Généricité
(polymorphisme paramétrique)
en Java

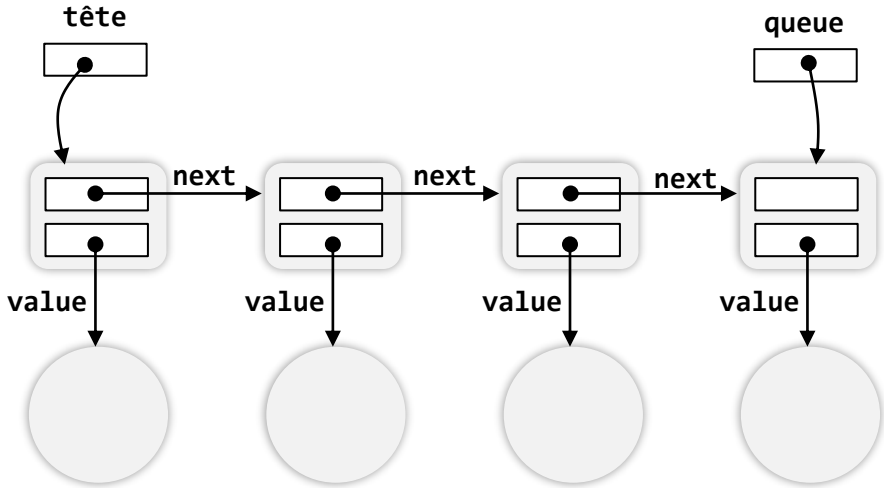
Philippe Genoud
dernière mise à jour : 10/02/2024 15:04

- Supposons que l'on développe du code pour gérer une file d'attente (FIFO First In First Out) et que l'on veuille utiliser ce code pour
  - *une file d'entiers*
  - *une file de chaînes de caractères (`String`)*
  - *une file d'objets `Personne`*
  - *…*

Comment procéder ?

- opérations
  - `ajouter(value)` : *insère élément en fin de file d'attente*
  - `value retirer()` : *retire et renvoie le premier élément de la file (`null` si la file est vide)*
  - `boolean estVide()` : *renvoie `true` si la file est vide, `false` sinon*

- implémentation
  - *une liste chaînée pour stocker les valeurs*
  - *un pointeur tête sur le premier élément de la liste*
  - *un pointeur queue sur le dernier élément de la liste*



- **1ère solution** : écrire une classe pour chaque type de valeur que l'on peut mettre dans la file

  File d'entiers

```
class ElementInt {

    private final int value;
    private ElementInt next;

    ElementInt(int val) {
        this.value = val;
        next = null;
    }

    void inserer(ElementInt elt) {
        this.next = elt;
    }

    int getValue() {
        return value;
    }

    ElementInt getNext() {
        return next;
    }
}
```

```
public class IllegalOpException
                  extends RuntimeException {

    public IllegalOpException(String mess) {
        super(mess);
     }
}
```

```
public class FileAttenteInt {

    private ElementInt tête = null;
    private ElementInt queue = null;

    public void ajouter(int val) {
        ElementInt elt = new ElementInt(val);
        if (estVide()) {
            tête = queue = elt;
        } else {
            queue.inserer(elt);
            queue = elt;
        }
    }

    public boolean estVide() {
        return tête == null;
    }

    public int retirer() {
       if (this.estVide()) {
            throw new IllegalOpException("liste vide");
       }
       int val = tête.getValue();
       tête = tete.getNext();
       return val;
    }
}
```

- **1ère solution** : écrire une classe pour chaque type de valeur que l'on peut mettre dans la file

File de Personnes

```
class ElementPersonne {

    private final Personne value;
    private ElementPersonne next;

    ElementPersonne(Personne val) {
        this.value = val;
        next = null;
    }

    void inserer(ElementPersonne elt) {
        this.next = elt;
    }

    Personne getValue() {
        return value;
    }

    ElementPersonne getNext() {
        return next;
    }
}
```

```
public class FileAttentePersonne {

    private ElementPersonne tête = null;
    private ElementPersonne queue = null;

    public void ajouter(Personne val) {
        ElementPersonne elt = new ElementPersonne(val);
        if (estVide()) {
            tête = queue = elt;
        } else {
            queue.inserer(elt);
            queue = elt;
        }
    }

    public boolean estVide() {
        return tête == null;
    }

    public Personne retirer() {
        if (this.estVide()) {{
           throw new IllegalOpException("liste vide");
        }
        Personne val = tête.getValue();
        tête = tête.getNext();
        return val;
    }
}
```

- Duplication du code → source d'erreurs à l'écriture et lors de modifications du programme
- Nécessité de prévoir toutes les combinaisons possibles pour une application

- 2ème solution : utiliser un type "universel", `Object` en Java
  - *toute classe héritant de* ***`Object`*** *il est possible d'utiliser* ***`FileAttente`*** *pour stocker n'importe quel type d'objet*

```
class Element {

    private final Object value;
    private Element next;

    Element(Object val) {
        this.value = val;
        next = null;
    }

    void inserer(Element elt) {
        this.next = elt;
    }

    public Object getValue() {
        return value;
    }

    public Element getNext() {
        return next;
    }
}
```

une file d'attente de personnes

```
FileAttente f1 = new FileAttente();
f1.add(new Personne(...));
```

une file d'attente d'événements

```
FileAttente f2 = new FileAttente();
f2.add(new Event(...));
```

```
public class FileAttente {

    private Element tête = null;
    private Element queue = null;

    public void ajouter(Object val) {
        Element elt = new Element(val);
        if (estVide()) {
            tête = queue = elt;
        } else {
            queue.inserer(elt);
            queue = elt;
        }
    }

    public boolean estVide() {
        return tête == null;
    }

    public Object retirer() {
        Object val = null;
        if (!estVide()) {
            val = tête.getValue();
            tête = tête.getNext();
        }
        return val;
    }
}
```

- Mais ...

```
FileAttente  f1 = new FileAttente();
f1.ajouter(new Personne(...));
...
// on veut récupérer le nom de la personne
// en tête de file.
String nom = (Personne)(f1.retirer()).getNom();
```

Transtypage

Object

- *obligation pour le programmeur d'effectuer un transtypage lorsqu'il accède aux éléments de la file d'attente*

```
f1.ajouter("Hello");
String nom = (Personne) (f1.retirer()).getNom();
```

ClassCastException

- *pas de contrôle sur les valeurs rangées dans la file d'attente*

→ *code lourd, moins lisible, plus difficile à maintenir*

→ *risques d'erreurs d'exécution.*

- Mais ...

```
FileAttente  f1 = n
f1.ajouter(new Perso
...
// on veut récupére
// en tête de file.
String nom = (Perso
```

Transtyp

ur d'effectuer un
x éléments de la file

```
f1.ajouter("Hello");
String nom = (Personne) (f1.retirer()).getNom
```

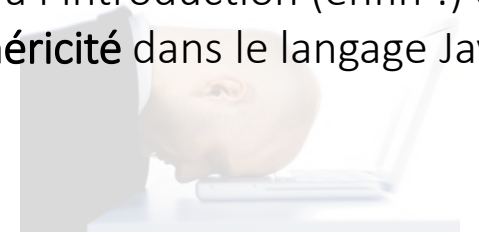
ClassCastException

s de contrôle sur les valeurs rangées dans la file
ttente

→ *code lourd, moins lisible, plus difficile*

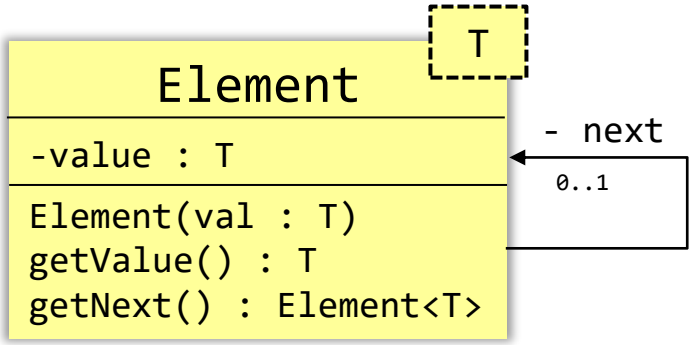
→ *risques d'erreurs d'exécution.*

depuis la version 1.5 de java (Tiger)
cela n'est qu'un mauvais souvenir
grâce à l'introduction (enfin !) de
la **généricité** dans le langage Java.



- La généricité (ou polymorphisme paramétrique) permet de faire des abstractions sur les types
  - *utiliser des types (Classes ou Interfaces) pour paramétrer une définition de classe ou d'interface ou de méthode → réutiliser le même code avec des types de données différents.*
  - *dans l'exemple précédent le type des éléments serait un paramètre des classes* `FileAttente` *et* `Element`

- Présente dans de nombreux langages de programmation avant introduction en Java: Eiffel, Ada, C++, Haskell, ...
- Présente dans le langage de modélisation UML



Les classes `Element` et `FileAttente` sont paramétrées en fonction d'un type formel `T`

T est utilisé comme type d'une ou plusieurs caractéristiques. Les classes manipulent des informations de type T. Elles ignorent tout de ces informations.

Dans le code du type générique, le paramètre de type peut être utilisé comme les autres types :

1. pour déclarer des variables
2. pour définir le type de retour de méthodes
3. pour définir le type des paramètres de méthodes
4. comme argument d'autres types génériques

```
public class FileAttente<T> {

    private Element<T> tête = null;
    private Element<T> queue = null;

    public void ajouter(T val) {
        Element<T> elt = new Element<T>(val);
        if (estVide()) {
            tête = queue = elt;
        } else {
            queue.inserer(elt);
            queue = elt;
        }
    }

    public boolean estVide() {
        return tête == null;
    }

    public T retirer() {
        T val = null;
        if (!estVide()) {
            val = tête.getValue();
            tête = tête.getNext();
        }
        return val;
    }
}
```

paramètre de type :représente un type inconnu au moment de la compilation

```
class Element<T> {

    private final T value;
    private Element next;

    Element(T val) {
        this.value = val;
        next = null;
    }

    void inserer(Element<T> elt) {
        this.next = elt;
    }

    public T getValue() {
        return value;
    }

    public Element<T> getNext() {
        return next;
    }
}
```

Les classes `Element` et `FileAttente` sont paramétrées en fonction d'un type formel `T`

T est utilisé comme type d'une ou plusieurs caractéristiques. Les classes manipulent des informations de type T. Elles ignorent tout de ces informations.

Dans le code du type générique, le paramètre de type peut être utilisé comme les autres types :

1. pour déclarer des variables
2. pour définir le type de retour de méthodes
3. pour définir le type des paramètres de méthodes
4. comme argument d'autres types génériques

```
public class FileAttente <T> {

    private Element <T> tête = null;
    private Element <T> queue = null;

    public void ajouter( T val) {
        Element <T> elt = new Element <T>(val);
        if (estVide()) {
            tête = queue = elt;
        } else {
            queue.inserer(elt);
            queue = elt;
        }
    }

    public boolean estVide() {
        return tête == null;
    }

    public T retirer() {
        T val = null;
        if (!estVide()) {
            val = tête.getValue();
            tête = tête.getNext();
        }
        return val;
    }
}
```

paramètre de type :représente un type inconnu au moment de la compilation

```
class Element <T> {

    private final T value;
    private Element next;

    Element( T val) {
        this.value = val;
        next = null;
    }

    void inserer(Element <T> elt) {
        this.next = elt;
    }

    public T getValue() {
        return value;
    }

    public Element <T> getNext() {
        return next;
    }
}
```

Lors de l'instanciation de `FileAttente<T>` le type formel `T` est remplacé par un type (Classe ou Interface) existant

le type est passé en argument

une file d'attente de personnes

```
FileAttente<Personne> f1 = new FileAttente<Personne>();
f1.add(new Personne(...));
```

une file d'attente d'événements

```
FileAttente<Event> f2 = new FileAttente<>();
f2.add(new Event(...));
```

Diamond
on n'est pas obligé de mentionner les arguments de type dans le constructeur

Java 7

sans généricité (< JDK 5)

```
FileAttente  f1 = new FileAttente();
f1.ajouter(new Personne(...));
...
// on veut récupérer le nom de la personne
// en tête de file.
String nom = (Personne)(f1.retirer()).getNom();
```

Transtypage obligatoire

```
f1.ajouter("Hello");
String nom = (Personne) (f1.retirer()).getNom();
```

ClassCastException

Erreur à l'exécution

avec généricité (JDK 5+)

```
f1.ajouter(new Personne(...));
...
// on veut récupérer le nom de la personne
// en tête de file.
String nom = f1.retirer().getNom();
```

Plus de transtypage

Erreur détectée dès la compilation

```
f1.ajouter("Hello");
String nom = f1.retirer().getNom();
```

Type incorrect

→ La généricité simplifie la programmation, évite de dupliquer du code et le rend plus robuste

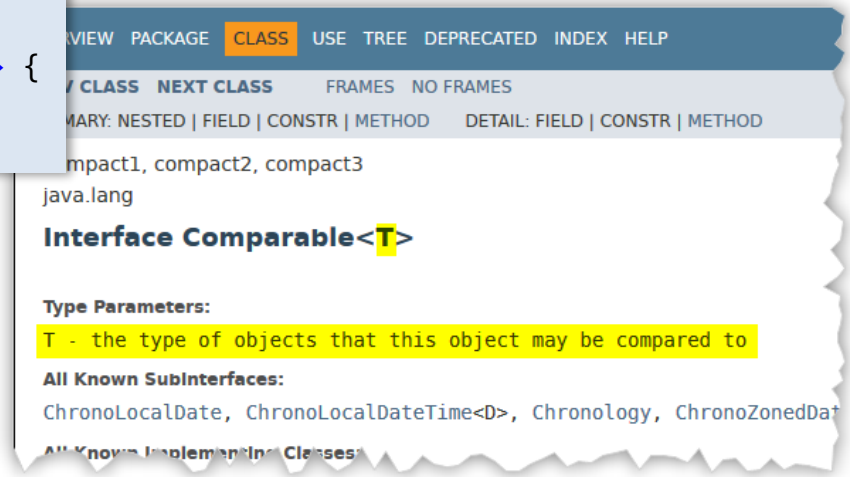
- **Type générique** : Classe ou interface paramétrée par un ou plusieurs types

classe générique à deux paramètres

```
public class Paire<T1, T2> {

    private final T1 first;
    private final T2 second;

    public Paire(T1 first, T2 second) {
           this.first = first;
           this.second = second;
    }

    public T1 getFirst() { return first; }
    public T2 getSecond() { return second; }
}
```

interface générique

```
pacakge java.lang;

public interface Comparable<T> {
    int compareTo(T o);
}
```



- Instanciation (ou invocation) d'un type générique consiste à valuer les paramètres de type
  - *Les arguments de type (paramètres effectifs) peuvent être :*
    - *des classes(concrètes ou abstraites)*
    - *des interfaces*
    - *des types paramétrés*
    - *des paramètres de type*

```
Paire<String, Forme> c1 = new Paire<>("Cercle 1",new Cercle(0,0,10));
```

```
Paire<String, IDessinable> c2 = new Paire<>("Visage 1", new VisageRond());
```

```
Paire<String, Paire<String,Forme>> = new Paire<>("Paire 3",c1);
```

```
public class FileAttente<E> {
    private Element<E> tête = null;
    ...
```

Implémentation

Compilation de code générique

- compilation de code générique basée sur le mécanisme d'effacement (*erasure*)

```
public class Paire<T1,T2> {
    private final T1 first;
    private final T2 second;

  public Paire(T1 first, T2 second) {
        this.first = first;
        this.second = second;
  }

  public T1 getFirst() {
        return first;
  }

  public T2 getSecond() {
        return second;
  }

}
```

toutes les informations de type placées entre chevrons sont effacées

Les variables de type sont remplacées par **`Object`** (en fait leur borne supérieure en cas de généricité contrainte)
voir slides suivants

javac

```
public class Paire {
    private final Object first;
    private final Object second;

    public PaireO(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() {
        return first;
    }

    public Object getSecond() {
        return second;
    }
}
```

Le type `Paire<T1,T2>` a été remplacée par un type brut (*raw type*) en substituant `Object` à `T1` et `T2`

`Paire.class`

```
Paire<String, String> c1;
c1 = new Paire<>("Hello","World");
```

```
String s = c1.getFirst();
```

insertion d'opérations de transtypage si nécessaire

```
String s = (String) c1.getFirst();
```

```
Personne p = c1.getFirst();
```

rejeté dès la compilation : une `String` n'est pas une `Personne`

javac

conséquences de l'effacement de type

- permet d'assurer la compatibilité ascendante du code.
- à l'exécution, il n'existe qu'une classe (le type brut , *raw type*) partagée par toutes les instanciations du type générique

```
Paire<String,String> c1;
Paire<Personne,Personne> p1;
c1 = new Paire<>("Hello","World");
p1 = new Paire<>(new Personne("DURAND", "Sophie"),new Personne("DUPONT", "Jean"));
System.out.println( c1.getClass() == p1.getClass() ); → true
```

En C++
création de code source spécifique pour chaque valeur de type

- mais cela induit un certain nombre de limitations
  - *les membres statiques d'une classe paramétrée sont partagés par toutes les instanciations de celle-ci*

```
public class Paire<T1,T2> {
  private static int nbInstances = 0;
  private final T1 first;
  private final T2 second;

  public Paire(T first, T2 second) {
        nbInstances++
        this.first = first;
        this.second = second;
  }

  public static int getNbInstances() {
        return nbInstances;
  }

  ...
```

```
Paire<String,String> c1;
Paire<Personne,Personne> p1;
c1 = new Paire<>("Hello","World");
p1 = new Paire<>(new Personne("DURAND", "Sophie"),
            new Personne("DUPONT", "Jean"));
System.out.println( Paire.getNbInstances()  ); → 2
```

conséquences de l'effacement de type

- limitations…
  - *seul le type brut est connu à l'exécution*

```
Paire<String, String> p1 = new Paire<>("Hello","World");
Paire<String, Personne> p1 = new Paire<>("personne 1", new Personne("DURAND", "Sophie"));
System.out.println( p1.getClass() == p2.getClass() );  → true
System.out.println( p1 instanceof Paire ); → true
System.out.println( p2 instanceof Paire ); → true
```

  - *on ne peut utiliser un type générique instancié dans un contexte de vérification de type*

```
✖ if ( p1 instanceof Paire<String, String>) { ... }
```

  - *dans un contexte de coercition de type (transtypage) on pourra avoir une erreur à l'exécution mais pas nécessairement là où on l'attend…*

```
Object obj = p2;
Paire<String, String> p3 = (Paire<String, String>) obj;
...
String s2 = p3.getSecond();
```

ClassCastException: Personne cannot be cast to String

  - *on ne peut instancier de tableaux d'un type générique*

```
 Paire<Personne,Personne>[] duos;
✖ duos = new Paire<Personne,Personne>[100] ;

✔ duos = new Paire[100];   // mais utiliser le type brut est possible
✔ duos[0] = p2;
✖ duos[1] = p1;
```

conséquences de l'effacement de type

- limitations…

```
public class MaClasseC<T> {

  ✔ T x1;
  ✔ T[] tab;
  ✖ static T x2;


  ✔ T method1(T param) {
       ...
    }


  ✖ static T method2(T param) {
       ...
    }

    void method3(...) {
       ...
  ✖   x1 = new T();
  ✖   tab = new T[10];
    }
    ...
}
```

✖ erreur de compilation

les membres statiques d'une classe paramétrée ne peuvent utiliser ses paramètres de type

✖ erreur de compilation

au sein d'une classe paramétrée on ne peut pas utiliser les paramètres de type pour instancier un objet (simple ou tableau)

- **Méthodes génériques** : méthodes qui définissent leur propres paramètres de type.
  - *similaire aux types génériques mais la portée du paramètre de type est limitée à la méthode où il est déclaré.*
  - *méthodes génériques peuvent être définies dans des types non génériques*

exemple : méthode `copyOf` de la classe `java.util.Arrays`
permet de créer un tableau d'une taille donnée qui est une copie d'un tableau passé en paramètre

Arrays propose des méthodes utilitaires pour faire de recherches, tris, copies de tableaux…

Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `null`. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of exactly the same class as the original array.

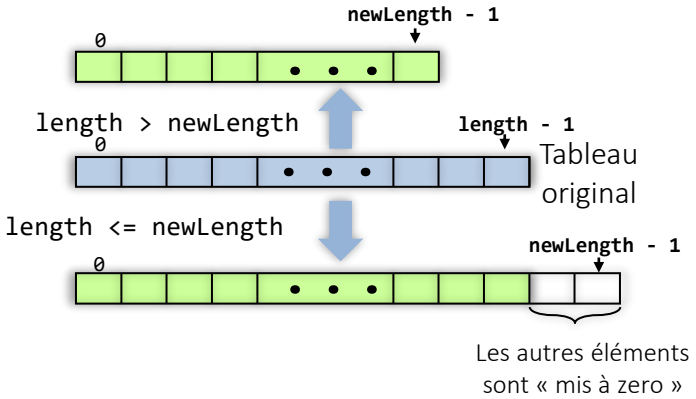
Parameters:

`original` - the array to be copied — un type tableau

`newLength` - the length of the copy to be returned — `int`

Returns: — le même type tableau que pour `original`

a copy of the original array, truncated or padded with nulls to obtain the specified length



Quelle signature pour cette méthode ?

méthodes génériques

- déclaration de **copyOf**

Il faut que ces types soient les mêmes

```
public static TypeTableau copyOf(TypeTableau original, int newLength)
```

Pour les types simples : surcharge de la méthode

```
public int[]  copyOf(int[] original, int newLength)
public float[]  copyOf(float[] original, int newLength)
   . . .
public boolean[]  copyOf(boolean[] original, int newLength)
```

Mais pour les types objets ? `Object[]` ≠ `Personne[]`

→ définition d'une **méthode générique**

`T` : un type java (classe ou interface)

```
public static <T> T[] copyOf(T[]original, int newLength)
```

Comme pour les classes génériques, le paramètre de type doit être déclaré, déclaration entre < > avant le type de retour de la méthode

- invocation d'une méthode générique

```
Personne[] tab1 = new Personne[200];
...
Personne[] tab2 = Arrays.<Personne[]>copyOf(tab1, 100);
```

```
Personne[] tab2 = Arrays.copyOf(tab1, 100);
```

argument de type peut être ignoré à l'invocation... mais attention on verra plus tard les conséquences

- Dans l'exemple précédent la méthode `copyOf` est statique, il est aussi possible de définir des méthodes non statiques génériques.

Méthode générique dans une classe non générique.

```
public class Class1{

    private int x;
    ...

    public Class1(...) {
       ...
    }

    public <T1,T2> T1 methodX(T1 p1, T2 p2) {
        ...
    }

    ...
}
```

teste si la seconde composante de la paire (`this`) est égale à la seconde composante d'une autre paire dont la première composante n'est pas forcément du même type que celle de `this`

Méthode générique dans une classe générique.

```
public class Paire<T1, T2> {

    private final T1 first;
    private final T2 second;

    public Paire(T1 first, T2 second) {
           this.first = first;
           this.second = second;
    }

    public T1 getFirst(){
      return first;
    }

    public T2 getSecond(){
      return second;
    }

    public <T3> boolean sameScnd(Paire<T3, T2> p) {
        return getSecond().equals(p.getSecond());
    }

}
```

Le (les) paramètre(s) de type de la méthode générique ne fait (font) pas partie des paramètres de type de la classe.

- Il existe des situations où il peut être utile d'imposer certaines contraintes sur les paramètre de type (pour une classe, interface ou une méthode générique).
- **généricité contrainte** (*bounded type parameters*)
  - *impose à un argument de type d'être dérivé (sous-classe) d'une classe donnée ou d'implémenter une ou plusieurs interfaces.*

Exemple avec une classe générique

```
public class Couple<T extends Personne> {

    private final T first;
    private final T second;

    public Couple(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }

    public T getSecond() { return second; }

    public int getAgeMoyen() {
        return (first.getAge() + second.getAge()) / 2;
    }
}
```

la classe couple ne pourra être instanciée qu'avec le type Personne ou un type dérivé

possibilité d'invoquer les méthodes définies dans la borne du type paramétré

```
Couple<Personne> cp;   ✔
Couple<Etudiant> ce;   ✔
```

```
Couple<Point> cpt;   ✖
```

erreur de compilation un point n'est pas une personne

A la compilation, le mécanisme d'effacement de type consiste à remplacer T par sa borne supérieure

```
public class Couple {

    private final Personne first;
    private final Personne second;

    public Couple(Personne first, Personne second) {
    ...
    }
    ...
}
```

type brut (*raw type*)

```
ce.getFirst();   →   (Personne) ce.getFirst();
```

- borne supérieure peut être soit une classe (concrète ou abstraite) soit une interface

```
<T1 extends T2>
```

`extends` est interprété avec un sens général, si `T1` est une classe et `T2` une interface `extends` doit être compris comme `implements`

- possibilité de définir des bornes multiples (plusieurs interfaces, une classe et une ou plusieurs interfaces)

```
<T extends B1 & B2 & B3>
```

T est un sous type de tous les types listés et séparés par **&**

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }
```

si l'une des bornes est une classe (et il ne peut y en avoir qu'une seule, car héritage simple en Java), celle-ci doit être spécifiée en premier

```
class D <T extends A & B & C> {
 ...
}
```



```
class D <T extends B & A & C> {
 ...
}
```


A la compilation, le mécanisme d'effacement de type consiste à remplacer **T** par la première de ses bornes supérieures (ici **A**)

- Dans l'exemple précédent (`Couple`) n'aurait-il pas été possible de réutiliser le code de la classe générique `Paire<T1,T2>` ?

```java
public class Paire<T1, T2> {

    private final T1 first;
    private final T2 second;

    public Paire(T1 first, T2 second) {
            this.first = first;
            this.second = second;
    }

    public T1 getFirst(){
      return first;
    }

    public T2 getSecond(){
      return second;
    }

}
```

```java
public class Couple<T extends Personne> {

    private final T first;
    private final T second;

    public Couple(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }

    public T getSecond() { return second; }

    public int getAgeMoyen() {
        return (first.getAge() +
                    second.getAge()) / 2;
    }
}
```

```java
public class Couple<T extends Personne> extends Paire<T, T> {

    Couple(T val1, T val2) {
        super(val1, val2);
    }

    public int getAgeMoyen() {
        return (getFirst().getAge() + getSecond().getAge()) / 2;
    }
}
```

Il est possible de sous typer une classe ou une interface générique en l'étendant ou en l'implémentant

héritage de classes génériques

- différentes manières de dériver (sous-classer) une classe générique

```
class ClasseA <T> { … }
```

  - *en conservant les paramètres de type de la classe de base*

```
class ClasseB<T> extends ClasseA<T>{
… }
```

  - *en ajoutant de nouveaux paramètres de type*

```
class ClasseB<T,U> extends ClasseA<T>{
… }
```

  - *en introduisant des contraintes sur un ou plusieurs des paramètres de la classe mère*

```
class ClasseB<T extends TypeC> extends ClasseA<T>{
… }
```

  - *en spécifiant une instance particulière de la classe mère*

```
class ClasseB extends ClasseA<String>{
… }
```

```
class ClasseB<T> extends ClasseA<String>{
… }
```

- situations incorrectes

l'inverse est possible, une classe générique peut hériter d'une classe non générique `class ClasseB<T> extends ClasseC { … }`

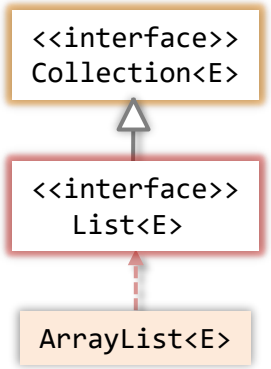
  - *`ClasseB` doit disposer au moins du paramètre `T`*

```
class ClasseB extends ClasseA<T>{
… }
```

  - *des contraintes ne peuvent être ajoutées sur les paramètres de la classe mère*

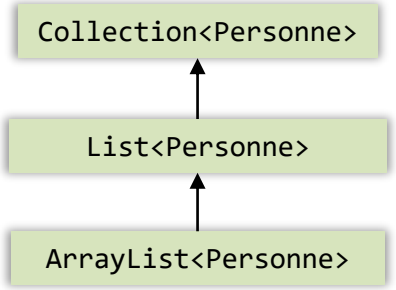
```
class ClasseB<T> extends ClasseA<T extends TypeC>{
… }
```

Généricité sous-typage

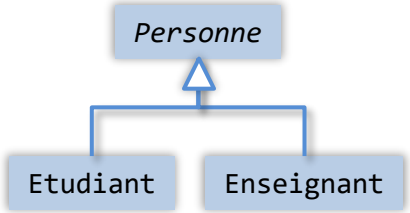


`ArrayList<E>` implémente `List<E>`
`List<E>` étend `Collection<E>`

→ `ArrayList<Personne>` est un sous type de `List<Personne>` qui est un sous type de `Collection<Personne>`




```
List<Personne> lesPersonnes = new ArrayList<Personne>();
```



Qu'en est-il de `ArrayList<Etudiant>` , est-ce une liste de `Personne`s ?

```
List<Personne> lesPersonnes = new ArrayList<Etudiant>();
```

⊗ Type mismatch: cannot convert from ArrayList<Etudiant> to List<Personne>

supposons que les types soient compatibles : on pourrait écrire

```
List<Etudiant> lesEtudiants = new ArrayList<Etudiant>(); ①
...
List<Personne> lesPersonnes = lesEtudiants; ②
```

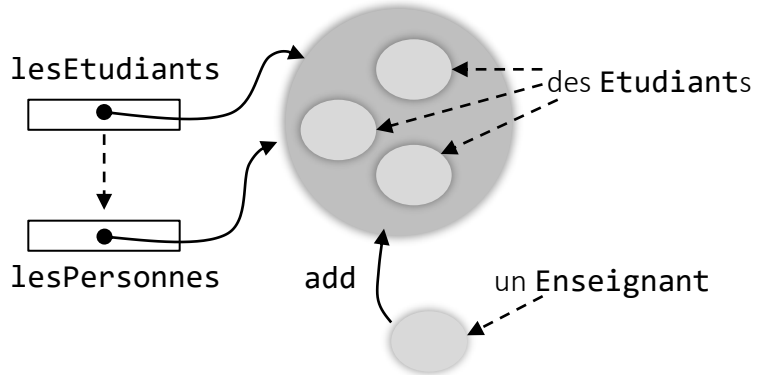
on pourrait ainsi ajouter un enseignant à la liste des étudiants en passant par `lesPersonnes`

```
lesPersonnes.add(new Enseignant("DUPOND","Jean"));
```

mais alors l'exécution de

```
lesEtudiants.get(lesEtudiants.size()-1);
```

donnerait une erreur de transtypage



C'est pourquoi le compilateur interdit l'instruction ② ⊗ Type mismatch: cannot convert from List<Etudiant> to List<Personne>

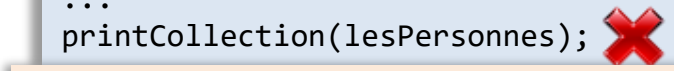
Jokers (wildcards)

- pb: écrire une méthode qui affiche tous les éléments d'une collection*
  - *approche naïve*

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

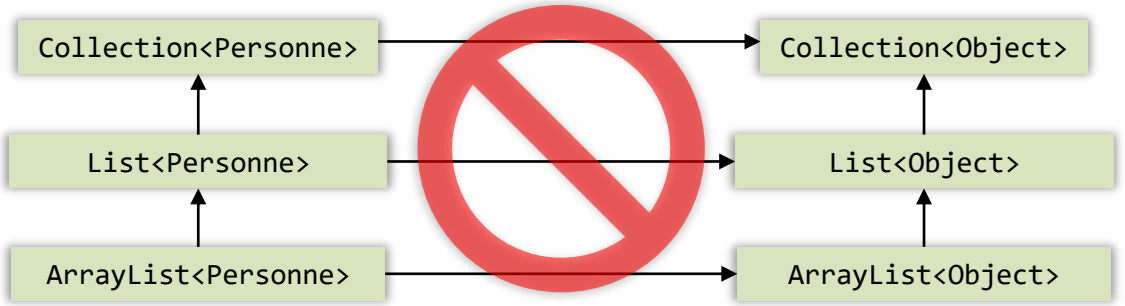
  - *pourquoi cette solution ne marche t'elle pas ?*

```
List<Personne> lesPersonnes = new ArrayList<Personne>();
...
printCollection(lesPersonnes);
```

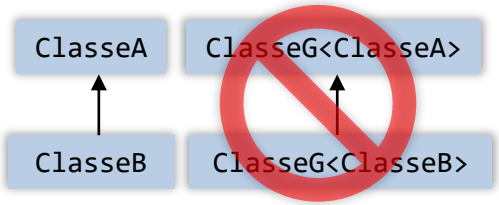


incompatibles types: List<Personne> cannot be converted to Collection<Object>

`Collection<Object>` n'est pas une super classe de n'importe quel type de Collection



de manière générale



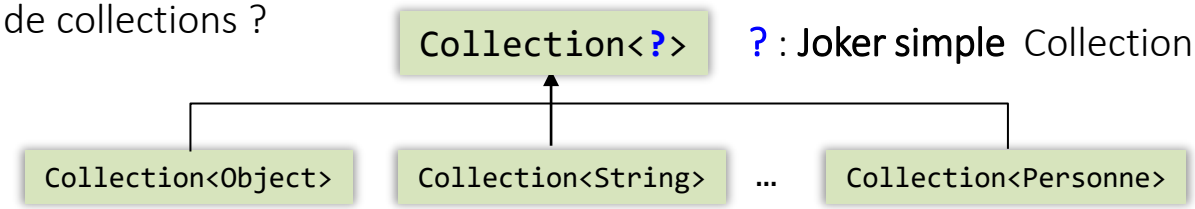
Pourtant il est toujours possible d'utiliser un objet de type `ClasseG<ClasseB>` comme un objet de `type ClasseG<ClasseA>` tant que l'on ne cherche pas à modifier sa valeur.

→ pour offrir cette possibilité les concepteurs de Java ont introduit la notion de **Joker** (*wildcard*)

exemple inspiré de : *Java tutorial, Lesson Generics* de Gilad Bracha https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html

Joker simple

Quel super-type pour les types de collections ?



- En utilisant un Joker pour typer le paramètre **`Collection`** de **`printCollection`**, il est ensuite possible d'utiliser cette méthode sur n'importe quel type de collection

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

```
List<Personne> lesPersonnes = new ArrayList<Personne>();
...
printCollection(lesPersonnes);
```

- Par contre le type des éléments de **`c`** étant inconnu, on ne peut lui ajouter des objets

```
Collection<?> c = new ArrayList<Personne>();
c.add(new Personne("DUPONT", "Jean"));
```

erreur détectée à la compilation

- De manière générale, ce ne sont pas simplement les modifications d'un objet de type générique `ClasseG<?>` qui sont interdites mais tout appel de méthode recevant un argument de type correspondant à **`?`**

```
FileAttente<Personne> fp = new FileAttente<>();
Personne p = new Personne("DUPONT", "Jean");
fp.ajouter(p);
FileAttente<?> f = fp;
f.contient (p);
```

jokers avec limitations

- comme pour les paramètres de type il est possible d'imposer des restrictions à un joker

- joker avec borne supérieure `<? extends T>`
  - *exemple la méthode* ***`addAll`*** *de* ***`Collection<E>`***

addAll

```
boolean addAll(Collection<? extends E> c)
```

Adds all of the elements in the specified collection to this collection

une collection de n'importe quel type qui
étend le type des éléments de cette collection

```
List<Etudiant> lesEtudiants = new ArrayList<>();
...
List<Personne> lesPersonnes = new ArrayList<>();
lesPersonnes.add(new Enseignant("DUPONT","Jean"));
lesPersonnes.add(new Etudiant("DURAND","Anne"));
lesPersonnes.addAll(lesEtudiants);
```

  - *comme pour les jokers simples, il n'est pas possible d'invoquer des méthodes ayant un argument correspondant à ? (risques de modifier l'objet de type générique avec une valeur dont le type ne serait pas compatible avec son type effectif)*

```
List<Etudiant> lesEtudiants = new ArrayList<>();
...
List<? Personne> l1 = lesEtudiants;
l1.add(new Etudiant("DURAND","Anne"));
```

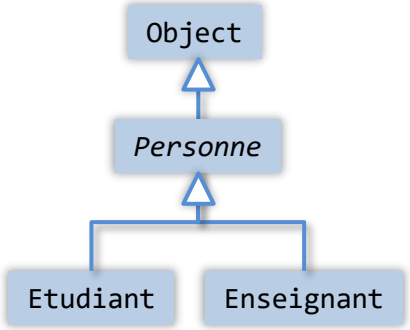
erreur de compilation

```
no suitable method found for add(Etudiant)
   method Collection.add(CAP#1) is not applicable
     (argument mismatch; Etudiant cannot be converted to CAP#1)
   method List.add(CAP#1) is not applicable
     (argument mismatch; Etudiant cannot be converted to CAP#1)
 where CAP#1 is a fresh type-variable:
   CAP#1 extends Personne from capture of ? extends Personne
```

jokers avec limitations

- joker avec borne inférieure `<? super T>`
  - *de la même manière qu'un joker avec une borne supérieure restreint le type inconnu à un type spécifique ou un sous-type de ce type spécifique, un joker avec une borne inférieure restreint le type inconnu à un type spécifique ou à un super type de ce type spécifique.*
  - *exemple: supposons que l'on veuille ajouter à la classe* ***`Etudiant`*** *une méthode qui permet d'ajouter l'étudiant à une liste*

Il n'est pas possible de spécifier simultanément une borne supérieure et une borne inférieure



quel type pour **uneListe** ?

```
/**
 * ajoute cet étudiant à une liste
 * @param uneListe la liste à laquelle cet etudiant est ajouté
*/
void ajouterA(List<? super Etudiant> uneListe){

    uneListe.add(this);
}
```

une liste peut être:
- une liste d'Etudiants `List<Etudiant>`
- une liste de Personnes `List<Personne>`
- une liste d'objets `List<Object>`

mais ne peut pas être une liste d'autre chose `List<String>`, `List<Enseignant>` …

à l'inverse des jokers simples ou des jokers avec borne supérieure, il est possible d'invoquer des méthodes ayant un argument correspondant à la borne inférieure

(cela ouvre la possibilité de modifier l'objet de type générique avec une valeur dont le type correspond à la borne inférieure). Par contre un transtypage est nécessaire pour les appels de méthodes dont le type de retour correspond au joker.

```
List<Personne> lp = new ArrayList<>();
lp.add(new Personne("DUPONT","Jean"));
lp.add(new Etudiant("DUPOND", "Marcel"));
List<? super Etudiant> l1 = lp;
l1.add(new Etudiant("DURAND", "Anne"));
Personne p = (Personne) l1.get(0);
l1.add(new Personne("DURANT", "Sophie"));
```

erreur de compilation

The method add(capture#4-of ? super Etudiant) in the type List<capture#4-of ? super Etudiant> is not applicable for the arguments (Personne)

- The Java Tutorials
  - *Lesson: Generics by Gilad Bracha*
    https://docs.oracle.com/javase/tutorial/extra/generics/index.html
  - *Lesson: Generics (Updated)*
    https://docs.oracle.com/javase/tutorial/java/generics/index.html

- Programmer en Java – 9ème Edition
  Claude Delannoy, Ed. Eyrolles, mai 2014
  - *chapitre 21 : La programmation générique*

- Génériques et collections Java
  Maurice Naftalin et Philip Wadler, Ed O'Reilly, juillet 2007