Programmation et Algorithmique II

Ch.9 – *Generics*

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

Generics

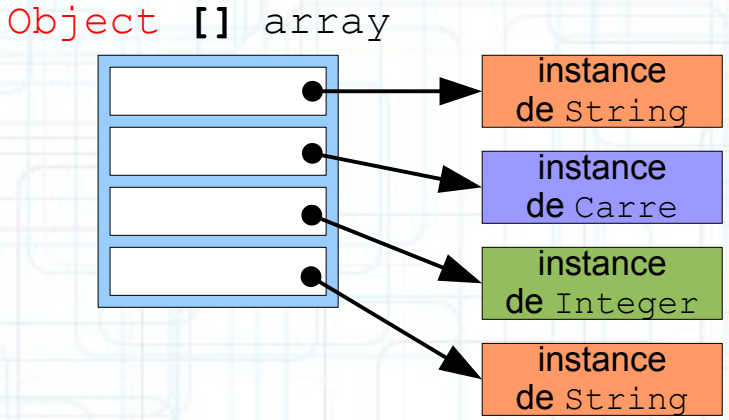
- **Introduction**
  - Un tableau ou une collection peuvent être utilisés pour gérer des éléments de types divers. On dit que ce sont des structures de données **génériques**.
    - Exemple : la même classe `ArrayList` peut manipuler des instances des classe `String` et `Carre`.
  - Jusqu'à présent, cette généricité était obtenue par le biais de l'**héritage** et des **références polymorphiques**.
    - Exemple : la classe `ArrayList` contient des références de type `Object` et, par héritage, des instances de n'importe quelle classe.
  - Cette approche souffre de deux inconvénients principaux

    1) nécessité d'utiliser le transtypage

    2) certaines erreurs de types ne sont pas détectables par le compilateur

Generics

- **Polymorphisme avec un tableau**
  - Exemple : Grâce aux *références polymorphiques*, un tableau dont les cellules sont des références `Object` peut référencer des instances de n'importe quelle *sous-classe* d'`Object`.



    - Appeler une méthode commune à chacune des instances (p.ex. `toString`) est un exemple de mise en oeuvre du polymorphisme. Le comportement exact de `toString` dépend d'une instance à l'autre (grâce à la re-définition de méthode et au *late-binding*), bien que les objets soient manipulés de la même façon.

Generics

- **Introduction**
  - Implémenter une classe générique en utilisant des références ce type `Object` a deux inconvénients principaux.

```
List l= new ArrayList();
l.add(new Integer(5));
l.add(new String("13"));
l.add(new Integer(7));

MyIterator i= l.iterator();
int total= 0;
int count= 0;
while (i.hasNext()) {
  total+= (Integer) i.getNext();
  count++;
}
System.out.println("La moyenne est "+(total/count));
```

Second inconvénient :
il est possible d'ajouter des instances de classes incompatibles !

Le compilateur ne peut pas nous aider pour détecter cette erreur (il n'a pas suffisamment d'information)

Premier inconvénient :
il est nécessaire d'utiliser un *transtypage* lorsque l'on récupère les éléments de la collection

Generics

- **Introduction**
  - Il nous faudrait un mécanisme pour restreindre le type des objets manipulables !
  - Depuis la version 5.0 du JDK, ce mécanisme appelé **Generics** (***polymorphisme paramétrique***) est disponible dans le langage Java. Ce mécanisme permet d'indiquer au compilateur des restrictions sur les types.
  - Il est désormais possible d'implémenter des classes génériques en restreignant le type des classes spécifiées à l'aide d'un ou plusieurs **paramètres de type**.
    - Note : souvenez-vous de la classe `ArrayList` dans laquelle nous avons déjà rencontré ce mécanisme.

Generics

- **Introduction**
  - Comment créer un nouveau type générique ?
    Et comment l'utiliser ?
  - Comment créer une méthode générique ?
  - Peut-on utiliser n'importe quel paramètre de type ?
  - Quelles sont les règles d'héritage entre types génériques ?

Table des Matières

Generics

- **Déclaration de type générique**
  - Une **déclaration de type générique** s'effectue en ajoutant un ou plusieurs paramètres de type à une déclaration de classe ou d'interface.

```
modificateurs class nomClasse < nomParamType1, ..., nomParamTypeN >
```

    - Un paramètre de type pourra désigner, par la suite, n'importe quelle classe ou interface mais pas un type primitif.
    - Les paramètres de type peuvent remplacer des types dans le corps de la classe ainsi définie.
  - Exemple :

```
public class ArrayList<E> ...

public class HashMap<K,V> ...
```

Generics

- **Invocation de type générique**
  - L'utilisation d'un type générique est appelée **invocation de type générique**. Le type ainsi créé est appelé *type paramétré*.

nomClasse < *nomType1*, ..., *nomTypeN* >

    - *nomType* peut être un nom de classe, d'interface ou un autre paramètre de type.
    - Il doit y avoir autant de types dans l'invocation que de paramètres de type dans la déclaration.

  - Exemple : l'extrait ci-dessous montre des invocations du type `ArrayList` (déclaration de variable et instanciation).

```
ArrayList<Integer> l= new ArrayList<Integer>();
```

  - Depuis Java 7, il est possible d'écrire

```
ArrayList<Integer> l= new ArrayList<>();
```

Le « ***diamant*** » indique que la valeur du paramètre de type est identique à celle utilisée dans le membre de gauche de l'affectation.

Generics

- **Invocation de type générique**
  - Exemple : Le *type paramétré* `ArrayList<Integer>` peut être vu comme une redéfinition d'`ArrayList` spécialisée pour gérer des instances d'`Integer` .

```
public class ArrayList<E> implements List<E>, ... {
   ...
   public boolean add(E o);
   public boolean add(int index, E o);
   ...
   public E get(int index);
   ...
}
```

`ArrayList<Integer>`

```
public class ArrayList<Integer> implements List<Integer>, ... {
   ...
   public boolean add(Integer o);
   public boolean add(int index, Integer o);
   ...
   public Integer get(int index);
   ...
}
```

Generics

- **Avantages**
  - Les deux inconvénients indiqués en début de section – nécessité du transtypage et risque d'ajouter des instances incompatibles – sont maintenant supprimés.
  - Le compilateur dispose d'assez d'information pour effectuer des vérifications supplémentaires.

```
ArrayList<Integer> l= new ArrayList<Integer>();
l.add(new Integer(5));
l.add(new String("13"));
l.add(new Integer(7));

int total= 0;
for (int i= 0; i < l.size(); i++)
  total+= l.get(i);
System.out.println("La moyenne est "+(total/l.size()));
```

Le compilateur peut générer une erreur : types incompatibles

Plus besoin de transtypage...

Generics

- ***Generics* et machine virtuelle**
  - Les informations de types ajoutées avec les *generics* **ne sont disponibles que pour le compilateur**; elles sont enlevées lors de la compilation (mécanisme appelé « *type erasure* »). Ceci a deux conséquences.
  - **Compilateur**
    - Peut effectuer des vérifications statiques de compatibilité.
    - Exemple : `ArrayList<String>` et `ArrayList<Integer>` sont considérés comme deux types incompatibles par le compilateur.
  - **Machine virtuelle**
    - Ne peut pas effectuer de vérification dynamique de compatibilité.
    - Exemple : `ArrayList<String>` et `ArrayList<Integer>` sont vus comme `ArrayList` et sont donc compatibles du point de vue de la machine virtuelle.

Generics

- **JVM : *type erasure***
  - Pour se convaincre que la machine virtuelle ne dispose pas des paramètres de type, considérons deux instances de la classe `ArrayList` obtenues par des invocations avec des valeurs différentes du paramètre type `E` : `String` et `Integer`.

```
ArrayList<Integer> alInt= new ArrayList<Integer>();
ArrayList<String> alStr= new ArrayList<String>();
System.out.println(alInt.getClass() == alStr.getClass());
                        /* affichera true */
```

Les deux appels retournent la même classe : `ArrayList`

  - Le compilateur n'a pas créé 2 nouvelles classes `ArrayList<Integer>` et `ArrayList<String>`. Les deux instances sont issues de la même classe `ArrayList`.

Table des Matières

Generics

- **Déclaration de méthode générique**
  - Il est possible de définir des *méthodes génériques* qui ont leur propre(s) paramètre(s) de type.
  - La déclaration d'une telle méthode suit la syntaxe suivante

modificateurs **<** *nomParamType* **>** *typeRetour* *nomMethode* **(** *arguments* **)**

    - Ici aussi, il est possible d'avoir plusieurs paramètres de type dans la déclaration de la méthode en les séparant par des virgules.
    - Les paramètres de type peuvent être utilisés dans le corps de la méthode, mais aussi dans la définition de ses arguments et de son type de retour.

Generics

- **Déclaration de méthode générique**
  - Exemple

```
public class ArrayAlgorithms
{
  public static <T> T getMiddle(T[] array) {
    return array[array.length / 2];
  }
  /* ... */
}
```

```
String[] users= { "James", "Q", "M", "Mrs MoneyPenny" };
String middle= ArrayAlgorithms.<String>getMiddle(users);
```

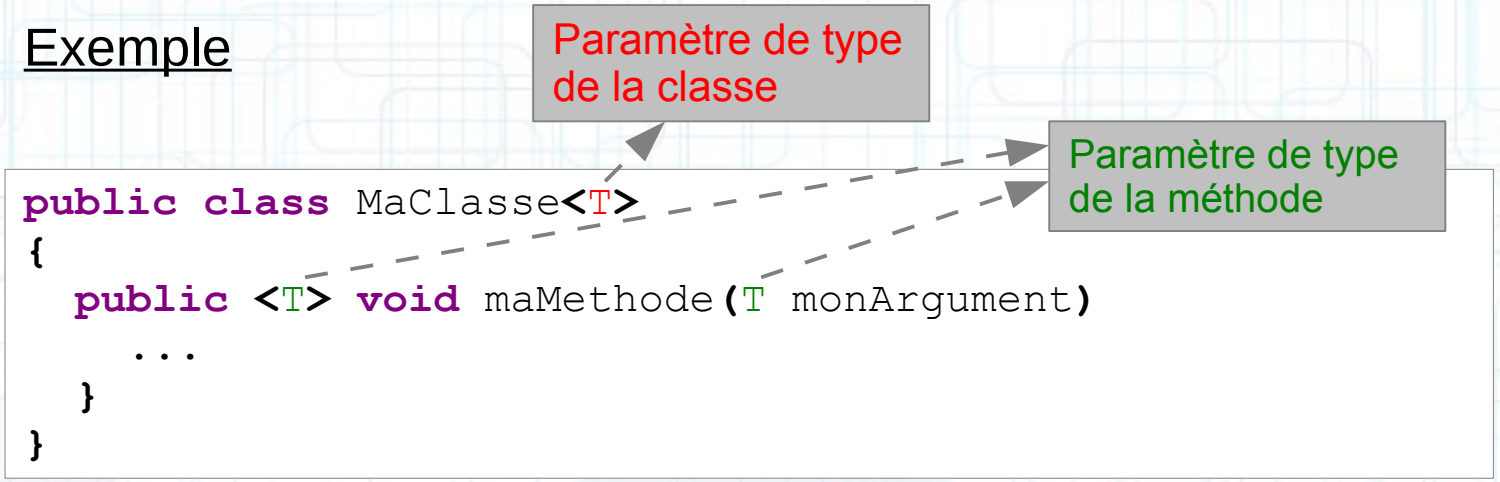
  - Note : dans la plupart des cas, le compilateur est capable d'inférer la valeur à utiliser pour le paramètres de type de la méthode. Il n'est alors pas nécessaire de spécifier cette valeur explicitement.

```
middle= ArrayAlgorithms.getMiddle(users);
```

Implicitement
`ArrayAlgorithms.<String>getMiddle(...)`

Generics

- **Déclaration de méthode générique**
  - Remarque : si un paramètre de type utilisé dans une déclaration de méthode générique a le même nom qu'un paramètre de type de la classe englobante, il s'agit d'une re-définition.
  - Exemple




```
public class MaClasse<T>
{
  public <T> void maMethode(T monArgument)
    ...
  }
}
```

```
MaClasse<Integer> mc= new MaClasse<Integer>();
mc.<String>maMethode("James");
```

Table des Matières

Generics

- **Restrictions sur les paramètres de type**
  - Quid si on souhaite que la valeur du paramètre de type implémente une interface particulière ou hérite d'une classe particulière, de façon à pouvoir appeler une méthode de cette interface ?
  - Il est souvent nécessaire de mettre des restrictions sur les valeurs des paramètres de type. De telles restrictions peuvent être spécifiée avec la syntaxe suivante utilisée dans une déclaration de type ou de méthode.

```
< nomVarType extends nomType >
```

    - Il est possible de spécifier plusieurs noms de type dont la valeur du paramètre de type doit être une implémentation ou une interface. Dans ce cas, les noms de type sont séparés par '&'.
    - Il peut y avoir au maximum un nom de classe (qui doit apparaître en premier).

Generics

- **Restrictions sur les paramètres de type**
  - Exemple : quel est le problème de la classe ci-dessous ?

```
public class ArrayAlgorithms
{
   ...
   public static <T> T getMinimum(T[] array) {
     if ((array == null) || (array.length == 0))
       return null;
     T smallest= array[0];
     for (int i= 1; i < array.length; i++)
       if (smallest.compareTo(array[i]) > 0)
         smallest= array[i];
     return smallest;
   }
   ...
}
```

Il faut s'assurer que la valeur du paramètres de type `T` soit une classe qui implémente `Comparable` ou une interface qui hérite de `Comparable`

Generics

- **Restrictions sur les paramètres de type**
  - Exemple : solution = imposer une restriction sur le paramètre de type

```
public class ArrayAlgorithms
{
   ...
   public static <T extends Comparable> T getMinimum(T[] array) {
     if ((array == null) || (array.length == 0))
       return null;
     T smallest= array[0];
     for (int i= 1; i < array.length; i++)
       if (smallest.compareTo(array[i]) > 0)
         smallest= array[i];
     return smallest;
   }
   ...
}
```

La contrainte sur la variable de type `T` assure que l'interface `Comparable` doit être implémentée.

Table des Matières

Generics

- **Règles d'héritage**
  - Les règles d'héritage ont été mises à jour pour considérer les classes paramétrées (*generics*).
  - Ces nouvelles règles peuvent être surprenantes dans certaines situations.
  - Règle : soit deux classes paramétrées **A<S>** et **B<T>** , **S** et **T** étant des paramètres de type

> **A<S>** ***is-a*** **B<T>** ssi ((**A** ***is-a*** **B**) et (**S** = **T**))

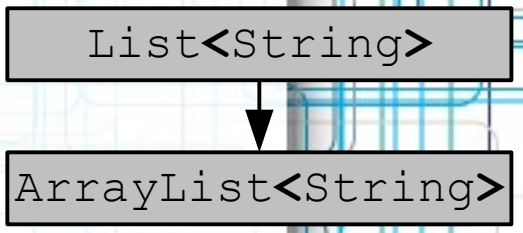
  - Quelles sont les conséquences de cette règle ?

Generics

- **Règles d'héritage**
  - Exemple

```
ArrayList<String> as= new ArrayList<String>();
List<String> ls= as; // OK
```

  - Dans ce cas, le compilateur pose la question
    - `ArrayList<String>` ***is-a*** `List<String>` ?



  - Appliquons la règle du *slide* précédent
    - **A**=`ArrayList`
    - **B**=`List`
    - **S**=`String`
    - **T**=`String`
  - La propriété ((**A** ***is-a*** **B**) et (**S** = **T**)) est vérifiée

Generics

- **Règles d'héritage**
  - Autre exemple

```
ArrayList<String> as= new ArrayList<String>();
ArrayList<Object> ao= as;
```

  - Dans ce cas-ci,
    - **A**=ArrayList
    - **B**=ArrayList
    - **S**=Object
    - **T**=String
  - La propriété ((**A** ***is-a*** **B**) et (**S** = **T**)) n'est donc **pas vérifiée** car (**S** ≠ **T**) !
  - Par conséquent, l'affectation demandée dans le programme ne peut pas être acceptée par le compilateur.

Generics

- **Règles d'héritage**
  - Le refus de l'affectation dans l'exemple qui précède peut sembler incompréhensible à priori. Il y a pourtant de **bonnes raisons** pour empêcher cette affectation.
    - Supposons, par contradiction que `ArrayList<String>` soit un sous-type de `ArrayList<Object>`.
    - Par conséquent, l'affectation suivante est autorisée.

```
ArrayList<String> as= new ArrayList<String>();
ArrayList<Object> ao= as;
```

    - Par conséquent, la variable `ao` est un alias pour `as` par lequel il est permis d'ajouter un élément non compatible (p.ex. une instance d'`Integer`) dans la collection `ArrayList<String>`.

```
ao.add(new Integer(5));
```

    - Cette situation permet donc de contourner la restriction imposée par `ArrayList<String>`, ce qui ne doit pas être autorisé !

Generics

- **Règles d'héritage**
  - Lorsque les *generics* ont été ajoutés à Java, toutes les situations pouvant amener à une violation du typage (comme dans l'exemple précédent) ont dû être empêchées
  - Seules sont autorisées les situations qui n'amènent pas à de telles violations. De telles situations sont appelée « *type-safe* ».
  - Afin d'assurer la compatibilité ascendante avec le code Java écrit avant l'introduction des *generics* (JDK < 5.0), il reste possible de convertir une référence vers un type paramétré **C<T>** en une référence vers le même type non-paramétré **C**.
  - Exemple

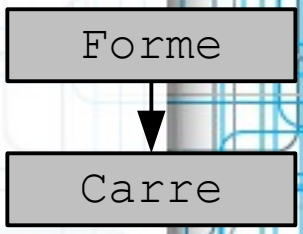
```
ArrayList<String> as= new ArrayList<String>();
ArrayList ao= as; // OK
```

Type non-paramétré (java n'effectue pas de vérifications)

Table des Matières

Generics

- **Type Joker (*wildcard type*)**
  - Les règles d'héritage qui viennent d'être décrites sont trop restrictives dans certaines situations.
  - Exemple
    - soit la classe `Carre` qui descend de la classe `Forme`.
    - soit la méthode `drawList` qui permet de dessiner à l'écran une liste d'instances de `Forme`.




```
public static void drawList(List<Forme> array, Graphics g) {
  for (int i= 0; i < array.size(); i++)
    array.get(i).draw(g);
}
```

```
ArrayList<Forme> arrayForme= ...;
ArrayList<Carre> arrayCarre= ...;
drawList(arrayForme, g); // OK
drawList(arrayCarre, g); // Pas permis
```

Il n'est pas « *safe* » de permettre la compatibilité entre `ArrayList<Forme>` et `ArrayList<Carre>` . C'est donc défendu par le compilateur (conformément à la règle vue précédemment).

Generics

- **Type Joker (*wildcard type*)**
  - Afin de lever les restrictions parfois trop fortes des règles d'héritage qui viennent d'être décrites les concepteurs des *generics* ont introduit le **type joke**r (*wildcard type*).
    - Le joker est utilisé dans une invocation de type générique avec la syntaxe suivante

```
< ? extends nomType >
```

    - La signification du joker est la suivante: le paramètre de type peut être quelconque tant qu'il hérite de la classe *nomType* ou qu'il implémente l'interface *nomType* (il est en relation ***is-a*** avec *nomType*).
    - Plusieurs types peuvent être mentionnés, séparés par '&'. Au maximum un type classe, qui doit apparaître en premier.

Generics

- **Type Joker (*wildcard type*)**
  - Exemple
    - Dans l'exemple précédent, l'utilisation du type joker apporte une solution au problème d'incompatibilité : le type attendu est maintenant une `ArrayList` dont le paramètre de type est une sous-classe de `Forme`. Cette définition est compatible avec `ArrayList<Carre>`.

```
public static void drawList(ArrayList<? extends Forme> array,
                            Graphics g) {
  for (int i= 0; i < array.size(); i++)
    array.get(i).draw(g);
}
```

```
ArrayList<Forme> arrayForme= ...;
ArrayList<Carre> arrayCarre= ...;
drawList(arrayForme, g); // OK
drawList(arrayCarre, g); // OK avec joker
```

L'utilisation du joker permet de passer en argument une instance d'`ArrayList` parametré avec un type qui descend de la classe `Forme`.

Generics

- **Type Joker (*wildcard type*)**
  - Il existe d'autres types de joker
    - **<?>** joker non borné → n'importe quel type
      i.e. équivalent à **<?** `extends Object`**>**
    - **<?** `super` *`nomType`***>** joker borne inférieure → une super classe de *nomType*.

      voir le lien suivant pour un exemple d'application
      https://docs.oracle.com/javase/tutorial/extra/generics/morefun.html

Table des Matières

Generics

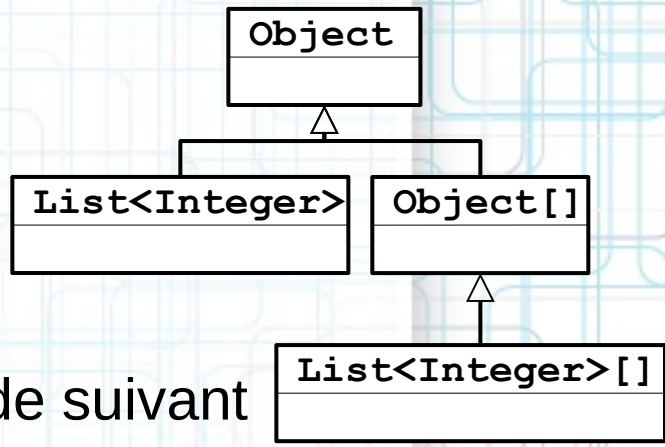
- **Tableaux génériques**
  - La création de tableaux génériques est interdite par le compilateur.

```
List<Integer> [] tab= new ArrayList<Integer>[100];
E [] tab2= new E[20];
```

Ces deux créations génèrent une erreur de compilation "generic array creation"

  - **Pourquoi ?**
  - Pour éviter des violations du système de type provenant de la combinaison
    - de la *type erasure*
    - de la *covariance* des tableaux

Generics



- **Tableaux génériques**
  - Supposons que le compilateur accepte le code suivant

```
List<Integer> [] tab= new ArrayList<Integer>[100];
```

  - Il est alors possible de tromper le système de type et de placer dans le tableau un objet d'un type incompatible

```
Object [] tabAlias= tab;
tabAlias[0]= "Niark Niark Niark";
```

Accepté car `List<Integer>[]` ***is-a*** `Object[]` (tableaux covariants)

Accepté car `String` ***is-a*** `Object` (référence polymorphique)

  - De plus, aucune exception de type `ArrayStoreException` ne se produira à l'exécution en raison de la *type erasure*. Celle-ci a pour effet que l'instance du tableau est de type `Object[]`.

Generics

- **Comment s'en sortir ? (1ère version)**
  - Une première approche consiste à utiliser un transtypage.

```
E[] tab= (E[]) new Object[100];
```

Attention !!! n'utiliser le transtypage de `Object[]` vers `E[]` que lors d'un `new` !!

  - Cette approche génère un avertissement (*warning*) lors de la compilation. Il est possible de masquer celui-ci en annotant la méthode qui contient le transtypage avec `@SuppressWarnings("unchecked")`.
  - Exemple

```
@SuppressWarnings("unchecked")
public E[] createGenericArray() {
  return (E[]) new Object[100];
}
```

Attention !!! l'utilisation de `@SuppressWarnings("unchecked")` supprime tous les avertissements de ce type dans la méthode. A utiliser avec parcimonie !

Generics

- **Comment s'en sortir ? (1ère version)**
  - L'approche précédente fonctionne avec un tableau générique mais pas avec un tableau de type paramétré

```
List<Integer>[] tab= (List<Integer>[]) new Object[100];
```

  - Le code est accepté par le compilateur, mais une exception `ClassCastException` est générée à l'exécution !
  - Cette exception se produit car java vérifie que les éléments du tableau sont de type `List` (non paramétré), ce qui n'est pas le cas.
  - La solution est donc d'utiliser

```
List<Integer>[] tab= (List<Integer>[]) new List[100];
```

Generics

- **Comment s'en sortir ? (2ème version)**
  - Une seconde approche se base sur l'API *reflection*. Celle-ci permet d'inspecter les classes et instances en cours d'exécution. Elle permet aussi de créer des objets dont le type est spécifié lors de l'exécution (contrairement à **new**).
  - Exemple

```
@SuppressWarnings("unchecked")
public E[] createGenericArray(E elt) {
  return (E[]) java.lang.reflect.Array.newInstance(
                                    elt.getClass(), 100);
}
```

Ici, l'instance **elt** n'est passée que pour fournir le type des élements du tableau.

Generics

- **Cas particulier : classes internes**
  - Les classes internes amènent des surprises supplémentaires…

```
public class A<E>
{
  public class B
  {
  }

  B [] tab= new B[32];

}
```

Erreur : "generic array creation"

  - Ici, il ne faut pas oublier que la classe interne `B` est en fait `A<E>.B` ⇒ il s'agit donc d'un type générique !