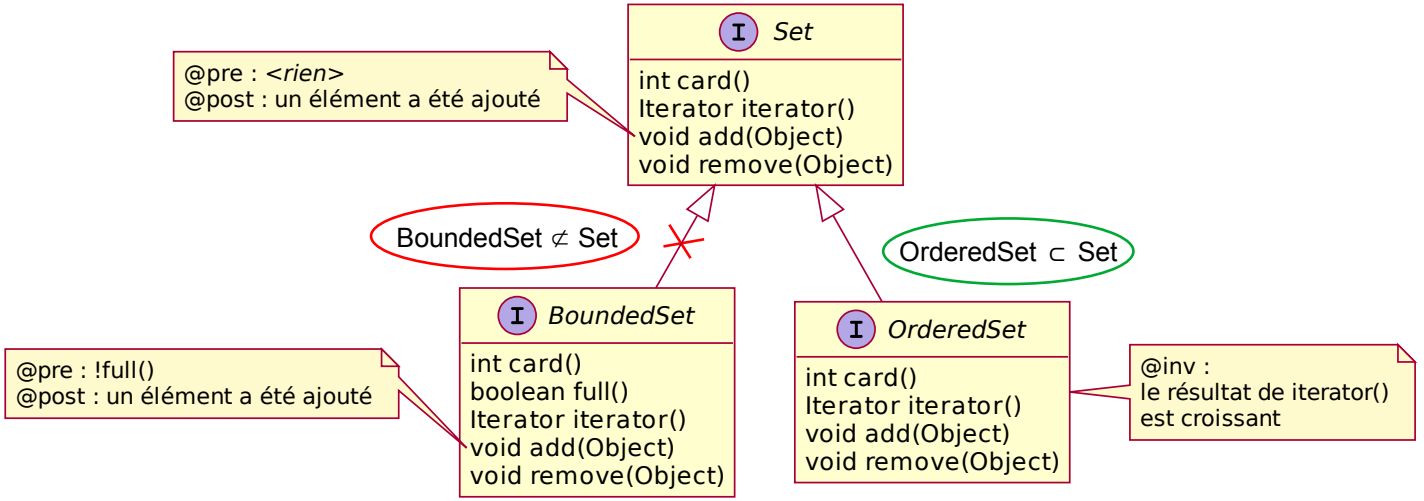
Méthodologie

Principe de substitution de Liskov (PSL) :
pour deux types S et T, si,
dans tout programme P,
on peut utiliser une valeur de S
partout où une valeur de T est attendue,
sans modifier la sémantique de P,
alors S ⊂ T (« *S est sous-type Liskov de T* »).

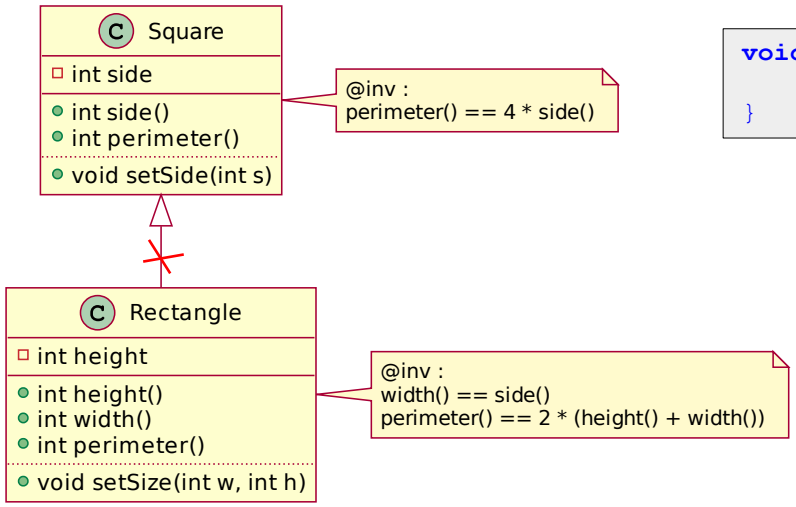


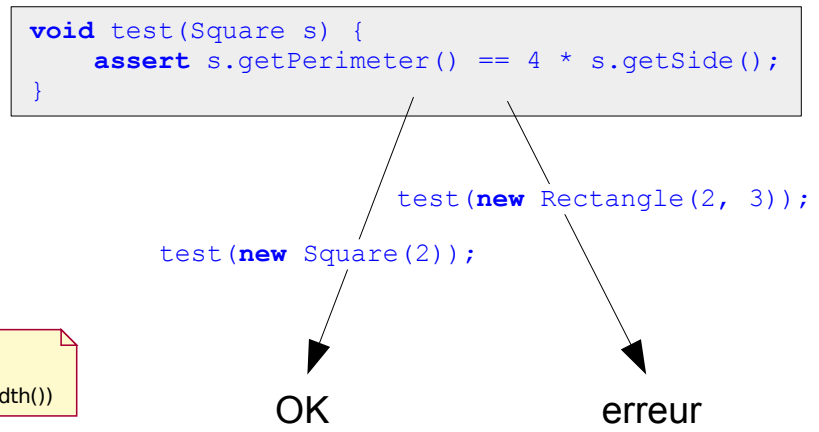
Traduction du PSL en Java : Si S <: T alors
- l'invariant de S peut *restreindre* celui de T
- les contrats des méthodes redéfinies dans S peuvent *raffiner* ceux des méthodes héritées de T

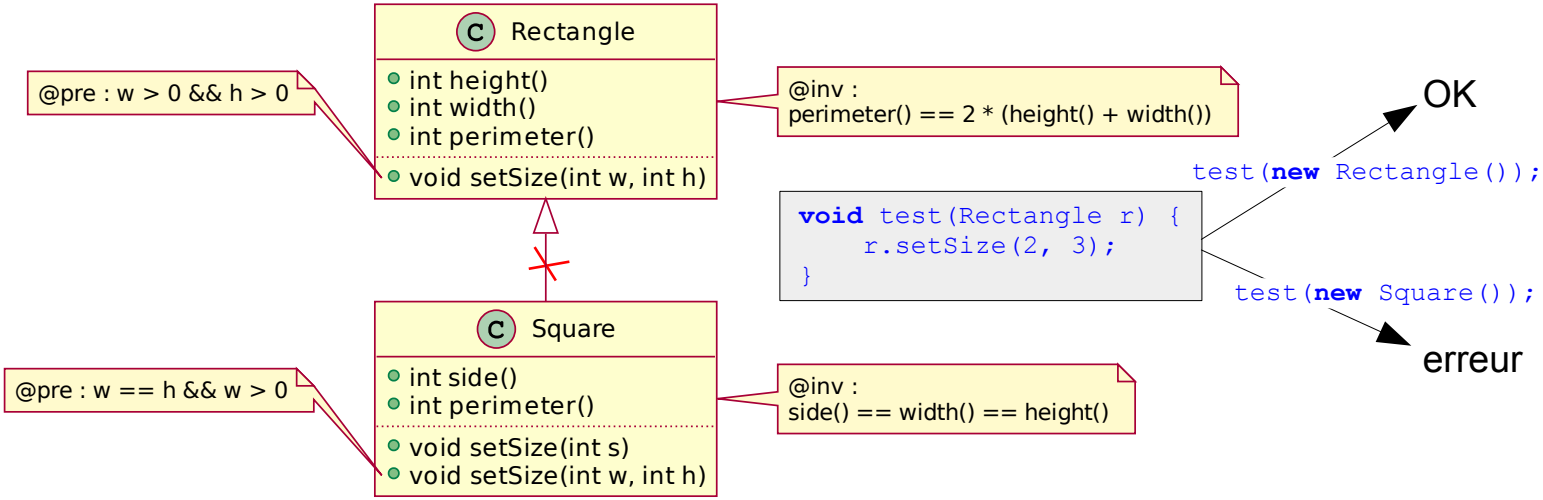
$pre(m_T) \Rightarrow pre(m_S)$
$post(m_S) \Rightarrow post(m_T)$

$inv(S) \Rightarrow inv(T)$

Méthodologie

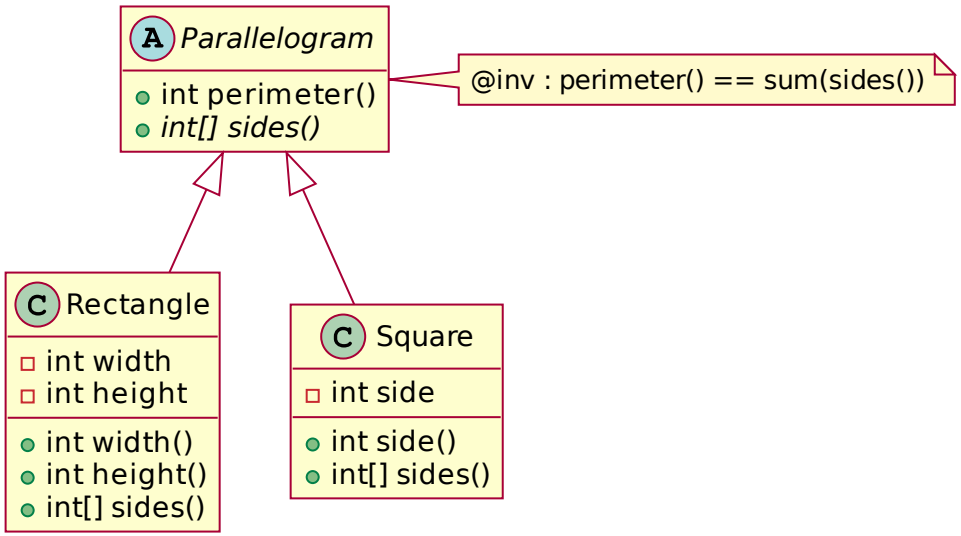




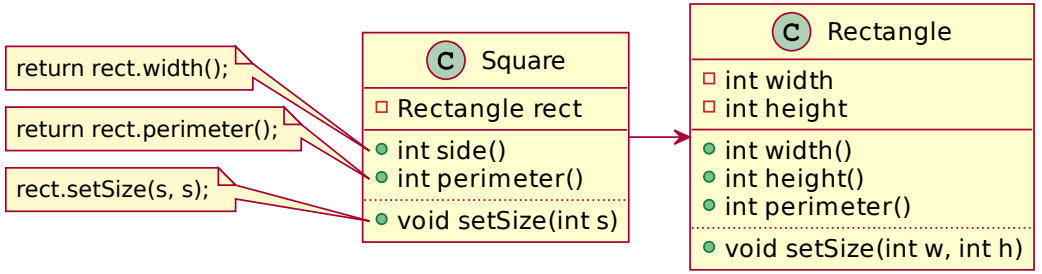


Méthodologie

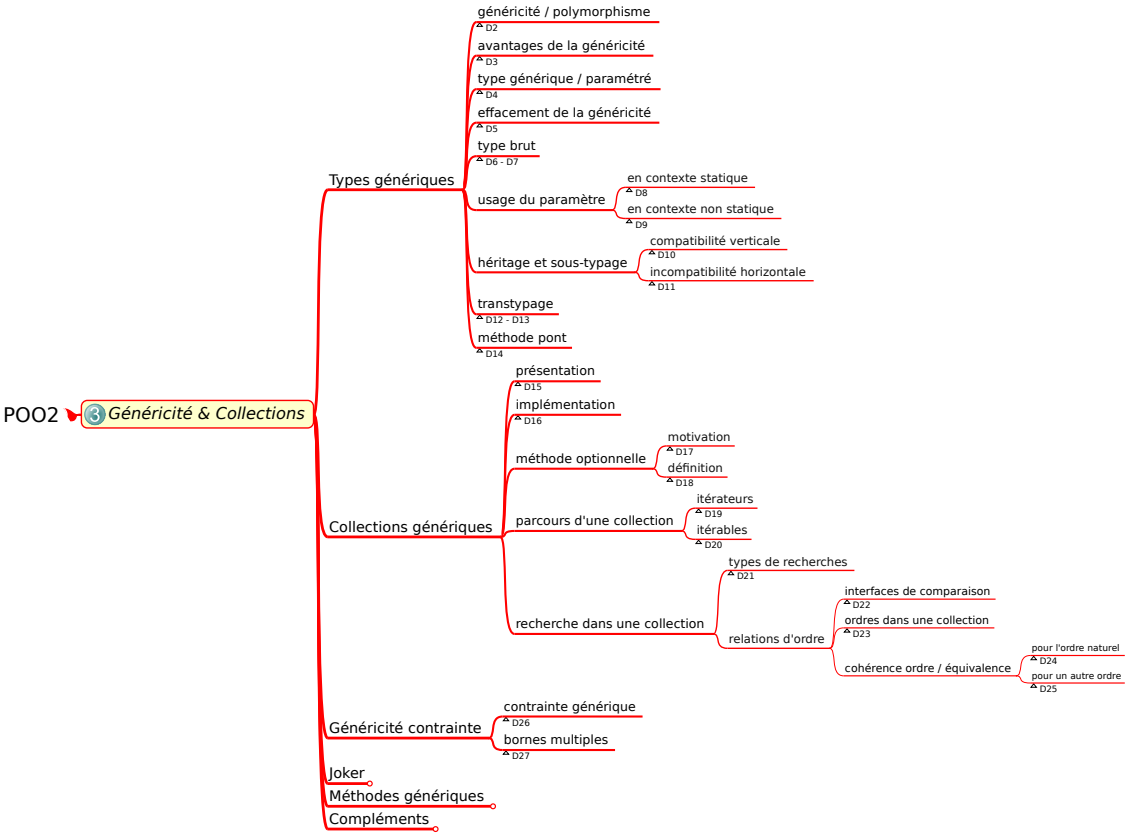
Si héritage impossible : factorisation...

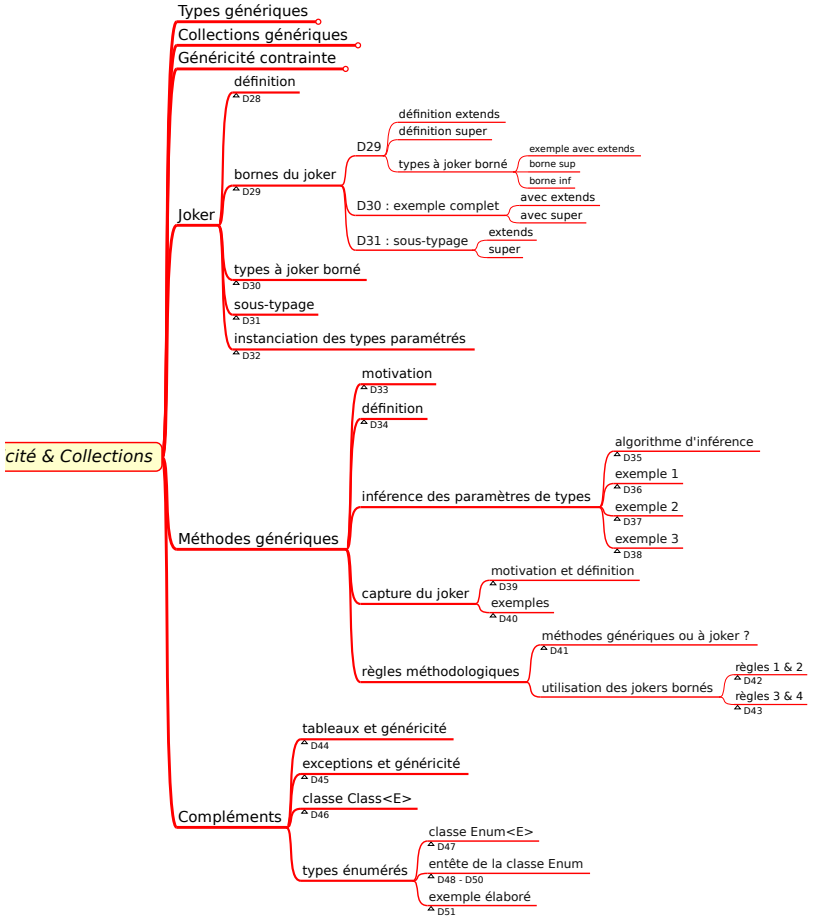


... ou délégation



Généricité (& Collections)





Généricité

Exemple de polymorphisme (polymorphisme de sous-typage)

```
public class Stack {
    public Stack() { ... }
    public Object push(Object item) { ... }
    public Object pop() { ... }
    public Object peek() { ... }
    public boolean empty() { ... }
}
```
version Java 1.4

```
Stack strStack = new Stack();
for (int i = 0; i < n; i++) {
    strStack.push(String.valueOf(i));
}
...
String s = (String) strStack.peek();
```

```
Stack intStack = new Stack();
for (int i = 0; i < n; i++) {
    intStack.push(Integer.valueOf(i));
}
...
Integer n = (Integer) intStack.peek();
```

`intStack` **n'est pas** une pile d'entiers
→ `intStack` **est** une pile d'objets

Exemple de généricité (polymorphisme paramétrique)

```
public class Stack<E> {
    public Stack() { ... }
    public E push(E item) { ... }
    public E pop() { ... }
    public E peek() { ... }
    public boolean empty() { ... }
}
```
version Java 6

```
Stack<String> strStack = new Stack<String>();
for (int i = 0; i < n; i++) {
    strStack.push(String.valueOf(i));
}
...
String s = strStack.peek();
```

```
Stack<Integer> intStack = new Stack<Integer>();
for (int i = 0; i < n; i++) {
    intStack.push(Integer.valueOf(i));
}
...
Integer n = intStack.peek();
```

`intStack` **n'est pas** une pile d'objets
→ `intStack` **est** une pile d'entiers

Généricité

Avantages de la généricité sur le polymorphisme :
- améliore la sûreté du typage
- améliore la lisibilité du code
- facilite le développement

polymorphisme

typage pas assez explicite

```
Stack intStack = new Stack();
intStack.push("aïe !");
...
Integer n = (Integer) intStack.peek();
```

erreur détectée à l'exécution et trop loin de l'origine

cast moins lisible

trop de types

```
public class IntegerStack {
    public IntegerStack() { ... }
    public Integer push(Integer item) { ... }
    public Integer pop() { ... }
    public Integer peek() { ... }
    public boolean empty() { ... }
}

public class StringStack {
    public StringStack() { ... }
    public String push(String item) { ... }
    public String pop() { ... }
    public String peek() { ... }
    public boolean empty() { ... }
}
```

généricité

typage bien explicite

```
Stack<Integer> intStack = new Stack<Integer>();
intStack.push("aïe !");
...
Integer n = intStack.peek();
```

erreur détectée à la compilation au bon endroit

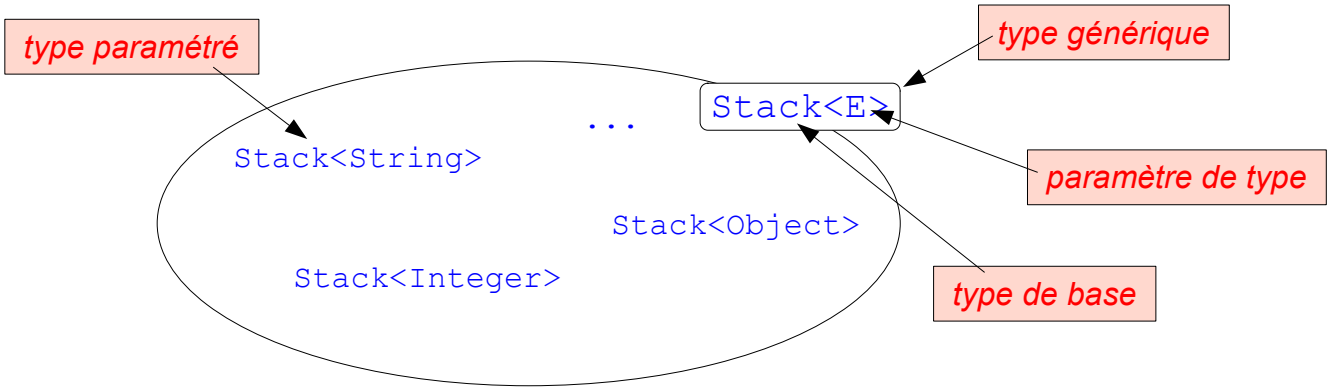
pas de cast plus lisible

un seul (schéma de) type

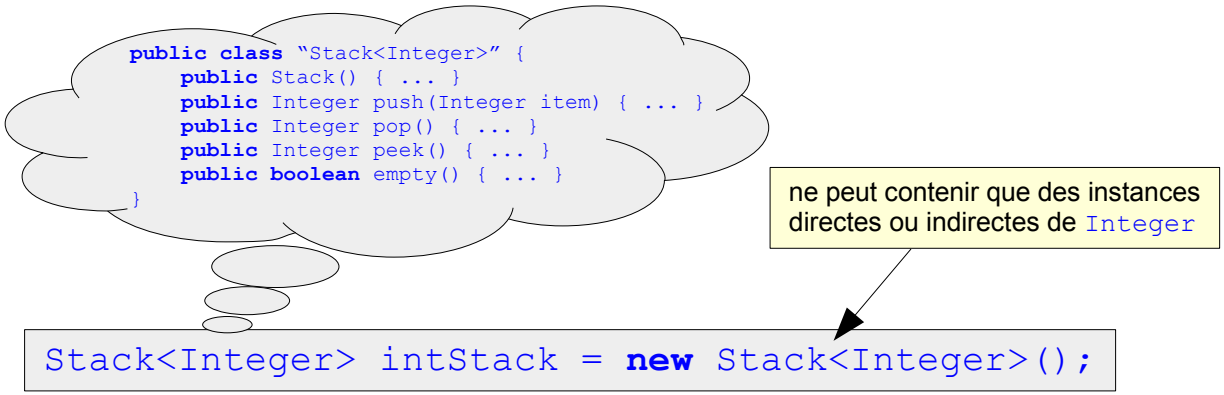
```
public class Stack<E> {
    public Stack() { ... }
    public E push(E item) { ... }
    public E pop() { ... }
    public E peek() { ... }
    public boolean empty() { ... }
}
```

Généricité

Type générique : famille de types,
basée sur une unique définition de type,
paramétrée par une ou plusieurs variables (ou paramètres) de types.



Type paramétré (= instance générique d'un type générique) :
l'un des éléments d'un type générique, obtenu par substitution d'un type référence pour chaque variable de type.



Généricité

Effacement de la généricité : suppression par le compilateur, dans le *bytecode*, de toute information relative à la généricité.

```
public class Stack extends Vector {
    ...
    public Object push(Object item) {
        addElement(item);
        return item;
    }
    ...
}
```

compilation — *version Java 1.4*

```
public class Stack<E> extends Vector<E> {
    ...
    public E push(E item) {
        addElement(item);
        return item;
    }
    ...
}
```

compilation — *version Java 6*

|Stack<E>| = Stack
|E| = Object

```
public class Stack extends java.util.Vector{
...
public java.lang.Object push(java.lang.Object);
  Code:
   0: aload_0
   1: aload_1
   2: invokevirtual  #20;  //Method addElement:(Ljava/lang/Object;)V
   5: aload_1
   6: areturn
...
}
```

```
void m(Stack s) {
    String v = (String) s.pop();
    ...
}
```

compilation — *version Java 1.4*

```
void m(Stack<String> s) {
    String v = s.pop();
    ...
}
```

compilation — *version Java 6*

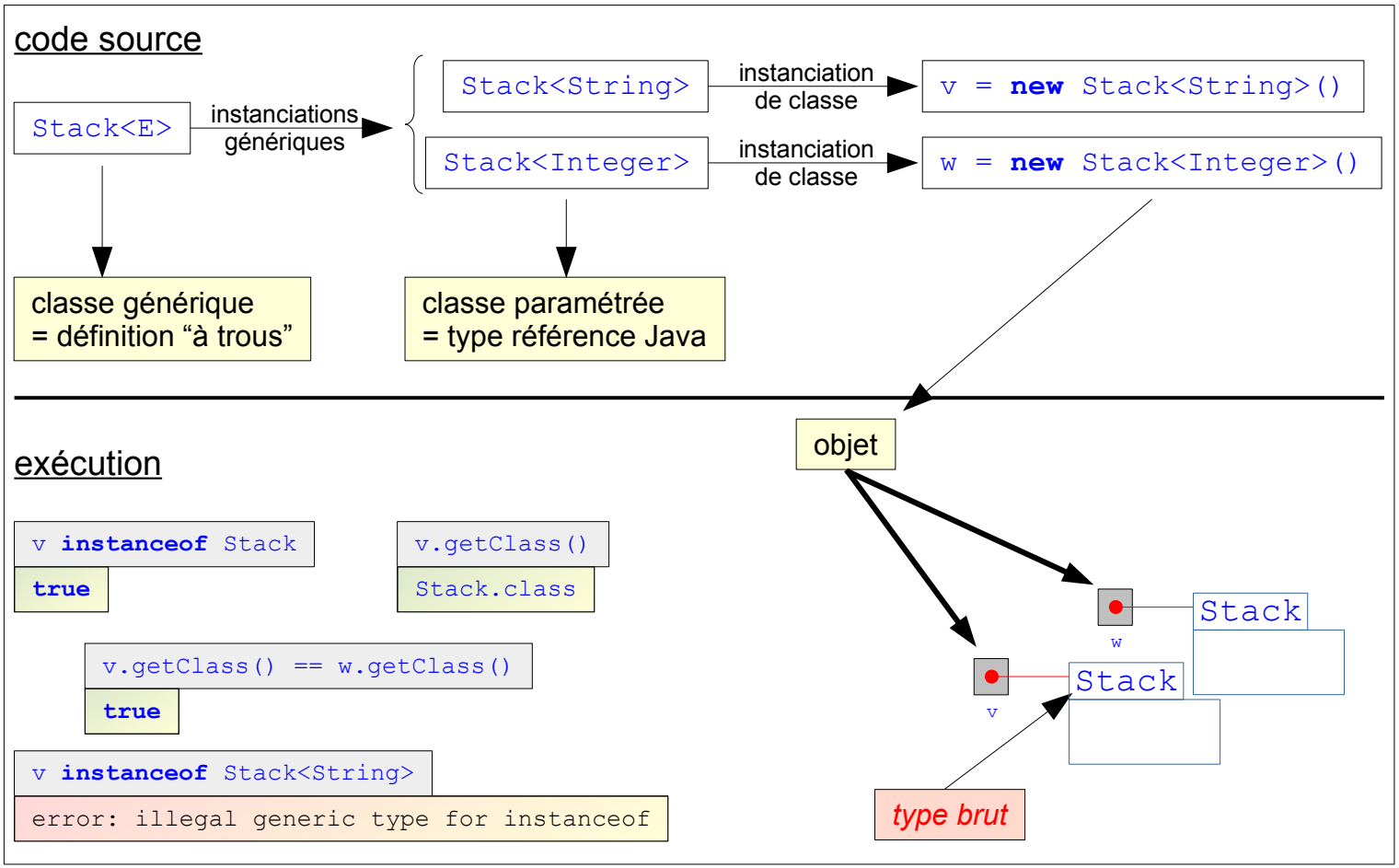
|Stack<String>| = Stack

```
void m(java.util.Stack);
    Code:
      0: aload_1
      1: invokevirtual  #91  // Method java/util/Stack.pop:()Ljava/lang/Object;
      4: checkcast      #57  // class java/lang/String
      7: astore_2
      8: ...
```

Généricité

Type brut

Équivaut au type de base d'un type générique.
C'est le seul type présent dans le *bytecode*, où il remplace à la fois le type générique et tous ses types paramétrés.



Généricité

N'utilisez pas de type brut dans votre code !

```
void test() {
    Stack x = new Stack();
    x.push("abc");
}
```

compilation avec Java 1.4 → OK (pas de généricité)

compilation avec Java 6 →

```
Note: StackTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

ce n'est pas une erreur...

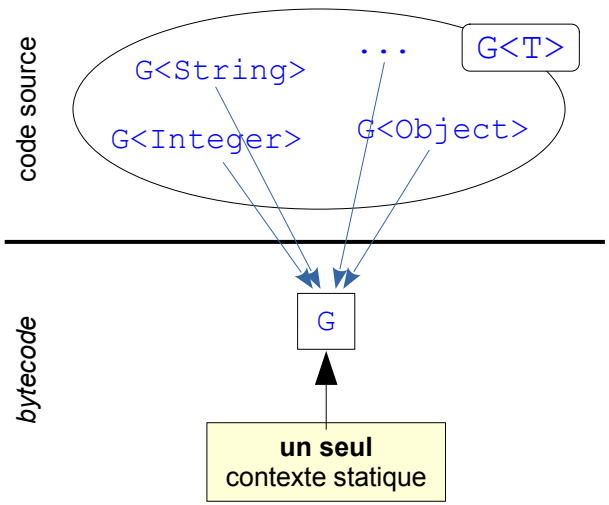
javac -Xlint:unchecked

```
warning: [unchecked] unchecked call to push(E)
         as a member of the raw type Stack
  x.push("abc");
       ^
```

... mais le typage n'est pas sûr
car le compilo ne peut plus vérifier
le type des éléments placés dans x !

Généricité

L'accès aux paramètres de types est impossible dans le contexte statique d'un type générique.




```
class G<T> {
    partie propre aux instances
        attributs d'instance
        constructeurs
        méthodes d'instance
    partie propre à la classe
        attributs statiques
        blocs d'initialisation statiques
        méthodes statiques
}
```

OUI — accès à T

NON

L'accès aux éléments statiques d'un type générique ne peut se faire qu'à travers son type brut.

```
class G<T> {
    public static final int NB = 5;
}
```

G<String>.NB

G<T>.NB

G.NB

Utilisations autorisées du paramètre de type dans le code : déclaration de variable ou type de valeur retournée.

Rappel : on est ici dans un contexte non statique

déclaration préalable du paramètre `T`

déclaration de paramètre formel

type de valeur retournée

déclaration de variable locale

déclaration d'héritage

déclaration d'attribut

```
class G<T> extends H<T> {

    private T x;

    T m(T x) {
        this.x = x;
        T y = x;
        return y;
    }
}
```

... toujours dans un contexte non statique

Utilisations non autorisées du paramètre de type dans le code : expressions nécessitant la connaissance de la valeur du paramètre effectif pendant l'exécution.

```
class G<T> {
    T m() {
        T x = new T();
        return x;
    }
}
```
/!\ *code incorrect*

effacement de la généricité

```
class G {
    Object m() {
        Object x = new [quoi ici ?]();
        return x;
    }
}
```
/!\ *code improbable*

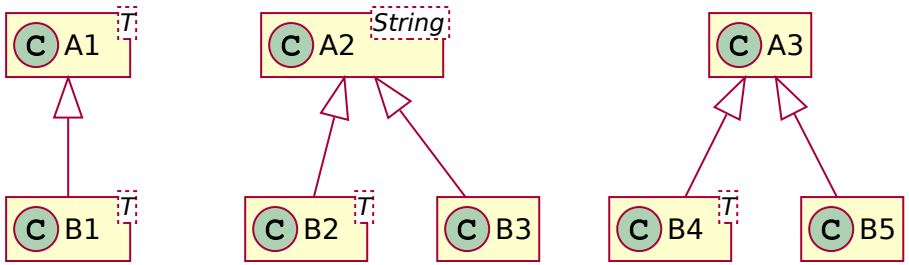
```
T.class
new T[n]
x instanceof T
x instanceof T[]
class A extends T
```

G<Integer> ⇒ [quoi ici ?] = Integer
G<String> ⇒ [quoi ici ?] = String
G<un type> ⇒ [quoi ici ?] = ce type

Généricité

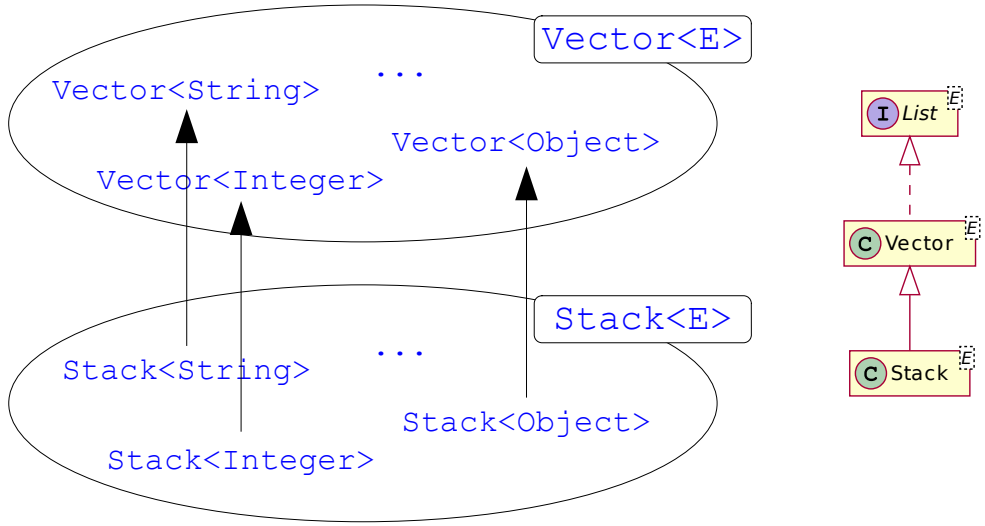
Règles d'héritage

Héritage



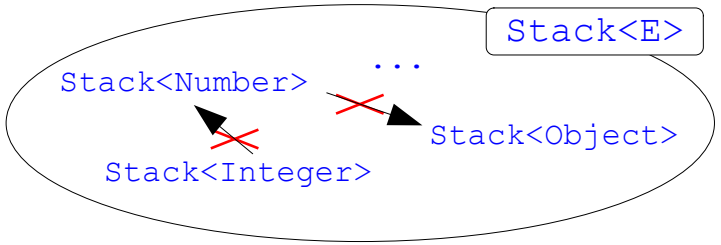
Règle de compatibilité verticale :
soient G<E> et H<E> tels que G <: H
alors ∀ A (type référence), G<A> <: H<A>

Sous-typage



Généricité

Règle d'incompatibilité horizontale :
soit G<E> un type générique
alors ∀ A, ∀ B (types références) ~~G<A> <: G<B>~~



Stack<Integer> ~~<:~~ Stack<Object>

```
Stack<Integer> intStack = ...
Stack<Object> objStack = intStack;
objStack.push("aïe !");
Integer n = intStack.peek();
```

/!\ code incorrect