Généricité avancée

Théorie et pratique de la programmation
Michel Schinz – 2013-04-29

Sous-typage

Sous-typage

Rappel : en Java, les classes et les interfaces définissent des types. Ces types sont liés entre-eux par une relation de sous-typage.
Par exemple, le type `String` est un **sous-type** du type `Object` car la classe `String` hérite de la classe `Object`.

De manière similaire, le type `Number` est un sous-type du type `Serializable` car la classe `Number` implémente l'interface `Serializable`.

Lorsqu'un type T_2 est sous-type d'un type T_1, on dit que T_1 est un **super-type** de T_2.

Propriétés du sous-typage

Formellement, la relation de sous-typage est :

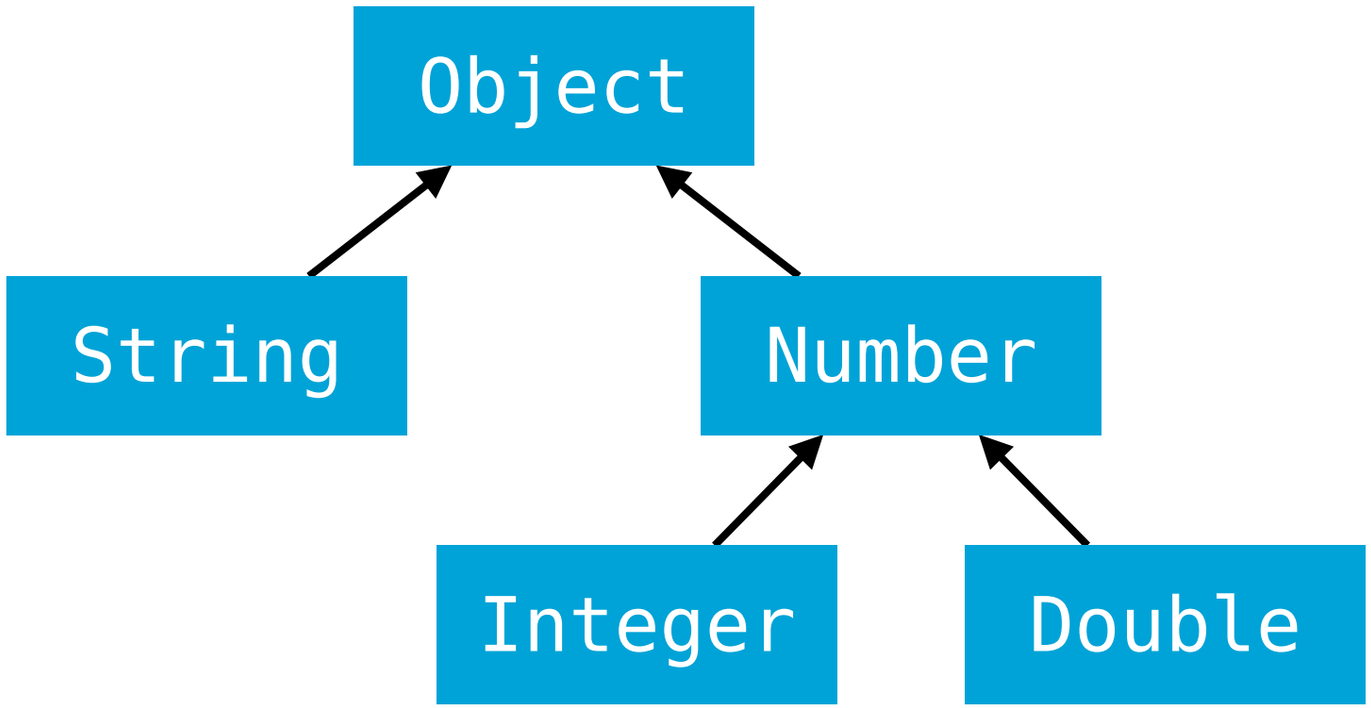
- **réflexive**, c-à-d que tout type est sous-type de lui-même,
- **transitive**, c-à-d que si un type T_1 est sous-type d'un type T_2 et T_2 est sous-type de T_3, alors T_1 est aussi sous-type de T_3,
- **anti-symétrique**, c-à-d que si T_1 est sous-type de T_2 et T_2 est sous-type de T_1, alors $T_1 = T_2$.

En mathématiques, une relation possédant ces trois propriétés est appelée **ordre partiel**.

Graphe des types

La relation de sous-typage peut être visualisée sous la forme d'un graphe dirigé dans lequel chaque type est un nœud et un arc lie le nœud d'un type à celui de ses super-types.

Un (minuscule) extrait du graphe des types standard Java :



Q : comment les types `Integer` et `Object` sont-ils liés par cette relation de sous-typage ? Et `String` et `Number` ?

Polymorphisme d'inclusion

Le **polymorphisme d'inclusion** permet de substituer à une valeur d'un type T_1 donné une valeur d'un autre type T_2 pour peu que T_2 soit un sous-type de T_1. On appelle aussi cela le **principe de substitution** (*substitution principle*).
Par exemple, si une fonction prend en argument une valeur de type `Number`, en plus d'une valeur de ce type on peut lui passer une valeur de type `Integer`, `Double`, etc.

```
Number add(Number n1, Number n2) {
  return new Double(n1.doubleValue()
                    + n2.doubleValue());
}
add(new Integer(1), new Double(3.14));
```

Sous-typage et généricité

Listes génériques

Pour illustrer les concepts que nous allons examiner, nous réutiliserons notre interface des listes génériques :

```
interface List<E> extends Iterable<E> {
  boolean isEmpty();
  int size();
  void add(E newElem);
  void remove(int index);
  E get(int index);
  void set(int index, E elem);
  Iterator<E> iterator();
}
```

Liste de nombres

Le principe de substitution nous permet d'ajouter n'importe quel type de nombre – c-à-d n'importe quel sous-type de `Number` – à une liste de `Number` :

```
List<Number> l = new LinkedList<>();
Integer i = 1;
l.add(1);
Double d = 3.14;
l.add(3.14);
```

Chacun des deux appels à `add` est valide car `Integer` et `Double` sont des sous-types de `Number`.

Ajout groupé

Pour pouvoir plus facilement ajouter tous les éléments d'une liste à une liste existante, essayons de définir une méthode `addAll` dans `List`. Premier essai :

```
interface List<E> {
  …
  void addAll(List<E> other);
}
class LinkedList<E> implements List<E> {
  …
  void addAll(List<E> other) {
    for (E e: other)
      add(e);
  }
}
```

Sous-typage et généricité

Malheureusement, la méthode `addAll` que nous venons de définir n'est pas utilisable comme nous le désirerions :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
li.add(1);
l.addAll(li); // refusé !
```

Généricité et sous-typage

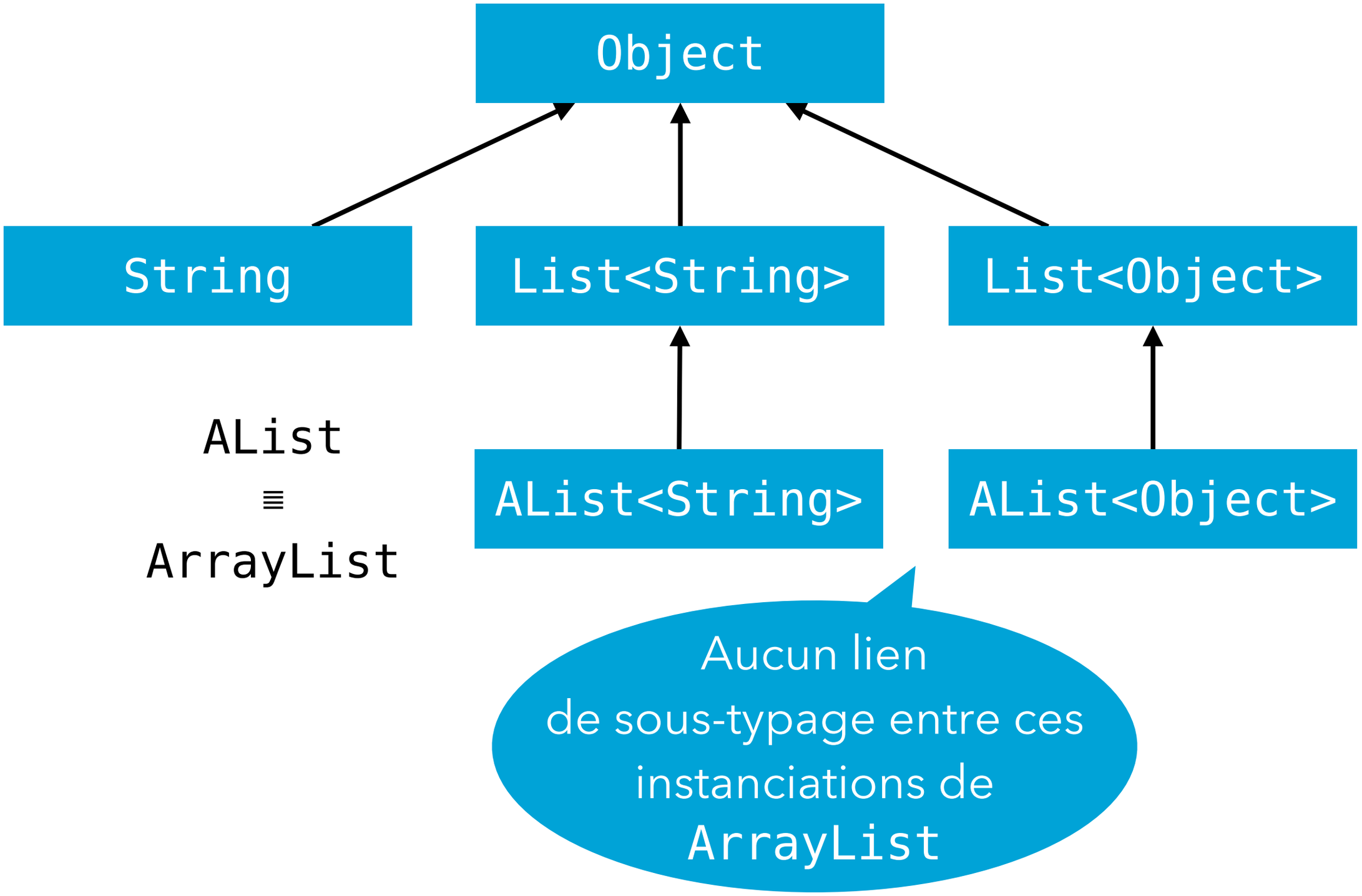
Le code précédent est refusé car, en Java, une instanciation d'un type générique n'est *jamais* sous-type d'une autre instanciation de ce même type générique...
Par exemple, le type `List<U>` n'est jamais sous-type de `List<V>` sauf dans le cas trivial où U=V.

Le seul moyen de rendre l'appel à `addAll` valide est donc de changer le type de la seconde liste pour en faire une liste de `Number`. Cela n'est pas très satisfaisant, car il est clairement valide d'ajouter une liste d'entiers à une liste de nombres. Il nous faudra trouver une solution plus tard !

Généricité et sous-typage

Attention, la restriction mentionnée ne signifie pas que deux types génériques **différents** ne peuvent pas être liés par une relation de sous-typage !
Par exemple, `ArrayList<String>` est un sous-type de `List<String>` car la classe générique `ArrayList<E>` implémente l'interface `List<E>`.
Par contre, deux instanciations différentes du **même** type générique ne sont jamais liées entre-elles par une relation de sous-typage.

Généricité et sous-typage



AList
≡
ArrayList

Justification de la limitation

Pourquoi les concepteurs de Java ont-ils choisi d'imposer cette restriction sur la généricité ?
Pour le comprendre, admettons que `List<Integer>` soit un sous-type de `List<Number>`. Cela nous autoriserait à écrire le code suivant :

```
void addPi(List<Number> l) {
  l.add(3.14);
}
List<Integer> l = new LinkedList<>();
addPi(l);
```

Quel problème cela pose-t-il ?

`addAll` v2

Question : n'est-il pas possible de définir une méthode `addAll` qui soit plus générale que celle définie précédemment, et qui permette l'ajout – valide – d'une liste d'entiers à une liste de nombres ?

addAll v2

Réponse : oui, en la rendant générique et bornée !

```
interface List<E> {
  …
  <F extends E> void addAll(List<F> other);
}
```

addAll v2

Avec cette nouvelle définition :

```
interface List<E> {
  …
  <F extends E> void addAll(List<F> other);
}
```

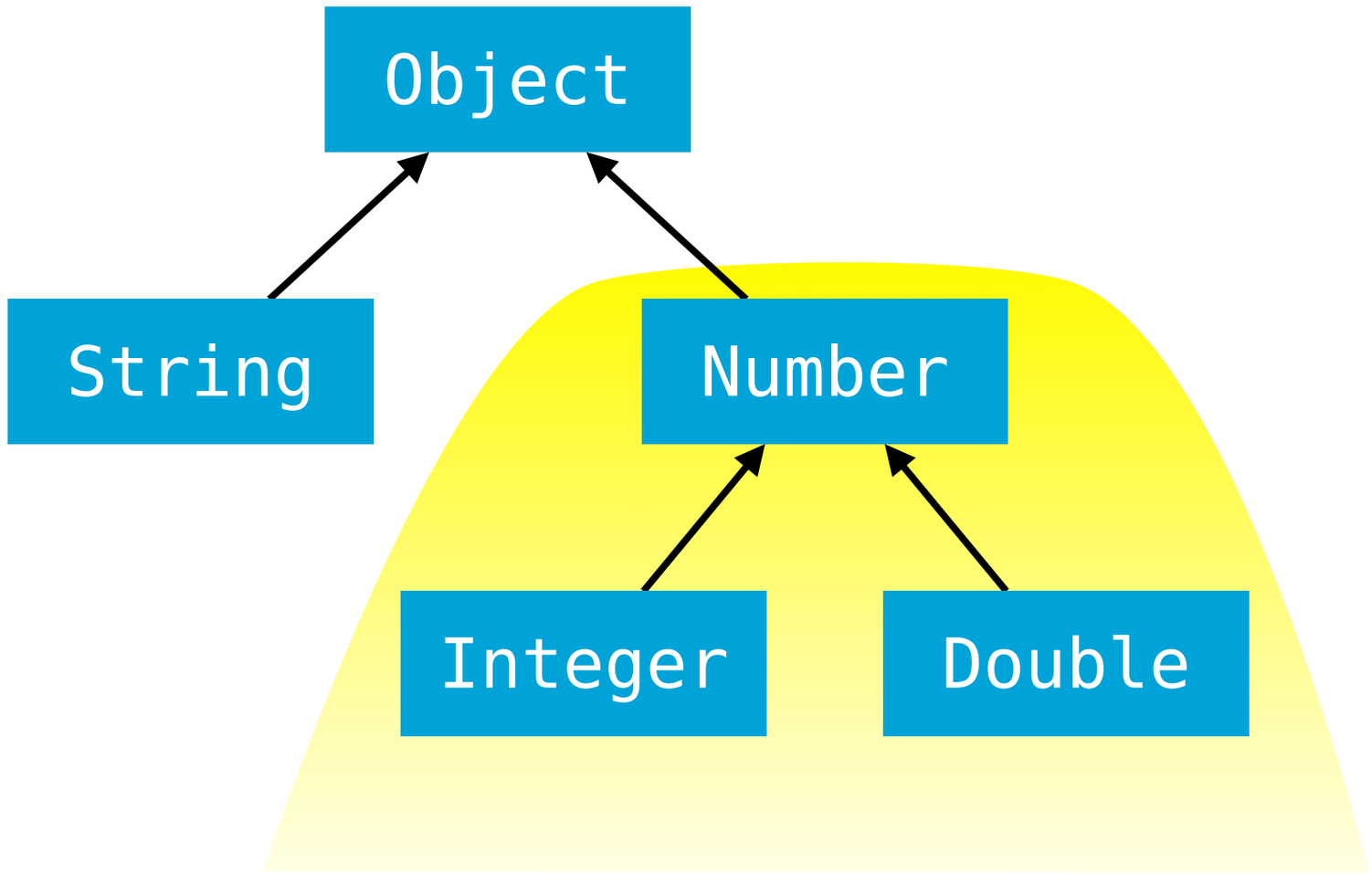
le code précédent est maintenant valide :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
li.add(1);
l.addAll(li);
```

Question : quel type est inféré pour le paramètre F dans l'appel à `addAll` ?

Borne supérieure

La borne (supérieure) permet l'utilisation de n'importe quel **sous-type** de la borne, ici `Number`.



addAll v2

Bien entendu, changer le type dans l'interface `List` n'a de sens que si les mises en œuvre concrètes de la méthode restent valides. Pour `addAll`, c'est le cas :

```
class LinkedList<E> implements List<E> {
  <F extends E> void addAll(List<F> other){
    for (E e: other)
      add(e);
  }
}
```

Question : pourquoi ce code est-il valide ?

Jokers

(ou *wildcards*)

Jokers

Le paramètre de type `F` de la méthode `addAll` n'est pas utilisé ailleurs que dans son type. Il n'est donc pas nécessaire de le nommer, et Java permet d'utiliser dans ce cas un **joker** (*wildcard*) borné :

```
interface List<E> {
   …
   void addAll(List<? extends E> other);
}
```

Ajout groupé inversé

Nous avons réussi à définir une méthode `addAll` satisfaisante. Essayons maintenant de définir une méthode `addAllInto` qui ajoute tous les éléments du récepteur dans la liste passée en argument. Premier essai :

```
interface List<E> {
  …
  void addAllInto(List<E> other);
}
class LinkedList<E> implements List<E> {
  …
  void addAllInto(List<E> other) {
    other.addAll(this);
  }
}
```

Ajout groupé inversé

Bien entendu, cette première version possède les mêmes limitations que notre première version de la méthode `addAll`, à savoir que le code suivant est invalide :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
li.add(1);
li.addAllInto(l); // refusé !
```

addAllInto v2

Nous pourrions bien entendu essayer de résoudre le problème de la même manière que pour `addAll`, c-à-d en utilisant un joker équipé d'une borne supérieure :

```
interface List<E> {
  …
  void addAllInto(List<? extends E> other);
}
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
li.add(1);
li.addAllInto(l);
```

Question : pourquoi cela ne fonctionne-t-il pas ?

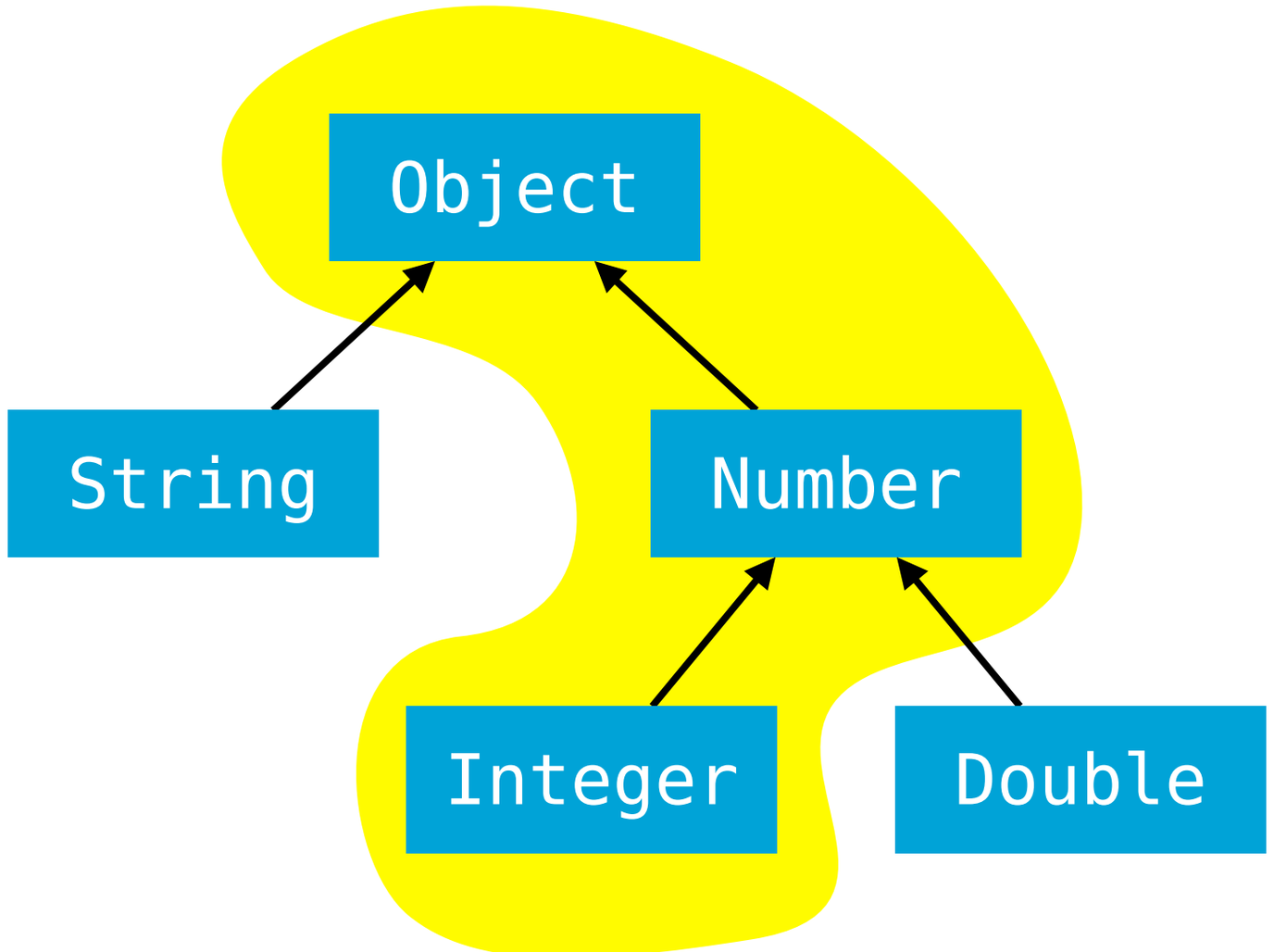
addAllInto v3

La borne de `addAllInto` doit être une borne *inférieure* et pas *supérieure* ! Heureusement, Java offre de telles bornes sur les jokers – mais pas sur les paramètres de type :

```
interface List<E> {
  …
  void addAllInto(List<? super E> other);
}
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
li.add(1);
li.addAllInto(l);
```

Borne inférieure

La borne inférieure permet l'utilisation de n'importe quel **super-type** de la borne, ici `Integer`.



Ajout groupé bidirectionnel

Admettons pour terminer que l'on désire définir une méthode `addAllFromAndInto` qui ajoute tous les éléments de l'argument au récepteur et inversement :

```
interface List<E> {
  …
  void addAllFromAndInto(List<E> other);
}
class LinkedList<E> implements List<E> {
  …
  void addAllFromAndInto(List<E> other) {
    this.addAll(other); other.addAll(this);
  }
}
```

Question : quel type de borne utiliser et pourquoi ?

Loi des bornes

La « loi des bornes » ci-dessous permet de se souvenir aisément de quelle genre de borne utiliser dans quelle situation :

- Lorsqu'on désire *uniquement lire* dans une structure, on utilise une *borne supérieure* (avec `extends`) ;
- lorsqu'on désire *uniquement écrire* dans une structure, on utilise une *borne inférieure* (avec `super`) ;
- lorsqu'on désire *à la fois lire et écrire* dans une structure, on n'utilise *aucune borne*.

(Par structure, on entend p.ex. une liste, etc.)

Principe PECS

En anglais, cette loi des bornes est parfois nommée ***PECS***, acronyme de *Producer Extends, Consumer Super*.
Cet acronyme permet de se souvenir facilement que :

- lorsque la structure que l'on utilise est un producteur, c-à-d qu'on y lit des valeurs, il faut utiliser `extends` pour borner son type, et
- lorsque la structure que l'on utilise est un consommateur, c-à-d qu'on y écrit des valeurs, il faut utiliser `super` pour borner son type.

Lorsqu'on désire à la fois lire et écrire, on ne peut borner son type.

Types bruts

Compatibilité

La généricité n'a été introduite que tardivement dans le langage Java, à un moment où beaucoup de code non-générique avait déjà été écrit.
Idéalement, tout ce code non-générique aurait dû être adapté du jour au lendemain, et la question de la compatibilité entre les deux formes de code ne se serait pas posée.
En pratique, cela n'est bien entendu pas possible, et les concepteurs de Java ont donc introduits des concepts facilitant la compatibilité entre le code générique et le code non-générique.
Nous ne considérerons ici que le cas du code non-générique utilisant du code générique.

Types bruts

Lorsque la généricité a été ajoutée à Java, sa bibliothèque standard a été modifiée pour en tirer parti. Par exemple, l'interface `List` a été transformée en `List<E>`, où `E` représente le type des éléments de la liste.

Rigoureusement, une fois cette modification faite, le type `List` (sans argument de type) est invalide et son utilisation devrait être refusée par le compilateur. Mais cela rendrait impossible la compilation de beaucoup d'ancien code...

Pour éviter ce problème, les concepteurs de Java ont introduit la notion de **type brut** (*raw type*), qui est simplement un type générique utilisé sans paramètres. Dans notre exemple, `List` est un tel type.

Types bruts

La version brute d'un type interagit avec la version générique de ce même type de la manière suivante :

- une version générique peut être utilisée partout où la version brute est attendue, sans provoquer l'affichage d'un avertissement,
- la version brute peut être utilisée partout où une version générique est attendue, mais cela provoque l'affichage d'un avertissement.

Par exemple, si on passe une valeur qui a le type brut `List` à une méthode qui attend une valeur de type `List<String>`, le code est accepté avec un avertissement.

Assertions (digression)

Propriétés du programme

Il arrive fréquemment que le programmeur sache que certaines propriétés de son programme sont vraies sans que cela ne soit directement apparent dans le code.
Exemples :

- Dans une liste chaînée, la tête (`head`) est nulle si et seulement si la taille vaut 0.
- Dans une liste chaînée circulaire, tous les nœuds ont un successeur et un prédécesseur non-nul.
- Dans un arbre binaire de recherche, tous les éléments du fils gauche sont strictement plus petits que celui à la racine, et tous les éléments du fils droit plus grands.

Expression des propriétés

Connaître ce genre de propriétés à propos d'un morceau de code facilite souvent sa compréhension. De plus, le fait qu'une condition qui devrait être vraie ne le soit pas signale la présence d'un bug.
Il est donc souvent intéressant d'exprimer ces conditions.
Comment faire ?

1. Utiliser des commentaires. Mais ils ne sont pas exécutables, donc pas moyen de vérifier qu'une condition est vraie...
2. Utiliser une expression booléenne dont on vérifie la validité.

Propriété de `List.add`, v1

Pour illustrer l'idée, utilisons la méthode `add` de `LinkedList`, qui doit distinguer deux cas : celui où la liste est vide et celui où elle ne l'est pas :

```
  void add(E newElem) {
    Node<E> newNode = new Node<>(newElem);
    if (size == 0) {
      // cas 1 : liste vide, head == null
      …
    } else {
      // cas 2 : liste non vide, head != null
      …
    }
    size += 1;
}
```

Propriété de `List.add`

Dans le premier cas, on sait que `head` doit être `null`, dans le second on sait que `head` ne doit pas l'être.

Mais que se passe-t-il si notre mise en œuvre est incorrecte et `head` ne vaut pas `null` alors que `size` vaut 0 ? Avec le code actuel, rien, le problème n'est pas détecté !

Pour le détecter, on peut ajouter du code vérifiant que la propriété est vraie et qui lève une exception dans le cas contraire.

Propriété de `List.add`, v2

```
void add(E newElem) {
  Node<E> newNode = new Node<>(newElem);
  if (size == 0) {
    // cas 1 : liste vide
    if (! (head == null))
      throw new Error("internal error!");
    …
  } else {
    // cas 2 : liste non vide
    if (! (head != null))
      throw new Error("internal error!");
    …
  }
  size += 1;
}
```

Assertions

Le code de vérification précédent a deux défauts :

1. Sa taille n'est pas négligeable puisqu'il contient au moins deux lignes.
2. Son coût à l'exécution peut être important si la condition à vérifier est chère. Ce n'est pas le cas ici mais pourrait p.ex. l'être si on vérifiait que tous les éléments du fils gauche d'un arbre binaire de recherche sont plus petits que ceux de son fils droit.

Pour résoudre ces problèmes, Java offre la notion d'**assertion**, introduite avec l'énoncé `assert`.

Propriété de `List.add`, v3

```
void add(E newElem) {
  Node<E> newNode = new Node<>(newElem);
  if (size == 0) {
    // cas 1 : liste vide
    assert head == null;
    …
  } else {
    // cas 2 : liste non vide
    assert head != null;
    …
  }
  size += 1;
}
```

Vérification des assertions

Les assertions ne sont vérifiées que si on le demande explicitement en passant l'option `-enableassertions` (ou `-ea`) à la machine virtuelle Java. Toute assertion dont l'expression est fausse provoque la levée de l'exception `AssertionError`.

Si on ne les active pas, les expressions des assertions sont ignorées, ce qui peut être utile lorsqu'on désire obtenir les meilleures performances possibles.

Message d'assertion

Il est possible d’associer un message à une assertion. En cas de violation, ce message est attaché à l’exception `AssertionError`, et apparaît ainsi à l’écran. Le message se place après l'expression booléenne, et les deux sont séparés par un double point (`:`).

Exemple :

```
assert head == null : "Tête de liste non "
  + "nulle : " + head;
```

Assertions ou exceptions

Quand faut-il utiliser une assertion et quand faut-il utiliser un test avec l'exception `IllegalArgumentException` p.ex. ?

- les assertions doivent être utilisées pour vérifier des conditions internes du programme qui devraient toujours être vraies sauf en présence d'un bug,
- les exceptions doivent être utilisées pour vérifier les paramètres passés par du code client.

Assertion ou exception ?

Pour les exemples suivants, pensez-vous qu'il est préférable d'utiliser une assertion ou un test levant une exception ?

- Pour vérifier que, dans un arbre binaire de recherche, le plus petit élément du fils droit est strictement supérieur à l'élément à la racine ?
- Pour vérifier qu'une méthode de calcul de factorielle est bien appliquée à un entier positif ?
- Pour vérifier que l'index passé à la méthode `get` de `List` soit dans les bornes acceptables ?
- Pour vérifier que, dans une liste doublement chaînée, le prédécesseur du successeur de chaque nœud est ce même nœud ?