

NFP121

Programmation avancée

La Généricité en Java

La Généricité

- Permet de **paramétrer du code avec des types de données**
- Avant le JDK 1.5 / depuis le JDK 1.5 (généricité)
- La généricité en Java, syntaxe, contraintes, conséquences
- La généricité et la JVM
- Un Quizz

Sans la généricité ⁽¹⁾

■ Avant Java 5 la généricité en Java n'existe pas :

■ Les collections contiennent habituellement des éléments d'un **type spécifique**.

■ Pourtant toutes les collections stockent des éléments de type **Object**

```
// Liste d'objets
List l = new ArrayList();

// Alimentation de la liste
l.add("Une String");
l.add("Une seconde String");

// Le cast est obligatoire pour la compilation
String valeur = (String) l.get(0);
```

Remarques ?

Sans la généricité ⁽²⁾

- **Avant Java 5 dans les collections tous les éléments sont de type Object :**
 - Possibilité de mélanger des éléments de types différents
 - Nécessité de « caster » les objets en sortie des collections (lisibilité --)
 - Risque de `ClassCastException` à l'exécution même si compilation sans erreurs

```
// Liste d'objets
List l = new ArrayList();

// Mélanges de types différents
l.add("Une String");
l.add(42);

// Le cast est obligatoire pour la compilation
// Compilation sans problème mais erreur à l'exécution ...
Integer valeur = (Integer) l.get(0);
```

Problèmes ?

Sans la généricité ⁽³⁾

- **Recherche d'une Solution pour :**
 - Eviter les « cast »
 - Vérifier les types à la compilation
- **Solution : une implémentation par type d'élément à stocker**
 - Une classe pour stocker les listes de String
 - Une classe pour stocker les listes de Integer
 - Une classe pour stocker les listes de
- ...

Sans la généricité ⁽⁴⁾

■ Une implémentation par type d'élément à stocker

```
public class ListeDeString {  
    /** Conteneur sous-jacent */  
    protected ArrayList listeInterne = new ArrayList();  
    /** Ajout d'un élément */  
    public boolean add(String str) {  
        return listeInterne.add(str);  
    }  
    /** Retourne l'élément à la position index */  
    public String get(int index) {  
        return (String) listeInterne.get(index);  
    }  
    //... autres méthodes  
}
```

```
public class ListeDeInteger {  
    /** Conteneur sous-jacent */  
    protected ArrayList listeInterne = new ArrayList();  
    /** Ajout d'un élément */  
    public boolean add(Integer integ) {  
        return listeInterne.add(integ);  
    }  
    /** Retourne l'élément à la position index */  
    public Integer get(int index) {  
        return (Integer) listeInterne.get(index);  
    }  
    //... autres méthodes  
}
```

Solution satisfaisante ? Vos critiques ...

La Généricité

- **Obtenue par le paramétrage** d'une **Classe** ou d'une **Interface** avec **un ou plusieurs Types**
- Les types deviennent un « **Paramètre** » de la classe
- Ces « **Paramètre** » sont « **définis** » dans les instructions d'instanciation des classes paramétrée.
- **Le compilateur vérifie** (analyse statique) pour toute opération sur la classe que les **types utilisés** sont **conformes** (*que signifie conforme ?*)
- Utilisation d'une stratégie **d'inférence de type** au moment de la compilation

L'**inférence de types** est un mécanisme qui permet à un compilateur ou un interpréteur de rechercher automatiquement les **types** associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source.

https://fr.wikipedia.org/wiki/Inf%C3%A9rence_de_types

La Généricité en Java 5: un exemple

- **Syntaxe:** Les types paramétrés sont cités entre chevrons après le nom de la classe « **<type>** »

```
/**
 * Collection à 1 seul élément
 * @param <E> type de l'élément
 */
public class CollectionSingleton<E> {
    private E element;
    /** affectation de l'élément
     * @param element
     */
    public void setElement(E element) {
        this.element = element;
    }
    /** lecture de l'élément*/
    public E getElement() {
        return element;
    }
}
```

```
// Collection singleton de String
CollectionSingleton<String> colSinglString = new CollectionSingleton<>();
colSinglString.setElement("Str element");
String valeurStr = colSinglString.getElement();
```

```
// Collection singleton de Integer
CollectionSingleton<Integer> colSinglInt = new CollectionSingleton<>();
colSinglInt.setElement(42);
Integer valeurInt = colSinglInt.getElement();
```

- Les « **paramètres de type** » ne sont pas conservés dans le bytecode produit: **Tout est objet**

La Généricité dans les collections en Java 5 ⁽¹⁾

■ Les Collections du JDK 5 sont génériques :

```
// Interface List du JDK 5
public interface List<E> extends Collection<E> {
    boolean add(E e);
    E get(int index);
    E remove(int index);
    //...
}
```

// Vocabulaire :

`List<E>` est une interface générique

`ArrayList<E>` est une classe générique

`ArrayList<Integer>` est une instantiation du type générique `ArrayList<E>`

`ArrayList` un type générique sans type paramétré est dit type « Raw » (utilisé dans les messages du compilateur)

```
// Classe ArrayList
public class ArrayList<E> extends
AbstractList<E> implements List<E>,
RandomAccess, Cloneable, java.io.Serializable {
    @Override
    public E get(int index) {
        //...
    }
    @Override
    public int size() {
        //...
    }
    //...
}
```

La Généricité dans les collections en Java 5 (2)

■ Utilisation des Collections génériques :

```
// Liste de Nombres
List<Number> l = new ArrayList<Number>();
// Impossible de Mélanger: que des Number
l.add(42.3);
l.add(55);
// Ne compile pas
String valeurStr = l.get(0);
// Je ne peux récupérer qu'un Number
Number maValeur = l.get(0);
Integer valeurInt = l.get(0); // Correct ?
Integer valeurInt2 = (Integer)l.get(0); // Correct ?
```

La Collection n'accepte et ne fournit que des éléments du type paramétré

La Généricité dans les collections en Java 5 (3)

■ Utilisation des Collections génériques :

```
// Liste de Nombres
List<Number> l= new ArrayList<Number>();
// Impossible de Mélanger: que des Number
l.add(42.3);
l.add(55);
// Ne compile pas
String valeurStr = l.get(0);
// Je ne peux récupérer qu'un Number
Number maValeur = l.get(0);
Integer valeurInt = l.get(0); // NON AUTORISE
Integer valeurInt2 = (Integer)l.get(0); // Compile mais produira une erreur à l'exécution
```

La Collection n'accepte et ne fournit que des éléments du type paramétré

Apports des collections génériques en Java 5

- Les Collections ne comportent que des éléments du même type
- Réutilisation de la même implémentation de collection quel que soit le type des éléments à stocker
- Suppression des « Cast » qui nuisent à la lisibilité du code
- Le compilateur vérifie pour toutes les opérations sur la collection que le type utilisé est conforme au type paramétré

PROBLEMES Collections < JDK 1.5 résolus. Mais pas sans conséquences...

Utilisation Autorisée des types paramétrés (1)

- En tant que type pour déclarer des propriétés ou des variables

```
public T element;
```

- En tant que type du paramètre d'une méthode

```
public void retirer(T element);
```

- En tant que type paramétré pour obtenir une instance d'un type générique

```
List<T> sousListe = new ArrayList<T>;
```

Utilisation Autorisée des types paramétrés (2)

- En tant que type de retour pour une méthode

```
public T get(int index)
```

- En tant que type destination dans un Cast

```
return (T) element;
```

- En tant que paramètre de type d'une classe de base générique (héritage)

```
public class UneSpecialisationDeListe<T> extends ArrayList<T> {  
    }  
}
```

Utilisation Interdite des types paramétrés

- Impossible de créer des instances du type paramétré (solutions ?)

```
T element = new T();
```

- Le type paramétré ne peut pas servir de classe de base pour une classe interne

```
public static class MaClasseInterne extends T {  
    //...  
}
```

- Le type ne peut pas être utilisé dans une instruction **instanceof**

```
if (element instanceof T) {  
    //...  
}
```

Exemple : la « paire » générique (2)

```
/**
 * Paire de 2 éléments
 * @param <T1> type de l'élément 1
 * @param <T2> type de l'élément 2
 */
public static class Paire<T1, T2> {

    protected T1 element1;
    protected T2 element2;

    public Paire(T1 element1, T2 element2) {
        this.element1 = element1;
        this.element2 = element2;
    }

    public T1 getElement1() {return element1;}
    public void setElement1(T1 element1) {this.element1 = element1;}
    public T2 getElement2() {return element2;}
    public void setElement2(T2 element2) {this.element2 = element2;}
}
```

```
// Création d'une paire avec en valeur un Texte et un Entier
Paire<String, Integer> paire1 = new Paire<String, Integer>("Texte", 22);
// Création d'une paire avec en valeur une paire et un Entier
Paire<Paire<String, Boolean>, Integer> paire2 = new Paire<Paire<String, Boolean>, Integer>( new Paire<String, Boolean>("Clé", true), 22);
// Création équivalente (utilisable depuis Java 7)
Paire<Paire<String, Boolean>, Integer> paire2Bis = new Paire<> ( new Paire<>("Clé", true), 22);
```

Utilisation de la « paire » générique (2)

```
// Création d'une paire avec en valeur un Texte et un Entier
Paire<String, Integer> paire1 = new Paire<String, Integer>("Texte", 22);

// Création d'une paire avec en valeur une paire et un Entier
Paire<Paire<String, Boolean>, Integer> paire2 = new Paire<Paire<String, Boolean>, Integer>( new
Paire<String, Boolean>("Clé", true), 22);

// Création équivalente (utilisable depuis Java 7)
Paire<Paire<String, Boolean>, Integer> paire2Bis = new Paire<>( new Paire<>("Clé", true), 22);

// Création d'une paire avec en valeur une String et un Number
Paire<String, Number> paire3 = new Paire<>("prixEntier", 22);
Paire<String, Number > paire4 = new Paire<>("prixDouble", 17.3d);

Integer valInt = paire3.getElement2();
Integer valInt = (Integer)paire3.getElement2();
Number valpaire3 = paire3.getElement2();
Object valpaire3 = paire3.getElement2();

Double valDb1 = paire4.getElement2();
```

Compatibilité entre types ⁽¹⁾

- Quelles affectations sont possibles entre types ?

// Types simples

```
Object obj = new ...;  
Integer integ = new ...;  
obj = integ; // Possible ?
```

// Tableaux

```
Object[] tObj = new ...;  
Integer[] tInteg = new ...;  
tObj = tInteg; // Possible ?
```

// Types génériques

```
ArrayList<Object> lObj = new ...;  
ArrayList<Integer> lInteg = new ...;  
lObj = lInteg; // Possible ?
```

Compatibilité entre types (2)

// Types simples

```
Object obj = new ...;  
Integer integ = new ...;  
obj = integ; // CORRECT car Integer est un Object
```

// Tableaux

```
Object[] tObj = new ...;  
Integer[] tInteg = new ...;  
tObj = tInteg; // CORRECT car Integer est un Object (Principe de covariance)
```

// Types génériques

```
ArrayList<Object> lObj = new ...;  
ArrayList<Integer> lInteg = new ...;  
lObj = lInteg; // INCORRECT car Integer est un Object mais liste d'Object != liste d'Integer
```

Compatibilité entre types génériques ⁽¹⁾

```
ArrayList<Object> lObj = new ...;
// Je peux évidemment ajouter un Object à une liste d'Object
lObj.add(new Object());
// Je peux ajouter un Integer à une liste d'Object
lObj.add(1);

ArrayList<Integer> lInteg = new ...;
// Je peux évidemment ajouter un Integer à une liste d'Integer
lInteg.add(2);
// Je ne peux pas ajouter un Object à une liste d'Integer
lInteg.add(new Object());

// ERREUR DE COMPILATION car même si Integer est un Object, liste d'Object != liste d'Integer
lObj = lInteg;

// SI C'ÉTAIT POSSIBLE on pourrait faire:
lObj.clear();
lObj.add(new Object()); // On pourrait donc ajouter un Object à la liste d'Integer affectée
Integer valInt = lInteg.get(0); // Et on obtiendrait un Object alors que l'on attend un Integer
```

Compatibilité entre types génériques (2)

- Donc un type générique n'est compatible qu'avec un type affectable de paramètre compatible

```
ArrayList<Integer> lInteg = new ArrayList<Integer>(); // Correct
Collection<Integer> collInteg = new ArrayList<Integer>(); // Correct
ArrayList<Number> lNumber = new ArrayList<Integer>(); // Incorrect
```

- Donc une méthode dont le paramètre est d'un type générique n'accepte que ce type générique

```
/**
 * Affiche les éléments de la liste de nombres
 * @param listeDeNombres
 */
public void afficherElements(ArrayList<Number> listeDeNombres) {
    // Itération sur les éléments
    for (Number nombre : listeDeNombres) {
        System.out.println(nombre);
    }
}
```

Et si je veux afficher une liste d'entier (Integer) ?

Compatibilité entre types génériques (3)

```
/**
 * Affiche les éléments de la liste d'entier
 * @param listeEntiers
 */
public void afficherElements(ArrayList<Integer> listeEntiers) {
    // Itération sur les éléments
    for(Integer entier : listeEntiers ) {
        System.out.println(entier);
    }
}

/**
 * Affiche les éléments de la liste de chaînes
 * @param listeDeChaines
 */
public void afficherElements(ArrayList<String> listeDeChaines) {
    // Itération sur les éléments
    for(String chaine : listeDeChaines ) {
        System.out.println(chaine);
    }
}
```

Chaque méthode d'affichage ne sait afficher qu'un et un seul type de liste !!!

Le Joker <?> (1)

- **Interdit la réutilisation de méthode:** 1 méthode par type d'élément de liste

```
public void afficherElements (ArrayList<Integer> listeEntiers)
public void afficherElements (ArrayList<String> listeDeChaines)
```

- **Typage strict des variables et des propriétés :** elles ne peuvent être initialisées que par une instance compatible avec leur type et de même type paramétré.

- Pour pallier ces limitations la notion de Joker ? a été introduite.
 '?' Signifie « n'importe quel type » d'où ArrayList<?> est une ArrayList de
 ...

Le Joker <?> (2)

- Réécriture de la méthode `afficherElements` pour la rendre utilisable pour toutes les listes quel que soit le type d'élément:

```
/** Affiche les éléments de la liste @param liste */
public void afficherElements(ArrayList<?> liste) {
    // Itération sur les éléments
    for(Object element : liste) {
        System.out.println(element);
    }
}
```

REMARQUE: On utilise `Object` pour typer les éléments mais ‘?’ ne signifie pas `Object`

```
public void traiterElement(ArrayList<?> liste, Object element) {
    // Quel est le type dynamique de elem0, quel est le type statique de elem0
    ... elem0 = liste.get(0);
    // Erreur de compilation: le compilateur ne peut supposer que la liste comporte des Object
    // ou que Object est un sous-type des éléments de la liste
    liste.add(element);
}
```

Le Joker <?> (3)

■ Résumé

```
// Liste d'entier
List<Integer> lInt = new ArrayList<Integer>();
lInt.add(42);
// Liste de ???
List<?> ljok = lInt;
// Lecture : Le type est inconnu il ne peut être que Object ou un sous-type de Object
Object valeur = ljok.get(0);
// Ecriture : Il est impossible de déterminer le type des éléments → AJOUT D'ELEMENTS IMPOSSIBLE
ljok.add(43); // Incorrect
ljok.add("Une String"); // Incorrect
ljok.add(new Object()); // Incorrect

// Seul ajout possible:
ljok.add(null);
```

Une collection générique avec ‘?’ « est en lecture seule »

Retour sur la compatibilité entre Tableaux

```
Object[] tObj = new Object[10];  
// Je peux évidemment affecter un Object à un tableau d'Object  
tObj[0] = new Object();  
// Je peux affecter un Integer à un tableau d'Object  
tObj[1] = 1;  
  
Integer[] tInteg = new Integer[10];  
// Je peux évidemment affecter un Integer à un tableau d'Integer  
tInteg[0] = 2;  
// Je ne peux pas affecter un Object à un tableau d'Integer  
tInteg[1] = new Object();  
  
// Passe la phase de compilation (Principe de Covariance)  
tObj = tInteg;  
// Mais à l'exécution:  
tObj[2] = new Object(); // Affectation d'un Object au tableau d'Integer affecté
```

L'instruction d'affectation lève une `ArrayStoreException` !!!

Covariance et Contravariance ⁽¹⁾

■ Covariance (depuis JDK 5) :

La methode `getSomething` de la classe `Sub` est en **Covariance** car elle retourne un type sous-classe du type retourné par la méthode `getSomething` de la classe `Super`. Le contrat est respecté puisque `String` est un `Object`.

```
class Super {  
    Object getSomething() {  
        return new Object();  
    }  
}  
  
class Sub extends Super {  
    String getSomething() {  
        return "Texte";  
    }  
}
```

■ Contravariance :

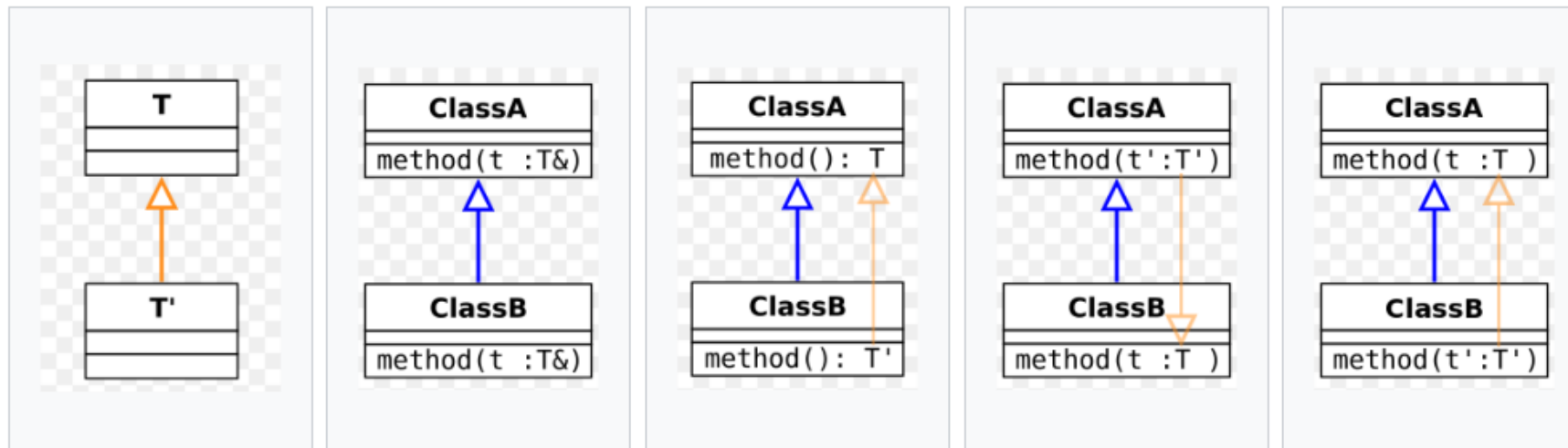
La methode `doSomething` de la classe `Sub` utilise un paramètre d'un super type du type du paramètre de la méthode `getSomething` de la classe `Super`
(Non autorisé en Java)

```
class Super {  
    void doSomething(String param) {  
        //...  
    }  
}  
  
class Sub extends Super {  
    void doSomething(Object param) {  
        //...  
    }  
}
```

<https://stackoverflow.com/questions/2501023/demonstrate-covariance-and-contravariance-in-java>

Covariance et Contravariance (2)

Variance and method overriding: overview



Subtyping of the argument/return type of the method.

Invariance. The signature of the overriding method is unchanged.

Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.

Contravariant argument type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.

Covariant argument type. Not type safe.

[https://en.wikipedia.org/wiki/Covariance_and_contravariance_\(computer_science\)](https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science))

Contraintes basées sur l'arbre d'héritage

- ? (Wildcard) signifie « n'importe quel type ».
- Définition de bornes min ou max basées sur l'arbre d'héritage (Bounded Wildcards)

`<? extends T> // Tout type étant un sous-type de T (basé sur paramètre formel T)`

`<? extends Number> // Tout type étant un sous-type de Number`

`<? super T> // Tout type étant un super-type de T (basé sur paramètre formel)`

`<? super Integer> // Tout type étant un super-type de Integer (Number ou Object)`

Contraintes « extends » ⁽¹⁾

<? extends T> (Upper Bounded Wildcards)

- T est la borne supérieure des types autorisés.
- Autrement dit : seul le type **T** ou les **sous-types de T** sont autorisés.

Exemples:

```
// Type statique: Liste de Number, Type dynamique: Liste de Integer
ArrayList<? extends Number> lNumber = lInteg;
... valeur = lNumber.get(0); // Quel(s) type(s) pour la variable « valeur »
lNumber.add(new Double(2.2)); // Correct ?
lNumber.add(new Integer(3)); // Correct ?
```

```
// Type statique: Liste de Serializable, Type dynamique: Liste de String
List<? extends Serializable> l = lString;
... valeur = l.get(0); // Quel(s) type(s) pour la variable « valeur »
```

Contraintes « extends » (2)

Réponses:

```
// Type statique: Liste de Number, Type dynamique: Liste de Integer  
ArrayList<? extends Number> lNumber = lInteg;
```

```
// L'élément lu peut être de type Object évidemment mais aussi Number  
Number valeur = lNumber.get(0);
```

```
// Incorrects: Impossible de déduire le type des éléments de la liste  
lNumber.add(new Double(2.2)); // Incorrect  
lNumber.add(new Integer(3));  // Incorrect
```

```
// Type statique: Liste de Serializable, Type dynamique: Liste de String  
List<? extends Serializable> l = lString;  
// L'élément lu peut être de type Object évidemment mais aussi Serializable  
Serializable valeur = l.get(0);
```

Contraintes « super » ⁽¹⁾

<? super T> (Lower Bounded Wildcards)

- T est la borne inférieure des types autorisés.
- Autrement dit : seuls le type **T** ou les **super-types de T** sont autorisés.

Exemples:

```
// Type statique: Liste de Integer ou de super-types de Integer
ArrayList<? super Integer> lNumber = lInteg;
... valeur = lNumber.get(0); // Quel(s) type(s) pour la variable « valeur »
lNumber.add(new Double(2.2)); // Correct ?
lNumber.add(new Integer(3));  // Correct ?
```

Contraintes « super » ⁽²⁾

Réponses:

```
// Type statique: Liste de Integer ou de super-types de Integer  
ArrayList<? super Integer> lNumber = lInteg;
```

```
// Le type des éléments est un super-type de Integer. MAIS LEQUEL ?  
Object valeur = lNumber.get(0); // seul Object est acceptable
```

```
// Incorrect on est certain que le type Double n'est pas un super-type de Integer  
lNumber.add(new Double(2.2));
```

```
// Correct puisque Integer est obligatoirement un sous-type du type des éléments de la liste  
lNumber.add(new Integer(3));
```

Contraintes composées

`<? extends Borne1 & Borne2 & ... & BorneN >`

- La contrainte impose que le **type** soit un **sous-type de toutes les bornes**
- Les bornes sont des classes ou des interfaces
- Si une classe fait partie des bornes elles doit être placée en premier (sinon erreur compil)

```
// Liste de Object et de sous-type de Object étant Serializable et Comparable  
ArrayList<? extends Object & Serializable & Comparable<?> > lNumber = lInteg;
```

La Généricité appliquée aux méthodes (1)

- Les méthodes d'une classe générique peuvent utiliser les types paramétrés de la classe

```
class TraiteurDeListe<T> {  
    // T est le paramètre générique de la classe utilisé par la méthode « traiter »  
    public T traiter(List<T> liste) {  
        for (T element : liste) {  
            // traitement de l'élément  
            //...  
        }  
        return null;  
    }  
}
```

- Une méthode peut être générique et avoir ses propres types paramétrés

```
// T est le paramètre générique propre à la méthode « traiter »  
public <T> void traiter(List<T> liste) {  
    for(T element : liste) {  
        // traitement de l'élément  
        // ...  
    }  
}
```

La Généricité appliquée aux méthodes (2)

```
// T et E sont les paramètres génériques de la méthode
public <T, E> void faireLeTruc (List<T> liste, E valeur) {
    //...
}

// TYPE_RETOUT est le paramètre générique de la méthode
public <TYPE_RETOUT> TYPE_RETOUT getProperty (String propertyName) {
    return (TYPE_RETOUT) mapProperties.get(propertyName);
}

// T est le paramètre générique de la méthode
public <T> void faireAutreTruc (T a, T b) {
    //...
}

// TYPE_MYSTERE est le paramètre générique de la méthode
public <TYPE_MYSTERE> void methodeGenerique() {
    TYPE_MYSTERE a = (TYPE_MYSTERE) "ABCD";
}
```

Comment le compilateur déduit (infère) les types paramétrés de ces méthodes génériques ?

La Généricité appliquée aux méthodes (3)

■ L'appel d'une méthode générique

```
// T et E sont les paramètres génériques de la méthode
public <T, E> void faireLeTruc (List<T> liste, E valeur) {
    //...
}

// Appel de faireLeTruc, le compilateur déduit que T est le type String et E est le type Double
// Le compilateur contrôle la cohérence de l'invocation de la méthode avec ces valeurs de type.
faireLeTruc(new ArrayList<String>(), 2.3d);

// TYPE_RETOUT est le paramètre générique de la méthode
public <TYPE_RETOUT> TYPE_RETOUT getProperty (String propertyName) {
    return (TYPE_RETOUT) mapProperties.get(propertyName);
}

// Appel de getProperty, le compilateur déduit que TYPE_RETOUT est Integer
// car la variable timeout qui sera affectée est un Integer
Integer timeout = getProperty("PARM_TIMEOUT")
```

La Généricité appliquée aux méthodes (4)

- Le compilateur est capable de découvrir les types dans la majorité des cas

```
// T est le paramètre générique de la méthode
public <T> void faireAutreTruc (T a, T b) {
    //...
}
// Appel de la méthode faireAutreTruc avec paramètres Integer et Double: Le compilateur infère le
type Number
faireAutreTruc(new Integer(42), new Double(2.4));
```

- Si le compilateur ne peut inférer les types paramétrés on peut les lui fournir explicitement

```
// Appel d'une méthode générique statique en lui fournissant le type
Classe.<Integer>methodeGeneriqueStatique()

// Appel d'une méthode générique en lui fournissant le type
this.<Integer>methodeGenerique();
instance.<Integer>methodeGenerique();
```

La Généricité appliquée aux méthodes (4)

- Si le compilateur ne peut inférer les types paramétrés on peut les lui fournir explicitement

```
// TYPE_MYSTERE est le paramètre générique de la méthode
// Le compilateur ne déduit pas que TYPE_MYSTERE est String
public static <TYPE_MYSTERE> void methodeGeneriqueStatique() {
    TYPE_MYSTERE a = (TYPE_MYSTERE) "ABC";
    String str = a; // ne passe pas à la compilation, nécessité d'un cast en String
    Integer i = (Integer) a;
    System.out.println(a);
}

// Appel en fournissant le type Boolean à TYPE_MYSTERE : le compilateur l'ignore
DemoClasseGenerique.<Boolean>methodeGeneriqueStatique();
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
at edu.aisl.genericite.DemoClasseGenerique.methodeGeneriqueStatique (DemoClasseGenerique.java:85)
at edu.aisl.genericite.DemoClasseGenerique.main (DemoClasseGenerique.java:92)
```

Le principe de « Type Erasure » ⁽¹⁾

- En C++ la généricité est obtenue par les **Templates** :
 - Des sources sont générées pour chaque version de type générique paramétrées
 - Ces sources sont compilées pour fournir des codes exécutables (code compilé)
- En Java les **types génériques compilés** ne donnent qu'**une seule Classe** (raw):
 - **ArrayList<String>** et **ArrayList<Integer>** n'ont pas été créées, il n'existera que **ArrayList**

Pourquoi ?

Le code écrit avant la généricité doit pouvoir être exécuté avec les versions de JVM qui proposent la généricité (contrainte issue de la spécification Java)

Comment ?

le « Type Erasure » (Effacement de type)

Le principe de « Type Erasure » ⁽²⁾

- Solution utilisée pour respecter la contrainte : le « **Type Erasure** » (Effacement de type)
- Le compilateur Java se charge des tâches pour faire fonctionner la généricité Java:
 - **Vérification du typage** (inférence de type)
 - **Remplacement des types génériques par leur version « Raw »** (type erasure)
 - **Remplacement des paramètres de type par des types « Simples »** (Object ou ...)
 - **Ajout des Cast nécessaires pour convertir les types en « Simples » en types paramétrés**

Remplacement :

`ArrayList<T>` : est remplacé par le type brut `ArrayList`

`T` : est remplacé par `Object`

`ArrayList<T extends Serializable>` : est remplacé par `ArrayList`

`T` : est remplacé par `Serializable`

Le principe de « Type Erasure » ⁽³⁾

■ Exemples de remplacements :

Réécriture de méthodes:

```
// Méthode « affecter » d'une classe paramétrée par T
public void affecter(T elem) { ... } // Version avec généricité
public void affecter(Object elem) { ... } // Version après remplacement
// Méthode « obtenir » d'une classe paramétrée par T
public T obtenir() { ... } // Version avec généricité
public Object obtenir() { ... } // Version après remplacement
```

Réécriture de codes utilisant des types génériques:

```
// Source origine avec généricité
ArrayList<String> listeString = new ArrayList<String>();
String element = listeString.get(0);

// Version après remplacements par le compilateur
ArrayList listeString = new ArrayList();
String element = (String)listeString.get(0);
```

Conséquences du type erasure ⁽¹⁾

- Impossible d'avoir 2 méthodes dont la **différence de prototypes** ne **dépend** que de **paramètres de types génériques**

```
// Version avec généricité
```

```
public void faireLeTruc(List<String> listeString) { ... }  
public void faireLeTruc(List<Integer> listeInteg) { ... }
```

```
// Version obtenue après remplacement par le compilateur (remplacement par type « raw »)
```

```
public void faireLeTruc(List listeString) { ... }  
public void faireLeTruc(List listeInteg) { ... }
```

Quel est le problème ?

Conséquences du type erasure ⁽²⁾

`List<String> listeString` est devenu `List listeString` Donc:

- `InstanceOf` pour un type générique avec un paramètre n'a aucun sens :

`instanceof List<String>`

- Cast vers un type générique avec un paramètre n'a aucun sens :

`List<String> listeString = (List<String>) l;`

- Les type génériques sont comparables seulement en fonction de leur type « raw »

`new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass();`

Interopérabilité avec du code non générique ⁽¹⁾

- Il est possible d'appeler des méthodes avec des types « raw » comme argument en passant des types paramétrés

```
// Méthode avec argument non générique
public void traiterListe(List liste) {
    //...
}
```

```
// Appel de la méthode non générique avec un argument de type générique
List<String> listeString = new ArrayList<String>();
traiterListe(listeString);
```

PAS DE PROBLEME ?

Interopérabilité avec du code non générique (2)

- Autorisé mais on perd l'apport de la généricité: la sûreté du typage

```
// Méthode avec argument non générique
public void traiterListe(List liste) {
    liste.add(22);
}
```

```
// Appel de la méthode non générique avec un argument de type générique
List<String> listeString = new ArrayList<String>();
traiterListe(listeString);
System.out.println(listeString); // affiche [22]
```

Conclusion sur la généricité

- Permet d'éviter des erreurs de type à l'exécution (contrôle à la compilation)
- Gain en lisibilité et code moins verbeux (suppression des Cast)
- Le paramétrage par joker est indispensable mais n'est pas sans conséquences
- Les contraintes se basent sur la hiérarchie d'héritage
- Notation parfois pas facile à interpréter à la lecture

Quizz sur la Généricité

<http://www.grayman.de/quiz/java-generics-en.quiz>

(copié sur cours CNAM Paris de JM Douin)

Quiz: Java 1.5 Generics: Question No. 1/14

With generics the compiler has more information about the types of the objects, so explicit casts don't have to be used and the compiler can produce type safe code.

What implications have the generics for the runtime performance of the program which uses them?

Choose the correct answer:

- a) ☐ With the generics the compiler can optimize the code for used types. This and the omission of the casts are the reasons why the code compiled with the generics is **quicker** than the one compiled without.
- b) ☐ The usage of generics has **no implications** for the runtime performance of the compiled programs.
- c) ☐ The improved flexibility and type safety means that the compiler has to generate concrete implementation from the generic template for each used type. This means that applications start **a bit slower**.

Answer

Quiz: Java 1.5 Generics: Question No. 2/14

As an example for a generic class we will use a very simple container. A **Basket** can contain only one element.

Here the source code:

```
public class Basket<E> {  
    private E element;  
  
    public void setElement(E x) {  
        element = x;  
    }  
  
    public E getElement() {  
        return element;  
    }  
}
```

We will store fruits in the baskets:

```
class Fruit {  
}  
  
class Apple extends Fruit {  
}  
  
class Orange extends Fruit {  
}
```

What would Java 1.5 do with the following source code?

```
Basket<Fruit> basket = new Basket<Fruit>(); // 1  
basket.setElement(new Apple()); // 2  
Apple apple = basket.getElement(); // 3
```

Choose the correct answer:

- a) ☐ The source code is OK. Neither the compiler will complain, nor an exception during the runtime will be thrown.
- b) ☐ Compile error in the line 2.
- c) ☐ Compile error in the line 3.

Quiz: Java 1.5 Generics: Question No. 3/14

Let's stay with our baskets. What do you think about the following source code?

```
Basket<Fruit> basket = new Basket<Fruit>();  
basket.setElement(new Apple());  
Orange orange = (Orange) basket.getElement();
```

Choose the correct answer:

- a) ☐ The source code is OK. Neither the compiler will complain, nor an exception during the runtime will be thrown.
- b) ☐ Compile error in the line 2.
- c) ☐ Compile error in the line 3.
- d) ☐ A **ClassCastException** will be thrown in the line 3.

Answer

Quiz: Java 1.5 Generics: Question No. 4/14

Which ones of the following lines can be compiled without an error?

Check all correct options:

- a) ☐ `Basket b = new Basket();`
- b) ☐ `Basket b1 = new Basket<Fruit>();`
- c) ☐ `Basket<Fruit> b2 = new Basket<Fruit>();`
- d) ☐ `Basket<Apple> b3 = new Basket<Fruit>();`
- e) ☐ `Basket<Fruit> b4 = new Basket<Apple>();`
- f) ☐ `Basket<?> b5 = new Basket<Apple>();`
- g) ☐ `Basket<Apple> b6 = new Basket<?>();`

Answer

Quiz: Java 1.5 Generics: Question No. 5/14

Let's have a look at the typeless baskets and the ones where the type is an unbounded wildcards.

```
// Source A
Basket<?> b5 = new Basket<Apple>();
b5.setElement(new Apple());
Apple apple = (Apple) b5.getElement();
```

```
// Source B
Basket b = new Basket();
b.setElement(new Apple());
Apple apple = (Apple) b.getElement();
```

```
// Source C
Basket b1 = new Basket<Orange>();
b1.setElement(new Apple());
Apple apple = (Apple) b1.getElement();
```

Which of the following statements are true?

Check all correct options:

- a) ☐ Source A cannot be compiled
- b) ☐ Source B will be compiled with warning(s). No exception will be thrown during the runtime.
- c) ☐ Source C will be compiled with warning(s). A **ClassCastException** exception will be thrown during the runtime.

Answer

Quiz: Java 1.5 Generics: Question No. 6/14

And what about this one?

```
Basket b = new Basket(); // 1
Basket<Apple> bA = b; // 2
Basket<Orange> bO = b; // 3
bA.setElement(new Apple()); // 4
Orange orange = bO.getElement(); // 5
```

Choose the correct answer:

- a) ☐ The lines 2 and 3 will cause a compile error.
- b) ☐ The line 4 will cause a compile error.
- c) ☐ The line 5 will cause a compile error because a cast is missing
- d) ☐ The source code will be compiled with warning(s). During the runtime a **ClassCastException** will be thrown in the line 5.
- e) ☐ The source code will be compiled with warning(s). No exception will be thrown during the runtime.

Answer

Quiz: Java 1.5 Generics: Question No. 7/14

In our rich class hierarchy the class **Apple** has following subclasses:

```
class GoldenDelicious extends Apple {}  
class Jonagold extends Apple {}
```

Our fruit processing application contains an utility class which can decide, whether an apple is ripe:

```
class FruitHelper {  
    public static boolean isRipe(Apple apple) {  
        ...  
    }  
}
```

In the class **FruitHelper** we want to implement a method which can look into any basket which can contain apples only and decide, whether the apple in the basket is ripe or not. Here the body of the method:

```
{  
    Apple apple = basket.getElement(); // 1  
    return isRipe(apple); // 2  
}
```

What should the signature of the method look like:

Choose the correct answer:

- a) ☐ public static boolean
isRipeInBasket(Basket basket)
- b) ☐ public static boolean
isRipeInBasket(Basket<Apple> basket)
- c) ☐ public static boolean
isRipeInBasket(Basket<?> basket)
- d) ☐ public static boolean
isRipeInBasket(Basket<? extends Apple> basket)
- e) ☐ public static <A extends Apple> boolean
isRipeInBasket(Basket<A> basket)
- f) ☐ public static <A> boolean
isRipeInBasket(Basket<A extends Apple> basket)
- g) ☐ public static boolean
isRipeInBasket(Basket<T super Apple> Basket)

Quiz: Java 1.5 Generics: Question No. 8/14

Now we want to implement a method which inserts only ripe apples into the basket. Here the method's body:

```
{  
    if (isRipe(apple)) { // 1  
        basket.setElement(apple); // 2  
    }  
}
```

Which of these signatures should we use?

Choose the correct answer:

- a) ☐ public static void
insertRipe(Apple apple, Basket<Apple> basket)
- b) ☐ public static void
insertRipe(Apple apple, Basket<? extends Apple> basket)
- c) ☐ public static void
insertRipe(Apple apple, Basket<? super Apple> basket)
- d) ☐ public static <A extends Apple> void
insertRipe(A apple, Basket<? super A> basket)
- e) ☐ public static <A super Apple> void
insertRipe(A apple, Basket<? extends A> basket)

Answer

Quiz: Java 1.5 Generics: Question No. 9/14



We could acquire some expertise in the orangeology and now we can decide whether an orange is ripe or not - and this in pure Java. Now we want to extend the class **FruitHelper**.

Here is our updated source code :

```
class FruitHelper {
    public static boolean isRipe(Apple apple) {
        ... // censored to protect our know-how
    }

    public static boolean isRipe(Orange orange) {
        ... // censored to protect our know-how
    }

    public static boolean isRipeInBasket(Basket<? extends Apple> basket) {
        Apple apple = basket.getElement();
        return isRipe(apple);
    }

    public static boolean isRipeInBasket(Basket<? extends Orange> basket) {
        Orange orange = basket.getElement();
        return isRipe(orange);
    }
}
```

Ist this source code OK?

Choose the correct answer:

- a) ☐ Yes. The source code is OK.
- b) ☐ No. The source code cannot be compiled.

Answer



Quiz: Java 1.5 Generics: Question No. 10/14

What about the following source code. Can it be compiled?

```
class FruitHelper {  
    public static boolean isRipe(Apple apple) {  
        ... // censored to protect our know-how  
    }  
  
    public static boolean isRipe(Orange orange) {  
        ... // censored to protect our know-how  
    }  
  
    public static <A extends Apple>  
    void insertRipe(A a, Basket<? super A> b)  
    {  
        if (isRipe(a)) {  
            b.setElement(a);  
        }  
    }  
  
    public static <G extends Orange>  
    void insertRipe(G g, Basket<? super G> b)  
    {  
        if (isRipe(g)) {  
            b.setElement(g);  
        }  
    }  
}
```

Choose the correct answer:

- a) ☐ Yes. The source code is OK.
- b) ☐ No. The source code cannot be compiled.

Answer

Quiz: Java 1.5 Generics: Question No. 11/14

The accounting departement needs to know, how many baskets we produce. So we've changed the class **Basket**:

```
public class Basket<E> {  
  
    ...  
  
    private static int theCount = 0;  
    public static int count() {  
        return theCount;  
    }  
  
    Basket() {  
        ++theCount;  
    }  
  
    ...  
}
```

What output would be produced by the following source code?

```
public static void main(String[] args) {  
    Basket<Apple> bA = new Basket<Apple>();  
    Basket<Orange> bG = new Basket<Orange>();  
    System.out.println(bA.count());  
}
```

Choose the correct answer:

- a) ☐ 1
- b) ☐ 2
- c) ☐ Compile error

Answer

Quiz: Java 1.5 Generics: Question No. 12/14

What about the following source code?

```
Basket<Orange> bG = new Basket<Orange>(); // 1
Basket b = bG; // 2
Basket<Apple> bA = (Basket<Apple>)b; // 3
bA.setElement(new Apple()); // 4
Orange g = bG.getElement(); // 5
```

Choose the correct answer:

- a) ☐ No compile error, no exception during the runtime
- b) ☐ Compile error in the line 3
- c) ☐ **ClassCastException** the line 3
- d) ☐ Compile warning in the line 3, **ClassCastException** in the line 4
- e) ☐ Compile warning in the line 3, **ClassCastException** in the line 5

Answer

Quiz: Java 1.5 Generics: Question No. 13/14

And what about this one?

```
Basket<Orange> bG = new Basket<Orange>(); // 1
Basket<? extends Fruit> b = bG; // 2
if (b instanceof Basket<Apple>) { // 3
    Basket<Apple> bA = (Basket<Apple>) b; // 4
    bA.setElement(new Apple()); // 5
} // 6
Orange g = bG.getElement(); // 7
```

Choose the correct answer:

- a) ☐ No compiler error, no exception
- b) ☐ Compiler error in the line 3
- c) ☐ Compiler warning in the lines 3 and 4. An exception will be thrown in the line 7.

Answer

Quiz: Java 1.5 Generics: Question No. 14/14

The last question is about arrays and generics. Which of the following lines can be compiled?

Check all correct options:

- a) ☐ `Basket<Apple>[] b = new Basket<Apple>[10];`
- b) ☐ `Basket<?>[] b = new Basket<Apple>[10];`
- c) ☐ `Basket<?>[] b = new Basket<?>[10];`
- d) ☐ `public <T> T[] test() {return null;}`
- e) ☐ `public <T> T[] test() {return new T[10];}`

Answer

SOLUTIONS DU QUIZZ

- 1/14 {b}
- 2/14 {c}
- 3/14 {d}
- 4/14 {a,b,c,f}
- 5/14 {a,b}
- 6/14 {d}
- 7/14 {d,e}
- 8/14 {d}
- 9/14 {b}
- 10/14 {a}
- 11/14 {b}
- 12/14 {e}
- 13/14 {b}
- 14/14 {c,d}