NFP121 — Programmation Avancée
—
Introspection en Java

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Sommaire

Sommaire

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

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 }
```

Exécution de la classe de test en faisant :

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

```
1 ..
2 Time: 0,002
3
4 OK (2 tests)
```

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` et `testDelete`).
- 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 permet l'**introspection** et **intercession**.

Charger une classe et afficher ses méthodes publiques

Définition de la classe

```
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      if (noms.length == 0) {
7        System.out.println("java " + afficheur.getClass().getName() + " Classe...");
8      } else {
9        for (String nom: noms) {
10         afficheur.listerMethodes(nom);
11   } } } // gain de place !
12
13   /** Afficher la signature des méthodes de la classe nommée nomClasse.
14   * @param nomClasse le nom de la classe */
15   public void listerMethodes(String nomClasse) {
16     try {
17       Class<?> laClasse = Class.forName(nomClasse);
18       System.out.println("Méthodes de " + laClasse.getName() + " : ");
19       for (java.lang.reflect.Method m : laClasse.getMethods()) {
20         System.out.println(" - " + m);
21       }
22     }
23     catch (ClassNotFoundException e) {
24       System.out.println("!!! Classe inconnue : " + nomClasse);
25 } } } // gain de place !
```

Charger une classe et afficher ses méthodes

Utilisation : java AfficherMethodes PointNomme

```
1  MÃ©thodes 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 double Point.distance(Point)
10  - public void Point.translater(double,double)
11  - public java.awt.Color Point.getCouleur()
12  - public void Point.setCouleur(java.awt.Color)
13  - public java.lang.String Point.toString()
14  - public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
15  - public final native void java.lang.Object.wait(long) 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.)

Sommaire

- Définitions
- 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 principales classes de l'API java.lang.reflect
- La classe `java.lang.Class`

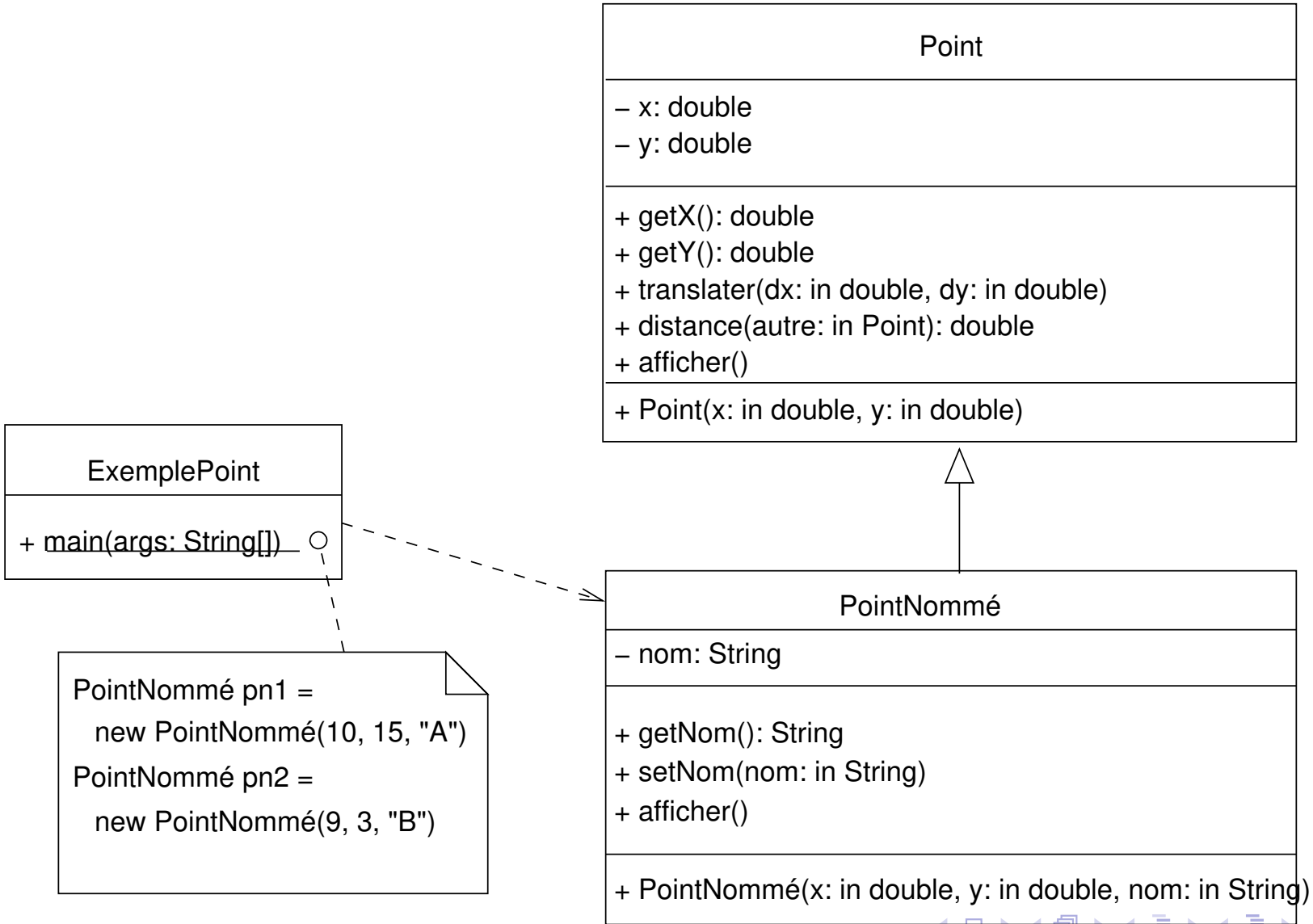
Définitions

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

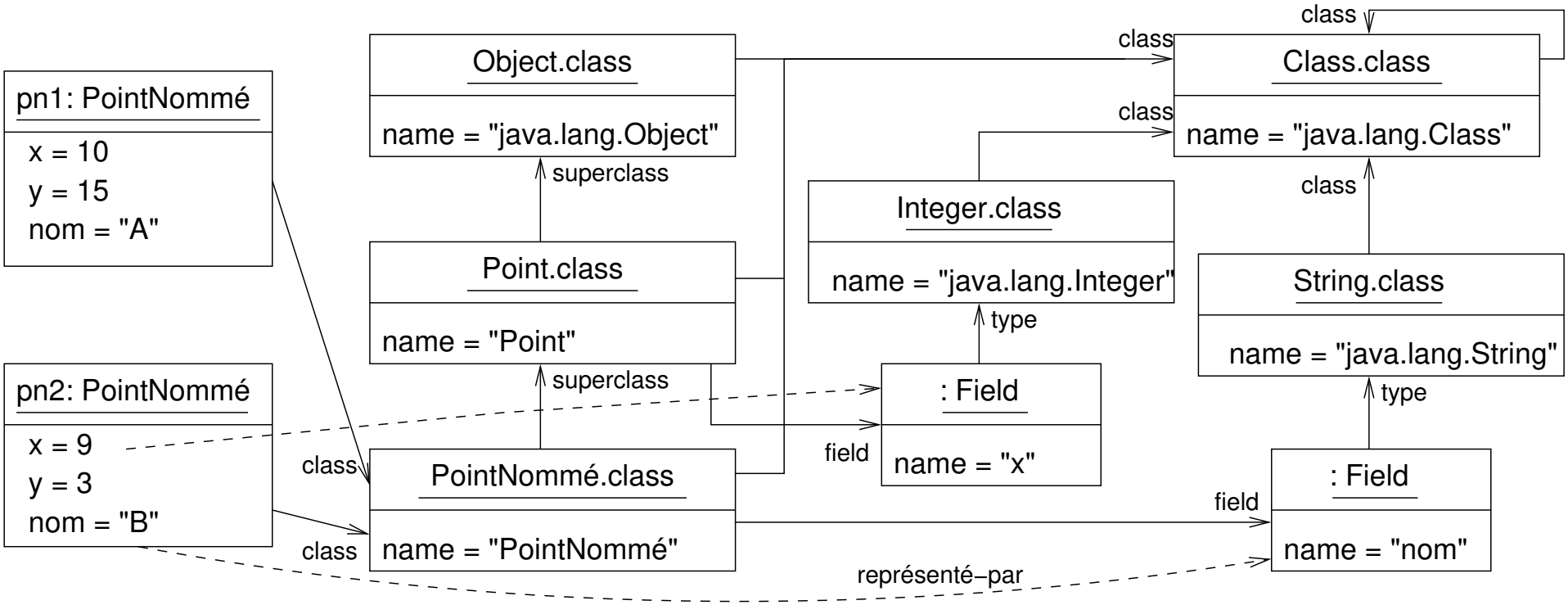
- les classes,
- les attributs,
- les méthodes,
- les constructeurs
- ...

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

Les classes Point et PointNommé

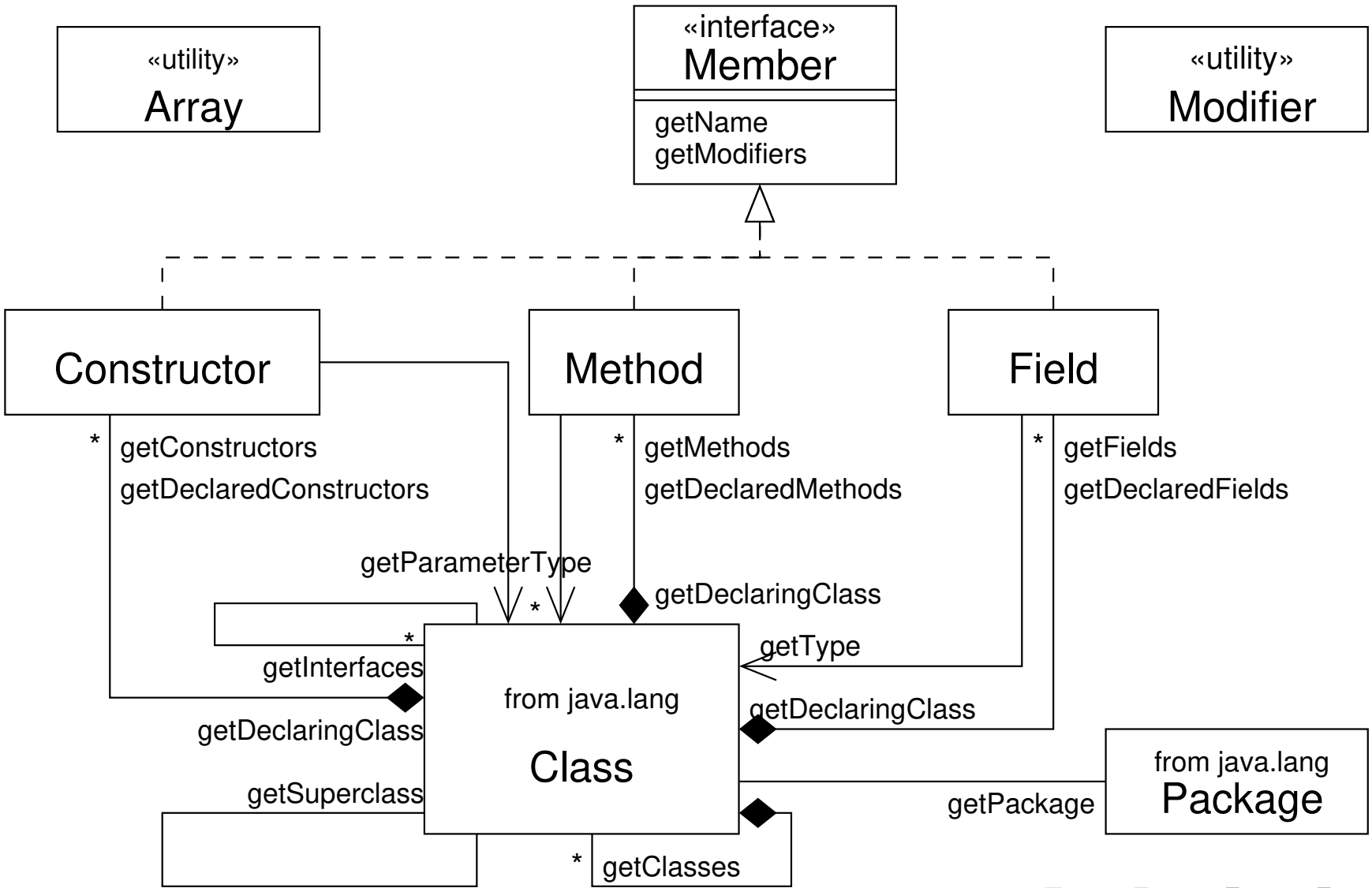


Les objets à l'exécution



- 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



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      Class<?> c1 = new PointNomme(1, 2, "A").getClass();
7      System.out.println("c1 = " + c1);
8      System.out.println("\"chaine\".class = " + "chaine".getClass());
9
10     // à partir d'une classe
11     Class<PointNomme> c2 = PointNomme.class;
12     System.out.println("c2 = " + c2);
13     System.out.println("int.class = " + int.class);
14     System.out.println("int[].class = " + int[].class);
15     System.out.println("double[].class = " + double[].class);
16     System.out.println("Integer.class = " + Integer.class);
17     System.out.println("Integer[].class = " + Integer[].class);
18     System.out.println("java.util.Iterator.class = " + java.util.Iterator.class);
19
20     // à partir du chargeur de classe (donc pas connu à la compilation !)
21     Class<?> c3 = Class.forName("java.lang.Float");
22     System.out.println("c3 = " + c3);
23
24     // attention à donner le nom qualifié !
25     Class<?> c4 = Class.forName("Float");
26     }
27 }
```

Class<T> : T est la classe représentée par cet objet `Class` !

Obtenir un objet de type Class

Résultat de l'exécution

```
c1 = class PointNomme
"chaine".class = class java.lang.String
c2 = class PointNomme
int.class = int
int[].class = class [I
double[].class = class [D
Integer.class = class java.lang.Integer
Integer[].class = class [Ljava.lang.Integer;
java.util.Iterator.class = interface java.util.Iterator
c3 = class java.lang.Float
Exception in thread "main" java.lang.ClassNotFoundException: Float
        at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:264)
        at ObtenirUnObjetClass.main(ObtenirUnObjetClass.java:25)
```

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
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         }
21     }
22 }
```

Obtenir les supertypes : super classe et interfaces

- 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
3      static void listerSuperTypes(Class c) {
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         System.out.println();
11     }
12
13     public static void main(String[] args) {
14         listerSuperTypes(PointNomme.class);
15         listerSuperTypes(java.util.List.class);
16         listerSuperTypes(java.util.ArrayList.class);
17         listerSuperTypes(Object.class);
18         listerSuperTypes(int.class);
19     }
20 }
```

Obtenir les supertypes : superclasse et interfaces

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

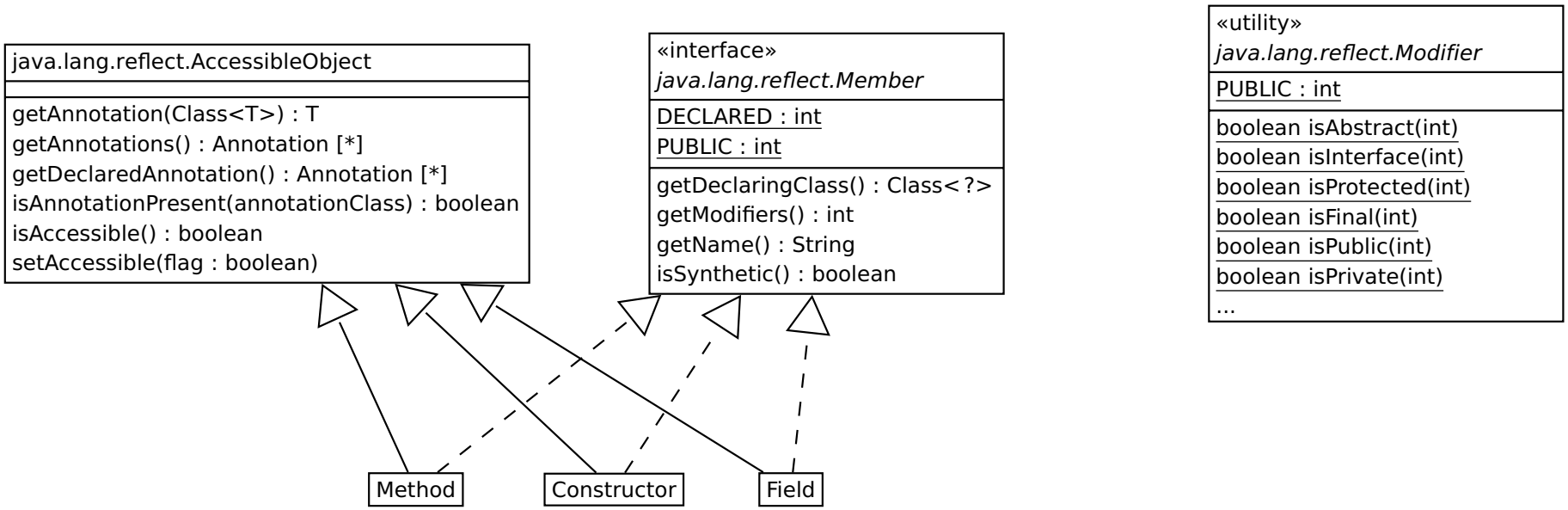
- **Récupérer toutes les méthodes :**
  - getMethods : toutes les méthodes publiques de la classe (y compris héritées)
  - getDeclaredMethods : toutes les méthodes déclarées dans la classe (quelque soit le droit d'accès). Donc pas les méthodes héritées.
- **Récupérer une seule méthode :** getMethod et 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 de paramètre
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     }
23 }
```

```
1   - public void PointNomme.afficher()
2   - public java.lang.String PointNomme.getNom()
3   - public void PointNomme.setNom(java.lang.String)
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

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()      // Modifiers : droit d'accès, final, abstraite, etc.
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, etc.
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()      // Modifiers : droit d'accès, final, etc.
2  String getName()        // le nom de l'attribut
3  public Class getType()  // le type de l'attribut
4  public boolean isEnumConstant() // constante d'un type énuméré ?
5  Class getDeclaringClass()       // la classe déclarant l'attribut
6  public 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()              // Modifiers : droit d'accès, final, abstraite, etc.
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
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 ...
```

Sommaire

- Créer un objet
- Manipuler un attribut
- Appeler une méthode

Créer un objet : Class.newInstance

```java
public class IntrospectionCreationNewInstance {
    public static void main(String[] args)
            throws InstantiationException, IllegalAccessException {
        // avec Class.newInstance (constructeur par défaut)
        Class<java.util.Date> cDate1 = java.util.Date.class;
        java.util.Date d1 = cDate1.newInstance();   // d1 du bon type !
        System.out.println("d1 = " + d1);

        Class<?> cDate2 = java.util.Date.class;
        Object d2 = cDate2.newInstance(); // Attention : perte du type !
        System.out.println("d2 = " + d2);

        // Erreur si pas de constructeur sans paramètre
        try {
            Object p1 = Point.class.newInstance();
        } catch (InstantiationException e) {
            System.out.println("Erreur1 : " + e);
        }
    }
}
```

```
d1 = Thu Dec 08 23:07:42 CET 2016
d2 = Thu Dec 08 23:07:42 CET 2016
Erreur1 : java.lang.InstantiationException: Point
```

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("Accès illégal : " + 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 appel constructeur : " + e);
22         }
23     }
24 }
```

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

Manipuler un attribut

Accès en lecture et modification

```
1  class A {
2      int attr1;
3      private int attr2;
4  }
5  public class IntrospectionIncrementerAttribut {
6      public static void main(String[] args) {
7          try {
8              A a = new A();
9              // récupérer l'attribut décrivant attr1
10             java.lang.reflect.Field fAttr1 = A.class.getDeclaredField("attr1");
11             System.out.println("fAttr1.getInt(a) = " + fAttr1.getInt(a));
12
13             // modifier l'attribut attr1
14             fAttr1.setInt(a, 11);
15             System.out.println("a.attr1 = " + a.attr1);
16
17             java.lang.reflect.Field fAttr2 = A.class.getDeclaredField("attr2");
18             System.out.println("fAttr2.getInt(a) = " + fAttr2.getInt(a));
19         } catch (Exception e) {
20             System.out.println("Erreur : " + e);
21         }
22     }
23 }
```

```
1 fAttr1.getInt(a) = 0
2 a.attr1 = 11
3 Erreur : java.lang.IllegalAccessException: Class IntrospectionIncrementerAttribut can not access a member
        of class A with modifiers "private"
```

Appeler une méthode

Exemple : 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).intValue() + 1;
15
16          // Affecter la nouvelle valeur
17          Method modifieur = classe.getMethod("set" + nomProp, int.class);
18          modifieur.invoke(objet, nlleValeur);
19      }
```

Sommaire

- Tester des méthodes privées
- Structure de données avec contrôle de type
- Agrandir un tableau

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     }
16
17 }
```

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     }
19
20     public static void main(String[] args) {
21         org.junit.runner.JUnitCore.main(MaxSimpleTest.class.getName());
22     }
23 }
```

...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 !

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class MaxTest {
5
6      static private java.lang.reflect.Method mMax2;
7
8      @BeforeClass static public void setUp() throws NoSuchMethodException {
9          mMax2 = Max.class.getDeclaredMethod("max2", int.class, int.class);
10         mMax2.setAccessible(true);
11     }
12
13     @Test public void testerMax2() throws IllegalAccessException, java.lang.reflect.
           InvocationTargetException {
14         assertEquals(5, mMax2.invoke(null, 3, 5));
15         assertEquals(9, mMax2.invoke(null, 9, 2));
16         assertEquals(4, mMax2.invoke(null, 4, 4));
17         assertEquals(7, mMax2.invoke(null, 7, -2));
18     }
19
20     @Test public void testerMax() {
21         assertEquals(5, Max.max(new int[] {5, 3, 1}));
22         assertEquals(5, Max.max(5, 3, 1));  // idem ci-dessus (1.5)
23         assertEquals(1, Max.max(1));
24         assertEquals(5, Max.max(-4, 0, 5, -5));
25     }
26
27     public static void main(String[] args) {
28         org.junit.runner.JUnitCore.main(MaxTest.class.getName());
29     }
30 }
```

La réflexivité à la rescousse ! (2)

Résultats :

```
1 JUnit version 4.12
2 ..
3 Time: 0,004
4
5 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 d'héritage */
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 posées 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                 Point s = (Point) pile.getSommet();
12         }
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: java.lang.Integer cannot be cast to Point
5         at ExemplePileFixeObject.main(ExemplePileFixeObject.java:11)
```

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); pileInterne.empiler(elt);
19   }
20   public void dépiler()         { pileInterne.dépiler(); }
21 }
```

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                 Point s = (Point) pile.getSommet();
12         }
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                  // Initialiser une liste
5                  List<String> ls = new ArrayList<String>();
6                  ls.add("ok ?");
7                  System.out.println("ls = " + ls);
8                  // Attention à ce qui suit !
9                  List l = ls;    // perte de l'information de type (-Xlint:unchecked)
10                 l.add(new Date());
11                 System.out.println("ls = " + ls);
12                 // Afficher la liste
13                 for (int i = 0; i < ls.size(); i++) {
14                         System.out.println(i + " --> " + ls.get(i));
15 }       }       }
```

```
1  ls = [ok ?]
2  ls = [ok ?, Thu Dec 08 23:07:38 CET 2016]
3  0 --> ok ?
4  Exception in thread "main" java.lang.ClassCastException: java.util.Date cannot be cast to java.lang.
       String
5          at ExempleListGenericite.main(ExempleListGenericite.java:14)
```

Oui, la preuve : Collections fournit des adaptateurs

```
1  import java.util.*;
2  public class ExempleCheckedListGenericite {
3          public static void main(String[] args) {
4                  // Initialiser une liste
5                  List<String> ls = Collections.checkedList(new ArrayList<String>(), String.class);
6                  ls.add("ok ?");
7                  System.out.println("ls = " + ls);
8                  // Attention à ce qui suit !
9                  List l = ls;    // perte de l'information de type (-Xlint:unchecked)
10                 l.add(new Date());
11                 System.out.println("ls = " + ls);
12                 // Afficher la liste
13                 for (int i = 0; i < ls.size(); i++) {
14                         System.out.println(i + " --> " + ls.get(i));
15 }       }       }
```

```
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.util.CollectionsCheckedCollection.typeCheck(Collections.java : 3037)atjava.util.Collections
               CheckedCollection.add(Collections.java:3080)
4          at ExempleCheckedListGenericite.main(ExempleCheckedListGenericite.java:10)
```

Écrire une méthode pour agrandir un tableau

La mauvaise façon

```
1  public class TableauAgrandirFaux {  // Voir [1]
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     }
17 }
```

Exécution :

```
Exception in thread "main" java.lang.ClassCastException: [Ljava.lang.Object; cannot be
     cast to [Ljava.lang.Integer;
        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 [1]
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     }
19 }
```

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.

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

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