Programmation orientée objet
-Langage JAVA-

DUT-GI (2ème année)
Année universitaire: 2009-2010

Pr. Mohamed NAIMI

Université Hassan 1er
Ecole Supérieure de Technologie –Berrechid (ESTB)

Plan du cours

- **Introduction.**
- **Syntaxe.**
- **Méthodes, classes et objets.**
- **Composition et héritage.**
- **Interfaces, paquetages et encapsulation.**
- **Collections.**
- **Exceptions.**
- **Fichiers et flux.**
- **Interfaces graphiques.**

Introduction

Introduction (1)

- La **P**rogrammation **O**rientée **O**bjet (**POO**) consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel (que l'on appelle domaine) en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objets. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, ...).

Introduction (2)

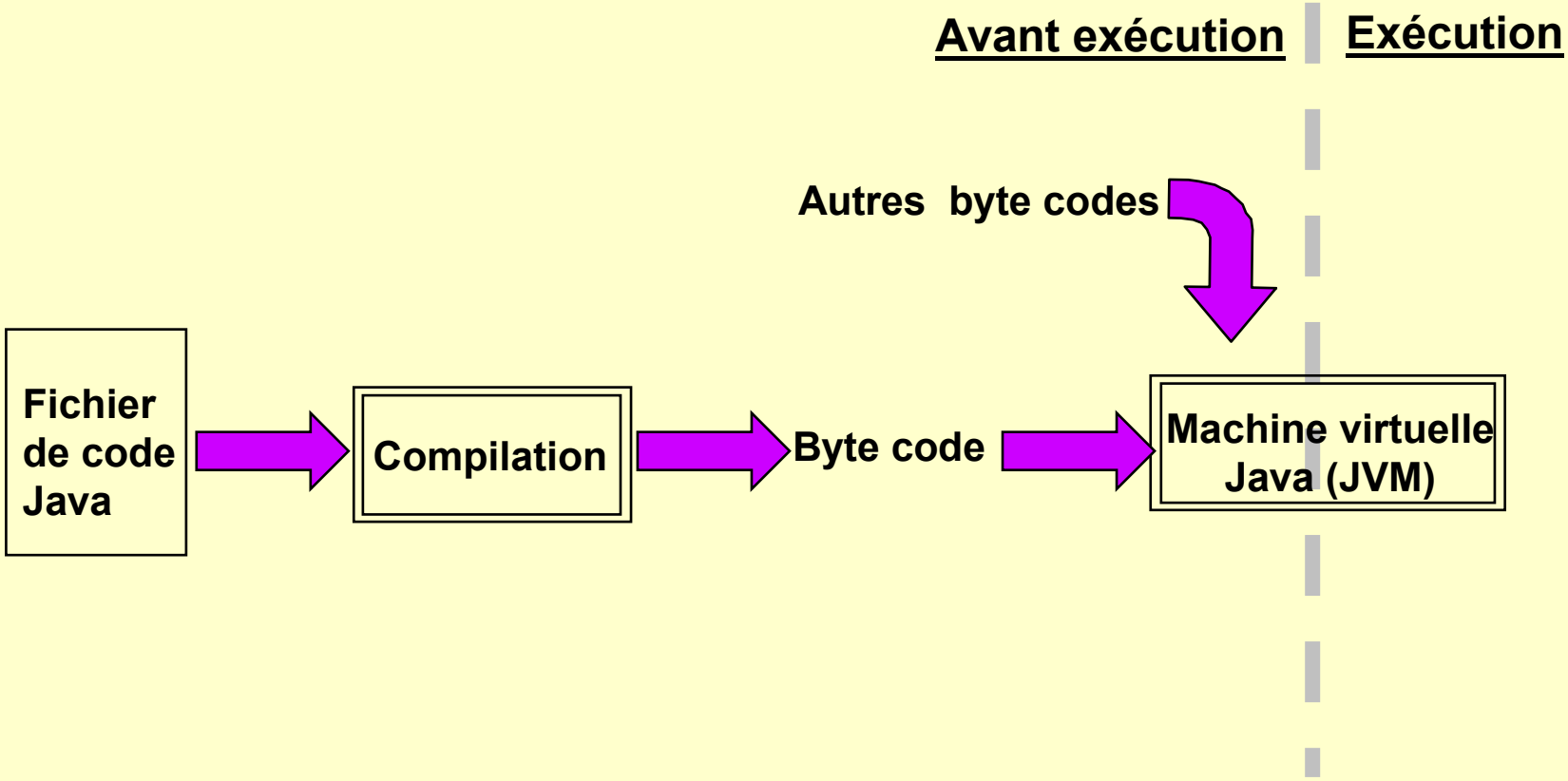
- Java a été développé à partir de décembre 1990 par une équipe de Sun Microsystems dirigée par James Gosling.
- Les fondateurs de Java ont réalisé un langage indépendant de toute architecture de telle sorte que Java devienne idéal pour programmer pour des réseaux hétérogènes, notamment Internet.

Introduction (3)

- Caractéristiques du langage Java:
  - **Portabilité**;
    - Indépendance par rapport aux plateformes.
  - **Sécurité et robustesse**;
    - Le compilateur interdit toute manipulation en mémoire.
  - **Gratuité**;
    - Les outils de développement Java sont fournis gratuitement.
  - **Richesse**;
    - Disponibilité d'une vaste collection de bibliothèques de classes.

Introduction (4)

D'où vient **la portabilité** de Java?

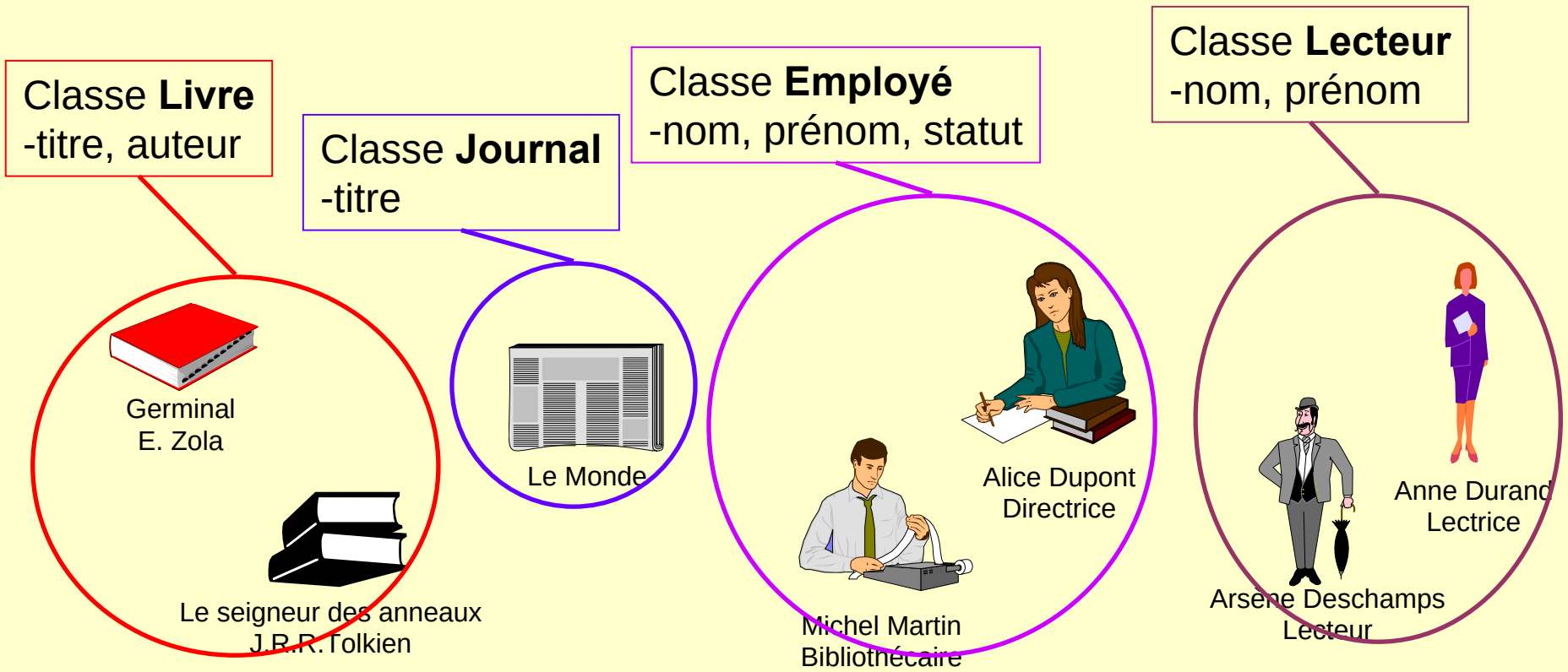


Introduction: Concepts de base (1)

- Une **classe** est la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet.
- Un **objet** est donc "issu" d'une classe, c'est le produit qui sort d'un moule. En réalité on dit qu'un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence).
- Une classe est composée de deux parties:
  - Les **attributs** ou **champs** (parfois appelés données membres): il s'agit des données représentant l'état de l'objet.
  - Les **méthodes** (parfois appelées fonctions membres): il s'agit des opérations applicables aux objets.

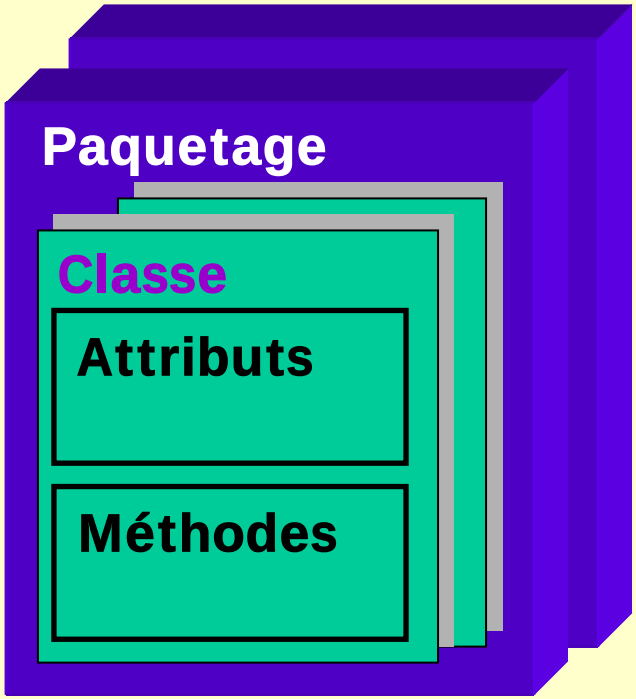
Introduction: Concepts de base (2)

Illustration:



Introduction: Structure d'un programme Java

- Un **programme Java** utilise un ensemble de **classes**.
- Les **classes** sont regroupées par **paquetage** (en anglais, **package**).
- Une **classe** regroupe un ensemble d'**attributs** et de **méthodes**.



Syntaxe du langage Java

Déclaration d'une classe

- Le nom de la classe est spécifié derrière le mot clé « **class** ».
- Le corps de la classe est délimité par des accolades.
- On définit dans le corps les attributs et les méthodes qui constituent la classe.

```
class Test {

        < corps de la classe >

}
```

Définition d'une méthode (1)

- Une méthode est constituée:
  - D'un nom.
  - D'un type de retour.
  - De paramètres ou arguments (éventuellement aucun).
  - D'un bloc d'instructions.
- Un paramètre est constitué:
  - D'un type.
  - D'un nom.
- « **void** » est le mot-clé signifiant que la méthode ne renvoie pas de valeur.

Définition d'une méthode (2)

- **Exemple:**

```
class Test {
  int calculer (int taux, float delta) {
            < corps de la méthode >
  }
}
```

Les commentaires

- /* **Commentaires sur une ou plusieurs lignes** */
  - Identiques à ceux existant dans le langage C.
- // **Commentaires de fin de ligne**
  - Identiques à ceux existant en C++.
- /** **Commentaires d'explication** */
  - Les commentaires d'explication se placent généralement juste avant une déclaration (d'attribut ou de méthode).

Instructions, blocs et blancs (1)

- Les **instructions** Java se terminent par un « **;** ».
- Les **blocs** sont délimités par deux accolades:
  - **{** pour le début de bloc
  - **}** pour la fin du bloc
    - Un bloc permet de définir un regroupement d'instructions. La définition d'une classe ou d'une méthode se fait dans un bloc.
- Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.

Instructions, blocs et blancs (2)

- Instructions possibles:
  - Déclaration d'une variable.
  - Appel de méthode.
  - Affectation.
  - Instruction de boucle (while, for…).
  - Instruction de test (if, switch).

Déclaration d'une variable

- Une **variable** possède un type et un nom.
- Le type peut être un type de base ou une classe.
- L'initialisation d'une variable peut se faire au moment de la déclaration.

```
{
  int compteur; // Déclaration
  int indice = 0; // Déclaration + Initialisation

  Voiture golf;
  Voiture twingo = new Voiture();
}
```

Portée d'une variable

- La **portée** d'une **variable** s'étend jusqu'à la fin du bloc dans lequel elle est définie.

```
{
  {
    int compteur;
    …
    // compteur est accessible
  }

  // compteur n'est plus accessible
}
```

L’affectation

- L’opérateur « = » permet d’affecter la valeur de l’expression qui se trouve à droite à la variable qui se trouve à gauche.

```
class Test {
  int calculer () {
            int i = 0;
            int j = 6;
            i = (j + 5) * 3; // Instruction d’affectation
            return i + j;
  }
}
```

Les opérateurs arithmétiques élémentaires

- Règles de **priorité** sur les opérateurs arithmétiques:

Niveau	Symbole	Signification
1	()	Parenthèse
2	*	Produit
	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

Les opérateurs de comparaison

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

Les opérateurs logiques

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
||	expr1 || expr2	Expr1 ou expr2, ou les deux sont vraies
|	expr1 | expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse
!=	expr1 != expr2	si expr1 est différent de expr2

Les opérateurs d'affectation

- L'opérateur de base est « = ».
- Il existe des opérateurs d'affectation qui réalisent à la fois une opération arithmétique, logique, ou bit à bit et l'affectation proprement dite:

Opérateur	Exemple	Équivalent à
+=	expr1 += expr2	expr1 = expr1 + expr2
-=	expr1 -= expr2	expr1 = expr1 – expr2
*=	expr1 *= expr2	expr1 = expr1 * expr2
/=	expr1 /= expr2	expr1 = expr1 / expr2
%=	expr1 %= expr2	expr1 = expr1 % expr2

Point d'entrée d'un programme Java

- Pour pouvoir faire un programme exécutable il faut toujours une classe qui contienne une méthode particulière: la méthode « **main** ».
  - C'est le point d'entrée dans le programme: le microprocesseur sait qu'il va commencer à exécuter les instructions à partir de cet endroit.

```
public static void main(String arg[ ])
{
…/…
}
```

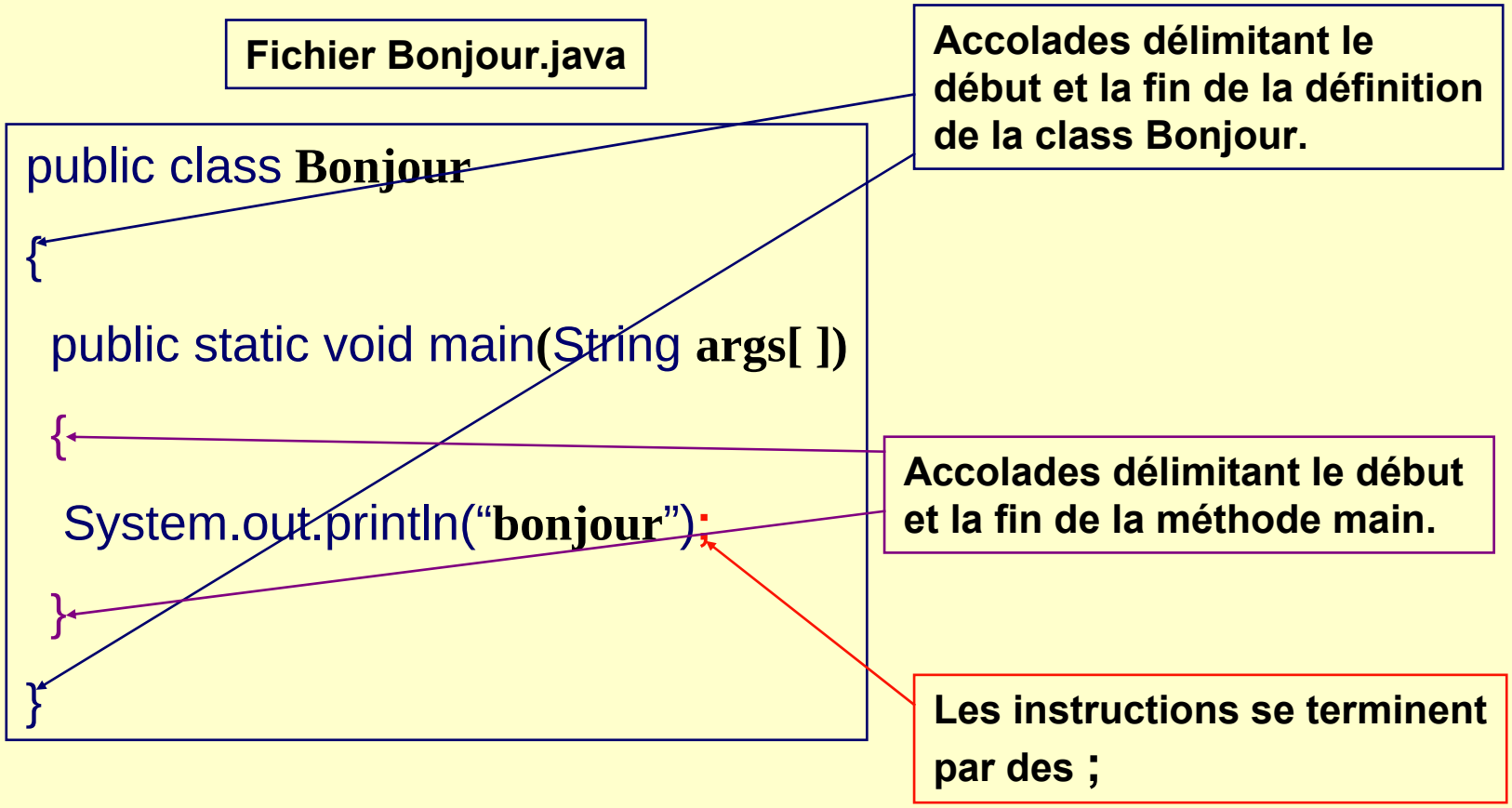
Exemple (1)

Fichier Bonjour.java

```
public class Bonjour
{    //Accolade débutant la classe Bonjour
  public static void main(String args[ ])
  { //Accolade débutant la méthode main
    /* Pour l'instant juste une instruction */
   System.out.println("bonjour");
  } //Accolade fermant la méthode main
} //Accolade fermant la classe Bonjour
```

La classe est l'unité de base de nos programmes. Le mot clé en Java pour définir une classe est class.

Exemple (2)


Fichier Bonjour.java
public class Bonjour
{
public static void main(String args[ ])
{
System.out.println("bonjour");
}
}
Accolades délimitant le début et la fin de la définition de la class Bonjour.
Accolades délimitant le début et la fin de la méthode main.
Les instructions se terminent par des ;

Exemple (3)

Fichier Bonjour.java

```
public class Bonjour
{
  public static void main(String args[ ])
  {
   System.out.println("bonjour");
  }
}
```

Une méthode peut recevoir des paramètres. Ici la méthode main reçoit le paramètre args qui est un tableau de chaîne de caractères.

Compilation et exécution (1)

Fichier Bonjour.java

Le nom du fichier est nécessairement celui de la classe suivi de l'extension « .java ». Java est sensible à la casse des lettres (exemple: Bonjour ≠ bonjour).

Compilation d'un programme java dans une console DOS:
javac Bonjour.java
Génère un fichier Bonjour.class
Exécution du programme (toujours depuis la console DOS) sur la JVM :
java Bonjour
Affichage de « bonjour » dans la console.

```
public class Bonjour
{
  public static void main(String[ ] args)
  {
   System.out.println("bonjour");
  }
}
```

Compilation et exécution (2)

- Pour résumer, dans une console DOS, si j'ai un fichier Bonjour.java pour la classe Bonjour:
  - **javac** Bonjour.java
    - **javac** est la commande qui lance le compilateur Java.
    - Compilation en bytecode java.
    - Indication des erreurs (éventuelles) de syntaxe.
    - Génération d'un fichier Bonjour.class s'il n'y a pas d'erreurs.
  - **java** Bonjour
    - **java** est la commande qui lance la machine virtuelle (JVM).
    - Exécution du bytecode.
- **Remarque:**
  - Nécessité de la méthode **main**, qui est le point d'entrée dans le programme.

Compilation et exécution (3)

- On peut utiliser ce qu'on appelle un environnement de développement intégré (**IDE**) pour compiler et exécuter des applications Java.
- Un **IDE** Java est un éditeur spécifique du code Java ayant une interface intuitive qui permet de faciliter l'édition, la compilation, la correction d'erreurs et l'exécution des applications Java.
- Exemples d'IDE Java:
  - JCreator.
  - netBeans (Open Source).
  - BlueJ.
  - JBuilder.

Identificateurs

- Un **identificateur** permet de désigner une classe, une méthode, une variable…
- Règles à respecter pour les identificateurs:
  - Interdiction d'utiliser les mots-clés (mots réservés de Java).
  - **Les identificateurs Commencent par:**
    - Une lettre.
    - Un « $ ».
    - Un « _ » (underscore).
  - **Ne commencent pas par:**
    - Un chiffre.
    - Un signe de ponctuation autre que « $ » ou « _ ».

Les mots réservés de Java

abstract	default	goto	null	synchronized
boolean	do	if	package	this
break	double	implements	private	throw
byte	else	import	protected	throws
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
continue	float	native	super	volatile
const	for	new	switch	while

Les types de base (1)

- En Java, tout est objet sauf les types de base.
- Il y a huit types de base:
  - Un type booléen pour représenter les variables ne pouvant prendre que 2 valeurs (vrai et faux, 0 ou 1, etc.): **Boolean** avec les valeurs associées **true** et **false**.
  - Un type pour représenter les caractères: **char**.
  - Quatre types pour représenter les entiers de divers taille: **byte**, **short**, **int** et **long**.
  - Deux types pour représenter les réelles: **float** et **double**.
- La taille nécessaire au stockage de ces types est indépendante de la machine.
  - Avantage: portabilité.
  - Inconvénient: « conversions » coûteuses.

Les types de base (2): Les entiers

- Les entiers (avec signe):
  - **byte**: codé sur 8 bits, peuvent représenter des entiers allant de -2^7 à $2^7 - 1$ (-128 à +127).
  - **short**: codé sur 16 bits, peuvent représenter des entiers allant de -2^{15} à $2^{15} - 1$.
  - **int**: codé sur 32 bits, peuvent représenter des entiers allant de -2^{31} à $2^{31} - 1$.
  - **long**: codé sur 64 bits, peuvent représenter des entiers allant de -2^{63} à $2^{63} - 1$.

Les types de base (3): Les entiers

- Opérations sur les entiers:
  - **+** : addition.
  - **-** : soustraction.
  - ***** : multiplication.
  - **/** : division entière.
  - **%** : reste de la division entière.
    - **Exemples:**
      - 15 / 4 donne 3.
      - 15 % 2 donne 1.

Les types de base (4): Les entiers

- Opérations sur les entiers (suite):
  - Les opérateurs d'incrémentation et de décrémentation ++ et -- :
    - + + : ajoute 1 à la valeur d'une variable entière.
    - - - : retranche 1 de la valeur d'une variable entière.
    - n++; « équivalent à » n = n+1;
    - n--; « équivalent à » n = n-1;
    - Exemple:
      - int n = 12;
      - n ++; // Maintenant n vaut 13.

Les types de base (5): Les entiers

- Opérations sur les entiers (suite):
  - 8++; est une instruction illégale.
  - L'incrémentation (resp. la décrémentation) peut être utilisée de manière suffixée: **++n**. La différence avec la version préfixée se voit quand on les utilisent dans les expressions. En version suffixée l'incrémentation (resp. la décrémentation) s'effectue en premier, alors qu'elle s'effectue en dernier en version préfixée.
  - Exemple illustratif:

```
int m=7; int n=7;
int a=2 * ++m; // a vaut 16, m vaut 8
int b=2 * n++; // b vaut 14, n vaut 8
```

Les types de base (6): Les réels

- Les réels:
  - **float**: codé sur 32 bits, peuvent représenter des nombres allant de -10^{35} à $+10^{35}$.
  - **double**: codé sur 64 bits, peuvent représenter des nombres allant de -10^{400} à $+10^{400}$.
- Notation:
  - 4.55 ou 4.55**D:** réel en double précision.
  - 4.55**f:** réel en simple précision.

Les types de base (7): Les réels

- Les opérateurs sur les réels:
  - Opérateurs classiques: +, -, *, /
  - Attention pour la division:
    - 15 / 4 donne 3.
    - 15.0 / 4 donne 3.75 (si l'un des termes de la division est un réel, la division retournera un réel).
  - Puissance:
    - Utilisation de la méthode **pow** de la classe **Math**:
      - double y = Math.pow(x, a) « équivalent à » x^a (i.e. x^a en notation mathématique), avec x et a étant de type double.

Les types de base (8): Les booléens

- Les booléens:
  - **boolean**: contient soit vrai (**true**) soit faux (**false**).
- Les opérateurs logiques de comparaison:
  - Egalité: opérateur ==
  - Différence: opérateur **!=**
  - Supérieur et inférieur strictement à: opérateurs > et <
  - Supérieur et inférieur ou égal: opérateurs >= et <=

Les types de base (9): Les booléens

- **Illustration:**

```
boolean x;
x= true;
x= false;
x= (5==5); // l'expression (5==5) est évaluée et la valeur est affectée à x
                qui vaut alors vrai
x= (5!=4); // x vaut vrai, ici on obtient vrai car 5 est différent de 4
x= (5>5); // x vaut faux, 5 n'est pas supérieur strictement à 5
x= (5<=5); // x vaut vrai, 5 est bien inférieur ou égal à 5
```

Les types de base (10): Les booléens

- Autres opérateurs logiques:
  - Et logique: **&&**
  - Ou logique: ||
  - Non logique: **!**
- **Exemples:**

```
boolean a,b, c;
a= true;
b= false;
c= (a && b); // c vaut false
c= (a || b); // c vaut true
c= !(a && b); // c vaut true
c=!a; // c vaut false
```

Les types de base (11): Les caractères

- Les caractères:
  - « **char** » contient un seul caractère (lettre, symbole, ponctuation…).
  - Le type **char** désigne des caractères en représentation Unicode.
    - Codage sur 2 octets contrairement à ASCII/ANSI codé sur 1 octet.
    - Notation hexadécimale des caractères Unicode de ‘ \u0000 ’ à ‘ \uFFFF ’.
  - **Remarque**:
    - Le codage ASCII/ANSI est un sous-ensemble de la représentation Unicode.
  - Pour plus d’information sur Unicode: www.unicode.org

Les types de base (12): Les caractères

- **Illustration:**

```
char a,b,c; // a,b et c sont des variables de type char
a='a'; // a contient la lettre « a »
b= '\u0022' // b contient le caractère guillemet  "
c=97; // c contient le caractère de rang 97: 'a'
```

Les types de base (13): Exemple et remarque

- **Exemple:**

int **x = 0, y = 0;**

float **z = 3.1415**F**;**

double **w = 3.1415;**

long **t = 99**L**;**

boolean **test = true;**

char **c = 'a';**

- **Remarque importante:**
  - Java exige que toutes les variables soient définies et initialisées. Le compilateur sait déterminer si une variable est susceptible d'être utilisée avant initialisation et produit une erreur de compilation.

Les structures de contrôle (1)

- Les structures de contrôle en Java :
  - **if, else**
  - **switch, case, default, break**
  - **for**
  - **while**
  - **do, while**

Les structures de contrôle (2): if, else

- Instructions conditionnelles:
  - On veut effectuer une ou plusieurs instructions seulement si une certaine condition est vraie:

    if (*condition***)** *instruction*;

    et plus généralement: **if (***condition***)** {*bloc d'instructions*}
  - Condition doit être un booléen ou doit renvoyer une valeur booléenne.
  - On veut effectuer une ou plusieurs instructions si une certaine condition est vérifiée sinon effectuer d'autres instructions:

    if (*condition***)** *instruction1;* **else** *instruction2*;

    et plus généralement: **if (***condition***)** {*1er bloc d'instructions*} **else** {*2ème bloc d'instructions*}

Les structures de contrôle (3): if, else

Max.java

```
import java.io.*;
public class Max {
   public static void main (String[ ] args) throws IOException {
            BufferedReader in;
            in = new BufferedReader (new InputStreamReader(System.in));
            System.out.println ("Entrer le premier entier:");
            String input1 = in.readLine();
            int nb1 = Integer.parseInt(input1);
            System.out.println("Entrer le second entier:");
            String input2 = in.readLine();
            int nb2 = Integer.parseInt(input2);
            if (nb1 > nb2)
                        System.out.println("L'entier le plus grand est:   " + nb1);
            else
                        System.out.println("L'entier le plus grand est:   " + nb2);

   }
}
```

Les structures de contrôle (4): while

- Boucles indéterminées (while):
  - On veut répéter une ou plusieurs instructions un nombre indéterminé de fois: On répète l'instruction ou le bloc d'instruction tant qu'une certaine condition reste vraie.
  - Voici la syntaxe de la boucle while (tant que):
    - **while** (*condition*) **{***bloc d'instructions***}**
  - Les instructions dans le bloc sont répétées tant que la condition reste vraie.
  - On ne rentre jamais dans la boucle si la condition est fausse dès le départ.

Les structures de contrôle (5): while

Facto1.java

```
import java.io.*;
public class Facto1
{
   public static void main (String[ ] args) throws IOException {
                int n, result, i;
                BufferedReader in;
                in = new BufferedReader (new InputStreamReader(System.in));
                System.out.println ("Entrer une valeur pour n:");
                String input = in.readLine();
                n = Integer.parseInt(input);
    result = 1; i = n;
    while (i > 1)
     {
       result = result * i;
       i--;
     }
      System.out.println ("La factorielle de  " + n + " vaut " + result);
  }
}
```

Les structures de contrôle (6): do, while

- Boucles indéterminées (do, while):
  - Voici la syntaxe de la boucle do, while:
    - **do** {*bloc d'instructions*} **while** (*condition*)
  - Les instructions dans le bloc sont répétées tant que la condition reste vraie.
  - On rentre toujours au moins une fois dans la boucle: La condition est testée en fin de boucle.

Les structures de contrôle (7): for

- Boucles déterminées (for):
  - On veut répéter une ou plusieurs instructions un nombre déterminé de fois: On répète l'instruction ou le bloc d'instructions pour un certain nombre d'itérations.
  - Syntaxe générale de la boucle for:
    - **for (int i = debut; i < fin; i++)**
      Instructions;
  - **Exemple:**
    - **for** (int i = 0; i < 10; i++)
      System.out.println(i); // affichage des nombres de 0 à 9
  - **Remarque:**
    - La syntaxe de la boucle **for** peut avoir d'autres variantes:
      - **for (int i = fin; i > debut; i--)**
        Instructions;

Les structures de contrôle (8): for

Facto2.java

```
import java.io.*;
public class Facto2
{
   public static void main (String[ ] args) throws IOException {
              int n, result, i;
              BufferedReader in;
              in = new BufferedReader (new InputStreamReader(System.in));
              System.out.println ("Entrer une valeur pour n:");
              String input = in.readLine();
              n = Integer.parseInt(input);
    result = 1;
    for(i =n; i > 1; i--)
     {
       result = result * i;
     }
     System.out.println ("La factorielle de  " + n + " vaut " + result);
  }
}
```

Les structures de contrôle (9): switch

- Sélection multiple (switch):
  - L'utilisation de « if, else » peut s'avérer lourde quand on doit traiter plusieurs sélections ou effectuer un choix parmi plusieurs alternatives.
  - Pour cela, il existe en Java l'instruction « **switch**, **case** » qui est identique à celle de C/C++.
  - La valeur sur laquelle on teste doit être un char ou un entier (à l'exclusion d'un **long**).
  - L'exécution des instructions correspondant à une alternative commence au niveau du **case** correspondant et se termine à la rencontre d'une instruction **break** ou arrivée à la fin du **switch**.

Les structures de contrôle (10): switch

- Syntaxe général de l'instruction switch:

```
Switch (variable) {
Case valeur 1:
Instructions; break;
Case valeur 2:
Instructions; break;
…
Case valeur N:
Instructions; break;
Default:
Instructions;
}
```

Les structures de contrôle (11): switch

Alternative.java

```
import java.io.*;
public class Alternative {
   public static void main(String[] args) throws IOException {
              BufferedReader in;
              in = new BufferedReader (new InputStreamReader(System.in));
              System.out.println ("Entrer un entier:");
              String input = in.readLine();
              int nb = Integer.parseInt(input);

   switch(nb)
    {
      case 1:
        System.out.println("1"); break;
      case 2:
        System.out.println("2"); break;
      default:
       System.out.println ("Autre nombre"); break;
     }
   }
}
```

Première alternative :
on affiche 1 et on sort
du bloc du switch au break;

Deuxième alternative :
on affiche 2 et on sort
du bloc du switch au break;

Alternative par défaut:
on réalise une action
par défaut.

Les tableaux (1)

- Les **tableaux** permettent de stocker plusieurs valeurs de même type dans une variable.
  - Les valeurs contenues dans la variable sont repérées par un indice.
  - En langage java, les tableaux sont des objets.
- Déclaration:
  - **Type Nom_Tableau[ ]; // Ou Type[ ] Nom_Tableau;**
    - int tab[ ]; // Ou int[ ] tab;
    - String chaines[ ]; // Ou String[ ] chaines;
- Création d'un tableau:
  - **Nom_Tableau = new Type[Taille_Tableau];**
    - tab = new int [20]; // tableau de 20 entiers
    - chaines = new String [100]; // tableau de 100 chaînes

Les tableaux (2)

- Le nombre d'éléments du tableau est mémorisé. Java peut ainsi détecter, lors de l'exécution, le dépassement d'indice et générer une exception.
- Mot clé pour récupérer la taille d'un tableau: **length**
  - Syntaxe:
    - **Nom_Tableau.length**;
  - **Exemple:**
    - int taille = tab.length; // taille vaut 20
- Comme en C/C++, les indices d'un tableau commencent à ' 0 '. Donc un tableau de taille 100 aura ses indices qui iront de 0 à 99.

Les tableaux (3)

- Initialisation:

```
tab[0] = 5;
tab[1] = 3; // etc.
chaines[0] = new String("Pierre");
chaines[1] = new String("Paul"); // etc.
```

- Création et initialisation simultanées:

```
int tab [ ] = {5, 3};
String chaines [ ] = {"Pierre", "Paul"};
```

Les tableaux (4)

Tab1.java

```
public class Tab1
{
  public static void main (String args[ ])
  {
    int tab[ ] ;
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```

Pour déclarer une variable tableau on indique le type des éléments du tableau et le nom de la variable tableau suivi de [ ].

On utilise new <type> [taille]; pour initialiser le tableau.

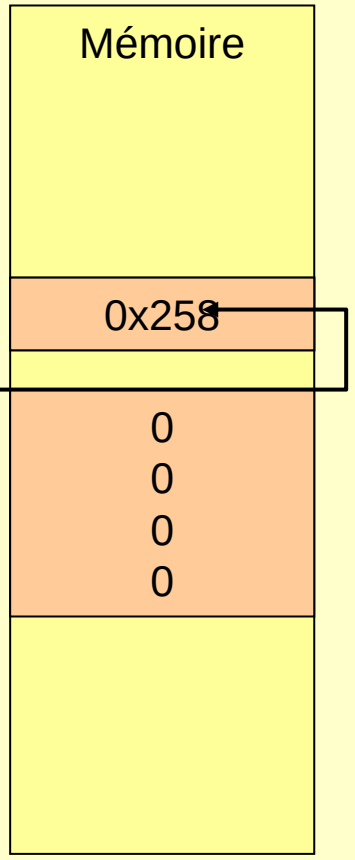
On peut ensuite affecter des valeurs au différentes cases du tableau:
<Nom_Tableau>[indice] = Valeur

Les indices vont toujours de 0 à (taille-1)

Les tableaux (5)

Tab1.java

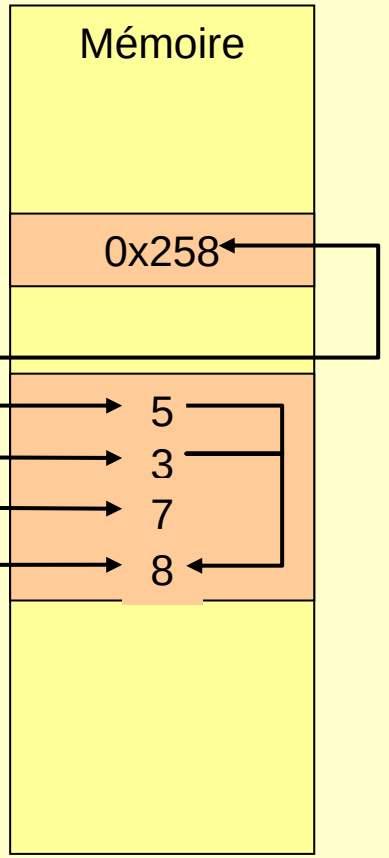
```
public class Tab1
{
  public static void main (String args[ ])
  {
    int tab[ ];
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```



Les tableaux (6)

Tab1.java

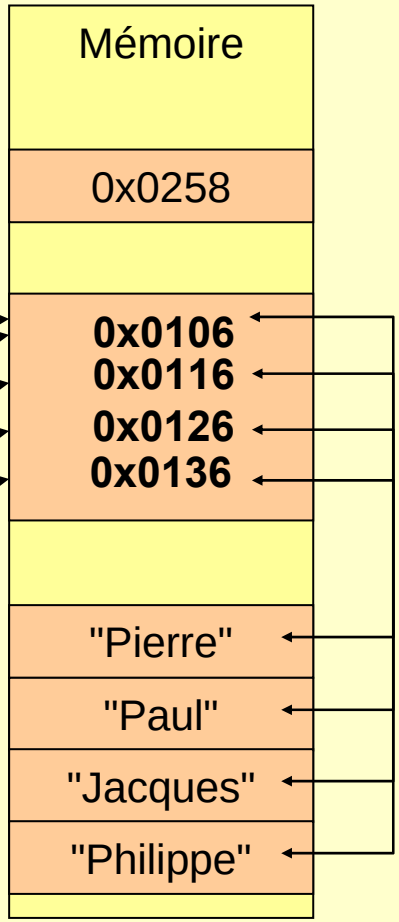
```
public class Tab1
{
  public static void main (String args[ ])
  {
    int tab[ ] ;
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```



Les tableaux (7)

Tab2.java

```
public class Tab2
{
  public static void main (String args[ ])
  {
    String chaines[ ] ;
 chaines = new String[4];
 chaines[0]=new String("Pierre");
 chaines[1]=new String("Paul");
 chaines[2]=new String("Jacques");
 chaines[3]=new String("Philippe");
  }
}
```



Les tableaux (8)

Tab2.java

```
public class Tab2
{
  public static void main (String args[ ])
  {
    String chaines[ ] ;
 chaines = new String[4];
 chaines[0]=new String("Pierre");
 chaines[1]=new String("Paul");
 chaines[2]=new String("Jacques");
 chaines[3]=new String("Philippe");
    for (int i=0;i< chaines.length;i++)
     {
       System.out.println(  " chaines[ " + i + "  ] = "
                                  + chaines[i]);
     }
 }
}
```

Modification du programme pour afficher le contenu du tableau.

Les tableaux (9)

- Copie de la référence d'un tableau:
  - En Java, un tableau est un objet et le nom du tableau contient la référence de cet objet.
  - L'affectation d'un tableau à un autre ne le duplique pas, mais affecte simplement une autre référence au même objet.
- **Exemple:**
  - **CopieRef.java** (voir page suivante).

Les tableaux (10)

```
public class CopieRef {
  public static void main(String[] args) {
    int[] a = {22, 44, 66, 55, 33};
    System.out.println("a: " + a);
    print(a);
    int[] aa;
    aa = a;
    System.out.println("aa: " + aa);
    print(aa);
    a[3] = 88;
    print(a);
    print(aa);
    aa[1] = 11;
    print(a);
    print(aa);
  }

  public static void print(int[] a) {
    for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Les tableaux (11)

Sortie du programme CopieRef.java:

```
a: [I@47858e
22 44 66 55 33
aa: [I@47858e
22 44 66 55 33
22 44 66 88 33
22 44 66 88 33
22 11 66 88 33
22 11 66 88 33
```

Les tableaux (12)

- Copie d'un tableau:
  - Pour copier le tableau lui-même au lieu de la référence, on doit copier chaque élément qui le compose.
  - **Copie intuitive:**
    - Supposons que l'on dispose de 2 tableaux a[ ] et aa[ ] et que l'on désire copier les éléments de a[ ] dans aa[ ]:
      - aa[0] = a[0];
      - aa[1] = a[1]; // etc.

Les tableaux (13)

- **Copie en utilisant la méthode arraycopy( ):**
  - La méthode arraycopy( ) de la classe System permet de copier rapidement un tableau:
  - **System.arraycopy(Src, SrcPos, Dest, DestPos, Length);**
    - Src: le tableau source.
    - SrcPos: l'index du premier élément du tableau source à copier.
    - Dest: le tableau cible.
    - DestPos: l'index où le premier élément doit être copié dans le tableau cible.
    - Length: le nombre d'éléments à copier.
  - **Exemple:**
    - **CopieTab.java** (voir page suivante).

Les tableaux (14)

```
public class CopieTab {
  public static void main(String[ ] args) {
    int[ ] a = {22, 33, 44, 55, 66, 77, 88, 99};
    System.out.println("a: " + a);
    print(a);
    int[ ] aa = new int[a.length];
    System.out.println("aa: " + aa);
    print(aa);
    System.arraycopy(a, 0, aa, 0, a.length);
    System.out.println("aa: " + aa);
    print(aa);
    aa[1] = 11;
    print(a);
    print(aa);
    aa = new int[12];
    System.arraycopy(a, 0, aa, 3, 8);
    System.out.println("aa: " + aa);
    print(aa);
    System.arraycopy(aa, 3, aa, 1, 5);
    System.out.println("aa: " + aa);
    print(aa);
  }

  public static void print(int[ ] a) {
    for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Les tableaux (15)

Sortie du programme CopieTab.java:

```
a: [I@47858e
22 33 44 55 66 77 88 99
aa: [I@19134f4
0 0 0 0 0 0 0 0
aa: [I@19134f4
22 33 44 55 66 77 88 99
22 33 44 55 66 77 88 99
22 11 44 55 66 77 88 99
aa: [I@2bbd86
0 0 0 22 33 44 55 66 77 88 99 0
aa: [I@2bbd86
0 22 33 44 55 66 55 66 77 88 99 0
```

Les tableaux (16)

- La classe Arrays:
  - La classe **Arrays** permet de traiter les tableaux. Elle est définie dans le paquetage **java.util**.
  - Cette classe contient plusieurs méthodes utilitaires:
    - **sort(a)**: permet de trier les éléments du tableau a.
    - **equals(a, b)**: renvoie « true » si les tableaux a et b sont égaux.
    - etc.
  - **Exemple:**
    - **TestArrays.java** (voir page suivante).

Les tableaux (17)

```
import java.util.Arrays;

public class TestArrays {
  public static void main(String[] args) {
    int[] a = {44, 77, 55, 22, 99, 88, 33, 66};
    print(a);
    Arrays.sort(a);
    print(a);
    int[] b = new int[8];
    print(b);
    System.arraycopy(a, 0, b, 0, a.length);
    print(b);
    System.out.println("Arrays.equals(a,b): " + Arrays.equals(a,b));
  }

  public static void print(int[] a) {
    for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Les tableaux (18)

Sortie du programme TestArrays.java:

```
44 77 55 22 99 88 33 66
22 33 44 55 66 77 88 99
0 0 0 0 0 0 0 0
22 33 44 55 66 77 88 99
Arrays.equals(a,b): true
```

Les tableaux (19)

- Tableaux bidimensionnels:
  - Les éléments d'un tableau peuvent être n'importe quel type d'objet.
  - Ces éléments peuvent être donc des tableaux. Le cas échéant, nous obtenons ce qu'on appelle un **tableau bidimensionnel** (ou **matrice**).
  - Syntaxe de déclaration et de création:

Type[ ][ ] Nom_Tableau = new Type[Nombre_Lignes][Nombre_Colonnes];

  - **Exemple:**
    - **UneMatrice.java** (voir page suivante).

Les tableaux (20)

```
public class UneMatrice {
  public static void main(String[ ] args) {
    int[ ][ ] M = new int[3][5];
    for (int i=0; i<3; i++)
      for (int j=0; j<5; j++)
        M[i][j] = 10*(i+1) + j;
    for (int i=0; i<3; i++)
      print(M[i]);
    System.out.println("M[2][4] = " + M[2][4]);
  }

  public static void print(int[ ] a) {
    for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
}
```

Les tableaux (21)

Sortie du programme UneMatrice.java:

```
10 11 12 13 14
20 21 22 23 24
30 31 32 33 34
M[2][4] = 34
```

Les chaînes: La classe String (1)

- Attention ce n'est pas un type de base. Il s'agit d'une classe défini dans l'API Java (Dans le package java.lang).
- Déclaration:
  - **String Nom_chaine;**
- Déclaration et initialisation:
  - **String s = "aaa";** // s contient la chaîne "aaa"
  - **String s = new String("aaa");** // s contient la chaîne "aaa"

Les chaînes: La classe String (2)

- La concaténation:
  - l'opérateur + permet de concaténer un nombre fini de chaînes de type String:
  - **Exemple:**
    - String str1 = "Bonjour ! ";
    - String str2 = null;
    - str2 = "Comment vas-tu ?";
    - String str3 = str1 + str2; /* str3 contient "Bonjour ! Comment vas-tu ?" */

Les chaînes: La classe String (3)

- Longueur d'un objet String:
  - La méthode **length( )** renvoie la longueur d'une chaîne de type String.
  - **Exemple:**
    - String str1 = "bonjour";
    - int n = str1.length( ); // n vaut 7

Les chaînes: La classe String (4)

- Les sous-chaînes:
  - La méthode **substring(int debut, int fin)** permet d'extraire une sous-chaîne à partir d'une chaîne de type String.
    - Extraction de la sous-chaîne depuis la position debut jusqu'à la position fin non-comprise.
    - **Exemple:**

```
String str1 = "bonjour";
String str2 = str1.substring(0,3); // str2 contient la valeur "bon"
```

  - **Remarque:**
    - Le premier caractère d'une chaîne occupe la position 0.
    - Le deuxième caractère occupe la position 1 et ainsi de suite.

Les chaînes: La classe String (5)

- Récupération d'un caractère dans une chaîne:
  - La méthode **charAt(int Pos)** permet de renvoyer le caractère situé à la position **Pos** dans la chaîne de caractères appelant cette méthode.
  - **Exemple:**

```
String str1 = "bonjour";
char UNj = str1.charAt(3); // UNj contient le caractère 'j'
```

Les chaînes: La classe String (6)

- Modification de la casse des lettres dans une chaîne:
  - La méthode **toLowerCase( )** permet de transformer toutes les lettres majuscules en des lettres minuscules dans la chaîne appelant cette méthode.
  - **Exemple:**

    String str1 = "BONJOUR";

    String str2 = str1.toLowerCase( ) // str2 contient la chaîne "bonjour"
  - De même, la méthode **toUpperCase( )** permet de transformer toutes les lettres minuscules en des lettres majuscules dans la chaîne appelant cette méthode.

Les chaînes: La classe String (7)

- Modification des objets String:
  - Les objets String sont **immuables** en Java, ce qui signifie qu'ils ne peuvent pas être modifiés.
  - **Remarque:**
    - Quand on a besoin de manipuler directement les chaînes de caractères, on peut utiliser la classe **StringBuffer**.

Les chaînes: La classe String (8)

Exemple récapitulatif:

```
class TestClasseString {
  public static void main(String[ ] args) {
    String alphabet = "ABCITALIEFGHIJKLMNOPQRSTUVWXYZ";
    System.out.println("Cette chaine est: " + alphabet);
    System.out.println("Sa longueur est: " + alphabet.length());
    System.out.println("Le caractère de l'indice 4 est: "
      + alphabet.charAt(4));
    System.out.println("Sa version en minuscules est: "
      + alphabet.toLowerCase());
    System.out.println("Cette chaine est toujours: " + alphabet);
  }
}
```

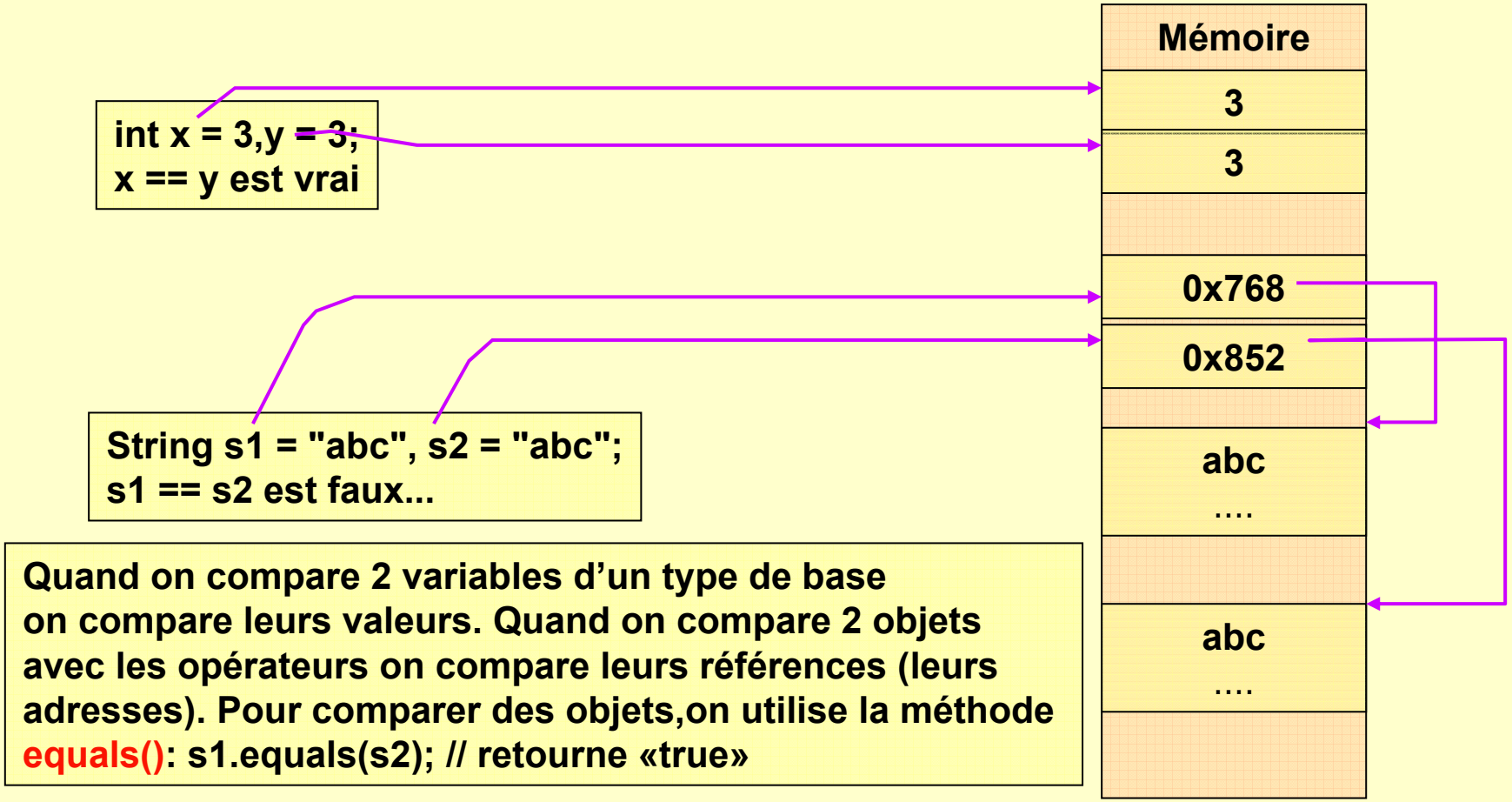
Les chaînes: La classe String (9)

Sortie du programme TestClasseString.java:

Cette chaine est: ABCITALIEFGHIJKLMNOPQRSTUVWXYZ
Sa longueur est: 26
Le caractère de l'indice 4 est: E
Sa version en minuscules est: abcdefghijklmnopqrstuvwxyz
Cette chaine est toujours:
ABCITALIEFGHIJKLMNOPQRSTUVWXYZ

Les chaînes: La classe String (10)

- Différence entre objet (ex. String) et type de base:



Les nombres aléatoires: La classe Random

- La classe **Random** est une classe utilitaire de l'API Java permettant de générer des nombres aléatoires.
- La classe Random est définie dans le paquetage java.util.
- Les méthodes les plus utilisées de la classe Random:
  - La méthode **nextInt( )** permet de renvoyer des nombres aléatoires entiers répartis de façon uniforme dans l'intervalle comportant les entiers de type int (actuellement, de -2 147 483 648 à 2 147 483 647).
  - La méthode **nextInt(*n*)** permet de renvoyer des nombres aléatoires entiers répartis de façon uniforme dans l'intervalle de **0** à ***n* – 1**.
  - La méthode **nextDouble( )** permet de renvoyer des nombres aléatoires de type double qui sont répartis uniformément dans l'intervalle de 0 à 1.

Les nombres aléatoires: La classe Random

- **Exemple:**

```
import java.util.Random;

public class EntierAleatoire {
  public static void main(String[ ] args) {
    Random random = new Random( );
    int n = random.nextInt( );
    System.out.println("n = " + n);
    if (n < 0)
              System.out.println("****  n < 0");
    else
               System.out.println("****  n >= 0");
  }
}
```

Méthodes, classes et objets

Méthodes (1)

- Une **méthode** est une séquence de déclarations et d'instructions exécutables.
- En Java, toute instruction exécutable doit se trouver dans une méthode.
- Les méthodes constituent les actions (ou traitements) à exécuter sur les objets.
- La **méthode principale** en Java est la méthode **main( )** qui constitue le point d'entrée d'un programme Java.

Méthodes (2)

- Syntaxe de déclaration d'une méthode:
  - **Modificateurs Type_Retour Nom_Méthode (Liste_Paramètres) Clause {Séquence_Instructions}**
    - **Modificateurs** (ex. public, static) spécifie la liste des modificateurs éventuels de la méthode. Ces modificateurs préscrivent la façon d'accéder à la méthode. Ils sont facultatifs.
    - **Type_Retour** spécifie le type de la valeur renvoyée par la méthode. Si Type_Retour n'est pas « void », alors au moins une de ses instructions doit inclure une instruction return qui renvoie une expression dont le type correspond à celui de Type_Retour.
    - **Nom_Méthode** spécifie le nom ou l'identificateur de la méthode. Par convention, Nom_Méthode est commencé par une lettre minuscule.

Méthodes (3)

- Syntaxe de déclaration d'une méthode (suite):
  - **Liste_Paramètres** spécifie la liste des paramètres (ou arguments) de la méthode ainsi que leurs types.
  - **Clause** spécifie une clause spécifique concernant la méthode (ex. la clause throws IOException qui concerne la méthode main( ) lorsque celle-ci doit contenir des instructions pour l'entrée interactive des données). Clause est facultative.
  - **Séquence_Instructions** spécifie la séquence d'instructions constituant le corps de la méthode.
- **Exemple:**
  - **TestCube.java** est un programme qui teste la méthode cube( ) définie dans la classe TestCube (voir page suivante).

Méthodes (4)

```
public class TestCube {
  public static void main(String[ ] args) {
    for (int i = 0; i < 6; i++)
      System.out.println(i + "\t" + cube(i));
  }

  static int cube(int n) {
    return n*n*n;
  }
}
```

Méthodes (5)

Sortie du programme TestCube.java:

```
0    0
1    1
2    8
3    27
4    64
5    125
```

Méthodes (6)

- Les variables locales:
  - Une **variable locale** est déclarée dans une méthode.
  - Elle peut être utilisée uniquement à l'intérieur de la méthode où elle est déclarée. Sa portée est donc limitée à cette méthode.
  - **Remarque:**
    - Selon l'emplacement des variables, on peut utiliser un seul nom pour plusieurs variables d'un même programme.
  - **Exemple:**
    - **VarLocale.java** (voir page suivante).

Méthodes (7)

```
public class VarLocale {
  public static void main(String[ ] args) {
    for (int n = 0; n < 9; n++)
      System.out.println("f(" + n + ") = " + f(n));
  }
  static long f(int n) {
    long f = 1;
    while (n > 1)
      f *= n--;
    return f;
  }
}
```

Méthodes (8)

- La récursivité:
  - Une méthode qui s'appelle elle-même est dite **récursive**.
  - Une fonction récursive est composée de deux parties principales:
    A. La base qui définit la fonction pour les premières valeurs.
    B. La partie récursive qui définit la fonction pour les autres valeurs.
  - **Exemple:**
    - Le programme **FactoRec.java** implémente une méthode f qui calcule la fonction factorielle de façon récursive (voir page suivante).

Méthodes (9)

```
public class FactoRec {
  public static void main(String[ ] args) {
    for (int i = 0; i < 9; i++)
      System.out.println("f(" + i + ") = " + f(i));
  }
  static long f(int n) {
    if (n < 2) return 1;
    return n*f(n-1);
  }
}
```

Méthodes (10)

- Les méthodes booléennes:
  - Une **méthode booléennes** se contente de renvoyer une valeur de type boolean (« true » ou « false »).
  - Les méthodes booléennes sont généralement utilisées dans des expressions où une condition est attendue.
  - Les méthodes booléennes sont parfois qualifiées de prédicats.
  - **Exemple:**
    - Le programme **TestNbrePremier.java** implémente une méthode nommée nbrePremier(int *n*) permettant de vérifier si le paramètre *n* est un nombre premier ou non (voir page suivante).

Méthodes (11)

```
public class TestNbrePremier {
  public static void main(String[ ] args) {
    for (int i = 0; i < 100; i++)
      if (nbrePremier(i)) System.out.print(i + " ");
    System.out.println();
  }

  static boolean nbrePremier(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n%2 == 0) return false;
    for (int d = 3; d <= Math.sqrt(n); d += 2)
      if (n%d == 0) return false;
    return true;
  }
}
```

Méthodes (12)

Sortie du programme TestNbrePremier.java:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Méthodes (13)

- La surcharge:
  - La **surcharge** signifie que plusieurs méthodes peuvent porter le même nom tant qu'elles n'ont pas le même nombre de paramètres ou des paramètres de types différents.
  - La signature d'une méthode est composée de son nom et de sa liste de paramètres (y compris les types de ces paramètres).
  - Les méthodes surchargées doivent avoir des signatures différentes.
  - **Exemple:**
    - **SurchargeMax.java** (voir page suivante).

Méthodes (14)

```
import java.util.Random;

public class SurchargeMax {
  public static void main(String[ ] args) {
    Random random = new Random();
    for (int i = 0; i < 4; i++) {
      int a = random.nextInt(100);
      int b = random.nextInt(100);
      int c = random.nextInt(100);
      System.out.println("max(" + a + "," + b + ") = " + max(a, b));
      System.out.println("max(" + a + "," + b + "," + c  + ") = " + max(a, b, c));
    }
  }

  static int max(int m, int n) {
    if (m > n) return m;
    return n;
  }

  static int max(int n1, int n2, int n3) {
    return max( max(n1, n2), n3);
  }
}
```

Classes et objets (1)

- Java est un langage orienté objet;
  - Un programme Java est structurée en **classes** qui spécifient le comportement des objets.
  - Les **objets** contrôlent les actions (méthodes) du programme.
  - Un programme Java est une collection d'un ou de plusieurs fichiers texte qui définissent les classes Java. L'une de ces classes doit être déclarée comme public et doit contenir la méthode main ( ); il s'agit de la classe principale.
- Une **classe** est un type de données qui n'est pas prédéfini dans le langage, mais qui est défini par le concepteur de la classe.

Classes et objets (2)

- Par convention, les noms de classes commencent généralement par une lettre majuscule.
- L'**instanciation** d'une classe consiste à créer des objets dont le type correspond à celui défini par cette classe.
- Un **objet** est une **instance de classe** qui possède les caractéristiques de sa classe.
- Une classe est composée de trois types de membres:
  1. Les **champs** qui spécifient le type de données contenues dans les objets.
  2. Les **méthodes** qui spécifient les opérations susceptibles d'être effectuées par les objets.
  3. Les **constructeurs** (méthodes particulières) qui spécifient le mode de création des objets.

Classes et objets (3)

- Syntaxe générale de la définition d'une classe:
  - **Modificateurs class Nom_Classe Clauses {**

    \\ Champs…;

    \\ Constructeurs {…}

    \\ Méthodes {…}

    }
  - Les **Modificateurs** spécifient le mode d'accès à la classe (ex. public, abstract). Ils sont facultatifs.
  - **Nom_Classe** spécifie le nom de la classe.
  - Les **Clauses** apparaissent, par exemple, lorsque la classe est incluse dans une hiérarchie d'héritage.

Classes et objets (4)

- Syntaxe générale de la déclaration d'un champ:
  - **Modificateurs Type_Champ Nom_Champ;**
    - Les **Modificateurs** spécifient le mode d'accès au champ (ex. static, public, private).
    - **Type_Champ** spécifie le type de données contenues dans le champ.
    - **Nom_Champ** spécifie le nom identifiant le champ.

Classes et objets: Les modificateurs (1)

- Modificateurs de classes:

Modificateur	Signification
abstract	La classe ne peut pas être instanciée.
public	Les membres de la classe sont accessibles depuis toutes les autres classes.

Classes et objets: Les modificateurs (2)

- Modificateurs de champs:

Modificateur	Signification
final	Il doit être initialisé et ne peut pas être modifié.
private	Il n'est accessible que depuis sa propre classe.
public	Il est accessible depuis toutes les classes.
static	Le même stockage est utilisé pour toutes les instances de la classe.

Classes et objets: Les modificateurs (3)

- Modificateurs de constructeurs:

Modificateur	Signification
private	Il est uniquement accessible depuis sa propre classe.
public	Il est accessible depuis toutes les classes.

Classes et objets: Les modificateurs (4)

- Modificateurs de méthodes:

Modificateur	Signification
private	Elle est uniquement accessible depuis sa propre classe.
public	Elle est accessible depuis toutes les classes.
static	Elle est une méthode de classe. Il n'est pas nécessaire de créer un objet pour l'appeler. La méthode peut être appelée en utilisant le nom de la classe.

Classes et objets: Les modificateurs (5)

- Modificateurs de variables locales:

Modificateur	Signification
final	Elle doit être initialisée et ne peut pas être modifiée.

Classes et objets: Les constructeurs (1)

- Les **constructeurs** sont des méthodes particulières permettent de créer des objets.
- Le corps d'un constructeur contient généralement les instructions qui permettent d'initialiser les champs de l'objet créé.
- A l'instar d'une méthode, un constructeur peut avoir:
  - Des modificateurs.
  - Des paramètres.
  - Des variables locales.
  - Des instructions exécutables.

Classes et objets: Les constructeurs (2)

- Contrairement à une méthode, un constructeur se caractérise par les propriétés suivantes:
  1. Il porte le même nom que sa classe.
  2. Il n'a aucun type de retour.
  3. Il est appelé par l'opérateur **new**.
  4. Il peut appeler d'autres constructeurs avec le mot clé **this**.

Classes et objets: Les constructeurs (3)

- Le constructeur par défaut:
  - Le **constructeur par défaut** a une liste de paramètres vide.
  - Syntaxe générale de définition d'un constructeur par défaut:
    - **Modificateur Nom_Classe( ) {**

      \\ Corps du constructeur

      }
  - **Remarques:**
    - Si la classe ne possède pas de constructeurs déclarés explicitement, le compilateur définit un constructeur par défaut public pour créer des objets. Ce constructeur est appelé le constructeur par défaut implicite.
    - Dans ce cas, le constructeur par défaut implicite initialise automatiquement tous les champs des objets avec leurs valeurs initiales par défaut.

Classes et objets: Les constructeurs (4)

- Le constructeur de copie:
  - Le **constructeur de copie** est un constructeur dont le seul paramètre est une référence à un objet de sa classe.
  - Syntaxe générale de définition d'un constructeur de copie:
    - **Modificateur Nom_Classe (Nom_Classe Nom_paramètre)**
      {
      \\ Corps du constructeur
      }
  - **Remarque:**
    - Le constructeur de copie est généralement utilisé afin de dupliquer un objet existant de la classe.

Classes et objets: Les constructeurs (5)

- Java nous permet de créer d'autres constructeurs qui sont différents du constructeur par défaut et du constructeur de copie.
- **Remarque:**
  - Lorsque la classe contient un constructeur défini explicitement, le constructeur par défaut implicite est détruit automatiquement.
- Syntaxe de création d'un objet:

Nom_classe Nom_Objet = new Nom_Classe (Paramètres_Constructeur);

Classes et objets: Appel de méthodes

- En langage Java, les méthodes sont appelées par les objets.
- L'**appel** est soit **implicite**, soit **explicite**:
  - Lorsque l'objet appelant la méthode n'est pas spécifié explicitement, la méthode est alors appelée implicitement par l'instance courante de sa classe.
  - On parle d'appel explicite lorsque la méthode est appelée par un objet créé explicitement. Dans ce cas, l'appel est réalisé par la syntaxe suivante:

    Nom_Objet.Nom_Méthode(Paramètres_Effectifs);

Classes et objets: Exemple récapitulatif (1)

- **Problème**:
  - Créer un programme Java permettant de décrire les propriétés mathématiques d'un point géométrique dans le plan cartésien (représentation, abscisse, ordonnée,…).

Classes et objets: Exemple récapitulatif (2)

- **Solution**:
  - Nous pouvons caractériser les points géométriques en définissant une classe appelée Point dont les objets représentent ces points dans le plan cartésien.
  - Les méthodes de la classe Point vont spécifier le comportement des objets de type Point, c'est-à-dire les propriétés mathématiques des points géométriques représentés par ces objets.

Classes et objets: Exemple récapitulatif (3)

```
public class Point {
  // Objets représentant un point dans le plan

  private double x, y;

  public Point( ) {  // Constructeur par défaut
    x = 1.0;
    y = -1.0;
  }

  public Point(Point p) {  // Constructeur de copie
    x = p.x;
    y = p.y;
  }

  public Point(double a, double b) {  // Un autre constructeur
    x = a;
    y = b;
  }
…
```

Classes et objets: Exemple récapitulatif (4)

```
public double abscisse( ) {  // Retourne l'abscisse de l'objet Point appelant
   return x;
 }

 public double ordonnee( ) {  // Retourne l'ordonnée de l'objet Point appelant
   return y;
 }

 public Point projectionX( ) {  // Retourne la projection sur l'axe des X
   return new Point(x, 0);
 }

 public Point projectionY( ) {  // Retourne la projection sur l'axe des Y
   return new Point(0, y);
 }

 public boolean equals(Point p) {  // Vérifie si 2 points sont confondus dans le plan
   return (x == p.x && y == p.y);
 }
…
```

Classes et objets: Exemple récapitulatif (5)

```
public static double distance(Point p1, Point p2) {  // Retourne la distance entre 2 points
   double dx = p2.x - p1.x;
   double dy = p2.y - p1.y;
   return Math.sqrt(dx*dx + dy*dy);
 }



/** La méthode toString( ) est appelée automatiquement par la méthode println( ).
    Elle permet à celle-ci d'afficher la valeur contenue dans l'objet passé en argument
    au lieu d'afficher sa référence */

 public String toString( ) {
   return new String("(" + x + ", "  + y + ")");
  }
} // Fin de définition de la classe Point (fichier Point.java)
```

Classes et objets: Exemple récapitulatif (6)

- Deux alternatives pour l'exécution de la classe Point:
  a) Insertion de la méthode main( ) dans le fichier contenant la définition de la classe Point (Point.java).
  b) Création d'un autre fichier (TestPoint.java) dans le même dossier (ou projet) où se trouve le fichier Point.java. Puis, définition de la classe TestPoint contenant la méthode main( ) dans le fichier TestPoint.java.
- **Remarque:**
  - Etapes de compilation et d'exécution concernant la 2ème alternative:
    - Compilation du fichier Point.java.
    - Compilation, puis exécution du fichier TestPoint.java.

Classes et objets: Exemple récapitulatif (7)

```
public class TestPoint {
  public static void main(String[ ] args) {
    Point p = new Point(2.0, -3.0);
    System.out.println("p: " + p);
    System.out.println("p.abscisse( ): " + p. abscisse( ));
    System.out.println("p.ordonnee( ): " + p. ordonnee( ));
    Point q = new Point(7.0, 9.0);
    System.out.println("q: " + q);
    Point r = new Point(p);
    System.out.println("r: " + r);
    System.out.println("q.equals(p): " + q.equals(p));
    System.out.println("r.equals(p): " + r.equals(p));
    System.out.println("distance(p,q): "  + Point.distance(p,q));
  }
}
```

Classes et objets: Exemple récapitulatif (8)

Sortie du programme TestPoint.java:

```
p: (2.0, -3.0)
p.abscisse( ): 2.0
p.ordonnee( ): -3.0
q: (7.0, 9.0)
r: (2.0, -3.0)
q.equals(p): false
r.equals(p): true
distance(p,q): 13.0
```

Composition et héritage

Composition (1)

- La **composition** consiste à créer une classe dont certains champs correspondent à des objets d'autres classes.
- Dans ce cas, on dit que:
  - Les objets utilisés dans la définition de la classe composée sont des objets **composants**.
  - La classe composée est une **agrégation** des objets composants.
- **Remarque:**
  - Les objets de la classe String sont les plus couramment utilisés en tant qu'objets composants des autres classes.

Composition (2)

- **Exemple 1:**
  - La classe **NomComplet** est une agrégation de 3 objets de la classe String (prenom, alias, nom).
  - Cette classe possède:
    a. 3 constructeurs (le constructeur par défaut et 2 autres constructeurs).
    b. 3 accesseurs (donnerPrenom( ), donnerAlias( ), donnerNom( )).
    c. 3 mutateurs (modifierPrenom( ), modifierAlias( ), modifierNom( )).
    d. La méthode toString( ) qui permet l'affichage de la chaîne contenant la valeur de l'objet passé en argument à la méthode println( ).
  - **Remarque:**
    - Le 2ème constructeur possède 2 arguments qui font références aux 2 champs prenom et nom. Ce constructeur omet donc le champ alias.

Composition (3)

- **Exemple 1 (NomComplet.java):**

```
public class NomComplet {
  public String prenom;
  public String alias;
  public String nom;

  public NomComplet( ) {  // Constructeur par défaut explicite
  }
  public NomComplet(String prenom, String nom) { // Constructeur omettant le champ alias
    this.prenom = prenom;
    this.nom = nom;
  }
 public NomComplet(String prenom, String alias, String nom) { // Constructeur initialisant tous les champs
    this(prenom,nom);
    this.alias = alias;
  }
```

Composition (4)

- **Exemple 1 (NomComplet.java) -Suite:**

```
public String donnerPrenom( ) {
   return prenom;
 }

 public String donnerAlias( ) {
   return alias;
 }

 public String donnerNom( ) {
   return nom;
 }
```

Composition (5)

- **Exemple 1 (NomComplet.java) -Suite:**

```
public void modifierPrenom(String prenom) {
   this.prenom = prenom;
 }

 public void modifierAlias(String alias) {
   this.alias = alias;
 }

 public void modifierNom(String nom) {
   this.nom = nom;
 }
```

Composition (6)

- **Exemple 1 (NomComplet.java) -Fin:**

```
public String toString( ) {
   String s = new String( );
   if (prenom != null) s += prenom + " ";
   if (alias != null) s += alias + " ";
   if (nom != null) s += nom + " ";
   return s;
 }
```

Composition (7)

- **Exemple 1 (TestNomComplet.java):**

```
class TestNomComplet {
  public static void main(String[ ] args) {
    NomComplet medecin = new NomComplet("Francis", "Harry Compton", "CRICK");
    System.out.println(medecin + " a eu le prix Nobel de Médecine en 1962.");
    System.out.println("Son prénom est " + medecin.donnerPrenom( ));
    System.out.println("Son alias est " + medecin.donnerAlias( ));
    System.out.println("Son nom est " + medecin.donnerNom( ));
    NomComplet ecrivain = new NomComplet("Najib", "MAHFOUD");
    System.out.println(ecrivain + " a eu le prix Nobel de Littérature en 1988.");
    System.out.println("Son prénom est " +ecrivain.donnerPrenom( ));
    System.out.println("Son alias est " + ecrivain.donnerAlias( ));
    System.out.println("Son nom est " + ecrivain.donnerNom( ));
  }
}
```

Composition (8)

- **Sortie du programme TestNomComplet.java:**

```
Francis Harry Compton CRICK a eu le prix Nobel de Médecine en 1962.
Son prénom est Francis
Son alias est Harry Compton
Son nom est CRICK
Najib MAHFOUD a eu le prix Nobel de Littérature en 1988.
Son prénom est Najib
Son alias est null
Son nom est MAHFOUD
```

Composition (9)

- **Exemple 2:**
  - La classe **Personne** implémente l'identité d'une personne donnée (prénom, alias, nom, sexe, code identifiant). Pour cela, elle utilise un champ de type NomComplet pour pouvoir réutiliser le code de la classe NomComplet.

Composition (10)

- **Exemple 2 (Personne.java):**

```
public class Personne {
  public NomComplet nom_complet;
  public char sexe;   // 'H' ou 'F'
  public String id;  // ex. Numéro de la CIN

  public Personne(NomComplet nom_complet, char sexe) {
    this.nom_complet = nom_complet;
    this.sexe = sexe;
  }
  public Personne(NomComplet nom_complet, char sexe, String id) {
    this.nom_complet = nom_complet;
    this.sexe = sexe;
    this.id = id;
  }
```

Composition (11)

- **Exemple 2 (Personne.java) -Suite:**

```
public NomComplet donnerNomComplet( ) {
   return nom_complet;
 }

 public char donnerSexe( ) {
   return sexe;
 }

 public String donnerId( ) {
   return id;
 }
```

Composition (12)

- **Exemple 2 (Personne.java) -Fin:**

```
public void modifierId(String id) {
    this.id = id;
  }

  /** La méthode toString( ) permet également d'afficher la valeur du champ id
      si celui-ci possède une valeur concrète */
  public String toString( ) {
    String s = new String(nom_complet + " (sexe: " + sexe);
    if (id != null) s += "; id: " + id;
    s += ")";
    return s;
  }
}
```

Composition (13)

- **Exemple 2 (TestPersonne.java):**

```
class TestPersonne {
  public static void main(String[ ] args) {
    NomComplet nomPersonne1 = new NomComplet("Robert", "LEE");
    Personne personne1 = new Personne(nomPersonne1, 'H');
    System.out.println("Première personne: " + personne1);
    personne1.nom_complet.modifierAlias("Edward");
    System.out.println("Première personne: " + personne1);
    Personne personne2 = new Personne(new NomComplet("Ann", "BAKER"), 'F');
    System.out.println("Deuxième personne: " + personne2);
    personne2.modifierId("C053011736");
    System.out.println("Deuxième personne: " + personne2);
  }
}
```

Composition (14)

- **Sortie du programme TestPersonne.java:**

```
Première personne: Robert LEE (sexe: M)
Première personne: Robert Edward LEE (sexe: M)
Deuxième personne: Ann BAKER (sexe: F)
Deuxième personne: Ann BAKER (sexe: F; id: C053011736)
```

Composition: Les classes récursives (1)

- Une **classe récursive** est une classe qui a certains champs qui font références aux objets de la classe elle-même.
- **Utilité:**
  - Les classes récursives sont particulièrement utiles pour créer des structures liées susceptibles de représenter des relations complexes.
  - **Exemple:**
    - Les arbres généalogiques. Par exemple, la classe Personne peut être modifiée en lui ajoutant 2 champs de type Personne (pere, mere) ainsi que des mutateurs qui permettent d'attribuer des valeurs aux champs ajoutés.
    - Voir la **2ème version de la classe Personne** (page suivante).

Composition: Les classes récursives (2)

- **Exemple (Personne.java) -2ème version:**

```
public class Personne {
  public NomComplet nom_complet;
  public char sexe;   // 'H' ou 'F'
  public String id;  // ex. Numéro de la CIN
  public Personne mere;
  public Personne pere;
  public static final String deuxEspaces = "  ";
  public static String tab = "";              // tab est initialisé à la chaîne vide (null)

  // Mêmes constructeurs que ceux de la première version
  …
  // Mêmes accesseurs donnerNomComplet( ), donnerSexe( ) et donnerId( ) que ceux de la première version
  …
```

Composition: Les classes récursives (3)

- **Exemple (Personne.java) -2ème version -Suite:**

```
 /** Mutateur permettant d'attribuer une valeur concrète au champ mere */
public void ajouterMere(Personne mere) {
   this.mere = mere;
 }

 /** Mutateur permettant d'attribuer une valeur concrète au champ pere */
  public void ajouterPere(Personne pere) {
   this.pere = pere;
 }
```

Composition: Les classes récursives (4)

- **Exemple (Personne.java) -2ème version -Fin:**

```
/** La méthode toString( ) est modifiée de telle sorte à avoir un affichage conforme
    à un arbre généalogique */

public String toString( ) {
    String s = new String(nom_complet + " (" + sexe + ")");
    if (id != null) s += "; id: " + id;
    s += "\n";                                  // ajoute un saut de ligne
    if (mere != null) {
      tab += deuxEspaces;                       // ajoute deux espaces vides
      s += tab + "Mère: " + mere;
      tab = tab.substring(2);                   // supprime deux espaces vides
    }
    if (pere != null) {
      tab += deuxEspaces;
      s += tab + "Père: " + pere;
      tab = tab.substring(2);
    }
    return s;
  }
}
```

Composition: Les classes récursives (5)

- **Exemple (TestPersonne.java) -2ème version:**

```
class TestPersonne {
  public static void main(String[ ] args) {
    Personne ww = new Personne(new NomComplet("William", "Windsor"), 'H');
    Personne cw = new Personne(new NomComplet("Charles", "Windsor"), 'H');
    Personne ds = new Personne(new NomComplet("Diana", "Spencer"), 'F');
    Personne es = new Personne(new NomComplet("Edward", "Spencer"), 'H');
    Personne ew = new Personne(new NomComplet("Elizabeth", "Windsor"), 'F');
    Personne pm = new Personne(new NomComplet("Philip", "Mountbatten"), 'H');
    Personne eb = new Personne(new NomComplet("Elizabeth", "Bowes-Lyon"), 'F');
    Personne gw = new Personne(new NomComplet("George", "Windsor"), 'H');
    ww.ajouterPere(cw);
    ww.ajouterMere(ds);
    ds.ajouterPere(es);
    cw.ajouterMere(ew);
    cw.ajouterPere(pm);
    ew.ajouterMere(eb);
    ew.ajouterPere(gw);
    System.out.println(ww);
  }
}
```

Composition: Les classes récursives (6)

- **Sortie du programmeTestPersonne.java -2ème version:**

```
William Windsor (H)
  Mère: Diana Spencer (F)
    Père: Edward Spencer (H)
  Père: Charles Windsor (H)
    Mère: Elizabeth Windsor (F)
      Mère: Elizabeth Bowes-Lyon (F)
      Père: George Windsor (H)
    Père: Philip Mountbatten (H)
```

Héritage (1)

- L'**héritage** consiste à créer une nouvelle classe à partir d'une classe existante en lui ajoutant d'autres fonctionnalités:
  - Nous disons alors que la nouvelle classe **hérite** de toutes les fonctionnalités de la classe existante.
  - La nouvelle classe est dite alors une **extension** (ou **sous-classe** ou **classe enfant**) de la classe existante.
  - La classe existante est dite alors une **classe étendue** (ou **superclasse** ou **classe parent** ou **classe de base**) de la nouvelle classe.

Héritage (2)

- Syntaxe:

```
Modificateurs Nom_Sous-classe extends Nom_Superclasse {
    // Corps de la sous-classe
}
```

- Tous les champs et méthodes de la superclasse sont alors considérés comme des membres de la sous-classe à moins que certains de ces membres sont déclarés comme « **private** »:
  - S'il existe des membres de la superclasse déclarés comme « private », alors la sous-classe ne peut pas accéder à de tels membres.

Héritage (3)

- Un membre déclaré comme « **protected** » signifie que ce membre est accessible depuis sa classe et tous les sous-classes de sa classe.
- L'héritage s'applique aux champs et méthodes, mais pas aux constructeurs:
  - Une sous-classe doit avoir son propre constructeur puisque celui-ci doit toujours avoir le même nom que sa classe.
- Le constructeur de la sous-classe peut appeler le constructeur de la superclasse en utilisant le mot-clé **super** comme nom de ce dernier.

Héritage (4)

- **Exemple 1:**
  - Prenons la classe **Point** que nous avons déjà définie (voir le chapitre concernant les classes et les objets) et modifions-la de façon à déclarer ses champs x et y comme « protected » au lieu de « private ».
  - Nous allons créer une sous-classe appelée **PointNomme** de la classe Point. En plus des champs x et y hérités de la classe Point, la classe PointNomme possède un autre champ appelé nom qui réfère au nom d'un point.
  - La classe PointNomme hérite également des méthodes de sa superclasse Point et définit son propre constructeur et ses propres méthodes.

Héritage (5)

- **Exemple 1 (PointNomme.java):**

```
public class PointNomme extends Point {
  final private String nom;

  public PointNomme(double x, double y, String nom) {
    super(x, y);
    this.nom = nom;
  }

  public String nomPoint( ) {
    return nom;
  }

  public String toString( ) {
    return new String(nom + "(" + x + ", "  + y + ")");
  }
}
```

Héritage (6)

- **Exemple 1 (TestPointNomme.java):**

```
public class TestPointNomme {
  public static void main(String[ ] args) {
    PointNomme p = new PointNomme(2.0, -3.0, "P");
    System.out.println("p: " + p);
    System.out.println("p.nomPoint( ): " + p.nomPoint( ));
    System.out.println("p.abscisse( ): " + p.abscisse( ));
    PointNomme q = new PointNomme(2.0, -3.0, "Q");
    System.out.println("q: " + q);
    System.out.println("q.equals(p): " + q.equals(p));
  }
}
```

Héritage (7)

- **Sortie du programme TestPointNomme.java:**

```
p: P(2.0, -3.0)
p.nomPoint( ): P
p.abscisse( ): 2.0
q: Q(2.0, -3.0)
q.equals(p): true
```

Héritage (8)

- **Exemple 2:**
  - L'exemple suivant illustre le fait qu'une sous-classes ne peut pas accéder aux champs « private » de sa superclasse:

```
class ClasseX {
  private int m;

  public String toString( ) {
    return new String("(" + m  + ")");
  }
}
…
```

Héritage (9)

- **Exemple 2 -Suite:**

```
public class ClasseY extends ClasseX {
  private int n;

  public String toString( ) {
    return new String("(" + m  + "," + n + ")");  // ERREUR: Accès interdit à m
  }
}
…
```

Héritage (10)

- **Exemple 2 -Fin:**

```
class TestClasseY {
  public static void main(String[ ] args) {
    ClasseX x = new ClasseX( );
    System.out.println("x = " + x);
    ClasseY y = new ClasseY( );
    System.out.println("y = " + y);
  }
}
```

Héritage versus composition (1)

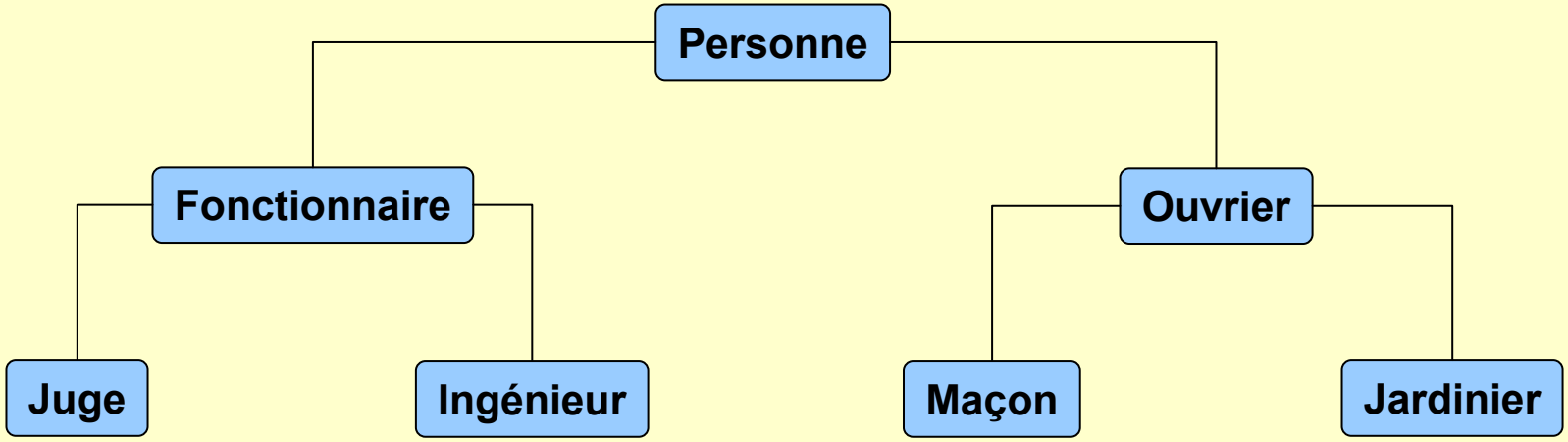
- L'**héritage** est synonyme de **spécialisation**:
  - Une sous-classe se spécialise en héritant de tous les champs et de toutes les méthodes de sa superclasse auxquels elle ajoute de nouveaux champs et/ou de nouvelles méthodes.
  - L'ensemble qui regroupe tous les objets de la sous-classe est donc un sous-ensemble de l'ensemble regroupant tous les objets de la superclasse.
- La **composition** est synonyme d'**agrégation**:
  - Une classe peut avoir des champs dont les types correspondent à d'autres classes. Le cas échéant, on dit que cette classe est une agrégation des classes qui la composent.

Héritage versus composition (2)

- Distinction entre héritage et composition:
  - **Héritage**: « **est un** ».
  - **Composition**: « **a un** ».
- **Exemple:**
  - Un point nommé (objet de la classe PointNomme) « est un » point (objet de type Point) qui, en plus, « a un » nom (objet de type String).

Héritage: Les hiérarchies de classes (1)

- Une classe peut avoir plusieurs sous-classes.
- **Exemple (hiérarchie de classes):**



Héritage: Les hiérarchies de classes (2)

- Dans une hiérarchie de classes, on dit que la classe Y est un **descendant** de la classe X s'il existe une séquence de classes commençant par Y et finissant par X dans laquelle chaque classe est la superclasse de celle qui la précède.
- Dans ce cas, on dit que la classe X est un **ancêtre** de la classe Y.
- **Exemple:**
  - Dans la hiérarchie de l'exemple précédent, la classe Jardinier est un descendant de la classe Personne.

Héritage: Les hiérarchies de classes (3)

- La classe Object:
  - Chaque classe Java est membre d'une vaste hiérarchie d'héritage dont la **racine** est la classe **Object**.
  - Chaque classe donc est une extension (directe ou indirecte) de la classe Object.
  - Toutes les classes héritent ainsi des méthodes définies dans cette classe racine. Parmi ces méthodes, on retrouve la méthode **equals( )** qui retourne « true » si et seulement les deux objets comparés ont la même référence.
  - Il est préférable donc que chaque classe spécifie sa propre méthode equals( ).

Héritage: Les classes abstraites (1)

- Une **classe abstraite** est une classe qui ne peut pas être instanciée.
- Une **méthode abstraite** est une méthode déclarée uniquement avec sa signature; elle n'a aucune implémentation.
- Une méthode abstraite doit être déclarée au sein d'une classe abstraite.
- Les classes et les méthodes sont déclarées comme abstraites à l'aide du modificateur **abstract**.

Héritage: Les classes abstraites (2)

- **Remarque:**
  - Pour pouvoir être instanciée, une sous-classe d'une classe abstraite doit redéfinir toutes les méthodes abstraites de sa superclasse;
    - Si l'une des méthodes n'est pas redéfinie de façon concrète, la sous-classe doit être déclarée comme « abstract ».

Héritage: Les classes abstraites (3)

- **Exemple:**
  - L'exemple suivant définit trois classes: La classe abstraite FormeGeometrique et les deux classes concrètes Circle et Carre. Les deux dernières sont des sous-classes de la première.

```
public abstract class FormeGeometrique {
  public abstract Point centre( );
  public abstract double diametre( );
  public abstract double surface( );
}
```

Héritage: Les classes abstraites (4)

- **Exemple -Suite:**

```
class Circle extends FormeGeometrique {
  private Point centre;
  private double rayon;

Circle(Point centre, double rayon) {
    this.centre = centre;
    this.rayon = rayon;
  }
public Point centre( ) {
    return centre;
  }
…
```

Héritage: Les classes abstraites (5)

- **Exemple -Suite:**

```
public double diametre( ) {
    return 2*rayon;
  }
public double surface( ) {
    return Math.PI*rayon*rayon;
  }
 public String toString( ) {
    return new String("{ centre = " + centre
      + ", rayon = " + rayon + "}");
  }
} // Fin de la classe Circle
```

Héritage: Les classes abstraites (6)

- **Exemple -Suite:**

```
class Carre extends FormeGeometrique {
  private Point coinNordOuest;
  private double cote;

  Carre(Point coinNordOuest, double cote) {
    this.coinNordOuest = coinNordOuest;
    this.cote = cote;
  }
  public Point centre( ) {
     x = coinNordOuest.abscisse( ) + cote/2;
     y = coinNordOuest.ordonnee( ) - cote/2;
    return new Point(x, y);
  }
```

Héritage: Les classes abstraites (7)

- **Exemple -Suite:**

```
public double diametre( ) {
   return cote*Math.sqrt(2.0);
 }
 public double surface( ) {
   return cote*cote;
 }
 public String toString( ) {
   return new String("{coinNordOuest = " + coinNordOuest
     + ", cote = " + cote + "}");
 }
} // Fin de la classe Carre
```

Héritage: Les classes abstraites (8)

- **Exemple -Suite:**

```
class TestCircle {
  public static void main(String[ ] args) {
    Circle circle = new Circle(new Point(3.0,1.0),2.0);
    System.out.println("Le circle est " + circle);
    System.out.println("Son centre est " + circle.centre( ));
    System.out.println("Son diamètre est " + circle.diametre( ));
    System.out.println("Sa surface est " + circle.surface( ));
  }
}
```

Héritage: Les classes abstraites (9)

- **Exemple -Suite:**

Sortie du programme TestCircle.java:

```
Le circle est { centre = (3.0, 1.0), rayon = 2.0}
Son centre est (3.0, 1.0)
Son diamètre est 4.0
Sa surface est 12.566370614359172
```

Héritage: Les classes abstraites (10)

- **Exemple -Suite:**

```
class TestCarre {
  public static void main(String[ ] args) {
    Carre carre = new Carre(new Point(1.0,5.0),3.0);
    System.out.println("Le carré est " + carre);
    System.out.println("Son centre est " + carre.centre( ));
    System.out.println("Son diamètre est " + carre.diametre( ));
    System.out.println("Sa surface est " + carre.surface( ));
  }
}
```

Héritage: Les classes abstraites (11)

- **Exemple -Fin:**

Sortie du programme TestCarre.java:

```
Le carré est {coinNordOuest = (1.0, 5.0), cote = 3.0}
Son centre est (2.0, 4.0)
Son diamètre est 4.242640687119286
Sa surface est 9.0
```

Héritage: Les classes abstraites (12)

- abstract versus final:
  - Une classe déclarée comme « **final** » ne peut pas être étendue.
  - Une méthode déclarée comme « **final** » dans une classe ne peut pas être remplacée dans les sous-classe de sa classe.
  - Au contraire, une méthode déclarée comme « **abstract** » est destinée à être remplacée dans chaque sous-classe de sa classe.

Héritage: Le polymorphisme (1)

- Soient ClasseX une classe et ClasseY une sous-classe de ClasseX.
- Une instance y de ClasseY est globalement identique à une instance x de ClasseX, mais elle a des données et des fonctionnalités supplémentaires. Cependant cette règle a des limites:
  - Si les deux classes déclarent une méthode g( ) avec la même signature, l'instruction y.g( ) appelle la méthode déclarée dans ClasseY, et non celle déclarée dans ClasseX.
  - Dans ce cas, on dit que la méthode g( ) de ClasseY **remplace** celle de ClasseX.

Héritage: Le polymorphisme (2)

- Règles de **remplacement de méthodes**:
  a) La méthode de remplacement doit avoir le même entête (même nom et même signature) que la méthode remplacée.
  b) La méthode de remplacement doit avoir un accès aussi important que la méthode remplacée. Par conséquent, une méthode « public » peut uniquement être remplacée par une autre méthode « public ».
- Le **remplacement des champs** est similaire au remplacement des méthodes;
  – Les déclarations sont identiques, mais elles sont définies dans des classes différentes (la superclasse et la sous-classe).

Héritage: Le polymorphisme (3)

- Le **polymorphisme** consiste au fait qu'une instance y d'une sous-classe ClasseY peut appeler une méthode h( ) de la superclasse ClasseX. Deux cas se présentent:
  a) La méthode h( ) n'est pas redéfinie (n'est pas remplacée) dans ClasseY. Dans ce cas, l'appel peut se faire de façon normale: **y.h( );**
  b) La méthode h( ) est redéfinie (remplacée) dans ClasseY. Dans ce cas, on utilise le mot-clé **super** pour appeler la méthode h( ) de la superclasse ClasseX: **super.h( );**

Héritage: Le polymorphisme (4)

- **Exemple:**
  - L'exemple suivant définit une classe Etudiant qui est une extension de la classe Personne que nous avons déjà vue auparavant.

```
class Etudiant extends Personne {
  protected double moyenne;   // Moyenne des notes

  Etudiant(NomComplet nom, char sexe, double moyenne) {
    super(nom, sexe);          // appelle le constructeur de la superclasse Personne
    this.moyenne = moyenne;
  }
...
```

Héritage: Le polymorphisme (5)

- **Exemple -Suite:**

```
double moyenne( ) {
   return moyenne;
 }
 public String toString( ) {
   String s;
   s = new String(super.toString( ));      // appelle toString( ) de la classe Personne
   s += "\n\tMoyenne:     " + moyenne;
   return s;
 }
} // Fin de la classe Etudiant
```

Héritage: Le polymorphisme (6)

- **Exemple -Fin:**

```
class TestEtudiant {
  public static void main(String[ ] args) {
    NomComplet nomEtudiante = new NomComplet("Ann", "Baker");
    Etudiant etudiante = new Etudiant(nomEtudiante, 'F', 13.5);
    System.out.println("Etudiante: " + etudiante);
  }
}
```

Héritage: Le polymorphisme (7)

- **Sortie du programme TestEtudiant.java:**

```
Etudiante: Ann Baker (sexe: F)
Moyenne:    13.5
```

Interfaces, paquetages et encapsulation

Interfaces (1)

- Une **interface** est une classe dont la définition ne peut comporter que:
  a) Des **constantes publiques** (champs déclarés comme « public final »).
  b) Des **entêtes de méthodes** (déclarations de méthodes sans corps).
- Une interface possède également les propriétés suivantes:
  - N'a pas de constructeurs.
  - Ne peut pas être instanciée.
  - Peut être implémentée par plusieurs classes.
  - Ne peut pas implémenter une autre interface.
  - Ne peut pas étendre une classe.
  - Peut étendre plusieurs autres interfaces.

Interfaces (2)

- Syntaxe de déclaration:

  interface Nom_Interface {

  // **Déclarations de constantes**

  // **Entêtes de méthodes**

  }

- Une interface a pour objectif de spécifier un ensemble de directives destinées aux classes d'implémentation.

Interfaces (3)

- Syntaxe de définition d'une classe d'implémentation:

```
class Nom_Classe implements interface Nom_Interface {
// Déclarations de champs
// Définitions de constructeurs
// Définitions des méthodes de l'interface Nom_Interface
// Définitions de méthodes supplémentaires
}
```

Interfaces (4)

- En Java, un **type** est soit une **interface**, soit une **classe**, soit un **tableau**, soit un **type de base**:
  - Les variables (champs ou variables locales) ou les paramètres peuvent être déclarés avec ces quatre types.
- **Remarque:**
  - Lorsqu'une classe implémente une interface, la classe est alors considérée comme **sous-type** de l'interface qui est, à son tour, qualifiée comme **supertype** de la classe.
  - De même, lorsqu'une classe (resp. une interface) étend une autre classe (resp. une autre interface), la première est alors considérée comme **sous-type** de la seconde qui est, à son tour, qualifiée comme **supertype** de la première.

Interfaces: L'interface Comparable (1)

- L'interface **Comparable** est définie dans les bibliothèques standard de Java (API), plus précisément dans le paquetage java.lang.
- L'interface Comparable comporte la déclaration d'une seule méthode **compareTo( )** décrivant la fonctionnalité de comparaison d'objets en vue de les trier.

Interfaces: L'interface Comparable (2)

- Définition de l'interface Comparable:

```
public interface Comparable {
public int compareTo(Object object);
}
```

Interfaces: L'interface Comparable (3)

- Contrat de l'interface Comparable:

Valeur renvoyée	**Interprétation**
x.compareTo(y) < 0	x < y
x.compareTo(y) = = 0	x **est égal à** y
x.compareTo(y) > 0	x > y

Interfaces: L'interface Comparable (4)

- La classe **String** fait partie des classes qui implémentent l'interface Comparable:
  - Comparaison lexicographique des chaînes en vue de leur organisation par ordre alphabétique.
- **Exemple 1:**
  - Le programme **CompareChaines.java** illustre la comparaison de chaînes (voir page suivante).

Interfaces: L'interface Comparable (5)

- **Exemple 2 (CompareChaines.java):**

```
public class CompareChaines {
  public static void main(String[ ] args) {
    String s = "COMPARER";
    System.out.println("s: " + s);
    System.out.println("s.compareTo(s): " + s.compareTo(s));
    System.out.println("s.compareTo(\"COMPARE\"): " + s.compareTo("COMPARE"));
    System.out.println("s.compareTo(\"COMPTER\"): " + s.compareTo("COMPTER"));
  }
}
```

Interfaces: L'interface Comparable (6)

- **Sortie du programme CompareChaines.java:**

```
s: COMPARER
s.compareTo(s): 0
s.compareTo("COMPARE"): 1
s.compareTo("COMPTER"): -19
```

Interfaces: L'interface Comparable (7)

- **Exemple 2:**
  - Ce deuxième exemple indique comment une classe définie par l'utilisateur peut implémenter l'interface Comparable.
  - La classe **ComparePoints**, définie dans cet exemple, étend la classe Point et implémente l'interface Comparable.
  - Etant donnés deux points P1 et P2, la classe ComparePoints spécifie le corps de la méthode compareTo( ) de la façon suivante:
    - Si P1.x < P2.x alors P1 < P2.
    - Si P1.x = = P2.x et P1.y < P2.y alors P1 < P2.
    - Si P1.x = = P2.x et P1.y = = P2.y alors P1 est égal à P2.
    - P1 > P2 dans les autres cas.

Interfaces: L'interface Comparable (8)

- **Exemple 2 –Suite (ComparePoints.java):**

```
public class ComparePoints extends Point implements Comparable {
   public ComparePoints(int x, int y) {
   super(x, y);
  }
   public int compareTo(Object object) {
   if (this == object) return 0;  // Même objet
   if (!(object instanceof ComparePoints))
     throw new IllegalArgumentException( );
   ComparePoints cetObjet = (ComparePoints)object;
   if (this.x < cetObjet.x) return -1;
   if (this.x == cetObjet.x)
      if (this.y < cetObjet.y) return -1;
      else if (this.y == cetObjet.y) return 0;
   return 1;
  }
}
```

Interfaces: L'interface Comparable (9)

- **Exemple 2 –Fin (TestComparePoints.java):**

```
public class TestComparePoints {
  public static void main(String[ ] args) {
    Comparable p1 = new ComparePoints(2.0, 4.0);
    Comparable p2 = new ComparePoints(2.0, -1.0);
    Comparable p3 = new ComparePoints(3.0, 1.0);
    System.out.println("p1: " + p1);
    System.out.println("p2: " + p2);
    System.out.println("p3: " + p3);
    System.out.println("p1.compareTo(p1): " + p1.compareTo(p1));
    System.out.println("p1.compareTo(p2): " + p1.compareTo(p2));
    System.out.println("p1.compareTo(p3): " + p1.compareTo(p3));
    System.out.println("p2.compareTo(p3): " + p2.compareTo(p3));
  }
}
```

Interfaces: L'interface Comparable (10)

- **Sortie du programme TestComparePoints.java:**

```
p1: (2.0, 4.0)
p2: (2.0, -1.0)
p3: (3.0, 1.0)
p1.compareTo(p1): 0
p1.compareTo(p2): 1
p1.compareTo(p3): -1
p2.compareTo(p3): -1
```

Interfaces: Types et Polymorphisme (1)

- Les sous-types prennent en charge le **polymorphisme des objets**:
  - Si l'objet x est de type T2 et T2 est un sous-type de T1, alors x est également de type T1. Ainsi, l'objet x est à la fois de type T2 et T1. Cet objet a donc plusieurs formes.
- **Remarque:**
  - Si T2 est un sous-type (resp. supertype) de T1, alors T2 [ ] est un sous-type (resp. supertype) de T1 [ ].

Interfaces: Types et Polymorphisme (2)

- **Exemple 1:**
  - Le programme **TestComparePoints.java** de l'exemple précédent illustre le polymorphisme des objets:
    - Les objets p1, p2 et p3 sont déclarés avec le type Comparable. Mais, ils sont instanciés comme des objets de la classe ComparePoints qui est un sous-type de l'interface Comparable.

Interfaces: Types et Polymorphisme (3)

- **Exemple 2:**
  - Le programme suivant (**Polymorphe.java**) illustre le cas d'un polymorphisme avec des variables locales et des paramètres de méthodes (voir page suivante).

Interfaces: Types et Polymorphisme (4)

- **Exemple 2 (Polymorphe.java) -Suite:**

```
public class Polymorphe {
   public Polymorphe( ) {
   Comparable[ ] s = { "TUNISIE", "MAROC", "EGYPTE", "LIBAN" };
   print(s);
   sort(s);
   print(s);
   s[0] = new ComparePoints(3.0, 5.0);
   s[1] = new ComparePoints(2.0, 4.0);
   s[2] = new ComparePoints(2.0, -1.0);
   s[3] = new ComparePoints(3.0, 1.0);
   print(s);
   sort(s);
   print(s);
  }
…
```

Interfaces: Types et Polymorphisme (5)

- **Exemple 2 (Polymorphe.java) –Suite:**

```
public static void print(Object[ ] a) {
    if (a == null || a.length == 0) return;
    System.out.print("{ " + a[0]);
    for (int i=1; i<a.length; i++)
      System.out.print(", " + a[i]);
    System.out.println(" }");
  }
…
```

Interfaces: Types et Polymorphisme (6)

- **Exemple 2 (Polymorphe.java) –Fin:**

```
void sort(Comparable[ ] a) {
    for (int i = 1; i < a.length; i++) {
      Comparable c = a[i];
      int j = i;
      for (j = i; j > 0 && a[j-1].compareTo(c) > 0; j--)
        a[j] = a[j-1];
      a[j] = c;
    }
  }

  public static void main(String[ ] args) {
    new Polymorphe( );
  }
}
```

Interfaces: Types et Polymorphisme (7)

- **Sortie du programme Polymorphe.java:**

```
{ TUNISIE, MAROC, EGYPTE, LIBAN }
{ EGYPTE, LIBAN, MAROC, TUNISIE }
{ (3.0, 5.0), (2.0, 4.0), (2.0, -1.0), (3.0, 1.0) }
{ (2.0, -1.0), (2.0, 4.0), (3.0, 1.0), (3.0, 5.0) }
```

Interfaces versus classes abstraites (1)

- Comparaison entre interfaces & classes abstraites:

Interface	**Classe abstraite**
Ne peut pas être instanciée	Ne peut pas être instanciée
Ne peut pas implémenter des méthodes	Peut implémenter des méthodes
Ne peut pas inclure des constructeurs	Peut inclure des constructeurs

Interfaces versus classes abstraites (2)

- Utilisation d'une interface:
  - Toute implémentation du contrat est remise à plus tard.
  - Trop restrictive dans certains cas.
- Utilisation d'une classe abstraite:
  - Possibilité d'une implémentation partielle du code.
  - Tout le code commun entre les sous-classes concrètes peut être implémenté une seule fois dans la superclasse abstraite.

Paquetages (1)

- Un **paquetage** (en anglais, **package**) est un dossier regroupant un ensemble de classes qui peuvent avoir, éventuellement, des liens entre elles.
- Pour ajouter une classe nommée **MaClasse** à un paquetage nommé **monpaquetage**:
  a) Créer d'abord un dossier nommé monpaquetage dans le répértoire courant.
  b) Ajouter ensuite l'instruction suivante au début du fichier contenant la définition de MaClasse:

  package monpaquetage;

Paquetages (2)

- Si la classe MaClasse est utilisée par une autre classe nommée UneClasse appartenant à un autre paquetage ou à un autre dossier, l'instruction suivante rend MaClasse visible par UneClasse:

  import monpaquetage.MaClasse;

- **Remarque:**
  - L'instruction ci-dessus doit être ajoutée au début du fichier contenant la définition de la classe UneClasse.
  - Cette instruction suppose que le dossier monpaquetage et le dossier contenant UneClasse se trouvent tous les deux dans le répertoire courant.

Paquetages (3)

- Pour importer toutes les classes appartenant à un paquetage donné:

  import Nom_Paquetage.* ;

- Les paquetages peuvent être organisés de façon hiérarchique:
  - Un paquetage Paq0 peut contenir un autre paquetage (sous-paquetage) Paq1 qui peut, à son tour, contenir un autre paquetage (sous-paquetage) Paq2 et ainsi de suite.
  - Dans ce cas, il faut respecter la hiérarchie des paquetages pour pouvoir les utiliser.
  - La hiérarchie doit être également respectée lorsque les paquetages sont contenus dans des dossiers.

Paquetages (4)

- **Illustration:**
  - Pour importer toutes les classes appartenant au paquetage Paq1:

    import Paq0.Paq1.* ;
  - Pour importer une classe nommée CetteClasse appartenant au paquetage Paq2:

    import Paq0.Paq1.Paq2.CetteClasse ;

Paquetages (5)

- Paquetages accessibles par défaut:
  - Le « **paquetage par défaut** » qui contient toutes les classes appartenant au répertoire courant et pour lesquelles aucun paquetage n'a été spécifié explicitement.
  - Le paquetage **java.lang** qui contient un certain nombre de classes d'usage général comme les classes System et Math.

Encapsulation (1)

- L'**encapsulation** est la faculté de distinguer les services offerts (interface) des détails d'implémentation de ces services (classes d'implémentation).
- Motivations:
  - Préservation de la sécurité des données:
    - Les données privées ne peuvent être lues ou modifiés que par des accesseurs ou mutateurs rendus publiques.
  - Préservation de l'intégrité des données:
    - Mécanismes de vérification et de validation des valeurs attribuées aux variables susceptibles d'être modifiées.

Encapsulation (2)

- On utilise les modificateurs d'accès pour préciser le niveau d'accessibilité (ou de visibilité) d'une variable ou d'une méthode.
- En Java, il y a quatre niveaux de visibilité:
  - **public**.
  - **private**.
  - **protected**.
  - « **par défaut** »: aucun modificateur d'accès n'est spécifié.

Encapsulation (3)

- Règles de visibilité:

Modificateur d'accès	Classe	Paquetage	Sous-classes	Toutes les classes
private	★			
protected	★	★	★	
public	★	★	★	★
« par défaut »	★	★		

Collections

Collections (1)

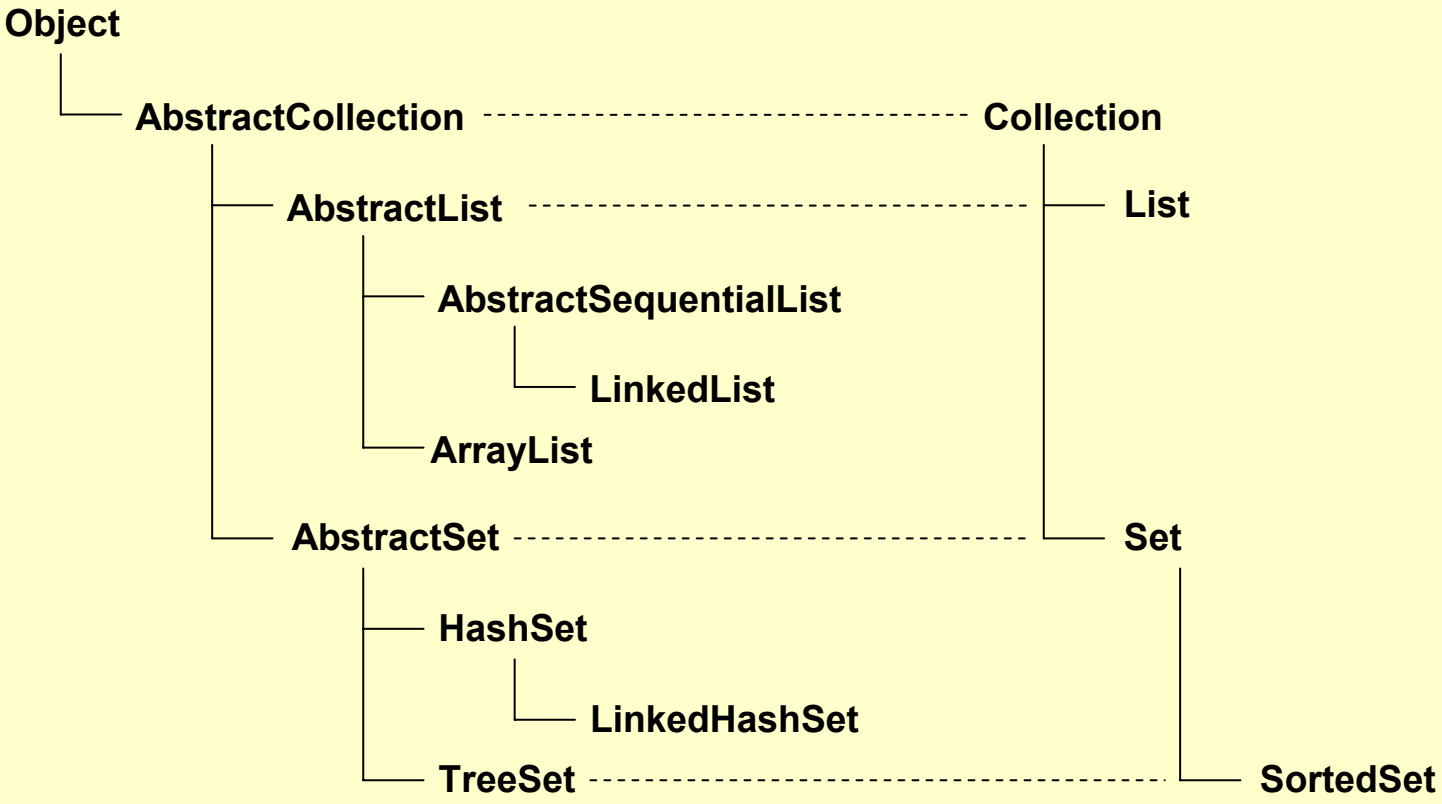
- Une **collection** est un objet qui contient d'autres objets.
- Les objets contenus dans une collection sont qualifiés d'**éléments de collection**.
- Le paquetage java.util définit trois types génériques de collection:
  - **List**: Séquence d'éléments.
  - **Set**: Collection non structurée d'éléments distincts.
  - **Map**: Collection non structurée de paires d'éléments (clé, valeur) avec des clés uniques.

Collections (2)

- Les deux premiers types de collection (**List** et **Set**) sont les plus utilisés dans la pratique.
- Ces deux types de collection sont définis comme des interfaces qui sont étendues et implémentées dans le « framework » de collections Java **JCF** (Java Collections Framework).

Collections (3)

- Partie du JCF qui concerne les types List et Set:



Collections (4)

- JCF implémente chaque interface avec plusieurs structures de données différentes:
  - Les tableaux indexés sont utilisés par les classes d'implémentation ArrayList et HashSet.
  - Les listes linéaires liées sont utilisées par la classe LinkedList.
  - Les arbres binaires liés sont utilisés par la classe TreeSet.
  - Les tableaux hybrides (indexés et liés) sont utilisés par la classe LinkedHashSet.

Collections: L'interface Collection (1)

- L'interface **Collection** est le **supertype** des deux types génériques List et Set.
- Elle spécifie toutes les méthodes communes à toutes les collections génériques:
  - Accesseurs: contains( ), containsAll( ), isEmpty( ), size( ), toArray( ), toArray(Object [ ]).
  - Mutateurs: add( ), addAll( ), clear( ), remove( ), removeAll( ), retainAll( ).
  - Remplacements de méthodes de la classe Object: equals( ), hashCode( ).
  - Méthode permettant de parcourir une collection: iterator( ).

Collections: L'interface Collection (2)

- Définition complète:

```
public interface Collection {
  public boolean add(Object object);
  public boolean addAll(Collection collection);
  public void clear( );
  public boolean contains(Object object);
  public boolean containsAll(Collection collection);
  public boolean equals(Object object);
  public int hashCode( );
...
```

Collections: L'interface Collection (3)

- Définition complète -Suite:

```
  public boolean isEmpty( );
  public Iterator iterator( );
  public boolean remove(Object object);
  public boolean removeAll(Collection collection);
  public boolean retainAll(Collection collection);
  public int size( );
  public Object[ ] toArray( );
  public Object[ ] toArray(Object[ ] objects);
} // Fin de définition de l'interface Collection
```

Collections: Les listes (1)

- Une **liste** est un conteneur séquentiel capable d'insérer et de supprimer des éléments localement de façon constante.
- En Java, les méthodes permettant de gérer les listes sont déclarées dans l'interface **List** qui est une sous-interface de l'interface Collection.
- Ces méthodes sont implémentées dans la classe **AbstractList** et ses extensions (**AbstractSequentialList**, **LinkedList** et **ArrayList**).

Collections: Les listes (2)

- **Exemple:**

```
import java.util.*;
public class TestListe {
  public static void main(String[ ] args) {
    Collection liste = new ArrayList( );
    liste.add("MAROC");
    liste.add("ALGERIE");
    liste.add("TUNISIE");
    liste.add("MORITANIE");
    System.out.println("Liste: " + liste);
    System.out.println("liste.contains(\"TUNISIE\"): " + liste.contains("TUNISIE"));
…
```

Collections: Les listes (3)

- **Exemple -Suite:**

```
    System.out.println("liste.contains(\"LYBIE\"): " + liste.contains("LYBIE"));
    Object[ ] a = liste.toArray( );
    liste.remove("TUNISIE");
    System.out.println("Liste: " + liste);
    System.out.println("liste.contains(\"TUNISIE\"): " + liste.contains("TUNISIE"));
    System.out.println("liste.size( ): " + liste.size( ));
    System.out.println("a[2]: " + a[2]);
  }
}
```

Collections: Les listes (4)

- **Sortie du programme TestListe.java:**

```
Liste: [MAROC, ALGERIE, TUNISIE, MORITANIE]
liste.contains("TUNISIE"): true
liste.contains("LYBIE"): false
Liste: [MAROC, ALGERIE, MORITANIE]
liste.contains("TUNISIE"): false
liste.size( ): 3
a[2]: TUNISIE
```

Collections: Les tableaux et les listes (1)

- La méthode **asList( )** de la classe **Arrays** (paquetage java.util) permet de générer une liste (objet de type List) qui contient les éléments du tableau qui lui est passé en argument.

Collections: Les tableaux et les listes (2)

- **Exemple:**

```
import java.util.*;
public class TestAsList {
  public static void main(String[ ] args) {
    String[ ] t = {"Salut","Bonjour","Bonsoir"};
    Collection c = Arrays.asList(t);
    print(t);
    System.out.println(c);
}
…
```

Collections: Les tableaux et les listes (3)

- **Exemple -Suite:**

```
  public static void print(String[ ] a) {
    for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println( );
  }
} // Fin de définition de la classe TestAsList
```

Collections: Les tableaux et les listes (4)

- **Sortie du programme TestAsList:**

```
Salut Bonjour Bonsoir
[Salut, Bonjour, Bonsoir]
```

Collections: Les ensembles (1)

- Un **ensemble** est une collection d'éléments distincts.
- En Java, les méthodes permettant de gérer les ensembles sont déclarées dans l'interface **Set** qui est une sous-interface de l'interface Collection.
- Ces méthodes sont implémentées dans la classe **AbstractSet** et ses extensions (**HashSet**, **LinkedHashSet** et **TreeSet**).

Collections: Les ensembles (2)

- L'interface Collection spécifie quatre méthodes destinées à l'implémentation des opérations classiques de la théorie des ensembles:
  a) x.addAll(y) ⇨ x = x ∪ y.
  b) x.removeAll(y) ⇨ x = x - y.
  c) x.retainAll(y) ⇨ x = x ∩ y.
  d) x.containsAll(y) ⇨ renvoie « true » si et seulement si y ⊆ x.

Collections: Les ensembles (3)

- **Exemple:**

```
public class TestEnsemble {
  public static void main(String[ ] args) {
    Set A = new HashSet( );
    A.add("SUISSE");
    A.add("CANADA");
    A.add("FRANCE");
    A.add("ITALIE");
    System.out.println("A: " + A);
    Set B = new HashSet(A);
    System.out.println("B: " + B);
    Set C = new HashSet( );
    C.add("SUISSE");
    C.add("JAPAN");
    C.add("CHINE");
    C.add("AUSTRALIE");
…
```

Collections: Les ensembles (4)

- **Exemple -Suite:**

```
    System.out.println("C: " + C);
    System.out.println("B.containsAll(C): " + B.containsAll(C));
    System.out.println("B.addAll(C): " + B.addAll(C));
    System.out.println("B: " + B);
    System.out.println("B.containsAll(C): " + B.containsAll(C));
    System.out.println("B.removeAll(C): " + B.removeAll(C));
    System.out.println("B: " + B);
    System.out.println("A.retainAll(C): " + A.retainAll(C));
    System.out.println("A: " + A);
    System.out.println("C: " + C);
  }
} // Fin de définition de la classe TestEnsemble
```

Collections: Les ensembles (5)

- **Sortie du programme TestEnsemble.java:**

```
A: [ITALIE, CANADA, SUISSE, FRANCE]
B: [ITALIE, CANADA, SUISSE, FRANCE]
C: [CHINE, JAPAN, AUSTRALIE, SUISSE]
B.containsAll(C): false
B.addAll(C): true
B: [ITALIE, CHINE, JAPAN, CANADA, AUSTRALIE, SUISSE, FRANCE]
B.containsAll(C): true
B.removeAll(C): true
B: [ITALIE, CANADA, FRANCE]
A.retainAll(C): true
A: [SUISSE]
C: [CHINE, JAPAN, AUSTRALIE, SUISSE]
```

Collections : Les itérateurs (1)

- Un **itérateur** est un objet qui donne accès aux éléments d'un objet Collection.
- Les capacités réelles d'un itérateur sont spécifiées dans l'interface **Iterator** du paquetage java.util:

```
public interface Iterator {
  public boolean hasNext( );
  public Object next( );
  public void remove( );
}
```

Collections : Les itérateurs (2)

- Méthode **next( )**: renvoie l'élément courant de la collection, puis passe à l'élément suivant.
- Méthode **hasNext( )**: renvoie « true » si la méthode next( ) peut être appelée à nouveau sans atteindre la fin de la collection.
- Méthode **remove( )**: supprime le dernier élément auquel accède next( ).

Collections : Les itérateurs (3)

- **Remarque:**
  - La méthode remove( ) laisse parfois l'itérateur dans un état non défini susceptible d'être corrigé uniquement par un appel de la méthode next( ).
- Un itérateur est obtenu à l'aide d'un appel à la méthode iterator( ) de la collection.
- La méthode iterator( ) renvoie un itérateur qui est initialisé au début de la collection.

Collections : Les itérateurs (4)

- **Exemple:**

```
import java.util.*;
public class TestIterateur {
  public static void main(String[ ] args) {
    Set A = new TreeSet( );
    A.add("SUISSE");
    A.add("CANADA");
    A.add("FRANCE");
    A.add("ITALIE");
    A.add("JAPAN");
    A.add("CHINE");
    A.add("AUSTRALIE");
…
```

Collections : Les itérateurs (5)

- **Exemple -Suite:**

```
    System.out.println("A: " + A);
    Iterator it = A.iterator( );
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.remove( ): ");
    it.remove( );
    System.out.println("A: " + A);
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.remove( ): ");
    it.remove( );
…
```

Collections : Les itérateurs (6)

- **Exemple -Fin:**

```
    System.out.println("A: " + A);
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.remove( ): ");
    it.remove( );
    System.out.println("A: " + A);
    System.out.println("it.hasNext( ): " + it.hasNext( ));
    System.out.println("it.next( ): " + it.next( ));
    System.out.println("it.hasNext( ): " + it.hasNext( ));
  }
} Fin de définition de la classe TestIterateur
```

Collections : Les itérateurs (7)

- **Sortie du programme TestIterateur.java:**

```
A: [AUSTRALIE, CANADA, ITALIE, FRANCE, JAPAN, CHINE, SUISSE]
it.next( ): AUSTRALIE
it.next( ): CANADA
it.next( ): ITALIE
it.remove( ):
A: [AUSTRALIE, CANADA, FRANCE, JAPAN, CHINE, SUISSE]
it.next( ): FRANCE
it.remove( ):
A: [AUSTRALIE, CANADA, JAPAN, CHINE, SUISSE]
it.next( ): JAPAN
it.next( ): CHINE
it.remove( ):
A: [AUSTRALIE, CANADA, JAPAN, SUISSE]
it.hasNext( ): true
it.next( ): SUISSE
it.hasNext( ): false
```

Collections : Les itérateurs (8)

- L'itérateur qui parcourt une collection procède comme l'index qui fournit un accès direct à un tableau.
- Les collections peuvent être parcourues dans une boucle for:

```
for ( Iterator it = collection.iterator( ); it.hasNext( ); )
    System.out.print( it.next( ) + " " );
```

Exceptions

Exceptions (1)

- Une **exception** est une erreur qui a lieu au cours de l'exécution d'un programme.
- Une exception Java est une instance de la classe **Throwable** ou de l'une de ses sous-classes:
  - Instanciation explicite à l'aide d'une instruction **throw** dans le programme.
  - Instanciation implicite par l'environnement d'exécution lorsque celui-ci est incapable d'exécuter une instruction dans le programme.

Exceptions (2)

- Une exception lancée peut être attrapée par la clause **catch** d'une instruction **try**.
- L'utilisation d'une instruction **try** pour attraper une exception est qualifiée de gestion des exceptions.

Exceptions (3)

- Il existe deux types d'exceptions:
  1. Exceptions **non vérifiées**: celles qui peuvent être évitées en améliorant le code. Ce sont des instances des classes Error et RuntimeException ainsi que de leurs extensions.
  2. Exceptions **vérifiées**: celles qui sont vérifiées par le compilateur avant l'exécution du programme. Les instructions qui les lancent doivent être soit insérées dans une instruction try, soit déclarées dans l'entête de la méthode courante.

Exceptions non vérifiées (1)

- **Exemple 1 (exception non vérifiée implicite):**

```
class ExceptionImplicite {
    public static void main(String[ ] args) {
        int n = 4/0;
        System.out.println("Attention! Division par 0.");
   }
}
```

Exceptions non vérifiées (2)

- **Sortie du programme ExceptionImplicite.java:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionImplicite.main(ExceptionImplicite.java:14)
```

Exceptions non vérifiées (3)

- **Exemple 2 (exception non vérifiée explicite):**

```
class ExceptionExplicite {
  static double sqrt(double x) {
    if (x < 0) throw new IllegalArgumentException( );
    return Math.sqrt(x);
  }

  public static void main(String[ ] args) {
    System.out.println(sqrt(-25));
    System.out.println("Fin de la méthode main( ).");
  }
}
```

Exceptions non vérifiées (4)

- **Sortie du programme ExceptionExplicite.java:**

```
Exception in thread "main" java.lang.IllegalArgumentException
    at ExceptionExplicite.sqrt(ExceptionExplicite.java:13)
    at ExceptionExplicite.main(ExceptionExplicite.java:18)
```

Exceptions non vérifiées (5)

- Dans les exemples précédents, l'exception n'est pas attrapée, ce qui entraîne l'arrêt immédiat du programme après le lancement de l'exception.

Exceptions non vérifiées (6)

- Dans le cas général, on procède à attraper les exceptions pour éviter le plantage du programme. Pour attraper une exception non vérifiée, on utilise l'instruction **try** de la façon suivante:

```
try {
        // Séquence d'instructions
}
catch (Exception exception) {
        // Séquences d'instructions
}
```

Exceptions non vérifiées (7)

- La syntaxe (Exception exception) de la clause catch ne peut contenir qu'un seul paramètre dont le type est la classe **Exception** ou l'une de ses extensions.
- Dans l'exemple suivant (**Exemple 3**), on ajoute au programme de l'exemple précédent un bloc try pour encapsuler le code qui risque de lancer une exception.

Exceptions non vérifiées (8)

- **Exemple 3 (exception non vérifiée attrapée):**

```
class ENVAttrap {
  static double sqrt(double x) {
    if (x < 0) throw new IllegalArgumentException( );
    return Math.sqrt(x);
  }

  public static void main(String[ ] args) {
    try { // Bloc try
        System.out.println(sqrt(-25));
    }
    catch (Exception exception) { // Clause catch
    System.out.println("Exception: " + exception);
    }
    System.out.println("L'exception a été attrapée.");
    System.out.println("Fin de la méthode main( ).");
  }
}
```

Exceptions non vérifiées (9)

- **Sortie du programme ENVAttrap.java:**

```
Exception: java.lang.IllegalArgumentException
L'exception a été attrapée.
Fin de la méthode main( ).
```

Exceptions vérifiées (1)

- Les erreurs dues aux exceptions vérifiées ne peuvent pas généralement être évitées:
  - Ces erreurs sont généralement provoquées par un accès externe.
- Ces erreurs peuvent être gérées soit en les relançant, soit en les attrapant.
- Une méthode qui lance une exception vérifiée doit déclarer celle-ci à l'aide d'une clause **throws** dans son en-tête.

Exceptions vérifiées (2)

- **Exemple 1 (exception vérifiée relancée):**

```
import java.io.*;
public class EVRelance {
   public static void main (String[ ] args) throws IOException {
                    int n;
                    BufferedReader in;
                    in = new BufferedReader (new InputStreamReader(System.in));
                    System.out.println ("Entrer une valeur pour n:");
                    String input = in.readLine( );
                    n = Integer.parseInt(input);
                    System.out.println ("n = " +n);
                    System.out.println ("Exception relancée.");
  }
}
```

Exceptions vérifiées (3)

- **Sortie du programme EVRelance.java:**

```
Entrer une valeur pour n:
3
n = 3
Exception relancée.
```

Exceptions vérifiées (4)

- Syntaxe pour attraper une exception vérifiée:

```
try {
        // Séquence d'instructions
}
catch (ExceptionType1 exception1) {
        // Séquence d'instructions
}
catch (ExceptionType2 exception2) {
        // Séquence d'instructions
}
…
catch (ExceptionTypeN exceptionN) {
        // Séquence d'instructions
}
finally {
        // Séquence d'instructions
}
```

Exceptions vérifiées (5)

- **Exemple 2 (exception vérifiée attrapée):**

```
import java.io.*;
public class EVAttrap {
   public static void main (String[ ] args) throws IOException {
          int n;
          BufferedReader in;
          in = new BufferedReader (new InputStreamReader(System.in));
          System.out.println ("Entrer une valeur pour n:");
          String input = in.readLine( );
          try {
          n = Integer.parseInt(input);
          System.out.println ("n = " +n);
          }
          catch (NumberFormatException exception) {
          System.out.println ("Exception attrapée: la chaîne saisie n'est pas un nombre.");
          }
  }
}
```

Exceptions vérifiées (6)

- **Une exécution du programme EVAttrap.java:**

```
Entrer une valeur pour n:
77
n = 77
```

- **Une autre exécution du programme EVAttrap.java:**

```
Entrer une valeur pour n:
st&1
Exception attrapée: la chaîne saisie n'est pas un nombre.
```

Fichiers et flux

Introduction

- Les Entrées et les Sorties (**E/S**) sont gérées par les objets de **flux**:
  - **Flux d'entrée**: permet de fournir des données au programme.
  - **Flux de sortie**: permet de récupérer les résultats du programme.
- Types de flux les plus utilisés:
  - Flux de saisie à travers le clavier.
  - Flux d'affichage à travers l'écran.
  - Fichiers de données externes.
- Les classes d'E/S sont définies dans le paquetage java.io.

Fichiers texte (1)

- Les **fichiers texte** sont lus et écrits par des éditeurs de texte.
- En Java, ces fichiers sont transférés sous forme de caractères Unicode 16 bits.
- La classe **Reader** est la classe de base permettant de gérer les entrées des fichiers texte.
- La classe **Writer** est la classe de base permettant de gérer les sorties des fichiers texte.

Fichiers texte (2)

- La classe **BufferedReader** est la sous-classe la plus courante de la classe Reader:
  - La méthode **readLine( )** est la méthode la plus utilisée de la classe BufferedReader.
- La classe **PrintWriter** est la sous-classe la plus courante de la classe Writer:
  - La méthode **println( )** est la méthode la plus utilisée de la classe PrintWriter.

Fichiers texte (3)

- **Exemple:**
  - Le programme **ConversionMaj.java** effectue les traitements suivants:
    - Il lit un fichier texte nommé Alice.txt et situé sur le lecteur C.
    - Ensuite, il copie ce fichier dans un nouveau fichier texte nommé AliceMaj.txt et situé sur le lecteur C en remplaçant les lettres minuscules par des lettres majuscules.

Fichiers texte (4)

- **Exemple (ConversionMaj.java):**

```
import java.io.*;


public class ConversionMaj {
  public static void main(String[ ] args) {
    String fichierEntree = "C:\\Alice.txt";
    String fichierSortie = "C:\\AliceMaj.txt";
…
```

Fichiers texte (5)

- **Exemple (ConversionMaj.java) -Suite:**

```
try {
    System.out.println("Lecture du fichier:\t" + fichierEntree);
    FileReader fileReader = new FileReader(fichierEntree);
    BufferedReader in = new BufferedReader(fileReader);
    System.out.println("Ecriture dans le fichier:\t" + fichierSortie);
    FileWriter fileWriter = new FileWriter(fichierSortie);
    PrintWriter out = new PrintWriter(fileWriter);
    String ligne = null;
    int nbreLignes = 0;
…
```

Fichiers texte (6)

- **Exemple (ConversionMaj.java) -Fin:**

```
while ((ligne = in.readLine( )) != null) {
        out.println(ligne.toUpperCase( ));
        ++nbreLignes;
}
in.close( );
out.close( );
System.out.println("Copie de " + nbreLignes + " lignes.");
} // Fin du bloc try
catch(IOException exception) {
      System.out.println("Exception: " + exception);
}
}
} // Fin de la classe ConversionMaj
```

Fichiers binaires (1)

- Les **fichiers binaires** peuvent être lus et écrits par des programmes exécutés.
- En Java, ces fichiers sont transférés sous forme d'octets binaires de 8 bits.
- **Exemples:**
  - Fichiers image, vidéo, audio.
  - Fichiers exécutables.

Fichiers binaires (2)

- La classe **InputStream** est la classe de base permettant de gérer les entrées des fichiers binaires.
- La classe **OutputStream** est la classe de base permettant de gérer les sorties des fichiers binaires.

Fichiers binaires (3)

- La classe **ObjectInputStream** est la sous-classe la plus courante de la classe InputStream.
  - La méthode **read( )** est la méthode la plus utilisée de la classe ObjectInputStream.
- La classe **ObjectOutputStream** est la sous-classe la plus courante de la classe OutputStream.
  - La méthode **write( )** est la méthode la plus utilisée de la classe ObjectOutputStream.

Sérialisation des objets (1)

- La **sérialisation** fait référence à la représentation d'un objet dans un flux d'octets de façon à permettre:
  - Son transfert via un réseau.
  - Sa sauvegarde dans un fichier ou une base de données.
- L'interface **Serializable** du paquetage java.io permet de marquer les classes dont les objets peuvent être sérialisés.

Sérialisation des objets (2)

- Les méthodes d'E/S destinées à la sérialisation des objets sont définies dans les classes **Object** et **ObjectOutputStream**:
  - Si output est une instance de la classe ObjectOutputStream, alors **output.writeObject(x);** permet de sérialiser l'objet x dans ce flux.

Sérialisation des objets (3)

- **Exemple:**
  - Nous définissons d'abord la classe **Pays** qui représente un pays.
  - Le programme **SerialisationPays.java** crée un tableau de cinq objets Pays en chargeant leurs données depuis le fichier texte nommé pays.txt, puis il sérialise chacun de ces objets en un fichier binaire nommé pays.dat.

Sérialisation des objets (4)

- **Exemple (classe Pays):**

```
public class Pays implements java.io.Serializable {
  private String id, nom, capitale;
  private int superficie, population;

  public Pays(String id) {
    this.id = id;
  }
...
```

Sérialisation des objets (5)

- **Exemple (classe Pays) -Suite:**

```
public Pays(String id, String nom, String capitale, int superficie, int population) {
    this.id = id;
    this.nom = nom;
    this.capitale = capitale;
    this.superficie = superficie;
    this.population = population;
  }
…
```

Sérialisation des objets (6)

- **Exemple (classe Pays) -Suite:**

```
public int donnerSuperficie( ) {
    return superficie;
 }
  public String donnerCapitale( ) {
    return capitale;
 }
  public String donnerId( ) {
    return id;
 }
...
```

Sérialisation des objets (7)

- **Exemple (classe Pays) -Suite:**

```
public String donnerNom( ) {
   return nom;
}
 public int donnerPopulation( ) {
   return population;
}
 public void modifierSuperficie(int superficie) {
   this.superficie = superficie;
}
…
```

Sérialisation des objets (8)

- **Exemple (classe Pays) -Suite:**

```
public void modifierCapitale(String capitale) {
    this.capitale = capitale;
 }
  public void modifierNom(String nom) {
    this.nom = nom;
 }
  public void modifierPopulation(int population) {
    this.population = population;
 }
...
```

Sérialisation des objets (9)

- **Exemple (classe Pays) -Fin:**

```
public String toString( ) {
    return id + ":\tNom: " + nom
            + "\n\tCapitale: " + capitale
            + "\n\tSuperficie: " + superficie
            + "\n\tPopulation: " + population;
  }
} // Fin de la classe Pays
```

Sérialisation des objets (10)

- **Exemple (SerialisationPays.java):**

```
import java.io.*;
import java.util.*;
public class SerialisationPays {
  private final String CHEMIN = "C:\\";
  private final String SOURCE = "pays.txt";
  private final String DESTINATION = "pays.dat";
…
```

Sérialisation des objets (11)

- **Exemple (SerialisationPays.java) -Suite:**

```
public ecrirePays( ) {
    try {
      Pays[ ] pays = extraireDonnees( );
      ecrireDestination(pays);
    } catch(IOException exception) {
      System.out.println("Exception: " + exception);
    }
  }
...
```

Sérialisation des objets (12)

- **Exemple (SerialisationPays.java) -Suite:**

```
private Pays[ ] extraireDonnees( ) throws IOException {
   List liste = new ArrayList( );
   BufferedReader in = null;
   int n = 0;
   try {
     in = new BufferedReader(new FileReader(CHEMIN + SOURCE));
     for (String ligne=in.readLine( ); ligne!=null; ligne=in.readLine( )) {
       StringTokenizer tok = new StringTokenizer(ligne, "\t");
…
```

Sérialisation des objets (13)

- **Exemple (SerialisationPays.java) -Suite:**

```
Pays pays = new Pays(tok.nextToken( ));
        pays.modifierNom(tok.nextToken( ));
        pays.modifierCapitale(tok.nextToken( ));
        pays.modifierSuperficie(Integer.parseInt(tok.nextToken( )));
        pays.modifierPopulation(Integer.parseInt(tok.nextToken( )));
        liste.add(pays);
        ++n;
      }
...
```

Sérialisation des objets (14)

- **Exemple (SerialisationPays.java) -Suite:**

```
} catch(IOException exception) {
      System.out.println("Exception: " + exception);
    } finally {
      if (in != null) in.close( );
    }
    System.out.println("Lecture de " + n + " lignes de:\t" + CHEMIN + SOURCE );
    return (Pays[ ])liste.toArray(new Pays[0]);
  }
...
```

Sérialisation des objets (15)

- **Exemple (SerialisationPays.java) -Suite:**

```
private void ecrireDestination(Pays[ ] p) throws IOException {
   ObjectOutputStream out = null;
   int n = p.length;
   try {
    out = new ObjectOutputStream(new FileOutputStream(CHEMIN + DESTINATION));
    for (int i=0; i<n; i++) {
      out.writeObject(p[i]);
    }
…
```

Sérialisation des objets (16)

- **Exemple (SerialisationPays.java) -Fin:**

```
} finally {
      if (out != null) out.close( );
    }
    System.out.println("Ecriture de " + n + " objets dans:\t" + CHEMIN + DESTINATION );
  }
  public static void main(String[ ] args) {
    new ecrirePays( );
  }
} // Fin du programme SerialisationPays.java
```

Désérialisation des objets (1)

- La désérialisation permet à un objet sérialisé de se reconstituer pour reprendre sa forme initiale.
- Les méthodes d'E/S destinées à la désérialisation des objets sont définies dans les classes Object et ObjectInputStream:
  - Si input est une instance de la classe ObjectInputStream liée à un objet sérialisé, alors
    input.readObject(x);
    permet de désérialiser l'objet en x.

Désérialisation des objets (2)

- **Exemple:**
  - Le programme DeserialisationPays.java désérialise les cinq objets Pays du fichier binaire pays.dat de l'exemple précédent, puis il utilise la méthode toString( ) de la classe Pays afin de les afficher.

Désérialisation des objets (3)

- **Exemple (DeserialisationPays.java):**

```
import java.io.*;
public class lirePays {
  private final String CHEMIN = "C:\\";
  private final String DESTINATION = "pays.dat";

  public lirePays( ) throws IOException {
    System.out.println("Lecture de:\t" + CHEMIN + DESTINATION);
    ObjectInputStream in = null;
…
```

Désérialisation des objets (4)

- **Exemple (DeserialisationPays.java) -Suite:**

```
try {
      in = new ObjectInputStream(new FileInputStream(CHEMIN + DESTINATION));
      Object objet = null;
      while ((objet = in.readObject( )) != null) {
        System.out.println(objet);
      }
    } catch(EOFException exception) {
      System.out.println("Fin de fichier.");
…
```

Désérialisation des objets (5)

- **Exemple (DeserialisationPays.java) -Suite:**

```
} catch(IOException exception) {
      System.out.println("Exception: " + exception);
    } catch(ClassNotFoundException exception) {
      System.out.println("Exception: " + exception);
    } finally {
      if (in != null) in.close( );
    }
    System.out.println("Fin de lecture des objets de:\t" + CHEMIN + DESTINATION );
  }
…
```

Désérialisation des objets (6)

- **Exemple (DeserialisationPays.java) -Fin:**

```
public static void main(String[ ] args) {
    try {
      new lirePays( );
    } catch(IOException exception) {
      System.out.println("Exception: " + exception);
    }
  }
} // Fin du programme DeserialisationPays.java
```

Fichiers à accès aléatoire (1)

- Les flux d'E/S, vus précédemment, permettent seulement un accès séquentiel aux fichiers.
- Les éléments des fichiers à accès aléatoire (ou direct) peuvent être lus et/ou écrits directement.
- La classe **RandomAccessFile** permet d'obtenir un accès aléatoire en lecture et/ou en écriture.

Fichiers à accès aléatoire (2)

- La création d'un fichier à accès aléatoire s'effectue à l'aide de deux paramètres:
  - Le premier est de type String; il indique le nom et le chemin du fichier.
  - Le deuxième est également de type String; il indique si le fichier est ouvert en lecture seule ("r") ou en lecture et écriture ("rw").

Fichiers à accès aléatoire (3)

- **Exemple:**
  - Le programme AccesAleatoire.java crée d'abord un fichier texte simple nommé Test.txt dans le lecteur C et le remplit avec 250 caractères '#' (50 caractères sur chaque ligne).
  - Ensuite, il utilise un accès direct pour remplacer 14 de ces caractères au milieu de la ligne centrale par la chaîne " Bonjour Amis! ".

Fichiers à accès aléatoire (4)

- **Exemple (AccesAleatoire.java):**

```
import java.io.*;
public class AccesAleatoire {
  private final static String CHEMIN = "C:\\";
  private final static String FICHIER = CHEMIN + "Test.txt";
  private final static int Nbre_Car_Ligne = 50;
  private final static int Nbre_Total_Car = 250;
  private final static int DIESE = 35;
  private final static int Position_Ecriture = 120;
  private final static String s = " Bonjour Amis! ";
…
```

Fichiers à accès aléatoire (5)

- **Exemple (AccesAleatoire.java) -Suite:**

```
public static void main(String[ ] args) throws IOException {
   OutputStream out = new FileOutputStream(FICHIER);
   for (int i=0; i<Nbre_Total_Car; i++) {
     if (i%Nbre_Car_Ligne == 0) out.write((byte)'\n');
     else out.write((byte)DIESE);
   }
   out.close( );
…
```

Fichiers à accès aléatoire (6)

- **Exemple (AccesAleatoire.java) -Suite:**

```
System.out.println("Ecriture de " + Nbre_Total_Car + " '" + (char)DIESE +
      "' characters to " + CHEMIN+"Test.txt");
   RandomAccessFile raf = new RandomAccessFile(FICHIER, "rw");
   raf.seek(Position_Ecriture);
   raf.writeBytes(s);
   System.out.println("La longueur du fichier est " + raf.length( ));
   raf.seek(100);
   raf.close( );
…
```

Fichiers à accès aléatoire (7)

- **Exemple (AccesAleatoire.java) -Fin:**

```
System.out.println("Lecture de:\t"+FICHIER);
    FileReader fileReader = new FileReader(FICHIER);
    BufferedReader in = new BufferedReader(fileReader);
    String ligne = null;
    while ((ligne = in.readLine( )) != null) {
      System.out.println(ligne);
    }
    in.close( );
  }
} // Fin du programme AccesAleatoire.java
```

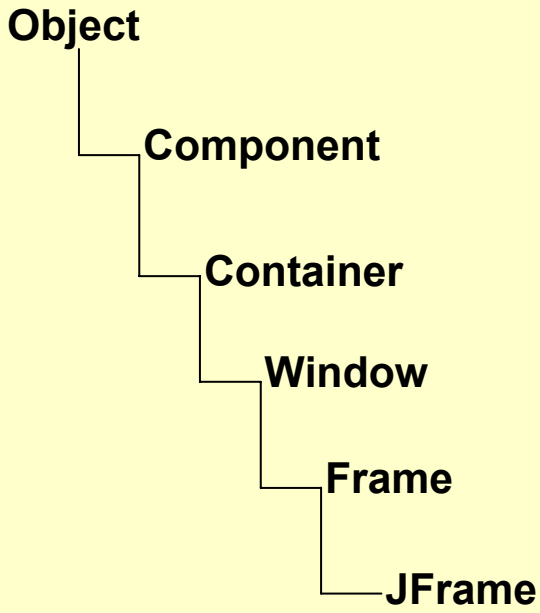
Interfaces graphiques

Introduction

- Java possède une bibliothèque de classes graphiques.
- La plupart de ces classes se trouvent dans l'un des deux paquetages javax.swing ou java.awt.

La classe JFrame (1)

- La classe JFrame se trouve dans le paquetage javax.swing. Elle permet de créer des graphiques (fenêtres, boutons,…).
- Cette classe hérite des fonctionnalités de ses classes mères dont la hiérarchie est la suivante:



La classe JFrame (2)

- **Exemple 1:**
  - Le code suivant indique comment créer un programme graphique simple à l'aide de la classe JFrame:
    - La classe MainFrame est définie comme extension de la classe JFrame; elle permet de définir un objet graphique sous forme d'une fenêtre avec un bouton de fermeture, un titre, une taille précise et un emplacement précis.
    - La classe TestMainFrame permet de tester la classe MainFrame.

La classe JFrame (3)

- **Exemple 1 –Suite:**

```
import javax.swing.*;
class MainFrame extends JFrame {
  MainFrame( ) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("Cadre principal");
    setSize(300, 200);
    setLocation(250, 150);
  }
}
```

La classe JFrame (4)

- **Exemple 1 –Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    JFrame fenetre = new MainFrame( );
    fenetre.show( );
  }
}
```

La classe JFrame (5)

- **Exemple 2:**
  - Le programme suivant modifie celui de l'exemple précédent en plaçant la fenêtre au centre de l'écran et en définissant sa taille à ¼ celle de l'écran.

La classe JFrame (6)

- **Exemple 2 -Suite:**

```
import java.awt.*;
import javax.swing.*;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Cadre principal");
    setSize(largeur, hauteur);
    setLocation(x, y);
  }
}
```

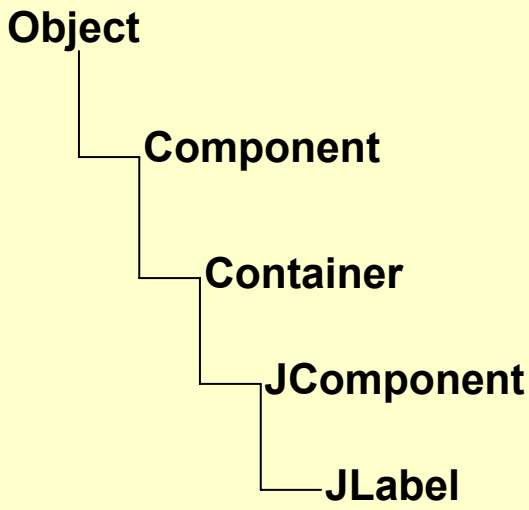
La classe JFrame (7)

- **Exemple 2 -Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    Dimension screenSize = Toolkit.getDefaultToolkit( ).getScreenSize( );
    int w = screenSize.width;
    int h = screenSize.height;
    JFrame fenetre = new MainFrame(w/2, h/2, w/4, h/4);
    fenetre.show( );
  }
}
```

La classe JLabel (1)

- La classe JLabel est également définie dans le paquetage javax.swing.
- Cette classe permet d'afficher un texte dans une fenêtre.
- La hiérarchie de la classe JLabel est la suivante:



La classe JLabel (2)

- **Exemple:**
  - Le programme suivant modifie celui de l'exemple 2 précédent en modifiant la taille de la fenêtre et en y ajoutant un message textuel.

La classe JLabel (3)

- **Exemple -Suite:**

```
import java.awt.*;
import javax.swing.*;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Cadre principal");
    setSize(largeur, hauteur);
    setLocation(x, y);
    JLabel texte = new JLabel("Bonjour tout le monde!");
    texte.setFont(new Font("Serif", Font.BOLD|Font.ITALIC, hauteur/2));
    getContentPane( ).add(texte);
  }
}
```

La classe JLabel (4)

- **Exemple –Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    Dimension screenSize = Toolkit.getDefaultToolkit( ).getScreenSize( );
    int w = screenSize.width;
    JFrame fenetre = new MainFrame(w/5, 100, 2*w/5, 0);
    fenetre.show( );
  }
}
```

Classification des graphiques (1)

- Types de graphiques:
  1) Les composants: Objets susceptibles d'être affichés à l'écran et d'interagir avec l'utilisateur. Par exemple, le bouton de fermeture est un composant.
  2) Les conteneurs: Composants qui contiennent d'autres composants. Par exemple, une fenêtre est un conteneur tandis qu'un bouton ne l'est pas.

Classification des graphiques (2)

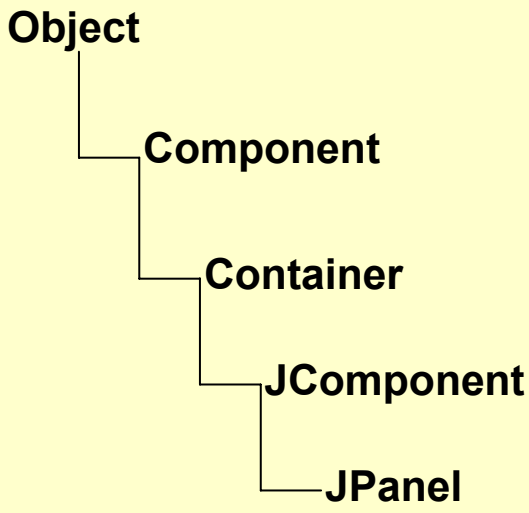
- Types de graphiques -Suite:
  3) Les fenêtres: Conteneurs ayant des propriétés spécifiques, notamment une boîte à outils facilitant la création des composants.
  4) Les cadres: Fenêtres composées d'une barre de titre, une barre de menus, une bordure, un curseur et une image d'icône. Les cadres sont des objets standard lors de la création des graphiques.

La classe JPanel (1)

- On peut ajouter un composant (ex. un objet JLabel) à une fenêtre graphique (ex. un cadre) de deux façons:
  1) Le composant peut être ajouté directement au panneau de contenu de la fenêtre.
  2) Le composant peut être ajouté à un conteneur intermédiaire qui est, lui-même, ajouté au panneau de contenu de la fenêtre.
- Dans le deuxième cas, la classe JPanel peut servir comme conteneur intermédiaire.

La classe JPanel (2)

- La classe JPanel est également définie dans le paquetage javax.swing.
- Voici la hiérarchie d'héritage de cette classe:



La classe JPanel (3)

- **Exemple:**
  - Le programme suivant modifie celui de l'exemple précédent en ajoutant un objet JPanel destiné à contenir l'objet JLabel qui sert à l'affichage d'un contenu textuel:
    - La classe MainPanel est définie comme extension de la classe JPanel. Cette classe demande au constructeur d'attribuer la couleur noir à l'arrière plan du panneau, puis de créer un objet JLabel permettant d'afficher un texte de couleur jaune. Ensuite, l'objet JLabel est ajouté à l'objet JPanel qui, à son tour, est ajouté à l'objet JFrame.

La classe JPanel (4)

- **Exemple -Suite:**

```
import java.awt.*;
import javax.swing.*;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Cadre principal");
    setSize(largeur, hauteur);
    setLocation(x, y);
    JPanel panneau = new MainPanel( );
    getContentPane( ).add(panneau);
  }
}
```

La classe JPanel (5)

- **Exemple -Suite:**

```
class MainPanel extends JPanel {
  MainPanel( ) {
    setBackground(Color.black);
    JLabel texte = new JLabel("Hello, World!");
    texte.setFont(new Font(null, Font.BOLD, 40));
    texte.setForeground(Color.yellow);
    add(texte);
  }
}
```

La classe JPanel (6)

- **Exemple -Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    Dimension screenSize = Toolkit.getDefaultToolkit( ).getScreenSize( );
    int w = screenSize.width;
    JFrame fenetre = new MainFrame(w/5, 100, 2*w/5, 0);
    fenetre.show( );
  }
}
```

La classe Color (1)

- La classe Color est définie dans le paquetage java.awt.
- Cette classe permet de gérer les couleurs des objets graphiques.
- Cette classe hérite directement de la classe Object.

La classe Color (2)

- La classe Component contient deux méthodes pour définir les couleurs d'arrière et d'avant plan des composants graphiques:
  - **setBackground(Color. nomCouleur);**
  - **setForeground(Color.nomCouleur);**
- La classe Color définit 13 champs (public static final) pour les couleurs:
  black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow.

La classe Color (3)

- **Exemple:**
  - Le programme suivant modifie celui de l'exemple précédent en affichant une série d'objets JPanel ayant, chacun, une couleur différente:
    - La classe ColoredPanel permet de construire un panneau coloré avec, éventuellement, une étiquette (un texte) qui s'affiche sur le panneau. Cette classe définit également un objet Border qui permet d'ajouter une bordure noire au panneau.

La classe Color (4)

- **Exemple -Suite:**

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.Border;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Couleurs");
    setSize(largeur, hauteur);
    setLocation(x, y);
…
```

La classe Color (5)

- **Exemple -Suite:**

```
    JPanel panneauPrincipal = new JPanel( );
    panneauPrincipal.setBackground(Color.white);
    panneauPrincipal.add(new ColoredPanel(Color.cyan, "blue ciel"));
    panneauPrincipal.add(new ColoredPanel(Color.magenta, "magenta"));
    panneauPrincipal.add(new ColoredPanel(Color.green, "vert"));
    panneauPrincipal.add(new ColoredPanel(Color.orange, "orange"));
    panneauPrincipal.add(new ColoredPanel(Color.gray, "gray"));
    panneauPrincipal.add(new ColoredPanel(Color.pink, "rose"));
    getContentPane( ).add(panneauPrincipal);
  }
} // Fin de la classe MainFrame
```

La classe Color (6)

- **Exemple -Suite:**

```
class ColoredPanel extends JPanel {
   Border ligneNoir = BorderFactory.createLineBorder(Color.black);

   ColoredPanel(Color couleur, String texte) {
   setBackground(couleur);
   setBorder(ligneNoir);
   add(new JLabel(texte));
}
```

La classe Color (7)

- **Exemple -Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    Dimension screenSize = Toolkit.getDefaultToolkit( ).getScreenSize( );
    int w = screenSize.width;
    JFrame fenetre = new MainFrame(w/5, 100, 2*w/5, 0);
    fenetre.show( );
  }
}
```

Les gestionnaires de présentation (1)

- La présentation fait référence à la disposition de plusieurs composants dans un conteneur.
- Les classes principales permettant de gérer la présentation sont BorderLayout, FlowLayout et GridLayout. Ces trois classes implémentent l'interface LayoutManager.
- La présentation d'un conteneur peut être définie en passant un objet LayoutManager à la méthode setLayout( ) du conteneur.

Les gestionnaires de présentation (2)

- Le gestionnaire FlowLayout organise les composants du conteneur de gauche à droite et de haut en bas. La classe FlowLayout est le gestionnaire par défaut des conteneurs JPanel.
- Le gestionnaire GridLayout organise les composants du conteneur de manière similaire à celle de FlowLayout. De plus, il spécifie le nombre de lignes et de colonnes à utiliser. Si la fenêtre qui contient la grille est redimensionnée, GridLayout redimensionne les formes des composants proportionnellement.

Les gestionnaires de présentation (3)

- **Exemple 1:**
  - Le programme suivant spécifie la présentation du panneau principal en utilisant le gestionnaire GridLayout:
    - Le panneau principal est créé dans la classe MainFrame et est présenté sous forme d'une grille de 4 lignes et 7 colonnes. Ensuite, 26 panneaux étiquetés sont ajoutés au panneau principal. Ces 26 panneaux et leurs étiquettes vont constituer les cellules de la grille.

Les gestionnaires de présentation (4)

- **Exemple 1 –Suite:**

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.Border;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Présentation en grille");
    setSize(largeur, hauteur);
    setLocation(x, y);
…
```

Les gestionnaires de présentation (5)

- **Exemple 1 –Suite:**

```
JPanel panneau = new JPanel( );
    panneau.setLayout(new GridLayout(4, 7));
    for (int i=0; i<26; i++) {
      panneau.add(new LabeledPanel("" + (char)('A' + i)));
    }
    getContentPane( ).add(panneau);
  }
}
```

Les gestionnaires de présentation (6)

- **Exemple 1 –Suite:**

```
class LabeledPanel extends JPanel {
    Border ligneNoir = BorderFactory.createLineBorder(Color.black);
    LabeledPanel(String texte) {
    setBackground(Color.white);
    setBorder(ligneNoir);
    add(new JLabel(texte));
  }
}
```

Les gestionnaires de présentation (7)

- **Exemple 1 –Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    JFrame fenetre = new MainFrame(300, 150, 400, 0);
    fenetre.show( );
  }
}
```

Les gestionnaires de présentation (8)

- **Exemple 2:**
  - Le programme suivant crée un panneau principal en utilisant le gestionnaire BorderLayout:
    - Le gestionnaire BorderLayout divise le panneau principal en 5 panneaux étiquetés. Les étiquettes sont centrées horizontalement dans chacune des 5 sections. De plus, les largeurs des sections « Ouest et Est » sont ajustées à la taille de leur contenu, tandis que celles des 3 autres sections sont étendues sur toute la largeur du panneau principal.

Les gestionnaires de présentation (9)

- **Exemple 2 –Suite:**

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.Border;
class MainFrame extends JFrame {
  MainFrame(int largeur, int hauteur, int x, int y) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle ("Présentation en sections");
    setSize(largeur, hauteur);
    setLocation(x, y);
…
```

Les gestionnaires de présentation (10)

- **Exemple 2 –Suite:**

```
    JPanel panneau = new JPanel( );
    panneau.setLayout(new BorderLayout( ));
    panneau.add(new LabeledPanel("Nord"), BorderLayout.NORTH);
    panneau.add(new LabeledPanel("Ouest"), BorderLayout.WEST);
    panneau.add(new LabeledPanel("Centre"), BorderLayout.CENTER);
    panneau.add(new LabeledPanel("Est"), BorderLayout.EAST);
    panneau.add(new LabeledPanel("Sud"), BorderLayout.SOUTH);
    getContentPane( ).add(panneau);
  }
}
```

Les gestionnaires de présentation (11)

- **Exemple 2 –Suite:**

```
class LabeledPanel extends JPanel {
    Border ligneNoir = BorderFactory.createLineBorder(Color.black);
    LabeledPanel(String texte) {
    setBackground(Color.white);
    setBorder(ligneNoir);
    add(new JLabel(texte));
  }
}
```

Les gestionnaires de présentation (12)

- **Exemple 2 –Fin:**

```
public class TestMainFrame {
  public static void main(String[ ] args) {
    JFrame fenetre = new MainFrame(200, 120, 400, 0);
    fenetre.show( );
  }
}
```