Data types and Stings

(Working with Java Data Types and String APIs (Java SE 11 Developer Certification 1Z0-819))

Table des matières

I. Understanding Primitive Types and Variables

- ***Introduction :***

Il existe 8 types primitifs :

Boolean	Byte
Short	Int
Long	Double
Float	Char

Field vs Local Variables :

a) Fields (attributs en français) :
- Sont déclarés au niveau de la classe.
- Elles peuvent être **static**.
- Elles sont initialisées avec une valeur par défaut (tableau).

b) Local variables :
- Sont déclarés dans des méthodes ou des blocs de codes.
- Elles ne sont **jamais** static.
- Elles **doivent** être initialisé !

Type	Size	Default
boolean	1 bit	False
Char	16 bit	'\u0000'
Byte	8 bits	0
Short	16 bits	0
Int	32 bits	0
Long	64 bits	0
Float	32 bits	0.0
Double	64 bits	0.0

Figure 1valeur par défaut

Les deux stockent des valeurs **réelles** tandis que les autres stockent des **pointeurs** vers des valeurs (exemple un String).

Ici le code ne compilera pas car la variable est locale et a besoin d'être initialisée avec une valeur :

```
public static void main(String[] args) {
    long variable;
}
```

- ***The byte, short, int, and long Primitives :***

Byte (octet en français) = 8bits donc valeurs de -128 à 127

Short = 16 bits donc valeurs de -32768 à 32767

Int = 32 bits et valeurs +-2.1 milliards.

Long = 64 bits et valeurs +-9.2quintillions (signalé par un L en fin de nombre, ce qui indique au compilateur que c'est un type long).

- ***Float and Double :***

Permet l'utilisation de fraction donc de nombre non entier (décimaux).

```
float piFloat = 3.14f;
double piDouble = 3.14;
```

Ils peuvent être aussi bien négatif que positif et Float doit inclure un F en fin de nombre. Les deux arrondissent le dernier chiffre après la virgule s'il y en a trop.

Ils peuvent se voir attribuer des valeurs entières ;

```
float longToFloat = 1111111112222222223L;
```

Le résultat du print sera alors obtenu en notation scientifique s'il y'a trop de chiffre : longToFloat = 1.11111113E18.

- ***Char :***

Un caractère unicode = 16 bits (comme un short).

Utilisation la plus commune : les caractères littéraux.

```
char aLetter = 'A';
```

Si on met un double quote (« ») il devient un String.

On peut l'utiliser pour l'alphabet Unicode, ici le caractère qui a un nombre hexadécimal de 0041.

```
char aUnicode = '\u0041';
```

Dans le programme Java les caractères Unicodes sont stockés sous forme d'Intégrale de sorte que 'a' et '\u0041' sont tous les deux stockés sous la forme du nombre 65.

```
char aNumber = 65;
```

Donc ces trois valeurs char, valent toutes « A ».

Donc, on peut attribuer une valeur à un char : de 0 à 65535 (il ne prend pas les décimaux).

- ***Narrowing, Widening, and Casting***

Widening = attribuer la valeur d'un primitif à un primitif plus grand :

```
byte byteValue = 100;
short shortValue = byteValue;
```

Narrowing = attribuer la valeur d'un primitif à un primitif plus petit. Ce n'est possible que si la valeur n'est pas trop grande pour ce nouveau type. Ici c'est 100 donc cela ne pose pas de problème.

Casting = convertir la variable :

```
intValue = (int) longValue;
shortValue = (short) intValue;
```

Si le nombre est trop grand il sera réduit ce qui donnera un nombre différent. C'est appelé **OVERFLOW** (débordement).

Exemple de résultat :

```
longValue = 9223372036854775807
intValue = -1
```

- ***Underscores in Numeric Literals :***

Pour mieux lire un nombre on ne peut malheureusement pas utiliser les points ou les virgules pour séparer les milliers. Par contre on peut utiliser l'underscore :

```
long worldPopulation = 7_674_000_000L;
```

On peut aussi en mettre un peu partout, **SAUF** devant ou derrière un nombre primitif.
Quelques exemples de **mauvaise** utilisation :

```
short fahrenheit = _-10;
float celsius = -_23.3333f;

float kgInPounds = 2_.20462f;
double poundInKg = 0._453591830542594d;
```

- ***Alternative Number Systems :***

Java prend en compte plusieurs système numérique :

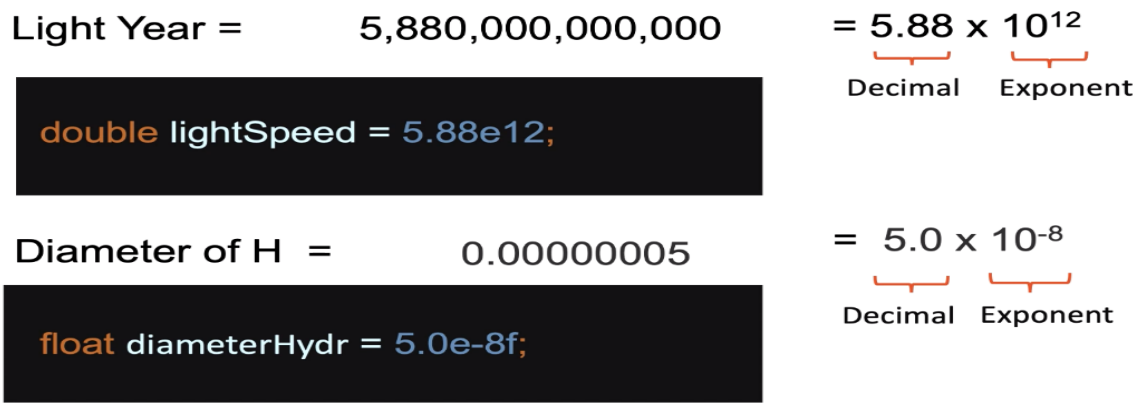
- **Arabe** (base 10)
- **Octal** (base 8) commence avec un 0 non significatif devant.
- **Hexadécimal** (base 16) commence toujours avec 0 et x (min ou maj peu importe).
- **Binaire** (base 2) commence toujours avec 0 et b (minuscule de préférence).

Expression d'une valeur identique en différent système numérique.

```
int w = 100;         // base 10 literal
int x = 0144;        // Octal literal
int y = 0x0064;      // Hexadecimal literal
int z = 0b1100100;   // binary literal
```

Note : on ne peut pas utiliser l'underscore avant ou après le préfixe du système numérique.

- ***Scientific Notation :***



Il faut juste reconnaitre une notation scientifique. Le « e » signifie exposant une quantité de chiffre.

II. Using Operators and Math APIs

- ***Arithmetic Operators and Promotion :***

Types d'opérations :

- Arithmétiques : (+,-,*,/)
- Assignement : (=, +=, *=)
- Relationnelle : (<,>,==)
- Logique : (&,&&,|,||)
- Manipulation de bytes (pas un sujet d'examen)

Opérateur arithmétique et la **promotion numérique** :

Ici, si on initialise un byte de cette manière il aura la valeur 8, c'est donc valable.

```
byte byteResult = 5 + 3; // 8
```

Par contre si on essaie d'additionner deux bytes :

```
byte xByte = 5;
byte yByte = 3;

byte intResult = xByte + yByte;
```

IntelliJ indiquera qu'il y'aura une erreur. C'est parce que Java caste d'office les deux variables en int. En arrière-plan on à ceci, ce qui entraine une erreur de compilation :

```
byteResult = (int)xByte + (int)yByte;
```

Règle : lorsqu'un **BYTE** ou un **SHORT** est utilisé dans une opération arithmétique, il est d'abord converti en int.

Si j'additionne un short et un byte et que je l'affecte à un int, cela sera **correct** :

```
intResult = xShort + yByte;
```

Idem pour les deux suivants quand j'ajoute un long, ou un double dans l'opération, le compilateur converti les valeurs vers la variable qui a le plus grand type après int, ou après float :

```
long longResult = xShort + yInt * zLong; //conversion en long

double doubleResult = xByte + yFloat * zDouble; //conversion en double
```

C'est le principe de la **promotion numérique**.

- ***Pre-and post unary Operator :***

a) **Pre-unary** :

++ incrémentation de 1.

-- décrémentation de 1.

Ils peuvent être utilisés dans des opérations arithmétiques et effectue le changement **AVANT** l'opération :

```
xInt = 4; // 5
int yInt = 7;
int zInt = 3; // 2
int result = ++xInt + yInt + --zInt; // 14
```

Le résultat vaut 14 car 5 +7 + 2.

On peut l'utiliser plusieurs fois sur la même variable dans une seule opération :

```
xInt = 5; // 6 + 6 + 7 = 19
result = ++xInt + xInt + ++xInt; // 19
```

Le résultat vaut 19.

b) **Post-unary** :

Règle : l'incrémentation ou la décrémentation se font **APRES** que l'opération soit exécutée. En d'autres mots d'abord la valeur originelle est utilisée, **ENSUITE** elle est incrémentée.

```
xInt = 1; // 3
result = xInt++ + xInt + xInt++; // 5
//          1       2       2   = 5
```

Pour résumer il y'a trois xInt :

Le premier est évalué à 1, **ENSUITE** on incrémente.

Le deuxième est évalué à 2 et on incrément pas car pas de ++.

Le troisième est aussi évalué à 2 mais incrémenté en fin d'opération. Donc la variable xInt vaudra à la fin de l'opération 3 !

- ***Assignement Operators :***

Principe, on utilise le = pour affecter une valeur :

```
int x = 5;
int y = 3;
```

L'opérateur d'affectation simple peut aussi être utilisé comme une opération elle-même :

```
x = 5;
y = 3;
z = 5 + (y = x + y); // 13
```

Autre exemple :

```
boolean flag = false; // true
z = 0; // 5
if(flag = true){
    z = 5;
}else{
    z = 3;
}
```

Ici « z » vaudra 5, car flag = true, est une opération qui change la valeur de flag en true. Ce n'est donc pas une comparaison (==) et c'est tout à fait valide.

Compound assignement operators :

+=, -=, *=, /=, %=

```
int yInt = 3;
yInt *= yInt;
```

yInt vaudra 9 à la fin de l'opération.

Note que la promotion numérique n'œuvre pas dans ce cas :

Exemple, la première ligne est valide car le compilateur fait en vérité ce qu'il y'a dans la deuxième ligne en arrière-plan.

```
xByte += yDouble;
xByte = (byte)(xByte + yDouble);
```

Pour rappel si je veux faire ceci j'ai une erreur :

```
xByte = xByte + yDouble;
```

Ceci fonctionne aussi car le compilateur cast tout en byte.

```
xByte = 5;
yDouble = 3.0;
float afloat = 5.0f;
long aLong = 10;
short aShort = 3;
xByte += yDouble + afloat * aLong % aShort; // 10
```

- ***Comparaison Operators :***

Règle :

- Les opérateurs relationnels (<,>, <=,>=) ne peuvent être utilisé qu'avec les types numériques.
- Les opérateurs == et !=, peuvent être utilisés avec des nombres primitifs, des références d'objet ou des booléens.
- InstanceOf compare un type à un autre :

```
result = (instanceA instanceof ClassA)
```

- ***Logical Operators :***

&, && , |, ||, ^ , !

a) Single AND (&) :

Les deux opérateurs doivent être **TRUE** pour avoir un résultat true :

```
boolean tru_1 = true;
boolean tru_2 = true;

result = tru_1 & tru_2; // true
```

Si il y'en a un qui est faux, le résultat sera false.

b) Le double AND (&&) ou « short-circuit and » :

Renvoi **TRUE** si les deux opérateurs sont vrais. Est un peu plus rapide que le simple & car s'il voit false dès le premier opérateur, il « court-circuite », s'arrête et renvoi faux d'office.

```
result = fls_2 && tru_2; // false short-circuited
```

c) Or (|) :

Renvoi vrai si un des deux côtés est **TRUE** (comme le single & il doit vérifier les deux cotés de l'opération)

```
result = tru_1 | fls_1; // true
```

d) Double Or (||) :

Idem que le double && avec court-circuit (donc plus rapide) si celui de gauche est **TRUE** :

```
result = tru_1 || tru_2; // true short-circuited
result = fls_2 || tru_2; // true
result = fls_1 || fls_2; // false
```

Attention cas intéressant avec le court-circuit :

```
int xInt = 7; // 8
boolean yBool = false; // false
boolean result = (xInt++ <= 7) || (yBool = true); // true
```

On voit que dans l'opération il y'a une assignation à yBool qui après l'opération devrait valoir true. Or le principe du court-circuit veut qu'on stop l'opération directement si le premier opérateur vaut **TRUE** donc l'affectation n'a pas lieu et yBool reste **FALSE**.

e) L'Or exclusif (^) :

Un des cotés doit être FALSE et l'autre TRUE pour renvoyer TRUE.

```
result = tru_1 ^ tru_2; // false
result = tru_1 ^ fls_1; // true
result = fls_1 ^ fls_2; // false
```

f) L'opérateur not (!) :

Il est unaire et il inverse la valeur du booléen en place :

Exemple :

```
rst = !true_1 ; résultat = false
```

- ***The ternary Operator :***

Seul opérateur qui prend trois opérandes (ndr, ne suis pas certain que la traduction de « opérande » soit exacte) et qui ressemble à un if/else classique :

```
boolean result;
float x = (float)Math.random() * 6;

if(x <= 3){
    result = true;
}else{
    result = false;
}
```

L'opérateur ternaire a besoin de :

- L'assignement : result = (x <= 3)
- Le premier opérande, le test : qui est l'équivalent du if qui devient point d'interrogation (?) et du else qui devient deux points (:)
- Les deux derniers opérandes : reçoivent les valeurs booléennes true et false.

La forme complète sera :

```
result = (x <= 3)? true : false;
```

Il peut fonctionner dans un if ou dans une loop, voir dans une opération classique :

```
if( (x <= 3)? true : false ){
    // do something
}
```

Les deux derniers opérateurs peuvent être de tout type et pas forcément identique.

```
double dValue = ((x <= 3)? 0.0 : 3.14) * 13;
```

- ***Other Operator :***

Moins, plus, divisé et multiplié.

Petits exemples les opérateurs unaires positifs devant un nombre sont implicite donc les deux lignes sont en fait **identiques** :

```
int x = 5 * -3;
int x = +5 * -3;
```

Ne pas les confondre avec les pré-unary operators :

```
x = 5;
y = --x * -3 + ++x;   // -7
```

- ***Order of opération :***

Ordre du haut vers le bas :

Operator	Symbols
Post-Unary	*expr++ expr--*
Pre-Unary	*++expr --expr*
Other Unary	*+expr -expr !expr*
Multiplicative	* / %
Additive	+ -
Relational	< > <= >= instanceof
Equality	== !=
Logical	& ^ |
Logical (short-circuit)	&& ||
Ternary	expr ? expr : expr
Assignment	= += -+ *= /= %=

Peut résumer comme ceci :

A. Pré, post, other unary.
B. Multiplication, Addition.
C. Relationnel
D. Egalité, logique Simple, logique double.
E. Ternaire
F. Assignement.

- ***Math API :***

On va étudier ici 5 méthodes statiques :

1. Random() :

L'écriture la plus basique renverra une valeur double comprise entre 0 et 1 :

```
double result = Math.random();
```

A noter qu'on peut faire un objet de classe random qui nous donnera des méthodes propres.

2. Round() :

Arrondi à l'entier le plus proche, si on lui passe un double il renvoie un long et si on lui passe un float il renvoie un int. La méthode regarde seulement le premier chiffre après la virgule décimale. Si inférieure à 0.5 arrondi vers le bas, si plus vers le haut.

```
double pI = 3.14;
long longX = Math.round(pI); // 3
```

3. Pow() :

Calcul un exposant :

```
double num = 2;
double exp = 3;

double result = Math.pow(num,exp); // 8
```

4. Min() ou Max() :

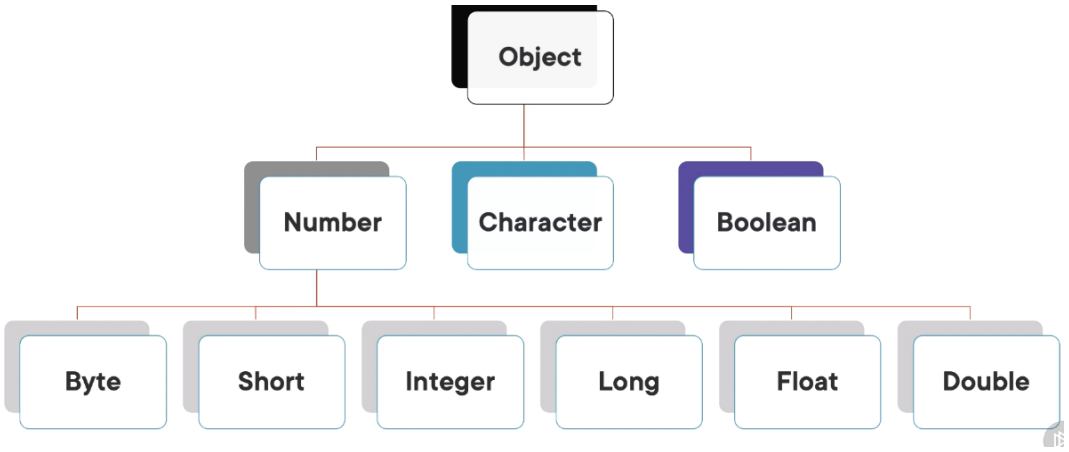
Renvoi la valeur la plus grande ou la plus petite. Il est possible d'utiliser des int, long, double et float.

```
int intA = 3;
int intB = 4;
int intMin = Math.min(intA,intB); // 3
int intMax = Math.max(intA, intB); // 4
```

III. Using Primitive Wrappers :

- ***Conversion method :***

Tous les primitifs du langage ont un « wrapper », une classe correspondante :



Il existe trois types de méthode de conversion :

1. La première (type)**Wrapper.**(type)**Value** convertit une instance du type « number » en l'un des types primitifs souhaité :

```
Integer intWrapper = new Integer(7);

byte   bytVal = intWrapper.byteValue( );   // 7
short  shtVal = intWrapper.shortValue( ); // 7
int    intVal = intWrapper.intValue( );    // 7
long   lngVal = intWrapper.longValue( );   // 7L
float  fltVal = intWrapper.floatValue( ); // 7.0f
double dblVal = intWrapper.doubleValue( );// 7.0
```

Cette méthode ne fonctionne que pour une conversion vers un type plus grand. Si j'essaie vers un type plus petit j'obtiendrai un mauvais résultat :

```
intWrapper = new Integer(200_000);
shtVal = intWrapper.shortValue(); // 3392
```

2. La deuxième, **parse+()** converti une chaine de caractère :

On peut passer d'un String à une classe de type number grace à la méthode parse+(nom du type) :

```
String strNum = "1234";
intVal = Integer.parseInt(strNum); // 1234
```

Le compilateur lancera une erreur si on essaye de convertir un String qui contient :

- Une chaine de charactère, ex : « ABC » vers un number.
- Un décimal vers une mauvaise classe, ex : «3.14» vers un int.
- Un null vers un number.

On peut aussi convertir un String en booléen tant qu'il est écrit correctement :

```
boolean booVal = Boolean.parseBoolean("TRUE"); // true
booVal = Boolean.parseBoolean("false"); // false
```

Par contre si on essaye de convertir autre chose en booléen il renverra d'office FALSE.

```
booVal = Boolean.parseBoolean("troop"); // false
booVal = Boolean.parseBoolean(null);   // false
```

3. La troisième méthode, **valueOf** qui prend un primitif et le renvoie dans un objet « wrapper ». (C'est l'inverse de la première méthode) :

```
Long longWrap = Long.valueOf(23234);         // new Long(23234)
Integer intWrap = Integer.valueOf("23234"); // new Long(23234)
```

- ***Autoboxing and unboxing :***

Créée depuis Java 5 ces méthodes sont censées limiter l'utilisation des méthodes de conversion plus classiques.

Principe : Conversion **automatique** d'un wrapper Class vers **SON** primitif et vice versa.

Exemple, la première ligne est un autoBoxing et la deuxième un unBoxing :

```
Integer integer = 234;
int intPrimitive = integer;
```

A titre de comparaison, les mêmes lignes avec les méthodes de conversion :

```
Integer integer2 = new Integer(234);
int intPrimitive2 = integer2.intValue();
```

L'élargissement de type (**widening**) est pris en charge.
Un exemple avec un wrapper byte qui passe vers une variable primitive de type SHORT.
Idem pour le reste, shortWrapper vers int et etcetera :

```
sValue = byteWrapper;
iValue = shortWrapper;
lValue = intWrapper;
```

On ne **PEUT PAS** « caster », faire un unboxing et un **NARROWING** (rétrécissement) en même temps (byValue est ici une variable de type byte):

```
byValue = (byte) shortWrapper; //pas valide
```

Type	Size	Default
boolean	1 bit	False
Char	16 bit	'\u0000'
Byte	8 bits	0
Short	16 bits	0
Int	32 bits	0
Long	64 bits	0
Float	32 bits	0.0
Double	64 bits	0.0

Idem quand on passe de wrapper en wrapper en rétrécissant :

```
shortWrapper = (short)intWrapper; //pas valide
```

En élargissant c'est accepté, ici deux wrappers :

```
longWrapper = (long)(intWrapper);//valide
```

Un des avantages classe Wrapper est de pouvoir utiliser la valeur **NULL**.

- ***Custom Wrappers :***

Ici les caractéristiques d'un custom wrappers même si cela n'a que peu d'intérêt pour la certification.

Wrapper Class Idioms

Class is final
Instances are immutable
There is a range of values
Read, not write
Includes equals, hashcode, and toString
Helper methods
Serializable and Comparable

```
public final class Volume {

    public final double liters;
    private final static double MIN_LITERS = 0;
    private final static double MAX_LITERS = Double.MAX_VALUE;

    public Volume(double inLiters){
        if( inLiters >= MIN_LITERS ) throw new java.lang.NumberFormatException();

        liters = inLiters;
    }
    public double getVolume(){
        return liters;
    }
    public double inMilliliters(){
        return liters * 1000;
    }
    public double inQuarts(){
        return liters * 1.05669;
    }
    public boolean equals(Object obj){
        if( obj.getClass() != this.getClass() ) return false;
        Volume other = (Volume)obj;
        if(other.getVolume() == this.liters) return true;
        return false;
    }
    public String toString(){
        return Double.toString(liters) + "ltrs";
    }

    public int hashCode(){
        return this.toString().hashCode();
```

```
    }
}
```

IV. Understanding Variable Rules and Scopes :

- ***Fields, local variables Rules and Scope :***

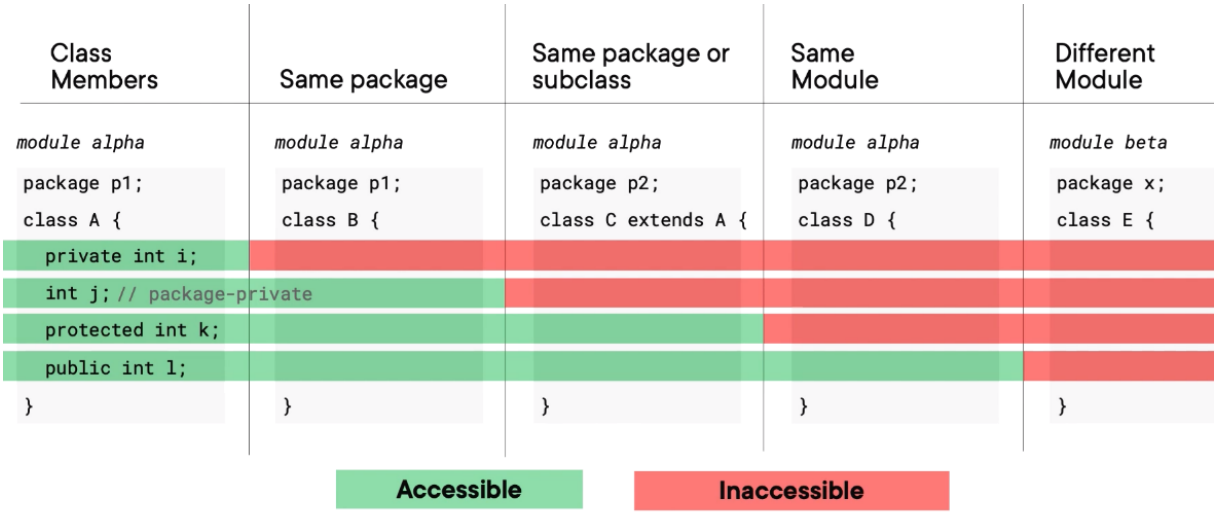
(Rappel du chapitre I partie 1).

- ***Members and member scope :***

Les members (membres) sont soit des fields (attributs), soit des méthodes de classes.

Members access modifier :

Petite précision : un module est un « emballage » qui peut contenir un package.
Comme on le voit, ne pas mettre d'accesseur en créé un par défaut qui est **semi-privé**.



Pour rappel :

Une variable (ou une méthode) de classe STATIC est accessible sans qu'on doive instancier la classe. Exemple : NomDeClasse.méthodeStatic(), ou Math.random().

Si ce n'est pas le cas il faut instancier : Exemple : Random rdn = new Random() ;

- ***Méthode Variable Scope :***

Les variables locales ne sont accessibles que dans le cadre du code dans lequel elles sont déclarées. C'est-à-dire le plus souvent la partie délimitée par des crochets {}.

```
public static void someMethod(int param1, int param2) {
    int localVar0 = 0;
    if (true) {
        int localVar1 = 0;
        if (true) {
            int localVar2 = localVar1;
        }
    } else {
        int localVar1 = 2;
        for (int i = 0; i < 10; i++) {
            int localVar2 = localVar1;
            // more code goes here
        }
    }
    int localVar3 = 3;
    while (true) {
        int localVar4 = param2;
        // more code goes here
       }
}
```

Ici :

Les deux paramètres (param1et2) sont accessibles partout dans la méthode SOMEMETHOD.

Idem pour la LOCALVAR0.

Les variables LOCALVAR1et2et3 ne sont accessibles que dans l'espace délimité par les crochets dans lesquels elles sont définies.

LOCALVAR3 est accessible partout sauf qu'il est déclaré en fin de méthode donc il ne peut-être lu que par les instructions qui suivent (la boucle while ici).

Shadowing (ombrage), il se produit uniquement lorsqu'une variable locale est déclarée avec le **même nom** qu'un attribut static ou d'instance. Ici on déclare deux fois la même variable memberVariable, mais dans des portées (scope) différentes, ce qui écrase les valeurs **SELON** leur champ d'application !!! :

```
public class ShadowingAndScope {
    public static int memberVariable = 200;

    public static void someMethod() {
        out.println(memberVariable); // 200
        int memberVariable = 2;
        out.println(memberVariable); // 2
        out.println(ShadowingAndScope.memberVariable); // 200
    }
}
```

- ***Variable Naming Rules :***

Il y'a des règles qui sont **OBLIGATOIRES** et des conventions qui sont **RECOMMANDEES**.

1. Le nom peut être très long (65000 charactères) donc pas vraiment de limite.
2. Doit être un caractère **ALPHANUMERIQUE** (a->z Maj ou min, 0->9), un signe **DOLLARS** ou un **UNDERSCORE.**
3. Le nom ne peut pas commencer par un chiffre ou avoir un espace.
4. Le nom est « **CASE SENSITIVE** », c'est-à-dire qu'un caractère majuscule n'est pas le même qu'une minuscule.
5. Ils ne peuvent pas être des mots **CLEFS**. (Utilisés par le langage comme « class » ou « public ») sauf si on change le « case ».

Quatre exemples dont deux valides et deux non-valides.

```
int thisI$variaBLEname_thatIS5verylong = 0; // valid
int ___$__$___  = 0; // valid
int thisIsAlso-AVaraibleName = 0; // error
int 1stMoney = 0; // error
```

- ***Variables naming conventions :***

On évite quand même d'utiliser des dollars et des underscore même si c'est permis par le compilateur.

On évite les abréviations et on favorise les noms **complets** et **descriptifs**.

On **débute** toujours le nom par une minuscule.

- ***Local variable Type Inference :***

Le type **VAR** peut contenir aussi bien un primitif qu'une référence. Une fois qu'il est initialisé sont type est **FIXE** et ne peut pas changer. Le **WIDENING** est autorisé.

Dans ce cas-ci var est automatique typé selon les valeurs qu'il reçoit :

```
var y = 1;      // y is an int type
var z = 3.14f; // z is a float type
var mot = "unMot"; //String objet
```

Il ne peut pas être utilisé pour instancier une variable de classe :

Il ne peut pas être STATIC ou être un attribut de classe. Il doit être une variable **LOCALE** donc on ne peut pas l'utiliser comme un paramètre de méthode.

Il doit être initialisé au moment de sa déclaration :

```
var intY ; //compile error
```

Il ne peut pas être null à moins d'être **casté** :

```
var name = null; //pas bon
var name = (String)null; //bon
```

On ne peut pas faire de multi déclaration :

```
//var pInt = 1, qInt = 2, rInt = 3; // compile error
```

Pour résumer :

- ***The Varargs Parameter :***

Paramètre qui peut prends un nombre indéfini d'argument du même type sans qu'on doive les lister individuellement représenté par le type suivi de 3 points et le nom. Ici : int…nums

```
public static void someMethod(int... nums){
    out.println(Arrays.toString(nums));
}
```

Il peut aussi être NULL ou vide.

```
someMethod(null); //valid
someMethod(); // valid
```

Il peut prend un **tableau** comme argument :

```
int[] values = {2,4,5,6};
someMethod(values);
```

On peut créer une méthode avec d'autres paramètre tant que le varargs est en **DERNIER**. Ici j'utilise un String et un double comme paramètre en plus.

```
public static void someMethod(String x, double y, int... nums){
    out.print(x+", "); out.print(y + ", ");
    out.println(Arrays.toString(nums));
}
```

V. Working with Strings, Dates and Times :

• The immutable String :

Un objet String encapsule un **TABLEAU** de caractère. Un fois créé il ne peut pas être modifié (càd ajouter, remplacer ou modifie un caractère). On dit qu'il est **IMMUABLE**.

Ils sont avec les primitifs, les seuls types qui peuvent être initialisé avec des littéraux :

```
String str = "Hello!";
String str2 = "Hello!";
```

Pour améliorer les performances le langage Java fournit ce qu'on appelle un **POOL** de littéraux de String. Donc quand on initialise un string avec un littéral, Java pioche dans la pool plutôt que de créer une nouvelle instance.

Donc si je fais ceci j'obtiens true bien que ce soit deux variables différentes. C'est parce que le double égal compare les références qui sont ici identiques.

```
(str == str2); // true
```

Si on utilise new, on crée une nouvelle instance donc on a une nouvelle référence:

```
String str3 = new String("Hello!");
```

La règle est la même si je crée un String via une méthode (ici subString):

```
String str4 = str.substring(5);
String str5 = str.substring(5);
out.println(str4 == str5); // false
```

• String Concatenation :

Les deux opérateurs pour concaténer sont : + et +=

Ils permettent de créer une nouvelle référence quand ils sont appliqués à un String :

```
String str = "Hi, ";
String str2 = str + "Bob!"; // "Hi, Bob!"
```

On peut aussi joindre des **primitifs** :

```
String pi = "3.14" + 15926;  // "3.1415926"
```

Attention tout de même à l'ordre des opérations, dans cet exemple les deux premiers int sont additionnés mais pas les deux derniers car au moment où on concatène 8 et XYZ il devient un string et continue la concaténation avec un 3 et puis un 5 :

```
String str3 = 5 + 3 + "xyz" + 3 + 5; // "8xyz35"
```

Le plus égal quant à lui donnera cela :

```
String str4 = "Pi is ";
str4 += "3.14";// "Pi is 3.14"
```

- ***String methods 1 :***

Il y'en a bcp mais pour l'examen il ne faut en retenir quelques-unes.

1. **equals**() : vérifie le contenu de deux String et renvoi true ou false:
2. **length**() : renvoi le nombre de caractère (espace et tabulation compris).
3. **UpperCase**() et **lowerCase**() convertissent en majuscule ou minuscule. (elles renvoient un nouvel objet String).
4. **startWith**(« String ») compare la chaine de caractère en argument au début d'une autre chaîne et renvoie true ou false.
5. **endWith**(« String ») idem que précédent mais pour la fin.
6. **Contains**(« String ») vérifie si la chaine en argument est contenue quelque par et renvoie true or false.
7. **trim**() :Supprime tous les espaces blancs d'une chaine. Il ne fonctionne pas bien avec l'ensemble du jeu de caractère Unicode donc pour cela on utilise plutôt la méthode suivante.
8. **strip()** : idem que précédent mais pour tous les caractères Unicode. (Mais n'enlève pas les tabulations, les nouvelles lignes, les retours ou les sauts de pages). Cette méthode à deux itérations, **stripLeading()** et **stripTrailing()** qui se concentrent respectivement sur le début ou la fin du String.

- ***String Methods 2 :***

9. **charAt**(nombre) : permet de rérupèrer sous forme de char, le caractère qui se trouve à l'index donné par le nombre. Le string étant un tableau, le premier caractère est à l'index 0.
10. **indexOf**('char') : renvoi l'index de la première occurrence du caractère en paramètre. Si pas de résultat, renvoi -1. On peut mettre un String comme argument. Il renverra alors l'index de la première lettre de la chaine. On peut luis passer un deuxième argument pour lui dire à partir de quel index chercher. **indexOf(char ou String, int)**.
11. **Substring**(int, int) : renvoi un nouveau string qui est un morceau d'un string. On peut délimiter ce morceau par des int en argument (soit un pour le début, soit deux pour le début et la fin).
12. **replace**(char, char) : remplace tous les caractères passé en premier argument par le deuxième. On peut avoir deux string comme argument (string, string).

Note : on peut enchainer les méthodes :

```
str2 = str.strip().replace("Hi", "Hello").toUpperCase();
```

- ***Dates and Times part1 :***

On doit surtout savoir comme les formatter en String.

Il existe deux types de dates et d'heures dont on doit se préoccuper : les dates et les heures **RELATIVES** à notre région et le type de date et d'heure de **ZONE**.

Les trois premières lignes sont locales et la dernière est de zone. Elles sont toutes initialisées avec les valeurs de la date et l'heure actuelle grace .now().

```
LocalDate localDate = LocalDate.now();
LocalTime localTime = LocalTime.now();
LocalDateTime localDateTime = LocalDateTime.now();

ZonedDateTime zonedDateTime = ZonedDateTime.now();
```

Le fuseau horaire dans les types **LOCAL** sont implicites (dans nos paramètres régionaux) et dépendent d'où on se situe. Il ne faut pas les préciser.

A l'inverse on peut le spécifier pour un ZONE.

Pour initialiser une date manuellement on utilise la méthode static LocalDate.of(int, int,int) :

```
localDate = LocalDate.of(2022,10,31); //  2022-10-31
```

Idem pour un objet de LocalTime :

```
localTime = LocalTime.of(9,45, 00,00); // 9:45
```

Idem encore pour un objet de localDateTime :

```
localDateTime = LocalDateTime.of(
        2022,10,31,
        9,45); // 2022-10-31T09:45
```

Le t du résultat sépare la date de l'heure.

Pour une zone (elles peuvent être retrouvé sur le site d'oracle ou sur une base de donnée en ligne)

```
zonedDateTime = ZonedDateTime.of(
        2022,10,31,
        9,45,00,00,
        ZoneId.of("America/Chicago"));// 2022-10-31T09:45
85:00[America/Chicago]
```

Pour formatter une date il y'a plusieurs méthodes :

1. On met tout dans un String en séparant par des espaces et en remplissant par des get()…

Comme on le voit cette manière est un peu fastidieuse.

```
String dateTimeString = ""
        +localDateTime.getDayOfWeek() + " "
        +localDateTime.getMonth() + " "
        +localDateTime.getDayOfMonth() + " "
        +localDateTime.getYear() + " at "
        +localDateTime.getHour() + ":"
        +localDateTime.getMinute();
out.println(dateTimeString); // MONDAY OCTOBER 31 2022 at 9:45
```

2. On utilise un DateTimeFormatter :

```
dateTimeString =
localDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
out.println(dateTimeString); // 2022-10-31T09:45
```

Le ISO doit correspondre à la bonne forme, ici c'est pour une LocalDateTime. Il y'en a beaucoup des différentes.

- ***Dates and Times part2 :***

Si l'on veut personnaliser le format d'affichage de la date, il faut d'abord créer une variable sur laquelle on applique la méthode DateTimeFormatter.ofPattern() :

```
var pattern = DateTimeFormatter.ofPattern("EEEE, LLLL dd, yyyy 'at'
hh:mm");
String dateTimeString = localDateTime.format(pattern);
out.println(dateTimeString); // Monday, October 31, 2022 at 09:45
```

Cette variable remplace le dateTimeformatter.ISO de la méthode précédente. Il prend un String comme argument qui doit contenir les différentes informations nécessaires, par exemple : yyyy = l'année, dd = le jour du mois etc…. (On peut retrouver ces infos en ligne sur le site d'Oracle). Attention que la variable pattern doit être appliqué au bon type. Ici on l'a fait avec un localDateTime, il doit donc être appliqué sur la méthode d'un objet localDateTime !

Ici on voit qu'appliqué au mauvais format, le pattern entrainera une exception.

```
/* invalid pattern applied to Local Time */
LocalTime localTime = LocalTime.of(
        9,45, 00,00);
//dateTimeString = localTime.format(pattern); PAS BON
```

On peut y placer du texte tant que celui-ci est entre des single quote ''. Il prend en charge les virgule et les doubles points et d'autres symboles mais pas tous.