

M1 info

GINF41B2 (Conception et Programmation Orientée Objet)



Cours #8 Généricité

Pierre Tchounikine

Plan

- Généricité : un mécanisme d'abstraction
- Exemple : utilisation de la généricité pour les collections
- Différents formats
 - Exemple : une classe/méthode générique
- Classes génériques et types hérités
- Généricité contrainte
- *Généricité hors POO : l'exemple de ADA*

Généricité : un mécanisme d'abstraction

3/57
Cours P. Tchounikine

Généricité

polymorphisme paramétrique

possibilité de définir des algorithmes et/ou des types complexes (classes, interfaces) paramétrés par des types

Exemples

- algorithmes

Tri, Recherche, etc.

l'algorithme ne dépend pas du type des objets manipulés

- structures

Liste, Pile, File, Arbre, etc.

les manipulations associées au type abstrait de données ne dépendent pas du type des objets manipulés

4/57
Cours P. Tchounikine

Généricité

- La généricité est un mécanisme d'abstraction

tri d'un tableau d'entier
entre 1 et 50



tri d'un tableau d'éléments de type T
entre une borne inf et une borne sup
en utilisant une relation d'ordre sur les T

- La généricité n'est pas spécifique à la POO
 - élément clé de langages comme ADA
 - permet de faire des choses que les concepts de la POO (héritage, interfaces, polymorphisme) permettent d'aborder **d'une autre façon mais pas avec les mêmes propriétés**



dans Java, « rajouté » pour des raisons de sécurité

en particulier : pour gérer les Collections

5/57
Cours P. Tchounikine

Généricité / Polymorphisme

sur l'exemple des structures

- Polymorphisme :
 - idée : la structure contient/manipule des objets de types différents
 - mise en œuvre : utilisation de l'héritage *mais liés*


Exemple prototypique

Tableau de Figures contenant des Figures, des Rectangles, des Carrés, ...

- Généricité
 - idée : la structure contient/manipule des objets d'un type non fixé lors de l'écriture (de la structure, méthode, classe) ...
... mais qui le sera lors de l'instanciation

Exemple prototypique

Tableau de <T> contenant des <T> + instanciation avec T = Carré, Etudiant, Entier, ...

 introduit pour sécuriser
les choses et garantir
l'unicité du type

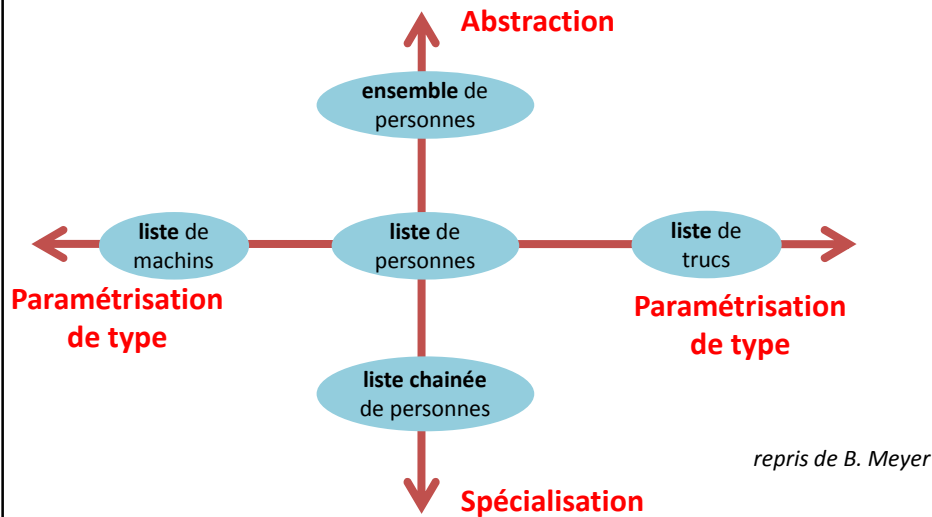
différence
entre

un ensemble de n'importe quoi
un ensemble avec des T1, des T2, etc.
un ensemble de T (T étant à fixer)

6/57
Cours P. Tchounikine

Abstraction et généricité

Liste = ensemble ordonné d'objets



7/57
Cours P. Tchounikine

Généricité et typage



Classe = module = type

Classe = module = modèle de type

8/57
Cours P. Tchounikine

Retour sur : structure polymorphe

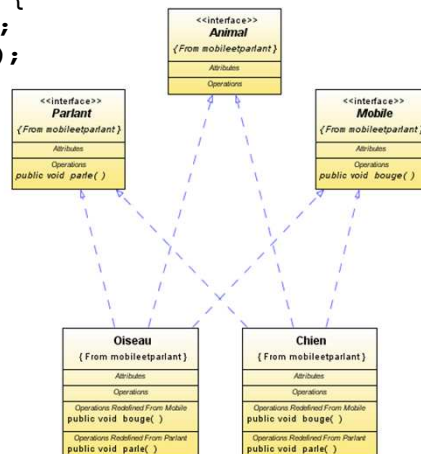
possibilité de définir des tableaux (des structures) d'objets différents

```
Animal zoo[ ] = {new Oiseau(), new Chien(), new Chien()};
for (int i=0; i<zoo.length; i++) {
    ((Mobile)zoo[i]).bouge( );
    ((Parlant)zoo[i]).parle( );
}
```

modélisation OK ou bidouille ?

Si Animal est une Classe ...

Si Animal est une Interface ...



Généricité en Java : format de base

- Définition d'une classe générique

```
public class Couple <T>
```

T est le paramètre (« paramétrage de classe »)

- Instanciation (ou invocation)

```
Couple <Integer> c1 = new Couple (i,j);
Couple <Personne> c2 = new Couple (p1,p2);
```

paramètre réel = une classe

→ pas un type primitif

(« Integer » et pas « int »)

(mais pas un souci avec l'autoboxing)

Généricité en Java



La généricité en Java est mise en œuvre par un mécanisme de **remplacement**

T → `Object` ou Classe ≈ borne supérieure si contrainte



Quelques limitations / effets de bord parfois inattendus

Pour une utilisation de la généricité plus avancée que l'utilisation des collections génériques, regarder en détail comment cela fonctionne

11/57
Cours P. Tchounikine

**Exemple :
utilisation de la généricité
pour les collections**

JDK >5

12/57
Cours P. Tchounikine

Interface Collection

java.util

Interface Collection<E>

All Superinterfaces:

[Iterable<E>](#)

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingDeque<E>](#), [BlockingQueue<E>](#), [Deque<E>](#), [List<E>](#), [NavigableSet<E>](#), [Queue<E>](#), [Set<E>](#), [SortedSet<E>](#)

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayDeque](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [ConcurrentSkipListSet](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [SynchronousQueue](#), [TreeSet](#), [Vector](#)

```
public interface Collection<E>
    extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like [Set](#) and [List](#). This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

13/57
Cours P. Tchounikine

Classe LinkedList

java.util

Class LinkedList<E>

```
java.lang.Object
├── java.util.AbstractCollection<E>
│   ├── java.util.AbstractList<E>
│   │   ├── java.util.AbstractSequentialList<E>
│   │   │   └── java.util.LinkedList<E>
```

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

[Serializable](#), [Cloneable](#), [Iterable<E>](#), [Collection<E>](#), [Deque<E>](#), [List<E>](#), [Queue<E>](#)

14/57
Cours P. Tchounikine

Classe Prof

```
public class Prof {
    private String nom;
    private int enseignement; // qualité de l'enseignement
    private int recherche;    // qualité de la recherche

    Prof(String nom, int enseignement, int recherche){
        this.nom=nom;
        this.enseignement=enseignement;
        this.recherche=recherche;}

    String getNom(){return nom;}
    int getEnseignement(){return enseignement;}
    int getRecherche(){return recherche;}

    void affiche(){...}
}
```

15/57
Cours P. Tchounikine

Main

```
import java.util.*;
public static void main(String[] args) {
```

```
    LinkedList <Prof> l = new LinkedList <Prof>();
```

```
    l.add(new Prof("Jean l'enseignant", 15, 4));
    l.add(new Prof("Alfred la recherche", 4, 15));
    l.add(new Prof("Gégé le nullard", 3, 3));
    l.add(new Prof("Philippe l'équilibré", 12, 12));
    l.add(new Prof("Pierre Tchounikine", 18, 18));
    Collections.shuffle(l);
```

java.util
Class Collections
java.lang.Object
↳ java.util.Collections

```
Collections.sort(l);
for(Prof m:l) m.affiche();
}
```

OK ?

Quel critère pour trier des Prof ?

**sort utilise une méthode
compareTo qu'il faut définir !**



```
Collections.sort(l);
cannot find symbol
symbol : method sort(java.util.LinkedList<genericite.Prof>)
location: class java.util.Collections
```


Collections

java.util

Class Collections

[java.lang.Object](#)
└─ [java.util.Collections](#)

```
public class Collections
extends Object
```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

T ou une sur-classe, cf. plus loin

- public static <T extends [Comparable](#)<? super T>> void **sort**([List](#)<T> list)
Sorts the specified list into ascending order, **according to the natural ordering of its elements**. All elements in the list must implement the Comparable interface.
- public static <T> void **sort**([List](#)<T> list, [Comparator](#)<? super T> c)
Sorts the specified list **according to the order induced by the specified comparator**.

17/57
Cours P. Tchounikine

Interface Comparable

java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

[Delayed](#), [Name](#), [RunnableScheduledFuture](#)<V>, [ScheduledFuture](#)<V>

All Known Implementing Classes:

[Authenticator.RequestorType](#), [BigDecimal](#), [BigInteger](#), [Boolean](#), [Byte](#), [ByteBuffer](#), [Calendar](#), [Character](#), [CharBuffer](#), [Charset](#), [ClientInfoStatus](#), [CollationKey](#), [ComponentBaselineResizeBehavior](#), [CompositeName](#), [CompoundName](#), [Date](#), [Date](#), [Desktop.Action](#), [Diagnostic.Kind](#), [Dialog.ModalExclusionType](#), [Dialog.ModalityType](#), [Double](#), [DoubleBuffer](#), [DropMode](#), [ElementKind](#), [ElementType](#), [Enum](#), [File](#), [Float](#), [FloatBuffer](#), [Formatter.BigDecimalLayoutForm](#), [FormSubmitEvent.MethodType](#), [GregorianCalendar](#), [GroupLayout.Alignment](#), [IntBuffer](#), [Integer](#), [JavaFileObject.Kind](#), [JTable.PrintMode](#), [KeyRep.Type](#), [LayoutStyle.ComponentPlacement](#), [LdapName](#), [Long](#), [LongBuffer](#), [MappedByteBuffer](#), [MemoryType](#), [MessageContext.Scope](#), [Modifier](#), [MultipleGradientPaint.ColorSpaceType](#), [MultipleGradientPaint.CycleMethod](#), [NestingKind](#), [Normalizer.Form](#), [ObjectName](#), [ObjectStreamField](#), [Proxy.Type](#), [Rdn](#), [Resource.AuthenticationType](#), [RetentionPolicy](#), [RoundingMode](#), [RowFilter.ComparisonType](#), [RowIdLifetime](#), [RowSorter.EventType](#), [Service.Mode](#), [Short](#), [ShortBuffer](#), [SOAPBinding.ParameterStyle](#), [SOAPBinding.Style](#), [SOAPBinding.Use](#), [SortOrder](#), [SourceVersion](#), [SSLEngineResult.HandshakeStatus](#), [SSLEngineResult.Status](#), [StandardLocation](#), [String](#), [SwingWorker.StateValue](#), [Thread.State](#), [Time](#), [Timestamp](#), [TimeUnit](#), [TravelIcon.MessageType](#), [TypeKind](#), [URI](#), [UUID](#), [WebParam.Mode](#), [XmlAccessOrder](#), [XmlAccessType](#), [XmlNsForm](#)

Method Summary

int [compareTo](#)([T](#) o)

Compares this object with the specified object for order.

18/57
Cours P. Tchounikine

Classe Prof

```
public class Prof implements Comparable <Prof> {
    // reste identique

    public int compareTo(Prof p){
        if (this.getEnseignement()+ this.getRecherche()
            < p.getEnseignement()+p.getRecherche()) return -1;
        else
            if (this.getEnseignement()+ this.getRecherche()
                > p.getEnseignement()+ p.getRecherche())return 1;
        else return 0;}
}
```

pas de cast !

Le fait de spécifier que l'on va implémenter un compareTo sur des Prof (et pas des Object) éviter un cast du genre

```
public int compareTo(Object o){
    Prof p = (Prof)o;
    ...
}
```

19/57
Cours P. Tchounikine

Exécution

```
System.out.println("Tri / enseignement + recherche");
Collections.sort(l);
for(Prof m:l) m.affiche();
```

Tri / enseignement + recherche

```
[Gégé le nullard Ens= 3 Rech= 3]
[Jean l'enseignant Ens= 15 Rech= 4]
[Alfred la recherche Ens= 4 Rech= 15]
[Philippe l'équilibré Ens= 12 Rech= 12]
[Pierre Tchounikine Ens= 18 Rech= 18]
```

20/57
Cours P. Tchounikine

Utilisation d'un comparateur

java.util

Interface Comparator<T>

Type Parameters:

T - the type of objects that may be compared by this comparator

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as [Collections.sort](#) or [Arrays.sort](#)) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as [sorted sets](#) or [sorted maps](#)), or to provide an ordering for collections of objects that don't have a [natural ordering](#).

Method Summary

int	compare (T o1, T o2) Compares its two arguments for order.
boolean	equals (Object obj) Indicates whether some other object is "equal to" this comparator.

21/57
Cours P. Tchounikine

Utilisation de Comparator

avec une classe interne

```
public static void main(String[] args) {

    LinkedList <Prof> l = new LinkedList <Prof>();
    // remplissage de la liste

    // définition d'une classe interne

    class ComparatorEnseignement implements Comparator <Prof> {
        public int compare (Prof p1, Prof p2){
            if (p1.getEnseignement() < p2.getEnseignement()) return -1;
            else if (p1.getEnseignement() > p2.getEnseignement()) return 1;
            else return 0;}
    }

    System.out.println("Tri / enseignement");
    Collections.sort(l, new ComparatorEnseignement());
    for(Prof m:l) m.affiche();

}
```

22/57
Cours P. Tchounikine

Exécution

```
System.out.println("Tri / enseignement");
Collections.sort(l,new ComparatorEnseignement());
for(Prof m:l) m.affiche();
```

```
Tri / enseignement
```

```
[Gégé le nullard Ens= 3 Rech= 3]
[Alfred la recherche Ens= 4 Rech= 15]
[Philippe l'équilibré Ens= 12 Rech= 12]
[Jean l'enseignant Ens= 15 Rech= 4]
[Pierre Tchounikine Ens= 18 Rech= 18]
```

23/57
Cours P. Tchounikine

Utilisation de Comparator

avec **des classes internes**

```
public static void main(String[] args) {

LinkedList <Prof> l = new LinkedList <Prof>();
// remplissage de la liste

// définition de classes internes
class ComparatorEnseignement implements Comparator <Prof> {}
    // en comparant / enseignement
class comparatorRecherche implements Comparator <Prof> {}
    // en comparant / recherche

System.out.println("Tri / enseignement");
Collections.sort(l,new comparatorEnseignement());
for(Prof m:l) m.affiche();

System.out.println("Tri / recherche");
Collections.sort(l,new comparatorRecherche());
for(Prof m:l) m.affiche();
```

24/57
Cours P. Tchounikine

Exécution

```
System.out.println("Tri / enseignement");
Collections.sort(l,new comparatorEnseignement());
for(Prof m:l) m.affiche();

System.out.println("Tri / recherche");
Collections.sort(l,new comparatorRecherche());
for(Prof m:l) m.affiche();
```

```
Tri / enseignement
[Gégé le nullard Ens= 3 Rech= 3]
[Alfred la recherche Ens= 4 Rech= 15]
[Philippe l'équilibré Ens= 12 Rech= 12]
[Jean l'enseignant Ens= 15 Rech= 4]
[Pierre Tchounikine Ens= 18 Rech= 18]
```

```
Tri / recherche
[Gégé le nullard Ens= 3 Rech= 3]
[Jean l'enseignant Ens= 15 Rech= 4]
[Philippe l'équilibré Ens= 12 Rech= 12]
[Alfred la recherche Ens= 4 Rech= 15]
[Pierre Tchounikine Ens= 18 Rech= 18]
```

25/57
Cours P. Tchounikine

Utilisation de Comparator

avec une classe anonyme

```
public static void main(String[] args) {

LinkedList <Prof> l = new LinkedList <Prof>();
// remplissage de la liste

// utilisation d'une classe anonyme
System.out.println("Tri réel");
Collections.sort(l,new Comparator <Prof>()
{
    public int compare (Prof p1, Prof p2){
        if (p1.getEnseignement()+ 5*p1.getRecherche())<
        p2.getEnseignement()+5*p2.getRecherche()) return -1;
        else if (p1.getEnseignement()+ 5*p1.getRecherche())>
        p2.getEnseignement()+ 5*p2.getRecherche())return 1;
        else return 0;}
    }
});

for(Prof m:l) m.affiche();
```

26/57
Cours P. Tchounikine

Exécution

```
Collections.sort(l,new Comparator <Prof>()
    { // la classe anonyme
    });
for(Prof m:l) m.affiche();
```

Tri réel

```
[Gégé le nullard Ens= 3 Rech= 3]
[Jean l'enseignant Ens= 15 Rech= 4]
[Philippe l'équilibré Ens= 12 Rech= 12]
[Alfred la recherche Ens= 4 Rech= 15]
[Pierre Tchounikine Ens= 18 Rech= 18]
```

27/57
Cours P. Tchounikine

Retour sur la liste

```
import java.util.*;
public static void main(String[] args) {

    LinkedList <Prof> l = new LinkedList <Prof>();

    l.add(new Prof("Jean l'enseignant", 15, 4));
    l.add(new Prof("Alfred la recherche", 4, 15));
    l.add(new Prof("Gégé le nullard", 3, 3));
    l.add(new Prof("Philippe l'équilibré", 12, 12));
    l.add(new Prof("Pierre Tchounikine", 18, 18));

    // ProfEmerite extends Prof
    l.add(new ProfEmerite("Papy Marcel", 12, 8));

    // Classe Machin
    Machin m1 = new Machin();
    l.add(m1);
}
```

OK ?

OK ?

28/57
Cours P. Tchounikine

Différents formats de généricité

29/57
Cours P. Tchounikine

Généricité : différents formats

- Classe générique
`class C <T>`
- Méthode au sein d'une classe générique C <T>
`public <T> method (...)`
- Méthode générique au sein d'une classe non générique
`class C`
`public <T> void method (T t ...)`
- Méthode générique au sein d'une classe générique C <T>
`class C <T>`
`public <U> void method (U u ...)`

30/57
Cours P. Tchounikine

Exemple : une méthode générique

exercice d'école

```
public class ClasseRecherche {
    static public <E> int recherche (E T[], E e, int i, int n){
        // cherche e dans T entre les indices i et n
        if (i>n) return -1;
        else if (T[i]==e) return i;
        else return recherche (T,e,i+1,n);
    }
}
```

méthode de classe

```
public static void main(String[] args) {
    Integer T1[]={12,4,7,34,66,9};
    Character T2[]={ 'd','n','j','p' };
    System.out.println("7 est à l'indice " +
        ClasseRecherche.recherche (T1,7,0,5));
    System.out.println("n est à l'indice " +
        ClasseRecherche.recherche (T2,'n',0,3));
}
```

31/57
Cours P. Tchounikine

Exemple : une méthode générique

exercice d'école

```
public class ClasseRecherche {
    static public <E> int recherche (E T[], E e, int i, int n){
        // cherche e dans T entre les indices i et n
        if (i>n) return -1;
        else if (T[i]==e) return i;
        else return recherche (T,e,i+1,n);
    }
}
```



```
public static void main(String[] args) {
    Integer T1[]={12,4,7,34,66,9};
    Character T2[]={ 'd','n','j','p' };
    System.out.println("7 est à l'indice " +
        ClasseRecherche.recherche (T1,7,0,5));
    System.out.println("n est à l'indice " +
        ClasseRecherche.recherche (T2,'n',0,3));
}
```

32/57
Cours P. Tchounikine

Exemple : une méthode générique

exercice d'école

```
public class ClasseRecherche {
    static public <E> int recherche (E T[], E e, int i, int n){
        // cherche e dans T entre les indices i et n
        if (i>n) return -1;
        else if (T[i]==e) return i;
        else return recherche (T,e,i+1,n);
    }
}
```

```
Integer T1[]={12,4,7,34,66,9};
Character T2[]={ 'd','n','j','p' };
System.out.println("toto est à l'indice " +
    ClasseRecherche.recherche (T1,"toto",0,3));
Machin m = new Machin(55,6.6);
System.out.println("m est à l'indice " +
    ClasseRecherche.recherche (T1,m,0,3));
```



type erasure
compilé comme si
on mettait partout
"Object"

Il faut forcer le compilateur en rappelant le type : // pb détecté à la compil
ClasseRecherche.<Integer>recherche (T1,"toto",0,5);

on pourrait aussi contraindre Recherche : <E extends Number> (...)

33/57
Cours P. Tchounikine

Classes génériques et types hérités

34/57
Cours P. Tchounikine

Un Etudiant est une Personne

```
public class Personne {
    private String nom;
    Personne (String nom){
        this.nom=nom;}
    String getNom(){return nom;}
}

public class Etudiant extends Personne{
    private int identifiant;

    Etudiant (String nom, int identifiant){
        super(nom);
        this.identifiant = identifiant;}

    int getIdentifiant(){
        return identifiant;}
}
```

35/57
Cours P. Tchounikine

Couple est une classe générique

```
public class Couple <T> {
    private T premier;
    private T second;

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

    public void setPremier (T t){this.premier=t;}

    public void setSecond (T t){this.second=t;}

    public T getPremier (){return premier;}

    public T getSecond (){return second;}
}
```

36/57
Cours P. Tchounikine

Contrôle des types à la compilation

```
Integer i=3, j=4;
Couple <Integer> c1 = new Couple (i,j);

Personne p1 = new Personne("Paul");
Personne p2 = new Personne("Jacques");
Couple <Personne> c2 = new Couple (p1,p2);

Couple <Etudiant> c3 = new Couple (new Etudiant ("Joe",1234),
                                   new Etudiant ("Bill",789));

c2.setPremier(p2);    légal ?    oui

c3.setPremier(p1);    légal ?    non

c2.setPremier(new Etudiant ("Jack",832)); légal ?    oui
```

37/57
Cours P. Tchounikine

Les Couples sont ils des Couples ?

```
Integer i=3, j=4;
Couple <Integer> c1 = new Couple (i,j);

Personne p1 = new Personne("Paul");
Personne p2 = new Personne("Jacques");
Couple <Personne> c2 = new Couple (p1,p2);

Couple <Etudiant> c3 = new Couple (new Etudiant ("Joe",1234),
                                   new Etudiant ("Bill",789));

c1=c2; légal ?    non, un couple de personnes n'est pas un couple d'entiers

c3=c2; légal ?    non, un couple de personnes n'est pas un couple d'étudiants

c2=c3; légal ?    est-ce qu'un couple d'étudiants est un couple de personnes ?
```

38/57
Cours P. Tchounikine

Les Couples sont ils des Couples ?

```

Personne p1 = new Personne("Paul");
Personne p2 = new Personne("Jacques");
Couple <Personne> c2 = new Couple (p1,p2);

Couple <Etudiant> c3 = new Couple (new Etudiant ("Joe",1234),
                                   new Etudiant ("Bill",789));

```

`c2=c3;` **légal ?** étant donné qu'un étudiant est une personne,
est-ce qu'un couple d'étudiants est un couple de personnes ?

Si `c2=c3` est légal

Est-ce que `c2.setPremier(p1)` est légal ?

Dans ce cas, quel est l'effet de `c2.setPremier(p1)` ?

Dans ce cas, quel est le résultat de `c3.getPremier().getIdentifiant();` ?

A hérite de B



C <A> hérite de C

pb de la double référence

39/57
Cours P. Tchounikine

Notion de « joker »

```

Integer i=3, j=4;
Couple <Integer> c1 = new Couple (i,j);

Couple <Personne> c2 = new Couple (p1,p2);
Personne p1 = new Personne("Paul");
Personne p2 = new Personne("Jacques");

Couple <Etudiant> c3 = new Couple (new Etudiant ("Joe",1234),
                                   new Etudiant ("Bill",789));

```

`c1=c2;` // illégal

`c3=c2;` // illégal

`c2=c3;` // illégal **mais pas pratique !**

//utilisation d'un joker

Couple <?> cj;

`cj=c2;` // légal

`cj=c3;` // légal

`c2.setPremier(p2);` // légal

`cj.setPremier(p2);` // illégal

les appels de méthodes
comportant un argument
de type ? sont interdits



accès en modification impossible

40/57
Cours P. Tchounikine

Utilisation des jokers

un Couple de Personne est un Couple de Personne

(et un Couple d'Etudiant n'est pas un Couple de Personne)

- Joker

- Couple <?> cj1;

// couple de n'importe quoi

- Joker + sémantique (« contrainte »)

- contrainte / descendance

Couple <? extends Personne> cj2;

// couple de Personne ou d'une sous-classe

- contrainte / ascendance

Couple <? super Personne> cj3;

// couple de Personne ou d'une sur-classe

Aller voir les détails !

41/57
Cours P. Tchounikine

Généricité et héritage : différents formats

- class C1 <T> extends C2 <T>

- class C1 <T,U> extends C2 <T,U>

- class C1 <T,U> extends C2 <T>

- class C1 <T extends U> extends C2 <T>

- class C1 <T> extends C2

- class C1 extends C2 <T>

- class C1 <T> extends C2 <T extends Number>

42/57
Cours P. Tchounikine

Généricité contrainte

43/57
Cours P. Tchounikine

Couple est une classe générique

```
public class Couple <T> {  
    private T premier;  
    private T second;  
  
    public Couple (T premier, T second){  
        this.premier=premier;  
        this.second=second;}  
  
    public void setPremier (T t){this.premier=t;}  
  
    public void setSecond (T t){this.second=t;}  
  
    public T getPremier (){return premier;}  
  
    public T getSecond (){return second;}  
}
```

**Comment faire pour
afficher un Couple ?**

44/57
Cours P. Tchounikine

Couple est une classe générique paramétrée

```
public class Couple <T extends Affichable> {
    private T premier;
    private T second;

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

    public void setPremier (T t){this.premier=t;}
    public void setSecond (T t){this.second=t;}
    public T getPremier (){return premier;}
    public T getSecond (){return second;}

    public void affiche(){
        this.premier.affiche();
        this.second.affiche();
    }
}
```

```
public interface Affichable {
    void affiche();
}
```

Je sais afficher un Couple si je
sais afficher les objets de type T

45/57
Cours P. Tchounikine

Personne implémente Affichable

```
public interface Affichable {
    void affiche();
}
```

```
public class Personne implements Affichable{
    private String nom;
    Personne (String nom){this.nom=nom;}
    String getNom(){return nom;}

    public void affiche(){
        System.out.println(this.getNom());
    }
}
```

46/57
Cours P. Tchounikine

Règles générales / contraintes

- Imposer à la classe T d'être une sous-classe d'une classe C

- `class ClG <T extends C>`

`class Couple <T extends Number>`

- Imposer à la classe T d'implémenter une interface I

- `class ClG <T extends I>`

`class Couple <T extends Comparable>`

- Imposer à la classe T d'être une sous-classe d'une classe donnée C et d'implémenter une ou plusieurs interfaces I, J,...

- `class ClG <T extends C & I & J>`

`class Couple <T extends Number & Comparable & Affichable>`

47/57
Cours P. Tchounikine

Couples mixtes ?



type erasure

```
Couple <Personne> c4 = new Couple (new Etudiant ("Joe",1234),p1);
i=c4.getPremier().getIdentifiant(); // refusé à la compil
i=c4.getSecond().getIdentifiant();// refusé à la compil
```

```
Couple <Etudiant> c5 = new Couple (new Etudiant ("Jack",1234),p1);
i=c5.getPremier().getIdentifiant(); // OK
i=c5.getSecond().getIdentifiant(); // plante à l'exécution
```

```
Etudiant e=new Etudiant ("Jack",1234);
Machin m=new Machin(); // Machin implements Affichable
Couple <Etudiant> c6 = new Couple (e,m);
i=c6.getPremier().getIdentifiant();
i=c6.getSecond().getIdentifiant(); // plante à l'exécution
```

pour empêcher de marier des chauve-souris et des parapluies :

`Couple <Etudiant> c7 = new Couple <Etudiant> (e,m);`

pb détecté
à la compil

`public class Couple <T extends Personne & Affichable>`

48/57
Cours P. Tchounikine

Généricité contrainte : conclusion

- Intérêt : **fixer la nature minimale de l'information dont on s'abstrait**
 - expliciter l'abstraction construite
 - expliciter les contraintes (→ vérification)
- Prix à payer
 - définir les classes et /ou interfaces correspondant aux contraintes
 - définir les classes permettant d'adapter le type utilisé aux contraintes stipulées

généricité non contrainte = généricité contrainte avec (contrainte = *object*)

49/57
Cours P. Tchounikine

Généricité hors POO : exemple de ADA

50/57
Cours P. Tchounikine

La généricité en ADA

- Principes ADA :
 - langage impératif classique, structure de blocs
 - compilation séparée dépendante (typage fort, vérification des correspondances à la compilation)
 - réutilisation de composants stockés dans des librairies
 - composants prédéfinis
 - composants du projet
- Typage fort → sécurité, mais peut amener à dupliquer du code
 - exemples : Swap de variables, Tri d'un tableau, etc.



composants réutilisables génériques

51/57
Cours P. Tchounikine

Paquetage de gestion de pile

spécification

```
generic
  type element is private;
package gest_pile is
  type pile(taille : positive) is private;
  procedure empiler (p : in out pile; e : in element);
  procedure depiler (p : in out pile; e : out element);
  function pile_vide(p : pile) return boolean;
  pilevide, pilepleine : exception;

private
  type tableau is array (positive range<>) of element;
  type pile(taille : positive) is
    record
      table : tableau(1..taille);
      sommet : natural := 0;
    end record;
end gest_pile;
```

52/56
Cours P. Tchounikine

Paquetage de gestion de pile

corps

```

package body gest_pile is
  procedure empiler (p : in out pile; e : in element) is
  begin
    if (p.sommet = p.taille) then raise pilepleine; end if;
    p.sommet := p.sommet + 1;
    p.table(p.sommet) := e;
  end;
  procedure depiler (p : in out pile; e : out element) is
  begin
    if (p.sommet = 0) then raise pilevide; end if;
    e := p.table(p.sommet);
    p.sommet := p.sommet - 1;
  end;
  function pile_vide(p : pile) return boolean is
  begin
    return (p.sommet = 0);
  end;
end gest_pile;

```

53/56
Cours P. Tchounikine

Utilisation

```

with gest_pile, text_io, integer_text_io;
use text_io, integer_text_io;
procedure essai is
  package gp is new gest_pile(integer);
  use gp;
  p : pile(10);
  x : integer;
  begin
    empiler(p,101);
    depiler(p,x);
    put(x);
    new_line;
    depiler(p,x);
  exception
    when pilevide => begin
      -- gestion du problème
      -- ou propagation
    end;
  end essai;

```

54/56
Cours P. Tchounikine

Recherche dichotomique

généricité contrainte

spécification

```
generic
  type element is private;
  type indice is (<>);
  type tableau is array (indice range <>) of element;
  with function "<" (x,y : element) return boolean is <>;
  with function "=" (x,y : element) return boolean is <>;
  with function milieu(i1,i2 : indice) return indice;
procedure dichotomie (t : in tableau; e : in element;
  trouve : out boolean; val : out indice);
```

Généricité contrainte en ADA :

- ❖ met en avant la contrainte fonctionnelle (existence des opérations)
- ❖ différent / OO qui met l'accent sur la notion de type et d'encapsulation

55/56
Cours P. Tchounikine

Recherche dichotomique

généricité contrainte

corps

```
procedure dichotomie (t : in tableau; e : in element ;
  trouve: out boolean; val : out indice) is
  inf:indice:=t'first;
  sup:indice:=t'last;
  m:indice;
begin
  if (e<t(inf)) or (t(sup)<e) then trouve:=false;
  else
    while inf<sup loop
      m:=milieu(inf,sup);
      if not (t(m)<e) then sup:=m; else inf:=indice'succ(m);
      end if;
    end loop;
    if e=t(inf) then trouve:=true;val:=inf; else trouve:=false;
    end if;
  end if;
end dichotomie;
```

56/56
Cours P. Tchounikine