Cours 13

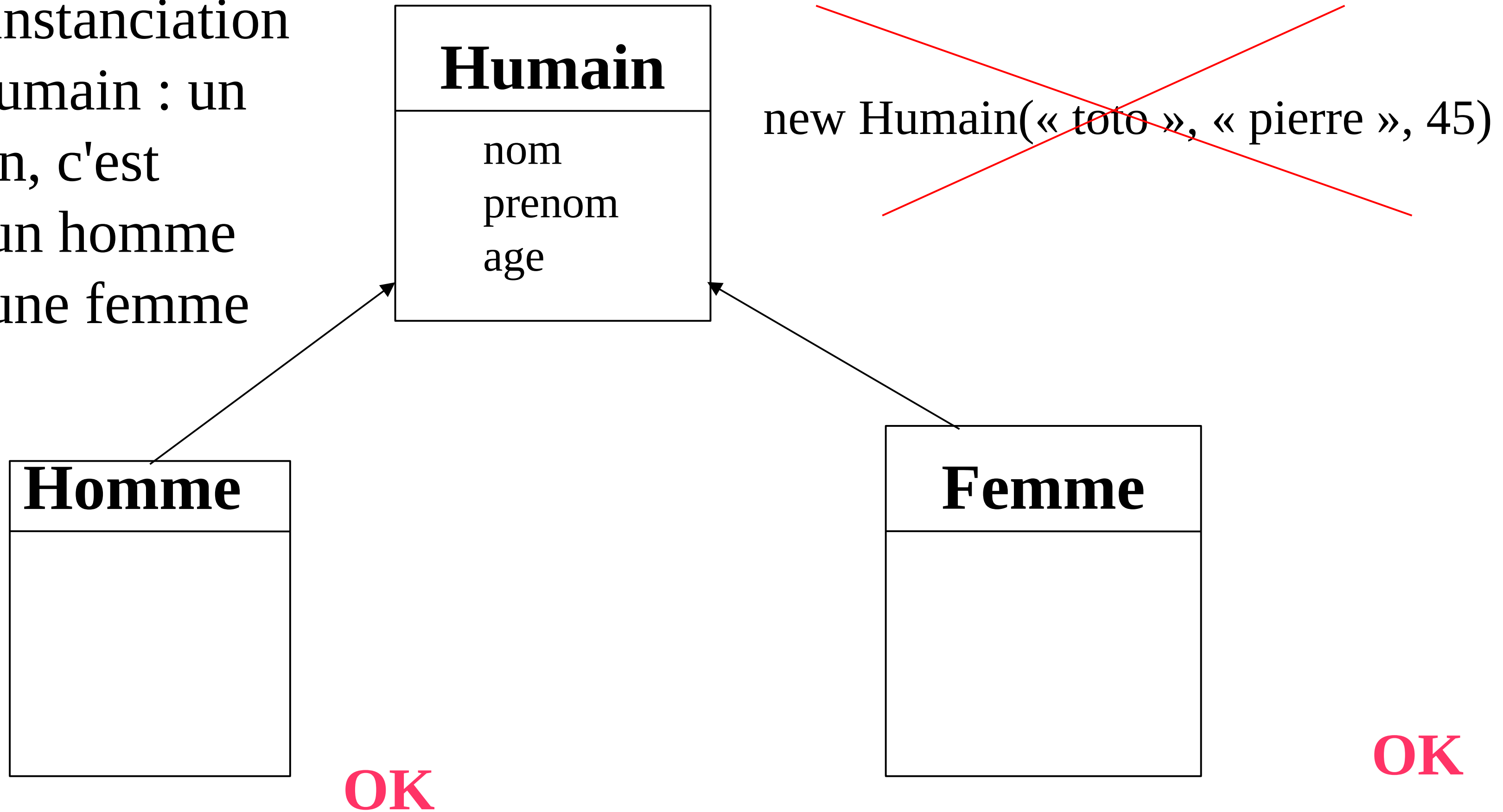
Les classes abstraites

Les méthodes abstraites

Les interfaces

Un exemple pour mieux comprendre

Pas d'instanciation d'un humain : un humain, c'est
- soit un homme
- soit une femme



new Homme("tyty","pierre",45);

new Femme("toto","anne",24);

La classe Humain sera une classe abstraite

```
public abstract class Humain {
private String nom, prenom;
private int age ;

public Humain(String s, String p, int a) {
    this.nom=s;
    this.prenom=p;
    this.age=a;
    }

public abstract char identite() ;
// retournera 'F' ou 'H' suivant les cas

public String getNom()      {
return this.nom;}

public String getPrenom()  {
return this.prenom;}

public int getAge()          {
return this.age;}
}

public String toString()     {
        return «nom »+this.nom+
           «  prénom »+this.prenom+
           « âge »+ this.age ;}
}
```

Les sous-classes de la classe abstraite donnent un corps à la méthode abstraite

```
public class Femme extends Humain {

public Femme(String s, String p, int a)
      {super(s,p,a);
      }

public char identite() {
     return 'F' ;     }

public String toString() {
String s =« femme » +super.toString();
return s ;
}
}
```

```
public class Homme extends Humain {

public Homme(String s, String p, int a)
      {super(s,p,a);
      }

public char identite() {
     return 'H' ;     }

public String toString() {
String s = «homme» +super.toString();
return s ;
}
}
```

Utiliser ces classes

```
public class TestHumain    {
public static void main(String[] args)            {
        Femme f= new Femme("toto","anne",24);
        Homme h= new Homme("tyty","pierre",45);

        Humain[] tabPers= new Humain[2];
        tabPers[0] = f;
        tabPers[1] = h;
        for (Humain hf : tabPers)   {
                System.out.print(hf.identite()+"--") ;
                System.out.println(hf.toString());
                }
   }}
```

Résultat F--femme nom toto prénom anne âge 24
H--homme nom tyty prénom pierre d'âge 45

Méthode abstraite et classe abstraite

- Une méthode abstraite est une méthode dont on ne donne que la signature et pas le corps
  - la signature (et une documentation) décrira le service rendu
- Une classe abstraite est une classe qui :
  - ne peut pas avoir d'instances : pas de new possible

```
Humain p = new Humain("remio","pierre",67) ;
```

    - *Erreur de compilation : cannot instantiate the type Humain*
  - doit nécessairement définir au moins une méthode abstraite
- Dans les deux cas : mot-clé **abstract**

Synthèse

- Une classe normale (pas abstraite) définit un contrat concret :
  - chaque méthode rend un service concret (implémenté par des instructions)
- Une classe abstraite (contenant au moins une méthode abstraite) définit un contrat mixte (concret et abstrait)
- chaque méthode concrète définit un service concret
- chaque méthode abstraite définit une spécification (service abstrait) sans implémentation

Remarque : Toute classe ayant au moins une méthode abstraite est une classe abstraite même si le mot-clé abstract n'est pas présent dans sa définition

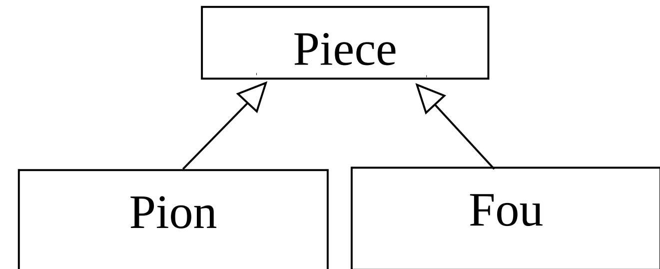
- Une classe abstraite n'est pas instanciable (pas de *new* possible)

Intérêts

- **Une classe abstraite n'a de sens que si elle possède des sous-classes**
  - Ces sous-classes devront obligatoirement donner un corps (des instructions) aux méthodes abstraites de la classe mère
- Utile pour la factorisation des variables d'instance, du code (éviter de réécrire du code plusieurs fois), même si tout n'est pas factorisable
- Utile pour le polymorphisme : revoir transparent 5

Exemple du jeu d'échecs

- Un échiquier a plusieurs pièces de types différents avec chacun un déplacement spécifique.
- On définit une hiérarchie
- On définit une classe Piece :
  - qui encapsule les caractéristiques communes à toutes les pièces (position sur l'échiquier et couleur)
  - qui définit une méthode abstraite de déplacement car le corps de cette méthode va dépendre de la pièce... On ne va dons pas lui donner un corps dans Piece !



On définit autant de sous-classes que de types de pièce (Pion, Fou, Roi, Cavalier, Dame,Tour)

- qui vont donner un corps à la méthode de déplacement

Exemple du jeu d'échecs : solution

```
public abstract class Piece {
  private int lig;
  private int col;
  boolean couleur;
public Piece (int x, int y, boolean b)
     {
     this.lig = x;
     this.col = y ;
     this.couleur = b; }

void abstract deplace(int x, int y);
} // fin classe Piece
```

```
class Pion extends Piece {
public Pion(int x,int y,boolean b)   {
            super(x,y,b); }
void deplace (int x, int y) {
...   <-----------
}
class Dame extends Piece {
public Dame(int x,int y,boolean b){
          super(x,y,b); }
void deplace (int x, int y) {
...       <--------
}
```

Intérêt et utilisation d'une classe abstraite

```
public class TestEchec {

public static void main(String[] args)    {

Piece [] pieces = new Piece [3] ;
blancs[0] =  new Pion(4,2) ;
blancs[1] =  new Pion(6,5) ;
blancs[2] =  new Dame(12,10) ;

for (Piece p : pieces) {
    p.deplace(2,4);
}
}
}
```

<u>A la compilation</u> :
le compilateur accepte car il constate qu'une méthode de signature *deplace(int, int)* est définie dans *Piece* (le fait qu'elle soit abstraite ne change rien)

<u>A l'exécution</u> :
Polymorphisme, la classe de l'objet référencé détermine la méthode *deplace* invoquée

Les interfaces

- **On peut vouloir définir un ensemble de services abstraits**

⇒ notion d'interface

le mot clé **interface** signale qu'on déclare une interface et non une classe

Que des méthodes abstraites (sans code)

```
public interface Deplacable {
        public abstract deplace(int, int);
}
```

Une interface

- Une interface en java (mot-clé interface et pas class) ne peut contenir que :
  - Des méthodes abstraites
  - Des constantes de classe

Une interface répond à la question « quels sont les services rendus ? »

mais

ne répond pas à la question « comment les services sont rendus ? »

Elle ne définit <u>aucune</u> implémentation

Exemple

Définition d'une interface

Des constantes de classe

```
public interface Dessinable   {
    public static final int taille = 2; // taille du trait

    public abstract void dessine();
    }
```

Des méthodes abstraites (sans code)

Classe implémentant une interface

- Une interface définit un contrat
- Ce contrat doit être réalisé, implémenté par une classe

- Une classe qui implémente une interface :
  - S'engage à remplir le contrat
  - C'est à dire s'engage à donner une implémentation (un corps, un ensemble d'instructions) à toutes les méthodes abstraites définies dans l'interface

Implémentations d'une interface

```
public interface Dessinable    {
    public static final int taille = 2;
    public abstract void dessine();
}
```

```
public class Triangle implements Dessinable   {
    private Point a, b, c ;
    public Triangle(Point p, Point q, Point r)    {
        this.a = p ;
        this.b = q ;
        this.c = r;}

    public Point getA()    {
            return this.a ;}

    public void dessine() {......................  }
}
```

Le corps de la méthode abstraite, cad les instructions donnant l'implémentation de l'interface Dessinable pour un Triangle

Classes, interfaces et héritage (1)

- Plusieurs classes différentes peuvent implémenter une interface
  - Elles s'engagent à remplir le même contrat

```
public class Cercle implements Dessinable{
    public void dessine(){.....................  }
    }
```

- Une classe peut hériter d'une autre classe et implémenter une interface (remplir son contrat)

```
class PointCol extends Point implements Dessinable{
....}
```

- Une classe peut implémenter plusieurs interfaces (remplir plusieurs contrats)

```
class PointColDepl extends Point implements Dessinable,
  Deplacable{
....}
```

Classes, interfaces et héritage (2)

L'héritage existe entre interfaces

- Une interface peut hériter d'une ou plusieurs autres interfaces

(elle hérite des constantes et méthodes abstraites des interfaces mères)

```
public interface DessDeplacable extends Dessinable,
   Deplacable
```

- DessDeplacable aura :

Attention de bien distinguer

héritage de structure et de comportements (classe)

réalisation de contrats et héritage de contrats

Intérêt des interfaces : nouvelle comptabilité entre types

class PointDes extends Point implements Dessinable
class Triangle implements Dessinable

- Une interface permet de définir un nouveau type, partagé par toutes les classes qui l'implémentent

```
PointDes p = new PointDes(3,5);
Triangle t = new Triangle(....);

Dessinable[] tab = new Dessinable[2];

tab[0] = p;
tab[1] = t;
```

On peut définir un tableau de ce nouveau type

p, un objet de type *PointDes*, peut être considéré comme un *Dessinable*

t, unobjet de type Triangle, peut être considéré comme un *Dessinable*

Intérêt des interfaces : encore le polymorphisme !

- Une interface permet de définir un nouveau type, partagé par toutes les classes qui l'implémentent

```
PointDes p = new PointDes(3,5);
Triangle t = new Triangle(....);
Dessinable[] tab = new Dessinable[2];
tab[0] = p;
tab[1] = t;
for (Dessinable d : tab)
        d.dessine() ;
```

À la compilation,

A l'exécution, c'est la méthode dessine de l'objet référencé par d qui est appelée

Intérêts des interfaces : nouvelle comptabilité entre types pour les paramètres

```
public class EnsFigures {
private ArrayList<Dessinable> ens ;
…
public void ajoute(Dessinable d)  {
….}
public static void main(String[] args)   {
PointDes p = new PointDes(3,5);
Triangle t = new Triangle(....);
EnsFigures ens = new EnsFigures();
ens.ajoute(p);
ens.ajoute(t);
```

comme la méthode **ajoute** exige un paramètre de type **Dessinable**, on peut passer tout objet implémentant cette interface : donc un **PointDes**

donc aussi un **Triangle**

- Permettent une couche d'abstraction dans la programmation, la rendant beaucoup plus flexible.
- Pour le java graphique
  - les listeners d'événements
  - Les « stocks » à constantes : *WindowConstants*
- Pour le java de base
  - l'interface *Comparable* (définie dans *java.lang*)
  - l'interface *Clonable*
  - L'interface *Serializable*

L'interface *Comparable<T> très, très utile...*

après java 5

lorsqu'on veut comparer des objets... si ça a un sens

L'interface Comparable<T> contient une seule méthode abstraite définissant un ordre entre les objets de type T

```
public interface Comparable<T>    {

   public int compareTo(T o) ;
```

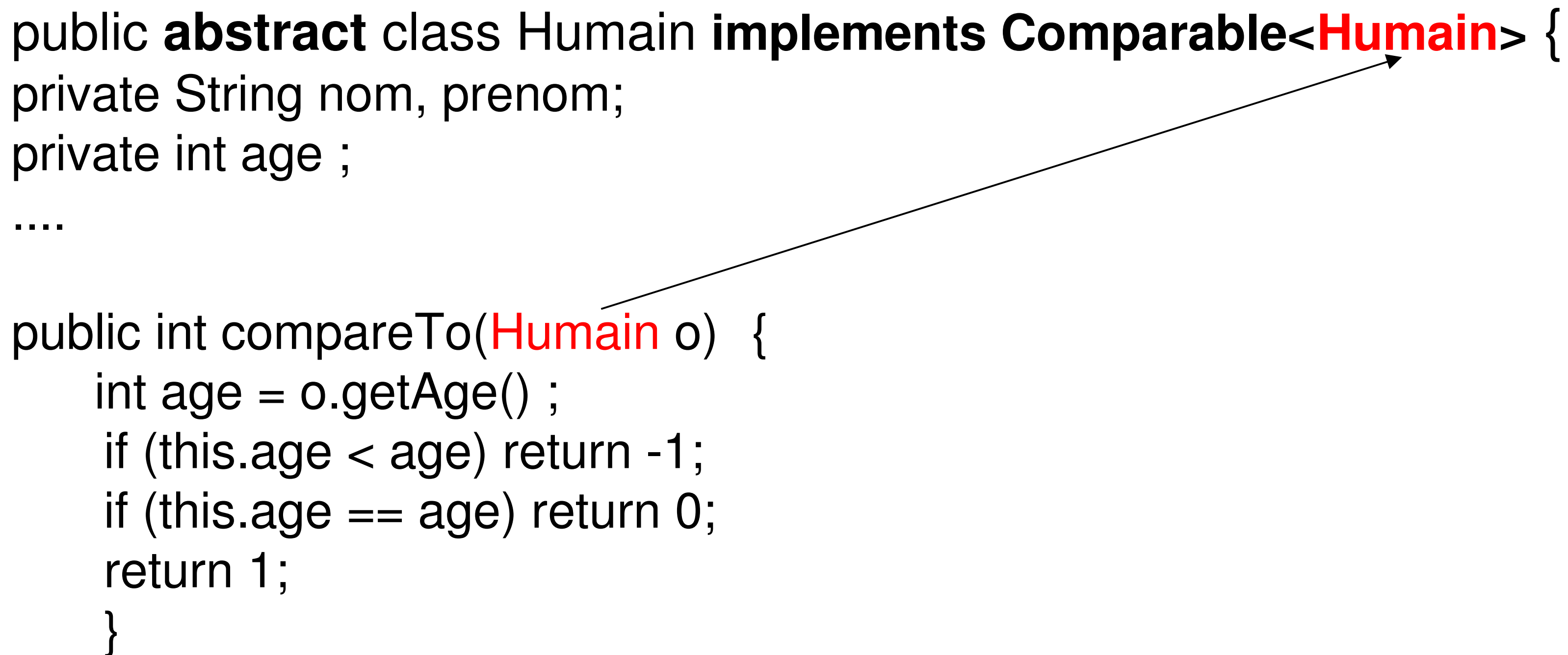
/ l'objet receveur de la méthode est comparé à l'objet passé en paramètre o. La méthode retourne :*

-1 si this < o   - 0 si this equals o   +1 si this > o

```
}
```

Exemple : Pouvoir comparer les Humains par âge
==> implémentation du contrat *Comparable* par la classe Humain

Définition ainsi d'un ordre <u>total</u> sur tous les Humains (toutes les instances de la classe Humain seront « comparables »)



```
public abstract class Humain implements Comparable<Humain> {
private String nom, prenom;
private int age ;
....

public int compareTo(Humain o)  {
      int age = o.getAge() ;
       if (this.age < age) return -1;
       if (this.age == age) return 0;
       return 1;
       }
```

Attention si *o* == null, une exception (NullPointerException) sera lancée

Un programme client

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

Femme f= new Femme("toto","anne",24);
Homme h= new Homme("tyty","pierre",24);

int comp = h.compareTo(f) ;

if ( comp ==-1)
      System.out.println(h.getNom()+" est plus jeune que "+f.getNom());
else
      if (comp==0)
            System.out.println(h.getNom()+" et "+f.getNom()+ " ont le même âge");

     else System.out.println(h.getNom()+" est plus vieux que "+f.getNom());}
```

Faire des tris facilement : utilisation de la classe outil *Arrays*

- La classe outil *Arrays* fournit des méthodes de classe (static) dont
  - une méthode de tri *sort* de signature
    - public static void sort (Object[] tabObj)
  - Cette méthode utilise la méthode *compareTo* implémentée dans le type des objets du tableau
- Pour trier n'importe quel tableau d'un type implémentant Comparable, plutôt que refaire un tri à la main, on utilise cette méthode
  - **C'est MAGIQUE !!!! et pratique**

Tri d'un tableau d'Humains

```
public class TestHumain {
public static void main(String[] args)          {
        Femme f= new Femme("toto","anne",24);
        Homme h= new Homme("tyty","pierre",48);
        Homme h2 = new Homme("gayral","paul",52);
        Homme h3 = new Homme("dupont","paolo",28);
        Femme f2= new Femme("cardoso","sylvie",43);
        Humain[] tabPers= {f, h , h2, h3, f2};
        System.out.println("avant le tri :");
        for (Humain h : tabPers)
                System.out.print(h.getAge()+"-" );
        Arrays.sort(tabPers);
        System.out.println("\naprès le tri :");
        for (Humain h : tabPers)
                System.out.println(h.getAge()+"-" );
```

Résultat

avant le tri :
24-48-52-28-43-
après le tri :
24-28-43-48-52-

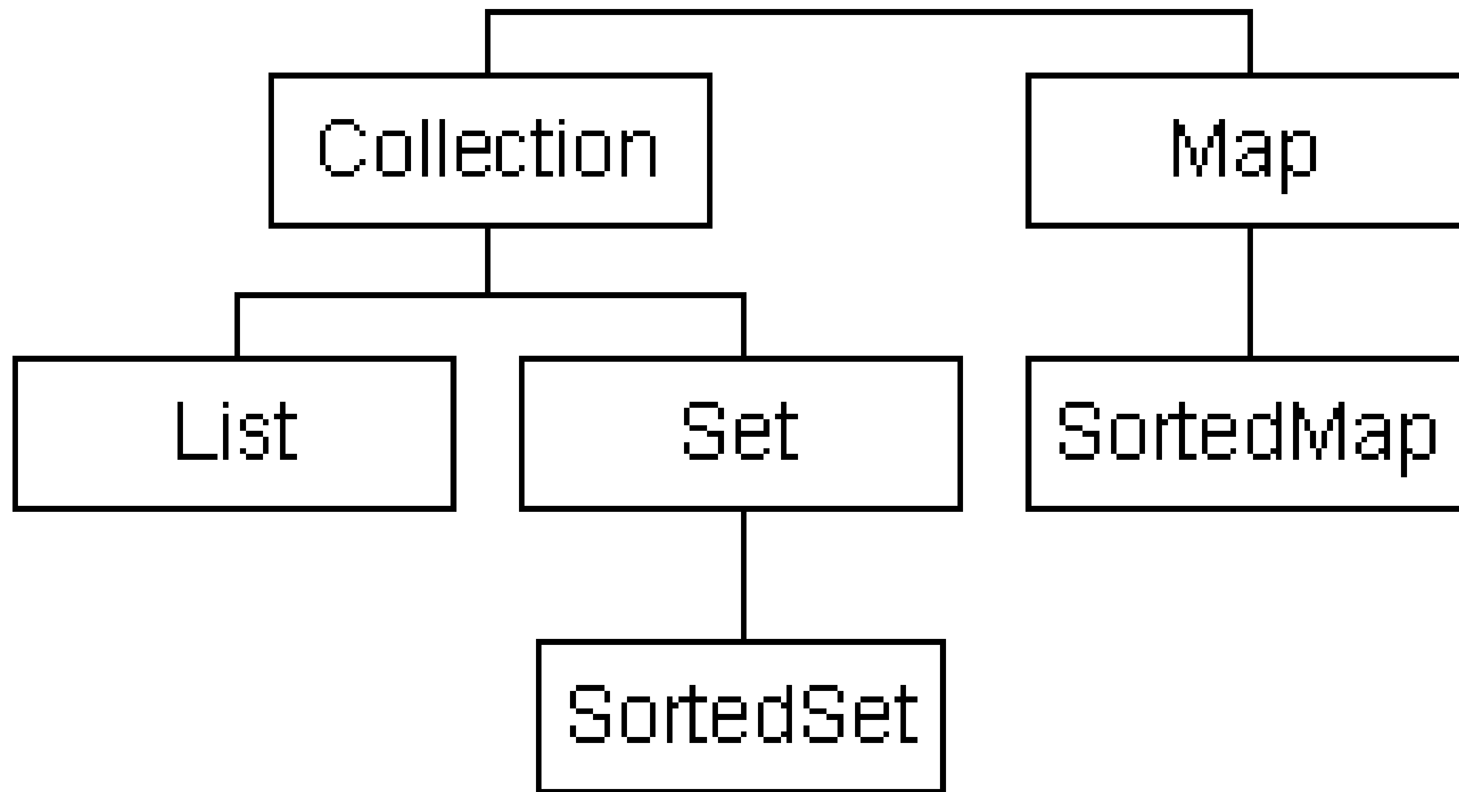
Changer l'ordre : simplissime !

On peut changer notre façon d'ordonner les instances simplement en modifiant le corps de la méthode compareTo

- Au lieu d'ordonner les Humain par âge, on peut ordonner
  - Par ordre alphabétique des noms
  - Par ordre alphabétique des prénoms
  - Ou tout autre ordre

Un bel exemple d'organisation du code avec des interfaces : les collections

Hiérarchie des interfaces pour les Collections



Détail sur l'organisation des collections

- **Collection** : interface qui est implémentée par la plupart des objets qui gèrent des collections
  - pas de classes qui implémentent directement l'interface Collection
- **Map** : interface qui définit des collections associatives, sous la forme clé/valeur
  - HashTable, HashMap : classe qui implémente l'interface Map
- **Set** : interface pour des ensembles sans doublons

  HashSet : classe qui implémente l'interface Set

Un exemple d'organisation du code avec des interfaces : les collections

List : interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément

- classes implémentant List

  - classe ArrayList   implémentation des listes par tableaux extensibles
  - Classe LinkedList : implémentation des listes par chaînage

La classe outil *Collections* : très , très pratique !

- La classe outil *Collections* fournit **des méthodes de classe (static)**
  - Object max(Collection c) : renvoie le plus grand élément de la Collection c
  - Object min(Collection c) : renvoie le plus petit élément de la collection c selon l'ordre naturel des éléments
  - void reverse(List) inverse l'ordre de la liste fournie en paramètre
  - void shuffle(List) réordonne tous les éléments de la liste de façon aléatoire
  - void sort(List) trie la liste dans un ordre ascendant selon l'ordre naturel des éléments
  - replaceAll(List l, Object anc, Object nouv) remplace dans l anc par nouv

```
ArrayList<String> list = new ArrayList<String>();
 list.add("Bart");
 list.add("Lisa");
 list.add("Marge");
 list.add("Barney");
 list.add("Homer");
 list.add("Maggie");

// le premier

// remplacer Homer par Anne

// trier la liste

// mettre la liste à l'envers
```