

POO-IG

# Programmation Orientée Objet et Interfaces Graphiques

---

Cristina Sirangelo  
IRIF, Université Paris Cité  
[cristina@irif.fr](mailto:cristina@irif.fr)

# Généricité et collections

---

Exemples et matériel empruntés :

- \* Transparents de cours de H.Fauconnier
- \* Core Java - C.Horstmann - Prentice Hall Ed.

# Généricité : principes

- Paramétrer une classe ou une méthode par un type:
  - une pile d'elements de type T

```
public class Pile<T> {  
    ...  
    void empiler(T) {...}  
    T depiler () {...}  
}
```

un "type variable" (aussi dit "paramètre de type") est déclaré à coté du nom de classe

une fois déclaré, un type variable peut être utilisé dans la classe comme tout autre type

- La généricité en Java est un mécanisme "statique" assez complexe
- présente depuis Java 1.5
- la généricité existe dans d'autres langages (exemple C++ et Ada) (mais de façon différente)

# Exemple: File générique

---

```
class Cellule<E> {
    private E element;
    private Cellule<E> suivant;

    public Cellule(E val) {
        this.element = val;
    }
    public Cellule(E val, Cellule<E> suivant){
        this.element = val; this.suivant = suivant;
    }
    public E getElement () { return element; }
    public void setElement (E v) {
        element = v;
    }
    public Cellule<E> getSuivant () { return suivant; }
    public void setSuivant (Cellule<E> s){
        this.suivant=s;
    }
}
```

# Exemple: File générique (suite)

---

```
public class File<E> {
    protected Cellule<E> tete;
    protected Cellule<E> fin;
    private int taille = 0;
    public boolean estVide(){
        return taille == 0;
    }
    public void enfiler(E item){
        Cellule<E> c = new Cellule<E>(item);
        if (estVide())
            tete=fin=c;
        else{
            fin.setSuivant(c);
            fin = c;
        }
        taille++;
    } //...
```

# Exemple: File générique (suite)

---

```
public E defiler(){
    if (estVide())
        return null;
    Cellule<E> tmp = tete;
    tete = tete.getSuivant();
    taille--;
    return tmp.getElement();
}
public int getTaille(){
    return taille;
}
} //File
```

# Classe générique : invocation

- Pour utiliser une classe générique il faut l' "instancier"
  - lui fournir un type : la valeur de la variable  $T$
- On appelle l'utilisation d'une classe générique une **invocation**
  - **similaire à l'invocation de méthode : on passe en paramètre les types**

```
File<Integer> fi = new File<Integer>();  
// ou new File<>(), inference de Type
```

- On peut avoir plusieurs instances différentes de la même classe générique dans le même programme

```
File<String> fs = new File<String>();  
File<Object> fobj = new File<Object>();
```

- Remarque : une classe générique, par ex. Cellule, peut être invoquée avec un type variable

```
class File<E> {  
    protected Cellule<E> tete;  
    ...  
}
```

# Classe générique : usage

---

- Une fois instanciée, une classe générique peut être utilisée **comme si** le type `T` était remplacé par sa valeur
- I.e. `File<String>` a pour méthodes :

```
boolean estVide()  
void enfiler(String item)  
String defiler()
```



# Classe générique : usage

- **Conceptuellement** une classe générique de paramètre T définit donc une **famille de classes**, une pour chaque valeur possible du type T
  - (évidemment les classes génériques ne sont pas implémentées de cette façon...)
- Exemple

```
File<Integer> fi = new File<Integer>();
File<String> fs = new File<String>();
String[] st={"zéro","un","deux",
            "trois","quatre","cinq"};
for(int i=0; i < st.length; i++){
    fs.enfiler(st[i]);
    fi.enfiler(i);
}
```

# Contrôle de type

---

- Refusés à la compilation :
  - ~~fs.enfiler(4);~~
  - ~~String s = fi.defiler();~~
  
- Auto-boxing et auto-unboxing pour les classes wrapper des types primitifs :
  - `fi.enfiler (new Integer(23) )` peut être abrégé comme `fi.enfiler (23)`
  - `int i = fi.defiler().intValue()` peut être abrégé comme `int i = fi.defiler()`

# Inference de type

- Le compilateur peut inférer la valeur du type variable. Appelé opérateur "diamond"

```
File<Integer> fi = new File<>();
```

<Integer> inféré du type de fi

- Le "diamond" est autorisé également à la création d'une classe anonyme (uniquement depuis Java 9) :

```
ArrayList<String> passwords = new ArrayList<>() {  
    public String get(int n) {  
        return super.get(n).replaceAll(".", "*");  
    }  
};
```

<String> inféré du type de passwords

# Types variables

---

- Plusieurs types variables possibles : `NomClasse<V,K>`
- Noms arbitraires pour ces variables, mais la Java API utilise la convention :
  - E pour les elements d'une collection
  - K, V pour le clefs et valeurs d'une collection indexée
  - T, U, S pour les types arbitraires

# Classes génériques et collections

---

- Les classes "collections" de la bibliothèque Java (`java.util`) modélisent des structures de données (listes, files, ensembles...)
- Sont aujourd'hui génériques
- Mais les variantes non-génériques (même nom, sans paramètre de type) sont encore supportées pour des questions de compatibilité avec le code précédent Java 1.5
  - les éléments sont des `Object`

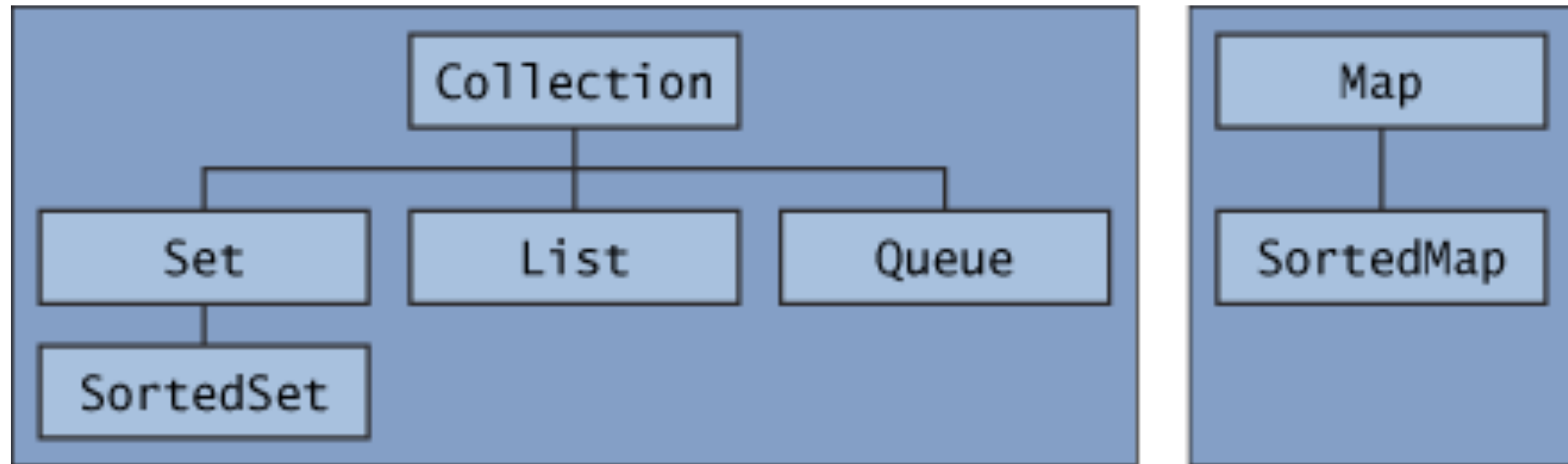
# Collections

---

- Dans la bibliothèque Java on retrouve
  - interfaces collection
  - implémentations
  - algorithmes

# Collections

## □ Interfaces:



Il s'agit interfaces génériques

- Collection<E>: base de la hiérarchie
  - Set<E>: ensemble d'éléments de type E (sans duplication)
    - SortedSet<E>: ensembles ordonnés
  - List<E>: suite d'éléments de type E (avec duplication)
  - Queue<E>: file d'éléments de type E
- Map<K,V>: association clés-valeurs (clés de type K, valeurs de type V)
- SortedMap<K,V> avec un ordre sur les clefs

# Interface Collection

```
public interface Collection<E> extends Iterable<E> {
    // operations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           //optionnel
    boolean remove(Object element);   //optionnel
    Iterator<E> iterator();           //herité de l'interface Iterable

    // operations des collections
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optionnel
    boolean removeAll(Collection<?> c);        //optionnel
    boolean retainAll(Collection<?> c);        //optionnel
    void clear();                               //optionnel

    // Array
    Object[] toArray();
    <T> T[] toArray(T[] a);
    default <T> T[] toArray (IntFunction<T[]> generator); //depuis Java 11

    ...
}
```



# Interface Collection

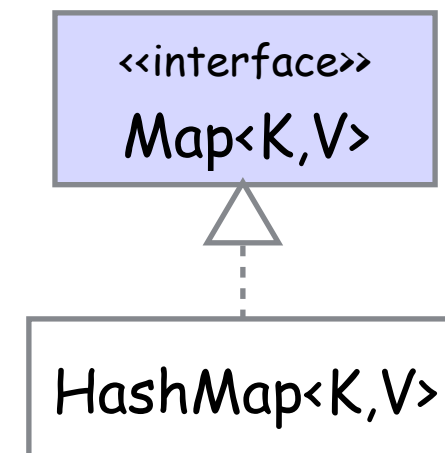
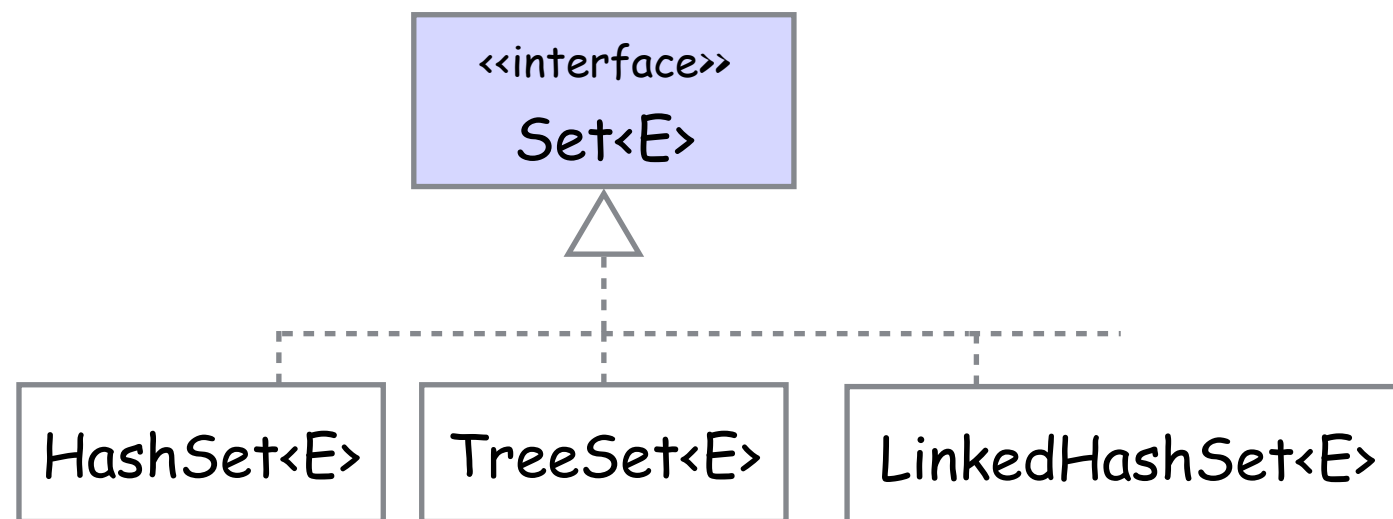
- `extends Iterable<E>`
  - Contient la méthode `Iterator<E> iterator()`
  - On peut parcourir les éléments par boucle « foreach »:
    - Exemple si `c` est de type `Collection<String>`

```
for (String s : c)
    System.out.println(s);
```

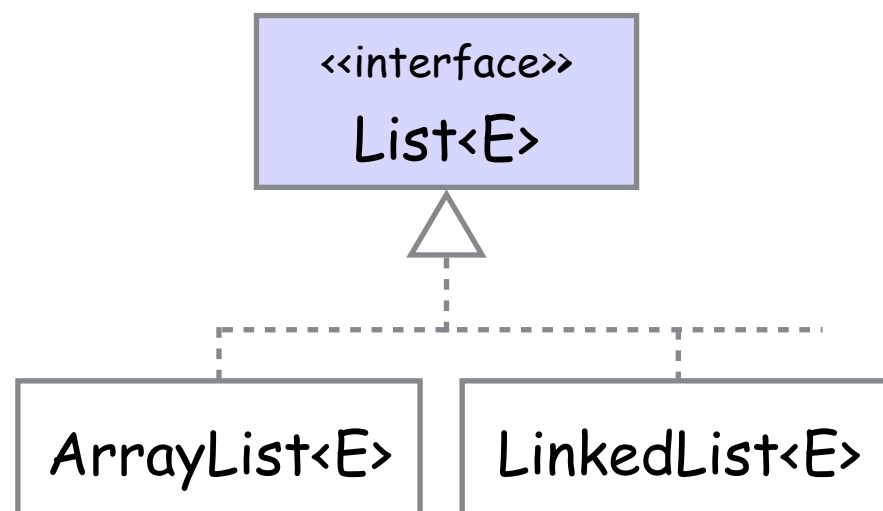
- Ou avec un `Iterator`:

```
for (Iterator<String> it = c.iterator(); it.hasNext();) {
    String s = it.next();
    ...
}
```

# Quelques classes qui implémentent les interfaces



(étend `AbstractMap<K,V>`  
qui implemente `Map<K,V>`)



(étendent `AbstractList<E>`  
qui implemente `List<E>`)

# Collections et contrôle de type

- Les variants non-génériques `Set`, `List`, `LinkedList` ne devraient pas être utilisées
- On veut en général garantir l'intégrité des collections (tous les éléments du même type)
- **Inconvenient** des variants non-génériques : le contrôle de type est à faire "à la main"

```
//Pour avoir une liste chaînée de chaînes de caractères
List l = new LinkedList(); // une liste de Object
//on s'astreint à n'insérer que des chaînes
l.add("abc"); l.add("bcd");
//mais rien n'empêche l.add(5);
//conversion nécessaire pour récupérer les chaînes
String s = (String) l.get(0);
```

# Collections et contrôle de type (suite)

- **Avantages des variantes génériques** : contrôle de type et casting automatique (par le compilateur)


```
//Pour avoir une liste chaînée de chaînes de caractères
List<String> l = new LinkedList<String>();
//on ne peut insérer que des chaînes
l.add("abc"); l.add("bcd");
//le compilateur refuse l.add(5)
//conversion automatique quand on récupère les chaînes
String s = l.get(0);
```

- => **Eviter d'utiliser les variants non-génériques**, sauf quand on s'interface à de l'ancien code (voir plus loin)

# Méthodes génériques

- On peut également définir des méthodes génériques (i.e. avec leur propres paramètres de type)
  - E.g. une méthode qui peut recevoir un tableau de type arbitraire et renvoie un élément du même type

```
static <T> T pick(T[] tab) {...}
```



Déclaration du type variable après les modificateurs et avant le type de retour

- Elles peuvent appartenir à des classes normales, ainsi qu'à des classes génériques

# Invocation de méthodes génériques

```
public class Utilite {  
    public static <T> T[] permuter (T[] tab){...}  
    ...  
}
```

- On peut explicitement préciser le type:

```
String[] tab ={"Bonjour", "Tout", "le monde"};  
String[] s = Utilite.<String>permuter (tab);
```

- Mais le compilateur peut, lui-même, trouver le type le plus spécifique:

```
String [] tab ={"Bonjour", "Tout", "le monde"};  
String[] s = Utilite.permuter (tab);
```

(infère du type de tab que T est String)

# Types bornés

- Un type variable <T> représente tout type possible (sauf les types primitifs)
- À la déclaration de T on peut ajouter des "bornes" pour limiter les types acceptés comme valeur de T

```
public class Figure {...}
```

- `public class A<T extends Figure> {...}`  
T borné par la classe Figure (T doit être sous-classe de Figure ou Figure)
- `public class B<T extends Serializable> {...}`  
T borné par l'interface Serializable (T doit implémenter Serializable) :
- `public class C<T extends Serializable & Runnable> {...}`  
T borné par les interfaces Serializable et Runnable (T doit implémenter les deux)
- `public class D<T extends Figure & Serializable > {...}`  
T borné par l'interface Serializable et la classe Figure (T doit implémenter Serializable et étendre Figure)

# Types bornés

---

- Remarque : on utilise `extends` à la fois pour les bornes de classe et d'interface
- Un type `T` peut être borné par plusieurs interfaces, mais une seule classe (la classe doit être la première borne)
- À l'intérieur de la classe / méthode qui déclare un type borné `<T extends C>`, les méthodes de `C` peuvent être utilisés sur les objets de type `T`
- Tout type variable est implicitement borné
  - `<T>` équivalent à `<T extends Object>`



# Types bornés : exemple

- Rappel:

```
public interface Comparable<T> { int compareTo(T o); }
```

- Exemple : méthode générique avec type borné par Comparable

```
class Utilite { ...  
    static <T extends Comparable<T>> T min(T[] tab) {  
        if (tab == null || tab.length == 0) return null;  
        T m = tab[0];  
        for (int i = 1; i < tab.length; i++)  
            if (m.compareTo(tab[i]) > 0) m = tab[i];  
        return m;  
    }  
}  
class A implements Comparable<A> {...}  
...  
A[] t = new A[10]; ...; A a = Utilite.min(t);
```

# Généricité et covariance

- Les types génériques ne sont pas "covariants"
- Cela veut dire :

si A est une sous-type de B, alors C<A> n'est pas un sous-type de C<B> !

- Cela garantit l'intégrité des collections génériques :  

```
ArrayList<String> l = new ArrayList<String>();  
ArrayList<Object> lo;
```

  - Si on pouvait affecter `lo = l` on pourrait ensuite insérer n'importe quel objet dans l :  

```
lo.add (0, new Integer(5)); // ajouterait un entier à l
```
  - la liste l ne contiendrait pas que des String
- Toutefois l'absence de covariance est parfois contre-intuitive (cf. prochain transparent)

# Généricité et covariance

---

```
public class Person {...}
public class Student extends Person {...}
...
public static void changeAddress (Person p, String ad) {...}
public static void invite (ArrayList<Person> l) {...}
...
Student s1 = new Student(...); Student s2 = new Student(...);
ArrayList<Student> l = new ArrayList<Student>();
l.add(s1); l.add(s2);

changeAddress (s1, "5 rue Monge"); // OK : Student est un
                                   // sous-type de Person
invite (l); //ERREUR : ArrayList<Student> n'est pas un
              //sous-type de ArrayList<Person>
```

(pas de panique, on verra plus loin comment remédier...)

# Covariance des tableaux

- Remarque : Les tableaux (Array) sont en revanche covariants

si A est une sous-type de B, alors A[] est un sous-type de B[]

```
public static void  changeAddress (Person p, String ad) {...}
public static void invite (Person[] l) {...}
Student s1 = new Student(...); Student s2 = new Student(...);
Student[] t = new Student[2];
t[0] = s1; t[1] = s2;
changeAddress (s1, "5 rue Monge"); // OK : Student est un
                                   // sous-type de Person
invite (t); //OK : Student[] est un sous-type de Person[]
```

Quid du problème d'intégrité de type dont on a parlé pour les collections ?

# Covariance des tableaux

- Java garantit l'intégrité d'un tableau par un **contrôle de type implicite à runtime** (un tableau connaît le type de ses éléments et l'impose dynamiquement)

```
Student[] t = new Student[2];  
Object[] to = t; // OK (covariance)  
to[0] = new Object(); //ERREUR à l'exécution:  
                        // Object ne peut pas être converti vers Student
```

- Inconvénient : l'erreur est détecté seulement à runtime (ArrayStoreException)

# Type anonyme (wildcard)

- Le type anonyme (dénnoté '?') permet de spécifier un type arbitraire sans lui donner un nom

```
void printList(ArrayList<?> l) {  
    for (Object e : l) { System.out.println(e); }  
}
```

- n'importe quel type (sous-type de Object) est compatible avec '?'

```
ArrayList<String> l1 = new ArrayList<>();  
ArrayList<Integer> l2 = new ArrayList<>();  
...  
printList (l1);  
printList(l2);
```

# Wildcard vs. type variable

---

```
void printList(ArrayList<?> l) {  
    for (Object e : l) { System.out.println(e); }  
}
```

- On aurait pu obtenir cela aussi avec un type variable :

```
public <T> void printList(ArrayList<T> l) {  
    for (Object e : l) { System.out.println(e); }  
}
```

- Mais usages différents :

- ? à utiliser quand aucun autre type dépend du type anonyme
- Exemple : dans `printList` on n'a pas besoin de faire référence à `T` (pour afficher les éléments on peut les traiter comme des `Object`)

# Wildcard vs. type variable

---

- En revanche dans ce cas on ne peut pas obtenir le même effet avec un wildcard :

```
public static <T> T[] permuter (T[] tab){...}
```

- On aurait perdu la dépendance entre le type d'entrée et le type de sortie



# Wildcard vs. type variable

---

- Remarque : un wildcard ne doit pas être déclaré

```
class A {  
    ...  
    void printList(ArrayList<?> l) {...}  
  
    void f() {  
        ArrayList<?> l = new ArrayList<String>(); ...  
    }  
    ...  
}
```

# Wildcard avec bornes

---

- Comme les types variables, aussi ? peut être borné
  - mais par un seul type (i.e. pas de &)
- De plus ? admet deux types de bornes (mais pas en même temps)
  - inférieure (`extends`) et
  - supérieure (`super`)

# Wildcard avec borne inférieure

- `List<? extends Number>` une liste d'elements de n'importe quel type qui hérite de `Number` ou `Number` (le type doit être "au moins" un `Number`)

```
static double somme(List<? extends Number> l){  
    double res = 0.0;  
    for(Number n: l)  
        res += n.doubleValue();  
    return res;  
}
```

- Remarque : `n` peut être un `Number` parce que le type des éléments de `l` hérite de `Number`

# Wildcard avec borne inférieure

- `List<? extends Number>` une liste d'elements de n'importe quel type qui hérite de `Number` ou `Number` (le type doit être "au moins" un `Number`)

```
static double somme(List<? extends Number> l){  
    double res = 0.0;  
    for(Number n: l)  
        res += n.doubleValue();  
    return res;  
}
```

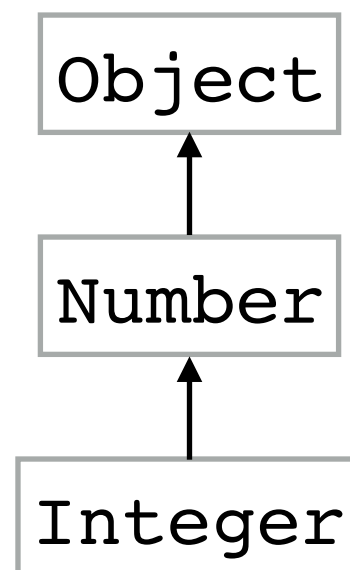
```
...  
ArrayList<Integer> l1; ArrayList<Double> l2;  
...  
double d = somme(l1);  
d = somme (l2);
```

# Wildcard avec borne supérieure

- `List<? super Number>` une liste d'elements de n'importe quel type super-classe de `Number` ou `Number` (le type doit être "au plus" un `Number`, pas plus spécifique)

```
static void plusOne(List<? super Number> l){  
    l.add(new Integer(1));  
}
```

- Remarque : on peut ajouter un `Integer` à la liste `l` parce que le type des elements de `l` est super-classe de (`Number` et donc de) `Integer`



# Wildcard avec borne supérieure

- `List<? super Number>` une liste d'elements de n'importe quel type super-classe de `Number` ou `Number` (le type doit être "au plus" un `Number`, pas plus spécifique)

```
static void plusOne(List<? super Number> l){  
    l.add(new Integer(1));  
}
```

...

```
ArrayList<Number> l1;
```

```
ArrayList<Object> l2; ArrayList<String> l3;
```

...

```
plusOne (l1);
```

```
plusOne (l2);
```

```
plusOne (l3); // ERREUR : String n'est pas une super-  
classe de Number
```

# Wildcard et covariance

- L'utilisation du wildcard permet de simuler la covariance entre types génériques

- Rappel du problème :

```
public class Person {...}
```

```
public class Student extends Person {...}
```

```
...
```

```
public static void invite (ArrayList<Person> l) {...}
```

```
...
```

```
Student s1 = new Student(...); Student s2 = new Student(...);
```

```
ArrayList<Student> l = new ArrayList<Student>();
```

```
l.add(s1); l.add(s2);
```

```
invite(l); //ERREUR : ArrayList<Student> n'est pas un  
//sous-type de ArrayList<Person>
```

- En revanche

`ArrayList<Student>` est un sous-type de  
`ArrayList<? extends Person>` (covariance)

# Wildcard et covariance (suite)

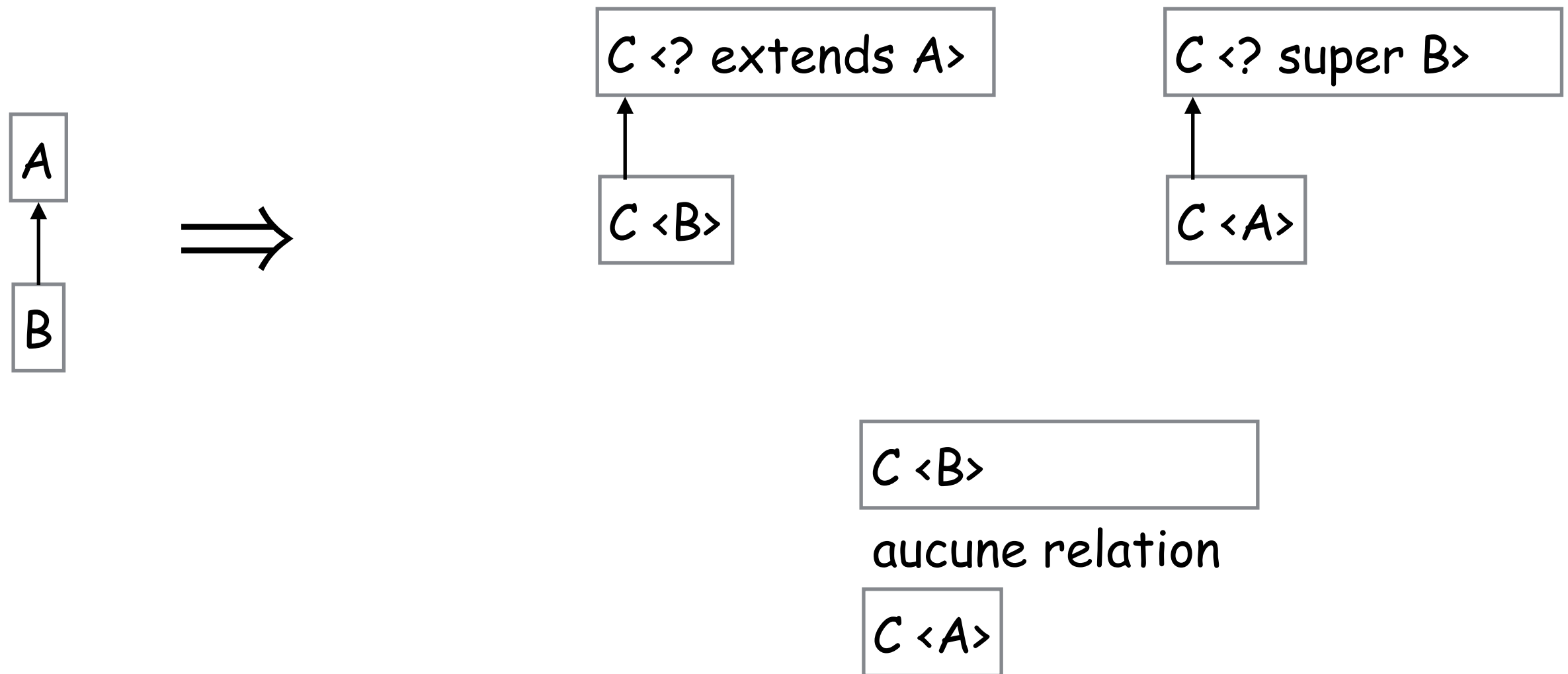
- Cela permet de résoudre le problème pour n'importe quelle collection générique

```
public class Person {...}
public class Student extends Person {...}
...
public static void invite (ArrayList<? extends Person> l)
    {...}
...
Student s1 = new Student(...); Student s2 = new Student(...);
ArrayList<Student> l = new ArrayList<Student>();
l.add(s1); l.add(s2);
invite (l); //OK : ArrayList<Student> est un
            //sous-type de ArrayList<? extends Person>
```



# Wildcard et covariance (suite)

## □ Règle générale



# Capture du wildcard

- Avant affectation, le type `<? extends A>` est compatible avec `A` et toute sous-classe de `A` (et le duale pour `<? super A>`)
- Toutefois **après affectation** (passage de paramètres ou affectation d'une variable), le `?` est **capturé**, c-à-d :
  - `<? extends A>` est considéré par le compilateur comme un type `T` dont la seule information disponible est : `T` est une sous-classe de `A` (ou `A` même). Le duale vaut pour `<? super A>`

compatible avec tout  
type qui étend Number

```
static double somme(List<? extends Number> l){  
    double res = 0.0;  
    for(Number n: l)  
        res += n.doubleValue();  
    return res;  
}
```

? **capturé** : `l` est de  
type `List<T>` pour un  
certain type `T` qui  
étend `Number`

# Capture de <? extends A> : conséquences

## □ Après la capture :

- Un objet de type `C<? extends A>` peut être accédé en lecture  
Ses éléments de type variable peuvent être affectés  
uniquement à des variables de classe A, ou super-classe de A

```
static double somme(List<? extends Number> l){  
    double res = 0.0;  
    for(Number n: l)  
        res += n.doubleValue();  
    return res;  
}
```

affecte à une variable  
Number un élément  
d'un type T qui étend  
Number : OK!

# Capture de <? extends A> : conséquences

- Après la capture :
  - Un objet de type `C<? extends A>` ne peut pas être accédé en écriture (pas d'info suffisante sur son type)

```
static double sommePlus(List<? extends Number> l) {  
    ...  
    Number n = ...;  
    l.add(n); //ERREUR  
    l.add(new Integer(1)); //ERREUR  
    ...  
}
```

- Première erreur : Le type des elements de `l` pourrait être une sous-classe stricte de `Number` (et `add` attend un paramètre de cette sous-classe)
- Deuxième erreur : Le type des elements de `l` pourrait ne pas être une super-classe de `Integer` (p.ex. pourrait être `Double`)

# Capture de <? extends A>

- Remarque
  - <?> équivalent à <? extends Object>
- => On peut faire très peu de choses avec un paramètre C<?>
  - pas d'écriture
  - lecture des éléments de type variable uniquement par des variables Object
- Usage typique (rappel) : accéder des éléments d'une collection uniquement par des méthodes de Object, quel qu'il soit leur type effectif

```
void printList(ArrayList<?> l) {  
    for (Object e : l) { System.out.println(e); }  
}
```

# Capture de <? super A> : conséquences

- **Après la capture :**
  - Un objet de type `C<? super A>` peut être accédé en écriture
    - Les méthodes de `C` qui attendent un paramètre du type variable peuvent recevoir un argument de classe `A` ou sous-classe de `A`

```
static void plusOne(List<? super Number> l){  
    l.add(new Integer(1));  
}
```

Ajoute un Integer à une liste dont le type des éléments est super-classe de Number (qui est super-classe de Integer) : OK!

# Capture de <? super A> : conséquences

- Après la capture :
  - Un objet de type `C<? super A>` peut être accédé en lecture de façon très restreinte (pas d'info suffisante sur son type)

```
static Number getFirst(List<? super Integer> l){  
    return l.get(0); // ERREUR  
}
```

`l.get` retourne un objet du type des éléments de `l`.  
Seule info : une super-classe de `Integer`. On ne sait pas laquelle (`Number?`, `Object?`)  
Conversion vers `Number` refusée

# Capture de <? super A> : consequences

- **Après la capture :**
  - Un objet de type `C<? super A>` peut être accédé en lecture de façon très restreinte (pas d'info suffisante sur son type)
    - Les elements de type variable peuvent être affectés uniquement à des variables `Object`

```
static Object getFirst(List<? super Integer> l){  
    return l.get(0); // OK  
}
```

`l.get` retourne un objet du type des elements de `l`.  
Seule info : une super-classe de `Integer` (au pire `Object`)  
Conversion implicite vers `Object`



# Wildcard, covariance et intégrité des collections

- Covariance :  
`C<SousType>` est un sous-type de `C<? extends Type>`
- Ne mine pas l'intégrité des collections
- **Rappel** : cela serait le cas si les types génériques arbitraires étaient covariants

```
ArrayList<String> l = new ArrayList<String>();  
ArrayList<Object> lo;
```

- Si on pouvait affecter `lo = l` on pourrait ensuite insérer n'importe quel objet dans `l` :  
`lo.add (0, new Integer(5)); //ajouterait un entier à l`
- la liste `l` ne contiendrait pas que des `String`

# Wildcard, covariance et intégrité des collections

- Covariance :  
`C<SousType>` est un sous-type de `C<? extends Type>`
- Ne mine pas l'intégrité des collections :

```
ArrayList<String> l = new ArrayList<String>();  
ArrayList<? extends Object> lo = l;
```

```
lo.add(new Integer(5)); // ERREUR en compilation : le  
type des elements de lo pourrait ne pas être une  
super-classe de Integer
```

- On ne peut pas modifier `l` en passant par `lo` !

# Imbrication des types variables

- Les déclarations de types variables peuvent contenir des types variables
- Rappel : Comparable est une interface générique qui indique la possibilité de comparer des objets.
  - Une classe de valeurs comparables entre eux :

```
class Point implements Comparable<Point>{  
    ...  
    public int compareTo(Point p){  
        return distance() - p.distance ();  
    }  
}
```

- Un EnsembleOrdonne est construit sur un type E d'éléments comparables entre eux, donc qui implémente Comparable<E>, d'où on pourrait avoir :

```
class EnsembleOrdonne<E extends Comparable<E>>{...}
```

# Imbrication des types variables

- Mais `<E extends Comparable<E>>` est une contrainte trop forte dans ce cas
- Raison

```
class DataPoint extends Point {  
    ...  
    //ne redéfinit pas compareTo : deux DataPoint se  
    comparent comme des Point  
}
```
- Par héritage `DataPoint implémente Comparable<Point>`, et non pas `Comparable<DataPoint>`, mais cela suffit pour cette sous-classe
- On a cependant un problème quand on veut construire un ensemble ordonné de `DataPoint` :

```
EnsembleOrdonne<DataPoint> s; //ERREUR : DataPoint  
// ne correspond pas au type <E implements Comparable <E>>
```

# Imbrication des types variables

- Solution : pour que les elements d'un type  $E$  soient comparables entre eux il suffit que  $E$  implémente `Comparable<T>` pour  $T$  égal à  $E$  ou  $T$  super-classe de  $E$   
(si un element de  $E$  est comparable à tous les éléments de type  $T$ , a fortiori on peut le comparer à tous les elements de type  $E$ , puisqu'ils ils sont également de type  $T$ )
- D'où la bonne definition de notre classe générique :  

```
class EnsembleOrdonne<E extends Comparable<? super E>> {...}
```
- La bibliothèque Java fait une utilisation massive de ces formes de généricité ...

# Implementation des classes génériques

---

- L'utilisation des classes génériques et sujette à plusieurs restrictions
- Pour comprendre la nature de ces restrictions : quelques notions sur la façon dont Java implémente les classes génériques

# Effacement

- Une invocation d'une classe générique ne crée pas un nouveau type
- Exemple :

```
public class Suite <E>{  
    ... //données  
    void ajoute(E x) {...}  
    E premier () {...}  
}
```
- Suite<Integer> pourrait correspondre à (comme en C++) :

```
public class IntegerSuite {  
    ...  
    void ajoute(Integer x);  
    Integer premier();  
}
```
- Mais en Java il n'y a pas de type pour Suite<Integer>

Une déclaration d'un type générique C<E> {...} crée **une seule classe C {...}** appelée **"effacement" (erasure)** de la classe générique ou bien classe "brute" (raw class)

# Effacement

---

- Effacement de Suite<E>
- Le type variable est "effacé" et remplacé par Object
  - (ou par A si Suite <E extends A,B,C>)

```
public class Suite {  
    ... /*T remplacé par Object dans les données*/...  
    void ajoute(Object x) {.../*T remplacé par Object*/...}  
    Object premier () {.../*T remplacé par Object*/...}  
}
```

- La classe brute est la seule que la machine virtuelle voit
  - La machine virtuelle n'a pas connaissance de la généricité ! Elle ne manipule que les classes ordinaires



# Effacement

- Pour garantir une utilisation de la classe brute qui respecte le type effectif, le compilateur introduit des casts.
- Par exemple

```
Suite<Integer> l = new Suite<Integer>;  
//utilisation de l erronée à empêcher  
//(possible si l est de type Suite):  
l.ajoute (new Point(0,0));  
Point p = (Point)l.premier();
```

- Cela est traduit par le compilateur comme :

```
Suite l = new Suite();  
l.ajoute ((Integer)new Point(0,0)); //ERREUR de cast  
Point p = (Point)(Integer)l.premier(); //ERREUR de cast
```

Ce qui génère des erreurs de cast en compilation !

# Conséquences de l'effacement et restrictions

---

- Une classe générique ne peut pas être invoquée avec un type primitif

~~ArrayList<int>;~~ //ERREUR

Raison : après l'effacement tous les elements du type variable sont traité comme des Object => doivent être des objets

# Conséquences de l'effacement et restrictions

---

- Deux objets

```
C<A> a = new C<A> (...);
```

```
C<B> b = new C<B> (...);
```

à runtime ont le même type dynamique (la classe brute *C*)

- `a.getClass() == b.getClass()` est vrai
- `a instanceof C` et `b instanceof C` sont vrai

- En effet à l'exécution `a` ne porte plus aucune information sur quelle invocation a permis sa construction

# Conséquences de l'effacement et restrictions

- Aucune instantiation n'est possible pour un type variable

```
class MaClasse<E> {  
    ...  
    public E g() {...}  
    public void f() {  
        E v = new E(); //impossible  
        E v = g(); //OK. Uniquement new est interdit  
    }  
}
```

- En effet `new E()` serait traduit par le compilateur comme `new Object()`
- `new` ne serait donc pas appelé sur le type passé à l'invocation

# Conséquences de l'effacement et restrictions

- (suite) Remarque : on peut en revanche passer à `f` le constructeur à utiliser pour construire un objet de type `E` :

```
class MaClasse<E> {  
    ...  
    public void f (Supplier<E> constr) {  
        E v = constr.get(); //OK  
        ...  
    }  
}  
Ensuite : MaClasse<String> m;... m.f(String::new);
```

- Rappel :

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get ();  
}
```

# Conséquences de l'effacement et restrictions

- On ne peut pas créer un tableau d'elements de type variable

```
class MaClasse<E> {  
    ...  
    public void f() {  
        E[] tab = new E[10]; //impossible  
        ... // variables/arguments de type E[] possibles  
    }  
}
```

- Raison : traduit comme `Object[]` par l'effacement  
=> et non pas un tableau du type passé à l'invocation
- "Workaround" :
  - Utiliser `ArrayList<E>`
  - ou `Object[]` (mais pas de contrôle de type)

# Conséquences de l'effacement et restrictions

- On ne peut pas créer un tableau d'elements de type générique

```
new ArrayList<String>[10]; //interdit
```

- Raison : serait traduit comme `new ArrayList[10]` par l'effacement  
=> on ne pourrait pas garantir le contrôle de type à l'exécution  
(on peut garantir tous les elements de type `ArrayList`,  
mais pas `ArrayList<String>`)

- Unique Exception : le wildcard non borné

```
ArrayList<?>[] l = new ArrayList<?>[10]; //OK
```

parce que cela permet explicitement des tableaux hétérogènes :

```
l[0] = new ArrayList<String>();  
l[1] = new ArrayList<Integer>();
```

# Conséquences de l'effacement et restrictions

- On ne peut pas faire référence au type variable d'une classe générique dans une méthode ou champ **statique** de cette classe

```
public class MaClasse<T> {  
    private static T var; // ERREUR  
    public static T getVar(); // ERREUR  
}
```

- Intuition : les variable/méthodes statiques appartiennent à la classe, mais il y a une seule classe MaClasse après effacement : et elle ne peut pas avoir une variable/méthode pour chaque valeur possible de T
- Remarque : cela ne concerne pas les méthodes génériques, qui peuvent en revanche être statiques :

```
public static <T> T[] permuter (T[] tab){...}
```



# Généricité et héritage

---

- On peut hériter d'une classe générique (et implementer une interface générique)
- Ex. dans `java.util`  

```
public class ArrayList<E> extends AbstractList<E>  
                                implements List<E>
```
- Effet obtenu :
  - `ArrayList<String>` étend `AbstractList<String>` et implémente `List<String>`
  - `ArrayList<Integer>` étend `AbstractList<Integer>` et implémente `List<Integer>`
  - etc.
- Mais il faut faire attention avec l'**overriding**...

# Généricité et héritage : overriding

- Pour qu'il y ait overriding, deux méthodes doivent avoir la même signature après effacement

```
class C<T> {  
    void f ( T x ) {...};  
}
```

```
class F<T> extends C<T> {  
    void f ( T x ) {...}; //overriding : deux f(Object)  
}
```

```
class H<T> extends C<T> {  
    void f ( Object x ) {...}; //overriding : deux f(Object)  
}
```

```
class G<T> extends C<T> {  
    void f ( Object x ) {...};  
    void f ( T x ) {...}; //ERREUR : deux f(Object)  
}
```

# Généricité et héritage : overriding et surcharge

```
class Base<T>{  
    void m(int x){};  
    void m(T t){};  
    void m(String s){};  
    <N extends Number> void m(N n){};  
    void m(File<?> q){};  
}
```

Après effacement :

```
m(int)  
m(Object)  
m(String)  
m(Number)  
m(File)
```

```
class D<T> extends Base<T>{  
    void m(Integer i){} //nouveau (surcharge)  
    void m(Object t){}  // redéfinit m(T t)  
    void m(Number n){}  // redéfinit m(N n)  
}
```

# Généricité et héritage : overriding et surcharge

- Pour qu'il y ait overriding, deux méthodes doivent avoir la même signature après effacement

```
class C<T> {  
    void f ( T x ) {...};  
}  
class D<T> extends C<T> {  
    void f ( String x ) {...}; //surcharge : f(Object) et f(String)  
}  
...  
D<Integer> g = new D<Integer>(); g.f("Coucou"); // f de D  
D<String> d = new D<String>();  
//surcharge entre f(Object) et f(String) même dans D<String>  
//pas d'erreur  
d.f("Coucou"); //ERREUR : ambigu, deux méthodes possibles
```

# Généricité et héritage

- On peut hériter d'une **instance** d'une classe (ou interface) générique

```
class C<T> {  
    void f ( T x ) {...};  
}  
class D<T> extends C<String> {  
    void f ( String x ) {...}; // overriding !  
}
```

- Le compilateur implémente ce genre d'héritage de façon à garantir l'overriding des méthodes **instanciées** de la classe mère (introduction de méthodes "bridge")

# Code antecédent les classes génériques

---

- Les classes brutes sont en principe utilisables :

```
ArrayList l = new ArrayList();
```

- Mais en général pas de raison de les utiliser !
- Sauf quand on doit s'interfacer avec de l'ancien code ("legacy code")
  - Les collections n'étaient pas génériques avant Java 5.0

# Code antecédent les classes génériques

---

```
class AncienCode {
    public static void f ( List l) {l. add(new Integer(1));}
    ...
}
class MaClasse {
    public static void main (String[] args) {
        ArrayList<String> a =...;
        AncienCode.f (a); //possible : a est de type
                          // ArrayList après effacement
    }
}
```

- Toutefois on a un warning en compilation :  
[unchecked] unchecked call to add(E) as a member of the  
raw type List
- En effet plus aucun cast ne contrôle le type des éléments de l dans AncienCode
  - Intégrité de type possiblement violée

# Code antecédent les classes génériques

---

- Si on est sûr que la méthode `f ( )` ne peut pas violer l'intégrité de type (pas le cas dans le slide précédent) on peut ignorer warning

```
class AncienCode {  
    public static void f ( List l) { /*affiche l*/}  
    ...  
}  
class MaClasse {  
    public static void main (String[] args) {  
        List<String> a =...;  
        AncienCode.f (a);  
    }  
}
```



# Code antecédent les classes génériques

- Situation similaire pour les types retournés par les classes legacy

```
class AncienCode {  
    public static ArrayList g () {...}  
    ...  
}  
class MaClasse {  
    public static void main (String[] args) {  
        ArrayList<String> a = AncienCode.g(); //possible :  
        //a est de type ArrayList après effacement  
    }  
}
```

- Warning en compilation :  
[unchecked] unchecked conversion
- En effet rien ne garantit que la liste retournée par g ( ) contient des String

# Code antecédent les classes génériques

---

- Après avoir vérifié que `g()` renvoie le bon type, on peut supprimer ce type de warning

```
class AncienCode {  
    public static ArrayList g () {...}  
    ...  
}  
@SuppressWarnings("unchecked")  
class MaClasse {  
    public static void main (String[] args) {  
        ArrayList<String> a = AncienCode.g();  
    }  
}
```

# Exemple : conversion d'une collection en tableau

- Supposons que l'on veuille convertir en tableau une `File<E>`
  - on a vu précédemment que l'on ne peut ni instancier un objet `E` ni créer un tableau de `E`
  - **Première solution** : créer et retourner un **tableau de `Object`**
    - mais on perd l'avantage du contrôle de type
  - **Deuxième solution** : on peut passer un **tableau du type variable** de la taille appropriée à une méthode qui retourne ce tableau rempli avec les éléments de la file
  - **Troisième solution** : on peut passer un **constructeur de tableau du type variable** à une méthode qui crée un tableau avec ce constructeur et le remplit avec les éléments de la file

# Conversion d'une collection en tableau : tableau de Object

---

```
public class File<E> {
    protected Cellule<E> tete;
    protected Cellule<E> queue;
    ...
    public Object[] toArray(){
        Object[] tab = new Object[getTaille()];
        int i = 0;
        for(Cellule<E> c = tete; c != null; c = c.getSuivant())
            tab[i++] = c.getElement();

        return tab;
    }
}
```

# Conversion d'une collection en tableau : tableau de type variable

---

Un essai :

```
public class File<E> {
    protected Cellule<E> tete;
    protected Cellule<E> queue;
    ...
    public E[] toArrayEssai(E[] tab){
        int i=0;
        for(Cellule<E> c = tete; c != null && i< tab.length;
                                c=c.getSuivant())
            tab[i++] = c.getElement();
        return tab;
    }
}
```

Utilisation

```
File<String> fs = new File<String>();
String[] u;
u = fs.toArrayEssai(new String[fs.getTaille()]);
```

# Conversion d'une collection en tableau : tableau de type variable

---

- Mais,
  - il faut que le tableau passé en paramètre soit un tableau de  $E$ , alors qu'un tableau d'une super-classe de  $E$  devrait fonctionner (si  $F$  est une superclasse de  $E$ , un tableau de  $F$  peut contenir des objets  $E$ ).
  - possible avec une méthode générique :

## Conversion d'une collection en tableau : tableau de type variable

---

```
public class File<E> {  
    ...  
    public <T> T[] toArray(T[] tab){  
  
        int i=0;  
        for(Cellule<E> c= tete; c != null && i< tab.length;  
                                c=c.getSuivant())  
            //à tab[i++] on doit affecter c.getElement()  
            // mais tab[i] -> T, c.getElement() -> E  
        return tab;  
    }  
}
```

# Conversion d'une collection en tableau : tableau de type variable

```
public class File<E> {  
    ...  
    public <T> T[] toArray(T[] tab){  
  
        int i=0;  
        for(Cellule<E> c= tete; c != null && i< tab.length;  
                                c=c.getSuivant())  
            //à tab[i++] on doit affecter c.getElement()  
            // mais tab[i] -> T, c.getElement() -> E  
            tab[i++] = (T)c.getElement();  
        return tab;  
    }  
}
```

- OK mais provoque un warning "File.java uses unchecked or unsafe operations"
- Compiler avec option -Xlint pour plus de details
  - "unchecked cast" : l'effacement ne permet pas de vérifier si la conversion de type est correcte
- Si T n'est pas une super-classe de E, erreur à l'exécution



# Conversion d'une collection en tableau : tableau de type variable

- Autre solution (pas de warning mais même possibilité d'erreur runtime)

```
public class File<E> {  
    ...  
    public <T> T[] toArray(T[] tab){  
        Object[] tmp = tab; //covariance des tableaux  
        int i=0;  
        for(Cellule<E> c= tete; c != null && i< tab.length;  
                                c=c.getSuivant())  
            tmp[i++] = c.getElement();  
        return tmp;  
    }  
}
```

- Notons que tmp est un tableau d'Object, ce qui est nécessaire pour lui affecter des éléments de type E
- Si T n'est pas une super-classe de E : erreur à l'exécution
  - tmp[i++] = c.getElement(); génère une **ArrayStoreException**
- Notons enfin que T ne sert pas dans le corps de la méthode
  - sert uniquement à imposer que le type du tableau retourné soit égal au type du tableau de l'argument

# Conversion d'une collection en tableau : constructeur de tableau

## □ Rappel :

```
@FunctionalInterface
public interface IntFunction<R> {
    R apply (int value);
}
```

peut être utilisé comme type pour le constructeur d'un tableau : choisir T[] pour R

```
public class File<E> {
    ...
    public <T> T[] toArray (IntFunction<T[]> generator){
        T[] tmp = generator.apply(getTaille());
        int i=0;
        for(Cellule<E> c= tete; c != null; c=c.getSuivant())
            tmp[i++] = (T)c.getElement();
        return tmp;
    }
}
```

- Warning en compilation : `tmp[i++] = (T)c.getElement();`  
unchecked cast
- Si T n'est pas une super-classe de E : erreur à l'exécution

# Conversion d'une collection en tableau : constructeur de tableau

## □ Rappel :

```
@FunctionalInterface
public interface IntFunction<R> {
    R apply (int value);
}
```

peut être utilisé comme type pour le constructeur d'un tableau : choisir T[] pour R

```
public class File<E> {
    ...
    public <T> T[] toArray (IntFunction<T[]> generator) {
        T[] tmp = generator.apply(getTaille());
        int i=0;
        for(Cellule<E> c= tete; c != null; c=c.getSuivant())
            tmp[i++] = (T)c.getElement();
        return tmp;
    }
}
```

## □ Utilisation sur fs de type File<DataPoint> :

```
Point[] t = fs.toArray(Point[]::new);
```

## Conversion d'une collection en tableau : dans l'interface Collection

---

- ▣ Les trois solutions sont disponibles dans l'interface Collection de Java :

```
public interface Collection<E> extends Iterable<E> {  
    ...  
  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    default <T> T[] toArray (IntFunction<T[]> generator)...  
        //depuis Java 11  
}
```