Inférence de type : le compilateur sait inférer (= calculer) les valeurs des paramètres de type lors d'un appel de méthode générique.

```
<T> void addAll(T[] t, Collection<T> c)
```

```
Integer[] ia = new Integer[] { 1, 2 };
List<Number> nl = new ArrayList<Number>();
obj.<Number>addAll(ia, nl);
```

↕

```
Integer[] ia = new Integer[] { 1, 2 };
List<Number> nl = new ArrayList<Number>();
obj.addAll(ia, nl);
```

le compilateur sait calculer T ::= Number

$DT_i \rightarrow T_i$ **extends** $U_{i,1}$ & ... & U_{i,h_i}

Entête : $<DT_1, \ldots, DT_n>$ R p(F_1 f_1, ..., F_m f_m) ($n > 0, m \geq 0$)
Appel : obj.p(e_1, ..., e_m) avec E_j = CT(e_j)

algorithme d'inférence

Résultat : $<Z_1, \ldots, Z_n>$ = le plus spécifique des $<X_1, \ldots, X_n>$
sous les contraintes $E_j <: F_j[T_1|X_1, \ldots, T_n|X_n]$ ($1 \leq j \leq m$)

s'il reste des indices i tq Z_i n'a pas pu être déterminé ($i \in I$)

poursuite du calcul avec les contraintes
- $X_i = Z_i$ ($i \notin I$)
- $\forall i \in I, X_i <: U_{i,h}[T_k|(k \in I ? X_k : Z_k)]$ ($1 \leq h \leq h_i$)
- $R[T_k|(k \in I ? X_k : Z_k)] <: S$

pour déterminer les Z_i manquants

S est déterminé par le contexte d'appel

Entête : `<T> void addAll(T[] t, Collection<T> c)`
Appel : `obj.addAll(ia, nl)`
(F_1, F_2) = (CT(t), CT(c)) = (T[], Collection<T>)
(E_1, E_2) = (CT(ia), CT(nl)) = (Integer[], List<Number>)

algorithme d'inférence

résultat : Z, le plus spécifique des X tels que
Integer[] <: X[]
List<Number> <: Collection<X>

T ::= Z

Généricité


```
<T> void addAll(T[] t, Collection<T> c)
```

```
Integer[] ia = new Integer[] { 1, 2 };
List<Number> nl = new ArrayList<Number>();
obj.addAll(ia, nl);
```

$DT_i \rightarrow T_i$ **extends** $U_{i,1}$ & ... & U_{i,h_i}

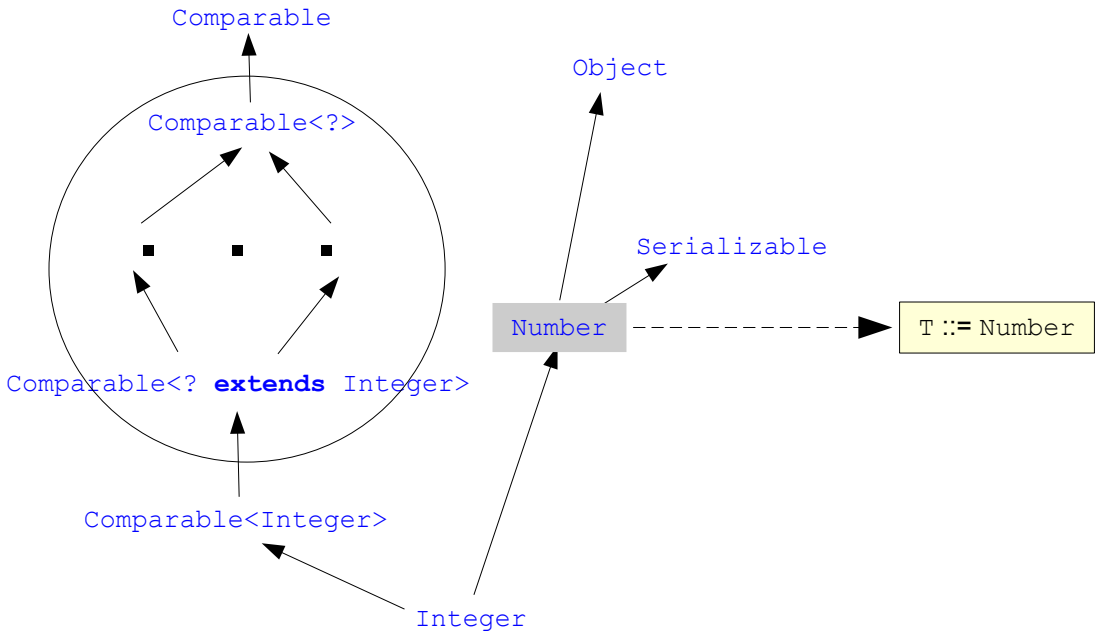
$n = 1, h_1 = 0, DT_1 \rightarrow T_1$
(T_1 est noté T par la suite)

Entête : $<DT_1, \ldots, DT_n>$ R p(F_1 f_1, ..., F_m f_m) ($n > 0, m \geq 0$)
Appel : obj.p(e_1, ..., e_m) avec $E_j = CT(e_j)$

$m = 2$
$(F_1, F_2) = (T[], Collection<T>)$
$(E_1, E_2) = (Integer[], List<Number>)$

Résultat : $<Z_1, \ldots, Z_n>$ = le plus spécifique des $<X_1, \ldots, X_n>$
sous les contraintes $E_j <: F_j[T_1|X_1, \ldots, T_n|X_n]$ ($1 \leq j \leq m$)

Contraintes :
Integer[] <: X[]
List<Number> <: Collection<X>



```
<T extends Number> List<T> emptyList()
List<Integer> il = obj.emptyList();
```

$DT_i \rightarrow T_i$ **extends** $U_{i,1}$ & ... & U_{i,h_i}

$n = 1, h_1 = 1, DT_1 \rightarrow T_1$ **extends** Number
(T_1 est noté T par la suite)

Entête : $<DT_1, \ldots, DT_n>$ R p(F_1 f_1, ..., F_m f_m) ($n > 0, m \geq 0$)
Appel : obj.p(e_1, ..., e_m) avec E_j = CT(e_j)

$m = 0$

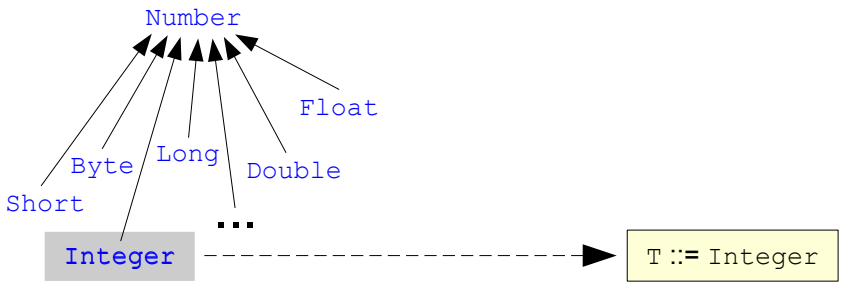
Résultat : $<Z_1, \ldots, Z_n>$ = le plus spécifique des $<X_1, \ldots, X_n>$
sous les contraintes $E_j <: F_j[T_1|X_1, \ldots, T_n|X_n]$ ($1 \leq j \leq m$)

Aucune contrainte :
Z reste indéterminé

poursuite du calcul avec les contraintes
- $X_i = Z_i$ ($i \notin I$)
- $\forall i \in I, X_i <: U_{i,h}[T_k|(k \in I ? X_k : Z_k)]$ ($1 \leq h \leq h_i$)
- $R[T_k|(k \in I ? X_k : Z_k)] <: S$

pour déterminer les Z_i manquants

Nouvelles contraintes :
- X <: Number
- List<X> <: List<Integer>



Généricité


```
<T> List<T> toList(T a, T b)
```

```
List<Object> ol = obj.toList(5, "5");
```

$DT_i \rightarrow T_i$ **extends** $U_{i,1}$ & ... & U_{i,h_i}

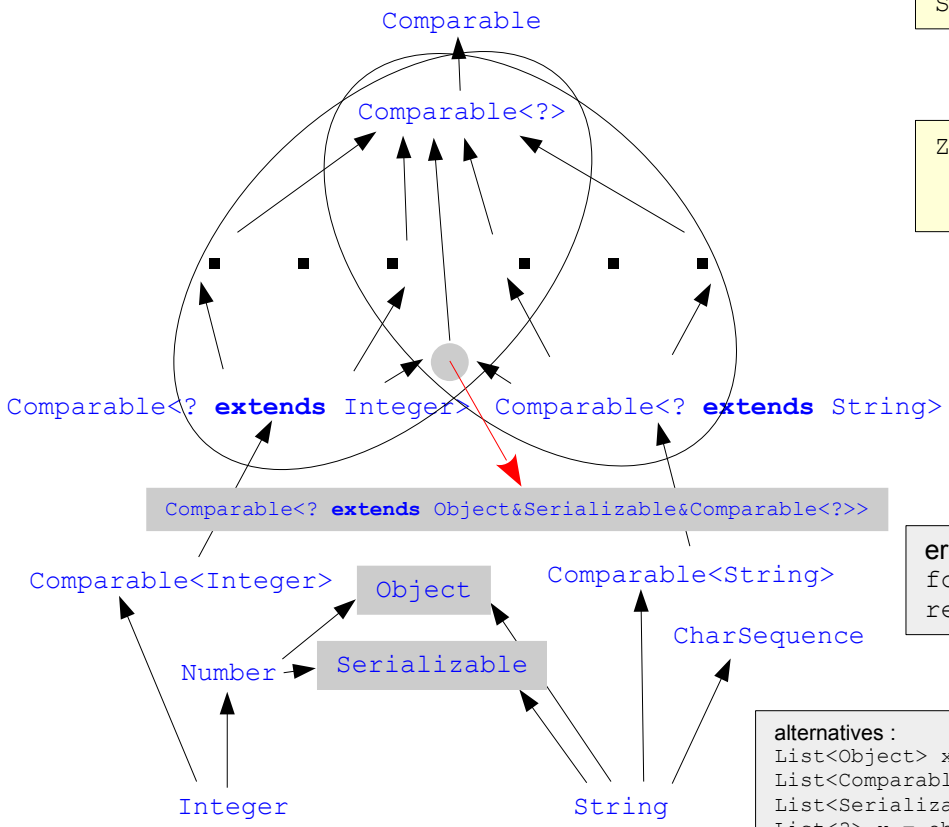
$n = 1, h_1 = 0, DT_1 \rightarrow T_1$
(T_1 est noté T par la suite)

Entête : $<DT_1, \ldots, DT_n> R\ p(F_1\ f_1, \ldots, F_m\ f_m)$ $(n > 0, m \geq 0)$
Appel : $obj.p(e_1, \ldots, e_m)$ avec $E_j = CT(e_j)$

$m = 2$
$(F_1, F_2) = (T, T)$
$(E_1, E_2) = (Integer, String)$

Résultat : $<Z_1, \ldots, Z_n>$ = le plus spécifique des $<X_1, \ldots, X_n>$
sous les contraintes $E_j <: F_j[T_1|X_1, \ldots, T_n|X_n]$ $(1 \leq j \leq m)$

Contraintes :
$Integer <: X$
$String <: X$



$Z =$ Object
& Serializable
& Comparable<? ...>

```
List<Z> <: List<Object>
```

erreur à la compilation :
found : List<Object&Serializable&...>
required : List<Object>

alternatives :

```
List<Object> x = obj.<Object>toList(5, "5");
List<Comparable<?>> x = obj.<Comparable<?>>toList(5, "5");
List<Serializable> x = obj.<Serializable>toList(5, "5");
List<?> x = obj.toList(5, "5");
```

Capture de joker (wildcard capture) :
lors d'un appel de méthode générique, pendant l'inférence de types,
si `E` est de la forme `H<?>` et si `F` est de la forme `G<T>`
alors le joker est remplacé par un nouveau nom de type
de la forme `capture#n` et alors `T` ::= `capture#n`.

```
void permuteHead(List<?> list) {
    Object a = list.get(0);
    Object b = list.get(1);
    list.set(0, b);
    list.set(1, a);
}
```

erreur :
impossible d'ajouter `a` ou `b` dans `list`

```
<T> void permuteHead(List<T> list) {
    T a = list.get(0);
    T b = list.get(1);
    list.set(0, b);
    list.set(1, a);
}
```

correct car `a`, `b`, et `list` sont typés avec `T`

capture de joker
durant l'inférence de types
`F` = `List<T>`
`E` = `List<?>`

```
public void permuteHead(List<?> list) {
    pH(list);
}

private <T> void pH(List<T> list) {
    T a = list.get(0);
    T b = list.get(1);
    list.set(0, b);
    list.set(1, a);
}
```

1. `?` est remplacé momentanément par le nom `capture#1`
2. `Z` = `capture#1`
3. `pH(list)` → `this.<capture#1>pH(list)`