Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Électronique et d'Informatique

Département d'Informatique

Programmation Orientée Objets
Support de TP

Dr. Ilhem BOUSSAÏD

iboussaid@usthb.dz

Licence 2 académique

Année universitaire 2014/2015

Table des matières

Chapitre Un

Programmation Orientée Objets avec Java – Introduction et éléments de base

1.1 Bref historique du langage java

Le langage Java[1] a été conçu, vers 1990, au sein de l'entreprise **Sun Microsystems** (par une équipe dirigée par James Gosling).

Le langage a été popularisé par :

- Sa portabilité (indépendance des plates-formes)
- Le slogan "**Write Once Run Anywhere**"
- Une syntaxe très proche de C/C++
- Des éléments de sécurité intégrés
- La gratuité de son kit de développement (JDK)
- Son adoption dans la formation (écoles, universités)
- Gestion dynamique de la mémoire, utilisation d'un GC (*Garbage Collector*)
- Gestion des erreurs (exceptions)
- Richesse des librairies de classes : *Application Programming Interface* (API)
- Multitâche intégré au langage (*Multi-threading*)
- Java et le réseau (*network is the computer*)
- Pas d'arithmétique de pointeur

1. Java est synonyme de "café" en argot américain

1.2 Compilation et Interprétation

Voir figure 1.1.

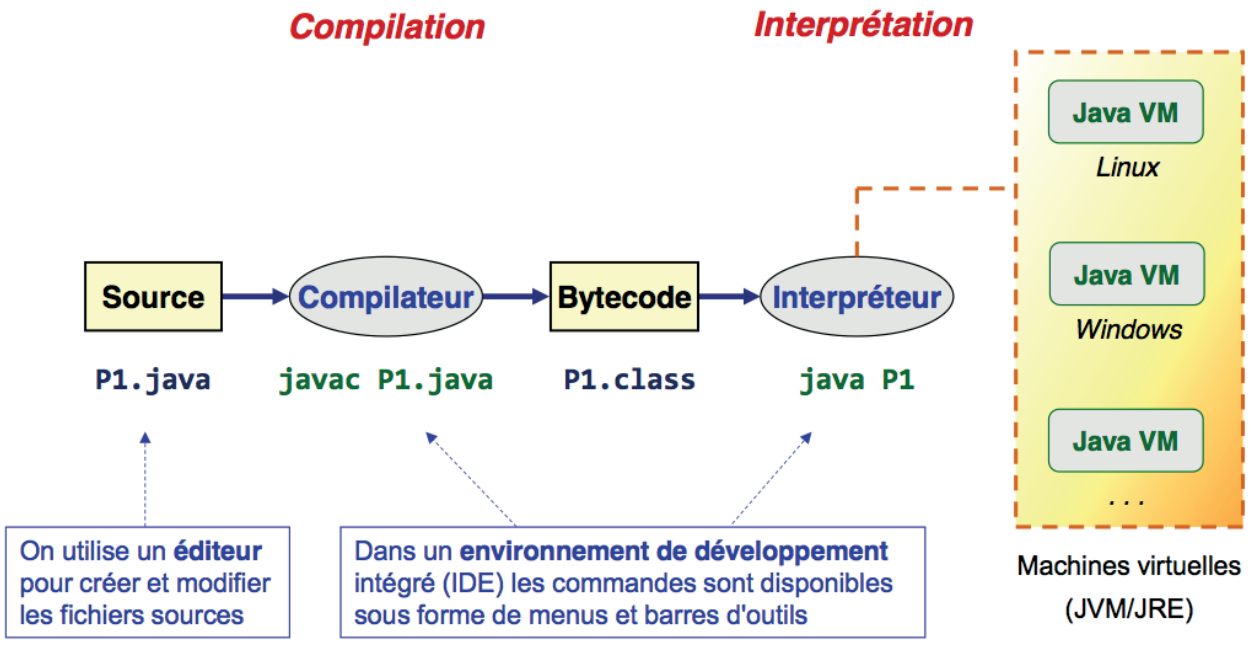

Figure 1.1: Compilation et Interprétation

1.2.1 Compilation en mode "lignes de commandes"

Java est un langage compilé : compilateur (de base) = `javac`

`NomClasse.java` ⇒ `NomClasse.class`

Pour compiler le programme il faut taper la commande : `> javac NomClasse.java`

Cette commande génère un fichier compilé de nom : `NomClasse.class`. Et pour lancer l'exécution de ce programme il faut taper :

```
> java NomClasse [args]
```

à condition que la classe `NomClasse` définisse la méthode statique `public static void main(String[`

1.2.2 CLASSPATH

voir variable système `PATH`

- La variable d'environnement `CLASSPATH` est utilisée pour localiser toutes les classes nécessaires pour la compilation ou l'exécution.
- Elle contient la liste des répertoires où chercher les classes nécessaires.
- Par défaut elle est réduite au répertoire courant (. . .).

- Les classes fournies de base avec le jdk sont également automatiquement trouvées.
- Il est possible de spécifier un `classpath` propre à une exécution/compilation :
  (Windows) : `java/javac -classpath lib;.;/truc/classes;\%CLASSPAT\% ...`
  (Linux) : `java/javac -classpath lib:.:/truc/classes:$CLASSPATH ...`

1.2.3 JavaDoc

FichierClasse.java⇒ FichierClasse.html

- Commentaires encadrés par /** ... */
- utilisation possible de tags HTML
- Tags spécifiques :
  - classe `@version`, `@author`, `@see`, `@since`
  - méthode `@param`, `@return`, `@exception`, `@see`, `@deprecated`
- conservation de l'arborescence des paquetages
- liens hypertextes entre classes

```
javadoc TestJavaDoc.java -d ../javadoc
```

1.3 Commentaires

Trois formes de commentaires :

1. `// ...Texte...`

   - Commence dès `//` et se termine à la fin de la ligne
   - Sur une seule ligne
   - A utiliser de préférence pour les commentaires généraux

2. `/* ...Texte... */`

   - Le texte entre `/*` et `*/` est ignoré par le compilateur
   - Peuvent s'étendre sur plusieurs lignes
   - Ne peuvent pas être imbriqués
   - Peuvent être utiles pour inactiver (temporairement) une zone de code

3. `/** ...Texte... */`

   - Commentaire de documentation (comme `/* ... */` mais avec fonction spéciale)
   - Interprétés par l'utilitaire javadoc pour créer une documentation au format HTML
   - Peuvent contenir des balises de documentation (@author, @param, ...)
   - Peuvent contenir des balises HTML (Tags)

```
/**
* Somme : Calcule une somme *
* @author xxxx
* @version 2.3 10.02.2014 */
   public class Somme {
/*-------------------------------------------------+
| Programme principal | +
-------------------------------------------------*/
public static void main(String[] params) {
       //--- Initialisation des variables
        inta =1;
        intb =2;
        int total; // Totalisateur
        total = a + b;
       //--- Affichage du résultat de l'addition
       System.out.println("Résultat = " + total);
     }
}
```

1.4 Identificateurs

Les identificateurs sont des noms symboliques permettant de référencer les éléments des programmes Java (variables, fonctions, ...).

Règles pour les identificateurs

- Doivent **commencer par une lettre ou un souligné** (ou, à éviter, un caractère monétaire)
- Suivi (éventuellement) d'un nombre quelconque de lettres, chiffres ou soulignés (ou, à éviter, de caractères monétaires)
- **Distinction entre les majuscules et les minuscules**
- **Éviter les caractères alphabétiques spéciaux** (caractères accentués, lettres grecques, ...(peuvent poser des problèmes sur certaines plateformes)
- Les mots réservés du langage sont exclus (`if`, `this`, `implements`, ...)

1.5 Types de base

Le tableau 2.1 représente les types primitifs.

`3` dans `a = 3` n'est pas un `int` en Java, la valeur dépend du type de `a`

```
byte b = 3; // ok
```

Type	Déclaration	taille	Ensemble de valeurs	Opérateurs principaux
Booléens	`boolean`	1 bit	`true` et `false`	! (non), && (et), || (ou)
Octets	`byte`	8 bits	-128 à 127	-+ * / % > >= < <= ==!= ++ –
Entiers courts (peu utilisés)	`short`	16 bits	de -32768 à 32767	-+ * / % > >= < <= ==!= ++ –
Entiers	`int`	32 bits	de -2147483648 à 2147483647	? (unaire, binaire) + * / % (modulo) > >= < <= == (égalité) != (inégalité)
Entiers longs	`long`	64 bits	de -2^{63} à $2^{63}-1$	-+ * / % > >= < <= ==!= ++ –
Flottants simple précision (peu utilisés)	`float`	32 bits	de $-3,4028235 \times 10^{+38}$ à $-1,4 \times 10^{-45}$, 0 et de $1,4 \times 10^{-45}$ à $3,4028235 \times 10^{+38}$	-+ * / > >= < <= ==!=
Flottans double précision	`double`	64 bits	de $-1,7976931348623157 \times 10^{+308}$ à $-4,9 \times 10^{-324}$, 0 et de $4,9 \times 10^{-324}$ à $1,7976931348623157 \times 10^{+308}$	-+ * / > >= < <= ==!=
Caractères (unicode)	`char`	16 bits	Tous les caractères Unicode	==!= < <= > >=

Tableau 1.1: Classes des types élémentaires

```
short s = 3; // ok
int s = 3; // ok
long l = 3; // ok
```

Ce mécanisme n'existe pas pour les valeurs flottantes

```
float f = 3.0; // compile pas
```

1.5.1 Les valeurs flottantes

Utilise la norme IEEE 754 pour représenter les valeurs à virgule flottante.

`3.0` est un double (64bits), `3.0f` (ou `3.0F`) est un float (32 bits)

La norme IEEE 754 introduit trois valeurs particulières par types de flottants

- `+Infinity` est le résultat de `i/0` avec `i` positif
- `-Infinity` est le résultat de `i/0` avec `i` négatif
- `NaN` est le résultat de `0/0`
  - `x == x` est faux si `x` vaut `Double.NaN`
  - Donc on doit tester `NaN` avec `Float.isNaN()` ou `Double.isNaN()`

1.5.2 Conversions de types

Mis à part le type booléen, tous les types primitifs peuvent être convertis entre eux.

– L'encodage Unicode permet (si nécessaire) de considérer les caractères comme des nombres. Le type char peut donc agir comme un entier `short` mais il est non signé (contrairement aux autres entiers).
– **Conversion élargissante** (automatique) (avec perte éventuelle de précision lors du passage en virgule flottante) – voir figure 1.2

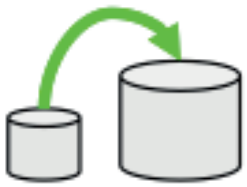

Figure 1.2: Conversion élargissante

– **Conversion restrictive** explicite (par **transtypage** / ***casting***) (sous la responsabilité du programmeur) – voir figure 1.3.

Figure 1.3: Conversion restrictive

– **Exemples :**

```
int i=17;
byte b = 4; // Conversion implicite : littéral int -> byte
b = i;  // Erreur à la compilation
b = (byte)i; // Conversion explicite (int -> byte)

float f = 23.86f;
i = (int)f;      // Valeur tronquée (i==23)
```

1.6 Variables

– **Déclaration**

`< type> <nom de variable> [ = <expression` dinitialisation>];'

Le type d'une variable peut être soit un **type primitif**, soit un **type référence** (c'est-à-dire le nom d'une classe ou d'un tableau)

– **Exemples**

```
int monEntier; // Type primitif int (valeur par défaut : 0)
double monFlottant = 2.212; // Type primitif double
char monChar = 'a'; // Type primitif char

String motDePasse; // Classe String (prédéfinie)
Point position;  // Classe Point
char[] voyelles; // Tableau de caractères
Point[] nuage; // Tableau de Point(s)
```

- **Affectation**

  `<variable> = <expression>;`

  ```
  monEntier = 8; // variantes += -= etc. (comme en C/C++)
  ```

- **Expression**

  ```
  System.out.println(monEntier+monFlottant*2);
  ```

1.7 Tableaux

- **Déclaration**

  `<type>[] <nom tableau> [= { <constante element 0>, <element 1>, ...} ];`

- **Exemples :**

  ```
  int[] monTableau;
  double[] monTableau2 = { 3.12, 4.11, 8.3 };
  ```

- **Allocation**

  `<tableau> = new <type>[<taille entier>];` ex. :

  ```
  monTableau = new int[20];
  ```

- **Affectation** `<tableau>[<index entier>] = <expression>;` ex. :

  ```
  monTableau[5] = 8;
  ```

- **Accès (expression)** `<tableau>[<index entier>]` ex. :

  ```
  System.out.println(monTableau[5]);
  ```

- **Remarques**
  - index de 0 à n-1 où n est la taille du tableau
  - taille du tableau : `<tableau>.length` ex. :

    ```
    System.out.println("taille␣=␣"+monTableau.length);
    ```

  - exemple tableau à dimension multiple :

    ```
    double[][] maMatrice = new double[3][3];
    maMatrice[1][2]=12; // ligne 1 , colonne 2
    ```

- taille statique et vérification des débordements
  - ex. `maMatrice[3][0]=3.2;` ⇒ `exception ArrayOutOfBoundsException`
  - tableaux de taille dynamique : cf. ArrayList (API Java)

1.8 Affichage

- **Sortie standard :**

```
System.out.print("Bonjour"); // sans retour de ligne
System.out.println("␣le␣monde␣"+2+"␣!"); // avec retour de ligne
```

- **Sortie d'erreur :**

```
System.err.println("Problème␣!");
```

1.9 Structures de contrôle

1.9.1 Alternatives

`if(<condition booléenne>) <conséquent> [ else <alternant> ]`

- si la condition est `true` alors exécuter le `<conséquent>`, sinon exécuter l'`<alternant>` (version avec `else`) ou ne rien faire (version sans `else`)

Remarque :

`<conséquent>` et `<alternant>` : instruction `<instr>;` ou bloc `{ <instr1>; <instr2>; ... }`

1. Exemple 1 :

```
if(monEntier>12)
System.out.print("grand␣"); // conséquent
else
   System.out.println("petit␣"); // alternant
   System.out.println(monEntier);
        // (attention: ce n'est pas l'alternant !)
```

2. Exemple 2 :

```
int a = 8; int b = 9;
if (b>a) {
int temp = a;
a = b;
b = temp;
}
System.out.println("a="+a+",␣b="+b);
```

1.9.2 L'opérateur conditionnel ternaire

L'opérateur conditionnel (`? :`) est un opérateur ternaire qui fournit, dans certains cas, une forme raccourcie du `if ... else`. Sa syntaxe est la suivante :

```
<condition> ? <expression1> : <expression2>
```

Cette expression est égale à `<expression1>` si `<condition>` est satisfaite (`true`), et à `<expression2>` sinon. Par exemple, l'expression

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre.

1.9.3 Traitement par cas

Test Multiple numérique ou string ou enum :

```
switch(<expression int char byte short String enum>) {
case <valeur1> : <instrs cas 1> ; break;
        // pas de break ? => teste les cas suivants
case <valeur2> : <instructions cas 2> ; break;
...
default: <instructions par défaut>;
}
```

- en fonction de la valeur de l'`<expression>`, exécuter le `<cas 1>` (séquence d'instructions) `si <valeur 1>`, le `<cas 2>` si `<valeur 2>`. Le cas par défaut (`default`) est inconditionnel.
- **Attention** : si on ne met pas de `break` alors plusieurs cas (en particulier le `default`) peuvent s'exécuter.

Exemple :

```
int monEntier = 1;
String chaine = "";
switch(monEntier) {

case 0 : chaine="zéro"; break;
case 1 : chaine="un"; break;
case 2 : chaine="deux"; break;
default : chaine="plutôt␣grand";

}
System.out.println(monEntier+"␣est␣"+chaine);
```

```
switch(args.length) {
        case 0:
                return;
        case 2:
                doSomething();
                break;
        default:
                doSomethingElse();
}

switch(args[0]) {
        case "-a":
                lsAll();
                break;
        default:
                lsSimple();
}

switch(threadState) {
        case NEW:
                runIt();
                break;
        case RUNNABLE:
                break;
        case BLOCKED:
                unblock();
}
```

1.9.4 Boucles while

`while (<condition de poursuite>) <corps>`

`<corps>` est une instruction ou un bloc

Exemples

1. Exemple 1 :

```
 int i= 0;
while(i<tab.length) // on suppose un tableau tab
System.out.println(tab[i++]);
```

2. Exemple 2 :

```
 int j = 10; int total = 0;
while(j>0) {
total += total*j;
```

```
    j--;
    }
    System.out.println("total␣=␣"+total);
```

1.9.5 Variante do while

On exécute au moins une fois le <corps> avant de tester la <condition de poursuite>

```
int j=0; int total = 1;
do { total+=total * j; j--; } while(j>0);
System.out.println("total␣do/while␣=␣"+total);
j=0;total = 1;
while(j>0) { total+=total*j; j--; }
System.out.println("total␣while␣=␣"+total);
```

1.9.6 Boucles for

for(<déclaration/inits> ; <condition de poursuite> ; <mises à jour>)<corps>

Tant que la condition de poursuite est true, le <corps> s'exécute. Après exécution du <corps>, les mises à jour sont effectuées. Les variables déclarées et initialisées sont visibles dans le <corps>.

Remarques :

- déclarations et mises à jour multiples : séparées par des virgules
- <corps> est une instruction simple
- <instr>;' ou un bloc
- <instr1>; <instr2>; ... '
- Exemple 1 :

```
for(int i=0;i<tab.length;i++) // on suppose un tableau tab
System.out.println(tab[i]);
```

- Exemple2 :

```
int total1 = 0; int total2 = 0;
for(int j=10;j>0;j--) {
total1 += total1*j;
total2 += total2+j;
}
System.out.println("total1␣=␣"+total1);
System.out.println("total2␣=␣"+total2);
```

1.9.7 Boucles for-each (depuis java 1.5)

`for(<type> <variable> : <collection>) <corps>`

Pour chaque élément de la `<collection>` (tableau ou collection `ArrayList`, `HashSet`, etc.), exécuter le `<corps>` dans lequel la `<variable>` de type `<type>` est visible et reçoit l'élément courant

– Exemple 1 :

```
int taille = 0;
for(Object obj : tab) {
System.out.println("L'élément␣courant␣est␣:␣"+obj);
taille++;
}
System.out.println("La␣taille␣du␣tableau␣est␣:␣"+taille);
```

– Exemple 2 :

```
ArrayList<String> maListe = new ArrayList<String>();
maListe.add("Kant");
maListe.add("Aristote");
maListe.add("Platon");
for(String philosophe : maListe)
System.out.println(philosophe+"␣est␣un␣philosophe␣connu");
```

1.9.8 break/continue label

`break [label]` : permet d'interrompre le bloc (en cours ou celui désigné par le label).

`continue [label]` : permet de sauter à l'incrémentation suivante

```
public static boolean find(int val,int v1,int v2) {
loop: for(int i=1;i<=v1;i++)
for(int j=1;j<=v2;j++) {
int mult=v1*v2;
if (mult==val)
return true;
if (mult>val)
continue loop;
}
return false;
}
```

Chapitre Deux

Quelques classes utiles

2.1 Les classes enveloppes (*Wrappers*)

Afin que les données de types primitifs puissent être traitées comme des objets lorsque cela est nécessaire, Java fournit huit **classes-enveloppes** (*Wrappers*), une pour chaque type primitif : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` et `Character`. Ces classes portent le même nom que le type élémentaire sur lequel elles reposent avec la première lettre en majuscule.

Classe	Rôle
`Integer`	pour les valeurs entières (`integer`)
`Long`	pour les entiers longs signés (`long`)
`Float`	pour les nombres à virgules flottante (`float`)
`Double`	pour les nombres à virgule flottante en double précision (`double`)
...	...

Tableau 2.1: Classes-enveloppes des types primitifs

Ces classes ont à peu près les mêmes méthodes ; parmi les principales :

- un constructeur prenant pour argument la valeur du type primitif qu'il s'agit d'envelopper. Ex : `Integer x = new Integer (10);`
- La réciproque : une méthode `xxxValue` (`xxx` vaut `int`, `double`, etc.) pour obtenir la valeur de type primitif enveloppée. Ex : `intValue`, `doubleValue`, `booleanValue`
- une méthode nommée `toString` pour obtenir cette valeur sous forme de chaîne de caractèeres,
- éventuellement, une méthode statique, appelée `parseXxx`, permettant d'obtenir une valeur d'un type primitif à partir de sa représentation textuelle,
- éventuellement, une méthode statique `valueOf` pour construire un tel objet à partir de sa représentation sous forme de texte. Par exemple, `IntegerValueOf(uneChaine)` donne le même résultat que `new Integer(Integer.parseInt(uneChaine))` ;

Classe `Integer` :

```
Integer(int i)              //conversion int -> Integer
int intValue()              //conversion Integer -> int
```

```
String toString()          //conversion Integer -> String
static Integer valueOf(String s)   //conversion String -> Integer
static int parseInt(String s)      //conversion String -> int
```

Classe `Double` :

```
Double(double d)           //conversion double -> Double
double doubleValue()       //conversion Double -> double
String toString()          //conversion Double -> String
static Double valueOf(String s)    //conversion String -> Double
static double parseDouble(String s)//conversion String -> double
```

Classe `Boolean` :

```
Boolean(boolean b)         //conversion boolean -> Boolean
boolean booleanValue()     //conversion Boolean -> boolean
String toString()          //conversion Boolean -> String
static Boolean valueOf(String s)   //conversion String -> Boolean
```

Classe `Character` :

```
Character(char c)  //conversion char -> Character
char charValue()   //conversion Character -> char
String toString()  //conversion Character -> String
```

À propos de conversions, signalons également l'existence dans la classe `String` des méthodes suivantes :

```
static String valueOf(int i)     //conversion int -> String
static String valueOf(long l)    //conversion long -> String
static String valueOf(float f)   //conversion float -> String
static String valueOf(double d) //conversion double -> String
static String valueOf(boolean b)//conversion boolean -> String
static String valueOf(char c)    //conversion char -> String
static String valueOf(Object o) //conversion Object -> String
```

Exemples de conversions

```
int i;  // Variable de type int
float f;        // Variable de type float
double d;       // Variable de type double
String s;       // Variable de type String
s = "412";
i = Integer.parseInt(s);        // Conversion String -> int
s="1.234";
d = Double.parseDouble(s);  // Conversion String -> double
f = 123.8F;
s = Float.toString(f);      // Conversion float -> String
```

```
s = String.valueOf(f);        // Conversion float -> String
```

Attention

Ne pas confondre les littéraux de type caractère ou chaîne de caractères avec les littéraux numériques. Par exemple '4', "4" et 4. Le premier est un caractère (type `char`), le deuxième une chaîne de caractères (type `String`) et le troisième est une valeur numérique entière (type `int`).

2.2 Lectures simples au clavier : la classe Scanner

La classe `java.util.Scanner` permet d'effectuer simplement des lectures à la console (comme on le fait en C avec la fonction scanf). Pour utiliser la classe `Scanner`, il faut d'abord l'importer :

```
import java.util.Scanner;
```

Ensuite il faut créer un objet de la classe `Scanner` (nommée ici `sc`) connectée à un flot en entrée, qui très souvent est l'entrée standard `System.in` :

```
Scanner sc = new Scanner(System.in);
```

Pour récupérer les données, il faut faire appel sur l'objet `sc` aux méthodes décrites ci-dessous. Ces méthodes parcourent la donnée suivante lue sur l'entrée et la retourne :

- `String next()` : donnée de la classe `String` qui forme un mot,
- `String nextLine()` : donnée de la classe `String` qui forme une ligne,
- `boolean nextBoolean()` : donnée booléenne,
- `int nextInt()` : donnée entière de type `int`,
- `double nextDouble()` : donnée réelle de type `double`.
- ...

2.2.1 Erreurs de lecture

Les erreurs lors de la lecture de données sont des fautes que le programmeur ne peut pas éviter, quel que soit le soin apporté à la conception de son programme. Il doit donc se contenter de faire le nécessaire pour les détecter lors de l'exécution et y réagir en conséquence. En Java cela passe par le mécanisme des exceptions :

```
...
double x = 0;
for (;;) {
    System.out.print("Donnez␣x:␣");
    try {
```

```
            x = entree.nextDouble();
            break;
        } catch (InputMismatchException ime) {
            entree.nextLine();
            System.out.println("Erreur de lecture - Recommencez");
         }
}
System.out.print("x = " + x); ...
```

Remarque 2.1. *Remarquez, dans l'exemple précédent, comment dans le cas d'une erreur de lecture il convient de "nettoyer" le tampon d'entrée (c'est le rôle de l'instruction entree.nextLine() ;) afin que la tentative suivante puisse se faire avec un tampon vide.*

2.2.2 Tests de présence de données

Il peut être utile de vérifier le type d'une donnée avant de la lire. La classe `Scanner` offre aussi toute une série de méthodes booléennes pour tester le type de la prochaine donnée disponible, sans lire cette dernière :

- `boolean hasNext()` : renvoie `true` s'il y a une donnée à lire,
- `boolean hasNext(String pattern)` : renvoie `true` si la prochaine donnée à lire forme le mot pattern,
- `boolean hasNextLine()` : renvoie `true` s'il y a une ligne à lire,
- `boolean hasNextBoolean()` : renvoie `true` s'il y a un booléen à lire,
- `boolean hasNextInt()` : renvoie `true` s'il y a un entier à lire,
- `boolean hasNextDouble()` : renvoie `true` s'il y a un double à lire.
- ...

Il existe d'autres méthodes de la classe `Scanner`. Si cela vous intéresse, allez consulter l'API de java.

Exemple très simple :

```
public class TestScanner {
    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);
        System.out.print("nom et prénom? ");
        String nom = entree.nextLine();
        System.out.print("âge? ");
        int age = entree.nextInt();
        System.out.print("taille (en m)? ");
        float taille = entree.nextFloat();
        System.out.println("lu: " + nom + ", " + age + " ans, " + taille
    }}
```

2.3 Quel jour sommes-nous ? quelle heure est-il ?

But : essayer diverses manières d'obtenir et afficher la date courante.

Première manière :

la méthode `System.currentTimeMillis()` donne la date courante, exprimée comme le nombre de millisecondes qui se sont écoulées depuis le 1er janvier 1970 à 0 heures GMT. C'est précis, mais pas très pratique pour organiser sa semaine ! (Retenez quand même l'existence de cette méthode, elle est utile pour mesurer et comparer les performances des programmes).

Deuxième manière :

créer une instance de la classe `java.util.Date()` (par une expression comme `Date d = new Date()`) et la donner à afficher à `System.out`. C'est mieux, on obtient bien une date, mais écrite à la manière anglo-saxonne.

Troisième manière :

créer une instance de la classe `java.util.Calendar` et obtenir séparément les composants de la date (jour de la semaine, jour du mois, mois, année) pour les afficher comme bon nous semble.

En étudiant la documentation de la classe `Calendar` vous découvrirez que, démunie de constructeurs publics, on en crée des instances par une expression comme `Calendar c = Calendar.getInstance()`.

Ensuite, on obtient les divers composants par des expressions de la forme `c.get(Calendar.MONTH)`, etc.

Des tableaux de chaînes constantes déclarés comme ceci peuvent vous aider à obtenir une présentation adéquate :

```
String[] mois = { "janvier", "février", ... "décembre" };
```

Quatrième manière (la meilleure) :

construire un objet `d` de type `Date` comme dans la deuxième manière et un objet `f` de type `SimpleDateFormat` (qui est une variété de `DateFormat`) et faire formater le premier par le second (par une expression comme `f.format(d)`).