

Polytech'Paris-Sud - 4^{ème} année

Département Informatique

« Polymorphisme et Généricité en Java »

Frédéric Voisin

Département Informatique

Co-variance vs Contra-variance (rappel !)

Soit `Figure Id (X: Figure) { ... }` dans `Figure`

Dans `Cercle` on peut **redéfinir** `Id` en gardant son en-tête:

```
Figure Cercle::Id (X: Figure ) { ... }      -- OK
```

Lors d'une **redéfinition**, peut-on spécialiser/élargir le type des paramètres ou du résultat ?

```
Id (X: Cercle) return Figure is ...      -- KO
```

```
Id (X: Figure) return Cercle is ...     -- OK
```

```
Id (X: FigureAbstraite) return Figure is ... -- OK
```

```
Id (X: Figure) return FigureAbstraite is ... -- KO
```

Co-variance pour le résultat (ex : `clone()` en Java)

Contra-variance pour les paramètres (pas forcément implémenté dans les langages et vu plutôt comme une surcharge)

Polymorphisme paramétrique à la Ada (rappel)

```
generic  
  type Elem is private;  
  type Table is array(integer range <>) of elem;  
  with function "<="(E1, E2 : Elem) return boolean;
```

↑
paramètres
↓

```
procedure Tri(t: in out Table); - tri générique...
```

Idée ? Pouvoir paramétrer une procédure ou un type par :

- ♦ des types, des procédures, des fonctions
- ♦ décrire les « propriétés » attendues de ces paramètres

« Concrétisation » explicite (Ada) ou implicite (C++) qui associe éléments concrets aux paramètres de généricité.

```
procedure TriIntDecr is -- à la Ada  
  new Tri(Elem => integer, "<=" => ">=", Table => MesTableInt);
```

On n'exécute pas le composant générique mais la forme concrétisée

La généricité à partir de Java 5.0

```
public class Paire<T1, T2> {           // détail dans le prochain transparent
    protected T1 first;
    protected T2 second;
    public Paire(T1 a, T2 b) { first = a; second = b; }
    ...
}
Paire<String, Integer> maPaire = new Paire<String, Integer>("a", 1);
Paire<String, Integer> maPaire = new Paire<>("a", 1);
```

Contraintes possibles sur le paramètre :

Au plus une classe, forcément en tête, et des interfaces;

Si seulement des interfaces, leur ordre importe (voir après) !

```
public class C <T extends T1 & I1 & I2> { ... }
C<A> monObj;           // OK si A est sous-classe de T1 et implémente I1 et I2
```

Une classe générique simple

```
public class Paire<T1, T2> {  
    private T1 first; private T2 second;  
    public Paire(T1 a, T2 b) { first = a; second = b; }  
    public T1 first() { return first; }  
    public T2 second() { return second; }  
    public void setFirst(T1 val) { first = val; }  
    public void setSecond(T2 val) { second = val; }  
    ... etc ...  
}
```

Ici, simple car il n'y a aucune contrainte sur T1 et T2.

En pratique, moins simple à concevoir qu'on pourrait croire, si on veut qu'elle soit *réellement* réutilisable

Autres éléments génériques Java 5.0

Interfaces génériques :

```
interface Comparable<T> { int compareTo(T o); }
public class C implements Comparable<C> { int compareTo(C o) {...} }
interface Collection<E> extends Iterable<E> { ... }
```

Méthodes génériques (ici supposées dans une classe Utilitaires)

```
public static <T> T anyOne(T[] tab)...
// version provisoire de l'en-tête ...
public static <T extends Comparable<T>> T max(Collection<T> c)

String[] maTable = { "Hello", "world" };
Collection<String> l = Arrays.asList("Hello", "world");

String s = Utilitaires.<String>anyOne(maTable); // <String> explicite
String s = Utilitaires.max(l);                // <String> implicite
```

La généricité en Java : Principes de base

- ◆ Renforcer le typage statique et diminuer le besoin en transtypage explicite
- ◆ **Compatibilité avec le code existant et avec la JVM Java4.**

La généricité est traitée **uniquement à la compilation**

- ◆ Pas de classe générique à l'exécution : la **JVM reste non générique.**
- ◆ Le compilateur convertit les génériques en non-génériques
 - Ajout de contrôles statiques
 - Ajout automatique et silencieux de transtypages

```
Paire<String, String> ps; ps.setFirst(e); String s = ps.first();
```

- ◆ À l'exécution **une seule version** de la classe, basée sur le « type limitatif » du paramètre : les paramètres de type sont « **effacés** » !

Différent donc du modèle Ada/C++ :
Dynamiquement, il existe une unique version, non générique

Exemples d'effacement de type

```
class ArrayList<E>
    implements List<E>, ...
{ ... }
```

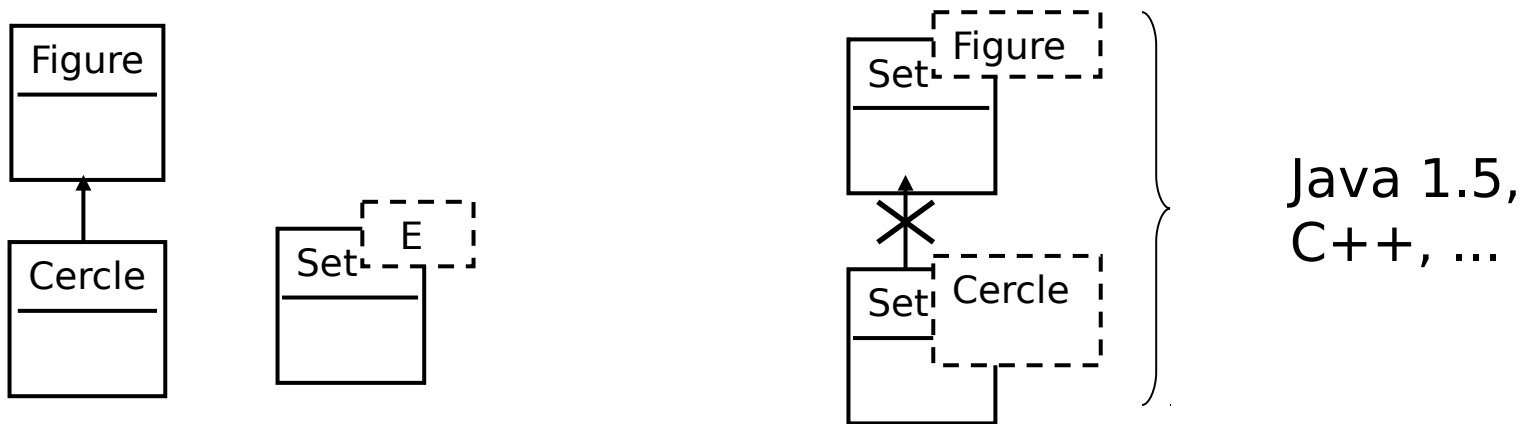
- ◆ **ArrayList<E>** :
la version « effacée » est ArrayList, i.e. ArrayList<Object>
- ◆ **public class Paire<T1, T2>** : T1 et T2 n'ont pas de contraintes.
la version « effacée » est Paire, i.e. Paire<Object, Object>
- ◆ **public class C <T extends C1 & I1 & I2, ... > { ... }**
la version « effacée » est C, i.e. équivalente à C<C1>
- ◆ **public class Intervalle <T extends Comparable> { ... }**
la version « effacée » est Intervalle, i.e. Intervalle<Comparable>

Ada/C++: Paire, ou Paire<T1, T2>, n'existe pas à l'exécution !

Java : Paire est une classe standard. Dynamiquement il existe une unique version non générique !

Paire p; -- compile, mais représente quelle paire ?

Polymorphisme générique et par inclusion



Même problème dans tout langage avec héritage+généricité:
lié à la contravariance sur les paramètres des méthodes.

Polymorphisme générique et par inclusion

```
public class Set<Elem> {           // syntaxe Java pour l'exemple
    public void insert(Elem e){ ... }
};
```

par instantiation du paramètre Elem de Set, on obtient :

```
Set<Figure>           // définit void insert(Figure e) { ... }
Set<Cercle>           // définit void insert(Cercle e) { ... }
```

Avec teste() pourrait-on ajouter un Rectangle à un Set<Cercle> ?

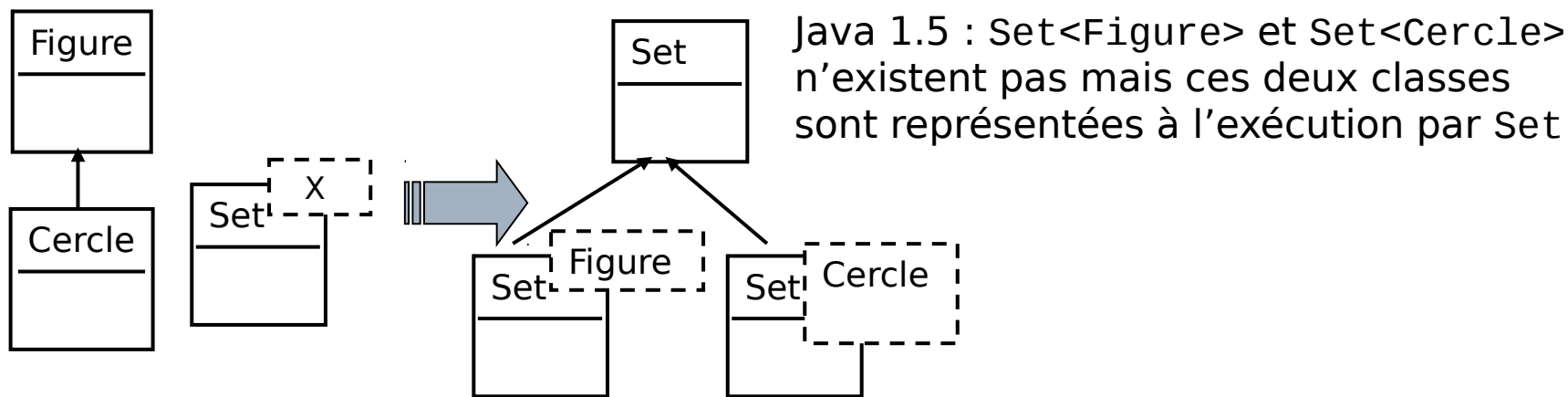
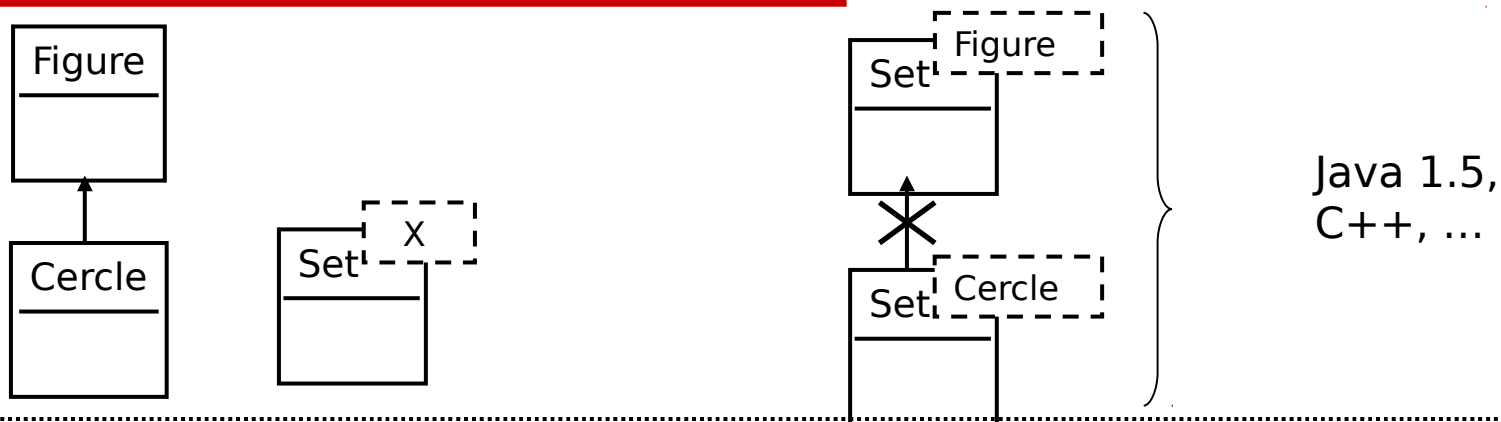
```
void teste(Set<Figure> s) { s.insert(new Rectangle()); }
teste(new Set<Cercle>());
```

Si on autorisait Set<Cercle> à être sous-classe de Set<Figure>, on aurait une incohérence potentielle.

=> **Set<Cercle> ne doit pas être sous-classe de Set<Figure>**

=> **teste(new Set<Cercle>());** *ne doit pas compiler !*

Polymorphisme générique et par inclusion



Le poids du passé ...

Java 5 : compatibilité délicate avec le code existant non générique

```
static ArrayList f(ArrayList arg) { return arg; } // Java 4
ArrayList<Integer> a1 = new ArrayList<Integer>();
ArrayList<Object> a2 = new ArrayList<Object>();
a1 = f(a1); // OK ?
a2 = f(a1); // OK ?
a2 = a1; // OK ?
a1 = a2; // OK ?
a1 = (ArrayList<Integer>) a2; // OK ?
a1 = f(a2); // OK ?
a2.add("Sad world"); Integer i = a1.get(0);
```

Dans la suite on s'en sert pour révéler des mécanismes internes et certaines limitations !

Pas de problème de type si on ne mélange pas Java4- et Java5+
A la compilation, des « warnings » préviennent du problème

Cas particuliers des tableaux (rappel)

```
Figure[] tf = new Figure[2];
Cercle[] tc = new Cercle[2];
Rectangle r = new Rectangle(...);
Figure f;
tf[0] = r;           // OK
tf = tc;            // OK en Java ! tf référence donc un tableau de cercles...
tf[0] = r;         // KO à l'exécution (ArrayStoreException)
tf[0] = f;         // KO possible à l'exécution (ArrayStoreException)
```

Les tableaux doivent mémoriser **dynamiquement** le type d'éléments stockables pour pouvoir lever l'exception.

Contrôle explicite à **l'exécution** ... et risque de levée d'exception !

Était-ce une bonne idée d'autoriser `tf = tc` ?

Les tableaux sont vus comme une construction covariante en Java

Quelques restrictions sur les génériques

- ◆ Pas de classe générique **d'exceptions**.

- ◆ Pas de **type primitif** dans les instantiations :

```
Paire<int, double> p; // KO : serait Paire<Object, Object>  
Paire<Integer, Double> p; // OK
```

Inconvénient allégé par « boxing/unboxing » automatique !

- ◆ Perte de fiabilité des **informations dynamiques** de type

```
ArrayList<Integer> ai = new ArrayList<Integer>();  
ArrayList<String> as = new ArrayList<String>();  
ArrayList a = ai; // on passe par du Java4-  
ai = (ArrayList<Integer>) a; // OK ? }  
as = (ArrayList<String>) a; // OK ? } !
```

instanceof est interdit sur les classes paramétrées : **dynamiquement**, pas de différence entre `Paire<String, String>` et `Paire<Integer, Integer>`

Quelques restrictions (suite)

- ◆ Pas de **tableaux** d'instantiation de génériques

```
Paire<String, Integer>[] t = ... // KO
```

pb avec l'effacement de types et la gestion de `ArrayStoreException`

- ◆ Pas d'**objet** d'un type paramètre

```
public class Paire<T1, T2> {  
    public static T1 f() {  
        return new T1(...); // le new est dynamique: créerait quoi ?  
    }  
}
```

- ◆ Pas de **variable** de type générique pour **méthodes ou attributs static** :

```
public class Singleton<T> {  
    static T theInst; // KO.  
    static T getInst() { ... } // KO.  
}
```

donnerait un drôle de « singleton » après instantiation !

Retour sur un exemple simple...

```
public class Paire<T1, T2> {  
    private T1 first;  
    private T2 second;  
    public Paire(T1 a, T2 b) { first = a; second = b; }  
    public T1 first() { return first; }  
    public T2 second() { return second; }  
    public void setFirst(T1 val) { first = val; }  
    public void setSecond(T2 val) { second = val; }  
}
```

Qu'obtient-on à l'exécution avec l'effacement de type ?

Après effacement ...

Tout se passe comme si, **dans la JVM**, il y avait en fait:

```
public class Paire {  
    private Object first;  
    private Object second;  
    public Paire(Object a, Object b){ first=a; second=b;}  
    public Object first() { return first; }  
    public Object second() { return second; }  
    public void setFirst(Object v) { first = v; }  
    public void setSecond(Object v) { second = v; }  
}
```

sauf que le compilateur ajoute à votre code contrôles et transtypages pour garantir le typage sûr.

Les méthodes de Paire à l'exécution

On vérifie par introspection:

```
Paire p = ...;  
Method[] methodes = p.getClass().getMethods();  
for(Method m: methodes) { System.out.println(m); }
```

```
public java.lang.Object Pair.second()  
public void Pair.setSecond(java.lang.Object)  
public void Pair.setFirst(java.lang.Object)  
public boolean Pair.equals(java.lang.Object)  
public java.lang.Object Pair.second()  
public java.lang.String java.lang.Object.toString()  
public native int java.lang.Object.hashCode()  
...
```

Une utilisation de Paire

```
static int teste(Paire<Integer, Integer> pi) {
    return pi.first() + pi.second();    // OK
}

static void t2(Paire<Integer, Integer> pi) { pi.second(); } // OK

public static void main(String[] argv) {
    Paire<Integer, Integer> pi = new Paire<>(1, 2);
    Paire p = pi ;                    // OK ! Warning à la compilation
    teste(pi);                        // OK !
    pi.setSecond("Coucou");           // KO à la compilation, normal
    p.setSecond("Coucou");           // OK
    teste(pi);                        // à l'exécution ?
    t2(pi);                            // à l'exécution ?
}
```

Rédéfinition dans une sous-classe de Paire

```
public class PaireInteger extends Paire<Integer, Integer> {  
    public void setSecond(Integer i) { // est-ce une redéfinition ?  
        ... super.setSecond(2*i); // OK ?  
    }  
} // PaireInteger n'est pas générique.
```

Avec l'effacement de types, s'agit-il d'une **redéfinition** ou une **surcharge** ?

Dans la JVM, PaireInteger a apparemment **deux** méthodes setSecond :

```
public void setSecond(Object v) - héritée de Paire<Integer, Integer>  
public void setSecond(Integer i) - locale, redéfinit la méthode héritée
```

La seconde est-elle une redéfinition : les signatures diffèrent !

```
PaireInteger pi = new PaireInteger();  
Paire p = pi ;  
p.setSecond(1); // Compile ? Quelle méthode setSecond ?  
p.setSecond("hello"); // mêmes questions
```

Méthodes « bridges » (1)

But : concilier « effacement de type » et comportement attendu du polymorphisme en cas de redéfinition

Le compilateur ajoute dans `PaireInteger` une méthode « bridge » qui **redéfinit** la version de la classe générique et fait le lien avec la méthode **surchargée** ! Le code correspond à :

```
public void setSecond(Object v) {  
    this.setSecond((Integer) v); // Cast risqué ou pas ?  
}
```

Méthode ajoutée automatiquement par le compilateur, non référençable par le programmeur (n'existe que dans le bytecode).

Une conséquence est qu'il est impossible de définir soi-même dans `PaireInteger` une méthode avec cette signature !

Les méthodes de PaireInteger à l'exécution

On vérifie par introspection:

```
public void PaireInteger.setSecond(java.lang.Integer)  
// la méthode ci-dessous a été ajoutée automatiquement  
public void PaireInteger.setSecond(java.lang.Object)  
public void Pair.setFirst(java.lang.Object)  
public java.lang.Object Pair.second()  
public boolean Pair.equals(java.lang.Object)  
public java.lang.Object Pair.first()  
public java.lang.String java.lang.Object.toString()  
public native int java.lang.Object.hashCode()  
...
```

Illustration des méthodes « bridges » (1/2)

```
public static void main(String[] args) {
    PairInt pi = new PairInt(); Pair p = pi;
    pi.setFirst(5); pi.setSecond(12);
    System.out.println(pi.first() + pi.second());

    // pi.setSecond("Coucou");           // Ne compile pas.

    // Exception levée par le « cast » dans la méthode bridge.
    // L'exception est levée avant l'appel à la fonction définie dans PaireInteger
    p.setsecond("Coucou");
}
```

Trace d'exécution:

```
PairInt::setSecond-Integer
```

```
Pair.second
```

```
29
```

```
Exception in thread "main" java.lang.ClassCastException:
```

```
java.lang.String cannot be cast to java.lang.Integer
```

```
    at PairInt.setSecond(PairInt.java:1)           // ligne 1 ??
```

```
    at TestePairInt.main(TestePairInt.java:37)
```

Méthodes « bridges » (2)

Même problème pour gérer la covariance du type de retour !

```
public class A { public A f() { return new A(); } }
public class B extends A {
    public B f() { // Redéfinition? Surcharge ?
        B res1 = this.f(); // Boucle à l'exécution: Normal.
        A res2 = super.f(); // Simple: version A de f appelée
        B res3 = ((A) this).f(); // Compile pas: normal.
        A res4 = ((A) this).f(); // Compile mais exécute quoi ?
    }
}
```

Génération automatique d'une méthode "bridge" dans B

```
public A f() { return this.f();}
```

Deux méthodes `f()` sans paramètre dans la JVM pour B: une à valeurs dans A, l'autre à valeurs dans B.

Illustration des méthodes « bridges » (2/2)

```
public class A {
    public A f() { System.out.println("A::f"); return new A(); }
}

public class B extends A {
    private static int cpt = 1;    // compteur d'appels
    public B f() {
        System.out.println("B::f");
        if (cpt++ == 5) {          // leve une exception au 5eme appel
            throw new UnsupportedOperationException();
        } else {
            A aux = ((A) this.f()); // appel recursif via la méthode bridge
        }
        return new B();
    }
}

public static void main(String [] arg) { B x = new (); x.f(); }
}
```

La trace de l'exemple précédent

Examen des méthodes de B par introspection :

```
public BridgeB BridgeB.f()  
public BridgeA BridgeB.f()
```

Impression et trace d'exception

```
BridgeB::f()  
BridgeB::f()  
BridgeB::f()  
BridgeB::f()  
BridgeB::f()
```

Exception in thread "main" java.lang.UnsupportedOperationException

```
at BridgeB.f(BridgeB.java:12)  
at BridgeB.f(BridgeB.java:1)  
at BridgeB.f(BridgeB.java:14)  
at BridgeB.f(BridgeB.java:1)  
at BridgeB.f(BridgeB.java:14)  
at BridgeB.f(BridgeB.java:1)  
at BridgeB.f(BridgeB.java:14)  
at BridgeB.f(BridgeB.java:1)  
at BridgeB.f(BridgeB.java:14)  
at BridgeB.main(BridgeB.java:21)
```

Écriture de méthodes/classes génériques

- La « non-compatibilité » entre instantiation et héritage complique l'écriture de méthodes réutilisables ...

```
static void dessineTout (Collection<Figure> c) {  
    for(Figure f: c) f.dessine();           // OK  
}
```

```
Collection<Cercle> ac = new ArrayList<>();  
dessineTout(ac);                          // KO !  
dessineTout((Collection<Figure>) ac);    // KO !
```

Pourtant, `dessineTout` ne fait que parcourir `c` sans la modifier.

- Avec les bibliothèques génériques (`Collection`, `Comparable`, etc.), cela reste généralement transparent !
 - Merci aux concepteurs des bibliothèques !
 - Spécification parfois surprenante des paramètres ;-|
- Il faut des mécanismes pour différencier les cas « problématiques »

Tentatives de solutions...

Passage par une procédure générique ?

```
public static <T extends Figure> void dessineTout(Collection<T> c)
{
    for(T f: c) f.dessine();
}
```

artificiel : dessineTout n'a pas de raison d'être générique et cela resterait un palliatif.

Dynamiquement il y aura une seule version de dessineTout donc perte d'information sur ce que contient réellement la collection.

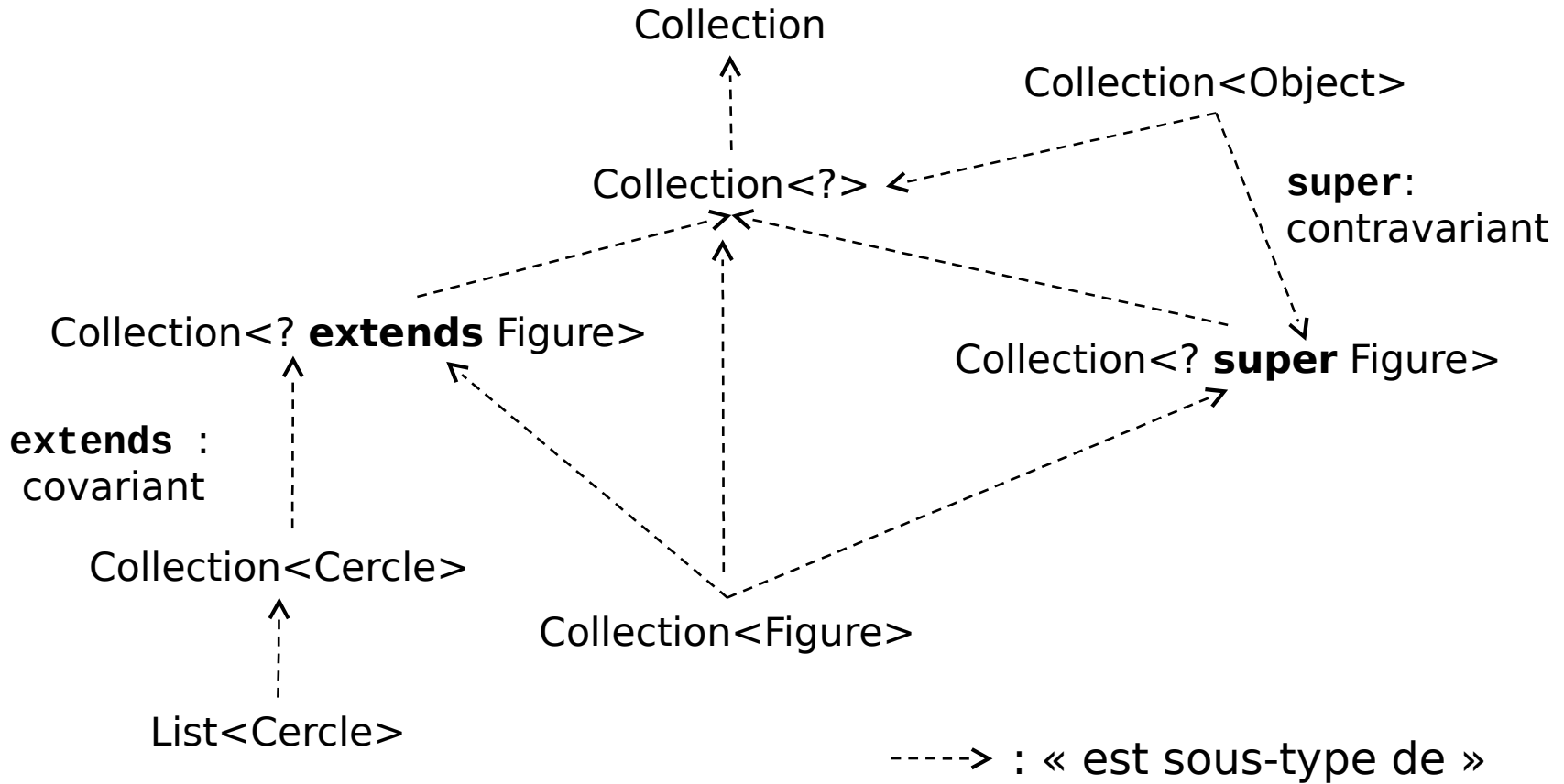
- ☞ Introduction de « **joker** » (« wildcard ») dans la description des paramètres ou du résultat d'une méthode pour mieux faire cohabiter instantiation et héritage !

Lié à nouveau aux notions de covariance/contravariance

Types « Joker »

- ◆ Possibilité d'avoir des types « joker », bornés inférieurement ou supérieurement, plutôt qu'un type précis:
 - < ? > : « n'importe quel type » (un type précis mais inconnu, pas inféré à la compilation comme pour une méthode générique)
 - < ? **extends** Figure> n'importe quel **sous-type** de Figure (inclus)
 - < ? **super** Figure> n'importe quel **super-type** de Figure (inclus)
- ◆ Utilisable pour typer des paramètres de méthodes, des arguments de génériques, des valeurs de retour
- ◆ Différents usages/limitations associés à ces types joker
- ◆ `ArrayList` \neq `ArrayList<Object>` \neq `ArrayList<?>`
les règles de typage sont différentes !

Hiérarchies de sous-typage avec Joker...



Restrictions liées aux jokers

```
class ArrayList<E> ... {
    boolean add(E elt);
    E get(int i) ;
}
```

```
public static void f(ArrayList<?> arg, Object o) {
    arg.add(o);          // KO : ici, pas d'argument acceptable pour add
} // arg : collection d'un type précis mais inconnu.
```

```
public static void g(ArrayList<? extends Figure> arg) {
    Figure f = ...
    arg.add(f);          // de même
    arg.add(new Cercle()); // de même
}
```

```
public static Figure g(ArrayList<? extends Figure> arg) {
    return arg.get(0);   // OK, mais pas typable mieux que Figure
}
```

Restrictions basées sur la distinction entre positions covariantes et contra-variantes...

Restrictions liées aux jokers (2)

`ArrayList<? super Figure>` : *restrictions duales...*

Exemple : Ajoute à une collection les éléments d'une autre collection :

```
static void ajout(ArrayList<? extends Figure> src,  
                  ArrayList<? super Figure> dst) {  
    for(Figure f: src) dst.add(f);    // get implicite sur src  
} // avec ces jokers on peut parcourir src et modifier dst :
```

```
ArrayList<Cercle> ac = new ArrayList<Cercle>();  
ArrayList<Object> os = new ArrayList<Object>();  
ac.add(new Cercle()); ac.add(new Cercle());  
os.add("coucou"); os.add(1);  
ajout(ac, os);
```


Restrictions liées aux jokers

```
public class C<E> {  
    E x;  
    void f(E e) { x = e; }  
    E g() { return x;}  
}
```

Considérons une méthode `m(C<? extends Figure> arg)`

Si une méthode de `C<E>` a un **paramètre** de type `E` on ne peut pas passer un argument spécifique (`Figure`, `Cercle`, ...) pour `arg`.

Si une méthode de `C<E>` renvoie un **résultat** de type `E`, on peut l'utiliser là où on attend une instance de type `Figure` !

Joker « **extends** » : de nature covariante

=> interdit l'usage de `arg` en position contravariante

=> `m` peut maintenant être appelée avec en instanciant `C` avec n'importe quelle sous-classe de `Figure`

```
static void dessineTout (Collection< ? extends Figure> c)  
    { for(Figure f: c) f.dessine(); }
```

```
Collection<Cercle> ac = ... ; Collection<Rectangle> ar = ... ;  
dessineTout(ac);  
dessineTout(ar);
```

Les jokers dans l'API

```
class ArrayList<E> ... {  
    boolean add(E elt);  
    E get(int i) ;  
}
```

Dans `Collection<E>` et ses variantes, certains paramètres de méthodes ne sont pas typés `E` mais `Object` pour contourner la limitation liée aux jokers, si cela ne compromet pas l'intégrité de la collection.

La signature de (la plupart) des méthodes reste compatible avec Java4...

```
public boolean add(E e) { ... }  
public boolean contains(Object o) { ... } // Object, pas E !  
public boolean remove(Object o) { ... }
```

```
public static void t(ArrayList<? extends Figure> arg, Figure f) {  
    arg.remove(f); // OK.  
    arg.add(f);    // KO  
}
```

Question : quelle est la signature de `clone()` dans `ArrayList` ?

Rôle de `<? super T>`

```
interface Comparator<E> {  
    int compare(E o1, E o2);  
} // Java 1.5
```

```
class TreeSet<E> {  
    TreeSet(Comparator c)  
} // Java 1.4 ...
```

On considère `TreeSet<Cercle> s = new TreeSet(monCompareteur);`

Les types possibles pour `monCompareteur` dépendent de la description du paramètre « compareteur » dans la version Java 1.5 de `TreeSet`

Version naïve : `TreeSet(Comparator<E> c) { ... }`

=> imposerait une méthode `int compare(Cercle o1, Cercle o2)`

=> n'accepterait que les compareteurs entre cercles :-)

Version améliorée : `TreeSet(Comparator<? super E> c) { ... }`

accepte en plus des compareteurs plus généraux qui existeraient :

```
int compare(Figure o1, Figure o2);
```

```
voire int compare(Object o1, Object o2);
```

max revisitée (cf transparent 6)

```
static <T extends Comparable<T>> T max(Collection<T> tab) { ... }
```

Trop restrictif : impose à C une méthode `int compareTo(C arg)`

```
public class C implements Comparable<Object> {  
    public int compareTo(Object o) { ... }  
}
```

```
Collection<C> maTab = ... ; C monC = max(maTab); // KO
```

Une meilleure signature pour max :

```
static <T extends Comparable<? super T>> T max(Collection<T> tab)
```

La signature exacte de max dans Collections

```
static <T extends Object & Comparable<? super T> >  
    T max(Collection<? extends T> tab)
```

la version « effacée » est : `Object max(Collection arg)` comme en Java 4

Joker vs méthode paramétrée ?

```
static <T> ArrayList<T> add(ArrayList<T> a1,  
                            ArrayList<? super T> l) { -- l ne sert pas !  
    ArrayList<T> r = new ArrayList<T>(); // T ? viendra d'où ? Compile ?  
    for(T e: a1) r.add(e); // Compile ?? A l'exécution ??  
    return r;  
}
```

```
static <T> ArrayList<T> add2(ArrayList<? extends T> a1,  
                             ArrayList<T> l){  
    ArrayList<T> r = new ArrayList<T>(); // T ? viendra d'où ? Compile ?  
    for(T e: a1) r.add(e); // Compile ?  
    return r;  
}
```

```
ArrayList<Figure> af = new ArrayList<>();  
ArrayList<object> ao = new ArrayList<>();  
af.add(new Cercle()); af.add(new Rectangle());  
ao.add("coucou"); ao.add(1);
```

Joker vs méthode paramétrée (suite)

```
ArrayList<Figure> af = new ArrayList<>();
```

```
ArrayList<object> ao = new ArrayList<>();
```

Qu'est-ce qui est correct ? Vérifiez les appels à add et add2 et les affectations

```
ArrayList<Figure> s1 = add(af, ao); // OK ou KO ?
```

```
ArrayList<Object> s2 = add(af, ao); // OK ou KO ?
```

```
ArrayList<Figure> s3 = add2(af, ao); // OK ou KO ?
```

```
ArrayList<Object> s4 = add2(af, ao); // OK ou KO ?
```

Pour résumer

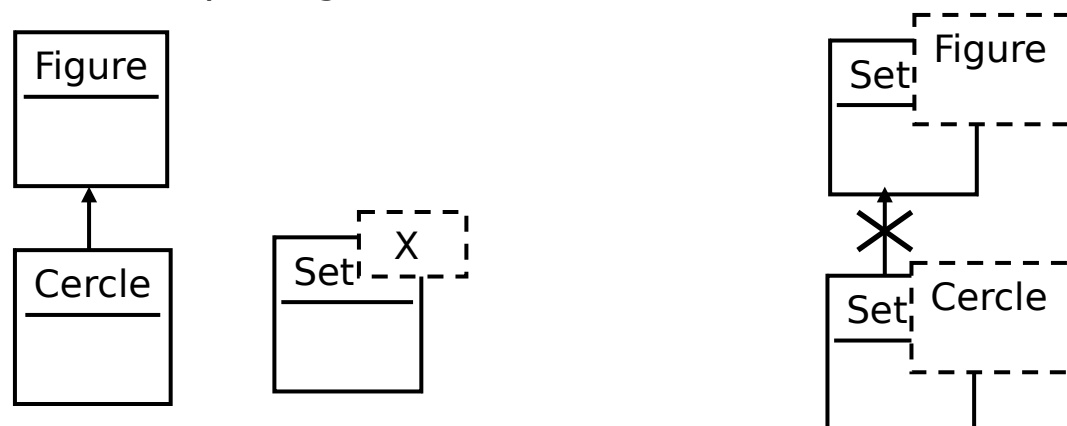
- ◆ Le mélange avec du code 4.0 réintroduit les risques d'exceptions à l'exécution (mais « warnings » à la compilation)
- ◆ Les bibliothèques sont bien conçues et utilisables facilement parfois au prix de la perte de précision du typage des paramètres.
Ex : `Collection<E> : contains(Object e)` plutôt que `contains(E e)`
- ◆ Méthodes « bridges » nécessaires pour avoir le « comportement attendu » du polymorphisme

Concevoir une classe générique réutilisable ?

- ◆ Compréhension des positions covariantes/contravariantes
- ◆ Nécessite de bien maîtriser les types « Joker »
- ◆ Méthode générique ? Classe générique ?

Que manque-t-il encore ?

- ◆ Passage de méthodes en paramètre (« Lambda expressions » Java 1.8)
- ◆ Plus de flexibilité pour gérer co- et contra-variance à *la Scala*.



- ◆ Déclaration de la variance au niveau de la classe, globalement, plutôt qu'au niveau des méthodes.
- ◆ On décrit l'usage attendu de la classe et on en déduit les restrictions
`class Set<X+>` OU `class Set<X->` OU `class Set<X>`