDeclaringClass()    // la classe déclarant l'attribut
6  boolean isSynthetic()           // engendré par le compilateur ?
7
8  // obtenir la valeur d'un attribut
9  Object get(Object récepteur) throws IllegalArgumentException,IllegalAccessException
10 ttt getTtt(Object récepteur)
11         throws IllegalArgumentException,IllegalAccessException
12     // avec ttt = boolean, byte, short, char, int, long, float, double
13
14 // changer la valeur d'un attribut
15 public void set(Object récepteur, Object valeur)
16     throws IllegalArgumentException,IllegalAccessException
17 public void setTtt(Object récepteur, ttt valeur)
18     throws IllegalArgumentException,IllegalAccessException
```

La classe java.lang.Class

Principe : Pour chaque classe chargée par le chargeur de classe (ClassLoader), il existe un objet de type Class qui en décrit les propriétés.

- Cette classe est générique : Class<T>
  - T est le type représenté par cette objet Class.

```
1  // identification de la classe
2  String getName()                // nom de la classe (qualifié)
3  String getSimpleName()          // nom tel qu'il apparaît dans le source
4  String getCanonicalName()       // nom canonique d'après JLS
5
6  int getModifiers()              // Modifieurs : droit d'accès, final, abstraite...
7
8  // Quelle est la nature de la classe ?
9  boolean isArray()       // un tableau ?
10 boolean isEnum()        // un type énuméré ?
11 boolean isPrimitive()   // un type primitif ?
12 boolean isInterface()   // une interface ?
13 boolean isMemberClass() // une classe membre (interne non statique)
14 boolean isLocalClass()  // une classe locale (interne statique)
15 boolean isSynthetic()   // engendrée par le compilateur ?
```

La classe java.lang.Class (2)

```
16 // Chargement d'une classe
17 ClassLoader getClassLoader()    // chargeur de cette classe
18 static Class<?> forName(String) throws ClassNotFoundException
19             // charger une classe à partir de son nom (qualifié) en utilisant
20             // le chargeur courant (getClassLoader())
21
22 // contexte de la classe
23 Package getPackage()                    // dans un paquetage
24 Class<?> getEnclosingClass()            // classe interne
25 Constructor getEnclosingConstructor()   // classe locale ou anonyme
26 Method getEnclosingMethod()             // classe locale ou anonyme
27 Class<?> getDeclaringClass()            // classe dont this est membre
28
29 // Accès au contenu
30 Class<?>[] getClasses() // toutes les classes ou interfaces publiques internes
31 Class<?>[] getDeclaredClasses() throws SecurityException
32                 // les classes ou interfaces déclarées dans cette classe
33 Object[] getEnumConstants()     // les constantes d'un type énuméré (isEnum)
34 Class<?> getComponentType()     // type des éléments du tableau (isArray)
```

La classe java.lang.Class (3)

```
35 // Accès aux attributs
36 Field getField(String) throws NoSuchFieldException,SecurityException
37 Field[] getFields() throws SecurityException
38 Field getDeclaredField(String) throws NoSuchFieldException,SecurityException
39 Field[] getDeclaredFields() throws SecurityException
40
41 // Accès aux méthodes
42 Method[] getMethods() throws SecurityException
43 Method getMethod(String,Class<?>...) throws NoSuchMethodException,SecurityException
44 Method getDeclaredMethod(String,Class<?>...) throws NoSuchMethodException,SecurityException
45 Method[] getDeclaredMethods() throws SecurityException
46
47 // Accès aux constructeurs
48 Constructor getConstructor(Class<?>...) throws NoSuchMethodException,SecurityException
49 Constructor[] getConstructors() throws SecurityException
50 Constructor getDeclaredConstructor(Class<?>...) throws NoSuchMethodException,SecurityExceptio
51 Constructor[] getDeclaredConstructors() throws SecurityException
```

La classe java.lang.Class (4)

```
52 // Sous-typage
53 Class<?>[] getInterfaces()      // les interfaces que cette classe réalise
54 Class<?> getSuperclass()        // superclasse ou null si Object interface primitif ou void
55 boolean isInstance(Object)      // équivalent de instanceof
56 T cast(Object)  // transtype l'objet en utilisant le type this
57 boolean isAssignableFrom(Class<?>) // un objet de type this peut-il être initialisé
58                                   // avec un objet dont le type est le paramètre ?
59
60 // Créer une nouvelle instance de cette classe (obsolète depuis Java9)
61 Object newInstance() throws InstantiationException,IllegalAccessException
62
63 // Récupérer les informations de généricité
64 Type[] getGenericInterfaces()
65 Type getGenericSuperclass()
66 <U> Class<? extends U> asSubclass(Class<U>)     // transtype this avec le type en paramètre
67 TypeVariable[] getTypeParameters()
68
69 // lié aux annotations
70 ...
```

La classe java.lang.reflect.Array

But : Manipuler ou créer un tableau par introspection.

```
1  static int getLength(Object tab)        // équivalent de tab.length
2
3  static Object get(Object tab, int i);   // l'objet à l'indice i du tableau tab : tab[i]
4  static ttt getTtt(Object tab, int i);   // idem avec types de base : Ttt = int, etc.
5
6  static void set(Object tab, int i, Object v);   // Réalise : tab[i] = v
7  static void setTtt(Object tab, int i, Ttt v);   // idem avec types de base : Ttt = int, etc.
8
9  static Object newInstance(Class<?> t,int c);
10                 // Crée un tableau d'éléments du type t de capacité c
11                 // Exemple : Array.newInstance(int.class, 10) ~= new int [10]
12
13 static Object newInstance(Class<?> t,int... c);
14                 // Crée un tableau d'éléments du type t
15                 // à c.length dimensions de capacités données par c
16                 // Exemple : Array.newInstance(int.class, 10, 20) ~= new int [10][20]
```

Remarque : Pour avoir le type des éléments d'un tableau tab : `tab.getClass().getComponentType()`

Sommaire

- Créer un objet
- Incrémenter une propriété JavaBean de type int
- Tester des méthodes privées
- Structure de données avec contrôle de type
- Agrandir un tableau

Créer un objet : Class.newInstance

```
1  public class IntrospectionCreationNewInstance {
2      public static void main(String[] args)
3              throws InstantiationException, IllegalAccessException {
4          // avec Class.newInstance (via le constructeur par défaut)
5          Class<java.util.Date> cDate1 = java.util.Date.class;
6          java.util.Date d1 = cDate1.newInstance();   // d1 du bon type !
7          System.out.println("d1 = " + d1);
8
9          Class<?> cDate2 = java.util.Date.class;
10         Object d2 = cDate2.newInstance(); // Attention : perte du type !
11         System.out.println("d2 = " + d2);
12
13         // Erreur si pas de constructeur sans paramètre
14         try {
15             Object p1 = Point.class.newInstance();
16         } catch (InstantiationException e) {
17             System.out.println("Erreur : " + e);
18 }   }   }
```

```
1 d1 = Wed Aug 26 10:13:50 CEST 2020
2 d2 = Wed Aug 26 10:13:50 CEST 2020
3 Erreur : java.lang.InstantiationException: Point
```

Depuis Java9, Class.newInstance est obsolète !

Créer un objet : Constructor.newInstance

```
1  public class IntrospectionCreationConstructor {
2      public static void main(String[] args) {
3          // Utiliser un constructeur de la classe
4          try {
5              java.lang.reflect.Constructor c =   // en fait Constructor<?>
6                      Point.class.getConstructor(double.class, double.class);
7              Object p2 = c.newInstance(1, 2);
8              System.out.println("p2 = " + p2);
9
10             // Avec le bon type ?  Pourquoi ça marche ?
11             Point p3 = Point.class.getConstructor(double.class, double.class)
12                     .newInstance(4, 7);
13             System.out.println("p3 = " + p3);
14         } catch (java.lang.NoSuchMethodException e) {
15             System.out.println("Constructeur non trouvé : " + e);
16         } catch (java.lang.IllegalAccessException e) {
17             System.out.println("Droits d'accès insuffisants : " + e);
18         } catch (java.lang.InstantiationException e) {
19             System.out.println("Erreur sur création d'instance : " + e);
20         } catch (java.lang.reflect.InvocationTargetException e) {
21             System.out.println("Erreur sur exécution du constructeur : " + e);
```

```
1  p2 = (1.0, 2.0)
2  p3 = (4.0, 7.0)
```

Incrémenter une propriété JavaBean de type int

```
1  static public void incrementerPropriete(Object objet, String nom)
2      throws IllegalAccessException, NoSuchMethodException,
3          java.lang.reflect.InvocationTargetException
4  {
5      String nomProp = Character.toUpperCase(nom.charAt(0))
6              + nom.substring(1);
7      Class<?> classe = objet.getClass();
8
9      // Récupérer la valeur
10     Method accesseur = classe.getMethod("get" + nomProp);
11     Object valeur = accesseur.invoke(objet);
12
13     // Incrémenter la valeur
14     int nlleValeur = ((Integer) valeur) + 1;
15
16     // Affecter la nouvelle valeur
17     Method modifieur = classe.getMethod("set" + nomProp, int.class);
18     modifieur.invoke(objet, nlleValeur);
19 }
```

Comment tester des méthodes privées ?

Question Comment tester la méthode privée max2 de la classe suivante ?

```
1  public class Max {
2      static private int max2(int a, int b) {
3          return a > b ? a : b;
4      }
5
6      static public int max(int... a) {
7          if (a.length == 0) {
8              throw new IllegalArgumentException("empty array");
9          }
10         int max = a[0];
11         for (int i = 1; i < a.length; i++) {
12             max = max2(max, a[i]);
13         }
14         return max;
15 }     }
```

Une classe de test...

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class MaxSimpleTest {
5
6      @Test public void testerMax2() {
7          assertEquals(5, Max.max2(3, 5));
8          assertEquals(9, Max.max2(9, 2));
9          assertEquals(4, Max.max2(4, 4));
10         assertEquals(7, Max.max2(7, -2));
11     }
12
13     @Test public void testerMax() {
14         assertEquals(5, Max.max(new int[] {5, 3, 1}));
15         assertEquals(5, Max.max(5, 3, 1));  // idem ci-dessus (1.5)
16         assertEquals(1, Max.max(1));
17         assertEquals(5, Max.max(-4, 0, 5, -5));
18 }   }
```

...qui ne fonctionne pas

```
1  MaxSimpleTest.java:7: error: max2(int,int) has private access in Max
2          assertEquals(5, Max.max2(3, 5));
3                          ^
4  MaxSimpleTest.java:8: error: max2(int,int) has private access in Max
5          assertEquals(9, Max.max2(9, 2));
6                          ^
7  MaxSimpleTest.java:9: error: max2(int,int) has private access in Max
8          assertEquals(4, Max.max2(4, 4));
9                          ^
10 MaxSimpleTest.java:10: error: max2(int,int) has private access in Max
11         assertEquals(7, Max.max2(7, -2));
12                         ^
13 4 errors
```

La réflexivité à la rescousse !

```java
import org.junit.*;
import static org.junit.Assert.*;

public class MaxTest {

    static private java.lang.reflect.Method mMax2;

    @BeforeClass static public void setUp() throws NoSuchMethodException {
        mMax2 = Max.class.getDeclaredMethod("max2", int.class, int.class);
        mMax2.setAccessible(true);
    }

    @Test public void testerMax2() throws IllegalAccessException, java.lang.
          reflect.InvocationTargetException {
        assertEquals(5, mMax2.invoke(null, 3, 5));
        assertEquals(9, mMax2.invoke(null, 9, 2));
        assertEquals(4, mMax2.invoke(null, 4, 4));
        assertEquals(7, mMax2.invoke(null, 7, -2));
    }

    @Test public void testerMax() {
        assertEquals(5, Max.max(new int[] {5, 3, 1}));
        assertEquals(5, Max.max(5, 3, 1));  // idem ci-dessus (1.5)
        assertEquals(1, Max.max(1));
        assertEquals(5, Max.max(-4, 0, 5, -5));
}   }
```

Résultats :

```
JUnit version 4.12
..
Time: 0,003

OK (2 tests)
```

Définir une pile avec contrôle de type

Sans utiliser la généricité

- **Contexte :** Java 1.4 (pas de généricité) et structures d'objets.

```
/** Une pile de capacité fixe générale :
 * polymorphisme de sous-typage */
public class PileFixeObject {

  private Object[] éléments;
  private int nb;

  public PileFixeObject(int capacité) {
    éléments = new Object[capacité];
    nb = 0;
  }

  public boolean estVide() {
    return nb == 0;
  }

  public Object getSommet() {
    return éléments[nb-1];
  }

  public void empiler(Object elt) {
    éléments[nb++] = elt;
  }

  public void dépiler() {
    nb--;
    éléments[nb] = null;
  }
}
```

- **Objectif :** Ajouter un contrôle de type sur `empiler` .

Problèmes posés par PileFixeObject

- Quand l'erreur est détectée ?
- Toutes les erreurs sont-elles détectées ?

```
1  public class ExemplePileFixeObject {
2      public static void main(String[] args) {
3          PileFixeObject pile = new PileFixeObject(10);
4              // une pile de points !
5          pile.empiler(new Point(1, 1));
6          System.out.println("sommet␣:␣" + pile.getSommet());
7          pile.empiler(new PointNomme("B", 2, 2));
8          System.out.println("sommet␣:␣" + pile.getSommet());
9          pile.empiler(new Integer(5));
10         System.out.println("sommet␣:␣" + pile.getSommet());
11
12         Point s = (Point) pile.getSommet();
13 }   }
```

```
1 sommet : (1.0, 1.0)
2 sommet : B:(2.0, 2.0)
3 sommet : 5
4 Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class
        Point (java.lang.Integer is in module java.base of loader 'bootstrap'; Point is in unnamed module
        of loader 'app')
5         at ExemplePileFixeObject.main(ExemplePileFixeObject.java:12)
```

Pile vérifiée

Fournir le type des éléments et vérifier l'élément empilé

```
1  public class PileFixeVerifiee {
2      private PileFixeObject pileInterne;
3      private Class type; // le type des éléments de la Pile
4
5      public PileFixeVerifiee(Class type, int taille) {
6          this.type = type;
7          this.pileInterne = new PileFixeObject(taille);
8      }
9      private void verifierType(Object e) {
10         if (!this.type.isInstance(e))
11             throw new ClassCastException("Element␣of␣wrong␣type."
12                     + "\nExpected:␣" + this.type.getName()
13                     + "\nFound:␣" + e.getClass().getName());
14     }
15     public boolean estVide()    { return pileInterne.estVide(); }
16     public Object getSommet()   { return pileInterne.getSommet(); }
17     public void empiler(Object elt) {
18         verifierType(elt);
19         pileInterne.empiler(elt);
20     }
21     public void dépiler()       { pileInterne.dépiler(); }
22 }
```

Pile vérifiée

Exemple d'utilisation

```
1  public class ExemplePileFixeVerifiee {
2      public static void main(String[] args) {
3          PileFixeVerifiee pile = new PileFixeVerifiee(Point.class, 10);
4              // une pile de points !
5          pile.empiler(new Point(1, 1));
6          System.out.println("sommet : " + pile.getSommet());
7          pile.empiler(new PointNomme("B", 2, 2));
8          System.out.println("sommet : " + pile.getSommet());
9          pile.empiler(new Integer(5));
10         System.out.println("sommet : " + pile.getSommet());
11
12         Point s = (Point) pile.getSommet();
13 }   }
```

Résultats :

```
1 sommet : (1.0, 1.0)
2 sommet : B:(2.0, 2.0)
3 Exception in thread "main" java.lang.ClassCastException: Element of wrong type.
4 Expected: Point
5 Found: java.lang.Integer
6         at PileFixeVerifiee.verifierType(PileFixeVerifiee.java:13)
7         at PileFixeVerifiee.empiler(PileFixeVerifiee.java:18)
8         at ExemplePileFixeVerifiee.main(ExemplePileFixeVerifiee.java:9)
```

Est-ce encore utile depuis la généricité ?

- Est-ce que cette technique est toujours utile depuis Java 1.5 et la généricité ?
- **Ou :** Est-ce que la généricité permet de détecter toutes les erreurs ?

```
1  import java.util.*;
2  public class ExempleListGenericite {
3      public static void main(String[] args) {
4          List<String> ls = new ArrayList<String>();  // Une liste de String (pléonasme !)
5          ls.add("ok ?");     // ajouter String : ok
6          System.out.println("ls = " + ls);
7          List l = ls;    // perte de l'information de type
8          l.add(new Date());  // ajouter Date : Erreur !
9          System.out.println("ls = " + ls);
10
11         for (int i = 0; i < ls.size(); i++) {   // Afficher la liste
12             System.out.println(i + " --> " + ls.get(i));
13 }   }   }
```

```
1 ls = [ok ?]
2 ls = [ok ?, Wed Aug 26 10:13:40 CEST 2020]
3 0 --> ok ?
4 Exception in thread "main" java.lang.ClassCastException: class java.util.Date cannot be cast to class
      java.lang.String (java.util.Date and java.lang.String are in module java.base of loader 'bootstrap
      ')
5         at ExempleListGenericite.main(ExempleListGenericite.java:12)
```

Oui, la preuve : Collections fournit des adaptateurs

```
1  import java.util.*;
2  public class ExempleCheckedListGenericite {
3      public static void main(String[] args) {
4          List<String> ls = Collections.checkedList(new ArrayList<String>(), String.class);
5          ls.add("ok ?");     // ajouter String : ok
6          System.out.println("ls = " + ls);
7          List l = ls;    // perte de l'information de type
8          l.add(new Date());  // ajouter Date : Erreur !
9          System.out.println("ls = " + ls);
10
11         for (int i = 0; i < ls.size(); i++) {   // Afficher la liste
12             System.out.println(i + " --> " + ls.get(i));
13 }   }   }
```

```
1 ls = [ok ?]
2 Exception in thread "main" java.lang.ClassCastException: Attempt to insert class java.util.Date element
      into collection with element type class java.lang.String
3         at java.base/java.util.Collections$CheckedCollection.typeCheck(Collections.java:3049)
4         at java.base/java.util.Collections$CheckedCollection.add(Collections.java:3097)
5         at ExempleCheckedListGenericite.main(ExempleCheckedListGenericite.java:8)
```

Écrire une méthode pour agrandir un tableau

La mauvaise façon

```
1  public class TableauAgrandirFaux {  // Voir [2]
2      /** Obtenir un tableau plus grand, contenant les éléments de tab. */
3      public static Object[] agrandir(Object[] tab, int increment) {
4          assert tab != null;
5          assert increment > 0;
6          Object[] nouveau =  new Object[tab.length + increment];
7          System.arraycopy(tab, 0, nouveau, 0, tab.length);
8          return nouveau;
9      }
10     public static void main(String[] args) {
11         Integer[] t1 = { 1, 2, 3, 4 };
12         Integer[] t2 = (Integer[]) agrandir(t1, 2);
13         System.out.println("t2.length = " + t2.length);
14         System.out.println("t2[3] = " + t2[3]);
15         System.out.println("t2[4] = " + t2[4]);
16 }   }
```

Exécution :

```
Exception in thread "main" java.lang.ClassCastException: class [Ljava.lang.Object;
     cannot be cast to class [Ljava.lang.Integer; ([Ljava.lang.Object; and [Ljava.lang.
     Integer; are in module java.base of loader 'bootstrap')
        at TableauAgrandirFaux.main(TableauAgrandirFaux.java:12)
```

Écrire une méthode pour agrandir un tableau

La bonne façon

```
1  import java.lang.reflect.Array;
2  public class TableauAgrandirBon {   // Voir [2]
3      /** Obtenir un tableau plus grand, contenant les éléments de tab. */
4      public static Object agrandir(Object[] tab, int increment) {
5          assert tab != null;
6          assert increment > 0;
7          Class type = tab.getClass().getComponentType();;
8          Object nouveau = Array.newInstance(type, tab.length + increment);
9          System.arraycopy(tab, 0, nouveau, 0, tab.length);
10         return nouveau;
11     }
12     public static void main(String[] args) {
13         Integer[] t1 = { 1, 2, 3, 4 };
14         Integer[] t2 = (Integer[]) agrandir(t1, 2);
15         System.out.println("t2.length = " + t2.length);
16         System.out.println("t2[3] = " + t2[3]);
17         System.out.println("t2[4] = " + t2[4]);
18 }   }
```

Exécution :

```
t2.length = 6
t2[3] = 4
t2[4] = null
```

Sommaire

Conclusion

- Introspection et intercession sont des techniques utiles pour écrire des applications :
  - évolutives dynamiquement (prendre en compte de nouvelles classes non connues au moment de l'écriture d'une application) ;
  - adaptables dynamiquement (modifier l'application en fonction d'information découverte à l'exécution)
- Code plus lourd à écrire (lié à l'introspection)
- Perte de toutes les aides qu'un compilateur apporte :
  - existance d'un attribut, d'une méthode ou d'un constructeur
  - vérification des types (seulement à l'exécution !)
  - ...
- Attention, forte pénalité en terme temps d'exécution.