R2.01 : cours n°2

I.Borne

1. Types primitifs, objets, classes wrappers
2. Visibilité des éléments d'une classe et des variables
3. Les chaînes de caractères : la classe String
4 . Rappel sur les opérateurs logiques
5. Documentation Javadoc

1- Rappel des types primitifs de Java

- *char* : type caractère ('1', '?', '(', ...)
- *byte* : type entier sur 8 bits
- *short* : type entier sur 16 bits
- *int* : type entier sur 32 bits
- *long* : type entier sur 64 bits
- *float* : type réel sur 32 bits
- *boolean* : type booleen (true, false)
- *double* : type réel sur 64 bits

Limites et valeurs par défauts des types primitifs

	min	max	défaut
byte	-128	127	0
short	-32 768	32 767	0
int	-2 147 483 648	2 147 483 647	0
long			0L
float			0.0f
double			0.0d
boolean			false
char	'\u0000' soit 0	'\uffff' soit 65 535	'\u0000'

Objets versus types primitifs

- Dans un langage à objets tout est objet.
- *Java est un langage où presque tout est objet.*
- *Les objets se manipulent **toujours** par référence.*
- *Une référence, c'est l'adresse de l'objet en mémoire.*

Les types primitifs NE SONT PAS des objets => Ils ne se manipulent PAS par référence mais par valeur.

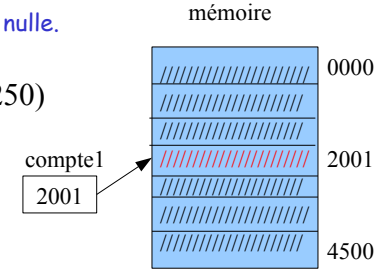
La notion de référence

CompteBancaire compte1 ;

Déclaration d'une variable compte1 qui contiendra la référence (adresse) d'un objet de type CompteBancaire.
A la déclaration, la référence est nulle.

compte1 = new CompteBancaire(250)

La variable compte1 « pointe » maintenant sur un objet de type CompteBancaire en mémoire.

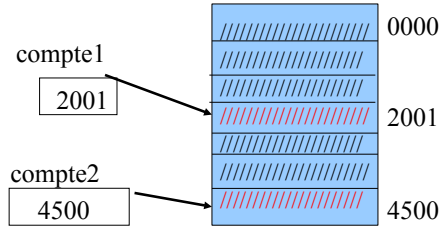


Comparer des références

```
CompteBancaire compte1 = new CompteBancaire(248);
CompteBancaire compte2 = new CompteBancaire(250);
```

Le test d'égalité des deux objets c1 et c2
revient à tester l'égalité des deux références

```
compte1 == compte2
      FAUX
```



L'affectation

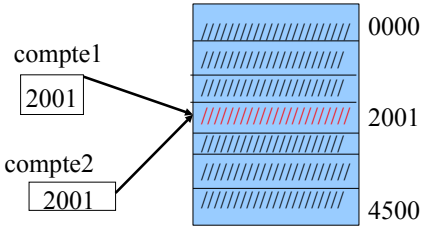
```
CompteBancaire compte1, compte2;
compte1 = new CompteBancaire(248);
compte2 = compte1;
```

L'affectation est possible si les deux objets sont de même type

Le test d'égalité répond vrai.

```
compte1 == compte2

      VRAI
```



Classes *Wrapper*

- A chaque type de donnée primitif Java fournit une classe *wrapper* qui permet à un type primitif d'être représenté par un objet.

Type primitif	Classe wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Conversion de données

- Les classes **wrapper** peuvent être utilisées pour des conversions de données.
- Conversion d'une chaîne de caractères en un entier :

```
int n = Integer.parseInt ("125");
```


2. Visibilité

- des déclarations
- des attributs
- des méthodes
- des constructeurs
- portée des variables

Visibilité des déclarations en Java

- Les visibilités aident à respecter les principes d'encapsulation des objets.
- Une classe est généralement déclarée public

```
public class Carte {
      ...
 }
```

les noms des classes commencent par une majuscule

Visibilité des attributs d'instance

- Les attributs d'instance (ou variables d'instance) doivent être privés, et ne sont visibles qu'à l'intérieur de la classe.
- Ils sont invisibles à l'extérieur de la classe
- Leur nom commence par une minuscule.
- En java ils sont déclarés « private »

```
public class Carte {
            private int rang;
            private int couleur;
            private boolean faceDessus;
        ...
 }
```

Visibilité des attributs d'une classe : règle 1

Tous les attributs et méthodes déclarés dans une classe sont visibles partout dans la classe et donc utilisables depuis le code de toutes les méthodes de la classe.

Les accesseurs et modificateurs

- Les accesseurs et modificateurs des attributs d'instance permettent d'accéder et de modifier l'état de l'objet par envoi de messages à l'intérieur et à l'extérieur de la classe.

```
public class Carte {
  private int rang;
                ...
  // accesseurs
  public int getRang() {
       return this.rang;
  }
  // modificateurs
  public void setRang(int r) {
       this.rang = r;
  }
}
```


Visibilité des constructeurs

- Les constructeurs sont des méthodes particulières qui permettent de construire les objets/instances d'une classe.
- Ils ont toujours le même nom que la classe et n'ont pas de type de retour.
- Ils sont appelés automatiquement lors de la création des objets par l'opérateur *new*.
- Les constructeurs généralement sont des méthodes publiques.

Visibilité des méthodes d'instance

Les principes sont :

Éviter d'exposer les détails d'implémentation d'une classe.

Définir une interface publique et cacher l'implémentation.

- Les méthodes déclarées « public » seront visibles à l'intérieur et à l'extérieur de la classe.
- Les méthodes déclarées « private » ne seront visibles qu'à l'intérieur de la classe et sont des méthodes qui sont utilisées par les autres méthodes de la classe.

Autre exemple de visibilité des attributs d'une classe

```
public class Cercle {
   private double rayon;
        ...
   public double getRayon() {
        return this.rayon;
   }
   public double circonference() {
        double res;
        res = 2*3.1416* this.rayon;
        return res;
   }
}
```


Visibilité des variable : règle 2

Les **variables locales** définies à l'intérieur d'un bloc ne sont visibles que dans le bloc où elles sont déclarées.

- Un bloc est délimité par des accolades : { }
  - bloc de la classe : attributs
  - bloc d'une méthode : variables de travail
  - bloc d'une structure de contrôle : variables de travail

Du bon usage de la visibilité

Les variables de travail doivent être **locales** dans la méthode où elles sont utilisées.

Exemple : mauvaise solution, pourquoi ?

```
public class Cercle {
   private double rayon ;
   private double res;

   public double surface() {

        if (this.rayon > 0) {
             double res;
             res =3.1415*r*r;
        }
       return res;
   }
}
```

Exemple (mauvaise solution)

```
public class Cercle {
   private double rayon ;
   private double res;

   public double surface() {

       if (this.rayon > 0) {
           double res;
           res =3.1415*r*r;
       }
      return res;
   }
}
```

Deux variables res sont définies dans des blocs différents

Horreur !

La variable res n'a pas à avoir le statut d'un attribut d'instance

Exemple (bonne solution)

```
public class Cercle {
   private double rayon ;

   public double surface() {
       double res ;
       if (this.rayon > 0) {
           res =3.1415*r*r;
       }
      return res;
   }
}
```

La variable locale `res` est définie à l 'intérieur du bloc de la méthode `surface`.

Exemple avec des erreurs de portée de variables

```
public class TestRacineCarree {          // mauvaise version
  public static void main (String[] args) {
*  int i = 99;
   double racineCarree = Math.sqrt(i);
   System.out.println("la racine carrée de " + i +
        " est " + racineCarree);
*  for (int i = 1; i<= 10; i++)  {
*     double racineCarree = Math.sqrt(i);
*     double carre = racineCarree * racineCarree;
      System.out.println("la racine carrée de " + i
           + " est " + racineCarree);
   }  //-- for
* System.out.println("la valeur finale du carre = "
                    +  carre);
}
```


La bonne version

```
public class TestRacineCarree {        // bonne version
 public static void main (String[] args) {
   int j= 99;
  double racineCarree = Math.sqrt(j);
  System.out.println("la racine carrée de " + j
                          + " est " + racineCarree);
  for (int i = 1; i<= 10; i++)  {
      racineCarree = Math.sqrt(i);
      double carre = racineCarree * racineCarree;
      System.out.println("la racine carrée de " + i
                        + " est " + racineCarree);
      System.out.println("la mise au carre donne " +
                        carre);
  }  //-- for
}
```

Première erreur : attention aux noms des variables et des index

```
   int i = 99;                    // mauvaise version
   ...
    for (int i = 1; i<= 10; i++) {    // conflit
    }  //-- for
   ...
---------------------------------------
   int i = 99;                    // version légale MAIS à éviter
   ...
   for (i = 1; i<= 10; i++)   {
   }  //-- for
   ...
```


Utiliser des indices différents

// version légale à préférer

```
   int j = 99;
   ...
   for (int i = 1; i<= 10; i++)   {
   }
```

L'indice de la boucle est à déclarer DANS la boucle

Deuxième erreur (mauvaise version)

```
int i = 99;
double racineCarree = Math.sqrt(i);
...
for (i = 1; i<= 10; i++)   {
    double racineCarree = Math.sqrt(i);
...
}  //-- for
```


Deuxième erreur (bonne version)

```
int j = 99;
double racineCarree = Math.sqrt(j);
...
for (int i = 1; i<= 10; i++)   {
    racineCarree = Math.sqrt(i);
...
}  //-- for
```

3 - Les chaînes de caractères

- La classe String (java.lang.String) représente des chaînes de caractères.

```
String  chaine = "abc";
```

est équivalent à

```
char donnee[] = {'a', 'b', 'c'};
String chaine = new String(donnee);
```

- Les objets chaînes sont constants.

```
String ch1 = "bonjour"
```

- **ch1** est une variable qui pointe sur un objet qui ne pourra pas être modifié.

Manipulation des chaînes

- Retourner un nouvel objet String qui est une sous-chaîne de l'objet receveur

```
String c ="abc".substring(2,3);
```

- Concaténer des chaînes et obtenir une nouvelle chaîne :

```
String ch1 = "bonjour" ;
String ch2 = " le monde";
String ch3 = ch1.concat(ch2) ;
```

On peut écrire également :

```
String ch3 = ch1+ch2 ;   // "bonjour le monde"
```


La comparaison de chaînes de caractères

```
String a, b ;
a = new String("toto") ;
b = new String("toto") ;

"toto "= "toto" Mais a#b
```

```
String a, b ;
a = "toto" ;
b = "toto" ;

"toto "= "toto" ET a==b
```

Pour comparer le contenu de deux `String` :

Ne jamais écrire :

~~a==b~~ car on compare des références !!

ce qu'il faut faire : utiliser **`boolean equals(Object)`**

`a.equals(b)` : renvoie `true` si le contenu de `a` est égal au contenu de `b`, sinon `false`.

Opérateurs logiques

o1	o2	o1 && o2	o1 || o2	o1 ^ o2	!o1
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

5 - Documentation avec JavaDoc

Les commentaires en Java

- Le commentaire dans un code Java est de deux types :
- Le commentaire qui explique un détail d'implémentation

```
// ces lignes sont mises en commentaire
// et ne seront pas compilées
/* ces lignes sont mises en commentaire
   et ne seront pas compilées non plus
   etc… etc… etc… etc… etc… etc…
   etc… etc… etc… etc… etc… etc… */
```

- Les commentaires de documentation embarquée JavaDoc.
  Commentaire qui sera extrait pour être transformé en page HTML.

```
/** ces lignes sont destinées à compléter
 * le code d'une documentation qui
 * explique à quoi sert la classe, la
 * méthode etc…
 */
```


Que documenter ? (3 niveaux)

- pour commenter le rôle général de la classe

```
/**
  * Le type Duree permet la manipulation …
  * etc.
  */
public class Duree {
     ...
}
```

- pour commenter le rôle d'un attribut de la classe

```
public class Duree {
/* La durée s'exprime en millisecondes
*/
     private long leTemps;
     ...
}
```


Que documenter ? (suite)

- pour commenter le rôle d'une méthode de la classe

```
public class Duree {
      ...
   /**
    *  Effectue une comparaison entre la
    *  durée courante et une autre durée.
    *  ...
    */
   public int compareA (Duree uneDuree) {
         ...
   }
}
```

Comment documenter ?

!! Le commentaire JavaDoc doit se placer **juste avant** la classe, l'attribut ou la méthode.

```
/**
  * Le type Duree permet la manipulation …
  *  etc.
*/
import java.util.ArrayList;

public class Machin {
     ...
}
```

Erreur de placement de l'import

Documenter le rôle d'une classe

Pour documenter le rôle d'une classe, javaDoc reconnaît un certain nombre de balises (tags) qui lui sont propres.
Nous utiliserons au moins :

@author « l'auteur de la classe »
@version « numéro de version de la classe »

```
/**
  * Le type Duree permet la manipulation etc.
  *
  * @author Jacques Dupond
  * @version 1.1.0
  */
public class Duree {
     ...
}
```


Documenter le rôle d'une méthode

Les balises JavaDoc spécifiques aux méthodes sont :

@param «nomParamètre» : «description du paramètre»

@return «description du type retourné»

@throws «type d'exception» «description du cas»

Exemple de documentation d'une méthode

```
/**
  * Effectue une comparaison entre la durée courante
  * et une autre durée.
  * @param  autreDuree : durée à comparer à la durée
    courante.
  * @return  une valeur entière avec la signification suivante
*/
public int compareA ( Duree autreDuree ) { … }
```

Production des pages HTML

Pour générer les pages HTML de documentation d'une classe, taper la commande :

```
javadoc [options] chemin/MyClass.java

[options] :
    -d cheminRépertoireHTML
    -version
    -author
    -private
    -...
```

Exemple à partir d'un répertoire ww:

```
javadoc -private -author -d ../doc ../source/*.java
```


Format des pages HTML produites

1. La description de la classe

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Duree

```
java.lang.Object
  |
  +--Duree
```

public class **Duree**
extends java.lang.Object

Description

Le type "Duree" permet la manipulation d'intervalles de temps : conversion en JHMS, addition de durées, comparaison entre durées.

Format des pages HTML produites

2. La description résumée des attributs et méthodes

Field Summary

private long	**leTemps** la durée s'exprime en millisecondes

Constructor Summary

Duree(Duree autreDuree) Constructeur qui clone une Duree existante.
Duree(int heures, int minutes, int secondes) Constructeur de Duree avec initialisation.
Duree(long millisec) Constructeur de Duree avec initialisation.

Method Summary

void	**ajoute**(Duree uneDuree) Modificateur qui ajoute une durée à la durée courante.
int	**compareA**(Duree autreDuree) Effectue une comparaison entre la durée courante et une autre durée.
private int[]	**enJHMS**() Effectue un découpage de la durée en intervalles (jours, heures, minutes, secondes, millisecondes).
java.lang.String	**enTexte**(char mode) Renvoie une image textuelle de la durée courante.
long	**getLeTemps**() Accesseur qui retourne la valeur de la durée en millisecondes.

Format des pages HTML produites

3. La description détaillée des attributs et méthodes

Field Detail

leTemps

```
private long leTemps
```

la durée s'exprime en millisecondes

Constructor Detail

Duree

```
public Duree(Duree autreDuree)
```

Constructeur qui clone une Duree existante.
Parameters:
autreDuree - autre durée à copier

Duree

```
public Duree(int heures,
             int minutes,
             int secondes)
```

Constructeur de Duree avec initialisation.
Parameters:
heures - nombre d'heures
minutes - nombre de minutes
secondes - nombre de secondes

Initialisation d'une durée par un nombre d'heures, minutes et secondes.

Format des pages HTML produites

Method Detail

getLeTemps

```
public long getLeTemps()
```

Accesseur qui retourne la valeur de la durée en millisecondes.

Returns:

la valeur actuelle de la durée en millisecondes

---

compareA

```
public int compareA(Duree autreDuree)
```

Effectue une comparaison entre la durée courante et une autre durée.

Parameters:

`autreDuree` - durée à comparer à la durée courante

Returns:

Une valeur entière avec la signification suivante :

- -1 : si la durée courante est plus petite que l'argument
- 0 : si les deux durées sont égales
- +1 : si la durée courante est plus grande que l'argument
