

R1.01

INITIATION AU DÉVELOPPEMENT






Cours 6, partie 1 : – Notion d'ordre

Hervé Blanchon & Anne Lejeune

Université Grenoble Alpes

IUT 2 – Département Informatique

Sommaire

-  Ordre sur les types primitifs et classes enveloppes associées
-  Ordre sur la classe **String**
-  Ordre naturel sur une classe définie par le programmeur
-  Ordre naturel sur la classe **Note**
-  Opérateurs **==** et **!=** sur des d'objets

ORDRE SUR LES TYPES PRIMITIFS ET LES CLASSES ENVELOPPES ASSOCIÉES

Comparaison de valeurs de types primitifs


 On peut comparer deux valeurs de types primitifs...

 byte, short, int, long, float, double, char

 ...avec les opérateurs suivants :

 ==, !=, <, <=, >, >=

 Ces types sont munis d'un ordre naturel

 On peut aussi utiliser ces opérateurs sur les classes enveloppes associées (unboxing)

ORDRE SUR LA CLASSE `String`

Comparaison de Strings



La classe String fournit quatre méthodes de comparaison :

Type du résultat	Méthode	Description
int	<code>compareTo(String anotherStr)</code>	Compare deux chaînes lexicographiquement, sensible à la casse. Le résultat est – négatif si cette chaîne précède strictement lexicographiquement <code>anotherStr</code> – positif si cette chaîne suit strictement lexicographiquement <code>anotherStr</code> – 0 si cette chaîne est lexicographiquement égale à <code>anotherStr</code>
int	<code>compareToIgnoreCase(String anotherStr)</code>	Compare deux chaînes lexicographiquement, en ignorant les différences de casse. Le résultat est – négatif si cette chaîne précède strictement lexicographiquement <code>anotherStr</code> – positif si cette chaîne suit strictement lexicographiquement <code>anotherStr</code> – 0 si cette chaîne est lexicographiquement égale à <code>anotherStr</code>
boolean	<code>equals(Object anObject)</code>	Compare cette chaîne à l'objet spécifié, sensible à la casse.
boolean	<code>equalsIgnoreCase(String anotherStr)</code>	Compare cette chaîne à <code>anotherStr</code> , en ignorant les différences de casse.

compareTo() sur la classe String

Code






```
public class ComparaisonString {  
  
    public static void main(String[] args) {  
        String s1 = "je suis une chaîne";  
        String s2 = "je suis une autre chaîne";  
        String s3 = "Je suis une chaîne";  
  
        System.out.println("s1.compareTo(s1) : " + s1.compareTo(s1));  
        System.out.println("s1.compareTo(s2) : " + s1.compareTo(s2));  
        System.out.println("s2.compareTo(s1) : " + s2.compareTo(s1));  
        System.out.println("s1.compareTo(s3) : " + s1.compareTo(s3));  
        System.out.println("s3.compareTo(s1) : " + s3.compareTo(s1));  
    }  
}
```

Trace

```
s1.compareTo(s1) : 0          // lexicographiquement ...  
                             // ... s1 est égale à s1  
  
s1.compareTo(s2) : 2          // ... s1 est supérieure à s2  
s2.compareTo(s1) : -2         // ... s2 est inférieure à s1  
  
s1.compareTo(s3) : 32         // ... s1 est supérieure à s3  
s3.compareTo(s1) : -32        // ... s3 est inférieure à s1
```

ORDRE NATUREL SUR UNE CLASSE DÉFINIE PAR LE PROGRAMMEUR


Comparer deux instances d'une classe qcq

-  En Java, il y a plusieurs moyens de mettre en place la comparaison de deux objets instances d'une même classe.
 -  cela revient à créer des relations d'ordre
-  En R1.01 nous allons nous intéresser uniquement à la relation d'ordre naturelle (ordre naturel)
 -  nous allons montrer, sans l'expliquer, comment doter une classe d'un ordre naturel
-  En R2.01, vous verrez que d'autres ordres sont possibles et vous comprendrez la syntaxe que l'on vous propose

Doter MaClasse d'un ordre naturel

 MaClasse va implanter l'interface Comparable comme suit :

 vous verrez en R2.01 ce qu'est une interface

 ajout de `implements Comparable<MaClasse>` dans l'entête de définition de la classe MaClasse


 ajout d'une méthode publique `compareTo(MaClasse o)` qui retourne un entier

 cette méthode sera précédée de l'annotation `@Override`


 MaClasse :

```
public class MaClasse implements Comparable<MaClasse> {  
  
    ...    // attributs, constructeurs et méthodes  
  
    @Override  
    public int compareTo(MaClasse o) {  
        ...    // code de la méthode compareTo  
    }  
}
```


Définir un ordre naturel sur MaClasse

 Pour définir un ordre naturel sur MaClasse, il va falloir définir celui-ci en utilisant les valeurs des attributs de MaClasse

 Par exemple

 sur la classe Note avec un attribut valeur, on va dire que c'est l'attribut valeur qui permet de définir l'ordre naturel

 ordre sur valeur

 sur la classe Etudiant avec des attributs nom et prenom, on va dire que ce sont les deux attributs nom et prenom qui permettent de définir l'ordre naturel

 ordre sur le nom et pour un même nom sur le prenom

Méthode compareTo() de MaClasse


 Entête :


 `int compareTo(MaClasse o)`


 Utilisation :

 `objetDeMaClasse.compareTo(autreObjetDeMaClasse)`

 Valeur à retourner :

 un **entier négatif** si **cet objet** (`objetDeMaClasse`) est **strictement inférieur** à l'objet (`autreObjetDeMaClasse`) fournit en paramètre

 un **entier positif** si **cet objet** (`objetDeMaClasse`) est **strictement supérieur** à l'objet (`autreObjetDeMaClasse`) fourni en paramètre

 **zéro (0)** si **cet objet** (`objetDeMaClasse`) est **égal** à l'objet (`autreObjetDeMaClasse`) fourni en paramètre

ORDRE NATUREL SUR LA CLASSE Note

Classe Note sans ordre naturel

```
public class Note {  
  
    private final float MAX = 20.0f;  
    private final float MIN = 0.0f;  
  
    private float valeur;  
  
    public Note(float valeur) {  
        if (valeur >= MIN & valeur <= MAX) {  
            this.valeur = valeur;  
        } else {  
            // pour éviter qu'un appel du constructeur initialise  
            // l'attribut valeur avec une valeur illégale  
            this.valeur = MIN;  
        }  
    }  
  
    public float getValeur() {  
        return valeur;  
    }  
  
    @Override  
    public String toString() {  
        return String.valueOf(valeur);  
    }  
}
```

Classe Note avec un ordre naturel

🗺 La classe est maintenant redéfinie comme suit :

```
public class Note implements Comparable<Note> {  
    ...    // attributs, constructeurs et méthodes  
  
    @Override  
    public int compareTo(Note o) {  
        ...    // code de la méthode compareTo  
               // cf. planche suivante  
    }  
}
```

Méthode compareTo() de Note

Proposition

-  on peut choisir que résultat soit -1, 0, 1 selon que la valeur de **cette Note** est respectivement inférieure, égale ou supérieure à la valeur de **o**

```
@Override
public int compareTo(Note o) {
    // comparer explicitement la valeur de cette Note
    // et la valeur de o
    if (this.valeur < o.valeur) {
        return -1; // cette Note est inférieure à o
    } else if (this.valeur > o.valeur) {
        return 1; // cette Note est supérieure à o
    } else {
        // this.valeur = o.valeur
        return 0; // cette Note est égale à o
    }
}
```


Utilisation de compareTo sur Note









Code	1	<code>public class Note_Main {</code>
	2	
	3	<code> public static void main(String[] args) {</code>
	4	
	5	<code> Note note1 = new Note(10.25f);</code>
	6	<code> Note note2 = new Note(10.255f);</code>
	7	
	8	<code> System.out.println("valeur de note1 : " + note1.getValeur() +</code>
	9	<code> " ; valeur de note2 : " + note2.getValeur());</code>
	10	
	11	<code> System.out.println("note1.compareTo(note2) : " +</code>
	12	<code> note1.compareTo(note2));</code>
	13	
	14	<code> System.out.println("note2.compareTo(note1) : " +</code>
	15	<code> note2.compareTo(note1));</code>
	16	<code> System.out.println("note2.compareTo(note2) : " +</code>
Trace	8	valeur de note1 : 10.25 ; valeur de note2 : 10.255
	10	note1.compareTo(note2) : -1
	12	note2.compareTo(note1) : 1
	14	note2.compareTo(note2) : 0

Se rappeler que :

- une variable ou un paramètre formel de type primitif contiennent une valeur
- une variable ou un paramètre formel de type Classe contiennent un pointeur vers un objet

OPÉRATEURS == ET != SUR DES OBJETS

Bilan

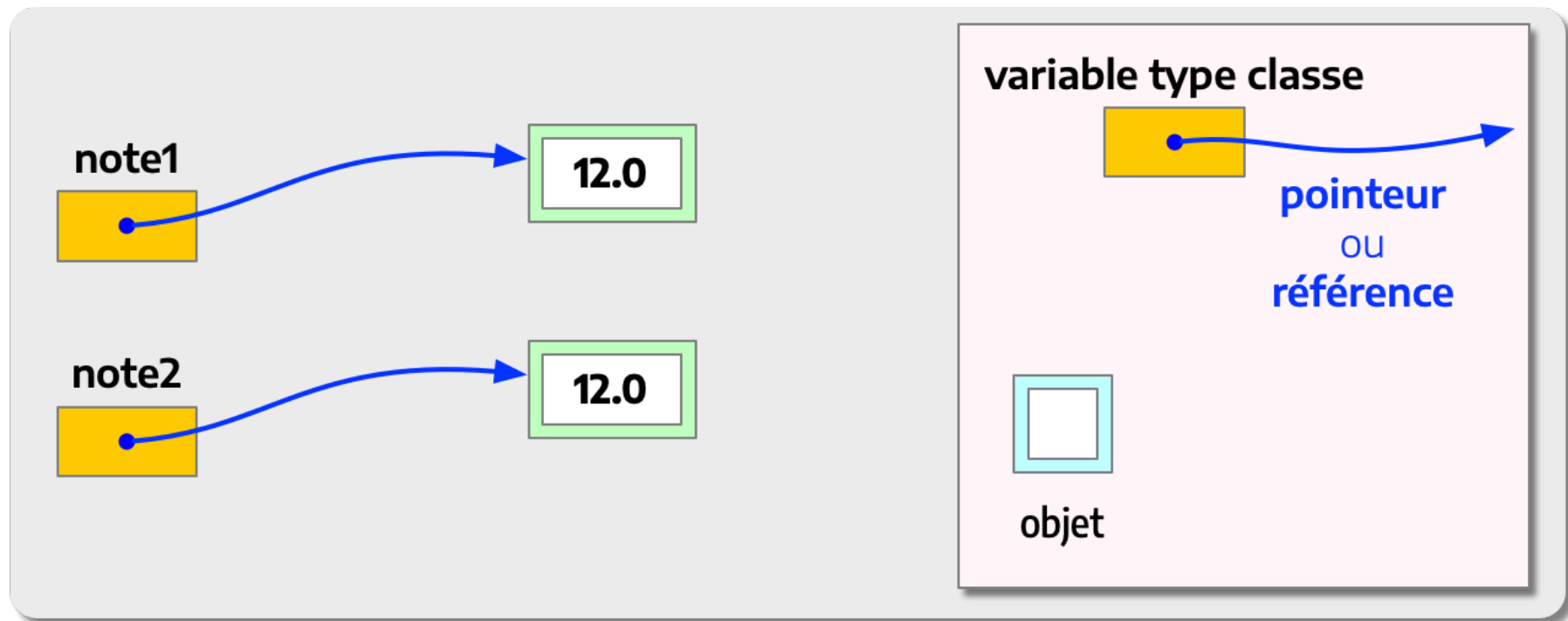
-  Doter une classe d'un ordre naturel permet de comparer deux éléments en fonction de la valeur des attributs pertinents pour cet ordre
-  La méthode `compareTo` permet de vérifier :
 -  l'infériorité au sens strict sur les attributs pertinents (< 0)
 -  la supériorité au sens strict sur les attributs pertinents (> 0)
 -  l'égalité sur les attributs pertinents ($= 0$)
-  En utilisant ≤ 0 ou ≥ 0 avec `compareTo` on a aussi :
 -  l'infériorité au sens large
 -  la supériorité au sens large

Soit le code suivant :

```
1 public class EgalEtDifferentSurNotes {
2
3     public static void main(String[] args) {
4
5         Note note1 = new Note(12f);
6         Note note2 = new Note(12f);
7
8         System.out.println("note1 : " + note1 + " ; note2 : " + note2);
9         System.out.println("note1.getValeur() : " + note1.getValeur() +
10             " ; note2.getValeur() : " + note2.getValeur());
11         System.out.println("note1.compareTo(note2) -> " +
12             note1.compareTo(note2));
13         System.out.println("note1 == note2 ? -> " + (note1 == note2));
14         System.out.println("note1 != note2 ? -> " + (note1 != note2));
15
16         note2 = note1;
17
18         System.out.println("note1 : " + note1 + " ; note2 : " + note2);
19         System.out.println("note1 == note2 ? -> " + (note1 == note2));
20         System.out.println("note1 != note2 ? -> " + (note1 != note2));
21     }
22 }
```

Effet sur la mémoire

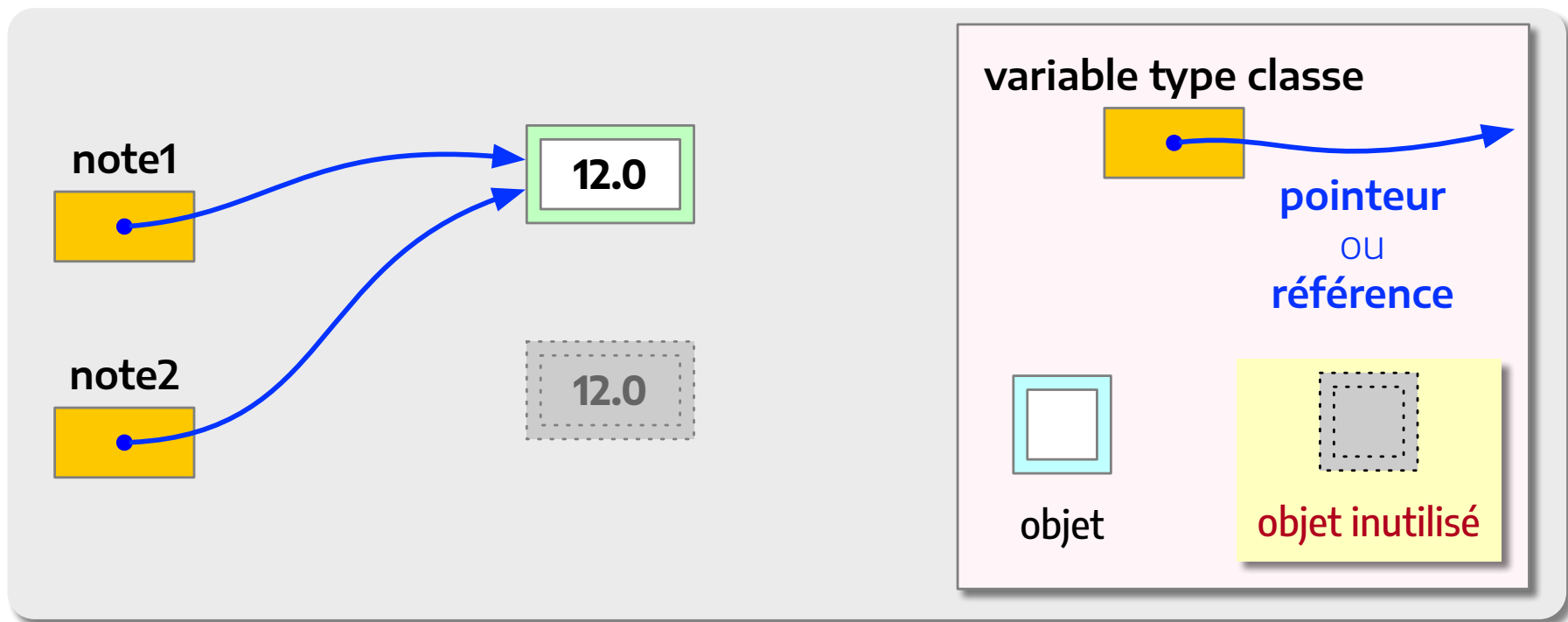
```
5 Note note1 = new Note(12f);  
6 Note note2 = new Note(12f);
```



- note1 et note2 pointent sur des objets différents
- les références sont différentes
- ces objets différents contiennent la même valeur

Effet sur la mémoire

```
16 note2 = note1;
```



■ `note1` et `note2` pointent sur le même objet

■ les références sont égales

Trace expliquée

```
8 // affichage de variables instances de la classe note avec println
// en simplifiant grandement, on a la classe et l'adresse de l'objet de type Note
note1 : Note@53d8d10a ; note2 : Note@e9e54c2
9 et 10 // affichage des valeurs contenues dans les deux instances note1 et note2
note1.getValeur() : 12.0 ; note2.getValeur() : 12.0
// utilisation de compareTo pour comparer deux Notes,
// les valeurs sont les mêmes, il est normal d'obtenir 0 avec compareTo
11 et 12 note1.compareTo(note2) -> 0
// == et != comparent le contenu des variables note1 et note2
// les deux variables contiennent des références (adresses) différentes (53d8d10a, e9e54c2)
13 note1 == note2 ? -> false
14 note1 != note2 ? -> true
// instruction de la ligne 16 : note2 = note1;
// la valeur (l'adresse) contenue dans la variable note1 est rangée dans note2
// note1 et note2 contiennent donc la même référence (même contenu, adresse)
18 note1 : Note@53d8d10a ; note2 : Note@53d8d10a
// == et != comparent le contenu des variables note1 et note2
// les deux variables contiennent des références (adresses) identiques (53d8d10a, 53d8d10a)
19 note1 == note2 ? -> true
20 note1 != note2 ? -> false
```

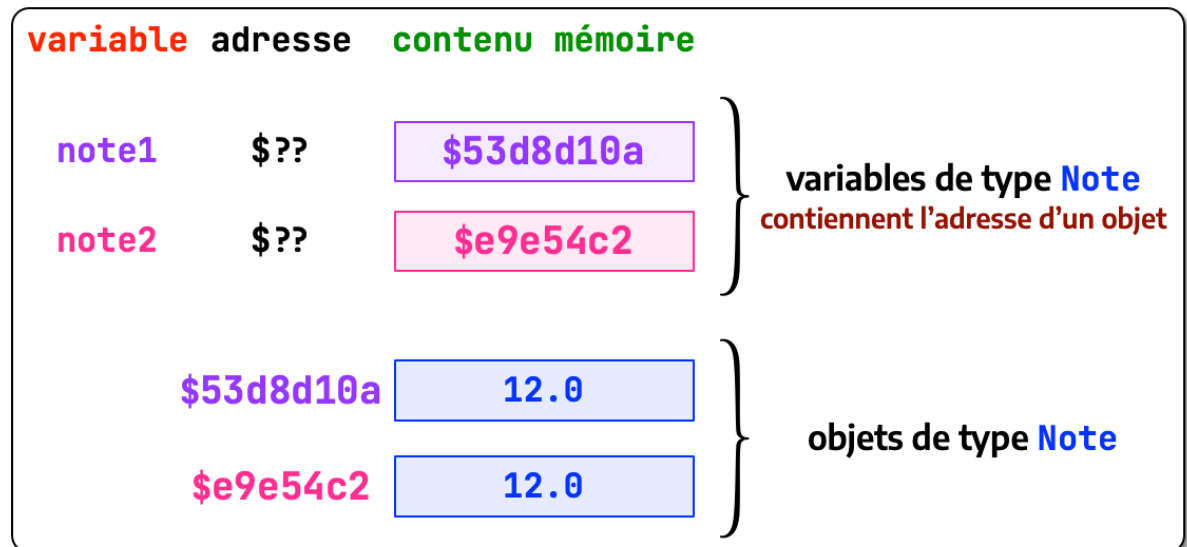
Effet sur la mémoire (autre vision)

Déclaration et construction

```
Note note1 = new Note(12f);
```


```
Note note2 = new Note(12f);
```

 les deux variables pointent sur des objets différents



Affectation

```
note2 = note1;
```

 les deux variables pointent maintenant sur le même objet

