NFP121 — Programmation Avancée

—

Interfaces – Généricité

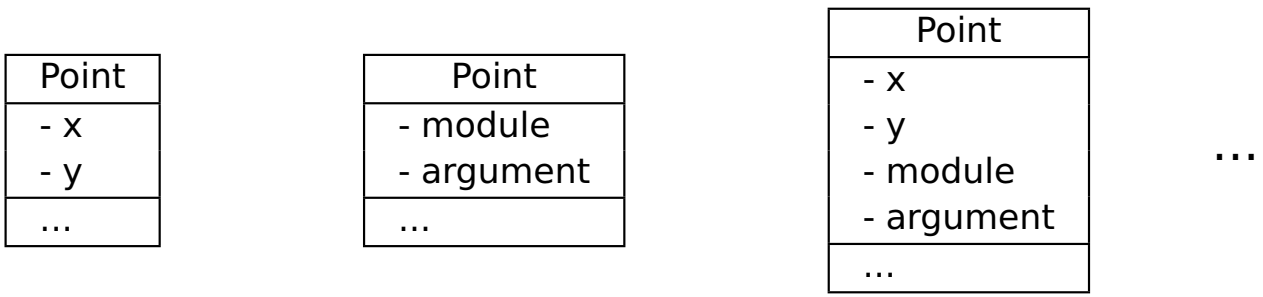
Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

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 ?

Motivation : 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
16  }
```

Questions

- Comment faire pour trier dans l'ordre décroissant ? les pairs avant les impairs ?
- Si tableau d'élèves, classer par ordre alphabétique ? Par âge ? Par moyenne ?...
- Et si on veut disposer de plusieurs algo (insertion, sélection, rapide, tas, etc.) ?

Interface

Définition : Une interface correspond à une spécification. Elle spécifie des opérations d'instance sans en donner le code.

Exemple : L'interface Comparateur (le mot-clé **`interface`** remplace **`class`**)

```
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 : si compare(n1, n2) > 0 alors n1 > n2.
7        * Seul le signe du résultat compte : -2 ou -1 sont < 0.
8        */
9      int compare(int n1, int n2);
10 }
```

Remarque : Ici, une seule opération. Il pourrait y en avoir d'autres.

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
18  }
```

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

Contraintes sur les interfaces

Contraintes sur la définition d'une interface :

- Une interface ne peut **pas contenir de code**
  Justification : Ce n'est qu'une spécification !
- Les méthodes qu'elle spécifie sont dites abstraites (**`abstract`**) :
  - Leur **implantation n'est pas encore donnée.**
- **La documentation est donc essentielle !**
  - informelle : javadoc
  - formelle : JML, etc.
- Ses méthodes sont nécessairement (et implicitement) **publiques** (**`public`**)
  Justification : Une interface est une vue utilisateur !
- Une interface peut contenir des **attributs** mais
  ils sont nécessairement (et implicitement) **`public static final`**.
  - Ce sont donc des constantes de classe.

Illustration

Un exemple d'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();
}
```

Remarque : javap permet de désassembler le byte code Java.

Ce qu'on ne peut pas faire

Une interface ne peut pas contenir :

- de méthodes de classe (**static**)
- d'attributs d'instance
- de constructeurs (ou d'initialiseurs)

On ne peut pas créer une instance d'une interface : c'est une notion abstraite !

```
1  public class CreerInterface {
2      public static void main(String[] args) {
3          UneInterface i = new UneInterface();
4          i.uneMethode(); // Quel code exécuter ?
5      }
6  }
```

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

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

Réalisation

Question : Comment faire pour avoir comparateur (paramètre de trier) non **`null`** ?

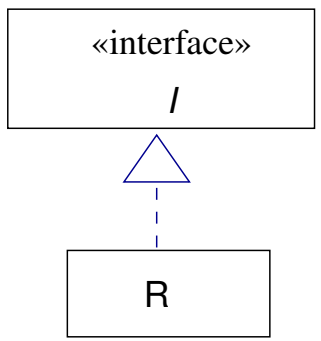
Réponse : Écrire une classe qui réalise l'interface `Comparateur`.

Définition : On appelle réalisation d'une interface une classe qui s'engage à définir les opérations spécifiées dans cette interface.

En Java : `implements`

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

En UML :



Le nom d'une interface s'écrit en italique

Exemples : Réalisations de `Comparateur`.

```
1  /** Ordre naturel sur les entiers. */
2  public class IntOrdreCroissant implements Comparateur {
3      public int compare(int n1, int n2) {
4          return n1 - n2;
5  }   }
```

```
1  /** Définit l'ordre total : pair < impair */
2  public class IntOrdrePairImpair implements Comparateur {
3      public int compare(int n1, int n2) {
4          return n1 % 2 - n2 % 2;
5  }   }
```

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 : Est-ce que 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]
```

Réalisation (compléments)

Dans une réalisation, on peut (bien sûr) :

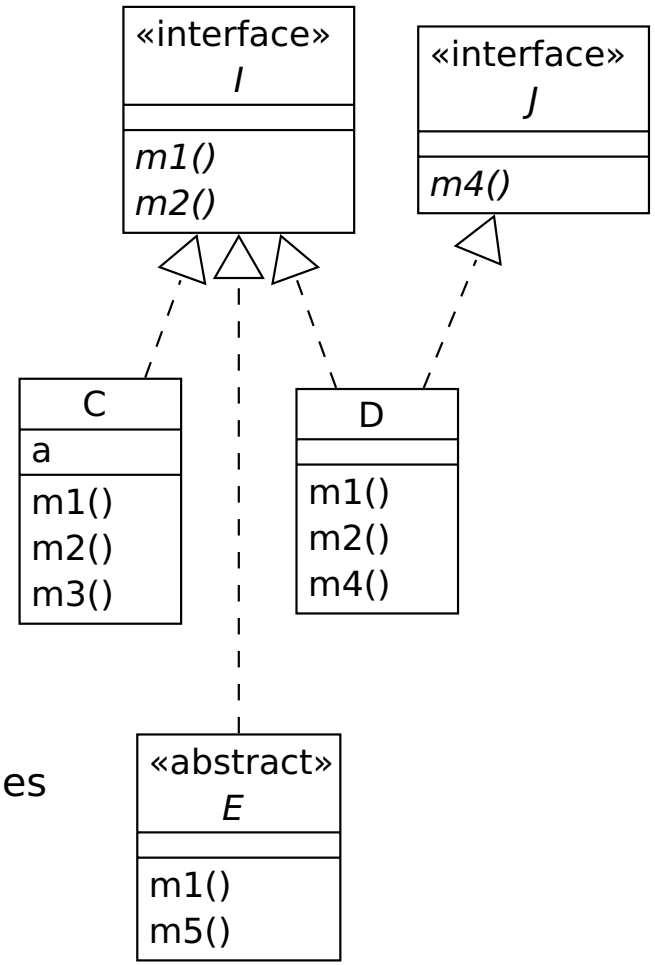
- ajouter de nouvelles méthodes,
- définir des attributs,
- définir des constructeurs,
- utiliser les droits d'accès...

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).
Incomplète, elle ne permet pas de créer des instances.



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 : `IntOrdreCroissant` est un sous-type de `Comparateur`.

Substitution : 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 de type interface peut être initialisée avec tout objet instance d'une classe réalisant cette interface.

```
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 ?
```

```
IntOrdreCroissant croissant = new IntOrdreCroissant();
Trieur trieur = new Trieur();
trieur.trier(t1, croissant);    // conduit à  : Comparateur comparateur = croissant;
```

Remarque : Tout comme en français, « Comparateur » est un terme (type) général, abstrait, qui désigne tout comparateur (IntOrdreCroissant, IntOrdreDecroissant, IntOrdrePairImpair...)

Un objet, plusieurs types !

Conséquence du sous-typage

- Un objet est toujours instance d'une et une seule classe.
- Un objet peut avoir plusieurs types :
  - sa classe,
  - toute interface réalisée par sa classe, etc.
- Une poignée n'a qu'un seul type mais peut donner accès à des objets de types différents

Exemples

```
                    // type apparent        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                    —
```

- un objet créé à partir de D 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.

Vocabulaire

- Type apparent : Type de (déclaration de) la poignée
- Type réel : Classe de l'objet attaché à la poignée

Conséquence de la liaison statique

- Le type apparent limite les opérations applicables sur l'objet attaché

Liaison dynamique (ou tardive)

```
1  int n1 = ..., n2 = ...;
2  Comparateur c;                    // exemple : trier(int[] tab, Comparateur c)
3  c = new IntOrdreCroissant();      // exemple : trier(t1, new IntOrdreCroissant())
4  c.compare(n1, n2);                // appel autorisé ?  Quelle méthode compare ?
```

Attention :

- La classe d'un objet ou le type d'une poignée **ne peuvent pas changer** !
- Mais un objet peut être attaché à des poignées de types différents
- Un seul type réel mais plusieurs types apparents possibles !
- Un seul type apparent mais plusieurs types réels possibles !

Principe : Quand une méthode est appliquée sur une poignée :

1. le compilateur vérifie que la méthode est déclarée dans le type apparent de la poignée (sinon erreur de compilation). C'est la **liaison statique** ;
2. la méthode effectivement exécutée est celle qui est définie sur le type réel de l'objet attaché à la poignée. C'est la **liaison dynamique** ou **tardive**.

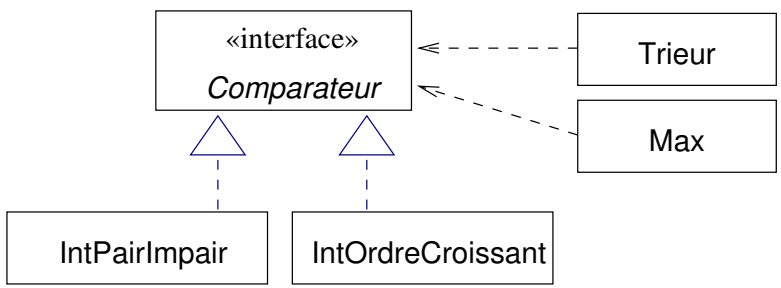
Exercice 1 Si I est une interface qui spécifie la méthode m() et p une poignée non nulle déclarée de type I, est-on sûr que p.m() a un sens ?

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 nouvelles classes utilisatrices (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`...).



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)

Surcharge et sous-typage

Qu'affiche l'exécution du programme principal suivant ?

```
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(c);
8      foo(q);
9  }
```

- le type réel des objets n'est pas connu pas le compilateur
- ce sont donc les types apparents qui sont utilisés pour la liaison statique

Affectation renversée

- **Problème posé :**

```
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

- **Transtypage :** Changer le type apparent : `(Type) expression`
  - Donne à expression le type apparent Type (accepté par le compilateur)
  - Cohérence contrôlée à l'exécution : Type doit être un type possible de expression sinon l'exception `ClassCastException` se produit !

```
C q = (C) p
```

- **Interrogation dynamique de type :** `expression` **`instanceof`** `Type`
  - vrai si Type est un type possible pour expression

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

```
if (p instanceof C) {
    ((C) p).m3();
    // non conseillée
}
```

- **`instanceof` est à utiliser avec modération !**

Classe anonyme

Exemple :

```
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     }
12 }
```

- La définition de la classe est donnée quand on crée un objet à partir de l'interface
- Intérêts :
  - Évite de nommer la classe (bof !)... mais on ne pourra pas la réutiliser !
  - Permet d'accéder aux variables locales (seulement si **`final`**)
- Il y a bien un **`.class`** engendré : `ExempleClasseAnonyme$1.class`

Sous-typage et programmation par contrat

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

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

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

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)

Questions sur trier :

- 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 ?

Question : En quoi les interfaces résolvent le problème posé sur les points ?

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 nouvelles classes utilisatrices peuvent être ajoutées
  - une classe qui définit une méthode max
- Elles pourront collaborer même si elles ne se connaissent pas
  - trier fonctionnera avec tous les comparateurs qui existent ou existeront
  - d'autres codes peuvent aussi s'appuyer sur cette notion de comparateur (max...)

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 !

La classe Couple

```java
/** Définition d'un couple. */
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 class ExempleCoupleErreurSimple {
   public static void main(String[] args) {
      Couple<String, Integer> c1 = new Couple<String, Integer>("I", 1);
      Couple<Integer, String> c2 = new Couple<Integer, String>(2, "II");
      c1.premier = "X";
      c1.second = 10;

      c1.premier = 3;
      c1.second = "III";
   }
}
```

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

Résultat de la compilation de ExempleCoupleErreurSimple

```
1  ExempleCoupleErreurSimple.java:8: error: incompatible types: int cannot be converted
       to String
2          c1.premier = 3;
3                       ^
4  ExempleCoupleErreurSimple.java:9: error: incompatible types: String cannot be
       converted to Integer
5          c1.second = "III";
6                      ^
7  2 errors
```

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

Explications

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

```
public class Couple<A, B> { ... }
```

Mais aussi des **interfaces** :

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

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

Convention : Pour le nom d'un paramètre de généricité, on utilise une lettre en majuscule qui correspond à l'initiale de l'information représentée

Instancier les paramètres de généricité : Pour obtenir une « classe » (appelée *type paramétré*), il faut préciser les paramètres de généricité (les valeurs des variables de type)

```
Couple<String, Point>...
Comparateur<Integer>            // et pas int !
```

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

Éviter la redondance de code

- mécanisme proche de la surcharge

Documentation :

- les types sont plus explicites :

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

- mais peuvent devenir bien longs !

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

Contrôle de type par le compilateur (voir T. 26)

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

Méthode générique : trier un tableau avec un comparateur

Exercice 2 : 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
13  }
```

```
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       }
```

Généricité contrainte : méthode max

Exercice 3 Obtenir le plus grand des éléments d'un tableau (sans Comparateur).

```
1  public class GeneriqueMax {
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 <T extends Comparable<T>> T max(T[] tab) {
6          T resultat = null;
7          for (T x : tab) {
8              if (resultat == null || resultat.compareTo(x) < 0) {
9                  resultat = x;
10             }
11         }
12         return resultat;
13     }
14
15     /** Programme de test de GeneriqueMax.max */
16     public static void main(String[] args) {
17         Integer[] ti = { 1, 2, 3, 5, 4 };
18         assert max(ti) == 5;
19
20         String[] ts = { "I", "II", "IV", "V" } ;
21         assert max(ts).equals("V");
22     }
23 }
```

Généricité contrainte : Explications

- Pour pouvoir déterminer le max, il faut comparer les éléments du tableau. Aussi, nous imposons qu'ils soient comparables (`java.lang.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  }
```

- Dans le code de 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). Voir Arrays.sort.

Remarque : Contraintes multiples : `C<T extends C & I1 & I2 & ... & In>`

Inteface fonctionnelles et Lambdas (Java8)

Inteface 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
15          trieur.trier(t1, (n1, n2) -> n1 - n2);
16          trieur.afficher("Croissant␣␣␣␣␣:␣", t1);
17  }   }
```

Remarque : Java8 ajoute aussi la notion de méthode par défaut dans les interfaces (méthodes avec un code et donc non abstraites).

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 }
```

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 !
- Java8 autorise aussi les méthodes **`static`** dans les interfaces.