Généricité

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

L'origine du problème

```
public class ListeString {
  private String[] l = new String[100];
  private int taille;
  public void add(String s) { l[taille++] = s; }
  public String get(int i) { return l[i]; }
}

public class ListeInteger {
  private Integer[] l = new Integer[100];
  private int taille;
  public void add(Integer s) { l[taille++] = s; }
  public Integer get(int i) { return l[i]; }
}
```

Observation ?

L'origine du problème

```java
public class ListeString {
  private String[] l = new String[100];
  private int taille;
  public void add(String s) { l[taille++] = s; }
  public String get(int i) { return l[i]; }
}

public class ListeInteger {
  private Integer[] l = new Integer[100];
  private int taille;
  public void add(Integer s) { l[taille++] = s; }
  public Integer get(int i) { return l[i]; }
}
```

Observation ?

- Même logique, seul le type change

$$\texttt{String} \longrightarrow \texttt{Integer}$$

- Duplication du code

Comment remédier à ce problème ?

Une première solution : la classe `Object`

```
public class ListeObject {
  private Object[] l = new Object[100];
  private int taille;

  public void add(Object s) {
    l[taille++] = s;
  }

  public Object get(int i) {
    return l[i];
  }
}


ListeObject textes = new ListeObject();
textes.add("Bonjour");

String s = (String) textes.get(0); // cast!
```

Avantage

- Un seul code pour tous les types ;

Limites ?

Une première solution : la classe `Object`

```
public class ListeObject {
  private Object[] l = new Object[100];
  private int taille;

  public void add(Object s) {
    l[taille++] = s;
  }

  public Object get(int i) {
    return l[i];
  }
}


ListeObject textes = new ListeObject();
textes.add("Bonjour");

String s = (String) textes.get(0); // cast!
```

Avantage

- Un seul code pour tous les types ;

Limites ?

- **Perte d'inforamtion sur le type** : tout devient un `Object` ;
- Cast obligatoire à chaque récupération ;

Le polymorphisme paramétrique

```
public class Liste<T> {
  private T[] l = new T[100]; // erreur
  private int taille;

  public void add(T s) {
    l[taille++] = s;
  }

  public T get(int i) {
    return l[i];
  }
}
```

```
Liste<String> textes = new Liste<>();
textes.add("Bonjour");
String s = textes.get(0); // sans cast
```

Idée clé

Remplacer le type concret par un **paramètre générique** <T> :

- T est un **type formel** choisi à l'instanciation ;
- le compilateur vérifie la cohérence des types ;
- suppression des casts explicites.

Le polymorphisme paramétrique

```
public class Liste<T> {
  private T[] l = new T[100]; // erreur
  private int taille;

  public void add(T s) {
    l[taille++] = s;
  }

  public T get(int i) {
    return l[i];
  }
}
```

```
Liste<String> textes = new Liste<>();
textes.add("Bonjour");
String s = textes.get(0); // sans cast
```

Idée clé

Remplacer le type concret par un **paramètre générique** <T> :

- T est un **type formel** choisi à l'instanciation ;
- le compilateur vérifie la cohérence des types ;
- suppression des casts explicites.

Attention

CE CODE EST FAUX EN JAVA!!

Petit problème technique

```
public class Liste<T> {
  private Object[] l = new Object[100];
  private int taille;

  public void add(T s) {
    l[taille++] = s;
  }

  public T get(int i) {
    return (T) l[i]; // cast nécessaire
  }
}


Liste<String> textes = new Liste<>();
textes.add("Bonjour");
String s = textes.get(0); // typage à la
compilation
```

Attention

Les tableaux Java et le polymorphisme paramétrique s'entendent mal.

Solution ?

Petit problème technique

```java
public class Liste<T> {
  private Object[] l = new Object[100];
  private int taille;

  public void add(T s) {
    l[taille++] = s;
  }

  public T get(int i) {
    return (T) l[i]; // cast nécessaire
  }
}
```

```java
Liste<String> textes = new Liste<>();
textes.add("Bonjour");
String s = textes.get(0); // typage à la compilation
```

Attention

Les tableaux Java et le polymorphisme paramétrique s'entendent mal.

Solution ?

Impossible de créer directement un tableau de type générique :

`new T[100]` $\rightarrow$ interdit.

On contourne avec un `Object[]` et un cast contrôlé.

Avec plus d'un paramètre de type

```
public class Couple<A, B> {
  private A premier;
  private B second;

  public Couple(A premier, B second) {
    this.premier = premier;
    this.second = second;
  }

  public A getPremier() { return premier; }
  public B getSecond() { return second; }
}


Couple<String, Integer> c =
    new Couple<>("maison", 4);
```

Principe

Une classe peut être paramétrée par plusieurs types :

`<A, B, C, ...>`

Usage typique

- Paires clé/valeur : `Map<K,V>` ;

Borne supérieure

```
public class ListeTriee<T extends Comparable<T>> {
  private List<T> elements = new ArrayList<>();

  public void add(T element) {
    elements.add(element);
    Collections.sort(elements);
  }

  public T get(int i) {
    return elements.get(i);
  }
}


ListeTriee<String> noms = new ListeTriee<>();
noms.add("Alice");
noms.add("Bob");
```

Principe

Une **borne supérieure** limite les types acceptés. Par exemple, T doit implémenter `Comparable<T>`.

On peut combiner plusieurs bornes :

`<T extends X & Y & Z>`

Méthodes génériques

```
public class Utils {

  public static <T extends Comparable<T>>
  T max(T a, T b) {
    return (a.compareTo(b) > 0) ? a : b;
  }

  public static void main(String[] args) {
    System.out.println(max(4, 7));
    System.out.println(max("abc", "def"));
  }
}
```

Principe

Une méthode peut être **générique**, même si la classe ne l'est pas.

La déclaration de la méthode est précédée de celle des types génériques.

À retenir

- Les méthodes génériques définissent leur propre type <T>;
- Elles sont souvent `static`;
- Le typage fort est maintenu sans cast.

Effacement des types (*Type Erasure*)

```java
public class Sums {
  public static int somme(List<Integer> l)
  { ... }
  public static double somme(List<Double> l)
  { ... }
}

// À l'exécution :
List<String> a = new ArrayList<>();
List<Integer> b = new ArrayList<>();

System.out.println(
    a.getClass() == b.getClass()
    ); // true
```

Principe

Les génériques existent uniquement à la **compilation**. Le type paramétré est **effacé** dans le bytecode.

Conséquences

- Pas de surcharge selon les types génériques.
- Pas d'accès au type générique à l'exécution (introspection) ;
- Tous les `List<T>` deviennent des `List`.

Principe de Substitution de Liskov

Principe de substitution de Liskov

Définition

Barbara Liskov (Prix Turing 2008) :

Si `B` est une sous-classe de `A`, alors tout objet de type `B` peut être utilisé partout où un objet de type `A` est attendu.

Formulation formelle

Principe de substitution de Liskov

Définition

Barbara Liskov (Prix Turing 2008) :

Si `B` est une sous-classe de `A`, alors tout objet de type `B` peut être utilisé partout où un objet de type `A` est attendu.

Formulation formelle

Pour tout prédicat P vérifié par les objets de classe A, $P(b)$ doit rester vrai pour tout objet b de classe B :

$$(B \text{ hérite de } A) \implies (P(A) \implies P(B))$$

En pratique

Principe de substitution de Liskov

Définition

Barbara Liskov (Prix Turing 2008) :

Si `B` est une sous-classe de `A`, alors tout objet de type `B` peut être utilisé partout où un objet de type `A` est attendu.

Formulation formelle

Pour tout prédicat P vérifié par les objets de classe A, $P(b)$ doit rester vrai pour tout objet b de classe B :

$$(B \text{ hérite de } A) \implies (P(A) \implies P(B))$$

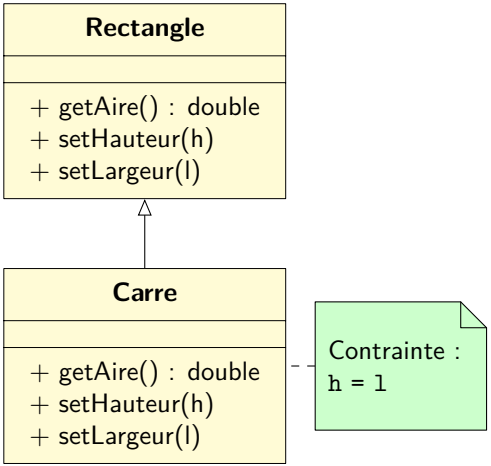
En pratique

- La classe fille ne doit pas restreindre le contrat établit par la classe mère ;
- Les méthodes héritées doivent rester cohérentes.

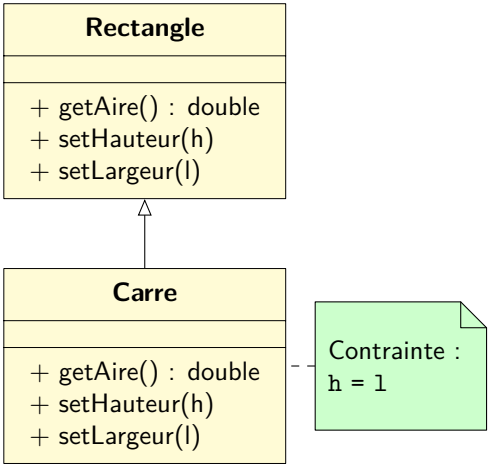
Exemple classique

`Carré` vs `Rectangle` : hériter n'est pas toujours respecter Liskov.

Problème : Carré vs Rectangle



Problème : Carré vs Rectangle



Observation

- Carré hérite logiquement de `Rectangle`, mais impose une contrainte supplémentaire (restreint le contrat):

`hauteur = largeur`

```
void testRectangle(Rectangle rect) {
    rect.setLargeur(5);
    rect.setHauteur(10);

    int aire = rect.getAire();
    assert aire == 50;
}
```

Liskov : pré-conditions et post-conditions

Si `B` hérite de `A`, alors les méthodes de `B` doivent respecter :

- Un sous-type B **ne doit pas renforcer** les préconditions de son parent A.

  Si le parent demande un Animal, la fille ne peut pas renforcer en demandant un Chat.

Liskov : pré-conditions et post-conditions

Si `B` hérite de `A`, alors les méthodes de `B` doivent respecter :

- Un sous-type B **ne doit pas renforcer** les préconditions de son parent A.
  Si le parent demande un Animal, la fille ne peut pas renforcer en demandant un Chat.
- Un sous-type B **ne doit pas affaiblir** les postconditions de son parent A.
  Si le parent fournit un Animal, la fille peut renforcer en fournissant un Chat

Liskov : pré-conditions et post-conditions

Si `B` hérite de `A`, alors les méthodes de `B` doivent respecter :

- Un sous-type B **ne doit pas renforcer** les préconditions de son parent A.
  Si le parent demande un Animal, la fille ne peut pas renforcer en demandant un Chat.
- Un sous-type B **ne doit pas affaiblir** les postconditions de son parent A.
  Si le parent fournit un Animal, la fille peut renforcer en fournissant un Chat
- Les **invariants** doivent être conservés.

Exemple Carré et Rectangle ?

Liskov : pré-conditions et post-conditions

Si `B` hérite de `A`, alors les méthodes de `B` doivent respecter :

- Un sous-type B **ne doit pas renforcer** les préconditions de son parent A.

  Si le parent demande un Animal, la fille ne peut pas renforcer en demandant un Chat.
- Un sous-type B **ne doit pas affaiblir** les postconditions de son parent A.

  Si le parent fournit un Animal, la fille peut renforcer en fournissant un Chat
- Les **invariants** doivent être conservés.

Exemple Carré et Rectangle ?

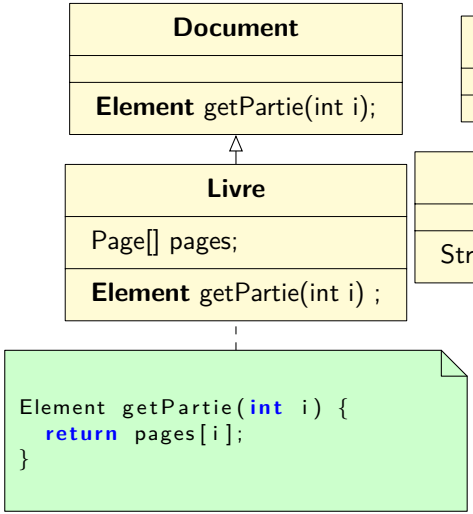
- L'invariant de `Rectangle.setLargeur()` est "La largeur et la hauteur doivent toujours être strictement positives"
- L'invariant de `Carre.setLargeur()` est "La largeur et la hauteur doivent être positives et égales"

Violation de LSP

`Carre` ne respecte pas le contrat de `Rectangle` : un carré ne peut pas remplacer un rectangle sans surprise.

Covariance et Contravariance

Exemple : Quel type de retour est plus adequat ?



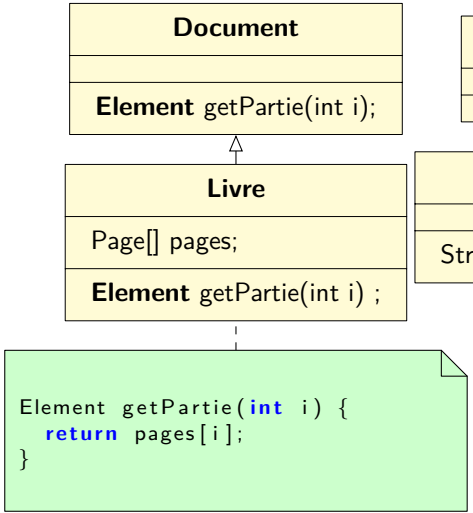
La redéfinition de la méthode getPartie de Document dans Livre est correcte. **Mais ...**

Inconvinient ?

Si j'ecris :

```
Element p = livre.getPartie(1);
```

Exemple : Quel type de retour est plus adequat ?



La redéfinition de la méthode getPartie de Document dans Livre est correcte. **Mais ...**

Inconvinient ?

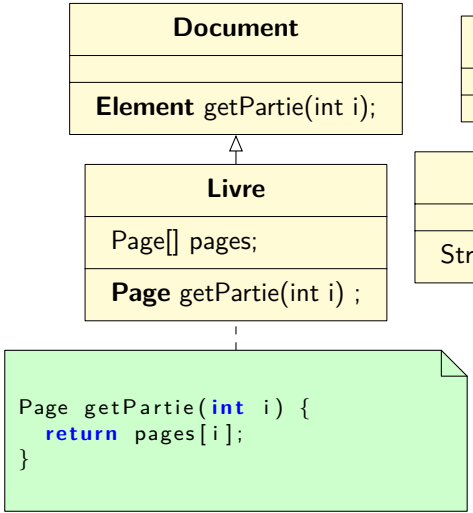
Si j'ecris :

```
Element p = livre.getPartie(1);
```

Je ne peux ecrire :

```
String txt = p.getTexte();
```

Exemple : Quel type de retour ?



Redéfinir la méthode getPartie de Document dans Livre est correcte.
Mais ... Il est plus logique de changer le type de retour à `Page`

Avantage

Je peux ecrire :

```
Page p = livre.getPartie(1);
String txt = p.getTexte();
```

Et **ça respecte Liskov: tout ce que je peux appeler sur un Document est faisable avec un Livre**

La covariance

Définition

On parle de **covariance** lorsqu'un type plus spécifique peut être utilisé à la place d'un type plus général.

B `extends` A $\Rightarrow$ B peut remplacer A

Si :
`Page extends Element`
et
`Livre extends Document`
Alors les types de retour de `getPartie()` dans Livre et Document sont Covariant

La covariance

Définition

On parle de **covariance** lorsqu'un type plus spécifique peut être utilisé à la place d'un type plus général.

B `extends` A $\Rightarrow$ B peut remplacer A

Si :
`Page extends Element`
et
`Livre extends Document`
Alors les types de retour de `getPartie()` dans Livre et Document sont Covariant

Attention

Pour respecter le LSP, la covariance s'applique :

- aux types de retour (depuis Java 1.5).
- pas aux arguments de méthode ($\rightarrow$ contravariance).

Résumé

Covariance = "peut retourner un type plus spécifique".
Cela respecte le principe de substitution de Liskov.

La contravariance

Selon le principe de Liskov

- Si B `extends` A
- Et A possède f(A1 x)
- alors B peut avoir f(B1 x)
- à condition que B1 soit plus général que A1 (Revoir les préconditions de LSP)
- Donc **A1 extends B1**

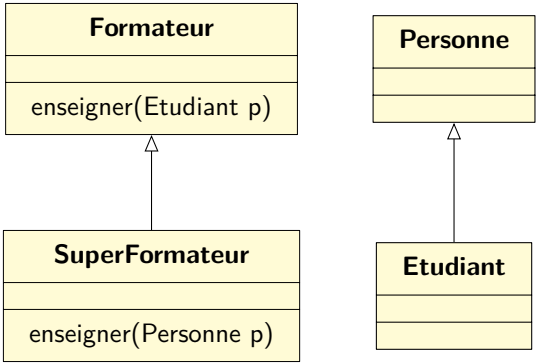
La contravariance

Contravariance = accepter un type plus général là où un type plus spécifique est attendu.

La contravariance

Selon le principe de Liskov

- Si B `extends` A
- Et A possède f(A1 x)
- alors B peut avoir f(B1 x)
- à condition que B1 soit plus général que A1 (Revoir les préconditions de LSP)
- Donc **A1 extends B1**



La contravariance

Contravariance = accepter un type plus général là où un type plus spécifique est attendu.

Attention

Cela ne fonctionne pas en Java.

Problème en Java

```
class Personne { }
class Etudiant extends Personne { }

List<Personne> lp = new ArrayList<>();
List<Etudiant> le = new ArrayList<>();

lp = le; // Possible ou non ?
```

Problème en Java

```
class Personne { }
class Etudiant extends Personne { }

List<Personne> lp = new ArrayList<>();
List<Etudiant> le = new ArrayList<>();

lp = le; // Possible ou non ?
```

Impossible

```
// Si c'était autorisé :
lp.add(new Personne()); // OK pour le
compilateur
Etudiant e = le.get(0); //
ClassCastException !
```

Problème en Java

```
class Personne { }
class Etudiant extends Personne { }

List<Personne> lp = new ArrayList<>();
List<Etudiant> le = new ArrayList<>();

lp = le; // Possible ou non ?
```

Impossible

```
// Si c'était autorisé :
lp.add(new Personne()); // OK pour le
compilateur
Etudiant e = le.get(0); //
ClassCastException !
```

Conséquence

Java refuse la contravariance implicite pour éviter toute incohérence dans les opérations d'écriture. Cela garantit la **sécurité du typage** :

- Pas d'ajout d'objets invalides dans une collection ;
- Pas d'erreur de type à l'exécution ;
- Prévention des `ClassCastException`.

En plus

Les **génériques Java sont invariants**.

Solution: Les Bounded wildcards

Les wildcards : ? extends et ? super

? extends T — Covariant (lecture)

```
List<? extends Personne> liste;

Personne p = liste.get(0); //
lecture autorisée
liste.add(new Etudiant()); //
écriture interdite
```

→ "Producteur" de T (on lit des T ou sous-types de T)

Les wildcards : ? extends et ? super

? extends T — Covariant (lecture)

```
List<? extends Personne> liste;

Personne p = liste.get(0); //
lecture autorisée
liste.add(new Etudiant()); //
écriture interdite
```

→ "Producteur" de T (on lit des T ou sous-types de T)

? super T — Contravariant (écriture)

```
List<? super Etudiant> liste;

liste.add(new Etudiant()); //
écriture autorisée
Etudiant e = liste.get(0); //
lecture interdite
```

→ On peut passer des Etudiants ou des dérivées de Etudiants)

Les wildcards : ? extends et ? super

? extends T — Covariant (lecture)

```
List<? extends Personne> liste;

Personne p = liste.get(0); //
lecture autorisée
liste.add(new Etudiant()); //
écriture interdite
```

→ "Producteur" de T (on lit des T ou sous-types de T)

? super T — Contravariant (écriture)

```
List<? super Etudiant> liste;

liste.add(new Etudiant()); //
écriture autorisée
Etudiant e = liste.get(0); //
lecture interdite
```

→ On peut passer des Etudiants ou des dérivées de Etudiants)

Objectif

Préserver la sécurité du typage tout en maintenant la flexibilité des génériques.

Résumé : covariance, contravariance et invariance

Les trois formes de variance

- **Covariance** : accepte un type plus spécifique. Exemple : `? extends T`
- **Contravariance** : accepte un type plus général. Exemple : `? super T`
- **Invariance** : n'accepte qu'un type exact. Exemple : `List<T>`

Rappel de la règle PECS

Producer Extends, Consumer Super

- `? extends T` → lecture (producteur)
- `? super T` → écriture (consommateur)

Conclusion

Java privilégie l'**invariance par défaut** pour garantir la sécurité du typage.