Chapitre 8
Java Generics

A titre informatif
Ce chapitre ne fait pas partie de la matière

Que sont les Generics ?

- Les bugs logiciels sont plus ou moins faciles à repérer et corriger en fonction du moment où ils sont détectés.

- Les bugs identifiés à la compilation (compile-time bugs) sont automatiquement et rapidement repérés par le compilateur. Le compilateur peut aussi fournir un message d'erreur permettant de comprendre le problème et comment le solutionner.

- Les bugs se produisant à l'exécution (runtime bugs) sont plus difficiles à détecter. Ils ne surviennent pas automatiquement et il n'est pas toujours évident de trouver la portion de code qui produit le bug.

Que sont les Generics ?

- Les Generics permettent de **typer les collections de données**
- Avant Java 1.5, les collections n'étaient pas typées.
  - exemple: utilisation d'un Vector

```
Vector v = new Vector();
v.add(new String("bonjour"));
v.add(new Integer(42));
v.add(new Object());
String s = (String) v.get(0);
int i = ((Integer)v.get(1)).intValue();
Object o = v.get(2);
```

- A partir de Java 1.5, les collections peuvent être typées.
  - exemple: utilisation d'une ArrayList de String

```
List<String> maliste = new ArrayList<>();
maliste.add(new String("bonjour"));
String s = maliste.get(0);
```

Pourquoi les Generics ?

- **Détection des erreurs de typage à la compilation.**
  - Typer les collections permet de réaliser le type-checking à la compilation
  - Les erreurs de typage (par exemple: insérer un objet d'un type non compatible avec le typage de la collection) sont identifiées à la compilation au lieu de lever une ClassCastException lors de l'exécution (au runtime).
- Plus besoin de caster lors de la récupération des objets de la collection (cf. slide précédent).
- Rend plus facile l'implémentation d'algorithmes génériques
  - Par exemple, des algorithmes traitant de collections de types différents.

Syntaxe et vocabulaire

- On peut définir :
  - des **types de données** génériques
  - des **méthodes** génériques

Types de données génériques

(Generic ADT)

```
// exemple de déclaration d'un ADT générique
public class MonGenericADT < T1, T2, …, Tn > {
   …
}
```

paramètres de type formels

- Déclaration de paramètres de type accessibles à l'ensemble de la classe `MonGenericADT`
- Les paramètres de type sont placés après le nom de classe entre chevrons (< >)
- Les paramètres de type peuvent être de n'importe quel type **non-primitif** (ADT, interfaces, tableaux)

Types de données génériques

(Generic ADT)

```
// exemple de déclaration d'une méthode utilisant les
// paramètres de type déclarés au niveau de l'ADT
 public class MonGenericADT <T1, T2, …, Tn > {

    public T1 doSomething(T2 input) { … }
}
```

- Les paramètres de type $T_1, T_2, \ldots, T_n$ peuvent être utilisés dans les signatures (et implémentations) des méthodes de l'ADT.
  - comme type pour les paramètres formels des méthodes
  - comme type de retour des méthodes

Méthodes génériques

```
public class MonGenericADT {

    // exemple de signature d'une méthode générique
    public <U> void MaMethodeGenerique (String s, U t1) {
        …
    }
}
```

paramètre de type générique

- Déclaration d'un type générique **restreint** à une méthode (ou procédure abstraite)
- **U** est un nouveau type formel introduit lors de la déclaration de la méthode
  - il est indiqué entre < > juste avant le type de retour de la méthode
- La portée des paramètres formels de type générique est limitée à la méthode

Méthodes génériques

```
public class MonGenericADT <T> {

    // exemple de signature d'une méthode générique
    public <U> void MaMethodeGenerique (T t1, String s, U t2)
    {
        …
    }
}
```

paramètres de type génériques

- **T** est un type formel introduit lors de la déclaration du type de données
- **U** est un nouveau type formel introduit lors de la déclaration de la méthode
  - il est indiqué entre < > juste avant le type de retour de la méthode
- La portée des paramètres formels de type générique est limitée à la méthode

Convention de nommage

- Les paramètres de type sont identifiés par **une seule lettre majuscule**.
  - E - Elément (utilisé partout dans le Java Collections Framework)
  - K - Key (clé)
  - N - Number (nombre)
  - T - Type
  - V - Value (valeur)

Utilisation typique d'un Generic

- Exemple: stockage d'objets dans une List et parcours des éléments de la List

```
// déclaration d'un objet de type List de String et
// instanciation sous forme d'une ArrayList de String
List<String> sList = new ArrayList<>();

// ajout d'instance de String dans la List
sList.add("bonjour");
sList.add(" tout");
sList.add(" le");
sList.add(" monde");
sList.add(" !");

 // parcours des éléments de la List
 for (String s : sList) {
     System.out.print(s); // instruction d'illustration
 }
```

Utilisation typique d'un Generic

- Exemple: stockage d'objets dans une List et parcours des éléments de la List

```
// déclaration d'un objet de type List de String et
// instanciation sous forme d'une ArrayList de String
List<String> sList = new ArrayList<>();

// ajout d'instance de String dans la List
sList.add("bonjour");
sList.add(" tout");
sList.add(" le");
sList.add(" monde");
sList.add(" !");

// parcours des éléments de la List
for (String s : sList) {
   System.out.print(s); // instruction d'illustration
}
```

Utilisation typique d'un Generic

- Exemple: stockage d'objets dans une List et parcours des éléments de la List

```
// déclaration d'un objet de type List de String et
// instanciation sous forme d'une ArrayList de String
List<String> sList = new ArrayList<>();

// ajout
sList.a
sList.
sList.a
sList.a
sList.add

// parcours des éléments de la List
 for (String s : sList) {
    System.out.print(s); // instruction d'illustration
 }
```

Comment le compilateur connaît-il le typage de la List ?

Utilisation typique d'un Generic

- Exemple: stockage d'objets dans une List et parcours des éléments de la List

```
// déclaration d'un objet de type List de String et
// instanciation sous forme d'une ArrayList de String
List<String> sList = new ArrayList<>();

// ajout
sList.a
sList.
sList.a
sList.a
sList.add

// parcours des éléments de la List
for (String s : sList){
    System.out.p          instruction d'illustration
}
```

Comment le compilateur connaît-il le typage de la List ?

mécanisme de « type inference »

Mécanisme de "type inference"

- La **type inference** est la capacité du compilateur Java d'inférer le type des paramètres pour rendre l'invocation de méthode applicable.
- L'algorithme d'inférence détermine le type des paramètres, et si disponible, le type de retour de la méthode.
- L'algorithme d'inférence cherche le type **le plus spécifique** convenant pour chaque paramètre (cf. Chapitre 2)

Mécanisme de "type inference"

- Exemple :

```
/**
 * @overview Fournit des procédures pour récupérer les éléments
 * d'une List<T> sous forme d'une List d'un autre type.
 */
public class BoxHelper {
    /**
     * @throws NullPointerException si boxes == null
     * @throws EmptyException si boxes.isEmpty()
     * @return une liste contenant les représentations sous forme de
     * chaines de caractères des éléments de boxes
     * (un élément dans la liste retournée par élément de boxes), sinon.
     */
    public static <T> List<String> toListOfString(List<T> boxes)
            throws NullPointerException, EmptyException {
        if (boxes == null) throw new EmptyException("…");
        if (boxes.isEmpty()) throw new EmptyException("…");
        List<String> returnedList = new ArrayList<>();
        for (T element : boxes) { returnedList.add(element.toString()); }
        return returnedList;
    }
    …
```

Mécanisme de "type inference"

```
    /**
     * @modifies boxes
     * @throws NullPointerException si boxes == null
     * @throws EmptyException si boxes.isEmpty()
     * @return une liste contenant les représentations sous forme de byte[]
     * des éléments de boxes (un élément dans la liste retournée par élément
     * de boxes), sinon.
     */
    public static <T> List<byte[]> toListOfByteArrays(List<T> boxes)
            throws NullPointerException, EmptyException {
        if (boxes == null) throw new EmptyException("…");
        if (boxes.isEmpty()) throw new EmptyException("…");
        List<byte[]> returnedList = new ArrayList<>();
        for (T element : boxes) {
            returnedList.add(element.toString().getBytes());
        }
        return returnedList;
    }
}
```

Mécanisme de "type inference"

```
public class TestBoxHelper {
    public static void main (String[] args) {
        List<Integer> box = new ArrayList<>();
        box.add(10);
        box.add(20);
        box.add(30);
        box.add(40);

        List<String> sBox = null;
        try {
            sBox = BoxHelper.toListOfString(box);
        } catch (NullPointerException e) {
            System.err.println("La liste est null");
        } catch (EmptyException e) {
            System.err.println("La liste est vide");
        }

        System.out.println(sBox);
    }
}
```

```
[10, 20, 30, 40]
```

Mécanisme de "type inference"

```
public class TestBoxHelper {

    public static void main (String[] args) {
        List<Article> box = new ArrayList<>();
        box.add(new Article("Queen", "Radio GaGa", "45 tours", 29.5f));
        box.add(new Article("RHCP", "Snow", "MP3", 9.5f));
        box.add(new Article("Black Sabbath", "Crazy train", "33 tours", 69.5f));

        List<String> sBox = null;
        try {
            sBox = BoxHelper.toListOfString(box);
        } catch (NullPointerException e) {
            System.err.println("La liste est null");
        } catch (EmptyException e) {
            System.err.println("La liste est vide");
        }

        System.out.println(sBox);
        …
}
```

```
[<Queen, Radio GaGa, 45 tours, 29.5>, <RHCP, Snow, MP3,
9.5>, <Black Sabbath, Crazy train, 33 tours, 69.5>]
```

Mécanisme de "type inference"

```
        List<byte[]> bBox = null;
        try {
            bBox = BoxHelper.toListOfByteArrays(box);
        } catch (NullPointerException e) {
            System.err.println("La liste est null");
        } catch (EmptyException e) {
            System.err.println("La liste est vide");
        }

        for (byte[] b : bBox) {
            System.out.println(Arrays.toString(b));
        }
    }
}
```

```
[60, 81, 117, 101, 101, 110, 44, 32, 82, 97, 100, 105, 111, 32, 71, 97, 71, 97, 44, 32, 52, 53, 32,
116, 111, 117, 114, 115, 44, 32, 50, 57, 46, 53, 62]
[60, 82, 72, 67, 80, 44, 32, 83, 110, 111, 119, 44, 32, 77, 80, 51, 44, 32, 57, 46, 53, 62]
[60, 66, 108, 97, 99, 107, 32, 83, 97, 98, 98, 97, 116, 104, 44, 32, 67, 114, 97, 122, 121, 32, 116,
114, 97, 105, 110, 44, 32, 51, 51, 32, 116, 111, 117, 114, 115, 44, 32, 54, 57, 46, 53, 62]
```

"Diamond"

- Le terme "**diamond**" fait référence à l'utilisation des symboles suivant : <>
- On utilise le **diamond** comme sucre syntaxique pour éviter d'indiquer le type effectif des paramètres de type formels lors de l'appel au constructeur d'un type générique
- L'utilisation du **diamond** est possible lorsque le compilateur est capable d'inférer le type effectif des paramètres de type formels du constructeur
- Exemple :

```
List<Integer> myIntList = new ArrayList<Integer>();
```

"Diamond"

- Le terme "**diamond**" fait référence à l'utilisation des symboles suivant : <>
- On utilise le **diamond** comme sucre syntaxique pour éviter d'indiquer le type effectif des paramètres de type formels lors de l'appel au constructeur d'un type générique
- L'utilisation du **diamond** est possible lorsque le compilateur est capable d'inférer le type effectifs des paramètres de type formels du constructeur
- Exemple :

```
List<Integer> myIntList = new ArrayList<Integer>();
```

"Diamond"

- Le terme "**diamond**" fait référence à l'utilisation des symboles suivant : <>
- On utilise le **diamond** comme sucre syntaxique pour éviter d'indiquer le type effectif des paramètres de type formels lors de l'appel au constructeur d'un type générique
- L'utilisation du **diamond** est possible lorsque le compilateur est capable d'inférer le type effectifs des paramètres de type formels du constructeur
- Exemple :

```
List<Integer> myIntList = new ArrayList<>();
```

Raw types

- Pour des raisons (entre autres) de rétrocompatibilité, il est possible d'utiliser les collections définies en Java sans les typer.
- Il est également possible d'utiliser des ADTs ou de définir de nouveaux ADTs sans utiliser le mécanisme des generics.
- Cette utilisation est hautement déconseillée **et proscrite dans le cadre de ce cours !**
- Utiliser un raw type lèvera un warning à la compilation :

```
List myIntList = new ArrayList();
```

Raw types

- Pour des raisons (entre autres) de rétrocompatibilité, il est possible d'utiliser les collections définies en Java sans les typer.
- Il est également possible d'utiliser des ADTs ou de définir de nouveaux ADTs sans utiliser le mécanisme des generics.
- Cette utilisation est hautement déconseillée **et proscrite dans le cadre de ce cours !**
- Utiliser un raw type lèvera un warning à la compilation :

```
List myIntList = new ArrayList();
```

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

Paramètres de type bornés

- en Anglais : **Bounded Type Parameters**
- permet de limiter/borner le type des paramètres effectifs utilisés lors de la création d'un objet d'un ADT générique ou lors de l'appel d'une méthode générique
- Syntaxe :

```
<T extends SomeADT>
```

- Utilisation du mot clé **`extends`** que le type étendu soit une classe ou une interface Java
- Syntaxe en cas de plusieurs supertypes :

```
<T extends ADT1 & ADT2 & ADT3>
```

en commençant par les classes avant les interfaces !

Wildcard

- L'utilisation d'un "?" appelé **wildcard** permet de faire référence à un ADT inconnu
  Exemple : List<?> signifie une liste d'un type inconnu

- Possibilité de définir soit une borne supérieure, soit une borne inférieure pour la wildcard
  - Borne supérieure : <? extends SomeADT>
  - Borne inférieure : <? super SomeADT>
  - une seule borne à la fois

Wildcard

- Exemple d'utilisation de la **wildcard**

Soit la procédure `afficherListe` :

```
public static void afficherListe(List<Object> liste) {
     for (Object element : liste) {
         System.out.println(element + " ");
     }
 }
```

- afficherListe a pour but d'afficher le contenu d'une liste de n'importe quel type.
- mais cette implémentation ne permet pas de remplir cette exigence : cette procédure n'affiche que le contenu des listes d'`Object`.

Wildcard

- Solution: utiliser une wildcard

```
public static void afficherListe(List<?> liste) {
     for (Object element : liste) {
         System.out.println(element + " ");
     }
 }
```

- Maintenant, les appels suivants fonctionneront comme prévu :

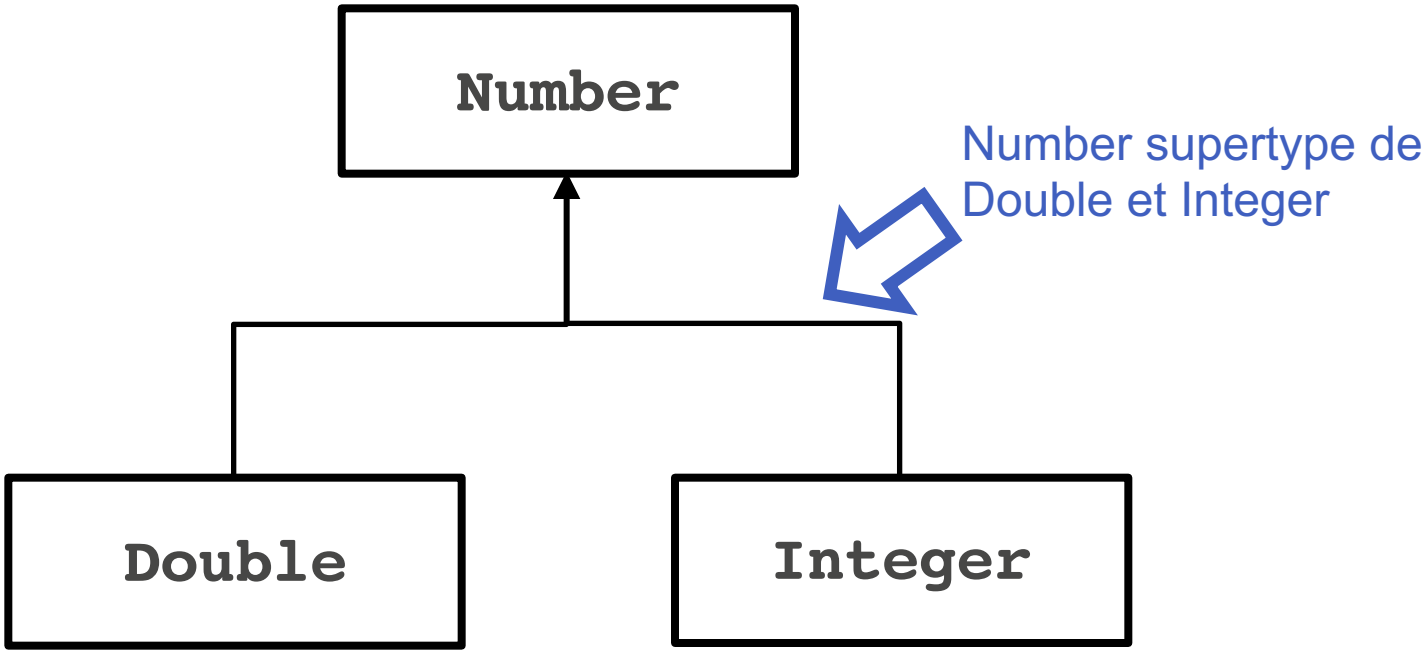
```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("une", "deux", "trois");
afficherListe(li);
afficherListe(ls);
```

Wildcard *vs.* Bounded Type Parameter

Wildcard	Paramètre de type borné
• À utiliser comme type de paramètre formel d'une méthode générique quand les objets de type *wildcard* sont manipulés à travers des méthodes de la classe `Object`. • Pour avoir une borne supérieure et/ou une borne inférieure • Ne peut pas être utilisé comme paramètre pour définir une classe générique • Ne peut pas être utilisé pour déclarer une variable de la méthode • Ne devrait jamais être utilisé pour définir un type de retour	• Peut avoir plusieurs bornes supérieures • Ne peut pas avoir de borne inférieure • Peut être utilisé pour déclarer une variable dans l'implémentation de la classe/méthode • Peut être utilisé pour définir le type de retour d'une méthode générique

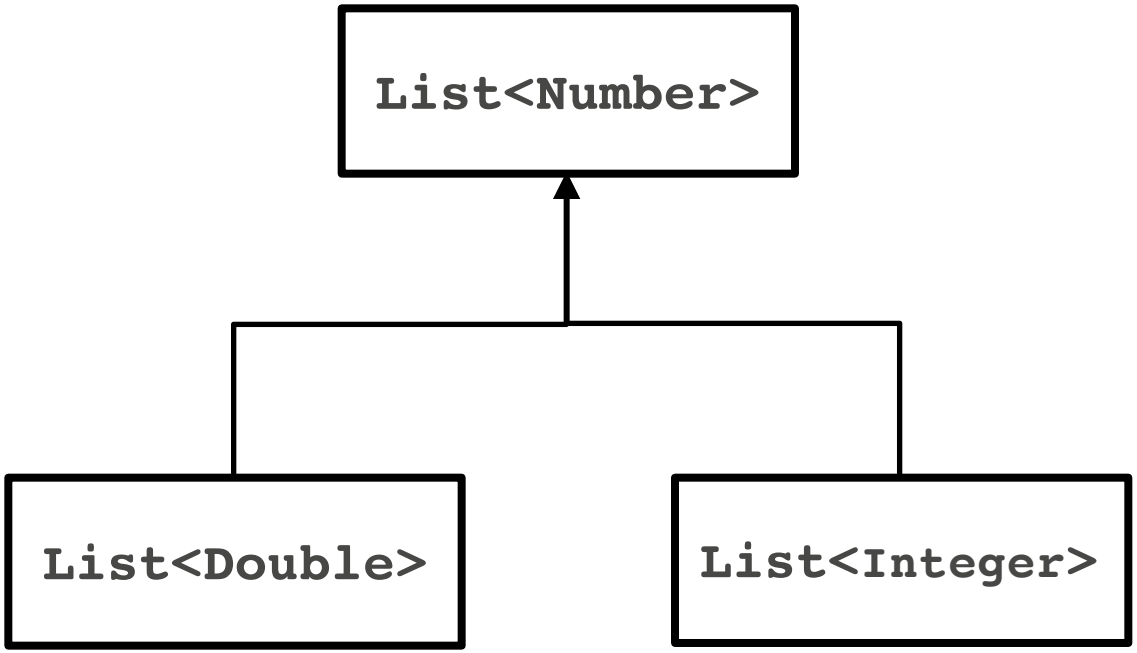
Generics et Hiérarchie d'ADT

- Soit les types `Number`, `Integer` et `Double`
- Nous savons que :



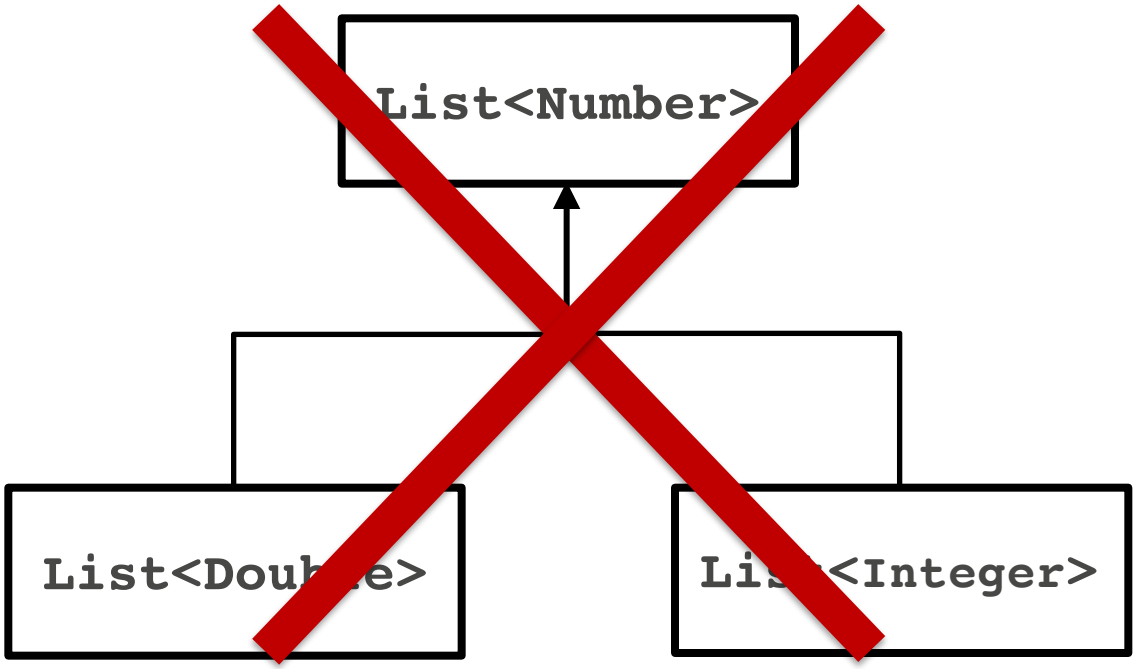
Generics et Hiérarchie d'ADT

- Est-ce que `List<Number>` est un supertype de `List<Double>` et `List<Integer>` ?



Generics et Hiérarchie d'ADT

- Est-ce que `List<Number>` est un supertype de `List<Double>` et `List<Integer>` ?

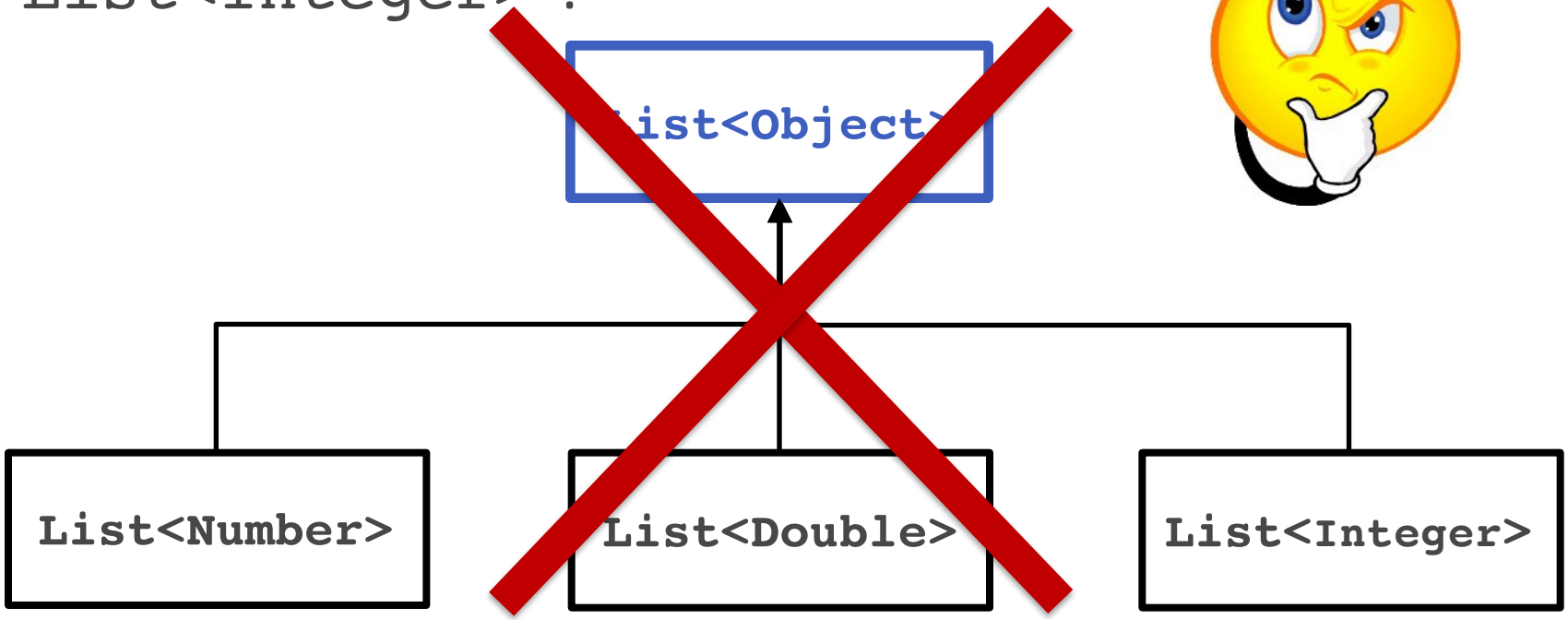


Generics et Hiérarchie d'ADT

- Est-ce que `List<Object>` est un supertype de `List<Number>` et `List<Double>` et `List<Integer>` ?

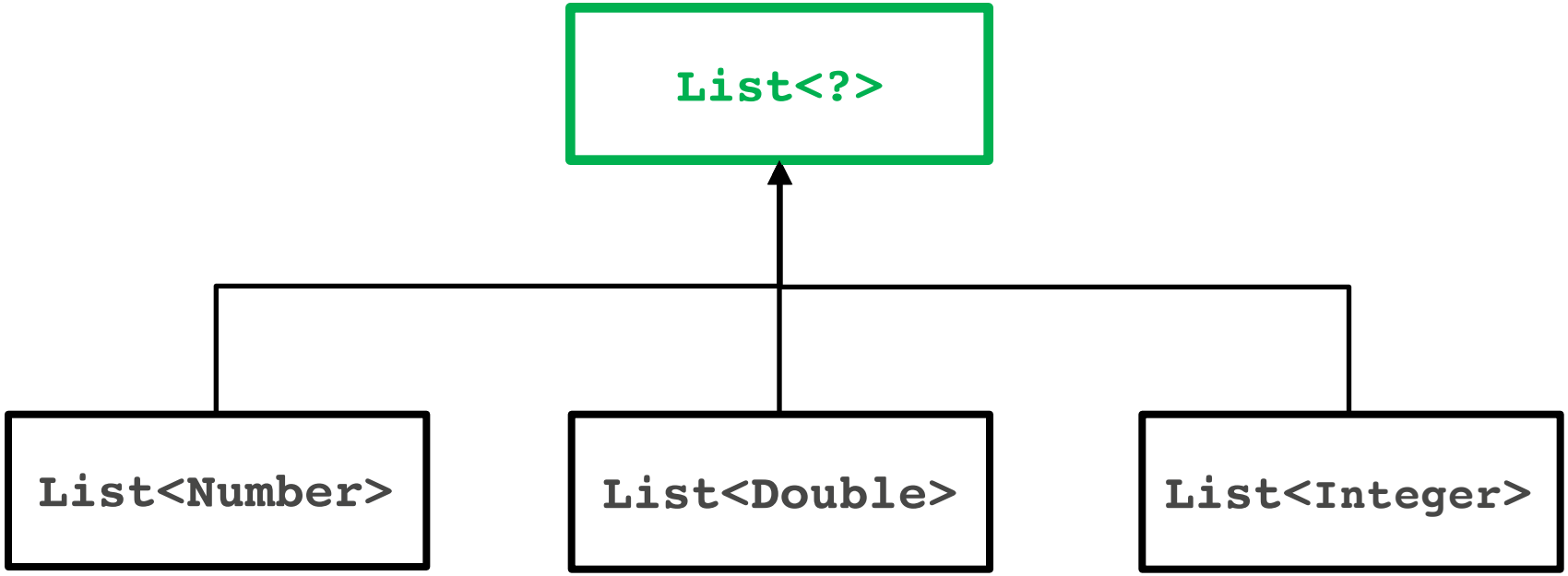
Generics et Hiérarchie d'ADT

- Est-ce que `List<Object>` est un supertype de `List<Number>` et `List<Double>` et `List<Integer>` ?



Generics et Hiérarchie d'ADT

- Le supertype commun est `List<?>`



Generics et Hiérarchie d'ADT

- Pourquoi l'héritage entre ADTs génériques ne fonctionne-t-il pas comme on pourrait s'y attendre ?

- Exemple :

```
// déclaration d'une List d'Integer et ajout de 3 Integers
List<Integer> lInteger = new ArrayList<>();
lInteger.add(5); lInteger.add(2); lInteger.add(33);
```

admettons que l'assignation suivante ne génère pas d'erreur à la compilation :

```
List<Object> lObject = lInteger;
```

alors, les instructions suivantes seraient tout à fait légitimes :

```
lObject.add(new Object());
lObject.add(new Object());
```

Generics et Hiérarchie d'ADT

mais, les instructions suivantes seraient également légitimes :

```
int accumulateur = 0;
for (Integer i : lInteger) {
    accumulateur += i;
}
System.out.println("accumulateur = " + accumulateur);
```

or, les instructions ci-dessus ne peuvent s'exécuter correctement si l'`ArrayList` référencée par `lInteger` (et par `lObject`) contient des objets de types effectifs `Object`

Type erasure

- Mécanisme mis en œuvre par le compilateur
- Consiste à :
  - remplacer tous les paramètres de type génériques par leur(s) borne(s) ou `Object` pour les paramètres de type non bornés.
  - insérer des castings de type si nécessaire pour préserver la type safety
  - générer des méthodes "bridge" pour préserver le polymorphisme
    - une méthode "bridge" est une méthode qui est automatiquement ajoutée par le compilateur au moment de la compilation
- Grâce au type erasure, aucune nouvelle classe n'est créée pour les ADT génériques, et donc aucun *overhead* à l'exécution du programme
- Mais le type erasure rend un ADT générique **non-réifiable**

Type erasure :
Remplacement des paramètres de type générique

- Soit la classe `Node<T>` dont l'implémentation est donnée ci-dessous :

```
public class Node<T> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }

    …
}
```

Type erasure :
Remplacement des paramètres de type générique

- Puisque le paramètre `T` n'est pas borné, le compilateur le remplace par `Object` :

```
public class Node {
    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }

    …
}
```

Type erasure :
Remplacement des paramètres de type générique

- Une autre version de la classe `Node<T extends Comparable<T>>` dont le paramètre de type générique est borné :

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    …
}
```

Type erasure :
Remplacement des paramètres de type générique

- Dans ce cas, le compilateur remplace par le paramètre `T` par `Comparable` :

```
public class Node {

  private Comparable data;
  private Node next;

  public Node(Comparable data, Node next) {
      this.data = data;
      this.next = next;
  }

  public Comparable getData() { return data; }

  …
}
```

Type erasure :
Remplacement des paramètres de type générique

- Soit une abstraction procédurale dans laquelle est définie un paramètre de type générique `T`

```
public class GenericMethodTypeErasure {
    /**
     * @requires tab != null et elem != null
     * @return  le nombre d'occurrences d'elem dans tab
     */
    public static <T> int compter(T[] tab, T elem) {
        int cnt = 0;
        for (T e : tab) {
            if (e.equals(elem)) { ++cnt; }
        }
        return cnt;
    }
}
```

Type erasure :
Remplacement des paramètres de type générique

- Dans ce cas, le compilateur remplace le paramètre `T` par `Object` :

```
public class GenericMethodTypeErasure {

    /**
     * @requires tab != null et elem != null
     * @return  le nombre d'occurrences d'elem dans tab
     */
    public static int compter(Object[] tab, Object elem) {
        int cnt = 0;
        for (Object e : tab) {
            if (e.equals(elem)) { ++cnt; }
        }
        return cnt;
    }
}
```

Type erasure :
Casting et méthodes "bridge"

- Considérons les ADTs génériques `Node<T>` et `MonNode` :

```
public class Node<T> {

    protected T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

Type erasure :
Casting et méthodes "bridge"

- Considérons les ADTs génériques `Node<T>` et `MonNode` :

```
public class MonNode extends Node<Integer> {

   public MonNode(Integer data) { super(data); }

   public void setData(Integer data) {
      System.out.println("MonNode.setData");
      super.setData(data);
   }
}
```

Type erasure :
Casting et méthodes "bridge"

- Et considérons aussi ces instructions :

```
public class TestMonNode {

   public static void main(String[] args) {
      Node<Integer> n = new MonNode(42);
      n.setData(25);
   }
}
```

- L'affichage à l'exécution donne :

```
MonNode.setData
Node.setData
```

Type erasure :
Casting et méthodes "bridge"

- Comment cela est-il possible puisqu'après la mise en place du type erasure, le code à exécuter est :

```
public class Node {
    protected Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

Type erasure :
Casting et méthodes "bridge"

- Comment cela est-il possible puisqu'après la mise en place du type erasure, le code à exécuter est (suite) :

```
public class MonNode extends Node {

    public MonNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MonNode.setData");
        super.setData(data);
    }
}
```

Type erasure :
Casting et méthodes "bridge"

- Et donc l'appel à `setData(25)` sur `n` (de type déclaré `Node<Integer>`) **aurait dû** provoquer l'affichage en console de "`Node.setData`" uniquement (cf. slides précédents et la notion de Dispatch Vector)

```
public class TestMonNode {

    public static void main(String[] args) {
        Node<Integer> n = new MonNode(42);
        n.setData(25);
    }
}
```

Type erasure :
Casting et méthodes "bridge"

- Cela fonctionne grâce à l'ajout d'une méthode "brigde" par le compilateur et à l'introduction de casting, également par le compilateur

- Conceptuellement, le code de la classe `MonNode` après réalisation du type erasure contient donc une méthode `public void setData(Object data)`
(voir slide suivant)

Type erasure :
Casting et méthodes "bridge"

```
public class MonNode extends Node {

    public MonNode(Integer data) { super(data); }

    public void setData(Object data){
        setData((Integer)data);
    }

    public void setData(Integer data) {
        System.out.println("MonNode.setData");
        super.setData(data);
    }
}
```

méthode "bridge" générée par le compilateur

casting introduit par le compilateur