Technologie Objet

—

Interfaces – Généricité

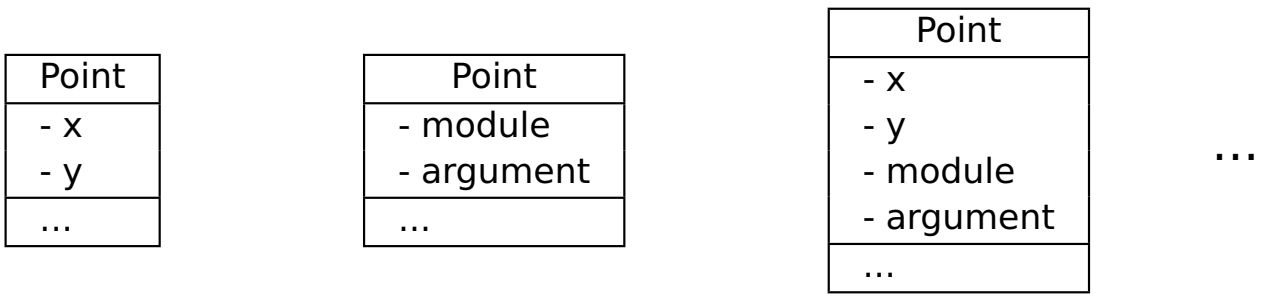
Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Sciences du Numérique

Motivation : Spécification et implantation de la classe Point

Démarche suivie pour spécifier puis implanter les points :

1. Spécifier un point : diagramme d'analyse (requêtes / commandes)
   - On connait ses opérations (les requêtes et les commandes)
   - On connait leur spécification (signature, javadoc, DBC, etc.)
2. Implanter : plusieurs possibilités (chacune a des avantages et inconvénients)



Questions :

- Comment faire pour garder toutes les versions de la classe Point ?
- Peut-on calculer la distance entre deux points de types différents ?
- Peut-on créer un segment à partir de points de types différents ?

Réponses aux questions

Comment faire pour garder toutes les versions de la classe Point ?

- Il faut donner un nom différent aux 3 classes
- Appelons PointCartesien, PointPolaire et PointMixte ces trois classes

Peut-on calculer la distance entre deux points de types différents ?

- Non car la signature de la méthode distance dans PointCartesien, par exemple, sera :

```
public double distance(PointCartesien autre) { ... }
```

- Le paramètre effectif peut donc être PointCartesien, mais ni PointPolaire, ni PointMixte !

Peut-on créer un segment à partir de points de types différents ?

- Les deux points en paramètre du constructeur de Segment sont de type Point,
- Il faudra donc choisir quel point : PointCartesien, PointPolaire, etc.
- Les autres ne seront plus acceptés !

La solution ? INTERFACE

Sommaire

Qu'est ce qu'une interface

Définition : Une interface correspond à une spécification. Elle définit un type (son nom) et la spécification des opérations qui peuvent lui être appliquées (méthodes sans code).

Définition : On appelle méthode abstraite une méthode dont le code n'est pas donnée.

Remarque : Une interface correspond à une vue utilisateur !
Une interface correspond au diagramme d'analyse (requêtes/commandes).

Point
requêtes
x : double
y : double
module : double
argument : double
distance(autre : Point) : double
commandes
translater(dx : double, dy : double)
afficher()
setX(nx : double)
setY(ny : double)
setModule(moudule : double)
setArgument(argument : double)

L'interface Point

```
/** Définition d'un point mathématique dans un plan qui peut être
  * considéré dans un repère cartésien ou polaire.  Un point peut être affiché,
  * translaté.  Sa distance par rapport à un autre point peut être obtenue.
  * @author Xavier Crégut <nom@n7.fr>
  */
public interface Point {

    /** Changer l'abscisse de ce point.
      * @param vx   la nouvelle valeur de l'abscisse
      */
    void setX(double vx);

    /** Changer l'ordonnée de ce point.
      * @param vy   la nouvelle valeur de l'ordonnée
      */
    void setY(double vy);

    /** Obtenir l'abscisse de ce point.
      * @return abscisse de ce point
      */
    double getX();

    /** Obtenir l'ordonnée de ce point.
```

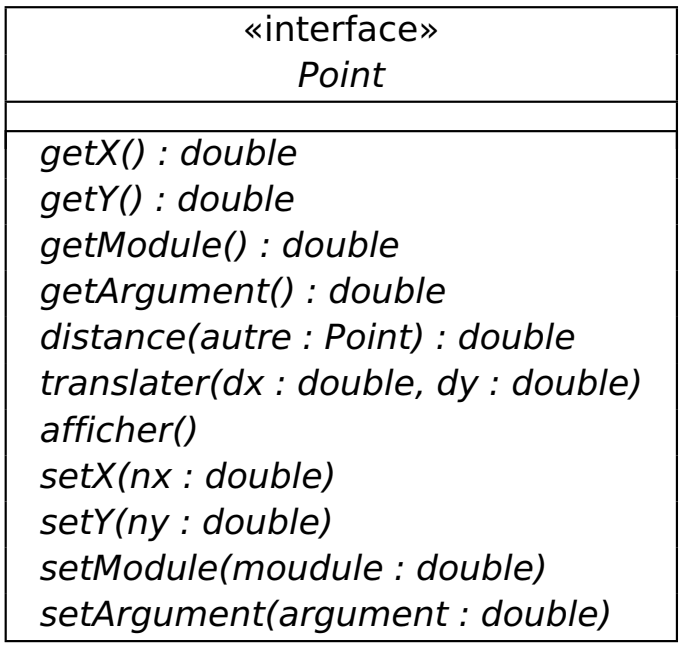
L'interface Point (2)

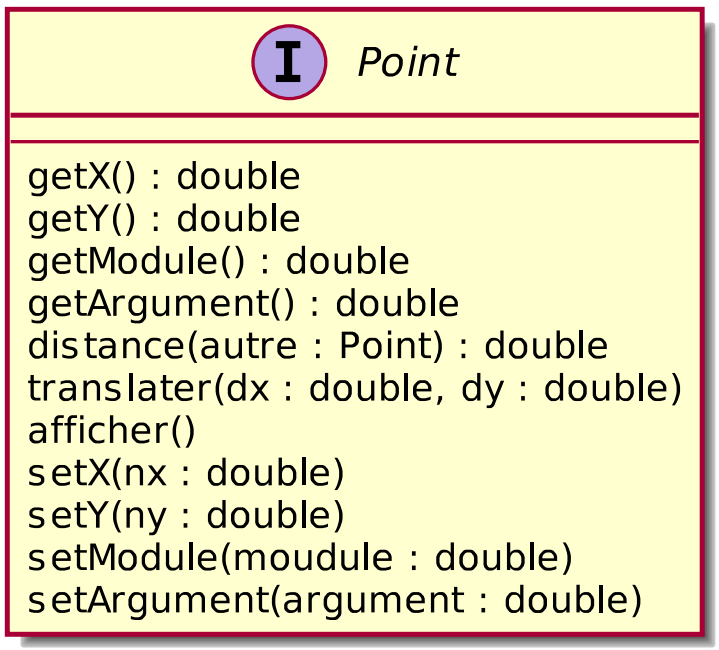
```
   * @return ordonnée de ce point
   */
double getY();

/** Obtenir le module de ce point.
   * @return module de ce point
   */
double getModule();

/** Obtenir l'argument de ce point.
   * @return l'argument de ce point
   */
double getArgument();

/** Changer le module de ce point.
   * @param nouveau_module la nouvelle valeur du module
   */
void setModule(double nouveau_module);

/** Changer l'argument de ce point.
   * @param nouvel_argument la nouvelle valeur de l'argument.
   */
void setArgument(double nouvel_argument);
```

L'interface Point (3)

```
    /** Afficher le point. */
    void afficher();

    /** Obtenir la distance de ce point par rapport à un autre point.
      * @param autre l'autre point
      * @return distance avec l'autre point
      */
    double distance(Point autre);

    /** Translater le point de dx suivant l'axe des X et de dy suivant les Y.
     * @param dx_ déplacement suivant l'axe des X
     * @param dy_ déplacement suivant l'axe des Y
     */
    void translater(double dx, double dy);

}
```

Notation UML





- Utilisation de la représentation d'une classe avec le stéréotype «interface» [1]
- Nom de l'interface et méthodes s'écrivent en italique (comme tout ce qui est abstrait)
- Mettre la contrainte {**abstract**} après une méthode si italique peu lisible [2]

1. Les stéréotypes, noms entre chevrons, permettent d'étendre le vocabulaire d'UML.
2. Les contraintes s'expriment entre accolades en UML et permettent d'ajouter une précision sur le modèle.

Contraintes sur les interfaces

Syntaxe proche d'une classe (**`interface`** remplace **`class`**) avec de nombreuses contraintes :

- Dans une interface, tout est public (nécessairement et implicitement)
  Justification : Une interface est une vue utilisateur !
- Une interface ne contient pas de code
  Justification : Ce n'est qu'une spécification ! Aucun choix de réalisation !
- Donc une interface ne contient que des méthodes abstraites (**`abstract`**) :
  - Leur implantation n'est pas donnée. *Justification :* C'est une spécification, vue utilisateur
  - La documentation est donc essentielle ! *Justification :* Pour comprendre les méthodes
- Une interface ne contient que des méthodes d'instance, pas de méthode de classe (**`static`**)
- Une interface ne peut contenir ni constructeur, ni initialiseur.
  Justification : Car pas de code dans une interface !
- Dans une interface, tout attribut est de classe et constant (nécessairement et implicitement) : **`public final static`**

Remarque : Java8 lève plusieurs de ces contraintes (méthodes de classe, implantation par défaut pour les méthodes d'instance).

Illustration

Une interface

```
1  public interface UneInterface {
2      int unAttribut = 10;    // erreur de compilation si non initialisé
3
4      /** Les commentaires sont essentiels puisqu'il n'y a pas de code !
5        * Ce commentaire n'est pas informatif car cette interface n'a pas
6        * de sens.  */
7      void uneMethode();
8
9  }
```

Résultat de "javap UneInterface"

```
Compiled from "UneInterface.java"
public interface UneInterface{
    public static final int unAttribut;
    public abstract void uneMethode();
}
```

Tout est public. La méthode est abstraite. L'attribut est de classe et constant.

Remarque : javap permet de désassembler le code intermédiaire Java (fichier .class).

Illustration : Pas de méthode de classe

```
1  public interface ErreurStatic {
2       static void methodeDeClasse();
3  }
```

```
1  > javac -source 7 ErreurStatic.java
2  ErreurStatic.java:2: error: static interface methods are not supported in -source 7
3       static void methodeDeClasse();
4                                   ^
5    (use -source 8 or higher to enable static interface methods)
6  1 error
```

Illustration : Pas de code pour les méthodes d'instance

```
1  public interface ErreurCode {
2       void methodeDInstance() {
3           System.out.println("OK␣?");
4       }
5  }
```

```
1  > javac -source 7 ErreurCode.java
2  ErreurCode.java:2: error: interface abstract methods cannot have body
3       void methodeDInstance() {
4                               ^
5  1 error
```

Et même avec Java9 :

```
1  ErreurCode.java:2: error: interface abstract methods cannot have body
2       void methodeDInstance() {
3                               ^
4  1 error
```

Une interface ne permet pas de créer d'objet

```
1  public class CreerInterface {
2      public static void main(String[] args) {
3          UneInterface i = new UneInterface();
4      }
5  }
```

```
CreerInterface.java:3: error: UneInterface is abstract; cannot be instantiated
                UneInterface i = new UneInterface();
                                 ^
1 error
```

Justification : Une interface est une notion abstraite (ses méthodes n'ont pas de code). Si on pouvait créer un objet à partir d'une interface, appeler une méthode sur cet objet conduirait à une incohérence car il n'y aurait pas de code à exécuter !

Question : Une interface ne permet pas de créer d'objets. Quel est son intérêt ?

Utilisation d'une interface

Dans la classe PointCartesien, on écrit la méthode distance comme suit (distance entre un PointCartesien et un Point) :

```
public double distance(Point autre) {
    double dx2 = Math.pow(autre.getX() - this.getX(), 2);
    double dy2 = Math.pow(autre.getY() - this.getY(), 2);
    return Math.sqrt(dx2 + dy2);
}
```

1. Est-ce que le code ci-dessus compile ?
2. Dans l'affirmative, quel est le code exécuté pour « `autre.getX()` » ?

Peut-on écrire une classe Segment qui n'utilise que l'interface Point ?

Réponses aux questions

Est-ce que le code de la méthode distance compile ?

- La question se pose pour « autre.getX() ».
- La réponse est OUI, c'est la liaison statique qui nous le dit !
- Le type de « autre », « Point », a une méthode « getX() » sans paramètre et c'est la seule.
- Conclusion : le compilateur accepte !

Quel est le code exécuté pour « autre.getX() » ?

- On ne sait pas !
- L'implantation de getX() n'est pas donnée dans l'interface Point !
- Une idée ? Encore un peu de patience...

Peut-on écrire une classe Segment qui n'utilise que l'interface Point ?

- Oui. On peut déclarer deux attributs de type Point (les extrémités), on peut écrire le constructeur qui prend en paramètre les deux points qui deviendront les extrémités, on peut écrire les méthodes translater, afficher, longueur... en s'appuyant sur les opérations spécifiées dans l'interface Point

Question : Comment fournir un paramètre effectif de type Point à distance ?

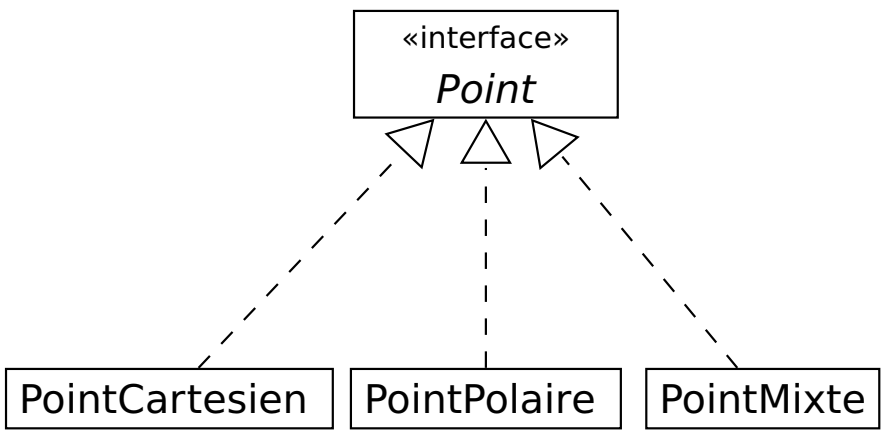
- Et donc comment créer un point ?

Réalisation

Question : Comment créer un objet de type Point ?

Réponse : Écrire une classe qui réalise l'interface `Point`.
PointCartesien, PointPolaire, PointMixte sont de bons candidats.

Définition : On appelle réalisation d'une interface une classe qui s'engage (mot-clé **`implements`** en Java) à définir les opérations spécifiées dans cette interface.




```java
/** ... */
public class PointCartesien implements Point {
    private double x;
    private double y;

    /** ... */
    public PointCartesien(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return this.x;
    }
    ...
```

En Java, c'est le mot-clé **`implements`** qui dit qu'une classe réalise un interface

Syntaxe

En Java :

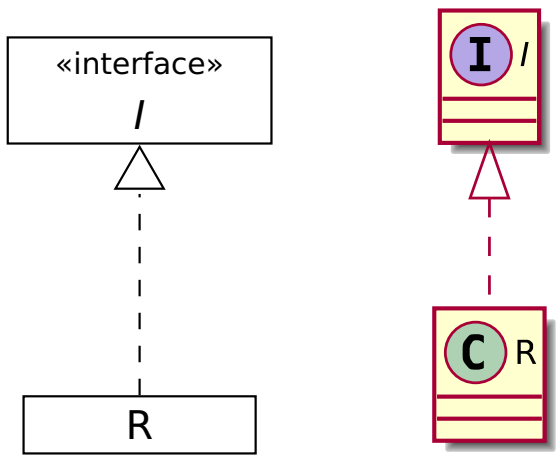
mot-clé : `implements`

```
class R implements I {
        ...
}
```

En UML :

Notation : trait interrompu et flèche fermée (triangle)

Définition : Elle s'appelle relation de réalisation.



Remarque : C'est un cas particulier de dépendance :
R dépend de I : si I change, il faudra peut-être changer R

Réalisation (compléments)

Dans une réalisation, on peut (bien sûr) :

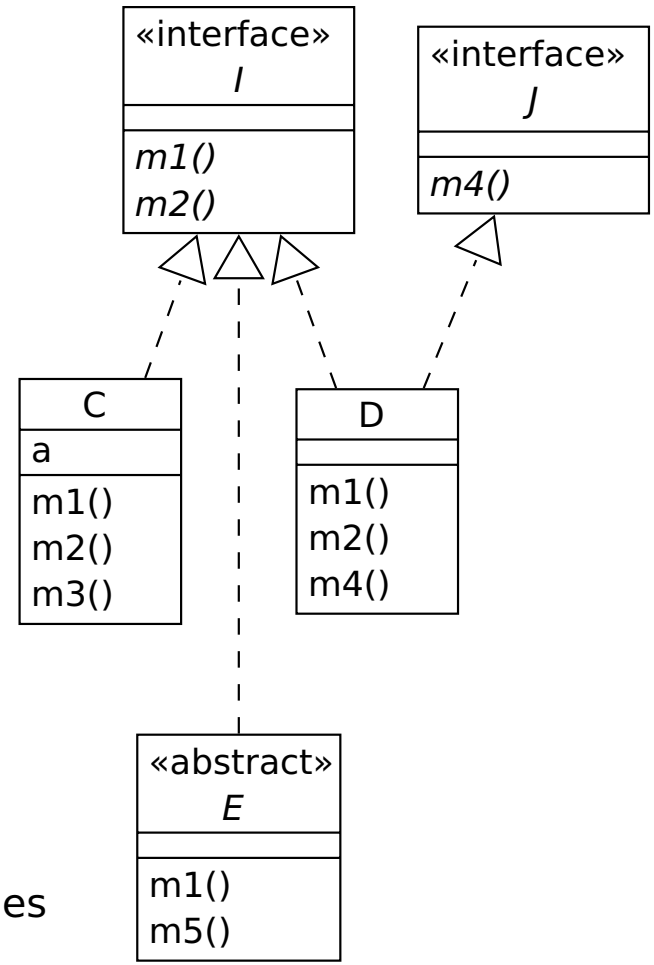
- ajouter de nouvelles méthodes,
- définir des attributs,
- définir des constructeurs,
- utiliser les droits d'accès...

PointCartesien ajoute des attributs et un constructeur.
Mais aurait pu ajouter la gestion de la couleur, pivoter...

Une classe peut réaliser plusieurs interfaces :

```
class R implements I1, I2, ..., In { ... }

class D implements I, J { ... }
```

Une classe qui ne définit pas toutes les méthodes des interfaces qu'elle réalise est dite abstraite (voir héritage) : exemple E.
Incomplète, elle ne permet pas de créer d'instance.



Le code Java du diagramme de classe précédent

```
1   interface I {
2       void m1();
3       void m2();
4   }
5
6   interface J {
7       void m4();
8   }
9
10  class C implements I {
11      private int a;
12      public void m1() { System.out.println("C.m1()"); }
13      public void m2() { System.out.println("C.m2()"); }
14      public void m3() { System.out.println("C.m3()"); }
15  }
16
17  class D implements I, J {
18      public void m1() { System.out.println("D.m1()"); }
19      public void m2() { System.out.println("D.m2()"); }
20      public void m4() { System.out.println("D.m4()"); }
21  }
22
23  abstract class E implements I {
24      public void m1() { System.out.println("E.m1()"); }
25      public void m5() { System.out.println("E.m5()"); }
26  }
```

Le mot-clé **abstract** devant **class** E est nécessaire car E contient une méthode abstraite (m2).
Résultat de la compilation si on omet **abstract** :

```
IJCDE.java:23: error: E is not abstract
     and does not override abstract method
     m2() in I
class E implements I {
^
1 error
```

Sous-type et principe de substitution

Sous-type : Si une classe C réalise une interface I alors le type C est un sous-type du type I.

Exemple : `PointCartesien` est un sous-type de `Point`.

Principe de substitution[3] **:** Si un type T_1 est un sous-type de T_2 alors partout où T_2 est déclaré, on peut utiliser un objet de type T_1.

Conséquence : Une poignée dont le type est une interface peut être initialisée avec toute expression dont le type est une classe réalisant cette interface.

```
Point p1 = new PointCartesien(1, 2);     // OK car PointCartesien sous-type de Point
Point p2 = new PointPolaire(10, 0);      // OK car PointPolaire sous-type de Point
PointPolaire pp = new PoinCartesien(1, 2); // NON, PointCartesien non sous-type de PointPolai
PointCartesien pc = new PoinPolaire(0, 0); // NON, PointPolaire non sous-type de PointCartesi
```

Remarque : Tout comme en français, « Point » est un terme (type) général, abstrait, qui désigne tout point (PointCartesien, PointPolaire, PointMixte...)

Questions : Considérons l'instruction suivante : `double x1 = p1.getX();`

1. Est-ce que cette instruction est autorisée ?
2. Quelle est la méthode getX() qui sera exécutée ?

3. de Barbara Liskov.

Réponses aux questions

Est-ce que cette instruction est autorisée ?

- La liaison statique répond OUI : dans le type de p1, Point, il y a bien une méthode nommée "getX" et ne prenant pas de paramètre.

Quelle est la méthode getX() qui sera exécutée ?

- Pas celle de Point ! Point est une interface donc getX() dans Point n'a pas d'implantation
- Rappel : p.m(...) consiste à appliquer la méthode m(...) sur l'objet attaché à la poignée p
- Il en va de même ici : on applique getX() sur l'objet attaché à p1. Quel est cet objet ?
- Ici, c'est un PointCartesien, c'est donc la méthode getX() définie dans PointCartesien (qui retourne la valeur de l'attribut x).
  Si on avait fait p1 = new PointPolaire(10, 0), ça aurait été la méthode getX() de PointPolaire (qui calcule l'abscisse en fonction du module et de l'argument)
- En général, le compilateur ne connait pas la classe de l'objet attaché à une poignée (le programme pourrait demander à l'utilisateur s'il veut un point cartésien, polaire, mixte...)
- En général, le choix de la méthode à exécuter doit donc être fait à l'exécution. On l'appelle donc *liaison dynamique* ou *liaison tardive*.

Liaison dynamique (ou liaison tardive)

```
1  // général          // appliqué aux points
2  I p;                Point p;                    // déclaration d'une poignée
3  p = new C();        p = new PointMixte(1, 2);   // initialisation de la poignée
4  p.m(a1, ..., an);   p.getX();                   // utilisation de la poignée
```

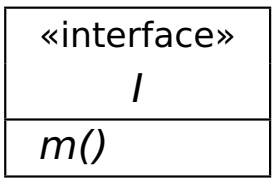
Principe : Quand une méthode est appliquée sur une poignée :

1. Le compilateur vérifie que l'appel est correct en s'appuyant sur le type de la poignée.
   C'est la **liaison statique** (**résolution de la surcharge**)
   Si l'appel est incorrect, c'est une erreur de compilation.
   - Ici, l'appel est correct et l'opération retenue est getX() de Point
2. La méthode exécutée est celle qui est définie sur la classe de l'objet attaché à la poignée.
   C'est la **liaison dynamique** ou **tardive** (car classe connue qu'à l'exécution en général).
   Si la poignée est non nulle, cette méthode existe forcément !
   - La poignée « p » est attachée à un PointMixte
   - La méthode correspondant à getX() appliquée sur « p » est donc celle de PointMixte !

Exercice 1 Si I est une interface qui spécifie la méthode m() et p une poignée non nulle déclarée de type I, on est sûr que p.m() marche. Pourquoi ?

Solution de l'exercice

Traduction de l'énoncé en UML et Java




```
I p;
...
assert p != null;
p.m();  // Est-on sûr que l'appel p.m() fonctionnera ?
```

Raisonnement :

1. p est non nulle, c'est donc que la poignée a été initialisée à partir d'un objet :

   ```
   p = new X();
   ```

2. Pour que cette affectation soit possible, il faut que X réalise I (sous-type).
3. Donc X contient la méthode `m()`.
4. De plus, on a pu créer un objet à partir de X, donc la méthode m() ne peut pas être abstraite (sinon la classe aurait été abstraite et n'aurait pas pu permettre de créer des objets).
5. Le code qui sera exécuté est de celui de la méthode m() dans X. CQFD.

Conclusion : Dès qu'on a une poignée non nulle, on peut appliquer dessus toute méthode spécifiée par le type de la poignée (même si c'est une interface).

Exercice sous-typage et liaison dynamique

Exercice 2 Considérons le diagramme de classe du transparent 19 et son code (T. 20). Pour chaque instruction, dire si elle est valide et, dans l'affirmative, dire, pour les appels de méthodes, quelle méthode sera exécutée.

```
C p1 = new C();   I p2 = new C();   J p3 = new C(); // valides ?
p1.m1();          p2.m1();          p3.m1();        // valides ? Méthodes exécutées ?
p1.m3();          p2.m3();          p3.m3();        // valides ? Méthodes exécutées ?
p1.m4();          p2.m4();          p3.m4();        // valides ? Méthodes exécutées ?
```

Réponses à l'exercice 2

Pour la première colonne.

```
C p1 = new C();   // OK : le type de la poignée et la classe sont les mêmes
p1.m1();          // OK : dans C, type de p1, il y a m1() : "C.m1()"
p1.m3();          // OK : dans C, type de p1, il y a m3() : "C.m3()"
p1.m4();          // ERREURS : dans C, type de p1, pas de méthode m4 !
```

Pour la deuxième colonne.

```
I p2 = new C();   // OK : C réalise I, est donc un sous-type, principe de substitution
p2.m1();          // OK : m1() existe dans I, type de p2.
                  // "C.m1()"  car la classe de p2 est C, donc C.m1() exécutée
p2.m3();          // ERREUR : Pas de méthode m3() dans I, type de p2
p2.m4();          // ERREUR : Pas de méthode m4() dans I, type de p2
```

Pour la troisième colonne.

```
J p3 = new C();   // ERREUR : C non compatible avec J (pas de sous-type)
p3.m1();          // ERREUR : pas de m1() dans J, type de p3
p3.m3();          // ERREUR : pas de m3() dans J, type de p3
p3.m4();          // OK : m4() existe dans J, type de p3
                  // mais on se sait pas ce qui ce passe à l'exécution
                  // car la classe de p3 n'est pas connue (erreur sur initialisation)
```

Un objet, plusieurs types !

- Un objet est toujours instance d'une et une seule classe qui ne peut pas changer
  - Un objet PointCartesien sera toujours un objet PointCartesien !
- Une poignée est déclarée d'un et un seul type qui ne peut pas changer
  - Une poignée de type Point sera toujours du type Point.
- Un objet peut avoir plusieurs types : sa classe, toute interface réalisée par sa classe, etc.
  - Un objet PointCartesien a les types PointCartesien et Point.
- Une poignée peut donner accès à des objets de classes différentes
  - Une poignée de type Point peut donner accès à un PointCatesien, un PointPolaire, etc.

Exemples

```
                        // type poignée (apparent)     classe objet (type réel)
D d = new D();          // D                           D
J j = d;                // J                           D
I i = d;                // I                           D
i = new C()             // I                           C
i = null;               // I                           —
```

- l'objet créé à partir de D est accessible par des poignées de types différents (D, J, I)
- la poignée i (de type I) permet d'atteindre des objets de types différents (D, C) ou aucun
- Type apparent : Type de (déclaration de) la poignée. Le seul connu du compilateur.
- Type réel : Classe de l'objet attaché à la poignée Seulement connu de la JVM, à l'exécution.

Utilisation (suite)

Maintenant que nous avons des réalisations de la classe Point, nous pouvons écrire un programme qui utilise la distance.
Pour rappel, le code de distance dans Point...

```
1  public double distance(Point autre) {
2      double dx2 = Math.pow(autre.getX() - this.getX(), 2);
3      double dy2 = Math.pow(autre.getY() - this.getY(), 2);
4      return Math.sqrt(dx2 + dy2);
5  }
```

```
1  void illustrer() {
2      PointCartesien pc = new PointCartesien(6, 3);
3      PointPolaire pp = new PointPolaire(10, 0);
4      assert 5 == pc.distance(pp);
5  }
```

1. Est-ce que la méthode illustrer() compile, en particulier l'utilisation de la distance ?
2. Dans l'affirmative, quelle est la méthode distance(Point) qui est exécutée ?
3. Dans l'affirmative, quelles sont les méthodes getX() et getY() qui sont exécutées dans distance(Point) ?

Réponses aux questions

L'appel pc.distance(pp) est accepté car :

- une seule méthode distance dans PointCartesien, type de pc
- elle a pour paramètre formel un Point
- pp est compatible avec Point, car son type PointPolaire réalise Point (substitution)

$\Longrightarrow$ L'appel est donc accepté.

Quelle méthode distance(Point) est exécutée ?

- C'est la méthode distance(Point) qui correspond à celle de PointCartesien (voir ci-dessus).
- La classe de pc est PointCartesien, c'est donc la méthode distance de PointCartesien

Quelles sont les méthodes getX() et getY() exécutées ?

- Ce sont les méthodes qui sont dans la classe de l'objet attaché à « autre ».
  La lecture du code de distance ne permet pas de conclure !
  Il faudra attendre l'exécution !
- Dans « illustrer » on voit que c'est un objet de la classe PointPolaire qui initialise « autre ».
- Ici, ce sont donc les méthodes de la classe PointPolaire qui seront exécutées.

Affectation renversée : problème posé

Question : Que penser des instructions suivantes :

```
1   I p = new C();   // ???
2   C q = p          // ???
```

- Ligne 1 est acceptée : C est un sous-type de I.
- Ligne 2 refusée : I, type de p, n'est pas un sous-type de C (même si le type réel de p est C)
- Logique car en ligne 1 on pourrait faire : `I p = new D()` !

Problème posé : Il est parfois intéressant de pouvoir accéder à des informations spécifiques d'un objet qui sont cachées par le type apparent, par exemple la méthode m3() de C.

```
1   I p = new C();   // ???
2   p.m3();          // ???
3   C q = p          // ???
4   q.m3();          // ???
```

- L'objet C est accessible par la poignée p de type I
- L'appel p.m3() est donc refusé par le compilateur
- L'appel q.m3() est accepté et portera bien sur l'objet C
- Mais l'affectation q = p est refusée, car dans le mauvais sens
- C'est le problème de l'affectation renversée [4]

4. Le terme *affectation renversée* vient du langage Eiffel qui propose l'opérateur « ?= » pour essayer de faire cette affectation.

Affectation renversée : solution

Transtypage : `(Type) expression`

```
3   C q = (C) p  // p considéré de type C, donc affectation OK.
```

- Donne à « expression » le *type apparent* « Type »
- Accepté par le compilateur mais **contrôlé à l'exécution** : si la classe de l'objet associé à « expression » n'est pas du type « Type », l'exception `ClassCastException` est levée !

Interrogation dynamique de type : `expression` **`instanceof`** `Type`

- vrai ssi l'objet attaché à « expression » est du type « Type »

```
if (p instanceof C) {
    C c = (C) p;
    c.m3();
}
```

```
if (p instanceof C) {
    ((C) p).m3(); // Bof !
    // . prioritaire sur (C)
}
```

```
if (p instanceof C c) {
    c.m3();
    // depuis Java 16
}
```

La 1re version est plus lisible et pratique que la 2ème : on a un nouvelle poignée c de type C sur le même objet. On peut donc utiliser toutes les opérations du type C (m3).
Java16 simplifie la syntaxe (3ème version).

`instanceof` est à utiliser avec modération !

- Son utilisation traduit souvent une mauvaise conception !

Type apparent vs type réel

Le type apparent d'une expression limite les opérations possibles sur l'objet associé

```
C p1 = new C();
I p2 = c;
```

- p1 et p2 sont deux poignées sur le même objet de la classe C
- à travers p1, on a accès à tout ce qui est sur C, en particulier m3() et l'attribut a
- à travers p2, on a accès qu'à ce qui est spécifié sur I, donc ni m3() ni l'attribut a

Question : Pourquoi avoir un type apparent différent du type réel ?

- **Pour écrire du code plus général** et donc qui s'appuie sur I et non C !
- **Exemple :** la méthode distance, spécifiée dans Point, implantée dans PointCartesien, etc.
  - Le paramètre « autre » de distance est du type Point : on accepte tout Point
  - Le paramètre effectif sera un « vrai » point : PointCartesien, PointPolaire...
  - Lors du passage de paramètre, on a bien le paramètre « autre » de type Point qui référence un objet dont la classe est PointCartesien, PointPolaire, etc.
- **Exemple :** une classe Segment s'appuyant sur l'interface Point :
  - les attributs extremite1 et extremite2 de Segment sont du type Point
  - mais ils références des PointCartesien, PointPolaire, etc.

Choisir pour type apparent le type le plus général qui spécifie les services utilisés

Surcharge et sous-typage

Qu'affiche l'exécution de la méthode « bar » suivante ?

```
1  void foo(I p)   { System.out.println("I"); }
2  void foo(C p)   { System.out.println("C"); }
3
4  void bar() {
5      C c = new C();
6      I q = c;
7      foo(q);
8      foo(c);
9  }
```

- Pour « foo(q) », seule « foo(I) » convient
  « foo(C) » ne convient pas car I n'est pas un sous-type de C !
  Affichage donc de "I"
- Pour « foo(c) », « foo(C) » et « foo(I) » conviennent !
  Pour « foo(C) », paramètres effectif et formel ont même type : distance de 0
  Pour « foo(I) », on s'appuie sur la susbstitution (et le sous-typage) : distance de 1
  « foo(C) » convient strictement mieux que les autres : elle est choisie
  Affichage donc de "C"
- **Rappel :** Le compilateur ne travaille que sur le type apparent pour résoudre la surcharge !

Sommaire

Algorithme de tri

```
1   public class Trieur {
2
3       /** Trier dans l'ordre croissant un tableau d'entiers (tri à bulle).
4         * @param tab le tableau à trier (!= null)
5         */
6       public void trier(int[] tab) {
7           for (int etape = 1; etape < tab.length; etape++) {
8               // faire une série de permutations
9               for (int i = 0; i < tab.length - etape; i++) {
10                  if (tab[i] > tab[i+1]) {
11                      // permuter tab[i] et tab[i+1]
12                      int memoire = tab[i];
13                      tab[i] = tab[i+1];
14                      tab[i+1] = memoire;
15  }   }   }   }   } // gain de place
```

Questions

- Comment faire pour trier dans l'ordre décroissant ?
- Comment faire que les pairs soient avant les impairs ?
- ....

Remarque : Tri à bulle peu efficace mais il tient facilement sur un transparent !

Principe de la solution

Mauvaise idée : Faire du copier/coller

- Copier/coller la méthode et remplacer `tab[i] > tab[i+1]` par `tab[i] < tab[i+1]`
- ou la comparaison qui va bien, `tab[i] % 2 > tab[i+1] % 2` pour pairs avant impairs
- **Mais ce n'est pas une bonne solution :**
  - Le copier/coller est toujours une mauvaise solution !
  - Si on change pour un algo plus performant, il faudra faire plein de changements
  - Idem si on doit corriger une erreur !
  - Et il faudra faire un nouveau copier/coller pour une nouvelle relation d'ordre.

La bonne solution : Paramétrer

1. La méthode trier dépend du comparateur à utiliser
2. C'est l'utilisateur de trier qui doit pouvoir choisir ce comparateur
3. On en fait un paramètre de trier :
   - le paramètre s'appelle `comparateur`
   - on doit définir son type : `Comparateur` (ou `OrdreTotal` ou ...)
   - on spécifie ce que l'on attend de ce comparateur : comparer deux entiers

   $\Rightarrow$ l'interface est le bon outil pour écrire cette spécification !

ESSENTIEL : Une interface est un outil essentiel pour spécifier ce sont on a besoin pour rendre un code plus général.

Comparateur

L'interface Comparateur :

```
1  /** Spécification d'un un ordre total
2    * sur les entiers (int).
3    */
4  public interface Comparateur {
5      /** Compare les deux arguments. Compare la différence n1 - n2 à 0.
6        * Exemple : compare(n1, n2) > 0 <==> n1 > n2.
7        * Seul le signe du résultat compte : 2 ou 1 sont > 0.
8        */
9      int compare(int n1, int n2);
10 }
```

Attention : Le commentaire de documentation est **essentiel**, n'est-ce pas ?

Utilisation

Ajouter un paramètre à trier pour savoir quel comparateur utiliser !

```
1   public class Trieur {
2
3       /** Trier un tableau d'entiers (tri à bulle).
4         * La relation d'ordre est donnée par le comparateur.
5         * @param tab le tableau à trier (!= null)
6         * @param comparateur utilisé pour comparer deux éléments (!= null)
7         */
8       public void trier(int[] tab, Comparateur comparateur) {
9           for (int etape = 1; etape < tab.length; etape++) {
10              // faire une série de permutations
11              for (int i = 0; i < tab.length - etape; i++) {
12                  if (comparateur.compare(tab[i], tab[i+1]) > 0) {
13                      // permuter tab[i] et tab[i+1]
14                      int memoire = tab[i];
15                      tab[i] = tab[i+1];
16                      tab[i+1] = memoire;
17  }   }   }   }   } // gain de place
```

Question : Est-ce que ce code compile ? Pourquoi ?
Question : Quel code est exécuté quand on utilise la méthode `compare` ?

Exemples de réalisations de l'interface Comparateur

```java
/** Ordre croissant sur les entiers. */
public class IntOrdreCroissant implements Comparateur {
    public int compare(int n1, int n2) {
        return n1 - n2;
}   }
```

```java
/** Ordre décroissant sur les entiers. */
public class IntOrdreDecroissant implements Comparateur {
    public int compare(int n1, int n2) {
        return n2 - n1;
    }
}
```

```java
/** Définit l'ordre total : pair < impair */
public class IntOrdrePairImpair implements Comparateur {
    public int compare(int n1, int n2) {
        return n1 % 2 - n2 % 2;
}   }
```

Exemple d'utilisation des interfaces et réalisations

```
1       public static void main(String[] args) {
2           Trieur trieur = new Trieur();
3
4           int[] t1 = { 5, 4, 1, 2, 4, 3, 10 };
5           trieur.trier(t1, new IntOrdreCroissant());
6           trieur.afficher("Croissant   : ", t1);
7
8           t1 = new int[] { 5, 4, 1, 2, 4, 3, 10 };
9           trieur.trier(t1, new IntOrdreDecroissant());
10          trieur.afficher("Décroissant : ", t1);
11
12          t1 = new int[] { 5, 4, 1, 2, 4, 3, 10 };
13          trieur.trier(t1, new IntOrdrePairImpair());
14          trieur.afficher("Pair/impair : ", t1);
15      }
```

Question : Pourquoi les appels à `trier` sont valides ?

Résultat de l'exécution :

```
Croissant   : [1, 2, 3, 4, 4, 5, 10]
Décroissant : [10, 5, 4, 4, 3, 2, 1]
Pair/impair : [4, 2, 4, 10, 5, 1, 3]
```

Un peu de recul

Comparateur et trier existent dans la bibliothèque Java

- l'interface java.util.Comparator
- la méthode `sort` de la classe `Arrays` (méthode de classe)

Pour aller plus loin...

- Est-ce qu'il serait possible/souhaitable de faire de trier une méthode de classe ?
- Il existe de nombreux algorithmes de tri (insertion, sélection, rapide, tas, etc.), comment faire pour permettre d'utiliser indifféremment ces divers algorithmes ?
  Ceci remet-il en cause la réponse à la question précédente ?

Sommaire

Classe anonyme

```
1  public class ExempleClasseAnonyme {
2      public static void main(String[] args) {
3          Trieur trieur = new Trieur();
4          int[] t1 = { 5, 4, 1, 2, 4, 3, 10 };
5          trieur.trier(t1, new Comparateur() {
6              public int compare(int n1, int n2) {
7                  return n2 - n1;
8              }
9          });
10         trieur.afficher("Décroissant␣␣␣:␣", t1);
11 }   }
```

- La définition de la classe est donnée quand on crée un objet à partir de l'interface
- Attention, syntaxe peu lisible : une classe dans une méthode dans une classe !
- Intérêts :
  - Évite de nommer la classe (bof !)... mais on ne pourra pas la réutiliser !
  - Permet d'accéder aux variables locales (seulement si déclarées **final**[5])
- Il y a bien un **.class** engendré : ExempleClasseAnonyme$1.**class**

5. Ou aurait pu être déclarées **final** depuis Java8.

Sous-typage et programmation par contrat

Principe de substitution de Barbara Liskov

Les contrats exprimés sur une interface s'appliquent sur les réalisations

Invariants :

- Les invariants d'une interface s'appliquent sur ses réalisations
- Une réalisation peut définir des invariants supplémentaires

Contraintes sur la définition d'une méthode spécifiée dans une interface :

- Ne pas renforcer les préconditions :
  - la méthode doit fonctionner au moins dans les cas prévus par sa spécification dans l'interface
- Ne pas affaiblir les postconditions :
  - la méthode doit faire au moins ce qui est attendu par sa spécification dans l'interface

Justification : Analogie avec la sous-traitance

- On sous-traite à quelqu'un qui fera ce qui est prévu (peut-être plus)
  $\Longrightarrow$ renforcement des postconditions
- Et qui ne coûtera pas plus cher !
  $\Longrightarrow$ affaiblissement des préconditions

Attention : C'est au concepteur/programmeur d'être vigilent !

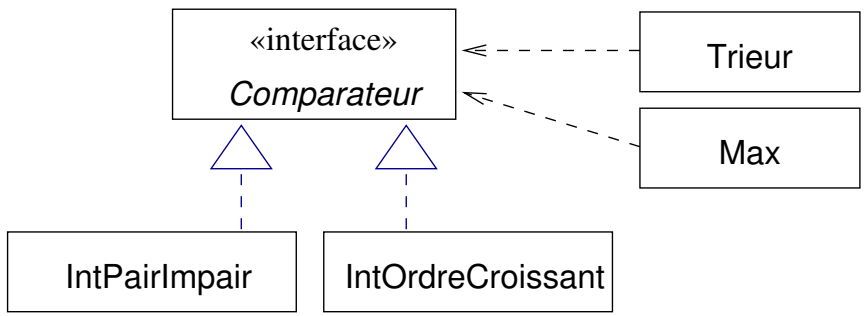
Sommaire

Intérêt d'une interface

Intérêt : Une interface est le point de jonction (ou soudure) entre des classes utilisant l'interface et des classes la réalisant.

- On peut ajouter de nouvelles réalisations (nouveaux comparateurs)
- On peut ajouter de nouveaux utilisateurs (exemple : max, min, etc.)

Exemple : L'interface `Comparateur` est le point de jonction entre la classe `Trieur` qui l'utilise et les réalisations concrètes (`IntOrdreCroissant`...). Max utilise aussi Comparateur, et peut donc utiliser tous les comparateurs concrets existants et futurs.



Définition : Une interface définit un contrat entre utilisateurs et réalisateurs :

- il doit être respecté par les réalisations
- il précise aux utilisateurs comment utiliser l'interface (et donc ses réalisations)

Pourquoi utiliser des interfaces ?

Spécifier le comportement d'un ensemble d'objets
qui pourront être manipulés indépendamment de leur type réel

- Exemples : les comparateurs, les points, etc.

Abstraire un comportement (et s'appuyer dessus)

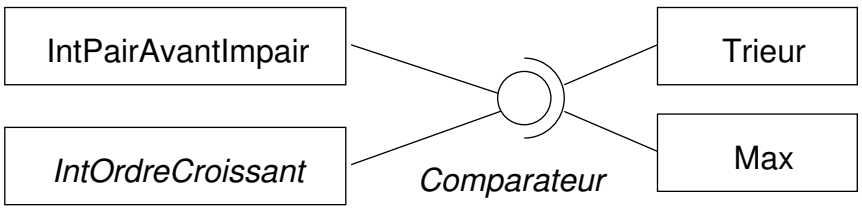
- Exemple : l'algo de tri à besoin de pouvoir comparer des entiers
  La relation d'ordre, le comparateur concret, sera donné ensuite
- Exemple : un segment s'appuie sur des points. Ils peuvent être cartésiens, polaires, etc.

Favoriser l'extensibilité

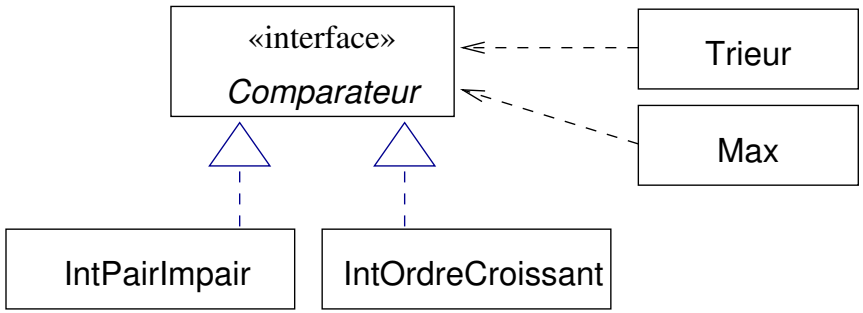
- De nouvelles réalisations peuvent être ajoutées
  - ajouter un nouveau comparateur, de nouveaux types de point
- De nouvelles classes utilisatrices peuvent être ajoutées
  - des méthodes max, min..., un polygone dont les sommets sont des points
- Elles pourront collaborer alors qu'elles ne se connaissent pas directement
  -

Notation alternative en UML

Notation lollipop



Notation « type classe »



- Une interface est représentée par un cercle.
- Une réalisation est reliée par un trait plein au cercle de l'interface.
- L'utilisation d'une interface par une classe est représentée par un trait plein vers un arc de cercle qui entoure le symbole de l'interface.
- Cette notation est en en particulier utlisée pour représenter les composant et faire apparaître les interfaces fournies et requises par un composant.

Sommaire

Interface fonctionnelles et Lambdas (Java8)

Interface fonctionnelle : Interface qui ne possède qu'une seule méthode (ex : Comparateur)

```
1   public class ExempleLambdaExpression {
2       // Cette méthode a la même signature que l'unique méthode abstraite de
3       // l'interface Comparable.
4       static public int paireAvantImpaire(Integer n1, Integer n2) {
5           return n1 % 2 - n2 % 2;
6       }
7
8       public static void main(String[] args) {
9           Trieur trieur = new Trieur();
10          Integer[] t1 = { 5, 4, 1, 2, 4, 3, 10 };
11          // Utilisation d'une méthode là où une interface est utilisée
12          trieur.trier(t1, ExempleLambdaExpression::paireAvantImpaire);
13          trieur.afficher("Pair␣/␣Impair␣:␣", t1);
14          // Utilisation d'une lambda (fonction anonyme)
15          trieur.trier(t1, (n1, n2) -> n1 - n2);
16          trieur.afficher("Croissant␣␣␣␣␣:␣", t1);
17  }   }
```

Méthodes par défaut

```
1  public interface Comparateur<E> {
2      int compare(E n1, E n2);
3
4      default boolean gt(E n1, E n2) {
5          return compare(n1, n2) > 0;
6      }
7
8      default boolean ge(E n1, E n2) {
9          return compare(n1, n2) >= 0;
10     }
11
12     default boolean lt(E n1, E n2) {
13         return compare(n1, n2) < 0;
14     }
15
16     default boolean le(E n1, E n2) {
17         return compare(n1, n2) <= 0;
18     }
19
20     default boolean equals(E n1, E n2) {
21         return compare(n1, n2) == 0;
22     }
23 }
```

Les méthodes par défaut sont des méthodes abstraites qui peuvent ne pas être définies par les réalisations. Le code de l'interface est alors utilisé (par défaut).

Méthodes par défaut : utilisation

Intérêt :

- Proposer une implantation par défaut pour les méthodes de l'interface

```
1  public class IntOrdreCroissant implements Comparateur<Integer> {
2      public int compare(Integer n1, Integer n2) {
3          return n1 - n2;
4  }   }
```

```
1  public class ExempleDefaultMethods {
2      public static void main(String[] args) {
3          IntOrdreCroissant croissant = new IntOrdreCroissant();
4          assert croissant.gt(10, 5);
5          assert croissant.ge(10, 5);
6          assert croissant.lt(4, 10);
7          assert ! croissant.equals(4, 10);
8          assert croissant.equals(10, 10);
9  }   }
```

Remarques :

- Le mot-clé **`default`** est obligatoire (sinon erreur de compilation, voir T. 13).
- Java8 autorise aussi les méthodes **`static`** dans les interfaces.

Sommaire

Motivation

Problème 1 :
On sait trier un tableau d'entiers. Comment faire pour trier un tableau de réels ? De chaînes de caractères ? De points ?...

Problème 2 :
Comment regrouper deux informations, éventuellement de types différents ?

- Faire une classe avec deux attributs !
- On ne veut pas faire autant de classes que de combinaisons possibles des types
- **Idée : paramétrer la classe par un ou plusieurs types !**

Attention : Ce qui est présenté ici n'est qu'une introduction à la généricité !

Classe ou interface générique

```java
public class Couple<A, B> {
    public A premier;
    public B second;

    public Couple(A premier, B second) {
        this.premier = premier;
        this.second = second;
}   }
```

```java
public interface Comparateur<E> {
    int compare(E n1, E n2);
}
```

Remarque : Pour simplifier la classe, les attributs sont publics !

Classe ou interface générique : classe ou interface paramétrée par un (ou plusieurs) types

- Les paramètres sont donnés juste après le nom de la classe et avant l'accolade entre <>
- Ils peuvent être utilisés dans le reste de la classe ou interface

Vocabulaire : E, A et B sont dits *variables de type*, *paramètres de type formels* ou *paramètres de généricité*

Convention : Le nom d'un paramètre de généricité est une lettre en majuscule qui correspond à l'initiale de l'information représentée

Exemple : E pour Élément, C pour Clé (ou K pour Key), V pour Valeur etc.

Instancier une classe générique

Instancier les paramètres de généricité : Pour obtenir une « classe » ou une « interface », il faut préciser la valeur des paramètres de généricité

```
Couple<String, Color>
Couple<Point, Date>
Comparateur<Point>
```

```
1  import java.awt.Color;
2  public class CoupleErreur {
3      public static void main(String[] args) {
4          Couple<String, Color> c1 = new Couple<String, Color>("rouge", Color.RED);
5          Couple<String, Color> c2 = new Couple<>("vert", Color.GREEN);
6              // <> depuis Java7, quand le compilateur peut inférer les paramètres réels
7
8          c1.premier = "RED";          // OK
9          c2.premier = Color.BLACK;    // Erreur !
10         Color c = c2.second;         // OK
11 }   }
```

```
1  CoupleErreur.java:9: error: incompatible types: Color cannot be converted to String
2                  c2.premier = Color.BLACK;       // Erreur !
3                                       ^
4  1 error
```

$\Longrightarrow$ **Le compilateur détecte effectivement les erreurs de type.**

Un paramètre de généricité est forcément un type référence

Limites de la généricité en Java :

- Un paramètre de généricité est nécessairement un type référence en Java
  - Un type référence est tout type sauf les types primitifs
  - Exemples : classe, interface, tableau, enum
- En particulier, un type primitif ne peut pas être paramètre de généricité

Exemple :

```
1  public class ExempleCoupleInt {
2    void m() {
3      Couple<int, Integer> c1;
4    }
5  }
```

```
1  ExempleCoupleInt.java:3: error: unexpected
       type
2      Couple<int, Integer> c1;
3             ^
4    required: reference
5    found:    int
6  1 error
```

Classe enveloppe : Quand on veut donner à un paramètre de généricité un type primitif, on doit utiliser la classe enveloppe correspondante :

Integer, Double, etc. au lieu de **int**, **double**, etc.

Auto boxing/unboxing : Le compilateur gère la transformation entre valeur de type primitif et objet des classes enveloppes correspondantes

Exemples de réalisations d'une interface générique

```
1  public interface Comparateur<E> {
2      int compare(E n1, E n2);
3  }
```

```
1  public class IntOrdreCroissant implements Comparateur<Integer> {
2      public int compare(Integer n1, Integer n2) {
3          return n1 - n2;
4  }   }
```

```
1  /** Ordre inverse d'un ordre total. */
2  public class OrdreInverse<E> implements Comparateur<E> {
3      private Comparateur<E> ordreInitial;
4
5      public OrdreInverse(Comparateur<E> ordre) {
6          this.ordreInitial = ordre;
7      }
8
9      public int compare(E e1, E e2) {
10         return - this.ordreInitial.compare(e1, e2);
11     }
12 }
```

```
Trieur trieur = new Trieur();
Comparateur<Integer> ordre = new OrdreInverse<Integer>(new IntOrdreCroissant());
Integer[] tab = { 5, 4, 1, 2, 4, 3, 10 };
trieur.trier(tab, ordre);
```

```
[10, 5, 4, 4, 3, 2, 1]
```

Intérêt de la généricité

Contrôle de type par le compilateur (voir T. 56)

- C'est le principal intérêt !
- **Attention :** Ne fonctionne que sur les types apparents (liaison statique)

Documentation :

- les types sont plus explicites :

```
Couple<String, Point>
```

- mais peuvent devenir bien longs !

```
Couple<Couple<String, Point>, Couple<Integer, String>>  // Ouf !
```

Éviter la redondance de code

- mécanisme proche de la surcharge

Méthode générique : trier un tableau avec un comparateur

Généraliser la méthode trier précédente

```
1   public class Trieur {
2
3       public <E> void trier(E[] tab, Comparateur<E> comparateur) {
4           for (int etape = 1; etape < tab.length; etape++) {
5               // faire une série de permutations
6               for (int i = 0; i < tab.length - etape; i++) {
7                   if (comparateur.compare(tab[i], tab[i+1]) > 0) {
8                       // permuter tab[i] et tab[i+1]
9                       E memoire = tab[i];
10                      tab[i] = tab[i+1];
11                      tab[i+1] = memoire;
12  }   }   }   }   } // gain de place
```

- La classe Trieur n'est pas générique.
- La méthode trier est généralisée en faisant du type des éléments un paramètre
- Le paramètre généricité est entre <>, juste avant le type de retour
- On pourrait avoir une classe générique qui contient une méthode avec un paramètre de généricité supplémentaire.

Utilisation d'une méthode générique

```
1       static void utiliserTrier() {
2           Trieur trieur = new Trieur();
3           Comparateur<Integer> croissant = new IntOrdreCroissant();
4           Integer[] t1 = { 5, 4, 1, 2, 4, 3, 10 };
5           trieur.trier(t1, croissant);
6           trieur.<Integer>trier(t1, croissant);   // En précisant la valeur de E
7           trieur.afficher("t1 = ", t1);
8       }
```

- En général, le compilateur peut inférer la valeur du paramètre de généricité (ligne 5).
- On peut le préciser explicitement juste devant le nom de la méthode (ligne 6).

Généricité contrainte : méthode max

Question : Écrire max qui retourne le plus grand élément d'un tableau.
On impose d'avoir seulement le tableau en paramètre !

Piste : S'il y a un seul paramètre, on pourrait écrire la signature ainsi :

```
public static <E> E max(E[] tab) { ... }
```

Problème : dans le code, il faudra pouvoir comparer les E, type des éléments du tableau.

- On ne peut pas ajouter un comparateur explicite comme dans trier (interdit par l'énoncé)
- Le type E doit donc être équipé de la méthode de comparaison.
- Cette méthode doit être spécifiée dans une interface : Java fournit Comparable

```
1  public interface Comparable<T> {
2      /** Compares this object with the specified object for order.
3        * Returns a negative integer, zero, or a positive integer
4        * as this object is less than, equal to, or greater than
5        * the specified object... */
6      int compareTo(T o);
7  }
```

- On peut donc imposer que E soit un sous-type de Comparable<E>.
  Le mot-clé utilisé est **extends** (voir héritage), proche de **implements**.

```
public static <E extends Comparable<E>> E max(E[] tab) { ... }
```

Généricité contrainte : méthode max

```
1   public class GenericiteMax {
2       /** Déterminer le plus grand élément d'un tableau.
3        * @param tab le tableau des éléments
4        * @return le plus grand élément de tab */
5       public static <E extends Comparable<E>> E max(E[] tab) {
6           E resultat = null;
7           for (E x : tab) {
8               if (resultat == null || resultat.compareTo(x) < 0) {
9                   resultat = x;
10          }   }
11          return resultat;
12      }
13
14      /** Programme de test de GenericiteMax.max */
15      public static void main(String[] args) {
16          Integer[] ti = { 1, 2, 3, 5, 4 };
17          assert max(ti) == 5;
18
19          String[] ts = { "I", "II", "IV", "V" } ;
20          assert max(ts).equals("V");
21  }   }
```

Généricité contrainte : Explications

- Le paramètre de généricité <E **extends** Comparable<E>> garantit que E est un sous-type de Comparable<E> (comparable avec lui-même)
- Dans la méthode `max`, on peut donc utiliser `compareTo` sur les éléments du tableau.
- `Integer` réalise l'interface `Comparable<Integer>` et `String` réalise `Comparable<String>`. On peut calculer le max des tableaux `ti` et `ts`.
- La méthode `max` utilise l'ordre naturel (Comparable). On pourrait la surcharger pour qu'elle accepte un comparateur explicite (Comparateur/Comparator). Voir java.util.Arrays.sort.
- Il est possible d'imposer plusieurs contraintes. La syntaxe est la suivante :

  $$\texttt{<T extends C \& } I_1 \texttt{ \& } I_2 \texttt{ \& ... \& } I_n \texttt{>}$$

  Une seule classe en première position et un nombre quelconque d'interfaces ensuite (voir héritage).