Ch 2 – élements du langage

I. Eléments de base du langage

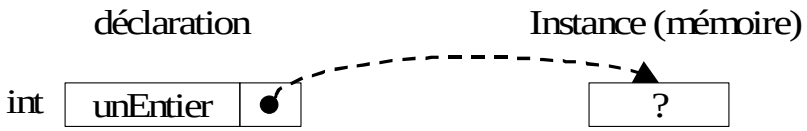
A. Types de données élémentaires et variables

Type de données	description	Exemple de déclaration de variables	Valeur possible
boolean	Valeur logique	`boolean test = true ;`	true, false
char	caractère	`char car = 'a' ;`	Toute valeur Unicode(1)
byte	Nombre entier (8 bits)	`byte numMois = 12 ;`	De -128 à +127
short	Idem. (16 bits)	`short annee = 2008 ;`	De -32768 à +32767
int	Idem.(32 bits)	`int distance = 0 ;`	De -2147483648 à +2147483647
long	Idem.(64 bits)	`long gdeDistance = 0 ;`	De -9223372036854775808 à +9223372036854775807
float	Nombre à virgule flottante (32 bits)	`float f = 3.14159 ;`	De $-3.40282347*10^{38}$ à $+3.40282347*10^{38}$
double	Idem. (64 bits)	`double d = 0 ;`	de $-1,7976931348623157*10^{308}$ à $+1,7976931348623157*10^{308}$

(1)Unicode : jeu de caractères standardisé permettant le codage des caractères de toutes les langues

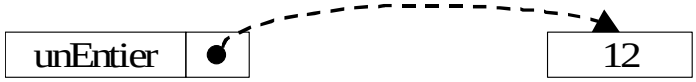
Les variables peuvent être déclarées aussi bien dans la partie membres attributs (portée de classe) d'une classe qu'au sein d'une méthode (portée de bloc et des blocs imbriqués).

```
// Déclaration
int unEntier ;
```



Une fois déclaré, l'entier est vide : il faut en initialiser la valeurs avant de l'utiliser :

```
// Initialisation
unEntier = 12;
```



Le type **`String`** ('chaîne de caractère') n'est pas un type de données élémentaire, mais une **classe**.

```
// Déclaration
String unMessage ;
```


```
// Instanciation
unMessage = new String("bonjour") ;
```

Cependant des facilités du langage permettent de l'utiliser comme un type primitif :

```
String message = "un message"; // écriture simplifiée
String message = new String("un message"); // création normale instance
```

B. Déclaration de constantes

```
static final float PI = 3.1415927 ;
```

C. Opérateurs

Type d'opérateur	opérateur	Exemples
Arithmétique	+, -, *, /, %	**`(4 + 3)`**
Affectation	=	**`i = (i+4)`**
Affectation composée	+=, -=, *=	**`i+=4 ;`** équivaut à **`i=i+4 ;`**
Unaire	++, --	**`i++ ;`** équivaut à **`i=i+1 ;`**
Comparaison logique	==, !=, <, >, <=, >=,	**`boolean b = (1 == 2) ;`**
Ternaire [(1)]	? :	**`int max = (i > j) ? i : j ;`** équivaut à : **`if (i>j) max = i ;`** **`else max = j ;`**
Connecteurs Logiques	&&, ||, !	**`boolean b = ((i==1)&&(j==2)) ;`**
Comparaison logique bit à bit	~, &, ^, <<, >>	Permet de réaliser des opérations OU / ET binaires et des décalages

instanciation	**new**	**Object o = new Object() ;** Construction d'une instance de classe **Object** à laquelle on pourra faire référence en utilisant le nom **o**

(1) existe aussi en C/C++

D. Transtypage (« cast »)

Le transtypage permet de faire passer une variable d'un type vers un autre type. Il est :

- **Implicite** : lorsque le type d'accueil est plus grand, la conversion est naturelle

```
int i = 23 ;
long j = i ;
```

- **Explicite** : lorsque le type d'accueil n'est pas assuré de pouvoir recevoir la valeur.

```
long j = 12345678 ;
int i = (int) j ;
```

E. Structures de contrôle

Type de structure	opérateur

Conditionnelles	`if (valeur logique) { ;}`
	`if (valeur logique) { ;} else { ;}`
	`if (valeur logique) { ;} else if (vl) { ;} else { ;}`

Choix multiple	`switch (exp_entier ou char) {`
	`  case val1 : break ;`
	`  default : ;`
	`}`

Répétitives	`while (valeur logique) { ;}`
	`do { ;} while (valeur logique);`
	`for (init ;valeur logique;increment) { ;}`

F. Structures de données : tableaux

Un tableau permet le stockage de variable de même type ; sa taille est fixe.
Un tableau est considéré comme un objet en Java et disposera de mécanismes de vérification de débordement.

```
// Déclaration
String [] tabChaine ;
int [] tabEntier1 ;
int [][] tabEntier2 ;
```

Une fois déclaré, la tableau est vide : il faut en initialiser les valeurs avant de l'utiliser :

```
// Construction (=création effective)
tabChaine = new String [10];
tabEntier1 = new int [365];
tabEntier2 = new int [52][7];
```

On peut connaître le nombre d'éléments déclarés d'un tableau grâce à son attribut public « **length** ».

```
int nombreElem = tabChaine.length ; // nombreElem contient 10
```

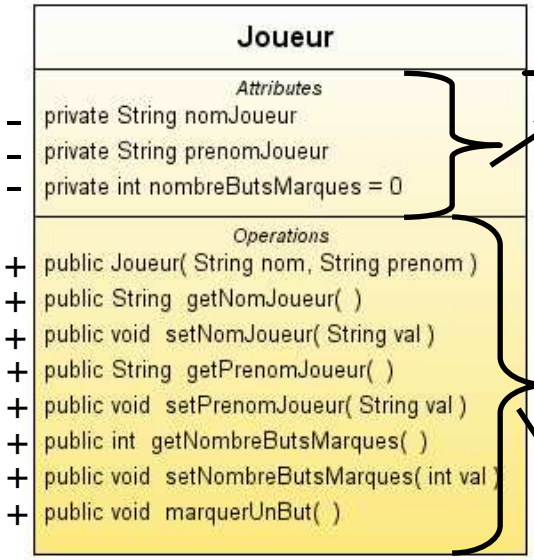
II. Classes et Objets

A. Classe

Une **classe** définit
- une structure de données (membres **attributs**)
- les opérations permettant de manipuler ces données (membres **méthodes**)

Une classe est une intention (un plan, un modèle, un moule), elle n'est pas concrète.

Classe Joueur (notation UML [1])



Classe Joueur (langage Java)

```
public class Joueur { // debut classe

    private String nomJoueur;
    private String prenomJoueur;
    private int nombreButsMarques = 0;

    public Joueur (String nom, String
prenom) {
        nomJoueur = nom;
        prenomJoueur = prenom;
    }

    public String getNomJoueur () {
        return nomJoueur;
    }
    public void setNomJoueur (String val) {
        this.nomJoueur = val;
    }

    public int getNombreButsMarques () {
        return nombreButsMarques;
    }
    public void setNombreButsMarques (int
val) {
        this.nombreButsMarques = val;
    }

    public String getPrenomJoueur () {
        return prenomJoueur;
    }
    public void setPrenomJoueur (String
val) {
        this.prenomJoueur = val;
    }

    public void marquerUnBut () {
        this.nombreButsMarques++;
    }

} // fin classe Joueur
```

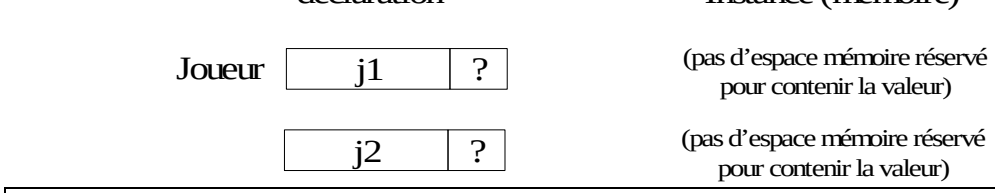
[1] UML (Unified Modeling Language) est la notation utilisée pour modéliser les applications orientées objets (mais également tous les systèmes logiciels développés aujourd'hui).

B. Objet, instance d'une classe, constructeur

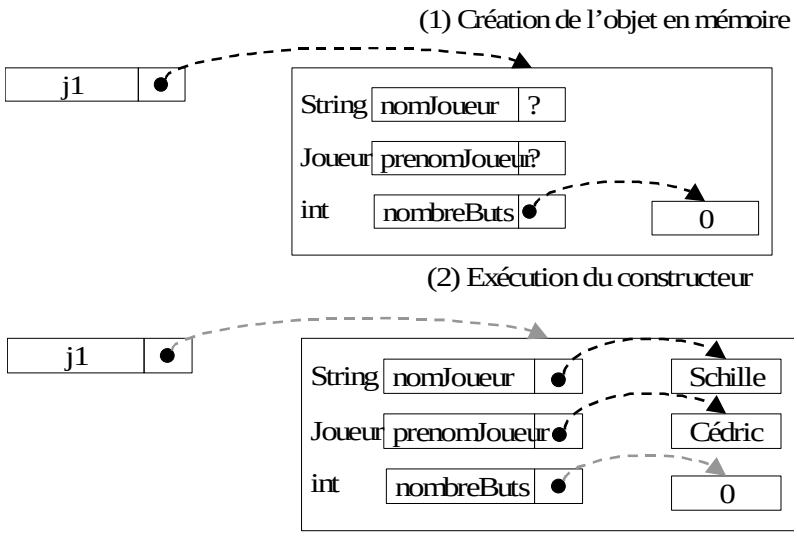
On appelle **objet** ou **instance de classe** ou **instance**, le résultat de la création en mémoire d'une zone qui va pouvoir contenir les membres d'une instance d'une classe (existe réellement).
Un objet possède 3 caractéristiques :
- une **identité** = une variable fait référence à cet objet

- un **état** = les valeurs des données qu'elle contient
- un **comportement** = l'ensemble de ses méthodes

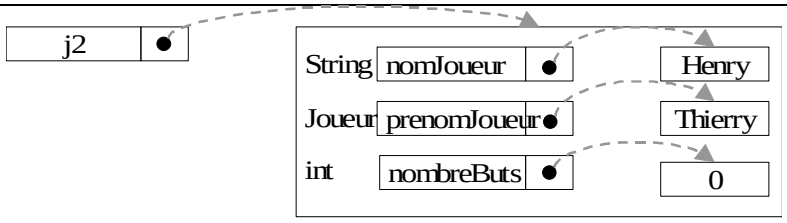
```
public class TesteurJoueur {

    public static void main (String[] args) {
        // Déclaration d'une variable d'instance
        Joueur j1, j2 ;
```




```
        // Instanciation (création d'un objet)
        j1 = new Joueur("Schille","Cédric"); (1) et (2)
```




```
        Joueur j2 = new Joueur("Henry","Thierry");
```



A ce moment, 2 instances de la classe Joueur existent en mémoire

```
    } // fin fonction main
} // fin classe TesteurJoueur
```

L'opérateur **new crée un nouvel objet**, crée une nouvelle instance d'une classe.

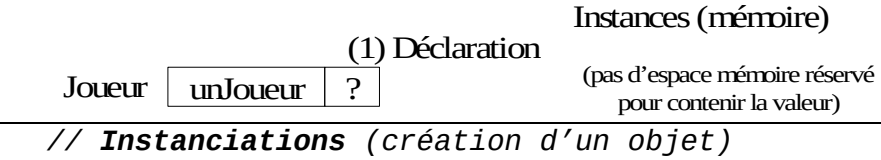
Le **constructeur** est la **méthode** d'une classe qui est **appelée lorsqu'on crée un nouvel objet** afin d'**initialiser ses propres attributs**. (= Préparer l'état initial de l'objet).
C'est une méthode particulière qui ne retourne aucune valeur.

Même si dans une classe, aucun constructeur n'est écrit, elle dispose toujours d'un constructeur par défaut qui ne prend aucun paramètre et qu'elle hérite de son ancêtre (Object, par exemple).

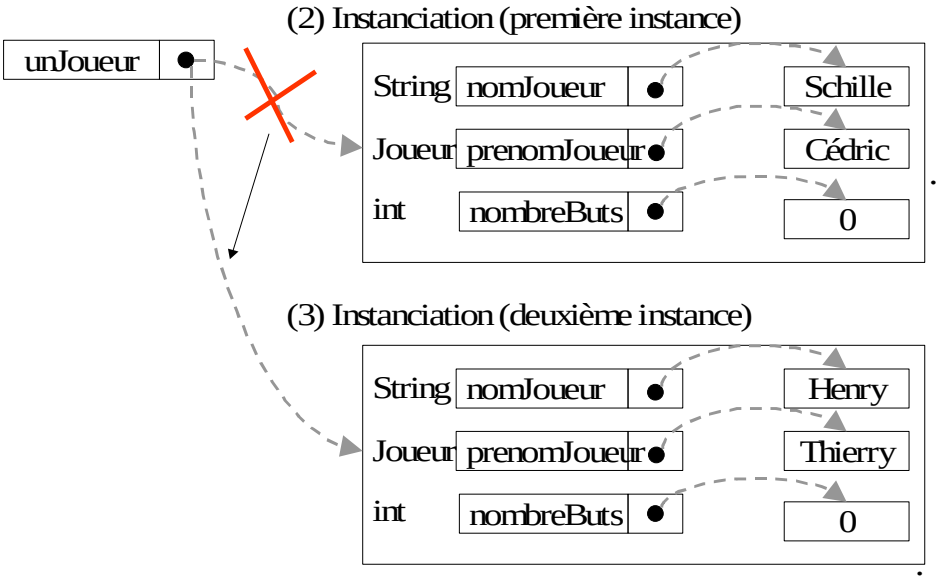
Une classe peut posséder plusieurs constructeurs : leurs signatures doit être différentes afin que le mécanisme de **polymorphisme** puisse sélectionner celui qui convient en fonction des arguments passés.

►►Attention : certaines instances peuvent ne plus être accessibles :

```
...
        // Déclaration d'une variable d'instance
        Joueur unJoueur ;
```



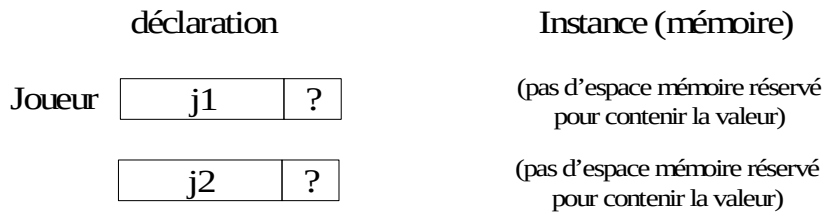

```
        // Instanciations (création d'un objet)
        unJoueur = new Joueur("Schille","Cédric"); (2)
        unJoueur = new Joueur("Henry","Thierry"); (3)
```



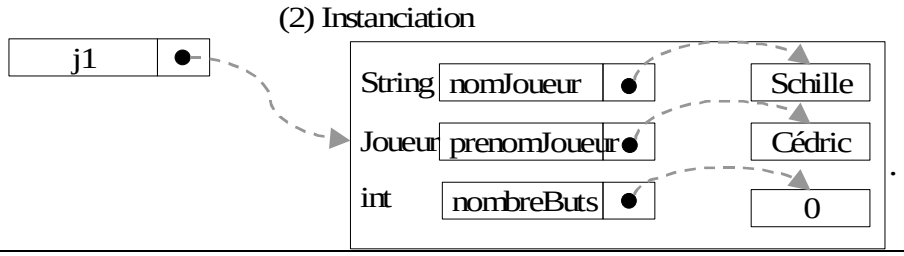
A ce moment, 2 instances de la classe Joueur existent en mémoire
MAIS une seule est référencée par l avariable d'instance 'unJoueur'
(L'autre n'est plus accessible et sera nettoyée par le ramasse-miettes - garbage collector -)

►►Attention : l'affectation d'une instance à une variable d'instance ne crée pas de nouvel objet :

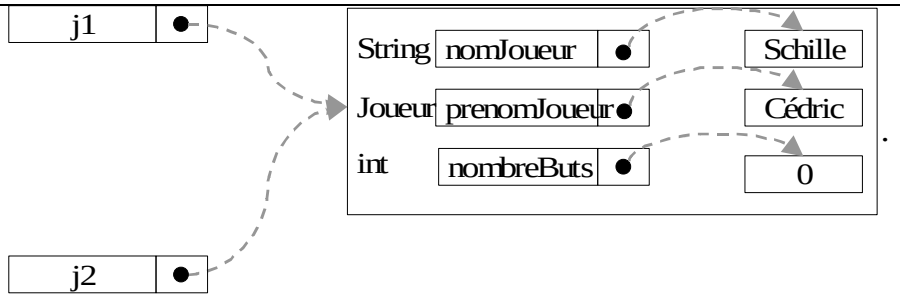
```
        // Déclaration d'une variable d'instance
        Joueur j1, j2 ;
```



```
// Instanciation (création d'un objet)
j1 = new Joueur("Schille","Cédric"); (1)
```




```
// affectation
j2 = j1;
```



A ce moment, 1 seule instance de la classe Joueur existe en mémoire
MAIS 2 variables d'instances font référence à la même instance

C. Messages et Méthodes, Accesseurs et Mutateurs

La Programmation Orienté Objet définit des objets s'échangeant des messages :

- un **message**, est le signal envoyé à une instance particulière, avec ou sans paramètres.
- une **méthode**, appliquée à une instance particulière, est l'algorithme déclenché par ce signal.

Plus concrètement, pour le développeur informatique,

- le **message** correspond à **l'appel d'une méthode**,
- et la **méthode** correspond à la **fonction qui va s'exécuter**.

```
// Instanciation (création d'un objet)
Joueur j1 = new Joueur("Schille","Cédric");
String nom = j1.getNomJoueur() ;
```

Dans certains contexte, ou bien pour établir un standard d'écriture des classes, 2 méthodes permettent d'accéder à chacun des attributs privés d'une classe :

- Le « **mutateur** », ou « setter » : affecte une valeur à un attribut (**modifie sa valeur**)
- L' « **accesseur** » ou « getter » : récupère la valeur d'un attribut (**lit sa valeur**).

Pour chaque attribut (nommé dans l'exemple XXXX), on trouvera ainsi les méthodes suivantes :

- Pour le mutateur :

```
public void setXXXX (String XXXX) {
    this.XXXX = XXXX ;
}
```

- pour l'accesseur (le type de retour dépend du type de donnée de l'attribut, ici String) :

```
public String getXXXX () {
    return XXXX ;
}
```

D. Visibilité : private, public

La visibilité est la caractéristique qui définit le niveau de protection (d'encapsulation) des membres d'une instance :

- **`private` :** un attribut ou une méthode privée n'est pas accessible en dehors de l'instance (signe '-' dans les diagrammes de classe).
- **`public` :** un attribut ou une méthode publique est accessible en dehors de l'instance (signe '+' dans les diagrammes de classe).

E. Méthodes et Attributs de classe : static

Le mot clef **`static`** permet de définir qu'une méthode ou un attribut seront uniques pour toutes les instances d'une classe. On parle **d'attributs ou méthodes de classe,** qui seront partagés par toutes les instances de cette classe.
Contrairement aux méthodes d'instance qui ne peuvent être appelée pour des instances de classe, une méthode **`static`** peut être appelée en spécifiant un nom de classe, sans qu'une instance ait été créée.
C'est pourquoi, par exemple, le point d'entrée d'une application Java est qualifié de **`static`**.

```
public class TesteurJoueur {

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

On peut appeler la méthode « main » sans instancier la classe « TesteurJoueur ».

F. Méthodes et Attributs terminaux : final

Le mot clef **`final`** permet de définir qu'une méthode ou un attribut ne pourront pas être redéfinis dans une classe héritée.
Les valeurs constantes sont ainsi qualifiées de **`final`** (leur valeur n'est plus modifiable).

```
static final float PI = 3.1415927 ;
```

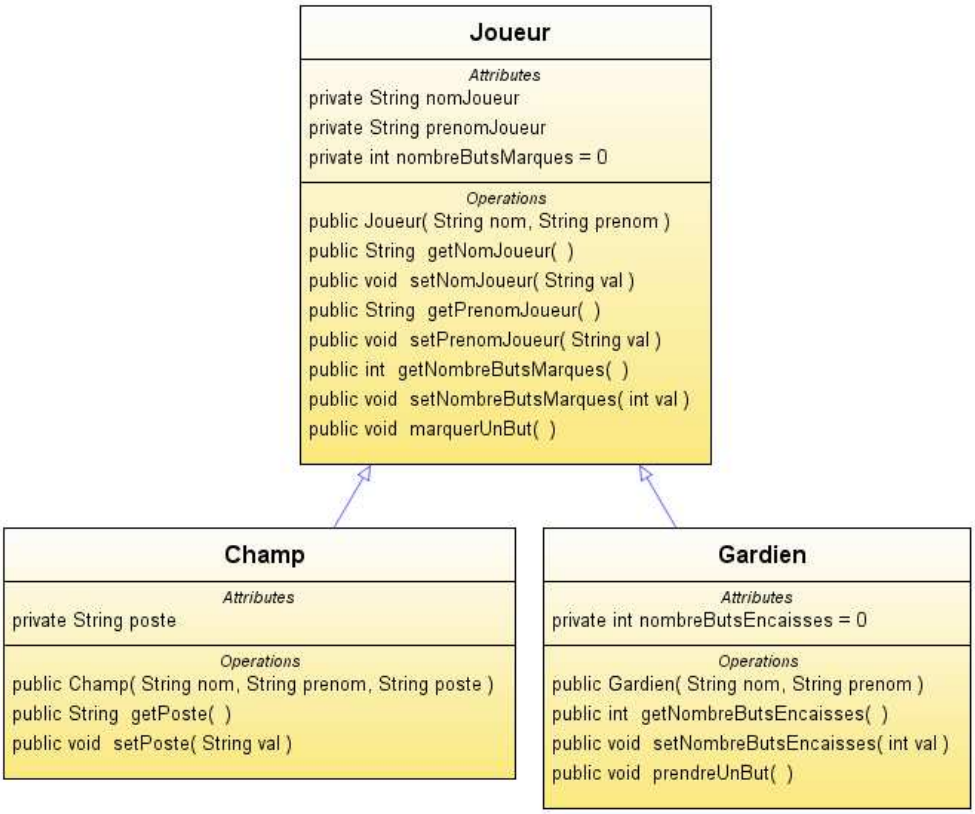
III. Héritage/généralisation, spécialisation

L'**héritage**, ou la relation de généralisation, précise pour 2 classes que l'une est spécialisation de l'autre : elle possède l'ensemble[1] des attributs et des méthodes de la première (sauf privés), plus les siens propres.

La **classe mère** – ou super classe – ou classe parent – est la classe qui lègue des membres par héritage.
La **classe fille** – ou sous classe – est une nouvelle classe ayant acquis par définition de l'héritage les membres de la classe mère. Les classes « filles » sont des spécialisations de la classe « mère ».

Exemple :

[1] On verra plus tard que la classe mère peut restreindre l'héritage de ses membres.



IV. Notion d' « interface »

Les « interfaces » correspondent à un squelette d'une classe avec seulement des signatures de méthodes. Il constitue une forme de contrat que devra respecter une carte qui implémentera cet interface. Elle devra obligatoirement implémenter toutes les méthodes définies dans l'interface.

Par exemple, le fichier « IJoueur.java » :

```
public interface IJoueur { // debut de l'interface

  public void manquer() ;
}
```

Dans une classe Java implémentant cet interface, on doit retrouver la méthode « marque » avec son implémentation (= son corps, les instructions qui y seront exécutées)

```
public class Joueur implements IJoueur { // debut classe

  // corps de la classe

     public void marquer() {
         // faire quelque chose
     }
}
```

V. Paquetages : package

Les paquetages servent à organiser les centaines de classes définies par le JDK pour Java.
Dans une application, on peut utiliser la notion de « package » pour organiser les classes d'une application.

Pour déclarer une classe dans un package, il suffit de d'ajouter dans le fichier source d'une classe l'indication de son package :

```
package fr.lycee.guymollet.sport ;

import java.util.* ;

public class Joueur { // debut classe

 // corps de la classe

}
```

Dans une autre classe Java, lorsqu'on aura besoin de réutiliser une classe, on pourra indiquer qu'on importe son package. :

```
import java.util.* ;

import fr.lycee.guymollet.sport.Joueur ;

public class Equipe { // debut classe

 // corps de la classe

}
```