Faculté des Sciences et Techniques de Tanger
Département Génie Informatique

Cycle Ingénieur
Logiciel et Système Intelligent

PROGRAMMATION ORIENTÉE OBJET
JAVA

Rappel

LES CONCEPTS DE L'ORIENTE OBJET

Concepts objets

I- Introduction

a) Crise du logiciel
b) Qualité du logiciel
c) Modularité
d) Évolution de la programmation

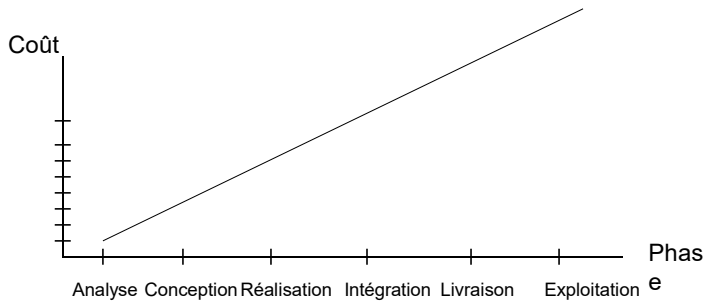
II- Concepts objets

a) Objets, message, attribut.
b) Abstraction
c) Encapsulation
d) Classe
e) Héritage/Association.
f) Polymorphisme

II- Conclusion

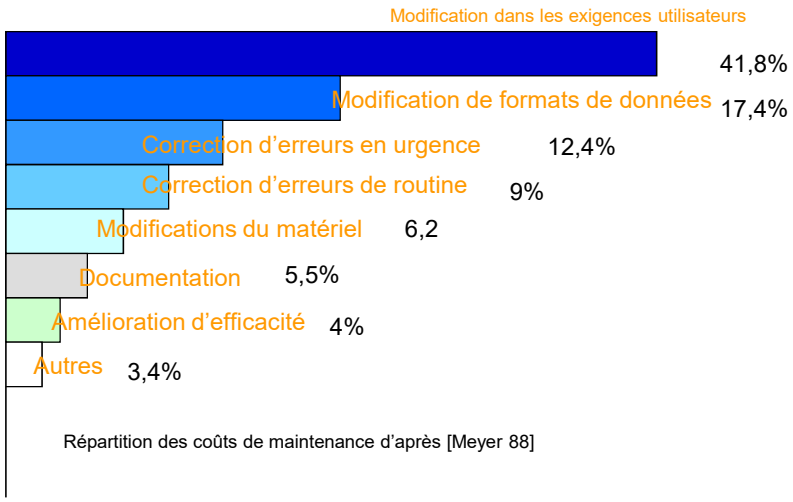
3

Crise du logiciel



4

Crise du logiciel (suite …)



Répartition des coûts de maintenance d'après [Meyer 88]

5

Défis des langages à objets

- Réduire les coûts de développement du logiciel.
- Réduire la maintenance en rééquilibrant la balance entre les coûts de développement et ceux de la maintenance.
- Faciliter la réutilisation de composants logiciels déjà réalisés.
- Accroître la qualité du logiciel

6

Qualité du logiciel

❖ Facteurs liés à l'utilisateur (client)

- Validité, Robustesse, facilité d'utilisation,
- Extensibilité, Compatibilité.

❖ Facteurs liés au concepteur

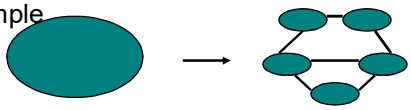
- lisibilité, modularité, réutilisabilité, vérifiabilité, portabilité.

7

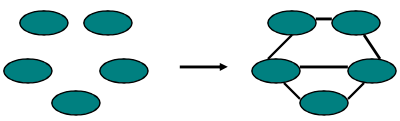
Modularité

➔ Critères de construction modulaire

❖ Décomposabilité modulaire : décomposition d'un logiciel complexe en un sous-système moins complexe organisé en une structure simple



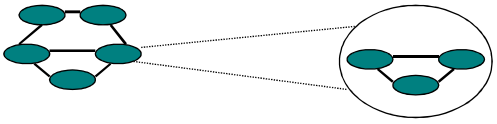
❖ Composabilité modulaire : Utilisation d'éléments existants pour construire de nouveaux logiciels.



8

Modularité (suite …)

❖ Compréhensibilité modulaire : Compréhension de chaque module pris isolément.



❖ Protection modulaire : Chaque module est responsable de ses dysfonctionnements et les traite localement.

9

Principes pour assurer la modularité

❖ Peu d'interfaces

❖ Interfaces réduites

❖ Interfaces explicites

❖ Masquage de l'information

❖ Ouverture

10

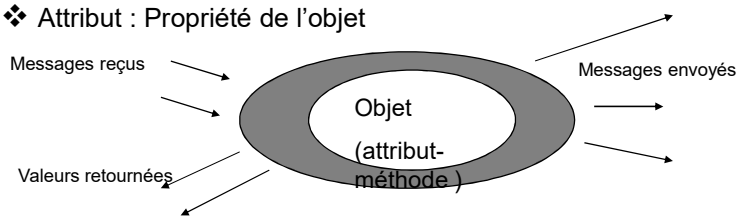
Objet, Message, Attribut

❖Objets : Module d'un concept du monde réel regroupant des structure de données ainsi qu'un comportement.

❖ Message : Requête d'une action que doit réaliser l'objet récepteur du message.

- Les messages sont mis en œuvre par les méthodes.
- Les messages définissent le comportement. Leur interprétation dépend du récepteur.
- Le comportement de l'objet se définit par des protocoles.

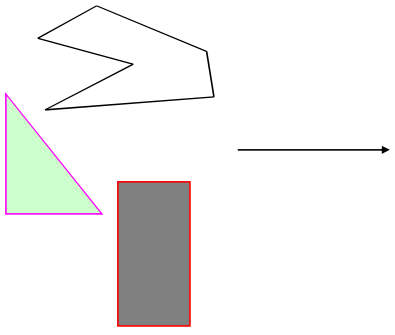
❖ Attribut : Propriété de l'objet



11

Abstraction

❖ Abstraction : analyse des concepts, définir les frontières conceptuelles, supprimer les détails en se concentrant sur la vue externe de l'objet



Classe polygone

Attributs

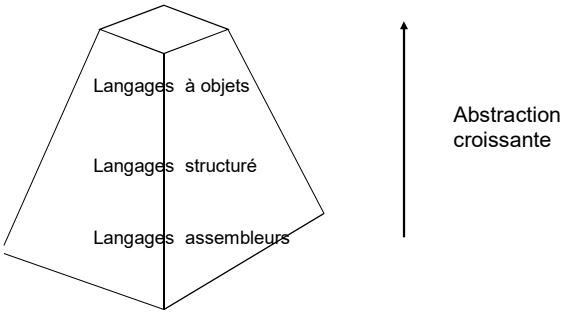
- arêtes
- couleur des bords
- couleur du fond

Méthodes

- dessiner
- effacer
- déplacer

12

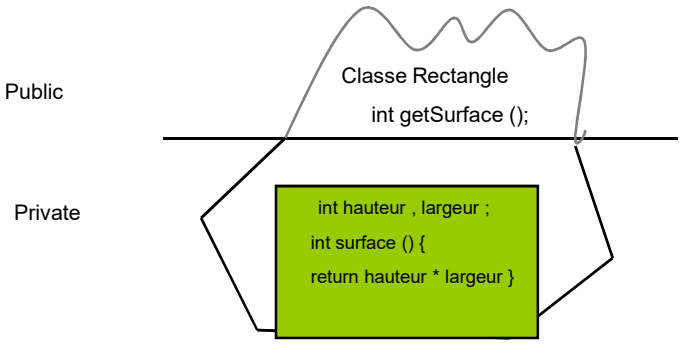
Abstraction des langages de programmation



13

Encapsulation

❖ Encapsulation : cacher la mise en œuvre (partie privée) de l'objet en offrant une interface (partie publique à ses clients).
Un client ne peut utiliser un objet qu'en respectant son interface.



14

Classe

❖ Classe : Spécification des attributs et des méthodes des objets ayant les même propriétés et comportements.

➢ une classe est une abstraction qui décrit les propriétés pertinentes pour l'application et ignore les autres.

⇨ Tout choix de classe est arbitraire

➢ une instance possède ses propres valeurs pour chaque attribut, mais partage les noms d'attributs et les méthodes avec les autres instances de la classe.

Classe Pièces_d_echec

Attributs

position

rôle

Méthodes

déplacer

15

Méthodes

➢ Exécutée par l'objet lorsqu'elle est appelée.
➢ Manipule les attributs de l'objet.
➢ Deux types de méthodes :
- ✓ Méthodes de classe : s'applique à toutes les instances de la classe.
- ✓ Méthodes d'instance : s'applique uniquement aux attributs de l'instance courante.

Attributs

➢ Décrit les caractéristiques et les états de l'objet.
➢ Deux types d'attributs :
- ✓ Attributs de classe : Caractéristique globale à l'ensemble des instances de la classe. Cette notion est similaire à une variable de classe.
- ✓ Attributs d'instance : caractéristique locale à une instance de la classe. Cette notion est similaire à une variable d'instance.

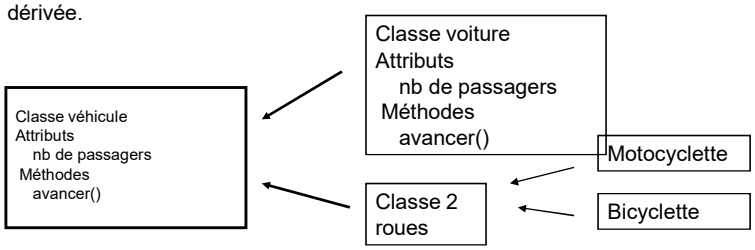
16

Héritage

➢ Regrouper les classes de même caractéristique et de même comportement.

➢ établit une relation entre deux classe A et B permettant de définir la classe B à partir de la classe A.

- ✓ Héritage de définition : les caractéristiques de la classe mère sont transmises à la classe dérivée.
- ✓ Héritage d'opération : les mises en œuvre de méthodes de la classe mère sont applicable à la classe dérivée.
- ✓ Héritage d'interface : la classe mère prévoit des comportement pour la classe dérivée.



17

Association vs. Héritage

➢ Association : « a-un», relations de composition, de collaboration entre les objets.

Les méthodes de A n'apparaissent pas dans l'interface de B. B contient une ou plusieurs instances de A.

➢ Héritage : « est-un», relations de spécialisation/généralisation, relations hiérarchique entre les objets.

L'interface de A apparaît au niveau de l'interface de B. B ajoute de nouvelles méthodes à celle de A.

18

Polymorphisme

➢ Propriété d'une méthode de pouvoir se comporter de manière différente
sur des classes différentes.

➢ Dissimuler des mises en œuvre différentes derrière la même interface.

➢ Surcharge :

- ✓ double mult (double , double);
- ✓ double mult (int , double);
- ✓ double mult (int , int);

19

Apport du modèle objet (1)

➢ En programmation structurée :

```
Recherche (x: element , t: table) : booleen
    début
            pos position;
            pos := premier (t);
            tant que  non épuisé (pos, t) faire
                    si  trouvé (pos, t, x)   alors retourner (vrai)
                pos := prochain (pos, t)
            Fait
            retourner (non);
    Fin
```

- ✓ Problème à résoudre
  - o Variations des types.
  - o Variations des structures de données et des méthodes.
  - o Grouper les routines.
  - o Indépendance vis à vis de la présentation.

20

Apport du modèle objet (2)

➢ En programmation structurée :

```
Pour chaque pièce faire
    switch type de la pièce
            cas  Tour : deplacerTour ()
            cas  Reine : deplacerReine()
            cas  Cavalier : deplacerCavalier ()
    fswitch
Fait
```

➢ En programmation par objet :

- o Définition d'une classe Pièce avec une méthode déplacer ()
- o Définition des sous-classes Tour,Reine et Cavalier de la classe de base Pièce.
- o Spécialisation de la méthode déplacer.

```
Pour chaque instance de type Pièce faire
        déplacerInstance ()
Fait
```

21

Apport du modèle objet (3)

➢ En programmation par objet :

✓ Solutions:

- o Généricité : indépendance de la recherche par rapport au type des éléments stockés dans la structure.
- o Héritage : Définition de l'interface et spécialisation selon les différentes organisations de données.
- o Classe : regroupant des routines implantant la recherche avec la structure de données dans laquelle la recherche est effectuée.
- o polymorphisme : définition d'un protocole de recherche identique à toutes les organisations de données.
- o Surcharge : factorisation du code de la mise en œuvre de la procédure de recherche.

22

LE LANGAGE DE PROGRAMMATION ORIENTE OBJET JAVA

23

Caractéristiques de Java

- **Un langage Orienté Objet**
  - pas de variables ni de fonctions globales, types primitifs et objet
- **Simple**
  - syntaxe C/C++ (types primitifs, instructions de test et boucles, …)
- **sûr**
  - pas de pointeurs, vérification du code à l 'exécution et des accès réseau et/ou fichiers
- **Robuste**
  - ramasse miettes, fortement typé, gestion des exceptions
- **Indépendant d 'une architecture**
  - Portabilité assurée par la présence d 'un interpréteur de bytecode sur chaque machine
- **Environnement riche**
  - classes standard complètes
  - classes pour les communications réseau
  - fonctions graphiques évoluées
  - ...

24

1er programme : afficher bonjour

```
import java.io.*;
public class Bonjour {
        public static void main(String args[]) {
                System.out.println("Bonjour le monde!");
        }
}
```

Compilation (JDK)

C:\prog_java\> **javac Bonjour.java**

→ **Création de Bonjour.class**

Exécution

C:\prog_java\> **java Bonjour**

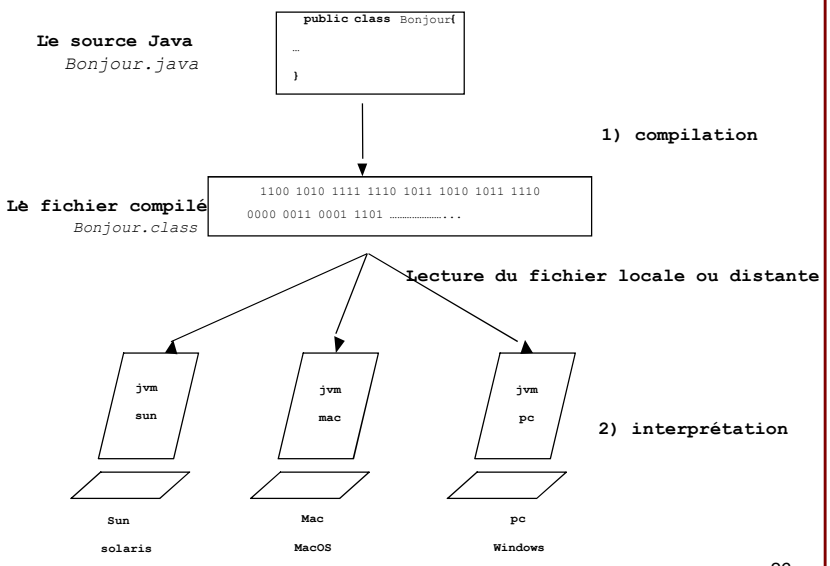
Attention: le répertoire où se trouve javac doit se trouver dans le PATH et prog_java dans la variable d'environnement CLASSPATH

2ème programme: Afficher bonjour en tenant compte d'un éventuel paramètre passé sur la ligne de commande

```
public class Bonjour2 {
 public static void main(String args[]) {
  if (args.length==0)
   System.out.println("Bonjour le monde!");
  else System.out.println("Bonjour "+args[0]+" ! ");
 }
}
```

25

Portable



26

Environnement (très) riche

- java.applet
- java.awt
- java.beans
- java.io
- java.lang
- java.math
- java.net
- java.rmi
- java.security
- java.sql
- java.util
- javax.swing
- org.omg.CORBA
- org.omg.CosNaming

- Liste des principaux paquetages de la plate-forme JDK 1.2
- soit environ 1500 classes !!! Et bien d 'autres A.P.I. JSDK, JINI, ...
- le JDK1.3/1850 classes
- JDK 1.4 encore plus, déjà le JDK 1.5 ….

27

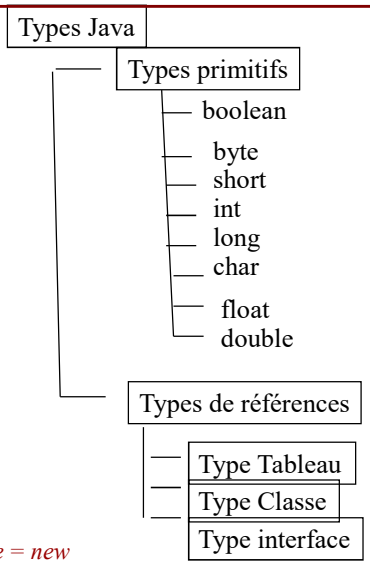
Le package java.lang
•Le package java.lang est chargé automatiquement, ses classes sont donc toujours utilisables. On y trouve, entre autres :
•la classe **Object** dont dérivent toutes les autres classes
•les classes représentant les types numériques de bases : **Boolean, Byte, Double, Float, Integer, Long**
•la classe **Math** qui fournit des méthodes de calcul des fonctions usuelles en mathématiques
•les classes **Character**, **String** et **StringBuffer** pour la gestion des caractères et des chaînes de caractères
•la classe **System** que nous utilisons pour afficher du texte sur la console DOS.

Les autres packages
Les autres packages doivent être déclarés (mot clé import) pour pouvoir être utilisés.
Nous aurons à étudier entre autres, les packages
java.awt pour la création d'interfaces graphiques
java.awt.event pour la gestion des évènements
java.util pour certaines structures de données
java.io pour la gestion des fichiers
java.net pour le téléchargement de fichiers

28

Variables et Objets

Une variable est :
- soit de type primitif
- soit une référence à un objet




```
int x, y;
boolean b = false;
double d = 3.14159;
```

```
Object
    PolygoneRegulier unPolygone = new
    PolygoneRegulier();
    Carré unCarre = new Carre();
    Object o = new Object();
```

29

Exemples: → *syntaxe de C : déclaration de variables de types primitifs Structures de contrôles*

```
class Compare1 {
 public static void main(String args[]) {
  int a = 10;
  int b = 12;
  System.out.println("L'entier A vaut
"+a+".");
  System.out.println("L'entier B vaut
"+b+".");
  if (a<b)
   System.out.println("A est plus petit
que B.");
  else if (a == b)
   System.out.println("A est egal à B.");
  else System.out.println("A est plus
grand que B.");
 }
}
```

```
public class note {

 public static void main(String args[]) {
  if (args.length==0) {
   System.out.println("Passer une note en
paramètre !");
   System.exit(1);
  }
  else {
   int n=Integer.parseInt(args[0]);
   if (n>=10)
    System.out.println("Vous êtes reçu.");
   else if (n>=8)
    System.out.println("Vous devez passer
l'oral.");
   else System.out.println("Vous êtes recalé.");
  }
 }

}
```

30

Concepts de l'orienté objet

- **Le vocable Objet :**

- Classe et objet (instance d 'une classe)
- Etat d'un objet et données d'instance
- Comportement d'un objet et méthodes
- Composition et interaction des objets
  - Est composé de, est constitué de
- Héritage
  - Le verbe être
- Polymorphisme

31

Classes et objets

Exemple de programme

```
class Circle{
    public double x, y; // Coordonnée du centre
    private double r;              // rayon du cercle

    public Circle(double r) {
        this.r = r;
        this.x=0; this.y=0;
    }
    public double area() {
        return 3.14159 * r * r;
    }
}

public class MonPremierProgramme {
    public static void main(String[] args) {
        Circle c;               // c est une référence sur un objet Circle, pas un objet
        c = new Circle(5.0);  // c référence maintenant un objet alloué en mémoire
        c.x = c.y = 10;
        System.out.println("Aire de c :" + c.area());
    }
}
```

32

Encapsulation

- **contrat avec le client**
  - interface publique
  - implémentation privée, ce sont des choix d'implémenteurs

Héritage et classification

- définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante
  - ajout de nouvelles fonctions
  - ajout de nouvelles données
  - redéfinition de certaines propriétés héritées (masquage)

- Une approche de la classification en langage naturel
- **Les carrés** sont **des polygones réguliers (ce serait l 'idéal**[33]**)**

Polymorphisme : définitions

- Polymorphisme ad'hoc

  Surcharge(overloading),

  plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est résolu statiquement dès la compilation
- Polymorphisme d'inclusion **(overridding),**

  est fondé sur la relation d'ordre partiel entre les types, relation induite par l'héritage. si le type B est inférieur selon cette relation au type A alors on peut passer un objet de type B à une méthode qui attend un paramètre de type A, le choix de la méthode est résolu dynamiquement en fonction du type de l'objet receveur
- Polymorphisme paramètrique

  ou généricité,

  consiste à définir un modèle de procédure, ensuite incarné ou instancié avec différents types.

34

Exemple en syntaxe Java

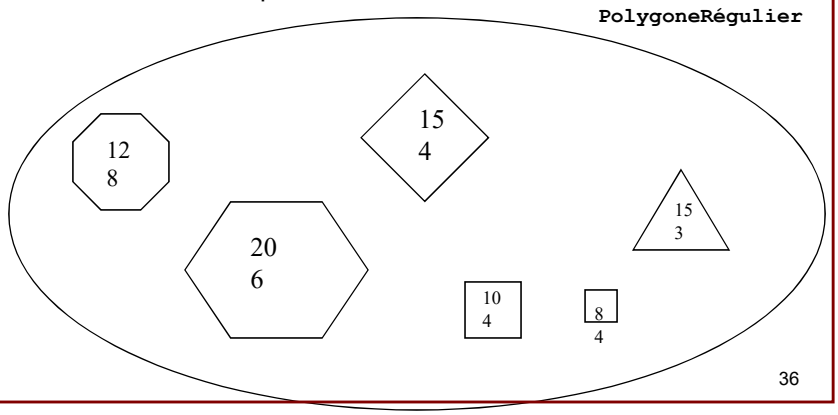
```
public class PolygoneRégulier{
    int longueurDuCôté;
    int nombreDeCôtés;
    public void initialiser(int nombre, int longueur){
        longueurDuCôté = longueur;
        nombreDeCôtés = nombre;
    }
    public int périmètre(){
        return longueurDuCôté * nombreDeCôtés ;
    }
    public int surface(){ …..}
}
// un usage de cette classe : objet p instance de la classe PolygoneRegulier
PolygoneRégulier p = new PolygoneRégulier();
p.initialiser(4,100);
int y = p.périmètre();
```

35

Champs d 'instance, état

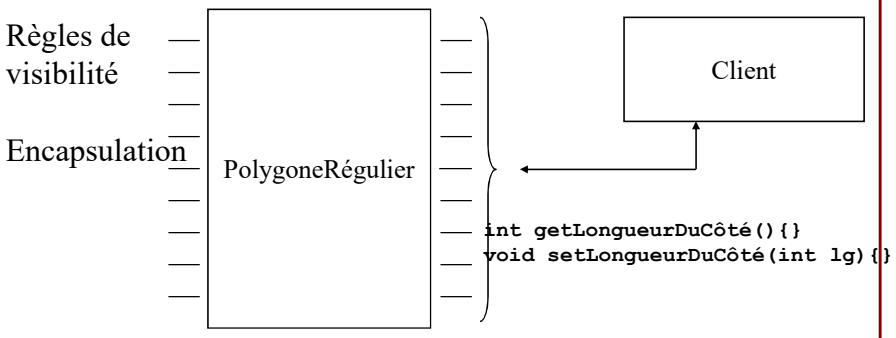
```
public class PolygoneRégulier{
    int longueurDuCôté;
    int nombreDeCôtés;
```

- Valeurs des champs de l 'instance : son état



36

Champs d 'instances privées



Règle : Les champs sont privés et accessibles par des méthodes

```
public class PolygoneRégulier{
    private int nombreDeCôtés;
    private int longueurDuCôté;
```

37

Méthodes/signatures

```
public class PolygoneRégulier{
    private int longueurDuCôté;
    private int nombreDeCôtés;

    public void initialiser(int nombre, int longueur){
            longueurDuCôté = longueur;
            nombreDeCôtés = nombre;
     }

public int périmètre(){
    return longueurDuCôté * nombreDeCôtés ;
     }

}
```

38

Méthodes « accesseur »

Par convention public *type* get***NomDuChamps***

```
public class PolygoneRégulier{
    private int longueurDuCôté;
    private int nombreDeCôtés;

  public int getNombreDeCôtés(){
    return nombreDeCôtés;
  }

  public int getLongueurDuCôté(){
    return longueurDuCôté;
  }
```

39

Méthodes « mutateur »

Par convention public *void* set***NomDuChamps(type id)***

```
public class PolygoneRégulier{
    private int longueurDuCôté;
    private int nombreDeCôtés;

  public void setLongueurDuCôté(int longueur){
    longueurDuCôté = longueur;
  }

  public void setNombreDeCôtés(int nombre){
    nombreDeCôtés = nombre;
  }
```

40

Constructeur

- **PolygoneRégulier unPolygone =** new PolygoneRégulier();
- unPolygone.initialiser(4,100)

- en plus concis, plus naturel ...
  - **PolygoneRégulier unPolygone =** new PolygoneRégulier(4,100);

```
public class PolygoneRégulier{
    private int longueurDuCôté;
    private int nombreDeCôtés;

    public PolygoneRégulier(int nombre, int longueur){
            longueurDuCôté = longueur;
            nombreDeCôtés = nombre;
    }
```

41

Constructeurs

- Plusieurs constructeurs sont possibles

CLASSE

```
public class Point {
    // ...
    Point(double ax,double ay) {
        x = ax;
        y = ay;
    }
    Point() {
        x = 0;
        y = 0;
    }
}
```

INSTANCES

```
Point p = new Point(1,2);
`
Point q = new Point();
```

42

- **Constructeurs par défaut**

CLASSE

```
public class Point {
    public double x;
    public double y;
}
```

INSTANCES

```
Point p = new Point();

Point q = new Point(1,2);
```

refusé à la compilation

CLASSE

```
public class Point {
    public double x;
    public double y;
    Point(double ax,double ay) {
        x = ax;
        y = ay;
    }
}
```

INSTANCES

```
Point p = new Point();

Point q = new Point(1,2);
```

CLASSE

```
public class Point {
    // ...
    Point(double x,double y) {
        x = x;
        y = y;
    }
    Point() {
        x = 0;
        y = 0;
    }
```

ambiguité!

redondance

CLASSE

```
public class Point {
    // ...
    void Point(double x,double y) {
        this.x = x;
        this.y = y;
    }
    void Point() {
        this(0,0);
    }
```

43

- **Constructeurs *par copie***

CLASSE

```
public class Point {
    public double x;
    public double y;
    Point(double ax,double ay) {
        x = ax;
        y = ay;
    }
    Point(Point p) {
        this(p.x,p.y);
    }
}
```

INSTANCES

```
Point p = new Point(1,2);

Point q = new Point(p);
```

44

```
public class Complexe
{
        private int Entier;
        private int Abstrait;
        public Complexe (int E, int A)
        {       Entier = E;
                Abstrait = A;
        }
        public Complexe ()
        {       this (0,0); }
        public void Afficher()
        { System.out.println ("" + Entier + " " + Abstrait + "i");      }
}
```

```
public class Exemple1
{     public static void main (String Arg[])
        {       Complexe N1;
                N1 = new Complexe(12,3);
                N1.Afficher();
                Complexe N3 = N1;
                N3.Afficher();
                (new Complexe()).Afficher();
        }
}
```

Les Tableaux en JAVA

Les tableaux en JAVA sont également des objets.

Ils ont tous un champs *length*, qui donne la longueur du tableau.

On peut faire des tableaux de tout type ou de plusieurs dimensions.

46

```
class Tableau
{
	public static void main (String Arg[])
{
	int Tab1[];
	Tab1 = new int[5];
	int Tab2[] = new int [5];
	int Tab3[][] = new int [5][5];
	for (int x=0; x<5; x++)
	{
		Tab1[x] = x;
		Tab2[x] = x;
		for (int y=0; y<5; y++)
			Tab3[x][y] = x*5 + y;
	}
	System.out.println ("Tab1 :" + Tab1.length);
		System.out.println ("Tab2 :" + Tab2.length);
		System.out.println ("Tab3 :" + Tab3.length);
		System.out.println ("Tab3[2] :" + Tab3[2].length);
}
}
```

47

La classe String

Déclarer et initialiser:

```
String souhait = "Bonne soirée" ;
String nomPre = new String("Ben Moh");
```

la longueur (le nombre de caractères) :
public int **length**()
exemples : souhait.length() vaut 12
"A BC".length() vaut 4

accès à un caractère à un indice donné
public char charAt(int index)
exemples :
souhait.charAt(0) vaut la lettre 'B'

48

conversion :
public String toLowerCase() // en minuscules
public String toUpperCase() // en majuscules
exemples :
String nom1 = " DougA" ;
String nom2 = nom1.toLowerCase();

comparaison :
public boolean equals (Object unObjet)
exemples :
"Bon".equals("Bon") vaut true
"Bon".equals("bon") vaut false

public boolean equalsIgnoreCase(String autre)
(sans tenir compte de la casse (Maj. vs minuscule)
"Bon".equalsIgnoreCase("bon") vaut true

public int compareTo(String autre)
"Bon".compareTo("Bon") vaut 0 (identiques)
"Bon".compareTo("Automne") > 0
"Bon".compareTo("Dinde") < 0

49

recherche :

public int indexOf(char unCar)

public int lastIndexOf(char unCar)

public int indexOf(String ch)

public int lastIndexOf(String ch)

Sous chaines :

public String substring(int debut, int fin)

public String substring(int debut)

50

La classe StringBuffer

- Si la chaine doit etre souvent modifiée, on utilise la classe StringBuffer qui posède des méthodes qui modifient le receveur du message et évitent la création de nouvelles instances
  - append
  - insert
  - replace
  - delete

51

Exercice 2:

Programmer un objet *calculatrice* qui reçoit deux entiers demandés à l'usager et qui retourne le résultat de l'addition. Utiliser la classe Utile, ici bas.

```
import java.io.*;
public class Utile
{

 public static String lireChaine(String message) throws IOException {

        BufferedReader entree = new BufferedReader
                              (new InputStreamReader(System.in));
        System.out.print(message);
        return entree.readLine();
 }

 public static int lireEntier(String message) throws IOException {
         return Integer.parseInt(lireChaine(message));
 }

public static char lireCaractere(String message) throws IOException
 {
   return lireChaine(message).charAt(0);
 }

]
```

52

Les classes « enveloppe »

- Permettent de représenter des types de base sous forme d'objets
- int entier => Integer n
- Integer n = new Integer(entier);
- entier = n. intValue();
- double =>Double, boolean => Boolean…
- intValue(), doubleValue() ...

53

Conversions de type

Les méthodes de conversion de type sont stockées dans des classes :

class java.lang.String

class java.lang.Integer

class java.lang.Long

class java.lang.Float

class java.lang.Double
…

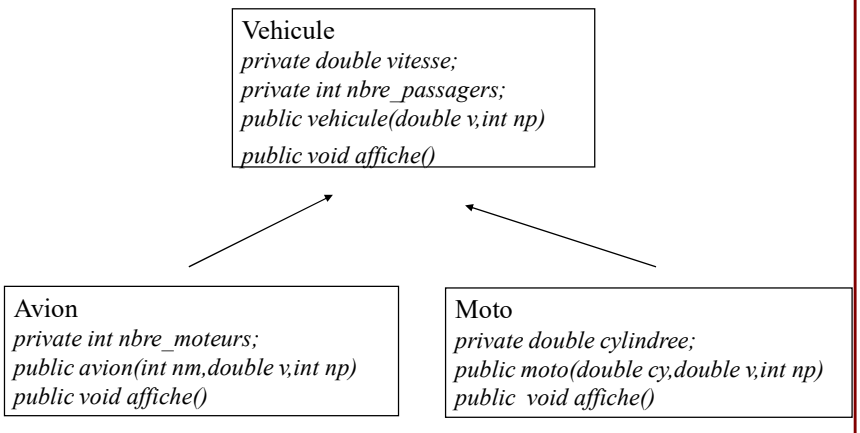
54

Exemples de conversion de type

```
int i = 10 ;
String chaine = String.valueOf (i) ;
String chaine="123" ;
int entier = Integer.parseInt (chaine) ;
class Integer
{
        ……
        public static int parseInt(String s) throws
        NumberFormatException;
        ……
}
try {
        entier = Integer.parseInt (chaine) ;
} catch (NumberFormatException e)
{
        // Si la variable chaine n'est pas convertible on
vient ici
}
```

55

L'héritage



56

Usage de l'héritage

- Une classe spécifique hérite des méthodes et des attributs de sa classe mère (accède a tous sauf ceux qui sont privés)
- On n'a pas besoin de les réécrire pour la classe fille
- On peut cependant redéfinir une méthode de la classe mère dans la classe fille (de même signature)
- Le constructeur d'un objet doit toujours commencer par appeler le constructeur de sa classe mère
- Un objet de type Voiture peut utiliser toutes les méthodes de la classe Vehicule
- Il doit disposer d'une valeur pour tous les attributs de la classe Vehicule
- A tout moment, une méthode qui utilise un objet de type Vehicule peut manipuler un objet de type Voiture en guise de Vehicule
- Cette dernière propriété est le polymorphisme.

57

Exemple

Classe de base: Vehicule

```
class Vehicule {
    private double vitesse;
    private int nbre_passagers;
    public Vehicule(double v,int np) {
       vitesse = v;
       nbre_passagers = np;
    }
    public void affiche() {
       System.out.println("vitesse: " + vitesse +
" ; nbre_passagers: " + nbre_passagers);
    }
}
```

58

Exemple (suite1)

Classe Dérivée: Avion

```
class Avion extends Vehicule {
     private int nbre_moteurs;
      public Avion(int nm,double v,int np) {
        super(v,np); // appel au constructeur de vehicule
        nbre_moteurs = nm;
     }
     public void affiche() {

        //appel de la méthode affiche de la classe de base
        super.affiche();

        System.out.println("nbre de moteurs: " + nbre_moteurs);
     }
}
```

59

Exemple (suite2)

Classe Dérivée: Moto

```
class Moto extends Vehicule {
     private double cylindree;
     public moto(double cy,double v,int np) {
        super(v,np); // appel au constructeur de vehicule
        cylindree = cy;
     }
     public void affiche() {
        super.affiche();
        System.out.println("La cylindrée de la moto: " +
                                              cylindree);
     }

}
```

60

Exemple (suite3)

Classe de test:

```
public class Test {
        public static void main(String [] args) {
       vehicule v = new vehicule(200.,300);
       v.affiche();
       avion boeing = new avion(4,200.,300);
       boeing.affiche();
       moto suzuki = new moto(1200.,200.,2);
       suzuki.affiche();
       System.exit(0);
    }
}
```

61

La classe Object

- **La classe Object est la classe mère dont héritent toutes les autres classes**
- **Toute classe définie en Java, que ce soit par l'utilisateur ou dans un package de l'API, hérite de la classe Object**
- **Ainsi, toute classe hérite d'emblée des méthodes de la classe Object**

62

Méthodes de la classe Object

- **Public final Class getClass()** renvoie le type de l'objet sous forme d'un objet de la classe Class (réflexivité)
- *if ((compte.getClass().getName()).equals("BanqueL.CompteEpargne")) (< == > if (compte instanceOf("BanqueL.CompteEpargne")*
- **forName()** renvoie un objet du type spécifié dans la chaîne, permet de construire un objet sans avoir de constructeur (réflexivité)

  Class classe = Class.forName("java.lang.String"); System.out.println("classe de l'objet chaine = "+classe.getName());
- **void finalize()** appelée par le Garbage Collector avant destruction de l'objet
- **boolean equals(Object o)** attention, == teste les références. Mais compare les références des attributs (=> redéfinir)
- **Object clone()** : crée une copie de l'objet. Mais copie les références des attributs (=> redéfinir)
- **String toString()** renvoie une String décrivant l'objet. Par défaut, renvoie le type et l'adresse de stockage (=> redéfinir)
- ...

63

```
class Point extends Object {
    private int x, y ;
    public Point() {…}
    public Point(int _x, int _y) {…}
    public int getX() {…}
    public int getY() {
    public static void swap(Point a, Point b){int tmp=0;
           tmp=a.getX();a.setX(b.x); b.setX(tmp);  … }
         //ou  Point tmp= new Point(); tmp=a; a=b; b=tmp;
    public boolean equals(Object o) {
       return (o != null) && ((this==(Point)o)
             || ((this.x == ((Point)o).x)
                && (this.y == ((Point)o).y))) ;
    }
     public String toString() {
       return "("+x+", "+y+")" ;
```

64

```
class TestPoint {
    public static void main(String[] args) {
       Point k = new Point() ;
       Point m = new Point(1,5) ;
      // grâce à la méthode toString
       System.out.println("k av : "+k+" m av : "+m) ;
       Point.swap(k, m) ;
       System.out.println("k : "+k+" m : "+m) ;
       Point p = new Point(1, 2);
       Point q = new Point(1, 2);
       Object r = new Point(1, 2);
       System.out.println(p.equals(q)) ;
       System.out.println(r.equals(p)) ;
       System.out.println(p.equals(r)) ;
    }
}
```

65

```
package fornametest;

import java.lang.reflect.Method;

 class ClassForNameTest {
   public void getData() {
      System.out.println("execution getData.....");
   }
}

public class ForNameTest {
   public static void main(String args[]) throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
      Class loadclass = Class.forName("fornametest.ClassForNameTest");
      System.out.println("number of methods:" + loadclass.getMethods().length);
      Method[] method = loadclass.getMethods();
      for (Method m : method) {
         System.out.println("Method Names:" + m.getName());
      }
      ClassForNameTest obj = (ClassForNameTest) loadclass.newInstance();
      obj.getData();
   }
}
```

```
class Test
{
   int x, y;
}

class Test2 implements Cloneable
{
   int a;
   int b;
   Test c = new Test();  // une réference de l'objet Test
   public Object clone() throws
            CloneNotSupportedException
   {
      return super.clone();   //copie superficielle
   }
}
```

```
public class Main
{
   public static void main(String args[]) throws
                  CloneNotSupportedException
   {
      Test2 t1 = new Test2();
      t1.a = 10;
      t1.b = 20;
      t1.c.x = 30;
      t1.c.y = 40;

      Test2 t2 = (Test2)t1.clone();
      // Le changement de type primitif de t2 ne sera pas
      //  reflété dans le champ t1
      t2.a = 100;
      // Le changement de champ de type d'objet sera
      // reflété à la fois dans t2 et t1 (copie superficielle)
      t2.c.x = 300;

      System.out.println(t1.a + " " + t1.b + " " +
                  t1.c.x + " " + t1.c.y);
      System.out.println(t2.a + " " + t2.b + " " +
                  t2.c.x + " " + t2.c.y);
   }
}
// 10 20 300 40
// 100 20 300 40
```

67

Le typage dynamique

- **Les objets connaissent en permanence le type selon lequel ils ont été créés**
- **Il peut donc y avoir différence entre le type de la référence et celui de l'objet**
- **Exemple :**

Point p = new PointCol();
System.out.println(p.getClass().getName());
=> Affiche PointCol et non pas Point

- **p.afficher () va exécuter afficher() de la classe PoinCol**

(si afficher() existe dans les 2 classes)

- **Pour que l'on puisse compiler et exécuter, il faut que le type effectif hérite du type déclaré**

68

Final

- **Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.**

```
public final void méthode() {
  ...
}
```

- **Une classe déclarée final ne peut plus être dérivée**

```
public final class ExempleClasse {
  ...
} //interdit l'héritage
```

On est ainsi certain que le contrat de la classe sera respecté.
cela permet d'interdire la redéfinition d'une méthode par le biais de l'héritage
ex : final boolean checkPassword()

- **Une variable ou un attribut déclarés final sont constants => le compilateur interdit de les modifier**

69

Les classes abstraites

- **Une classe abstraite est une classe qui ne permet pas d'instancier des objets.**
- **Elle ne peut servir que de classe de base pour une dérivation.**

Elle se déclare ainsi:

```
abstract class A {
  ...
}
```

70

Classe Abstraite: Règles

Dès qu'une classe abstraite comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé abstract devant sa déclaration).

- **A est considérée comme abstraite → 1 expression telle que new A(..) sera rejetée.**
- **Une méthode abstraite doit obligatoirement être déclarée public, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.**
- **Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.**

71

```
public abstract class FigureGeometrique {
        private double x, y;
        private String couleur ;
        public FigureGeometrique () {
                x=0; y=0;   couleur = "Bleu" ;
        }
        public FigureGeometrique(String _couleur, double_x, double_y ) {
                x= _x;      y= _x;
                couleur = _couleur ; }
        public void deplacer  (int dx, int dy)     {  x +=dx;  y +=dy;       }
        public String toString ()   {  return getClass (). getName();       }
        abstract public double perimetre ();
        abstract public double surface ();
}
```

72

```
public class Carre extends FigureGeometrique {
    private double cote ;
    public Carre(double _cote ) {
                super();
            cote = _cote ;
    }
    public Carre(String _couleur , double_x, double_y, double _cote ) {
        super(_couleur, _x, _y );
         cote = _cote ;
    }
    public String toString () {
            return super. toString () + " : de côté " + cote ;
    }
    public double perimetre () { return 4 * cote ; }
    public double surface () {  return cote * cote ;  }
}
```

73

```
public class Rectangle extends FigureGeometrique {
    private double coteX, coteY;
    public Carre(double _coteX ,  double _coteY) {
         super();
          coteX = _coteX ; coteY= _coteY;
    }
    public Carre(String _couleur , double_x, double_y, double _coteX,
    double _coteX) {
        super(_couleur, _x, _y );
        coteX = _coteX ; coteY = _coteY ;      }
    public String toString () {
        return super. toString () + " : de côtés" + coteX + " et  "  + coteY ;
    }
    public double perimetre () { return 2 * (coteX+coteY) ; }
    //public double surface () {  return coteX * coteY;  }
}
```

74

```
public class AbstractOG {

    public static void main(String[] args) {

        FigureGeometrique figures [] = new FigureGeometrique[2];
        figures[0] = new FigureGeometrique ();
        figures [0] = new Carre(5.0);
        figures [1] = new Rectangle(3.0,4.0);

        for (int i =0; i<figures.length; i++) {
                System.out.println(figures[i]+
                        " \nde perimetre = "+ figures[i].perimetre()+"\n");
        }
    }
}
```

---

run:
Carre : de côté 5.0
de perimetre = 20.0

Rectangle : de côtés 3.0 et 4.0
de perimetre = 14.0

75

Les Interfaces

Les interfaces sont des classes qui ne contiennent que:

- **des attributs statiques finaux (attributs: public static final)**
- **des méthodes publiques abstraites (public abstract)..**

On les définit avec le mot clé « *interface* » au lieu de « *class* ».

On peut implémenter plus d'une interface sur une classe en les séparant par des virgules.

On utilise le mot clé « *implements* » pour signaler qu'une classe hérite « est enfant » d'une interface.

76

Une interface est un type représentant une promesse de service ; toute variable possédant ce type peut rendre ce service.

```
interface Coloré {
    int getCouleur( ) ;  // fonction abstraite
}
class PointColoré extends Point implements Coloré {
    private int couleur ;
    public PointColoré( int x , int y , int col ) {
        super(x , y ) ;          couleur = col ;
    }
    public int getCouleur( ) {   return couleur ; }
}
class SegmentColoré extends Segment implements Coloré {
    private Point a, b; // les extrémités
    private int couleur ;
    public int getCouleur( ) {
            return couleur ;
    }}
```

77

Exemple d'utilisation de l'interface Comparable de l'API Java (java.lang)

```
interface Comparable {
int compareTo(Object o);
}

public class Etudiant implements Comparable<Etudiant>{
    int cne;
    String nom;
    int age;

    Etudiant(int cne,String nom,int age){
        this.cne=cne;
        this.nom=nom;
        this.age=age;
    }

    public int compareTo(Object st){
        if(age== <Etudiant > st.age)
            return 0;
        else if(age> <Etudiant > st.age)
            return 1;
        else
            return -1;
}}
```

78

Exemple d'utilisation de l'interface Comparable de l'API Java

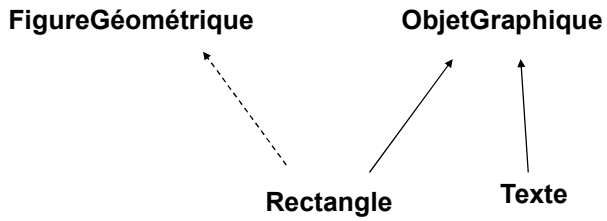
```
public static void main(String[] args) {
      ArrayList<Etudiant> classeM =new ArrayList<Etudiant>();
      classeM.add(new Etudiant(101,"Khalid",23));
      classeM.add(new Etudiant(106,"Rachid",27));
      classeM.add(new Etudiant(105,"Adil",21));

      Collections.sort(classeM);
      for(Etudiant etudiant:classeM){
         System.out.println(etudiant.cne+" "+etudiant.nom+" "+etudiant.age);
       }

  }
```

---

run:
105 Adil 21
101 Khalid 23
106 Rachid 27

Exercice: Triez pareillement la classe banque (solde ou par rapport au nom)

79



Une classe ne peut hériter que d'une seule structure (mot-clef extends) mais ellepeut hériter de plusieurs interfaces (mot-clef implements).

class B extends A implements I , J { /* ... */ }

Exemple:

```
public class Etudiant implements Comparable<Etudiant>, Serialisable  {

            …
}
```

80

Interface	Abstract Class
• spécifier la forme d'un concept: non le mettre en œuvre • ne peut pas avoir de données membres, constantes uniquement • léger à mettre en œuvre • implémentations multiples • héritage multiple	• classe incomplète (peut avoir implémentations partielles) qui a besoin d'une spécialisation (dérivation) • peut avoir des membres de données • classe de base: utilisée pour initialiser une hiérarchie de classes • héritage unique

81

La Gestion des exceptions

•Le traitement des exceptions permet à une application d'avoir un comportement adéquat à une situation inattendue.

•Beaucoup de méthodes sont susceptibles de déclencher une exception.

•Une exception peut être levée par une clause *try/catch* .

La clause try/catch

```
try <instruction> ;
        catch (<type exception1> ex1) instruction ;
        catch (<type exception2> ex2) instruction ;
        ....
finally <instruction> ; // en tt cas faire instruction
```

82

Exemples de gestion d'exceptions

```
String chaine ;
int valeur ;
…
try {
        valeur = Integer.parseInt (chaine) ;
} catch (NumberFormatException e)
{
        System.out.println ("mauvais format ") ;
}
try a = tab [i] ;
catch (ArrayIndexOutOfBounds ex) a = 0 ;
```

83

```
public  static void  fonction (int a, int b)  {

    Scanner clavier = new Scanner(System.in);

    try {
        System.out.print("Saisir a : ");
        a = clavier.nextInt();

        System.out.print("Saisir b : ");
        b = clavier.nextInt();

        System.out.println("a+b = " + (a + b));
    } catch (InputMismatchException e) {
        System.out.println(e);
    }

     clavier.close();
 }
```

84

Exceptions

• Une méthode pouvant produire une exception doit être appelée dans un bloc try/catch.

• On indique qu'une méthode m peut générer une exception E :

void m () throws E
{
…
}

• Le programmeur java a un devoir de captage des exceptions dans les fonctions appelantes pour ne pas désemparer les utilisateurs.

La fonction throw permet lancer un objet de type Exception

85

```
public static  void fonction (int a, int b) throws  InputMismatchException{

    Scanner clavier = new Scanner(System.in);

        System.out.print("Saisir a : ");
        a = clavier.nextInt();

        System.out.print("Saisir b : ");
        b = clavier.nextInt();

        System.out.println("a+b = " + (a + b));
        clavier.close();
 }

 public static void main(String args[]) {
    int a=0, b=0;
    try {
        fonction(a,b);
    } catch (InputMismatchException e) {
        System.out.println(e);
    }
 }
```

86

Gestion de fichiers, flux d'entrées/sorties, sérialisation

87

Gestion de fichiers

- La gestion de fichiers se fait par l'intermédiaire de la classe *java.io.File*.
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de fichiers du système d'exploitation.
- Un objet de la classe *java.io.File* peut représenter un fichier ou un répertoire.

88

Quelques méthodes de la classe java.io.File

```
File (String name)
File (String path, String name)
File (File dir, String name)
boolean isFile ()
boolean isDirectory ()
boolean mkdir ()
boolean exists ()
boolean delete ()
boolean canWrite ()
boolean canRead ()
File getParentFile ()
long lastModified ()
String [] list ()
boolean createNewFile();
…
```

89

Exemple d'utilisation de la classe

```
java.io.File
import java.io.* ;
public class ExempleFile
{
        static public void main (String args []) { new ExempleFile () ; }
        ExempleFile () { liste (new File ("d:\\")) ; } //banalisation du
caractère spécial  \
        private void liste (File dir)
        {
                if (dir.isDirectory () == true)
                {
                        String fichiers [] = dir.list () ;
                        for (int i = 0 ; i != fichiers.length ; i++)
                                System.out.println (fichiers [i]) ;
                }
                else
                {
                System.err.println (dir + " n'est pas un repertoire") ;
                }
        }
```

90

Les flux

• Difficulté d'un langage d'avoir un bon système d'entrées/sorties.

• Beaucoup de sources d'E/S de nature différentes (console, fichier, socket,…).

• Beaucoup d'accès différents (accès séquentiel, accès aléatoire, mise en mémoire tampon, binaire, caractère, par ligne, par mot, etc.).

• Un flux (stream) est un chemin de communication entre la source d'une information et sa destination

91

Les flux proposés par java

• Flux d'entrée/sortie de bytes.
java.io.InputStream
java.io.OutputStream

• Flux d'entrée/sortie de caractères depuis la version 1.1 de java.
java.io.Reader
java.io.Writer

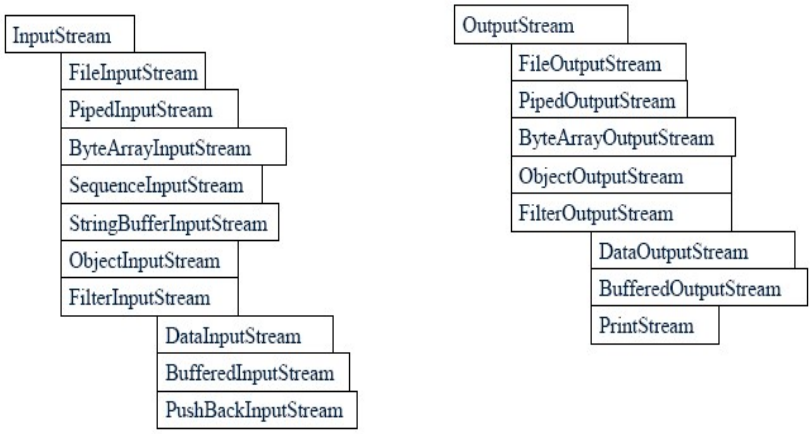
•Toutes les classes d'entrée/sortie sont dans le package *java.io*

•Toutes les méthodes peuvent générer une *java.io.IOException*

•*Le package java.io définit plusieurs classes qui héritent de ces classes*

92

Classes de flux de bytes



93

La classe java.io.InputStream

- **Les méthodes de lecture :**

```
public int read () ;
public int read (byte b [ ]) ;
public int read (byte b [ ], int off, int len) ;
```

- **Exemple :**

```
InputStream s = ..... ;
byte buffer [ ] = new byte [1024] ;
try {
s.read (buffer) ;
} catch (IOException e)
{ }
```

94

Exemple de flux d'entrée

```
import java.io.* ;
public class LitFichier
{
        public static void main (String args [])
        {
           try {
           InputStream s = new FileInputStream ("d:\\temp\\data.txt") ;
           byte buffer [ ] = new byte [s.available()] ;
           s.read (buffer) ;
           for (int i = 0 ; i != buffer.length ; i++)
                 System.out.print ( (char) buffer [i]) ;
            } catch (IOException e)
                 {
                    System.err.println ("Erreur lecture") ;
                 }
        }
}
```

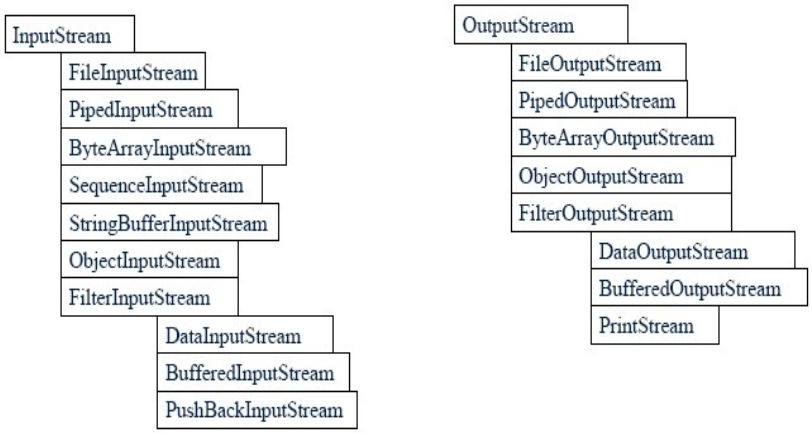
95

Exemple de flux de sortie

```
import java.io.* ;
public class EcritFichier
{
        static public void main (String args [])
        {
          String chaine = " 12.5" ;
          try {
         OutputStream f = new FileOutputStream ("c:\\temp\\data.txt") ;
          f.write (chaine.getBytes ()) ;
          f.close () ;
           } catch (IOException e)
             {
                 System.err.println ("Erreur ecriture") ;
             }
        }
}
```

96

Classes de flux de bytes



97

Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	Oui
Concaténation de flux	Sequence	Oui pour flux d'octets	Non
Conversion de données	Data	Oui pour flux d'octets	Oui pour flux d'octets
Numérotation des lignes	LineNumber	Oui pour les flux de caractères	Non
Lecture avec remise dans le flux des données	PushBack	Oui	Non
Impression	Print	Non	Oui
Sérialisation	Object	Oui pour flux d'octets	Oui pour flux d'octets
Conversion octets/caractères	InputStream / OutputStream	Oui pour flux d'octets	Oui pour flux d'octets

98

Ces deux classes servent de classes de base à des classes de gestion d'entrées/sorties plus évoluées:

• ***BufferedInputStream*** **et** ***BufferedOutputStream*** **permettent de lire et écrire des données à travers un tampon de lecture/écriture pour améliorer les performances.**

• ***DataInputStream*** **et** ***DataOutputStream*** **permettent de lire/écrire des données formatées (byte, int, char, float, double, etc.)**
• etc...

Exemples de lecture/écriture évoluée

---------- LIRE ----------

```
InputStream s = new FileInputStream ("fichier.txt") ;
DataInputStream data = new DataInputStream (s) ;
double valeur = data.readDouble () ;
```

---------- ECRIRE ----------

```
DataOutputStream o= new DataOutputStream (new FileOutputStream
("resultat.dat")) ; int n=30;
o.writeInt (n) ;
```

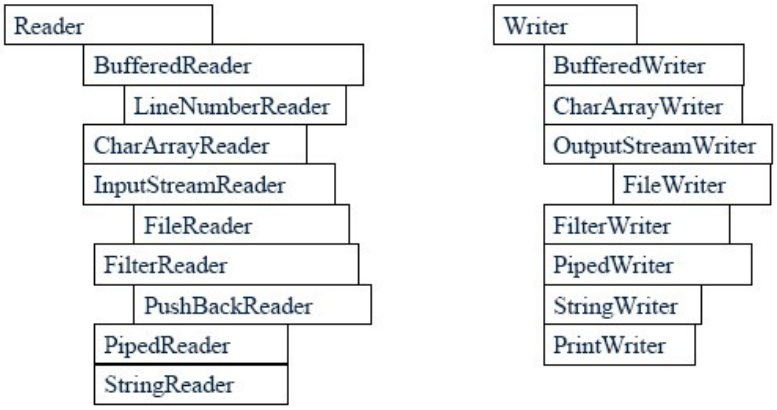
99

```
DataOutputStream dataOut = new DataOutputStream(new FileOutputStream("D:\\file.txt"));
dataOut.writeUTF("hello");

// Reading data from the same file
DataInputStream dataIn = new DataInputStream(new FileInputStream("D:\\file.txt"));

while(dataIn.available()>0) {
   String k = dataIn.readUTF();
   System.out.print(k+" ");
}
```

```
BufferedInputStream bin=new BufferedInputStream( ( FileInputStream("D:\\file.txt"));
int i;
while((i=bin.read())!=-1){
 System.out.print((char)i);
}
```

100

Les classes de flux de caractères



101

Exemple de BufferedReader

```
import java.io.*;
public class TestBufferedReader
{
        public static void main(String args[])
        {
           try {
           String ligne ;
           BufferedReader fichier = new BufferedReader
(new
FileReader("data.txt"));
            while ((ligne = fichier.readLine()) != null)
                System.out.println(ligne);
            fichier.close();
           } catch (IOException e)
                {
                System.err.println ("Erreur lecture") ;
                }
        }
}
```

102

```
import java.io.*;
public class Utile     // fait la même chose que Scanner
{

 public static String lireChaine(String message) throws IOException {

       BufferedReader entree = new BufferedReader
                              (new InputStreamReader(System.in));
       System.out.print(message);
       return entree.readLine();
 }  //idem que nextLine() de Scanner

 public static int lireEntier(String message) throws IOException {
        return Integer.parseInt(lireChaine(message));
 } // idem que nextInt de Scanner

public static char lireCaractere(String message) throws IOException
                                        {
                                return lireChaine(message).charAt(0);
                                        }

} // ...   lireDouble() ...

Int i=Utile.lireEntier(" entrez un entier:");
```

103

Quelques types flux I/O

- **Buffering(BufferdReader/BufferdWriter ; BufferdInputStream/BufferdOutputStream):** une zone tampon, pour réduire les accès en lecture/écriture
- **DataConversion(DataInputStream/DataOutputStream):** lecture/écriture dans un format indépendant de la machine.
- **File(FileReader/FileWriter ; FileInputStream/FileOutputStream):** lire et écrire à partir de fichiers
- **Printing(PrintWriter ; PrintOutputStream):** contient des méthodes données pour imprimer suivant un format donné.

104

La sérialisation

•La sérialisation d'un objet est le processus de stockage d'un objet complet (par exemple sur un disque).

• Le processus inverse est la désérialisation.

• Un objet sérialisé est dit persistant.

• Cette fonctionnalité est apparue dans la version 1.1 de Java.

105

Enregistrer des données sur disque nécessite une convention de format pour la lecture/écriture (cela peut être une opération complexe et difficile à maintenir).

 La sérialisation permet de rendre un objet persistant de manière simple et naturelle.

Si un objet contient d'autres objets sérialisables, ceux-ci seront automatiquement sérialisés.

 La plupart des classes de base (mais pas toutes) du langage Java sont sérialisables.

 Si la classe a été modifiée entre la sérialisation et la désérialisation, l'exception *java.io.InvalidClassException* est déclenchée.

106

Comment sérialiser

Une classe est sérialisable si elle implémente l'interface *java.io.Serializable*.

• Des objets de type *java.io.ObjectInputStream* et *java.io.ObjectOutputStream* vont permettre de sérialiser/désérialiser.

• Les données membres que l'on ne souhaite pas sauvegarder doivent être déclarées *transient*.

• Des méthodes de lecture/écriture peuvent être redéfinies le cas échéant:
private void writeObject (java.io.ObjectOutputStream out) throws IOException ;

private void readObject (java.io.ObjectInputStream in) throws IOException,ClassNotFoundException ;

107

Exemple Sérialisation/Désérialisation (1/4)

```
import java.io.* ;
class Info implements Serializable
{
        private String Nom = "" ;
        private String MotPasse = "" ;
        public Info(String n, String m)
        {
                Nom=n ; MotPasse = m ;
        }
        public String getNom () { return Nom ; }
        public String getPassword () { return MotPasse ; }
}
```

108

Exemple
Sérialisation/Désérialisation (2/4)

```
public class ExempleSerialisation
{
        static public void main (String args [])
        {
                new ExempleSerialisation () ;
        }
        public ExempleSerialisation ()
        {
                Info User = new Info ("Pierre","password") ;
                Ecrire (User) ;  //serialiser
                User = Lire () ; //deserialiser
                if (User != null)
                   System.out.println ("nom = " + User.getNom () +
                                " mot de passe = " + User.getPassword () ;
        }
// à suivre
```

109

Exemple
Sérialisation/Désérialisation (3/4)

```
        void Ecrire (Info user)  //serialisation
        {
                try {
                        FileOutputStream file = new
                        FileOutputStream ("c:\\travail\\info.txt") ;
                        ObjectOutputStream out = new
                        ObjectOutputStream (file) ;
                        out.writeObject (user) ;
                        out.flush () ;
                        out.close () ;
                } catch (IOException ex)
                {
                        System.err.println ("Erreur d'ecriture " + ex)
        ;
                }
        }
        //à suivre
```

110

Exemple
Sérialisation/Désérialisation (4/4)

```
Info Lire () // désérialiser
{
        Info User = null ;
        try {
        FileInputStream file = new FileInputStream
        ("c:\\travail\\info.txt") ;
        ObjectInputStream in = new ObjectInputStream (file);
        User = (Info) in.readObject() ;
        } catch (Exception ex)
        {
                System.err.println ("Erreur de lecture " + ex) ;
        }
        return User
}
} // Fin classe ExempleSerialisation
```

111

Ecriture:

```
objectOut = new ObjectOutputStream( new BufferedOutputStream( new FileOutputStream("d:\\info2.txt")));
objectOut.writeObject(obj1); // Write object
objectOut.writeObject(obj2); // Write object
objectOut.writeObject(obj3); // Write object
System.out.println("\n\nobj1:\n" + obj1 +"\n\nobj2:\n" + obj2 +"\n\nobj3:\n" + obj3);
```

boucle de lecture:

```
try {
   objectIn = new ObjectInputStream( new BufferedInputStream(new FileInputStream("d:\\info2.txt")));
   // Lire du début de stream jusu à la fin
   while(true) {
     object = (TypeObjet)objectIn.readObject();// lire object

    System.out.println(object);
   }
} catch(ClassNotFoundException e) {
   e.printStackTrace(System.err);
   System.exit(1);
} catch(EOFException e) { //  Fin de fichier
   System.out.println("EOF reached. "+ objectCount + " objects read.");
} catch(IOException e) { // This is for other I/O errors
   e.printStackTrace(System.err);
   System.exit(1);
}
```

112

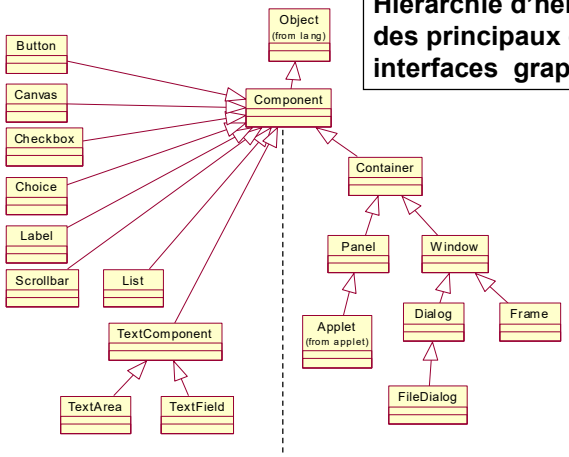
GUI: AWT & SWING

Conteneurs et composants (1)

- Une interface graphique en Java est un assemblage conteneurs (*Container*) et de **composants (*Component*).**
- **Un composant** est une sous-classes de la classe abstraite **java.awt.Component**.
  - Exemple : les boutons, les zones de textes ou de dessin, etc.
- **Un conteneur** est un espace dans lequel on peut positionner plusieurs composants.
  - Sous-classe de la classe **java.awt.Container**
  - Par exemple les fenêtres, les applets, etc.

114

Conteneurs et composants (2)



Hiérarchie d'héritage des principaux éléments des interfaces graphiques en Java

Les deux conteneurs les plus courants sont le Frame et le Panel.

115

Conteneurs et composants (5)

- On ajoute un composant dans un conteneur, avec la méthode add() :

  ```
  Panel p = new Panel();
  Button b = new Button();
  p.add(b);
  ```

- On retire

  ```
  p.remove(b);
  ```

- Un composant a (notamment) :
  - une taille préférée que l'on obtient avec getPreferredSize()
  - une taille minimum que l'on obtient avec getMinimunSize()
  - une taille maximum que l'on obtient avec getMaximunSize()

116

Conteneurs et composants (6)

```
import java.awt.*;

public class EssaiFenetre1
{
  public static void main(String[] args)
  {
    Frame f =new Frame("Ma première fenêtre");
    Button b= new Button("coucou");
    f.add(b);
    f.pack();
    f.setVisible(true);
  }
}
```

On demande à la fenêtre de choisir la taille minimum avec pack() et de se rendre visible avec setVisible(true)

117

Conteneurs et composants (7)

```
import javax.swing.*;

public class Fenetre {

   public static void main(String[] args) {

      JFrame window = new JFrame("Titre de la fenetre");
      int sizeX = 400;              // longueur en pixels
      int sizeY = 150;              // hauteur en pixels
      window.setBounds(50, 100, sizeX, sizeY);  // position et taille de la fenêtre
     //window.setSize(sizeX, sizeY);       // taille
    //window.setLocationRelativeTo(null);              // position au centre de l'écran

      JButton b= new JButton("coucou");
      window.add(b);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //pr la fermeture
de la fenêtre
      window.setVisible(true);
   }

}
```

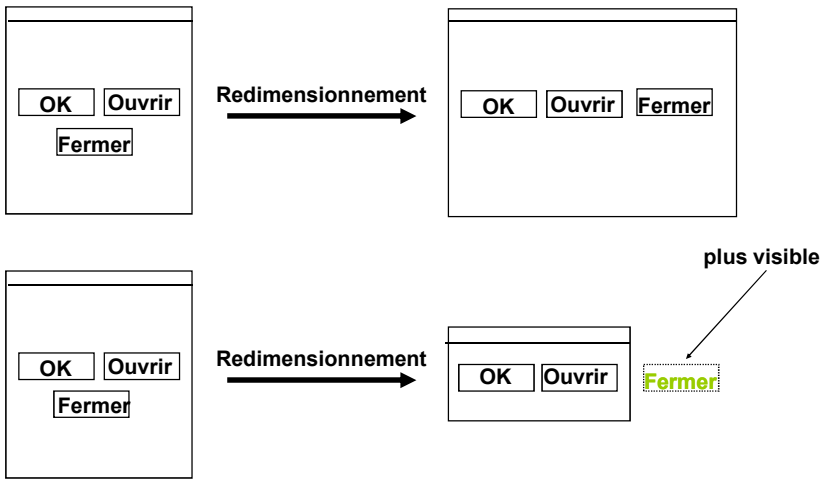
118

Gestionnaire de présentation (1)

- A chaque conteneur est associé un gestionnaire de présentation (*layout manager*)
- Le gestionnaire de présentation gère le positionnement et le (re)dimensionnement des composants d'un conteneur.
- Les principaux gestionnaire de présentation de l'AWT sont : FlowLayout, BorderLayout, GridLayout, CardLayout, GridBagLayout
- Tout instance de Container référence une instance de LayoutManager.
  - Il est possible d'en changer grâce à la méthode setLayout().
- Tout conteneur possède un gestionnaire de présentation par défaut:
  - Le BorderLayout pour Window et ses descendants (Frame, Dialog, …)
  - Le FlowLayout pour Panel et ses descendants (Applet, etc.)

119

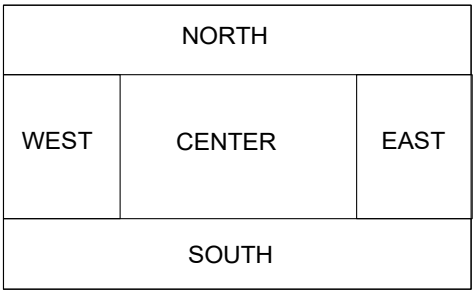
FlowLayout



120

BorderLayout

- Division de l'espace avec le BorderLayout



add("North", new Button("Le bouton nord !"));

121

BorderLayout

```
import java.awt.*;

public class EssaiBorderLayout extends Frame
{
  private Button b1,b2,b3,b4, b5;
   public EssaiBorderLayout() {
    setLayout(new BorderLayout());
    b1 = new Button ("Nord"); b2 = new Button ("Sud");
    b3 = new Button ("Est"); b4 = new Button ("Ouest");
    b5 = new Button ("Centre");
    this.add(b1, BorderLayout.NORTH);
    this.add(b2 , BorderLayout.SOUTH);
    this.add(b3, BorderLayout.EAST);
    this.add(b4, BorderLayout.WEST);
    this.add(b5, BorderLayout.CENTER);
  }
public static void main (String args []) {
    EssaiBorderLayout essai = new EssaiBorderLayout();
    essai.pack (); essai.setVisible(true) ;
  }}
```

122

GridLayout

```
import java.awt.*;
public class AppliGridLayout extends Frame
{
public AppliGridLayout()
 {
  super("AppliGridLayout");
  this.setLayout(new GridLayout(3,2));
  for (int i = 1; i < 7; i++)
    add(new Button(Integer.toString(i)));
  this.pack();
  this.setVisible(true);
 }

public static void main(String args[])
 {
   AppliGridLayout appli = new AppliGridLayout();
 }
}
```

123

On peut ne pas utiliser de gestionnaire de disposition et écrire :

setLayout(null);

On donne les coordonnées précises du composant dans le conteneur sous la forme (x,y,largeur,hauteur) où (x,y) sont les coordonnées du coin supérieur gauche du composant dans le conteneur.

```
conteneur=this.getContentPane();
conteneur.setLayout(null);
b=new JButton();
b.setText("Test");
// on fixe son emplacement et ses dimensions
b.setBounds(10,20,100,20);
conteneur.add(b);
```

124

```
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.awt.event.*;
public class MonForm extends JFrame {
        Container conteneur=null;
        JTextField t;
        JButton b;
        public MonForm() {
                this.setTitle("Formulaire avec bouton");
                this.setSize(new Dimension(500,500));
                WindowAdapter win=new WindowAdapter(){
                        public void windowClosing(WindowEvent e){System.exit(0);}
                };
                this.addWindowListener(win);
                conteneur=this.getContentPane();
                conteneur.setLayout(null);

                t=new JTextField("texte");
                conteneur.add(t);
                t.setBounds(30,50,100,40);
                b=new JButton();
                conteneur.add(b);
                b.setText("Test");
                b.setBounds(10,20,100,20);
                conteneur.add(b);
                b.addActionListener(new ActionListener()
                        {
                                public void actionPerformed(ActionEvent evt){
                                        t.setText("texte changé");
                                }
                        });
        }
        public static void main(String[] args) {
                new MonForm().setVisible(true);
        }
}
```

125

Récapitulatif

- FlowLayout
  - Flux : composants placés les uns derrière les autres
- BorderLayout
  - Ecran découpé en 5 zones (« North », « West », « South », « East », « Center »)
- GridLayout
  - Grille : une case par composant, chaque case de la même taille
- CardLayout
  - « Onglets » : on affiche un élément à la fois
- GridBagLayout
  - Grille complexe : plusieurs cases par composant

126

Les événements graphiques (1)

- **L'utilisateur effectue**
  - **une action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc)**
  - **alors un événement graphique est émis.**
- **Lorsqu'un événement se produit**
  - **il est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un champ de texte, etc.).**
  - **Ce composant transmet cet événement à un autre objet, un écouteur qui possède une méthode pour traiter l'événement (on parle de traitement d'événement)**

127

Les événements graphiques (2)

- La gestion des événements passe par l'utilisation d'objets "écouteur d'événements" (les *Listener*) et d'objets sources d'événements.
  - **Un objet écouteur est l'instance d'une classe implémentant l'interface *XXXXListener*.**
  - **Une source d'événements est un objet pouvant recenser des objets écouteurs et leur envoyer des objets événements.**

128

Les événements graphiques (3)

```
class MonAction implements ActionListener {
        public void actionPerformed (ActionEvent e)  {
                System.out.println ("Une action a eu lieu") ;}
}
public class TestBouton {
        public TestBouton(){
                JFrame f = new JFrame ("TestBouton");
                Button b = new Button ("Cliquer ici");
                f.setTitle("Evenements");
                f.setSize(500, 350);//f.setBounds(10, 10, 500, 350);
                Container conteneur=f.getContentPane();
                conteneur.setLayout(null);
                 b.setBounds(10, 20, 100, 50);
                conteneur.add(b);
                b.addActionListener (new MonAction ());
                f.add (b) ;   f.setVisible (true) ;
                b.addActionListener (new MonAction ());
                 f.addWindowListener(new WindowAction());

        }
        public static void main(String args[]) {
                TestBouton test = new TestBouton();}
}
```

129

Ajouter un ecouteur d' evennement wIndow

```
class WindowAction implements WindowListener {
   public void windowOpened(WindowEvent e){}
   public void windowClosing(WindowEvent e)
            {System.exit(0);}
   public void windowClosed(WindowEvent e){}
   public void windowIconified(WindowEvent e){}
   public void windowDeiconified(WindowEvent e){}
   public void windowActivated(WindowEvent e){}
   public void windowDeactivated(WindowEvent e){}
}
```

Il faut ajouter:

```
f.addWindowListener (new WindowAction());
```

Ou bien

```
WindowAdapter win=new WindowAdapter(){
                        public void windowClosing(WindowEvent
e){System.exit(0);}
                    };
f.addWindowListener(win);
```

Ou bien

```
 window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

130

Les événements graphiques (3)

```
// la classe elle même est un ecouteur de l'evennement
public class TestBouton  implements ActionListener{
        public TestBouton(){
                JFrame f = new JFrame ("TestBouton");
                Button b = new Button ("Cliquer ici");
                f.setTitle("Evenements");
                f.setSize(500, 350);//f.setBounds(10, 10, 500, 350);
                Container conteneur=f.getContentPane();
                conteneur.setLayout(null);
                 b.setBounds(10, 20, 100, 50);
                 conteneur.add(b);
                f.setVisible(true);
                f.add (b) ;   f.setVisible (true) ;
                b.addActionListener (this);
                public void actionPerformed (ActionEvent e)  {
                        System.out.println ("Une action a eu lieu") ;}
                f.addWindowListener(new WindowAction());
                }
        public static void main(String args[]) {
                TestBouton test = new TestBouton();}
}
```

131

```
import java.awt.*;
import java.awt.event.*;
// Ajouter un ecouteur ActionListener dans la classe
public class TestBouton {
        public TestBouton(){
                JFrame f = newJ Frame ("TestBouton");
                JButton b = new JButton ("Cliquer ici");
                …
                f.add (b) ;
                f.pack (); f.setVisible (true) ;

                  ActionListener act = new ActionListener (){
                        public void actionPerformed (ActionEvent e)  {
                                System.out.println ("Une action a eu lieu") ;}
                        };
                b.addActionListener ( act);
            }
    public static void main(String args[]) {
                TestBouton test = new TestBouton();}
}
```

132

```
import java.awt.event.*;
import javax.swing.*;
class MonAction implements ActionListener {
        public void actionPerformed (ActionEvent e)  {
                System.out.println ("Une action a eu lieu") ;}
}

public class Graphique extends JFrame{
  //private JPanel pan = new JPanel();
  private JButton b = new JButton("Mon bouton");

  public Graphique(){
    this.setTitle("Evenements");
    this.setSize(500, 350);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    this.add(b);
    b.addActionListener (new MonAction ());
    this.setVisible(true);
  }

    public static void main(String args[]) {
          Graphique ex = new Graphique();
     }
}
```

133

Les événements graphiques (4)

- Les écouteurs sont des interfaces
- Donc une même classe peut implémenter plusieurs interfaces écouteur.
  - **Par exemple une classe héritant de Frame implémentera les interfaces `MouseMotionListener` (pour les déplacements souris) et `MouseListener` (pour les clics souris).**
- Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers.
  - **Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée addXXXListener().**

134

Catégories d'événements graphiques (1)

Catégorie	Nom de l'interface	Méthodes
Action	ActionListener	actionPerformed (ActionEvent)
Item	ItemListener	itemStateChanged (ItemEvent)
Mouse	MouseMotionListener	mouseDragged (MouseEvent) mouseMoved (MouseEvent)
Mouse	MouseListener	mousePressed (MouseEvent) mouseReleased (MouseEvent) mouseEntered (MouseEvent) (MouseEvent) mouseExited mouseClicked
Key	KeyListener	keyPressed (KeyEvent) keyReleased (KeyEvent) keyTyped (KeyEvent)
Focus	FocusListener	focusGained (FocusEvent) focusLost (FocusEvent)

135

Catégories d'événements graphiques (3)

Adjustment	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
Component	ComponentListener	componentMoved (ComponentEvent) componentHiddent (ComponentEvent) componentResize (ComponentEvent) componentShown (ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

136

Catégories d'événements graphiques (4)

- ActionListener
  - **Action (clic) sur un bouton, retour chariot dans une zone de texte, « tic d'horloge » (Objet Timer)**
- WindowListener
  - **Fermeture, iconisation, etc. des fenêtres**
- TextListener
  - **Changement de valeur dans une zone de texte**
- ItemListener
  - **Sélection d'un item dans une liste**
- MouseListener
  - **Clic, enfoncement/relâchement des boutons de la souris, etc.**
- MouseMotionListener
  - **Déplacement de la souris, drag&drop avec la souris, etc.**

137

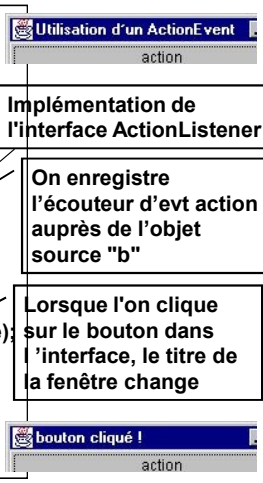
Catégories d'événements graphiques (5)

- AdjustmentListener
  - **Déplacement d'une échelle**
- ComponentListener
  - **Savoir si un composant a été caché, affiché . . .**
- ContainerListener
  - **Ajout d'un composant dans un Container**
- FocusListener
  - **Pour savoir si un élément a le "focus"**
- KeyListener
  - **Pour la gestion des événements clavier**

138

Catégories d'événements graphiques (6)

```
import java.awt.*;
import java.awt.event.*;
public class EssaiActionEvent1 extends Frame
                    implements ActionListener
{
  public static void main(String args[])
  {EssaiActionEvent1 f= new EssaiActionEvent1();}
  public EssaiActionEvent1()
  {
   super("Utilisation d'un ActionEvent");
   Button b = new Button("action");
   b.addActionListener(this);
   add(BorderLayout.CENTER,b);pack();setVisible(true);
  }
  public void actionPerformed( ActionEvent e )
  {
   setTitle("bouton cliqué !");
 }}
```



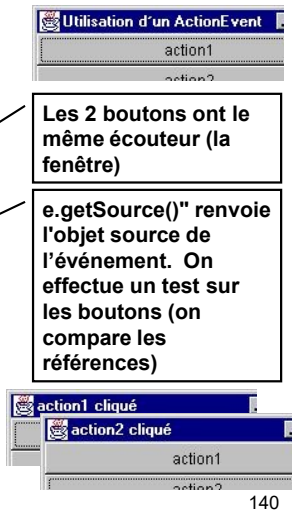
Implémentation de l'interface ActionListener

On enregistre l'écouteur d'evt action auprès de l'objet source "b"

Lorsque l'on clique sur le bouton dans l 'interface, le titre de la fenêtre change

Catégories d'événements graphiques (7)

```
public class EssaiActionEvent2 extends Frame
                    implements ActionListener
{ private Button b1,b2;
  public static void main(String args[])
  {EssaiActionEvent2 f= new EssaiActionEvent2();}
  public EssaiActionEvent2(){
   super("Utilisation d'un ActionEvent");
   b1 = new Button("action1");
   b2 = new Button("action2");
   b1.addActionListener(this);
   b2.addActionListener(this);
   add(BorderLayout.CENTER,b1);
   add(BorderLayout.SOUTH,b2);
   pack();setVisible(true); }
  public void actionPerformed( ActionEvent e ) {
   if (e.getSource() == b1) setTitle("action1 cliqué");
   if (e.getSource() == b2) setTitle("action2 cliqué");
  }}
```



Les 2 boutons ont le même écouteur (la fenêtre)

e.getSource()" renvoie l'objet source de l'événement. On effectue un test sur les boutons (on compare les références)

Catégories d'événements graphiques (9)

```
import java.awt.*; import java.awt.event.*;
public class WinEvt extends Frame
      implements WindowListener
{
 public static void main(String[] args) {
   WinEvt f= new WinEvt();}
   public WinEvt() {
     super("Cette fenêtre se ferme");
     addWindowListener(this);
     pack();setVisible(true);}
    public void windowOpened(WindowEvent e){}
    public void windowClosing(WindowEvent e)
    {System.exit(0);}
    public void windowClosed(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){} }
```

Implémenter cette interface impose l'implémentation de bcp de méthodes

La fenêtre est son propre écouteur

WindowClosing() est appelé lorsque l'on clique sur la croix de la fenêtre

"System.exit(0)" permet de quitter une application java

141