

# Généricité en Java

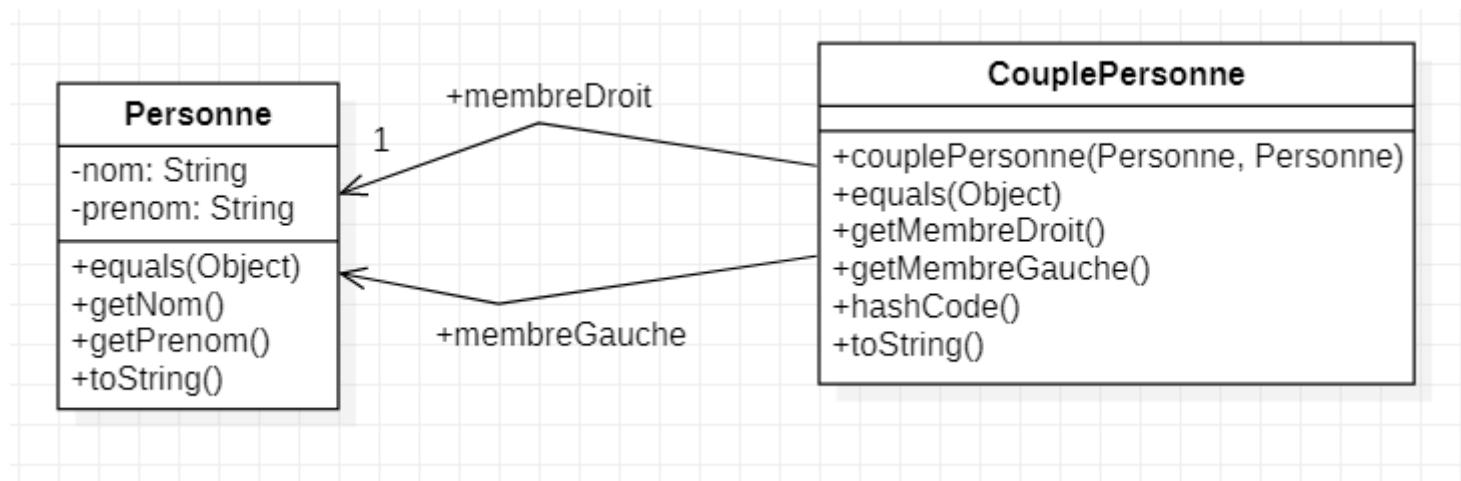


# Plan du cours

- Intérêt de la généricité
- Créer des classes « génériques »
- Contraintes sur les paramètres
- Le cas particulier des tableaux

# Exemple Couple

- Une classe **CouplePersonne** permettant de décrire un couple de Personne.



# Code de la classe Personne

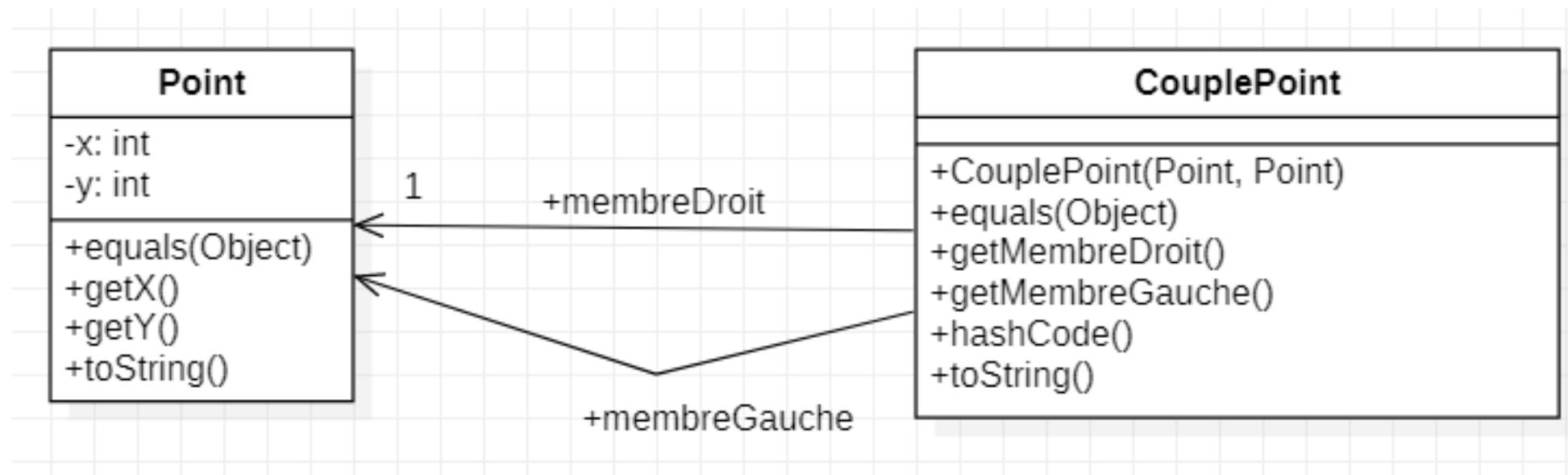
```
public class Personne {  
    private String nom;  
    private String prenom;  
  
    public Personne(Personne p) {  
        this.nom = new String(p.getNom());  
        this.prenom = new String(p.getPrenom());  
    }  
    public Personne(String nom, String prenom) {  
        super();  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    public String getNom() {  
        return nom;  
    }  
    public String getPrenom() {  
        return prenom;  
    }  
    public String getPre  
    public String getPrenom() {  
        return prenom;  
    }  
    @Override  
    public boolean equals(Object object) {  
        if (this == object) {  
            return true;  
        }  
        if (!(object instanceof Personne)) {  
            return false;  
        }  
        final Personne other = (Personne)object;  
        if (!(nom == null ? other.nom == null : nom.equals(other.nom))) {  
            return false;  
        }  
        if (!(prenom == null ? other.prenom == null : prenom.equals(other.prenom))) {  
            return false;  
        }  
        return true;  
    }  
    @Override  
    public String toString() {  
        return prenom + " " + nom;  
    } |
```

# Code de la classe CouplePersonne

```
public class CouplePersonne {  
    private Personne membreGauche;  
    private Personne membreDroit;  
  
    public CouplePersonne(Personne mg, Personne md) {  
        this.membreGauche = mg;  
        this.membreDroit = md;  
    }  
    public Personne getMg() {  
        return membreGauche;  
    }  
    public Personne getMd() {  
        return membreDroit;  
    }  
  
    @Override  
    public String toString() {  
        return "("+membreGauche+", "+membreDroit+");  
    }  
  
    @Override  
    public boolean equals(Object object) {  
        if (this == object) {  
            return true;  
        }  
        if (!(object instanceof CouplePersonne)) {  
            return false;  
        }  
        final CouplePersonne other = (CouplePersonne)object;  
        if (!(membreGauche == null ? other.membreGauche == null : membreGauche.equals(other.membreGauche)))  
            return false;  
        if (!(membreDroit == null ? other.membreDroit == null : membreDroit.equals(other.membreDroit))) {  
            return false;  
        }  
        return true;  
    }  
}
```

# Classe CouplePoint

- une classe permettant de décrire un couple de Point.



# Point

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        super();  
    }  
  
    public Point(int x, int y) {  
        super();  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
}
```



# Code de la classe CouplePoint

```
public class CouplePoint {  
    private Point membreGauche;  
    private Point membreDroit;  
  
    public CouplePoint(Point mg, Point md) {  
        this.membreGauche = mg;  
        this.membreDroit = md;  
    }  
    public Point getMembreGauche() {  
        return membreGauche;  
    }  
    public Point getMembreDroit() {  
        return membreDroit;  
    }  
  
    @Override  
    public String toString() {  
        return "(" +membreGauche+ " , "+membreDroit+ ")";  
    }  
    @Override  
    public boolean equals(Object object) {  
        if (this == object) {  
            return true;  
        }  
        if (!(object instanceof CouplePoint)) {  
            return false;  
        }  
        final CouplePoint other = (CouplePoint)object;  
        if (!(membreGauche == null ? other.membreGauche == null : membreGauche.equals(other.membreGauche))) {  
            return false;  
        }  
        if (!(membreDroit == null ? other.membreDroit == null : membreDroit.equals(other.membreDroit))) {  
            return false;  
        }  
        return true;  
    }  
}
```

# CouplePoint vs. CouplePersonne

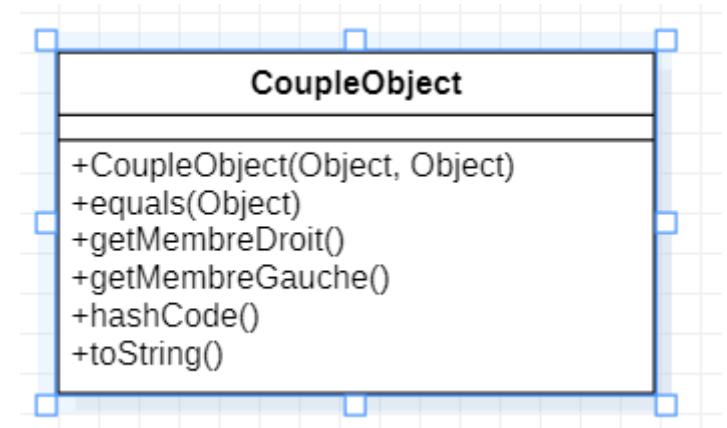
Redondance ?

```
public class CouplePoint {  
    private Point membreGauche;  
    private Point membreDroit;  
  
    public CouplePoint(Point mg, Point md) {  
        this.membreGauche = mg;  
        this.membreDroit = md;  
    }  
    public Point getMembreGauche() {  
        return membreGauche;  
    }  
    public Point getMembreDroit() {  
        return membreDroit;  
    }  
  
    @Override  
    public String toString() {  
        return "(" +membreGauche+ " , "+membreDroit+ ")";  
    }  
}
```

```
public class CouplePersonne {  
    private Personne membreGauche;  
    private Personne membreDroit;  
  
    public CouplePersonne(Personne mg, Personne md) {  
        this.membreGauche = mg;  
        this.membreDroit = md;  
    }  
    public Personne getMg() {  
        return membreGauche;  
    }  
    public Personne getMd() {  
        return membreDroit;  
    }  
  
    @Override  
    public String toString() {  
        return "(" +membreGauche+ " , "+membreDroit+ ")";  
    }  
}
```

# Redondance ?

- Les classes CouplePoint et CouplePersonne sont très similaires
  - (il suffit de remplacer Point par Personne...)
  - Si on veut un couple de Double
    - On copie/colle, duplique ?
    - => réécrire à chaque fois la même chose ???
- Une solution pourrait être de passer par une super classe telle que la classe Object....



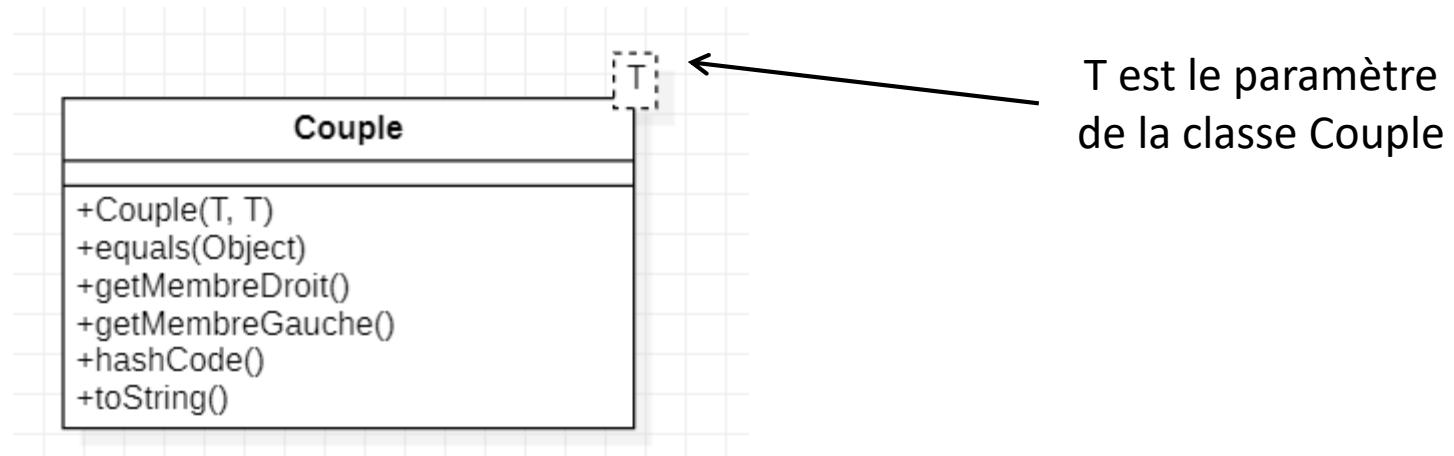
# Mais...

```
public class CoupleObject {  
    private Object membreGauche;  
    private Object membreDroit;  
  
    public CoupleObject(Object mg, Object md) {  
        this.membreGauche = mg;  
        this.membreDroit = md;  
    }  
    public Object getMg() {  
        return membreGauche;  
    }  
    public Object getMd() {  
        return membreDroit;  
    }  
  
    @Override  
    public String toString() {  
        return "("+membreGauche+", "+membreDroit+");  
    }  
  
    public boolean equals(Object objet) {  
        if (this == objet) {  
            return true;  
        }  
        if (!(objet instanceof CoupleObject)) {  
            return false;  
        }  
        final CoupleObject other = (CoupleObject)objet;  
        if (!(membreGauche == null ? other.membreGauche == null : membreGauche.equals(other.membreGauche))) {  
            return false;  
        }  
        if (!(membreDroit == null ? other.membreDroit == null : membreDroit.equals(other.membreDroit))) {  
            return false;  
        }  
        return true;  
    }  
}
```

```
public class TestCoupleObject {  
  
    public static void main(String[] args) {  
        Personne individu1 = new Personne("Young", "Angus");  
        Personne individu2 = new Personne("Young", "Malcolm");  
  
        CouplePersonne couplePersonne = new CouplePersonne(individu1, individu2);  
  
        Point point1 = new Point(10, 45);  
        Point point2 = new Point(10, 45);  
        CouplePoint couplePoint = new CouplePoint(point1, point2);  
  
        CoupleObject coupleObject = new CoupleObject(individu1, point1);  
  
    }  
}
```

=> Classe paramétrique

# La classe Couple ... paramétrée



# Code de la classe Couple paramétrée

```
/*
 * @param <T> : type des éléments du couple
 */
public class Couple <T> {
    private T membreGauche;
    private T membreDroit;

    public Couple(T mg, T md) {
        super();
        this.membreGauche = mg;
        this.membreDroit = md;
    }

    @Override
    public String toString() {
        return "(" +membreGauche+ , " +membreDroit+ )";
    }

    public T getMg() {
        return membreGauche;
    }

    public T getMd() {
        return membreDroit;
    }
}

@Override
public boolean equals(Object object) {
    if (this == object) {
        return true;
    }
    if (!(object instanceof Couple)) {
        return false;
    }
    final Couple other = (Couple) object;
    if (!(membreGauche == null ? other.membreGauche == null : membreGauche.equals(other.membreGauche))) {
        return false;
    }
    if (!(membreDroit == null ? other.membreDroit == null : membreDroit.equals(other.membreDroit))) {
        return false;
    }
    return true;
}
```

# Et on l'utilise

```
public class TestCoupleObject {

    public static void main(String[] args) {
        Personne individu1 = new Personne("Young", "Angus");
        Personne individu2 = new Personne("Young", "Malcolm");

        CouplePersonne couplePersonne = new CouplePersonne(individu1, individu2);

        Point point1 = new Point(10, 45);
        Point point2 = new Point(10, 45);
        CouplePoint couplePoint = new CouplePoint(point1, point2);

        CoupleObject coupleObject = new CoupleObject(individu1, point1);

        Couple<Personne> couplePersonneParametree = new Couple<Personne>(individu1, individu2);
        Couple<Point> couplePointParametree = new Couple<Point>(point1, point2);
        Couple<Object> coupleObjectBof = new Couple<Object>(individu1, point1);

    }
}
```

# Définir une classe paramétrique

- Tout d'abord à quoi correspondent les paramètres ?
  - Les paramètres correspondent à des « types » qui seront utilisés dans la classe.

- Notation :

```
public class NomDelaClasse <T,U,V>
```

//la classe possède 3 paramètres

# Instancier une classe paramétrée

```
public class TestCoupleObject {  
  
    public static void main(String[] args) {  
        Personne individu1 = new Personne("Young", "Angus");  
        Personne individu2 = new Personne("Young", "Malcolm");  
  
        CouplePersonne couplePersonne = new CouplePersonne(individu1, individu2);  
  
        Point point1 = new Point(10, 45);  
        Point point2 = new Point(10, 45);  
        CouplePoint couplePoint = new CouplePoint(point1, point2);  
  
        CoupleObject coupleObject = new CoupleObject(individu1, point1);  
  
        Couple<Personne> couplePersonneParametree = new Couple<Personne>(individu1, individu2);  
        Couple<Point> couplePointParametree = new Couple<Point>(point1, point2);  
        Couple<Object> coupleObjectBof = new Couple<Object>(individu1, point1);  
  
    }  
}
```

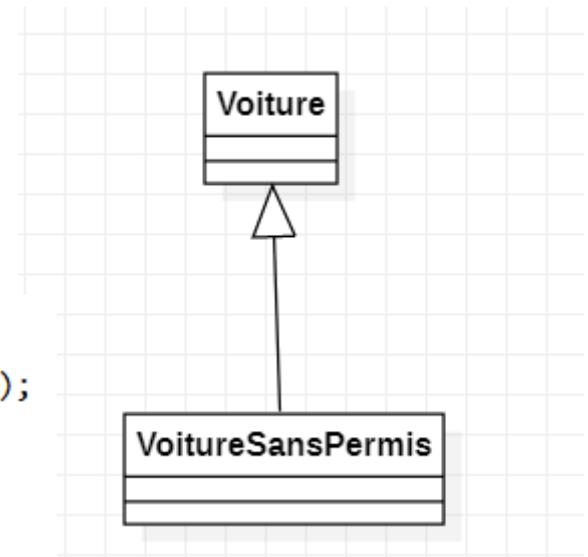
On ne peut passer en paramètre que :

- le nom d'une Classe,
- Ou
- le nom d'une Interface.

On ne peut pas passer le nom d'un type de base : int, double, etc.., il faut passer par les Wrapper-Class (ie. Double, Integer, etc...)

# Généricité et héritage

```
23  
24     ArrayList<Voiture> lesVoitures = new ArrayList<Voiture>();  
25     ArrayList<VoitureSansPermis> lesVoituresSansPermis = new ArrayList<VoitureSansPermis>();  
26     lesVoituresSansPermis = lesVoitures;  
27     lesVoitures = lesVoituresSansPermis;  
28  
29     for(Voiture v : lesVoitures) {  
30         lesVoituresSansPermis.add(v);  
31     }  
32  
33     for(VoitureSansPermis vsp : lesVoituresSansPermis) {  
34         lesVoitures.add(vsp);  
35     }
```



# Notion de wildcard

- Il existe un joker : ?
- `ArrayList<?> l;`
- Revient à déclarer l comme une liste de n'importe quelle type d'objet accessible en LECTURE SEULE!
- `ArrayList<? extends Number> ln;`
- ln est une liste de n'importe quel type implémentant l'interface Number (cette liste ne sera accessible qu'en lecture seule!!!).

# application

```
④ static void affiche(List<? extends Voiture> l){
    for(Voiture v : l)
        System.out.println(v.toString());
}

ArrayList<Voiture> lesVoitures = new ArrayList<Voiture>();
ArrayList<VoitureSansPermis> lesVoituresSansPermis = new ArrayList<VoitureSansPermis>();
//lesVoituresSansPermis = lesVoitures;
//lesVoitures = lesVoituresSansPermis;

//Ajout de voiture dans les listes de voitures
lesVoitures.add(new Voiture());
lesVoitures.add(new Voiture());

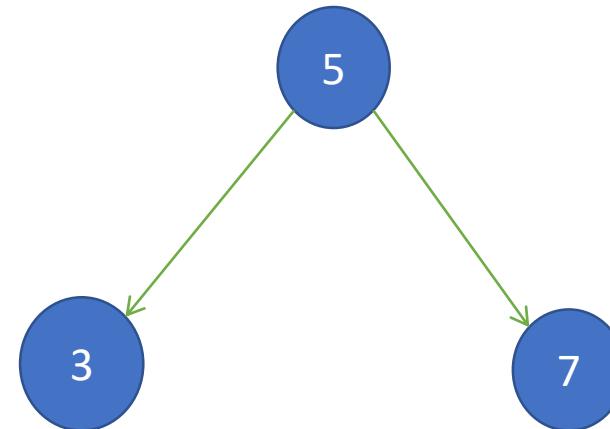
lesVoituresSansPermis.add(new VoitureSansPermis());
lesVoituresSansPermis.add(new VoitureSansPermis());

affiche(lesVoitures);
affiche(lesVoituresSansPermis);
```

<terminated> TestCoupleObject [Java Application] C:\P  
Genericite.Voiture@15db9742  
Genericite.Voiture@6d06d69c  
Genericite.VoitureSansPermis@7852e922  
Genericite.VoitureSansPermis@4e25154f

# Exemple : un arbre binaire ordonné

- Peut-on faire des arbres binaires ordonnés de n'importe quoi?
  - NON! Comme son nom l'indique il faut que les valeurs puissent être ordonnées.  
C'est-à-dire qu'on puisse comparer 2 valeurs



# Contraindre les paramètres

- Il est possible de spécifier que les paramètres satisfassent certaines contraintes
- Ces contraintes s'expriment généralement par le fait qu'un paramètre hérite d'une classe ou implémente une interface.
- Dans les 2 cas on utilise le mot-clé *extends*

# L'interface Comparable<T>

- Cette interface ne spécifie qu'une méthode :
  - public int compareTo(T o)

```
compareTo  
  
int compareTo(T o)  
  
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.  
  
The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)  
  
The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.  
  
Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.  
  
It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."  
  
In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.  
  
Parameters:  
o - the object to be compared.  
Returns:  
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.  
Throws:  
NullPointerException - if the specified object is null  
ClassCastException - if the specified object's type prevents it from being compared to this object.
```

# En Java...

```
public class ArbreBinaireOrdonne <T extends Comparable<T>>
{
    private Cellule<T> racine;

    public ArbreBinaireOrdonne()
    {
        racine = null;
    }

    public ArbreBinaireOrdonne(T value)
    {
        racine = new Cellule<T>(value);
    }

    public void ajouter(T v)
    {
        if (this.racine==null)
        {
            racine = new Cellule<T>(v);
        }
        else racine.ajouterValeur(v);
    }
}
```

```
public class Cellule<T extends Comparable<T>>
{
    private T value;
    private Cellule<T> gauche;
    private Cellule<T> droite;

    public Cellule(T value)
    {
        this.value = value;
        gauche = droite = null;
    }

    public Cellule(T val, Cellule<T> g, Cellule<T> d)
    {
        value = val;
        gauche = g;
        droite = d;
    }

    public void ajouterValeur(T v)
    {
        if(v.compareTo( value)<0)
            if(gauche == null)
                gauche = new Cellule<T>(v);
            else gauche.ajouterValeur(v);
        else if(droite == null)
                droite = new Cellule<T>(v);
            else droite.ajouterValeur(v);
    }
}
```

# Tableau et généricité

```
3 public class TableauGenerique <T>{  
4  
5     private T[] tableau;  
6  
7     public TableauGenerique(int taille) {  
8         super();  
9         this.tableau = new T[taille];  
10    }  
11  
12    public TableauGenerique(T[] tableau) {  
13        super();  
14        this.tableau = tableau;  
15    }  
16  
17  
18    public static void main(String[] args) {  
19        // TODO Auto-generated method stub  
20    }  
21  
22}  
23}
```

Cannot create a generic array of T

L'instanciation de tableaux génériques pose problème!!!  
Dans ce cas on pourra passer par un tableau d'*Object*

# Tableau et Généricité

```
3 import java.lang.reflect.Array;
4 import java.util.ArrayList;
5
6 public class TableauGenerique <T>{
7
8     private T[] tableau;
9     private T[] tableauArray;
10
11    public TableauGenerique(int taille) {
12        super();
13        this.tableau = new T[taille];
14        this.tableauArray = (T[]) Array.newInstance(ArrayList.class, taille);
15    }
16
17    public TableauGenerique(T[] tableau) {
18        super();
19        this.tableau = tableau;
20    }
21
22
23    public static void main(String[] args) {
24        // TODO Auto-generated method stub
25
26    }
27
28 }
29 }
```

Utilisation de la méthode de classe de la classe Array  
(import java.lang.reflect.Array) et des propriétés  
réflexives de Java