Cours programmation-orientée objet en Java

Licence d' informatique

Hugues Fauconnier

hf@liafa.jussieu.fr

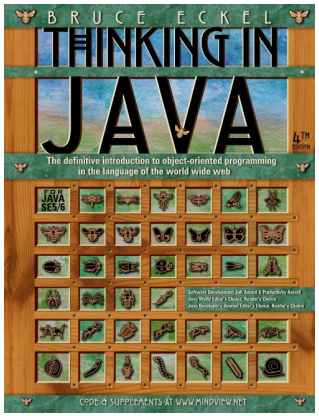
Plan du cours

- Introduction:
  - programmation objet pourquoi? Comment? Un exemple en Java
- Classes et objets (révision)
  - Méthodes et variables, constructeurs, contrôle d' accès, constructeurs
- Héritage: liaison dynamique et typage
  - Extension de classe, méthode et héritage, variables et héritage, constructeurs et héritage
- Héritage: compléments
  - classes abstraites et interfaces, classes internes et emboîtées, classe Object, clonage,
- Introduction à Swing
- Exceptions
  - Exceptions, assertions
- Divers: Noms, conversions, héritage et tableaux
- Généricité
  - Généralités, types génériques imbriqués, types paramètres bornés, méthodes génériques
- Types de données
  - String et expressions régulières, Collections, Conteneurs, itérations
- Entrées-sorties
- Threads
- Compléments
  - Reflections, annotations, documentation…

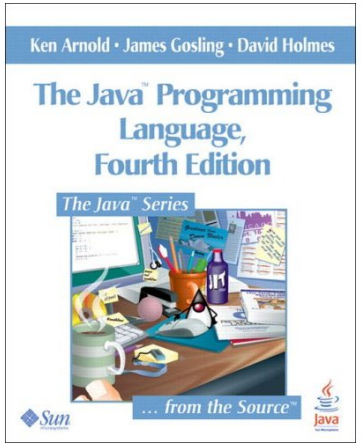
Le site du cours: http://www.liafa.jussieu.fr/~hf/verif/ens/an11-12/poo/L3.POO.html
Didel POO

Bibliographie

- De nombreux livres sur java (attention java >= 1.5)
- En ligne:
  - http://mindview.net/Books/TIJ4
  - Thinking in Java, 4th edition Bruce Eckel
  - http://java.sun.com/docs/index.html



- Livre conseillé:
  - The Java Programming language fourth edition AW Ken Arnold, James Gosling, David Holmes



Chapitre I

Introduction

A) Généralités

- Problème du logiciel:
  - Taille
  - Coût : développement et maintenance
  - Fiabilité
- Solutions :
  - Modularité
  - **Réutiliser le logiciel**
  - Certification

- Comment?

Typage…

- Histoire:
  - Fonctions et procédures (60 Fortran)
  - Typage des données (70) Pascal Algol
  - Modules: données + fonctions regroupées (80) ada
  - Programmation objet: classes, objets et héritage

B) Principes de base de la POO

- Objet et classe:
  - Classe = définitions pour des données (variables) + fonctions (méthodes) agissant sur ces données
  - Objet = élément d'une classe (instance) avec un état
  - (une méthode ou une variable peut être
    - de classe = commune à la classe ou
    - d'instance = dépendant de l'instance

)

Principes de bases (suite)

- Encapsulation et séparation de la spécification et de l'implémentation
  - Séparer l'implémentation de la spécification.
    - Ne doit être visible de l'extérieur que ce qui est nécessaire, les détails d'implémentation sont « cachés »
- Héritage:
  - Une classe peut hériter des propriétés d'une autre classe: un classe peut être une extension d'une autre classe.

Principes de bases de la POO

- Mais surtout notion de *polymorphisme*:
  - Si une classe A est une extension d'une classe B:
    - A doit pouvoir *redéfinir* certaines méthodes (disons f())
    - Un objet a de classe A doit pouvoir être considéré comme un objet de classe B
    - On doit donc accepter :
      - B b;
      - b=a; (a a toutes les propriétés d'un B)
      - b.f()
        - Doit appeler la méthode redéfinie dans A!
      - C'est le *transtypage*
    - (exemple: méthode paint des interfaces graphiques)

Principes de bases

- Polymorphisme:
  - Ici l'association entre le nom 'f()' et le code (code de A ou code de B) a lieu dynamiquement (=à l'exécution)

    Liaison dynamique
  - On peut aussi vouloir « paramétrer » une classe (ou une méthode) par une autre classe.

    Exemple: Pile *d'entiers*

    Dans ce cas aussi un nom peut correspondre à plusieurs codes, mais ici l'association peut avoir lieu de façon statique (au moment de la compilation)

C) Comment assurer la réutilisation du logiciel?

- Type abstrait de données
  - définir le type par ses propriétés (spécification)
- Interface, spécification et implémentation
  - Une interface et une spécification (=les propriétés à assurer) pour définir un type
  - Une (ou plusieurs) implémentation du type abstrait de données
    - Ces implémentations doivent vérifier la spécification

Comment assurer la réutilisation du logiciel?

- Pour l'utilisateur du type abstrait de données
  - Accès uniquement à l'interface (pas d'accès à l'implémentation)
  - Utilisation des propriétés du type abstrait telles que définies dans la spécification.
  - (L'utilisateur est lui-même un type abstrait avec une interface et une spécification)

Comment assurer la réutilisation du logiciel?

- Mais en utilisant un type *abstrait* l'utilisateur n'en connaît pas l'implémentation
  - il sait uniquement que la spécification du type abstrait est supposée être vérifiée par l'implémentation.
- Pour la réalisation *concrète*, une implémentation particulière est choisie
- Il y a naturellement polymorphisme

Notion de contrat (Eiffel)

- Un *client* et un *vendeur*
- Un *contrat* lie le vendeur et le client (*spécification*)
- Le client ne peut utiliser l'objet que par son *interface*
- La réalisation de l'objet est cachée au client
- Le contrat est conditionné par l'utilisation correcte de l'objet (*pré-condition*)
- Sous réserve de la pré-condition le vendeur s'engage à ce que l'objet vérifie sa spécification (*post-condition*)
- Le vendeur peut déléguer: l'objet délégué doit vérifier au moins le contrat (*héritage*)

D) Un exemple…

- Pile abstraite et diverses implémentations

Type abstrait de données

NOM
  pile[X]
FONCTIONS
  vide : pile[X] -> Boolean
  nouvelle : -> pile[X]
  empiler : X x pile[X] -> pile[X]
  dépiler : pile[X] -> X x pile[X]
PRECONDITIONS
  dépiler(s: pile[X]) <=> (not vide(s))
AXIOMES
  forall x in X, s in pile[X]
    vide(nouvelle())
    not vide(empiler(x,s))
    dépiler(empiler(x,s))=(x,s)

Remarques

- Le type est paramétré par un autre type
- Les axiomes correspondent aux pré-conditions
- Il n'y pas de représentation
- Il faudrait vérifier que cette définition caractérise bien un pile au sens usuel du terme (c'est possible)

Pile abstraite en java

```
package pile;

abstract class Pile <T>{
    abstract public T empiler(T v);
    abstract public T dépiler();
    abstract public Boolean estVide();
}
```

Divers

- `package`: regroupement de diverses classes
- `abstract`: signifie qu'il n'y a pas d'implémentation
- `public`: accessible de l'extérieur
- La classe est paramétrée par un type (java 1.5)

Implémentations

- On va implémenter la pile:
  - avec un objet de classe `Vector` (classe définie dans `java.util.package`) en fait il s'agit d'un `ArrayList`
  - Avec un objet de classe `LinkedList`
  - Avec `Integer` pour obtenir une pile de `Integer`

Une implémentation

```
package pile;
import java.util.EmptyStackException;
import java.util.Vector;
public class MaPile<T> extends Pile<T>{
    private Vector<T> items;
    // Vector devrait être remplacé par ArrayList
    public MaPile() {
        items =new Vector<T>(10);
    }
    public Boolean estVide(){
        return items.size()==0;
    }
    public T empiler(T item){
        items.addElement(item);
        return item;
    }
    //...
```

Suite

```
//…
public synchronized T dépiler(){
        int len = items.size();
        T item = null;
        if (len == 0)
            throw new EmptyStackException();
        item = items.elementAt(len - 1);
        items.removeElementAt(len - 1);
        return item;
    }
}
```

Autre implémentation avec listes

```
package pile;
import java.util.LinkedList;
public class SaPile<T> extends Pile<T> {
    private LinkedList<T> items;
    public SaPile(){
        items = new LinkedList<T>();
    }
    public Boolean estVide(){
        return items.isEmpty();
    }
    public T empiler(T item){
        items.addFirst(item);
        return item;
    }
    public T dépiler(){
        return items.removeFirst();
    }
}
```

Une pile de Integer

```
public class PileInteger extends Pile<Integer>{
    private Integer[] items;
    private int top=0;
    private int max=100;
    public PileInteger(){
        items = new Integer[max];
    }
    public Integer empiler(Integer item){
        if (this.estPleine())
            throw new EmptyStackException();
        items[top++] = item;
        return item;
    }
    //…
```

Suite…

```
public synchronized Integer dépiler(){
        Integer item = null;
        if (this.estVide())
            throw new EmptyStackException();
        item = items[--top];
        return item;
    }
    public Boolean estVide(){
        return (top == 0);
    }
    public boolean estPleine(){
        return (top == max -1);
    }
    protected void finalize() throws Throwable {
        items = null; super.finalize();
    }
}
```

Comment utiliser ces classes?

- Le but est de pouvoir écrire du code utilisant la classe Pile abstraite
- Au moment de l'exécution, bien sûr, ce code s'appliquera à un objet concret (qui a une implémentation)
- Mais ce code doit s'appliquer à toute implémentation de Pile

Un main

```
package pile;
public class Main {
    public static void vider(Pile p){
        while(!p.estVide()){
            System.out.println(p.dépiler());
        }
    }
public static void main(String[] args) {
        MaPile<Integer> p1= new MaPile<Integer>();
        for(int i=0;i<10;i++)
            p1.empiler(i);
        vider(p1);
        SaPile<String> p2= new SaPile<String>();
        p2.empiler("un");
        p2.empiler("deux");
        p2.empiler("trois");
        vider(p2);
    }
}
```

E) java: quelques rappels…

- Un source avec le suffixe `.java`
- Une classe par fichier source (en principe) même nom pour la classe et le fichier source (sans le suffixe `.java`)
- Méthode

```
public static void main(String[]);
```

  - `main` est le point d'entrée
- Compilation génère un `.class`
- Exécution en lançant la machine java

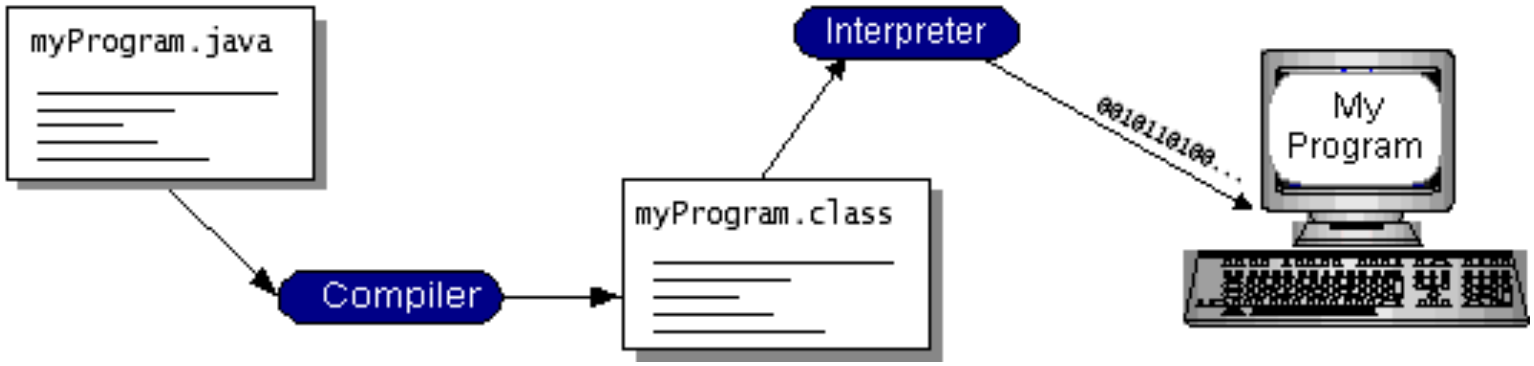
Généralités…

- Un peu plus qu' un langage de programmation:
  - "gratuit"! (licence GPL)
  - Indépendant de la plateforme
  - *Langage interprété et byte code*
  - Syntaxe à la *C*
  - Orienté objet (classes héritage)
    - Nombreuses bibliothèques
  - Pas de pointeurs! (ou que des pointeurs!)
    - Ramasse-miettes
  - Multi-thread
  - Distribué (WEB) applet, servlet, …
  - Dernière version Java SE 7 (GPL)
  - Site:http://www.java.com/fr

Plateforme Java

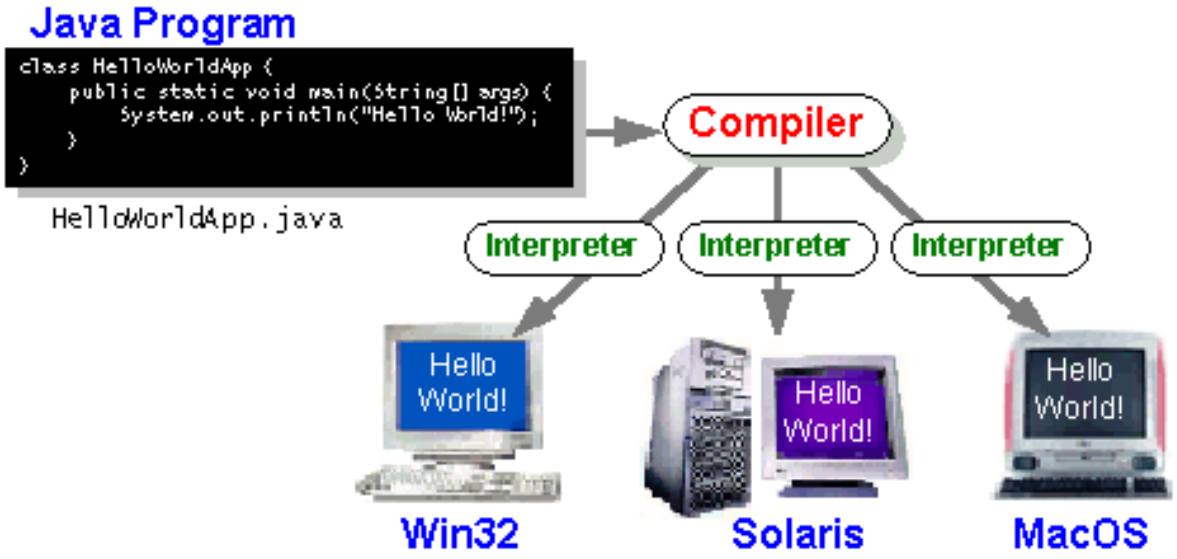
- La compilation génère un .class en bytecode (langage intermédiaire indépendant de la plateforme).
- Le bytecode est interprété par un interpréteur Java JVM

Compilation `javac`
interprétation `java`



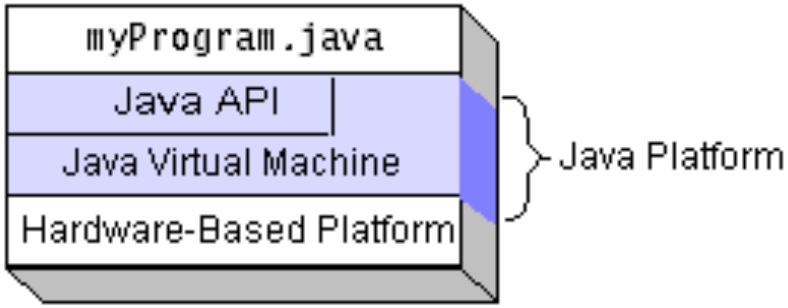
Langage intermédiaire et Interpréteur…

- Avantage: indépendance de la plateforme
  - Échange de byte-code (applet)
- Inconvénient: efficacité



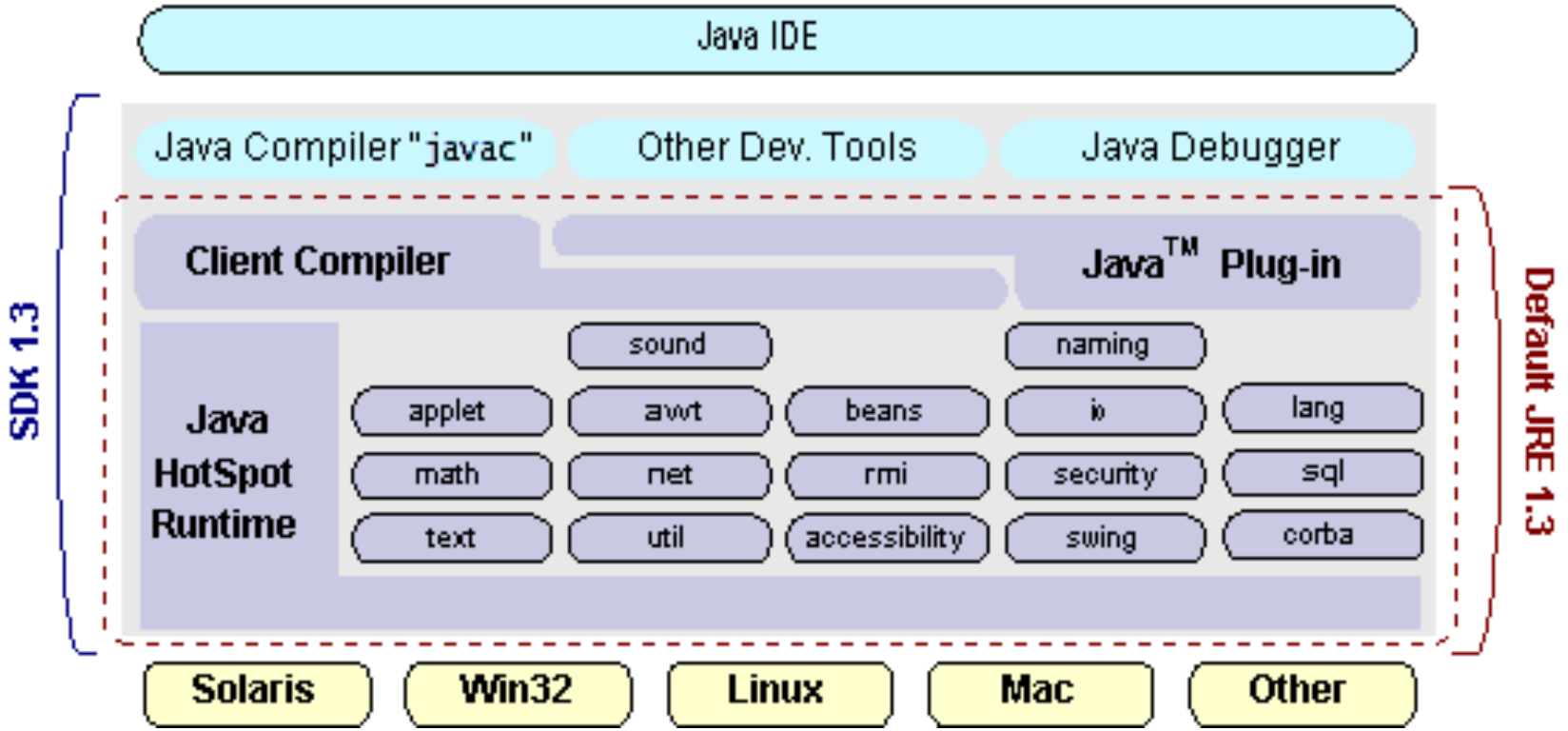
Plateforme Java

- La plateforme java: software au-dessus d'une plateforme exécutable sur un hardware (exemple MacOs, linux …)
- Java VM
- Java application Programming Interface (Java API):

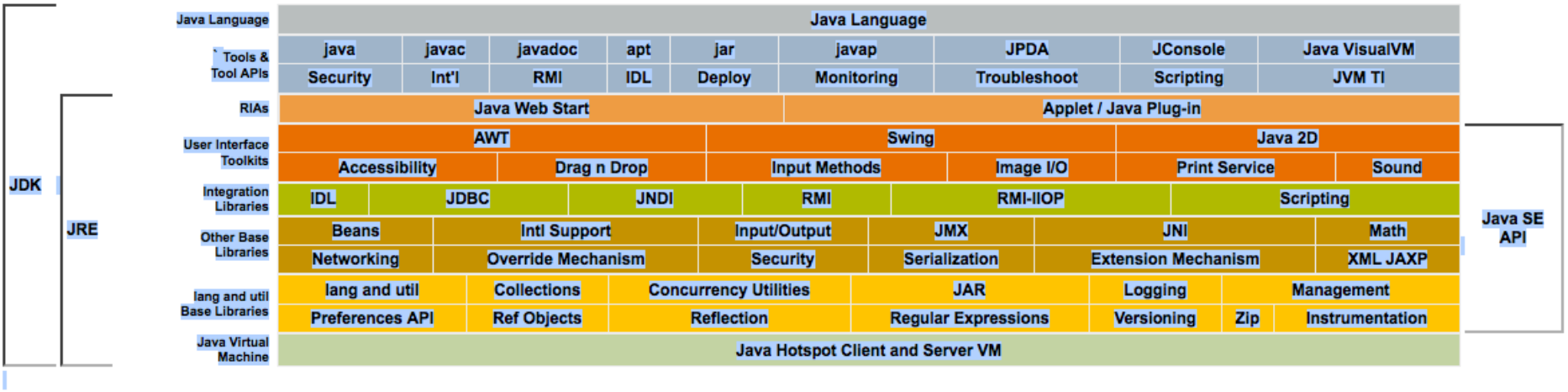


Tout un environnement…

- Java 2 sdk: JRE (java runtime environment + outils de développements compilateur, debogueurs etc…)



Tout un environnement…



Java SE 6 API Documentation

Trois exemples de base

- Une application
- Une applet
- Une application avec interface graphique

Application:

- Fichier Appli.java:

```
/**
 * Une application  basique...
 */
class Appli {
    public static void main(String[] args) {
     System.out.println("Bienvenue en L3...");
                //affichage
    }
}
```

Compiler, exécuter…

- Créer un fichier `Appli.java`
- Compilation:
  - `javac Appli.java`
- Création de `Appli.class` (bytecode)
- Interpréter le byte code:
  - `java Appli`
- Attention aux suffixes!!!
  - (il faut que javac et java soient dans $PATH)

```
Exception in thread "main" java.lang.NoClassDefFoundError:
```

  - Il ne trouve pas le main -> vérifier le nom!
  - Variable CLASSPATH ou option -classpath

Remarques

- Commentaires /* … */ et //
- Définition de classe
  - une classe contient des méthodes (=fonctions) et des variables
  - Pas de fonctions ou de variables globales (uniquement dans des classes ou des instances)
- Méthode `main`:
  - `public static void main(String[] arg)`
    - `public`
    - `static`
    - `Void`
    - `String`
  - Point d'entrée

Remarques

- Classe `System`
  - `out` est une variable de la classe `System`
  - `println` méthode de `System.out`
  - `out` est une variable de classe qui fait référence à une instance de la classe `PrintStream` qui implémente un flot de sortie.
    - Cette instance a une méthode `println`

Remarques…

- Classe: définit des méthodes et des variables (déclaration)
- Instance d'une classe (objet)
  - Méthode de classe: fonction associée à (toute la) classe.
  - Méthode d'instance: fonction associée à une instance particulière.
  - Variable de classe: associée à une classe (globale et partagée par toutes les instances)
  - Variable d'instance: associée à un objet (instancié)
- Patience…

Applet:

- Applet et WEB
  - Client (navigateur) et serveur WEB
  - Le client fait des requêtes html, le serveur répond par des pages html
  - Applet:
    - Le serveur répond par une page contenant des applets
    - Applet: byte code
    - Code exécuté par le client
    - Permet de faire des animations avec interfaces graphiques sur le client.
    - Une des causes du succès de java.

Exemple applet

- Fichier MonApplet.java:

```
/**
 * Une applet  basique...
 */
import java.applet.Applet;
import java.awt.Graphics;
public class MonApplet extends Applet {
    public void paint(Graphics g){
   g.drawString("Bienvenue en en L3...", 50,25);
    }
}
```

Remarques:

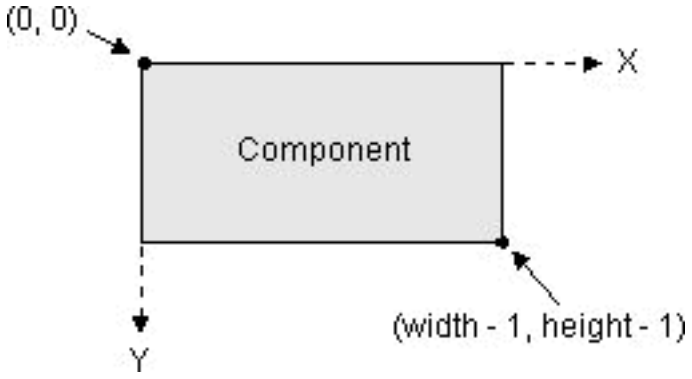
- import et package:
  - Un package est un regroupement de classes.
  - Toute classe est dans un package
  - Package par défaut (sans nom)
  - classpath
- import java.applet.*;
  - Importe le package java.applet
    - Applet est une classe de ce package,
    - Sans importation il faudrait java.applet.Applet

Remarques:

- La classe Applet contient ce qu'il faut pour écrire une applet
- … extends Applet:
  - La classe définie est une extension de la classe Applet:
    - Elle contient tout ce que contient la classe Applet
    - (et peut redéfinir certaines méthodes (paint))
  - Patience!!

Remarques…

- Une Applet contient les méthodes `paint` `start` et `init`. En redéfinissant `paint`, l'applet une fois lancée exécutera ce code redéfini.
- `Graphics g` argument de `paint` est un objet qui représente le contexte graphique de l'applet.
  - `drawString` est une méthode (d'instance) qui affiche une chaîne,
  - 50, 25: affichage à partir de la position (x,y) à partir du point (0,0) coin en haut à gauche de l'applet.



Pour exécuter l'applet

- L'applet doit être exécutée dans un navigateur capable d'interpréter du bytecode correspondant à des applet.
- Il faut créer un fichier HTML pour le navigateur.

Html pour l'applet

- Fichier Bienvenu.html:

```
<HTML>
<HEAD>
<TITLE> Une petite applet </TITLE>
<BODY>
<APPLET CODE='MonApplet.class' WIDTH=200
   Height=50>
</APPLET>
</BODY>
</HTML>
```

Html

- Structure avec balises:
- Exemples:
  - <HTML> </HTML>
  - url:
    - page de hf
- Ici:

```
<APPLET CODE='MonApplet.class' WIDTH=200
   Height=50>
</APPLET>
```

Exemple interface graphique

Fichier MonSwing.java:

```
/**
 * Une application  basique... avec interface graphique
 */
import javax.swing.*;
public class MonSwing {
    private static void creerFrame() {
        //Une formule magique...
        JFrame.setDefaultLookAndFeelDecorated(true);
        //Creation d'une Frame
        JFrame frame = new JFrame("MonSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         //Afficher un message
        JLabel label = new JLabel("Bienvenue en L3...");
        frame.getContentPane().add(label);
        //Afficher la fenêtre
        frame.pack();
        frame.setVisible(true);
    }
 public static void main(String[] args) {
     creerFrame();
    }
}
```

Remarques

- Importation de packages
- Définition d'un conteneur top-level JFrame, implémenté comme instance de la classe JFrame
- Affichage de ce conteneur
- Définition d'un composant JLabel, implémenté comme instance de JLabel
- Ajout du composant JLabel dans la JFrame
- Définition du comportement de la Jframe sur un click du bouton de fremeture
- Une méthode main qui crée la JFrame

Pour finir…

- Java 1.5 et 6 annotations, types méthodes paramétrés par des types
- Très nombreux packages
- Nombreux outils de développement (gratuits)
  - eclipse, netbeans..

En plus…

- Entrées-sorties

Entrée-sortie

```
public static void main(String[] args) {
     // sortie avec printf ou
     double a = 5.6d ;
     double b = 2d ;
     String mul = "multiplié par" ;
     String eq="égal";
     System.out.printf(Locale.ENGLISH,
                "%3.2f X %3.2f = %6.4f \n", a ,b , a*b);
     System.out.printf(Locale.FRENCH,
          "%3.2f %s %3.2f %s %6.4f \n", a, mul,b, eq,a*b);
      System.out.format(
         "Aujourd'hui %1$tA, %1$te %1$tB,"+
          " il est: %1$tH h %1$tM min %1$tS \n",
                Calendar.getInstance());
       // System.out.flush();
```

Sortie

5.60 X 2.00 = 11.2000

5,60 multiplié par 2,00 égal 11,2000

Aujourd'hui mardi, 10 octobre, il est: 15 h 31 min 01

Scanner

```
Scanner sc = new Scanner(System.in);
for(boolean fait=false; fait==false;){
    try {
       System.out.println("Répondre o ou O:");
       String s1 =sc.next(Pattern.compile("[Oo]"));
              fait=true;
    } catch(InputMismatchException e) {
    sc.next();
    }
}
if (sc.hasNextInt()){
    int i= sc.nextInt();
    System.out.println("entier lu "+i);
}
System.out.println("next token :"+sc.next());
sc.close();
```

Scanner

```
if (sc.hasNextInt()){
    int i= sc.nextInt();
    System.out.println("entier lu "+i);
}
System.out.println("next token :"+sc.next()); sc.close();
String input = "1 stop 2 stop éléphant gris stop rien";
Scanner s = new(Scanner(input).useDelimiter("\\s*stop\\s*");
        System.out.println(s.nextInt());
        System.out.println(s.nextInt());
        System.out.println(s.next());
        System.out.println(s.next());
        s.close();
    }
```

Sortie

- next token :o
- 1
- 2
- éléphant gris
- rien

Les classes…

- System
  - System.out variable (static) de classe PrintStream
    - PrintStream contient print (et printf)
  - System.in variable (static) de classe InputStream
- Scanner

Chapitre II
Classes et objets (rappels)

(mais pas d' héritage)

Classes et objets

- I) Introduction
- II) Classe: membres et modificateurs
- III) Champs: modificateurs
- IV) Vie et mort des objets, Constructeurs
- V) Méthodes
- VI) Exemple

I) Introduction

- Classe
  - Regrouper des données et des méthodes
    - Variables de classe
    - Méthodes de classe
  - Classes<->type
- Objet (ou instance)
  - Résultat de la création d'un objet
    - Variables d'instance
    - Variables de classe
- Toute classe hérite de la classe `Object`

II) Classes

- Membres d ' une classe sont:
  - Champs = données
  - Méthodes = fonctions
  - Classes imbriquées

Modificateur de classe

- Précède la déclaration de la classe
  - Annotations (plus tard…)
  - `public` (par défaut package)
  - `abstract`(incomplète, pas d'instance)
  - `final`(pas d'extension)
  - `Strictfp` (technique…)

III) Champs

- Modificateurs
  - annotations
  - Contrôle d'accès
    - `private`
    - `protected`
    - `public`
    - `package`
  - `static` (variables de classe)
  - `final` (constantes)
  - `transient`
  - `volatile`
- Initialisations
- Création par opérateur new

IV) Vie et mort des objets, constructeurs

- Création d'une instance: opérateur new
- Objet mort = plus aucune référence à cet objet -> garbage collector
  - on peut exécuter du code spécifique quand un objet est détruit :
    `protected void finalize() throws Throwable`

Références

- Une variable est (en général) une référence à un objet
  - Type primitif: directement une valeur
  - Type référence : une référence à un objet (existant ou créé par new)
    - null : référence universelle
    - conséquences:
      - dans le passage par valeur un type référence correspond à un passage par référence
      - 'a == b ' teste si les a et b référencent le *même* objet
      - Méthode `equals` qui peut être redéfinie (défaut this==obj)

Exemple

```
int i=0;
int j=0;
(i==j) // vrai
class A{
    int i=0;
}
A a;
A b=new A();
a=b;
(a==b) // vrai
b=new A();
(a==b) // faux
```

Constructeurs

- Appelés par l'opérateur new pour créer un objet
  - Peuvent avoir des paramètres (avec surcharge)
  - Initialisent les objets
  - Constructeur par défaut (si aucun constructeur n'est défini)
  - Constructeur de copie

Exemple:

```
public class Astre {
    private long idNum;
    private String nom = "<pasdenom>";
    private Astre orbite = null;
    private static long nextId = 0;
    /** Creation d'une nouvelle instance of Astre */
    private Astre() {
        idNum = nextId ++;
    }
    public Astre(String nom, Astre enOrbite){
        this();
        this.nom=nom;
        orbite=enOrbite;
    }
    public Astre(String nom){
        this(nom,null);
    }//…
```

Exemples…

- Copie

```
public Astre(Astre a){
        idNum = a.idNum;
        nom=a.nom;
        orbite=a.orbite;
    }
```

Statique - dynamique

- Statique <-> à la compilation
- Dynamique <-> à l'exécution
- Le type d'une variable est déterminé à la compilation (déclaration et portée)
- Avec la possibilité de l'héritage une variable peut être une référence sur un objet d'un autre type que le type de sa déclaration

Static

- Une variable (une méthode) déclarée static est une variable (méthode) de classe: elle est associée à la classe (pas à une instance particulière).
- Statique parce qu'elle peut être créée au moment de la compilation (pas de new()).
- Statique -> les initialisations doivent avoir lieu à la compilation.

Initialisations

```
private static long nextId = 0;
```

- Bloc d'initialisation

```
private static long netxId = 0;
{
        idNum = nextId++;
 }
```

Initialisation static

```
public class Puissancedeux {
    static int[] tab = new int[12];
    static{
        tab[0]=1;
        for(int i=0; i< tab.length-1;i++)
            tab[i+1]= suivant(tab[i]);
         }
    static int suivant(int i){
        return i*2;
    }
}
```

V) Méthodes

- Modificateurs:
  - Annotations
  - Contrôle d'accès (comme pour les variables)
  - `abstract`
  - `static` n'a pas accès aux variables d'instances
  - `final` ne peut pas être remplacée
  - `synchronized`
  - `native` (utilisation de fonctions « native »)
  - `strictfp`

Passage par valeur

```
public class ParamParVal {
    public static void parVal(int i){
        i=0;
        System.out.println("dans parVal i="+i);
    }
}
//…
 int i =100;
 System.out.println("Avant i="+i);
 ParamParVal.parVal(i);
 System.out.println("Avant i="+i);
---------------
Avant i=100
dans parVal i=0
Avant i=100
```

Mais…

- Comme les variables sont de références (sauf les types primitifs)…

```
public static void bidon(Astre a){
    a=new Astre("bidon", null);
    System.out.println("bidon a="+a);
}
 public static void bidonbis(Astre a){
     a.setNom("bidon");
     a.setOrbite(null);
    System.out.println("bidonbis a="+a);
}
```

Méthodes…

- Contrôler l'accès:

```
//…
    public void setNom(String n){
        nom=n;
    }
    public void setOrbite(Astre a){
        orbite=a;
    }
    public String getNom(){
        return nom;
    }
    public Astre getOrbite(){
        return orbite;
    }
```

Méthodes, remplacement…

```
public String toString(){
    String st=idNum + "("+nom+")";
    if (orbite != null)
        st += "en orbite "+ orbite;
    return st;
}
```

Remplace la méthode toString de la classe Object

Nombre variable d'arguments…

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

//…
affiche("un", "deux","trois");
```

Méthodes main

```
 public static void main(String[] args) {
        for(int j =0; j<args.length;j++){
            System.out.print(args[j] + " ");
        }
}
```

Le main est le point d'accès et peut avoir des arguments:

VI) exemple: Les astres…

```
package exempleclasses;

/**
 *
 * @author sans
 */
public class Astre {
    private long idNum;
    private String nom = "<pasdenom>";
    private Astre orbite = null;
    private static long nextId = 0;
    /** Creates a new instance of Astre */
    private Astre() {
        idNum = nextId ++;
    }
```

Suite

```
public Astre(String nom, Astre enOrbite){
     this();
     this.nom=nom;
     orbite=enOrbite;
 }
 public Astre(String nom){
     this(nom,null);
 }
 public Astre(Astre a){
     idNum = a.idNum;
     nom=a.nom;
     orbite=a.orbite;
 }//…
```

```
    public void setNom(String n){
        nom=n;
    }
    public void setOrbite(Astre a){
        orbite=a;
    }
    public String getNom(){
        return nom;
    }
    public Astre getOrbite(){
        return orbite;
    }
    public String toString(){
        String st=idNum + "("+nom+")";
        if (orbite != null)
            st += "en orbite "+ orbite;
        return st;
    }
}
```

Chapitre III
Héritage

Chapitre III: Héritage

- A) Extensions généralités
  - Affectation et transtypage
- B) Méthodes
  - Surcharge et signature
- C) Méthodes (suite)
  - Redéfinition et liaison dynamique
- D) Conséquences
  - Les variables
- E) Divers
  - Super, accès, final
- F) Constructeurs et héritage

A) Extension: généralités

- Principe de la programmation objet:
  - un berger allemand est un chien
    - il a donc toutes les caractéristiques des chiens
    - il peut avoir des propriétés supplémentaires
    - un chien est lui-même un mammifère qui est lui-même un animal: hiérarchie des classes
  - On en déduit:
    - Hiérarchie des classes (Object à la racine)
    - et si B est une extension de A alors un objet de B est un objet de A avec des propriétés supplémentaires

Extension: généralités

Quand B est une extension de la classe A:

- Tout objet de B a toutes les propriétés d'un objet de A (+ d'autres).
- Donc un objet B peut être considéré comme un objet A.
- Donc les variables définies pour un objet de A sont aussi présentes pour un objet de B (+ d'autres). (Mais elles peuvent être *occultées*)
- Idem pour les méthodes : Les méthodes de A sont présentes pour B et un objet B peut définir de nouvelles méthodes.
- Mais B peut *redéfinir* des méthodes de A.

Extension de classe

Si B est une extension de A

- pour les variables:
  - B peut ajouter des variables (et si le nom est identique cela *occultera* la variable de même nom dans A)

    (occulter = continuer à exister mais "caché")
  - *Les variables de A sont toutes présentes pour un objet B*, mais certaines peuvent être cachées
- pour les méthodes
  - B peut ajouter de nouvelles méthodes
  - B peut redéfinir des méthodes (même signature)

Remarques:

- pour les variables
  - c'est le nom de la variable qui est pris en compte (pas le type).
  - dans un contexte donné, à chaque nom de variable ne correspond qu'une seule déclaration.
  - (l'association entre le nom de la variable et sa déclaration est faite à la compilation)
- pour les méthodes
  - c'est la signature (nom + type des paramètres) qui est prise en compte:
    - on peut avoir des méthodes de même nom et de signatures différentes (surcharge)
    - dans un contexte donné, à un nom de méthode et à une signature correspond une seule définition
    - (l'association entre le nom de la méthode et sa déclaration est faite à la compilation, mais l'association entre le nom de la méthode et sa définition sera faite à l'exécution)

Extension (plus précisément)

- Si B est une extension de A (`class B extends A`)
  - Les variables et méthodes de A sont des méthodes de B (mais elles peuvent ne pas être accessibles: `private`)
  - B peut ajouter de nouvelles variables (si le *nom* est identique il y a occultation)
  - B peut ajouter des nouvelles méthodes si la *signature* est différente
  - B redéfinit des méthodes de A si la signature est identique

Remarques:

- Java est un langage typé
  - en particulier chaque variable a un type: celui de sa déclaration
  - à la compilation, la vérification du typage ne peut se faire que d'après les déclarations (implicites ou explicites)
  - le compilateur doit vérifier la légalité des appels des méthodes et des accès aux variables:
    - a.f() est légal si pour le type de la variable a il existe une méthode f() qui peut s'appliquer à un objet de ce type
    - a.m est légal si pour le type de la variable a il existe une variable m qui peut s'appliquer à un objet de ce type

En conséquence:

- Une variable déclarée comme étant de classe A peut référencer un objet de classe B ou plus généralement un objet d'une classe dérivée de A:
  - un tel objet contient tout ce qu'il faut pour être un objet de classe A
- Par contre une variable déclarée de classe B ne peut référencer un objet de classe A: il manque quelque chose!

Affectation downcast/upcast

```
class A{
    public int i;
    //...
}
class B extends A{
    public int j;
    //...
}
public class Affecter{
    static void essai(){
        A a = new A();
        B b = new B();
        //b=a; impossible que signifierait b.j??
        a=b; // a référence un objet B
        // b=a;
        b=(B)a; // comme a est un objet B ok!!
    }
}
```

Upcasting

- Si B est une extension de A, alors un objet de B peut être considéré comme un objet de A:
  - A a=new B();
- On pourrait aussi écrire:
  - A a=(A) new B();
- l'upcasting permet de considérer un objet d'une classe dérivée comme un objet d'une classe de base
- Upcasting: de spécifique vers moins spécifique (vers le haut dans la hiérarchie des classes)
- l'upcasting peut être implicite (il est sans risque!)
- attention
  - il ne s'agit pas réellement d'une conversion: l'objet n'est pas modifié

Downcasting

- Si B est une extension de A, il est possible qu'un objet de A soit en fait un objet de B. Dans ce cas on peut vouloir le considérer un objet de B
  - A a=new B();
  - B b=(B)a;
- Il faut dans ce cas un cast (transtypage) *explicite* (la "conversion" n'est pas toujours possible –l'objet référencé peut ne pas être d'un type dérivé de B)
- A l'exécution, on vérifiera que le cast est possible et que l'objet considéré est bien d'un type dérivé de B
- downcasting: affirme que l'objet considéré est d'un type plus spécifique que le type correspondant à sa décalration (vers le bas dans la hiérarchie des classes)
- le downcasting ne peut pas être implicite (il n'est pas toujours possible!)
- attention
  - il ne s'agit pas réellement d'une conversion: l'objet n'est pas modifié

Casting

- On peut tester la classe avant de faire du "downcasting":

```
Base sref;
Derive dref;
if(sref instanceof Derive)
    dref=(Derive) sref
```

B) Méthodes: Surcharge

- Méthodes et signature:
  - Signature: le nom et les arguments avec leur type (mais pas le type de la valeur retournée)
  - Seule la signature compte:
    - int f(int i)
    - char f(int i)
    - Les deux méthodes ont la même signature: c'est interdit
  - Surcharge possible:
    - Des signatures différentes pour un même nom

      int f(int i)

      int f(double f)
    - Le *compilateur* détermine par le type des arguments quelle fonction est utilisée (on verra les règles…)

Surcharge

- Un même nom de fonction pour plusieurs fonctions qui sont distinguées par leur signature

(Java, C++, Ada permettent la surcharge)

En C ' / ' est surchargé

3/2 division entière -> 1

3.0/2 division réelle -> 1,5

Surcharge

```
 public int f(int i){
        return i;
    }
//    public double f(int i){
//        return Math.sqrt( i);
//    }
    public int f(double i){
        return (int) Math.sqrt( i);
    }
    public int f(char c){
        return c;
    }
```

Remarques

- La résolution de la surcharge a lieu à la compilation
- La signature doit permettre cette résolution
- (quelques complications du fait du transtypage:
  - Exemple: un char est converti en int
  - Exemple: upcasting

)

C) Méthodes: Redéfinition

- Un classe hérite des méthodes des classes ancêtres
- Elle peut ajouter de nouvelles méthodes
- Elle peut surcharger des méthodes
- Elle peut aussi redéfinir des méthodes des ancêtres.

Exemple

```
class Mere{
    void f(int i){
        System.out.println("f("+i+") de Mere");
    }
    void f(String st){
        System.out.println("f("+st+") de Mere");
    }
}
```

Exemple (suite)

```
class Fille extends Mere{
    void f(){ //surcharge
        System.out.println("f() de Fille");
    }
//  char f(int i){
// même signature mais type de retour différent
//   }
    void g(){ //nouvelle méthode
        System.out.println("g() de Fille");
        f();
        f(3);
        f("bonjour");
    }
    void f(int i){ // redéfinition
        System.out.println("f("+i+") de Fille");
    }
}
```

Exemple

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

        Mere m=new Mere();
        Fille f=new Fille();
        m.f(3);
        f.f(4);
        m=f;
        m.f(5);
        //m.g();
        ((Fille)m).g();
        f.g();
    }
```

Résultat

f(3) de Mere
f(4) de Fille
f(5) de Fille
g() de Fille
f() de Fille
f(3) de Fille
f(bonjour) de Mere
g() de Fille
f() de Fille
f(3) de Fille
f(bonjour) de Mere

D) Conséquences

- Et les variables?
  - Un principe:
    - Une méthode (re)définie dans une classe A ne peut être évaluée que dans le contexte des variables définies dans la classe A.
      - Pourquoi?

Exemple

```
class A{
    public int i=4;
    public void f(){
        System.out.println("f() de A, i="+i);
    }
    public void g(){
        System.out.println("g() de A, i="+i);
    }
}
class B extends A{
    public int i=3;
    public void f(){
        System.out.println("f() de B, i="+i);
        g();
    }
}
```

Exemple suite:

```
A a=new B();
a.f();
System.out.println("a.i="+a.i);
System.out.println("((B) a).i="+((B)a).i);
```

Donnera:

- f() de B, i=3
- g() de A, i=4
- a.i=4
- ((B) a).i=3

Remarques:

- La variable i de A est *occultée* par la variable i de B
- La variable i de A est toujours présente dans tout objet de B
- Le méthode g de A a accès à toutes les variables définies dans A (et uniquement à celles-là)
- La méthode f de B *redéfinit* f. f() redéfinie a accès à toutes les variables définies dans B

E) Divers

- super
  - Le mot clé super permet d'accéder aux méthodes de la super classe
    - En particulier super permet d'appeler dans une méthode redéfinie la méthode d'origine

    (exemple: super.finalize() appelé dans une méthode qui redéfinit le finalize permet d'appeler le finalize de la classe de base)

Exemple

```
class Base{
    protected String nom(){
        return "Base";
    }
}
class Derive extends Base{
    protected String nom(){
        return "Derive";
    }
    protected void print(){
        Base maref = (Base) this;
        System.out.println("this.name():"+this.nom());
        System.out.println("maref.name():"+maref.nom());
        System.out.println("super.name():"+super.nom());
    }
-------------
this.name():Derive
maref.name():Derive
super.name():Base
```

Contrôle d'accès

- `protected`: accès dans les classes dérivées
- Le contrôle d'accès ne concerne pas la signature
- Une méthode redéfinie peut changer le contrôle d'accès mais uniquement pour élargir l'accès (de `protected` à `public`)
- Le contrôle d'accès est vérifié à la compilation

Interdire la redéfinition

- Le modificateur `final` interdit la redéfinition pour une méthode
- (Bien sûr une méthode de classe ne peut pas être redéfinie! Mais, elle peut être surchargée)
- Une variable avec modificateur `final` peut être occultée

E) Constructeurs et héritage

- Le constructeurs ne sont pas des méthodes comme les autres:
  - le redéfinition n'a pas de sens.
- Appeler un constructeur dans un constructeur:
  - super() appelle le constructeur de la super classe
  - this() appelle le constructeur de la classe elle-même
  - Ces appels doivent se faire au début du code du constructeur

Constructeurs

- Principes:
  - Quand une méthode d'instance est appelée l'objet est déjà créé.
  - Création de l'objet (récursivement)
    1. Invocation du constructeur de la super classe
    2. Initialisations des champs par les initialisateurs et les blocs d'initialisation
    3. Une fois toutes ces initialisations faites, appel du corps du constructeur (super() et this() ne font pas partie du corps)

Exemple

```
class X{
    protected int xMask=0x00ff;
    protected int fullMask;
    public X(){
        fullMask = xMask;
    }
    public int mask(int orig){
    return (orig & fullMask);
    }
}
class Y extends X{
    protected int yMask = 0xff00;
    public Y(){
        fullMask |= yMask;
    }
}
```

Résultat

	xMask	yMask	fullMask
Val. par défaut des champs	0	0	0
Appel Constructeur pour Y	0	0	0
Appel Constructeur pour X	0	0	0
Initialisation champ X	0x00ff	0	0
Constructeur X	0x00FF	0	0x00FF
Initialisation champs de Y	0x00FF	0xFF00	0x00FF
Constructeur Y	0x00FF	0xFF00	0xFFFF

La classe Object

- Toutes les classes héritent de la classe Object
- méthodes:
  - public final Class<? extends Object> **getClass**()
  - public int **hashCode**()
  - public boolean **equals**(Object obj)
  - protected Object **clone**() throws CloneNotSupportedException
  - public String **toString**()
  - protected void **finalize**() throws Throwable
  - (wait, notify,notfyall)

Exemple

```
class A{
    int i;
    int j;
    A(int i,int j){
        this.i=i;this.j=j;}
}
class D <T>{
    T i;
    D(T i){
        this.i=i;
    }
}
```

Suite

```
public static void main(String[] args) {
    A a=new A(1,2);
    A b=new A(1,2);
    A c=a;
    if (a==b)
        System.out.println("a==b");
    else
        System.out.println("a!=b");
    if (a.equals(b))
        System.out.println("a equals b");
    else
         System.out.println("a not equals b");
    System.out.println("Objet a: "+a.toString()+" classe "+a.getClass());
    System.out.println("a.hashCode()"+a.hashCode());
    System.out.println("b.hashCode()"+b.hashCode());
    System.out.println("c.hashCode()"+c.hashCode());
    D <Integer> x=new D<Integer>(10);
    System.out.println("Objet x: "+x.toString()+" classe "+x.getClass());
}
```

Résultat:

- a!=b
- a not equals b
- Objet a: A@18d107f classe class A
- a.hashCode()26022015
- b.hashCode()3541984
- c.hashCode()26022015
- Objet x: D@ad3ba4 classe class D

En redéfinissant equals

```
class B{
    int i;
    int j;
    B(int i,int j){
        this.i=i;this.j=j;
    }
    public boolean equals(Object o){
        if (o instanceof B)
            return i==((B)o).i && j==((B)o).j;
        else return false;
    }
}
```

Suite

```
B d=new B(1,2);
B e=new B(1,2);
B f=e;
if (d==e)
    System.out.println("e==d");
else
    System.out.println("d!=e");
if (d.equals(e))
    System.out.println("d equals e");
else
    System.out.println("a not equals b");
System.out.println("Objet d: "+d.toString());
System.out.println("Objet e: "+e.toString());
System.out.println("d.hashCode()"+d.hashCode());
System.out.println("e.hashCode()"+e.hashCode());
```

Résultat:

- d!=e
- d equals e
- Objet d: B@182f0db
- Objet e: B@192d342
- d.hashCode()25358555
- e.hashCode()26399554

Chapitre IV

Interfaces, classes imbriquées, Object

Chapitre IV

1. Interfaces
2. Classes imbriquées
3. Objets, clonage

classes abstraites

```
abstract class Benchmark{
    abstract void benchmark();
    public final long repeat(int c){
        long start =System.nanoTime();
        for(int i=0;i<c;i++)
            benchmark();
        return (System.nanoTime() -start);
    }
}
class MonBenchmark extends Benchmark{
    void benchmark(){
    }
    public static long mesurer(int i){
        return new MonBenchmark().repeat(i);

    }
}
```

suite

```
public static void main(String[] st){
   System.out.println("temps="+
            MonBenchmark.mesurer(1000000));
    }
```

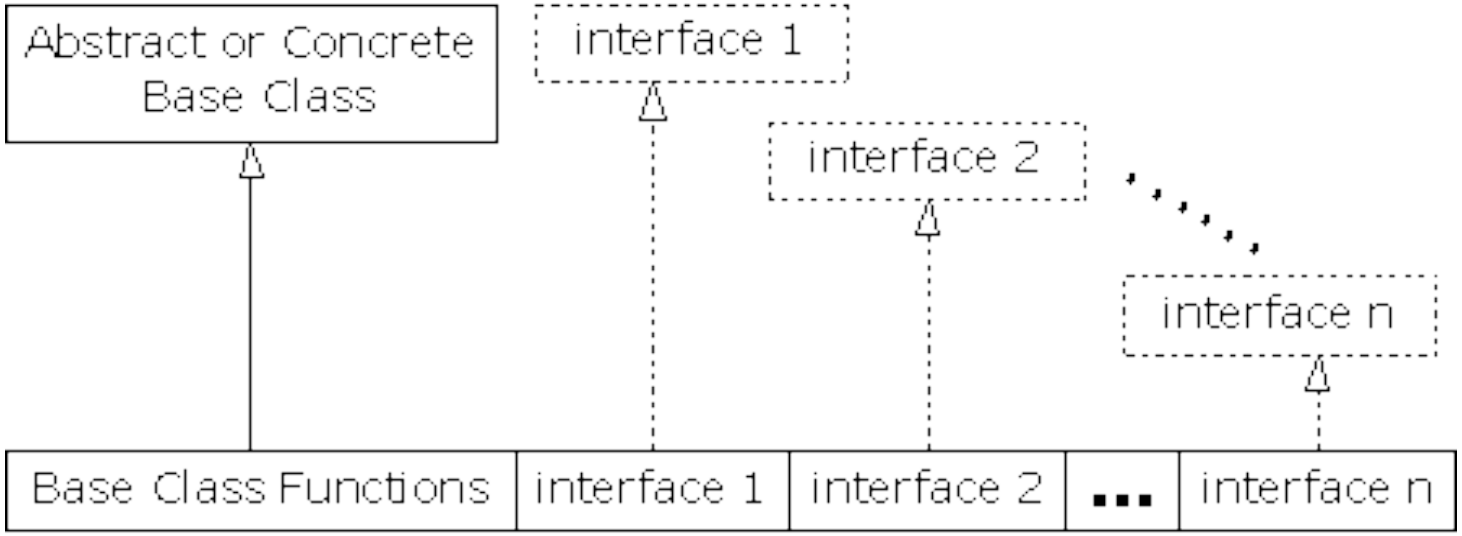
Résultat:

temps=6981893

Interfaces

- Il n'y a pas d'héritage multiple en Java: une classe ne peut être l'extension que d'une seule classe
- Par contre une classe peut implémenter plusieurs interfaces (et être l'extension d'une seule classe)
- Une interface ne contient (essentiellement) que des déclarations de méthodes
- Une interface est un peu comme une classe sans données membres et dont toutes les méthodes seraient abstraites

Héritage "multiple" en java



Exemple:

```
interface Comparable<T>{
    int compareTo(T obj);
}
class Couple implements Comparable<Couple>{
    int x,y;
    //
    public int compareTo(Couple c){
        if(x<c.x)return 1;
        else if (c.x==x)
            if (c.y==y)return 0;
        return -1;
    }
}
```

Remarques…

- Pourquoi, a priori, l'héritage multiple est plus difficile à implémenter que l'héritage simple?
- Pourquoi, a priori, implémenter plusieurs interfaces ne pose pas (trop) de problèmes?
- (Comment ferait-on dans un langage comme le C?)

Quelques interfaces…

- `Cloneable`: est une interface vide(!) un objet qui l'implémente peut redéfinir la méthode clone
- `Comparable`: est une interface qui permet de comparer les éléments (méthode `compareTo`)
- `runnable`: permet de définir des "threads"
- `Serializable`: un objet qui l'implémente peut être "sérialisé" = converti en une suite d'octets pour être sauvegarder.

Déclarations

- une interface peut déclarer:
  - des constantes (toutes les variables déclarées sont `static` `public` et `final`)
  - des méthodes (elles sont implicitement `abstract`)
  - des classes internes et des interfaces

Extension

les interfaces peuvent être étendues avec extends:

- Exemple:

  public interface SerializableRunnable
  extends Serializable, Runnable;

  (ainsi une interface peut étendre de plusieurs façons une même interface, mais comme il n'y a pas d'implémentation de méthodes et uniquement des constantes ce n'est pas un problème)

Exemple

```
interface X{
    int val=0;
}
interface Y extends X{
    int val=1;
    int somme=val+X.val;
}
class Z implements Y{}

public class InterfaceHeritage {
    public static void main(String[] st){
        System.out.println("Z.val="+Z.val+" Z.somme="+Z.somme);
        Z z=new Z();
         System.out.println("z.val="+z.val+
                 " ((Y)z).val="+((Y)z).val+
                 " ((X)z).val="+((X)z).val);
    }
}
---------------
```

Z.val=1 Z.somme=1

z.val=1 ((Y)z).val=1 ((X)z).val=0

Redéfinition, surcharge

```
interface A{
    void f();
    void g();
}
interface B{
    void f();
    void f(int i);
    void h();
}
interface C extends A,B{}
```

Rien n'indique que les deux méthodes `void f()` ont la même "sémantique". Comment remplir le double contrat?

Chapitre IV

1. Interfaces
2. Classes internes et imbriquées
3. Object, clonage

Classes imbriquées (nested classes)

- Classes membres statiques
  - membres statiques d'une autre classe
- Classes membres ou classes internes (inner classes)
  - membres d'une classe englobante
- Classes locales
  - classes définies dans un bloc de code
- Classes anonymes
  - classes locales sans nom

Classe imbriquée statique

- membre statique d'une autre classe
  - classe ou interface
  - mot clé `static`
  - similaire aux champs ou méthodes statiques: n'est pas associée à une instance et accès uniquement aux champs statiques

Exemple

```
class PileChainee{
        public static interface Chainable{
                public Chainable getSuivant();
                public void setSuivant(Chainable noeud);
        }
        Chainable tete;
        public void empiler(Chainable n){
            n.setSuivant(tete);
            tete=n;
        }
        public Object depiler(){
            Chainable tmp;
            if (!estVide()){
                tmp=tete;
                tete=tete.getSuivant();
                return tmp;
            }
            else return null;
        }
        public boolean estVide(){
            return tete==null;
        }
}
```

exemple (suite)

```
class EntierChainable implements PileChainee.Chainable{
    int i;
    public EntierChainable(int i){this.i=i;}
    PileChainee.Chainable next;
    public PileChainee.Chainable getSuivant(){
        return next;
    }
    public void setSuivant(PileChainee.Chainable n){
        next=n;
    }
    public int val(){return i;}
}
```

et le main

```
public static void main(String[] args) {
        PileChainee p;
        EntierChainable n;
        p=new PileChainee();
        for(int i=0; i<12;i++){
            n=new EntierChainable(i);
            p.empiler(n);
        }
        while (!p.estVide()){
            System.out.println(
                    ((EntierChainable)p.depiler()).val());
        }
    }
```

Remarques

- Noter l'usage du nom hiérarchique avec '.'
- On peut utiliser un import:
  - `import PileChainee.Chainable;`
  - `import PileChainee;`

(Exercice: réécrire le programme précédent sans utiliser de classes membres statiques)

Classe membre

- membre non statique d'une classe englobante
- peut accéder aux champs et méthodes de l'instance
- une classe interne ne peut pas avoir de membres statiques
- un objet d'une classe interne est une partie d'un objet de la classe englobante

Exemple

```
class CompteBanquaire{
    private long numero;
    private long balance;
    private Action der;
    public class Action{
        private String act;
        private long montant;
        Action(String act, long montant){
            this.act=act;
            this.montant= montant;
        }
        public String toString(){
            return numero"+":"+act+" "+montant;
        }
    }
```

Suite

```
 //…
    public void depot(long montant){
        balance += montant;
        der=new Action("depot",montant);
    }
    public void retrait(long montant){
        balance -= montant;
        der=new Action("retrait",montant);
    }

}
```

Remarques

- numero dans toString
- this:
  - der=this.new Action(…);
  - CompteBancaire.this.numero

Classe interne et héritage

```
class Externe{
    class Interne{}
}
class ExterneEtendue extends Externe{
    class InterneEtendue extends Interne{}
    Interne r=new InterneEtendue();
}
class Autre extends Externe.Interne{
    Autre(Externe r){
        r.super();
    }
}
```

(un objet Interne (ou d'une de ses extensions) n'a de sens qu'à l'intérieur d'un objet Externe)

Quelques petits problèmes

```
class X{
    int i;
    class H extends Y{
        void incremente(){i++;}
    }
}
```

Si i est une donnée membre de Y… c'est ce i qui est incrémenté

X.this.i et this.i lèvent cette ambiguïté.

Suite

```
class H{
    void print(){}
    void print(int i){}
    class I{
        void print(){};
        void show(){
            print();
            H.this.print();
        // print(1); tous les print sont occultés
        }
    }
}
```

Classes locales

- classes définies à l'intérieur d'un bloc de code,
- analogue à des variables locales: une classe interne locale n'est pas membre de la classe et donc pas d'accès,
- usage: créer des instances qui peuvent être passées en paramètres
- usage: créer des objets d'une extension d'une classe qui n'a de sens que localement (en particulier dans les interfaces graphiques)

Exemple

- classes Collections (ou Containers): classes correspondant à des structures de données.
  - exemples: List, Set, Queue, Map.
- L'interface Iterator permet de parcourir tous les éléments composant une structure de données.

Iterator

```
public interface Iterator<E>{
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove()throws
          UnsupportedOperationException,
          IllegalStateException;
}
```

Exemple: MaCollection

```
class MaCollection implements Iterator<Object>{
    Object[] data;
    MaCollection(int i){
        data=new Object[i];
    }
    MaCollection(Object ... l){
        data=new Object[l.length];
        for(int i=0;i<l.length;i++)
            data[i]=l[i];
    }
    private int pos=0;
    public boolean hasNext(){
        return (pos <data.length);
    }
    public Object next() throws NoSuchElementException{
        if (pos >= data.length)
            throw new NoSuchElementException();
        return data[pos++];
    }
    public void remove(){
        throw new  UnsupportedOperationException();
    }
}
```

Et une iteration:

```
public class Main {
    public static void afficher(Iterator it){
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
    public static void main(String[] args) {
        MaCollection m=new MaCollection(1,2,3,5,6,7);
        afficher(m);
    }
}
```

Classe locale

- Au lieu de créer d'implémenter Iterator on pourrait aussi créer une méthode qui retourne un iterateur.

Exemple parcourir

```
public static Iterator<Object> parcourir(final Object[] data){
        class Iter implements Iterator<Object>{
            private int pos=0;
            public boolean hasNext(){
                return (pos <data.length);
            }
            public Object next() throws NoSuchElementException{
                if (pos >= data.length)
                    throw new NoSuchElementException();
                return data[pos++];
            }
            public void remove(){
                throw new  UnsupportedOperationException();
            }
        }
        return new Iter();
    }
```

et l'appel

```
Integer[] tab=new Integer[12];
//…
afficher(parcourir(tab));
```

Remarques

- `parcourir()` retourne un itérateur pour le tableau passé en paramètre.
- l'itérateur implémente `Iterator`
  - mais dans une classe locale à la méthode `parcourir`
  - la méthode `parcourir` retourne un objet de cette classe.
- `data[]` est déclaré `final`:
  - même si tous les objets locaux sont dans la portée de la classe locale, la classe locale ne peut accéder aux variables locales que si elles sont déclarées `final`.

Anonymat…

- mais était-il utile de donner un nom à cette classe qui ne sert qu'à créer un objet Iter?

Classe anonyme

```
public static Iterator<Object> parcourir1( final Object[] data){
            return new Iterator<Object>(){
            private int pos=0;
            public boolean hasNext(){
                return (pos <data.length);
            }
            public Object next() throws NoSuchElementException{
                if (pos >= data.length)
                    throw new NoSuchElementException();
                return data[pos++];
            }
            public void remove(){
                throw new  UnsupportedOperationException();
            }
        };
    }
```

Exemple interface graphique:

```
jButton1.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent evt){
           jButton1ActionPerformed(evt);
   }
 });
```

Principe…

- ActionListener est une interface qui contient une seule méthode
  - void **actionPerformed**(ActionEvent e)
  - cette méthode définit le comportement voulu si on presse le bouton
- Il faut que le Button `jButton1` associe l'événement correspondant au fait que le bouton est pressé l'ActionListener voulu: `addActionListener`

Dans l'exemple

1. `jButton1ActionPerformed` est la méthode qui doit être activée
2. Création d'un objet de type ActionListener:
   1. (Re)définition de `ActionPerformed` dans l'interface ActionListener: appel de `jButton1ActionPerformed`
   2. classe anonyme pour ActionListener
   3. operateur new
3. ajout de cet ActionListener comme écouteur des événements de ce bouton `jButton1.addActionListener`

Chapitre IV

1. Interfaces
2. Classes imbriquées
3. Objets, clonage

Le clonage

- les variables sont des références sur des objets -> l'affectation ne modifie pas l'objet
- la méthode `clone` retourne un nouvel objet dont la valeur initiale est une copie de l'objet

Points techniques

- Par défaut la méthode `clone` de `Object` duplique les champs de l'objet (et dépend donc de la classe de l'objet)
- L'interface `Cloneable` doit être implémentée pour pouvoir utiliser la méthode `clone` de `Object`
  - Sinon la méthode `clone` de `Object` lance une exception `CloneNotSupportedException`
- De plus, la méthode `clone` est `protected` -> elle ne peut être utilisée que dans les méthodes définies dans la classe ou ses descendantes (ou dans le même package).

En conséquence

- en implémentant `Cloneable`, `Object.clone()` est possible pour la classe et les classes descendantes
  - Si `CloneNotSupportedException` est captée, le clonage est possible pour la classe et les descendants
  - Si on laisse passer `CloneNotSupportedException`, le clonage peut être possible pour la classe (et les descendants) (exemple dans une collection le clonage sera possible si les éléments de la collection le sont)
- en n'implémentant pas `Cloneable`, `Object.clone()` lance uniquement l'exception, en définissant une méthode clone qui lance une `CloneNotSupportedException`, le clonage n'est plus possible

Exemple

```
class A implements Cloneable{
    int i,j;
    A(int i,int j){
        this.i=i; this.j=j;
    }
    public String toString(){
        return "(i="+i+",j="+j+")";
    }
    protected Object clone()
       throws CloneNotSupportedException{
        return super.clone();
    }
}
```

Suite

```
A a1=new A(1,2);
A a2=null;
try {// nécessaire!
    a2 =(A) a1.clone();
} catch (CloneNotSupportedException ex) {
    ex.printStackTrace();
}
```

donnera:

a1=(i=1,j=2) a2=(i=1,j=2)

Suite

```
class D extends A{
    int k;
    D(int i,int j){
        super(i,j);
        k=0;
    }
     public String toString(){
        return ("(k="+k+")"+super.toString());
     }
}
//…
     D d1=new D(1,2);
     D d2=null;
     try { //nécessaire
            d2=(D) d1.clone();
     } catch (CloneNotSupportedException ex) {
            ex.printStackTrace();
      }
     System.out.println("d1="+d1+" d2="+d2);
}
```

Remarques

- `super.clone();` dans A est nécessaire il duplique *tous* les champs d'un objet de D
- Pour faire un clone d'un objet D il faut capter l'exception.

Suite

```
class B implements Cloneable{
    int i,j;
    B(int i,int j){
        this.i=i; this.j=j;
    }
    public String toString(){
        return "(i="+i+",j="+j+")";
    }

    protected Object clone(){
        try {
            return super.clone();
        } catch (CloneNotSupportedException ex) {
            ex.printStackTrace();
            return null;
        }
    }
}
```

Suite

```
class C extends B{
    int k;
    C(int i,int j){
        super(i,j);
        k=0;
    }
    public String toString(){
        return ("(k="+k+")"+super.toString());
    }
}//...
 B b1=new B(1,2);
 B b2 =(B) b1.clone();
 C c1=new C(1,2);
 C c2 =(C) c1.clone();
```

Pourquoi le clonage?

- Partager ou copier?
- Copie profonde ou superficielle?
  - par défaut la copie est superficielle:

Exemple

```
class IntegerStack implements Cloneable{
    private int[] buffer;
    private int sommet;
    public IntegerStack(int max){
        buffer=new int[max];
        sommet=-1;
    }
    public void empiler(int v){
        buffer[++sommet]=v;
    }
    public int dépiler(){
        return buffer[sommet--];
    }
    public IntegerStack clone(){
        try{
            return (IntegerStack)super.clone();
        }catch(CloneNotSupportedException e){
            throw new InternalError(e.toString());
        }
    }
}
```

Problème:

```
IntegerStack un=new IntegerStack(10);
un.emplier(3);
un.empiler(9)
InetegerStack deux=un.clone();
```

Les deux piles partagent les mêmes données…

Solution…

```
public IntegerStack clone(){
  try{
     IntegerStack nObj = (IntegerStack)super.clone();
     nObj.buffer=buffer.clone();
     return nObj;
  }catch(CloneNotSupportedException e){
       //impossible
       throw new InternalError(e.toString());
   }
 }
```

Copie profonde

```
public class CopieProfonde implements Cloneable{
    int val;
    CopieProfonde n=null;
    public CopieProfonde(int i) {
        val=i;
    }
    public CopieProfonde(int i, CopieProfonde n){
        this.val=i;
        this.n=n;
    }
    public Object clone(){
        CopieProfonde tmp=null;
        try{
            tmp=(CopieProfonde)super.clone();
            if(tmp.n!=null)
                tmp.n=(CopieProfonde)(tmp.n).clone();
        }catch(CloneNotSupportedException ex){}
        return tmp;
    }
}
```

Suite

```
class essai{
    static void affiche(CopieProfonde l){
        while(l!=null){
            System.out.println(l.val+" ");
            l=l.n;
        }
    }
    public static void main(String[] st){
        CopieProfonde l=new CopieProfonde(0);
        CopieProfonde tmp;
        for(int i=0;i<10;i++){
            tmp=new CopieProfonde(i,l);
            l=tmp;
        }
        affiche(l);
        CopieProfonde n=(CopieProfonde)l.clone();
        affiche(n);
    }
}
```

Chapitre V

Enumeration, tableaux, conversion de types, noms

Types énumérés

- Exemple:
  - `enum Couleur {PIQUE, CŒUR, CARREAU, TREFLE,}`
  - définit des constantes énumérées (champs static de la classe)
  - on peut définir des méthodes dans un enum
  - des méthodes
    - public static E[] values() retourne les constantes dans l'ordre de leur énumeration
    - public static E valueOf(String nom) la constante associé au nom
  - un type enum étend implicitement java.lang.Enum (aucune classe ne peut étendre cette classe)

Tableaux

- collection ordonnée d'éléments,
- les tableaux sont des Object
- les composants peuvent être de types primitifs, des références à des objets (y compris des références à des tableaux),

Tableaux

- int [] tab= new int t[3];
  - déclaration d'un tableau d'int
  - initialisé à un tableau de 3 int
- indices commencent à 0
- contrôle de dépassement
  - ArrayIndexOutOfBoundException
- length donne la taille du tableau

Tableaux

- un tableau final: la référence ne peut être changée (mais le tableau référencé peut l'être)
- tableaux de tableaux:
- exemple:

```
public static int[][] duplique(int[][] mat){
        int[][] res= new int[mat.length][];
        for(int i=0;i<mat.length;i++){
            res[i]=new int[mat[i].length];
            for (int j=0;j<mat[i].length;j++)
                res[i][j]=mat[i][j];
        }
        return res;
    }
```

Tableaux

- exemple:

```
public static void affiche(int [][] tab){
        for(int[] d:tab){
            System.out.println();
            for(int v:d)
                System.out.print(v+" ");
        }
    }
```

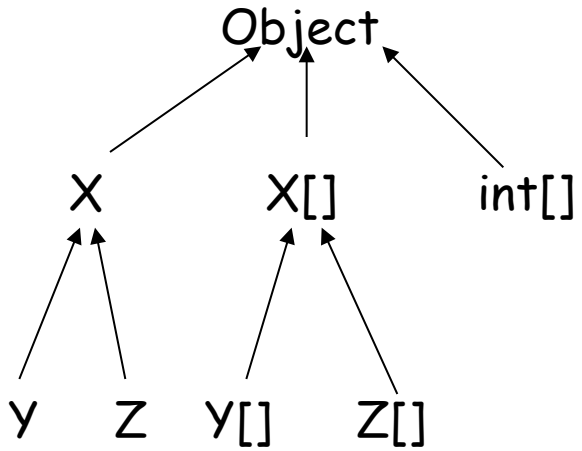
Initialisations

```
static int[][] pascal={
        {1},
        {1,1},
        {1,2,1},
        {1,3,3,1},
    };
String[] nombre= {"un", "deux", "trois", "quatre"};
```

Exemple

```
for(int i=1;i<p.length;i++){
      p[i]=new int[i+1];
      p[i][0]=1;
      for(int j=1; j<i;j++){
          p[i][j]=p[i-1][j-1]+p[i-1][j];
      }
      p[i][i]=1;
}
```

Tableau et héritage



- `Y[] yA=new Y[3];`
- `X[] xA=yA; //ok`
- `xA[0]=new Y();`
- `xA[1]=new X(); //non`
- `xA[1]=new Z(); //non`

Noms

- il ya 6 espaces de noms
  - package
  - type
  - champs
  - méthode
  - variable locale
  - étiquette

Noms!?

```
package divers;
class divers{
    divers divers(divers divers){
        divers:
            for(;;){
            if (divers.divers(divers)==divers)
                break divers;
            }
            return divers;
    }
}
```

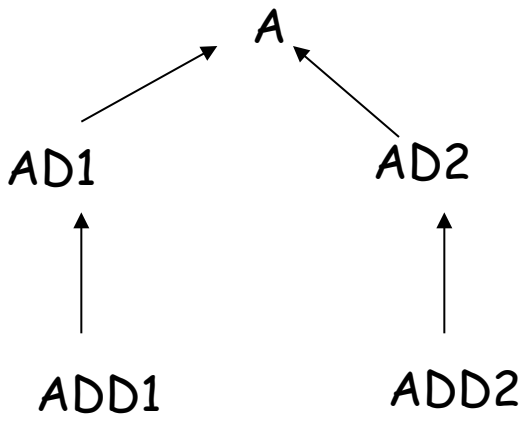
Trouver la bonne méthode

- Il faut pouvoir déterminer une seule méthode à partir d'une invocation avec des paramètres
- problèmes: héritage et surcharge
  - (en plus des problèmes liés à la généricité)
- principe trouver la méthode "la plus spécifique"

Règles

1. *déterminer dans quelle classe chercher la méthode (uniquement par le nom)*
2. trouver les méthodes dans la classe qui peuvent s'appliquer
    1. sans "boxing" sans nombre variable d'arguments
    2. avec boxing
    3. avec un nombre variable d'arguments
3. si une méthode a des types de paramètres qui peuvent être affectés à une autre des méthodes de l'ensemble -> la supprimer
4. s'il ne reste qu'une méthode c'est elle (sinon ambiguïté sauf s'il s'agit de méthodes abstraites)

Exemple




```
void f(A a,AD2 ad2)//un
void f(AD1 ad1,A a)//deux
void f(ADD1 add1, AD2 s)//trois
void f(A … a)//quatre

f(Aref, AD2ref);//a
f(ADD1ref, Aref);//b
f(ADD1ref, ADD2ref);//c
f(AD1ref, AD2ref);//d
f(AD2ref, AD1ref);//e
```

Exemple (suite)

- (a) correspond exactement à (un)
- (b) correspond à (deux)
- (c) peut correspondre aux trois premiers mais (un) est moins spécifique que (trois) idem entre (un) et (trois) d'où résultat (trois)
- (d) (un) et (deux) ne peuvent s'éliminer
- (e) uniquement (quatre)

Chapitre VI
Exceptions

VI) Exceptions

1. Principes généraux
2. Déclarations de throws
3. try, catch et finally
4. Transfert d'information: chainage, pile
5. Assertions

Exceptions et assertions

- principe:
  - traitement des "erreurs"
    - quand une exception est lancée:
      - rupture de l'exécution séquentielle
      - "dépiler" les méthodes et les blocs
      - jusqu'à un traite exception adapté
  - erreur:
    - rupture du contrat:
      - précondition violée

Exceptions

- checked ou unchecked
  - checked: cas exceptionnel, mais dont l'occurrence est prévue et peut être traitée (exemple: valeur en dehors des conditions de la précondition, …)
    - Une méthode qui peut lancer une checked exception doit le déclarer
  - unchecked: il n'y a rien à faire, (exemple une erreur interne de la JVM) ou une erreur à l'exécution (dépassement de tableau)
    - Une méthode qui peut lancer une unchecked exception ne doit pas le déclarer

Exceptions

- Une exception est un objet d'une classe dérivée de Throwable (mais, en fait, en général de Exception)
- Le mécanisme est le même que pour tout objets:
  - on peut définir des sous-classes
  - des constructeurs
  - redéfinir des méthodes
  - ajouter des méthodes

Exceptions et traite-exceptions

- Un traite exception déclare dans son entête un paramètre
- Le type du paramètre détermine si le traite-exception correspond à l'exception
  - même mécanisme que pour les méthodes et l'héritage

Throw

- Certaines exceptions et errors sont lancées par la JVM
- L'utilisateur peut définir ses propres exceptions et les lancer lui-même:

  `throw expression;`

  l'expression doit s'évaluer comme une valeur ou une variable qui peut être affectée à Throwable

Environnement

- Par définition une exception va transférer le contrôle vers un autre contexte
  - le contexte dans lequel l'exception est traitée est différent du contexte dans lequel elle est lancée
  - l'exception elle-même peut permettre de passer de l'information par son instanciation
  - l'état de la pile au moment de l'exception est aussi transmis
    public StackTraceElement[] **getStackTrace**() et
    public void **printStackTrace**()

Hiérarchie:

- java.lang.Throwable (implements java.io.Serializable)
  - java.lang.Error
    - java.lang.AssertionError
    - java.lang.LinkageError
    - java.lang.ThreadDeath
    - java.lang.VirtualMachineError
      - exemple:java.lang.StackOverflowError
  - java.lang.Exception
    - java.lang.ClassNotFoundException
    - java.lang.CloneNotSupportedException
    - java.lang.IllegalAccessException
    - java.lang.InstantiationException
    - java.lang.InterruptedException
    - java.lang.NoSuchFieldException
    - java.lang.NoSuchMethodException
    - java.lang.RuntimeException
      - exemple: java.lang.IndexOutOfBoundsException

Hiérarchie

- Throwable:
  - la super classe des erreurs et des exceptions
  - Error : unchecked
  - Exception :checked sauf RuntimeException

Exemple

```
public class MonException extends Exception{
    public final String nom;
    public MonException(String st) {
        super("le nombre "+st+" ne figure pas");
        nom=st;
    }
}
```

Exemple (suite)

```
class Essai{
    static String[] tab={"zéro","un","deux","trois","quatre"};
    static int chercher(String st ) throws MonException{
        for(int i=0;i<tab.length;i++)
            if (tab[i].equals(st))return i;
        throw new MonException(st);
    }

    public static void main(String st[]){
        try{
            chercher("zwei");
        }catch(Exception e){
            System.out.println(e);
        }

    }
}
```

Résultat

- Donnera:

```
exceptions.MonException: le nombre zwei ne figure pas
```

- e.printStackTrace(); dans le try bloc donnera:

```
exceptions.MonException: le nombre zwei ne figure pas
        at exceptions.Essai.chercher(MonException.java:29)
        at exceptions.Essai.main(MonException.java:34)
```

Throws

- principe:
  - toute méthode qui peut générer directement ou indirectement une (checked) exception doit le déclarer par une clause "throws" dans l'entête de la méthode.
    - (les initialiseurs statiques ne peuvent donc pas générer d'exceptions)
  - La vérification a lieu à la compilation

Clause throws

- Une méthode qui appelle une méthode qui peut lancer une exception peut
  - attraper (catch) cette exception dans un try bloc englobant la méthode qui peut lancer cette exception
  - attraper cette exception et la transformer en une exception déclarée dans la clause throws de la méthode
  - déclarer cette exception dans la clause throws de sa déclaration

Clause throws et héritage

- Si une classe dérivée redéfinit (ou implémente) une méthode la clause throws de la méthode redéfinie doit être compatible avec celle d'origine
  - compatible = les exceptions de la clause throws sont dérivées de celles de la méthode d'origine
  - pourquoi?

try, catch, finally

- On attrape les exceptions dans des try-bloc:

```
try{
   instructions
}catch(exception-type1 id1){
    instructions
} catch(exception-type2 id2){
    ...
}finally{
   instructions
}
```

Principe:

- le corps du try est exécuté jusqu'à ce qu'il termine ou qu'une exception est lancée
- Si une exception est lancée les clauses "catch" sont examinées dans l'ordre
  - la première dont le type peut correspondre à l'exception est choisie et son code exécuté
  - si aucun catch ne peut correspondre l'exception est propagée
  - si une clause finally figure son code est ensuite exécuté (toujours avec ou sans exception)

Exemple

```
class A extends Exception{
}
class B extends A{
}
class essai{
    public static void main(String[] st){
        try{
            throw new B();
        }catch (A a){
            System.out.println(a);
//        }catch (B b){
//            System.out.println(b);
        }finally{
            System.out.println("finally..");
        }
    }
}
```

finally

```
public boolean rechercher(String fichier,
      String mot) throws StreamException{
        Stream input=null;
        try{
            input=new Stream(fichier);
            while(!input.eof())
                if(input.next().equals(mot))
                    return true;
            return false;
        }finally{
            if (input != null)
                input.close();
        }
    }
```

Chaînage d'exceptions

- Une exception peut être causée par une autre.
- il peut être utile dans ce cas de transmettre la cause de l'exception
  - méthode:

public Throwable **initCause**(Throwable cause)

Transmission d'information

- en définissant une extension de la classe et en définissant des constructeurs
- par défaut on a les constructeurs

  public **Throwable**()

  public **Throwable**(String message)

  public **Throwable**(String message, Throwable cause)

Transmission d'information

- On peut récupérer ces informations:

```
public String getMessage()
public Throwable getCause()
```

- On peut obtenir l'état de la pile:

```
public void printStackTrace()
public StackTraceElement[]
   getStackTrace()
```

Exemple

```
class X extends Exception{
    public X(){}
    public X(String details){
        super(details);
    }
    public X(Throwable e){
        super(e);
    }
    public X(String details, Throwable e){
        super(details,e);
    }
}
```

Suite

```
try{
     throw new A();
 }catch (A a){
        try {
            throw new X(a);
        } catch (X ex) {
              ex.printStackTrace();
          }
   }
}
-----
X: A
      at essai.main(Finally.java:61)
Caused by: A
      at essai.main(Finally.java:58)
```

Remarque

- à la place de:

```
throw new X(a);
```

- on pourrait mettre

```
throw (X) new X().initCause(a);
```

(pourquoi le cast (X) est nécessaire?)

Assertions

- Une autre façon de garantir le contrat est de définir des assertions qui vérifient les invariants (ou les préconditions)
- Si l'assertion n'est pas vérifiée une `AssertionError` est lancée
- (une option de compilation permet de vérifier ou non les assertions)

Assertions

- Syntaxe:

  `assert` expr [: detail];

  - expr est une expression `boolean`
  - detail est optionnel et sera passé au constructeur de `AssertionError` (une string ou un `Throwable`)
- Exemple:

```
assert i!=0 : "i ="+i+" i devrait être non nul";
```

Assertions

- par défaut les assertions ne sont pas évaluées
- pour les évaluer:
  - -enableassertions:nom_du_package
  - -disableassertions:nom_du_package

  avec en argument le ou les paquetages concernés.

Java Swing

Principes de base

- Des composants graphiques
  (exemple: JFrame, JButton …)
  - Hiérarchie de classes
- Des événements et les actions à effectuer
  (exemple presser un bouton)
- (Et d'autres choses…)

Principes

- Définir les composants (instance de classes)
- Les placer à la main (layout Manager) dans un JPanel ou un content pane ou en utilisant des outils comme eclipse ou netbeans
- Définir les actions associées aux événements (Listener) et les associer aux composants graphiques

Principes

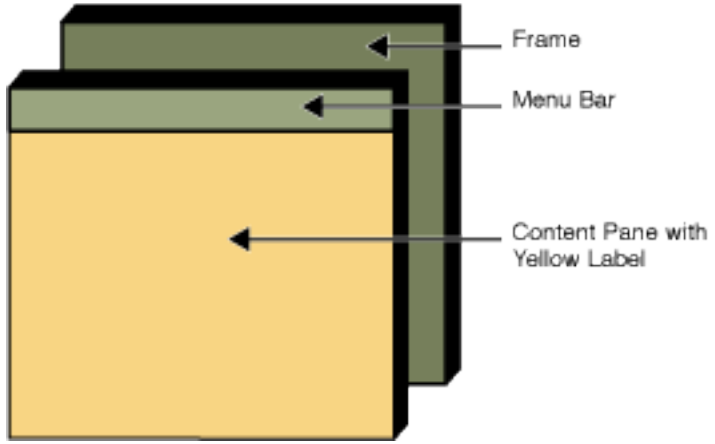
- Dans une interface graphique, le programme réagit aux interactions avec l'utilisateur
- Les interactions génèrent des événements
- Le programme est dirigé par les événements (event-driven)

Afficher…

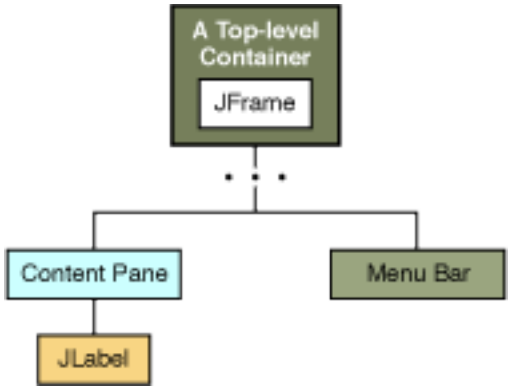
- Pour pouvoir être affiché, il faut que le composant soit dans un top-level conteneur:

  (JFrame, JDialog et JApplet)
- Hiérarchie des composants: arbre racine top-level

Exemple



- Correspond à la hiérarchie



Le code

```
import java.awt.*;
import javax.swing.*;

public class TopLevel {
    /**
     * Affiche une fenêtre JFrame top level
     * avec une barre de menu JMenuBar verte
     * et un JLabel jaune
     */
    private static void afficherMaFenetre() {
            //créer la Jframe
            //créer la JMenuBar
           //créer le Jlabel
           // mettre le JMenuBar et le Jlable dans la Jframe
           //afficher la Jframe
    }
}
```

Le code

```
//Creer la JFrame
  JFrame frame = new JFrame("TopLevelDemo");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  //Créer la JMenuBar
  JMenuBar greenMenuBar = new JMenuBar();
  greenMenuBar.setOpaque(true);
  greenMenuBar.setBackground(new Color(0, 200, 0));
  greenMenuBar.setPreferredSize(new Dimension(200, 20));
  //Créer le JLabel
  JLabel yellowLabel = new JLabel();
  yellowLabel.setOpaque(true);
  yellowLabel.setBackground(new Color(250, 250, 0));
  yellowLabel.setPreferredSize(new Dimension(200, 180));
  //mettre la JmenuBar et position le JLabel
  frame.setJMenuBar(greenMenuBar);
  frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
  //afficher...
  frame.pack();
  frame.setVisible(true);
```

Et le main

```
public class TopLevel {//afficherMaFenetre()
   public static void main(String[] args) {
      javax.swing.SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            afficherMaFenetre();
         }
      });
   }
}
```

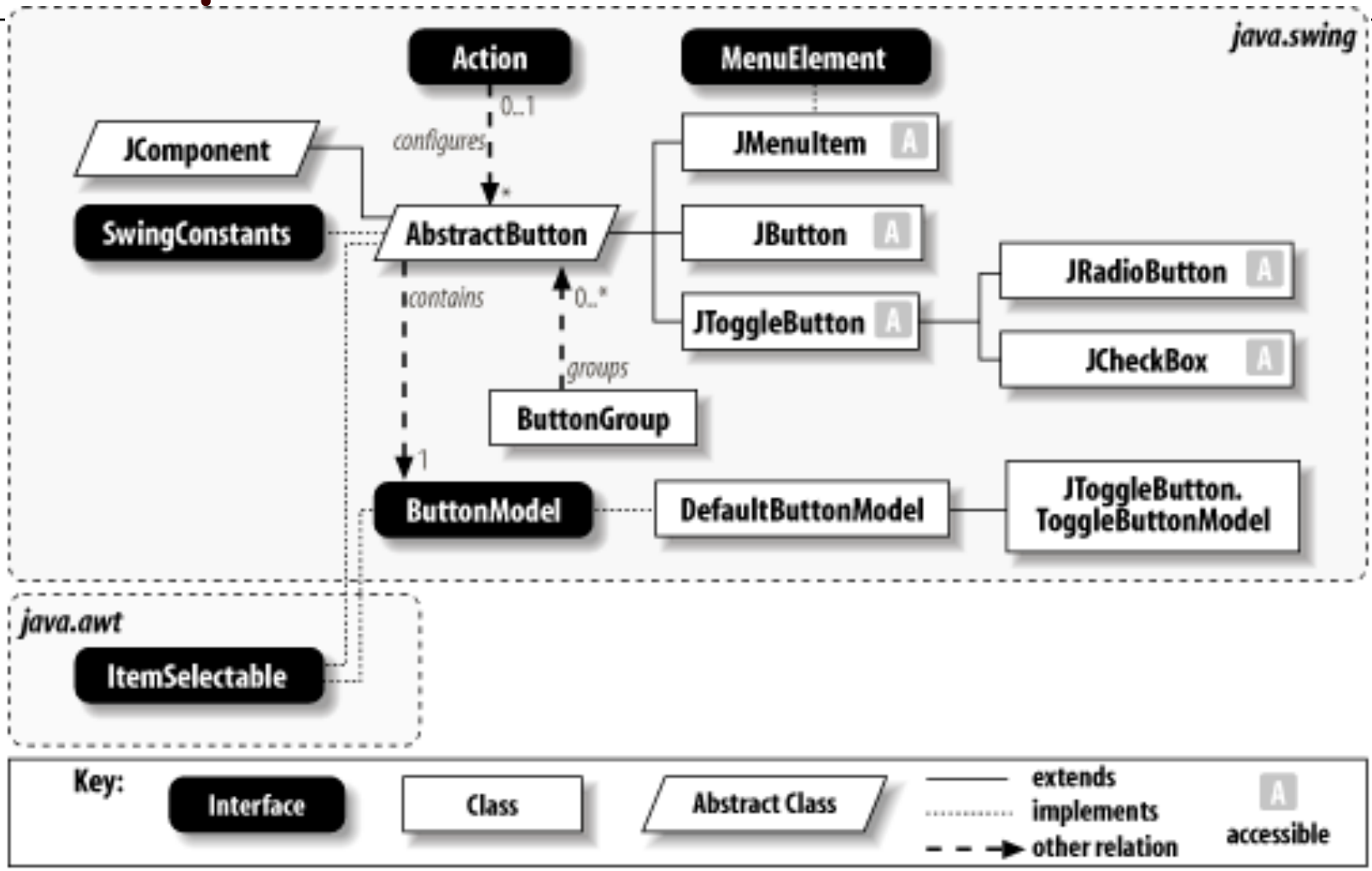
Evénements: principes

- Dans un système d'interface graphique:
  - Quand l'utilisateur presse un bouton, un "événement" est posté et va dans une boucle d'événements
  - Les événements dans la boucle d'événements sont transmis aux applications qui se sont enregistrées pour écouter.

Evénements

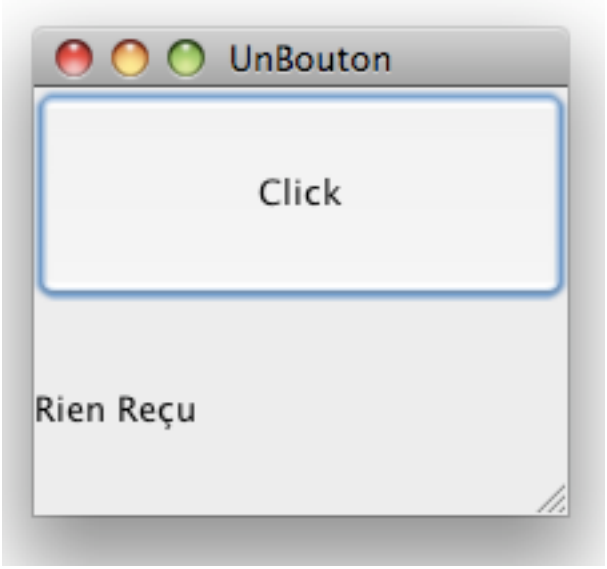
- Chaque composant génère des événements:
  - Presser un JButton génère un ActionEvent (système d'interface graphique)
    - Cet ActionEvent contient des infos (quel bouton, position de la souris, modificateurs…)
  - Un event listener (implémente ActionListener)
    - définit une méthode actionPerformed
    - S'enregistre auprès du bouton addActionListener
  - Quand le bouton est "clické",l'actionPerformed sera exécuté (avec l'ActionEvent comme paramètre)

Exemples Buttons



Un exemple

- Un bouton qui réagit



Le code:

- Un JButton
- Un JLabel
- Implementer ActionListener
  - actionPerfomed définit ce qui se passe quand le bouton est cliqué
- Placer le bouton et le label

Code:

```
import java.awt.*;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.Jcomponent;
import java.awt.Toolkit;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JLabel;

public class UnBouton extends Jpanel implements ActionListener {
    JButton bouton;
    String contenu="Rien Reçu";
    JLabel label=new JLabel(contenu);
    int cmp=0;
    public UnBouton() { //…}
    public void actionPerformed(ActionEvent e) {//…}
    private static void maFenetre(){//…}
    public static void main(String[] args) {//…}
}
```

Code

```
public UnBouton() {
        super(new BorderLayout());
        bouton = new JButton("Click");
        bouton.setPreferredSize(new Dimension(200, 80));
        add(bouton, BorderLayout.NORTH);
        label = new JLabel(contenu);
                label.setPreferredSize(new Dimension(200, 80));
        add(label,BorderLayout.SOUTH);
        bouton.addActionListener(this);
   }
public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
        label.setText("clické "+ (++cmp)+ " fois");
  }
```

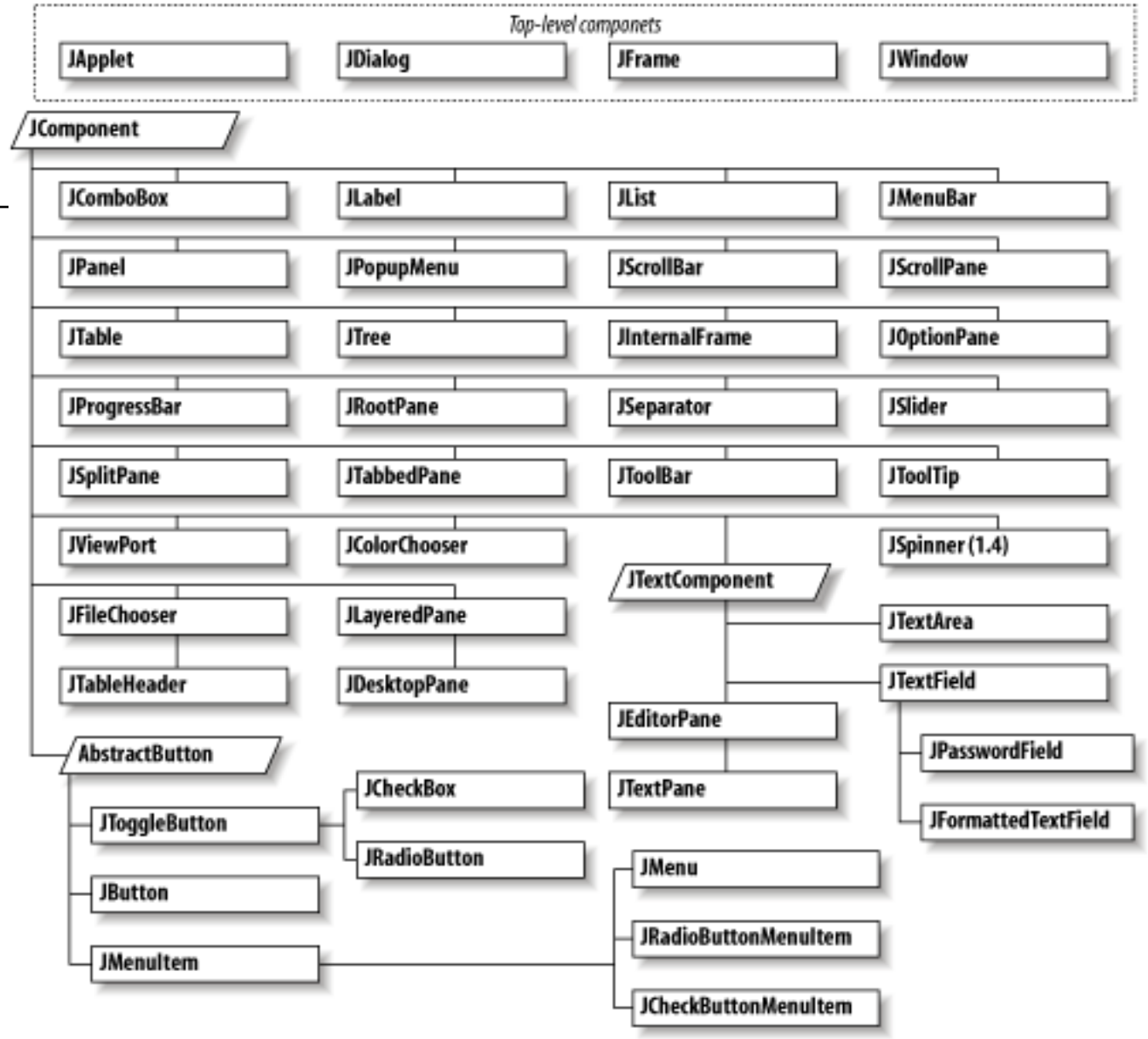
Code

```
private static void maFenetre() {
        JFrame frame = new JFrame("UnBouton");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent newContentPane = new UnBouton();
        newContentPane.setOpaque(true);
        frame.setContentPane(newContentPane);
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //Formule magique
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                maFenetre();
            }
        });
    }
```

Variante

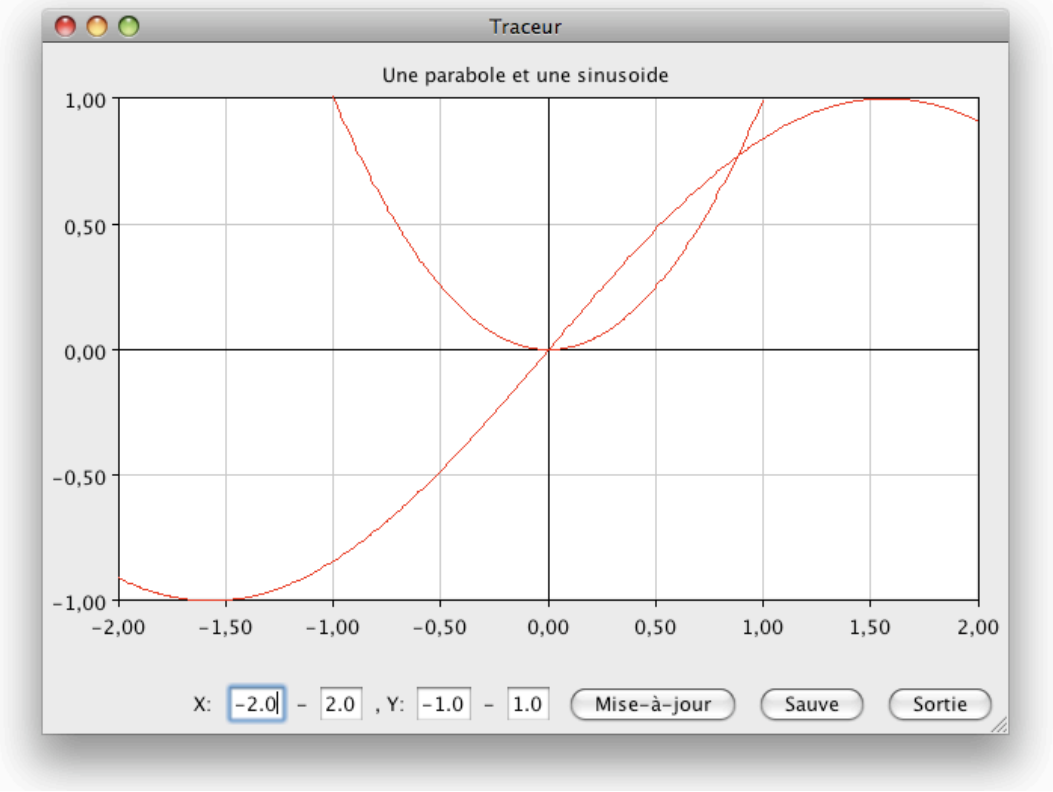
```
public class UnBoutonBis extends JPanel {
//…
 bouton.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
              Toolkit.getDefaultToolkit().beep();
              label.setText("clické " + (++cmp) + " fois");
          } });
      }
//…
}
```

Hiérarchie des classes…


Top-level componets
JApplet
JDialog
JFrame
JWindow
JComponent
JComboBox
JLabel
JList
JMenuBar
JPanel
JPopupMenu
JScrollBar
JScrollPane
JTable
JTree
JInternalFrame
JOptionPane
JProgressBar
JRootPane
JSeparator
JSlider
JSplitPane
JTabbedPane
JToolBar
JToolTip
JViewPort
JColorChooser
JSpinner (1.4)
JFileChooser
JLayeredPane
JTableHeader
JDesktopPane
JTextComponent
JTextArea
JTextField
JEditorPane
JPasswordField
JTextPane
JFormattedTextField
AbstractButton
JCheckBox
JToggleButton
JRadioButton
JButton
JMenu
JMenuItem
JRadioButtonMenuItem
JCheckButtonMenuItem

Un exemple

- Un traceur de fonctions
  - Une interface graphique swing



Organisation

- GrapheSwing contient un GraphePanel extension de Jpanel
  - GraphePanel méthode paintComponent qui affiche le graphe de la fonction
    - Graphe est la classe contenant le gaphe et définissant une méthode draw pour l'affichage
    - Cette méthode appelle tracer de la classe abstraite Traceur
      - FonctionTraceur étend Traceur

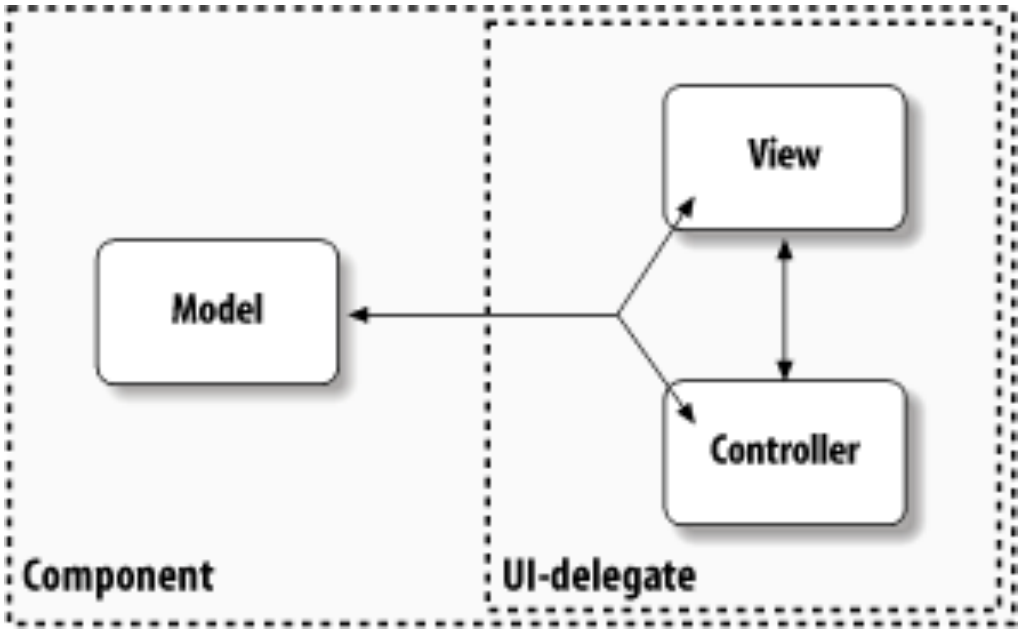
Le main

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

  public static Graphe unGraphe() {
     PlotSettings p = new PlotSettings(-2, 2, -1, 1);
     p.setPlotColor(Color.RED);
     p.setGridSpacingX(0.5);
     p.setGridSpacingY(0.5);
     p.setTitle("Une parabole et une sinusoide");
     Graphe graphe = new Graphe(p);
     graphe.functions.add(new Parabole());
     graphe.functions.add(new FonctionTraceur() {
        public double getY(double x) {
           return Math.sin(x);
        }
        public String getName() {
           return "Sin(x)";
        }
     });
     return graphe;
  }
```

Composants

- Modèle Vue Contrôleur



Préliminaires…

- Lightweight et heavyweight composants
  - Dépendent ou non du système d' interface graphique
    - Lightweight écrit en Java et dessiné dans un heavyweight composant- indépendant de la plateforme
    - Les heavyweight composants s' adressent directement à l' interface graphique du système
  - (certaines caractéristiques dépendent du « look and feel »).

Look and feel

- Look and feel:

  Possibilité de choisir l'apparence de l'interface graphique.

  UIManager gère l'apparence de l'interface

```
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }

    new SwingApplication(); //Create and show the GUI.
}
```

Multithreading

- Attention au « modèle, contrôleur, vue » en cas de multithreading:
  - Tous les événements de dessin de l'interface graphiques sont dans une unique file d'event-dispatching dans une seule thread.
  - La mise à jour du modèle doit se faire tout de suite après l'événement de visualisation dans cette thread.

Plus précisément

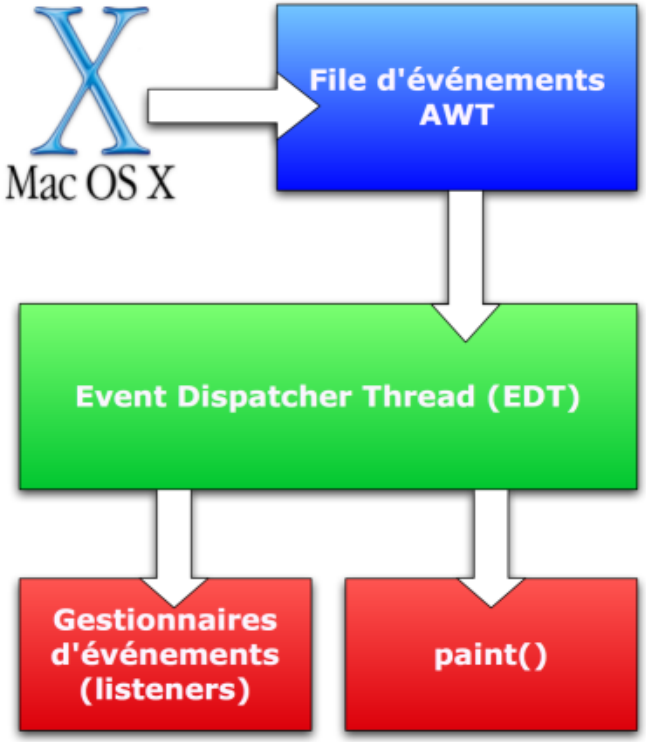
- Swing prend en charge la gestion des composants qui sont dessinés en code Java (lightweight)
- Les composants AWT sont eux liés aux composants natifs (heavyweight)
- Swing dessine le composants dans un canevas AWT et utilise le traitement des événements de AWT

Suite

Les threads

- Main application thread
- Toolkit thread
- Event dispatcher thread

- Toutes Les opérations d'affichage ont lieu dans une seule thread l'EDT



Principes

- Une tâche longue ne doit pas être exécutée dans l'EDT
- Un composant Swing doit s'exécuter dans l'EDT

Exemple

```
public void actionPerformed(ActionEvent e){
   try {
       Thread.sleep(4000);
} catch (InterruptedException e) {    }  }
```

Provoque une interruption de l'affichage pendant 4 secondes

Une solution

```
public void actionPerformed(ActionEvent e){
  try{
    SwingUtilities.invokeLater(newRunnable(
          { public void run() {
                //opération longue
                }
           });
   } catch (InterruptedException ie) {}
     catch (InvocationTargetException ite) {}
   }
}
```

Le main

- Normalement la création d'une fenêtre ne devrait avoir lieu que dans l'EDT:

```
public static void main(String[] args) {
        //Formule magique
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
                                        public void run() {maFenetre(); }
        });
    }
```

invokeLater crée une nouvelle thread qui poste la thread crée dans l'EDT

Attendre le résultat:

```
try {
        SwingUtilities.invokeAndWait(new Runnable() {
            public void run() {
                show();
            }
        });
    } catch (InterruptedException ie) {
    } catch (InvocationTargetException ite) {
    }
```

Chapitre VII

Généricité

Chapitre VII

1. Principes généraux
2. Types génériques imbriqués
3. Types paramètres bornés
4. Méthodes génériques

Principes

- Paramétrer une classe ou une méthode par un type:
  - une pile de X
- En java toute classe étant dérivée de Object, cela permet d'obtenir une forme de généricité sans contrôle des types
  - une pile d'Object
- La généricité en Java est un mécanisme "statique" assez complexe
- la généricité existe dans d'autres langages (exemple C++ et Ada) (mais de façon différente)

Exemple: File

```
public class Cellule<E>{
    private Cellule<E> suivant;
    private E element;
    public Cellule(E val) {
        this.element=val;
    }
    public Cellule(E val, Cellule suivant){
        this.element=val; this.suivant=suivant;
    }
    public E getElement(){ return element;}
    public void setElement(E v){
        element=v;
    }
    public Cellule<E> getSuivant(){ return suivant;}
    public void setSuivant(Cellule<E> s){
        this.suivant=s;
    }
}
```

Suite

```
class File<E>{
    protected Cellule<E> tete;
    protected Cellule<E> queue;
    private int taille=0;
    public boolean estVide(){
        return taille==0;
    }
    public void enfiler(E item){
        Cellule<E> c=new Cellule<E>(item);
        if (estVide())
            tete=queue=c;
        else{
            queue.setSuivant(c);
            queue=c;
        }
        taille++;
    } //..
```

suite

```
    public E defiler(){
         if (estVide())
             return null;
         Cellule<E> tmp=tete;
         tete=tete.getSuivant();
         taille--;
         return tmp.getElement();
     }
      public int getTaille(){
         return taille;
     }
}
```

Usage

```
Cellule<Integer> cel=new Cellule<Integer>(23);
File<Integer> fi=new File<Integer>();
File<String> fs=new File<String>();
File<Object> fobj=new File<Object>();
String[] st={"zéro","un","deux",
             "trois","quatre","cinq"};
for(int i=0;i<st.length;i++){
     fs.enfiler(st[i]);
     fi.enfiler(i);
 }
```

Remarques

- Une déclaration de type générique peut avoir plusieurs paramètres:
  - Map<K,V>
- Contrôle de type
  - fs.enfiler(4) est refusé à la compilation

Types génériques, pourquoi?

- Vérification de type:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

Et:

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
x=myIntList.iterator().next();
```

Invocation et type en paramètre

```
public interface List <E>{
        void add(E x);
        Iterator<E> iterator();
}
public interface Iterator<E>{ E next();    boolean hasNext();}
```

List<Integer> pourrait correspondre à (comme en C++):

```
public interface IntegerList {
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

Mais… une déclaration d'un type générique crée un vrai type (qui est compilé comme un tout) et il n'y a pas de type pour List<Integer>

Typage

- Une invocation ne crée pas un nouveau type:
  - `(fs.getClass()==fi.getClass())` est vrai
  - la classe est ici `File`
  - il s'agit surtout d'un contrôle (effectué à la compilation)
  - à l'exécution `fi` n'a plus aucune information sur quelle invocation a permis sa construction

Conséquences

- Aucune instanciation n'est possible pour un type argument
  - Dans l'exemple: `E v=new E();` est impossible
  - Pas de tableau de `E`

Exemple

```
public E[] toArray(File<E> f){
      E[] tab=new E[f.getTaille()]; //non
      for(int i=0;i<f.getTaille();i++)
                tab[i]=f.defiler();
}
```

- Comment construire un tableau sans connaître le type de base?
- La classe `Array` et la méthode `Array.newInstance()` permettraient de résoudre ce problème (mais sans contrôle de type)
- On peut aussi utiliser la classe `Object`.

Object

```
 public static <E> Object[] toArray(File<E> f){
  Object[] tab=new  Object[f.getTaille()];
  for(int i=0;i<f.getTaille();i++)
      tab[i]=f.defiler();
  return tab;
}
```

mais on perd l'avantage du contrôle de type.

Contrôle du type

- Pourtant, on peut passer un objet d'un type avec paramètre à une méthode.
- Comment se fait le passage des paramètres?
  - le compilateur passe le type le plus général (`Object`) et utilise le cast pour assurer le contrôle du typage.

Chapitre VII

1. Principes généraux
2. Types génériques imbriqués
3. Types paramètres bornés
4. Méthodes génériques

Types génériques imbriqués

```
public class FileSimpleChainageb <E>{
    public class Cellule{
        private Cellule suivant;
        private E element;
        public Cellule(E val) {
            this.element=val;
        }
        public Cellule(E val, Cellule suivant){
            this.element=val;
            this.suivant=suivant;
        }
        public E getElement(){
            return element;
        }
        public void setElement(E v){
            element=v;
        }//...
```

Suite

```
public Cellule getSuivant(){
          return suivant;
     }
public void setSuivant(Cellule s){
          this.suivant=s;
     }
 }
 protected Cellule tete;
 protected Cellule queue;
 private int taille=0;
 public boolean estVide(){
     return taille==0;
 }
 public int getTaille(){
     return taille;
 }
```

Fin…

```
    public void enfiler(E item){
        Cellule c=new Cellule(item);
        if (estVide())
            tete=queue=c;
        else{
            queue.setSuivant(c);
            queue=c;
        }
        taille++;
    }
    public E defiler(){
        if (estVide())
            return null;
        Cellule tmp=tete;
        tete=tete.getSuivant();
        taille--;
        return tmp.getElement();
    }
}
```

Chapitre VII

1. Principes généraux
2. Types génériques imbriqués
3. Types en paramètres bornés
4. Méthodes génériques

Sous-typage

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; //1
lo.add(new Object());//2
String s = ls.get(0); //3 !
```

Si A est une extension de B, F<A> n'est pas une extension de F<B>:
//1 est interdit

Pour les tableaux:

- si A est une extension de B un tableau de A est une extension de tableau de B.
- //1 est autorisé, mais ensuite //2 est interdit

Joker '?'

```
void printCollection(Collection<Object> c) {
    for (Object e : c) { System.out.println(e);}
}
```

Ne fonctionne pas avec une Collection<Integer>

Une collection de n'importe quoi ('?')

```
void printCollection(Collection<?> c) {
    for (Object e : c){ System.out.println(e);}
}
```

est possible (n'importe quoi est un objet).

Mais

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // erreur compilation
```

Mais

- Ce n'est pas suffisant…
  - On peut vouloir borner le type paramètre: comparable est une interface générique qui indique la possibilité de comparer des objets

    ```
    class valeur implements Comparable<Valeur>{..}
    ```
  - Une SortedCollection est construite sur des classes E qui implémentent Comparable<E> d'où:

```
interface SortedCollection<E extends Comparable<E>>{}
```

Exemple

```
static double somme(List<Number> l){
    double res=0.0;
    for(Number n:l)
        res+=n.doubleValue();
    return res;
}
public static void main(String[] st){
    List<Integer> l= new ArrayList<Integer>();
    for(int i=0;i<10;i++)l.add(i);
    double s=somme(l); //incorrect
}
```

Mais

Type paramètre borné

- Au moins un number:
- List<? extends Number> une liste constituée de n'importe quel type dérivé de Number

```
static double somme2(List<? extends Number> l){
      double res=0.0;
      for(Number n:l)
          res+=n.doubleValue();
      return res;
  }
```

Types bornés

- `List<? extends Number>`
  - indique que le type doit au moins être un `Number` (tout type qui dérive de `Number`)
  - *borné par le bas* : au moins un `Number`
- On peut aussi imposer que le type soit une superclasse d'un autre type
  - `List<? super Integer>`
  - *borné par le haut* : au plus un `Integer` (super-classe de `Integer`)

Sous-typage

- List<?> est une super classe de List<Object>

Et:

- List<?>
  - List<? extends Number>
    - List<Number>
    - List<? extends Integer>
      - List<Integer>

Types en paramètres bornés

- Exemple:
  - `SortedCollection` est composée d'éléments du type `E` et ces éléments peuvent être comparés.

    Mais `<E extends comparable<E>>` est trop fort:

    il suffit que `E extends Comparable<T>` pour `T` égal à `E` ou `T` superclasse de `E`

    (si `E extends Comparable<Object>` a fortiori on peut comparer les éléments de E) d'où:

    ```
    <E extends Comparable<? super E>>
    ```

Mais attention

```
File<?> str=new File<String>();
str.enfiler("un");
```

provoque une erreur à la compilation:

```
enfiler(capture of ?) in generique.File<capture of ?>
   cannot be applied to (java.lang.String)
```

de même:

```
File<? extends Number> num=new File<Number>();
num.enfiler(Integer.valueOf(12));
```

en effet File<? extends Number> peut être par exemple une File d'un type dérivé de Number.

Par contre:

```
File<? super Number> num=new File<Number>();
num.enfiler(Integer.valueOf(12));
```

est correct

Joker '?'

```
enfiler(capture of ?) in generique.File<capture of ?
    > cannot be applied to (java.lang.String)
```

- signifie que le type de str est File<capture of ?> qui n'est pas compatible avec String

Quelques explications

- ? peut correspondre à n'importe quel type enfiler(a) où a est de type A ne peut fonctionner si le type correspondant à ? est dérivé de A
- de même ? dans <? extends X> ne peut fonctionner car si ? est Y dérivé de X il faut un paramètre d'une classe dérivée de Y
- par contre ? dans <? super X> ne pouvant correspondre qu'à une classe "avant" X, tout Z dérivé de X fonctionne

Mais

- inversement pour la valeur retournée (avec la méthode défiler par exemple)
  - pour <?> quelque soit le type X correspondant on peut l'affecter à Object et à X
  - idem pour <? extends X>
  - mais pour <? super Y> si Z correspond à ? pour T un type quelconque on ne peut savoir si T peut être affecté par un Z

Chapitre VII

1. Principes généraux
2. Types génériques imbriqués
3. Types paramètres bornés
4. Méthodes génériques

Méthodes génériques

- Supposons que l'on veuille convertir en tableau une File de E
  - on a vu précédemment que l'on ne pouvait ni instancier un objet E ni créer un tableau de E
  - on peut cependant passer un tableau de la taille appropriée à une méthode qui retourne ce tableau:

enTableau1

```
public E[] enTableau1(E[] tab){
     Object[] tmp = tab;
     int i=0;
     for(Cellule<E> c= tete; c != null && i< tab.length;
     c=c.getSuivant())
          tab[i++] = c.getElement();
     return tab;
   }
```

- enTableau1 est une nouvelle méthode de File:

```
File<String> fs=new File<String>();
String[]  u;
u=fs.enTableau1(new String[fs.getTaille()]);
```

enTableau

- Mais,
  - il faut que le tableau passé en paramètre soit un tableau de E, alors qu'un tableau d'une super-classe de E devrait fonctionner (si F est une superclasse de E un tableau de F peut contenir des objets E).
  - avec une méthode générique:

enTableau

```
public <T> T[] enTableau(T[] tab){
    Object[] tmp = tab;
    int i=0;
    for(Cellule<E> c= tete; c != null && i< tab.length;
    c=c.getSuivant())
        tmp[i++] = c.getElement();
    return tab;
}
```

- la déclaration impose que le type du tableau retourné soit du type du tableau de l'argument
- Notons que `tmp` est un tableau d'`Object` ce qui est nécessaire pour le `getSuivant`
- Notons que normalement il faudrait que `T` soit une superclasse de `E` (à l'exécution il peut y avoir une erreur).
- Notons enfin que `'T'` ne sert pas dans le corps de la méthode.

Remarque

```
public <T> T[] enTableaubis(T[] tab){
        int i=0;
        for(Cellule<E> c= tete;
             c != null && i< tab.length;
            c=c.getSuivant())
            tab[i++] = (T)c.getElement();
        return tab;
}
```

- provoque un warning `"Cellule.java uses unchecked or unsafe operations".`
- (l'"effacement" ne permet pas de vérifier le type)

Avec Reflection…

- Une autre solution peut être, si on veut créer un vrai tableau, d'utiliser Array.newInstance de la classe: java.lang.reflect

Exemple avec Reflection

```
public E[] enTableau2(Class<E> type){
        int taille = getTaille();
        E[] arr=(E[])Array.newInstance(type,taille);
        int i=0;
        for(Cellule<E> c= tete; c != null && i< taille;
        c=c.getSuivant())
            arr[i++] = c.getElement();
        return arr;
}
```

- on crée ainsi un tableau de "E"
- "unchecked warning": le cast (E[]) n'a pas le sens usuel
- pour fs déclaré comme précédemment on aura:

```
String[] u=fs.enTableau2(String.class); //ok
Object[] v=fs.enTableau2(Object.class); //non
```

- car le type doit être exact

Avec une méthode générique

```
public <T> T[] enTableau3(Class<T> type){
     int taille = getTaille();
     T[] arr=(T[])Array.newInstance(type,taille);
     int i=0;
     Object[] tmp=arr;
     for(Cellule<E> c= tete; c != null && i< taille;
     c=c.getSuivant())
         tmp[i++] = c.getElement();
     return arr;
 }
```

Inférence de type

- Comment invoquer une méthode générique?
- Exemple:

```
static <T> T identite(T obj){
        return obj;
    }
```

Invocations

- On peut explicitement préciser le type:

```
String s1="Bonjour";
String s2= Main.<String>identite(s1);
```

- Mais le compilateur peut, lui-même, trouver le type le plus spécifique:

```
String s1=identite("Bonjour");
```

- On aura:

```
Object o1=identite(s1);           //ok
Object o2=identite((Object)s1);   //ok
s2=identite((Object) s1);         //non!!!
s2=(String)identite((Object) s1);//ok
```

Comment ça marche?

- Mécanisme de l'effacement ("erasure")
- Pour chaque type générique il n'y a qu'une classe: Cellule<String> et Cellule<Integer> ont la même classe
- Effacement:
  - Cellule<String> -> Cellule
    - Cellule est un type *brut*
  - Pour une variable type:
    - <E> -> Object
    - <E extends Number> -> Number
- Le compilateur remplace chaque variable type par son effacement

Comment ça marche?

- Si le résultat de l'effacement du générique ne correspond pas à la variable type, le compilateur génère un cast:
  - par effacement le type variable de File<E> est Object
  - pour un "defiler" sur un objet File<String> le compilateur insère un cast sur String

Comment ça marche?

- A cause de l'effacement, rien de ce qui nécessite de connaître la valeur d'un argument type n'est autorisé. Par exemple:
  - on ne peut pas instancier un type en paramètre: pas de new T() ou de new T[]
  - on ne peut pas utiliser `instanceof` pour une instance de type paramétré
  - on ne peut pas créer de tableau sur un type paramétré sauf s'il est non borné new `List<String>[10]` est interdit mais new `List<?>[10]` est possible.

Comment ça marche?

- les "cast" sont possibles mais n'ont pas la même signification que pour les types non paramétrés:
  - le cast est remplacé par un cast vers le type obtenu par effacement et génère un "unchecked warning" à la compilation.
  - Exemple:
    - on peut caster paramètre File<?> vers un File<String> pour un enfiler (ce qui génère le warning)
    - A l'exécution si le type effectif est File<Number> cela passe… mais le defiler provoquera un ClassCastException.

Comment ça marche?

- Exemple:

```
List<String> l=new ArrayList<String>();
Object o=identite(l);
List<String> l1=(List<String>)o;// warning
List<String> l2=(List)o;//warning
List<?> l3=(List) o; //ok
List<?> l4=(List<?>)o; //ok
```

Application: surcharge

- avoir la même signature s'étend aux méthodes avec variables types
- même signature pour des variables types = même type et même borne (modulo bien sûr renommage!)
- signatures équivalentes par annulation: mêmes signatures où l'effacement des signatures sont identiques

Surcharge

```
class Base<T>{
    void m(int x){};
    void m(T t){};
    void m(String s){};
    <N extends Number> void m(N n){};
    void m(File<?> q){};
}
-----
m(int)
m(Object)
m(String)
m(Number)
m(File)
```

Et héritage…

- exemple:

```
class D<T> extends Base<T>{
    void m(Integer i){} //nouveau
    void m(Object t){}  // redéfinit m(T t)
    void m(Number n){}   // redéfinit m(N n)
}
```

Chapitre VIII

Strings,

Plan

- A) String
- B) Expressions régulières
- C) Chaînes et tableaux (char et byte)
- D) Chaînes modifiables

String

- Une `String` est une chaîne de caractères *non modifiable*
  - (attention les caractères en java sont en Unicode)
- `StringBuilder` et `StringBuffer` sont des classes de chaînes qui peuvent être modifiées.
- `String` `StringBuilder` et `StringBuffer` implémentent l'interface `CharSequence`

CharSequence

- Interface:
  - char **charAt**(int index)
    Retourne le char the char en position index.
  - int **length**()
    Retourne la longueur de la séquence.
  - CharSequence **subSequence**(int start, int end)
    Retourne une sous CharSequence
  - String **toString**()
    Retourne la string correspondant à la séquence

String

- De nombreuses méthodes pour manipuler les chaines (attention un objet String n'est pas modifiable)
  - constructeurs
  - indexOf, lastIndexOf retourne la première (dernière) position
  - conversion valueOf(valeur d'un type primitif)
  - replace, trim, split
  - toLowerCase, touppercase()
  - ...

Comparaison

- '==' ne compare les contenus, MAIS deux littéraux ayant le même contenu sont identiques:

```
String st1="bonjour";
String st2="bonjour"
if(st1==st2)System.out.println("égal");
else System.out.println("différent");
```

donnera *égal*

La méthode `intern` permet de retourner un String de même référence

Manipulations sur les chaînes

- char **charAt**(int index)
- int **compareTo**(String anotherString) comparaison lexicographique
- boolean **contains**(CharSequence s)
- boolean **equals**(Object anObject)
- int **length**()
- boolean **matches**(String regex)
- boolean **regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len)
- boolean **regionMatches**(boolean ignoreCase, int toffset, String other, int ooffset, int len)
- String **replace**(CharSequence target, CharSequence replacement)
- String[] **split**(String regex)
- String **trim**()

Exemples

```
String string = "Madam, I am Adam";
 boolean  b = string.startsWith("Mad");  // true
 b = string.endsWith("dam");             // true
 b = string.indexOf("I am") > 0;         // true
 b = string.matches("(?i)mad.*");
 b = string.matches("(?i).*adam");
 b = string.matches("(?i).*i am.*");
```

Exemple: remplacement

```
static String replace(String str, String pattern,
                      String replace) {
    int s = 0;
    int e = 0;
    StringBuffer result = new StringBuffer();

    while ((e = str.indexOf(pattern, s)) >= 0) {
        result.append(str.substring(s, e));
        result.append(replace);
        s = e+pattern.length();
    }
    result.append(str.substring(s));
    return result.toString();
}
```

Exemple

```
Strinf st1="bonjour";
String st3=new String("bonjour");
if(st1==st3)System.out.println("égal");
else System.out.println("différent");
String st4=st3.intern();
if(st1==st4)System.out.println("égal");
else System.out.println("différent");
```

(la comparaison des *références* est bien sûr moins coûteuse que la comparaison des contenus)

(de plus dans tous les cas des chaînes de même contenus ont le *même* hashcode)

Expressions régulières

- Les expressions régulières permettent de définir un motif (pattern) objet de la classe Pattern.
- Un motif est créé par la méthode compile
  - Pattern pat = Pattern.compile("[a-z]*")

Expressions régulières

- la syntaxe des expressions régulières (rationnelles) est assez large:
  - caractères
    - exemples \t , a ,\xhh ,
  - classes de caractères
    - exemples [a-z], [^a-z], [a-z&&0-9]
  - classes de caractères prédéfinis
    - exemples . , \d, \D, \w, \W, \s, \S
  - classes prédéfinies ASCII, Character
    - exemples \p{Blank}, \p{javaLowerCase}
  - bords
    - exemples ^, $,\b (bord d'un mot)
  - itérations
    - exemples [a-z]?, [1-9][0-9]+, \W*,
  - capture et restitution
    - (X) définit la capture de X
    - \n correspond à la n-ième capture

Recherche de motif

- Principe:

```
Pattern pat=Pattern.compile(regex);
Matcher matcher=pat.match(entrée);
boolean trouve = matcher.find();
```

la classe Matcher contient les méthodes pour chercher (find()), retourner la sous-séquence trouvée (group())

Exemple: rechercher

```
String patternStr = "b";
Pattern pattern = Pattern.compile(patternStr);
CharSequence inputStr = "a b c b";
Matcher matcher = pattern.matcher(inputStr);
boolean matchFound = matcher.find();    // true
String match = matcher.group();         // b
int start = matcher.start();            // 2
int end = matcher.end();                // 3
matchFound = matcher.find();            // true
```

Remplacer

```
CharSequence inputStr = "ab12 cd efg34";
String patternStr = "([a-zA-Z]+[0-9]+)";
Pattern pattern = Pattern.compile(patternStr);
Matcher matcher = pattern.matcher(inputStr);
// Remplacer toutes les occurrences
StringBuffer buf = new StringBuffer();
boolean found = false;
while ((found = matcher.find())) {
    String replaceStr = matcher.group();
    replaceStr = replaceStr.toUpperCase();
    matcher.appendReplacement(buf, replaceStr);
}
matcher.appendTail(buf);
String result = buf.toString();
// AB12 cd EFG34
```

Chaines et tableaux de char

- Une string n'est pas un tableau de char mais on peut passer de l'un à l'autre
- constructeurs

  String(char[] value)

  String(char[] value, int offset, int count)
- méthode:

  void getChars(int srcBegin, int srcEnd,
  char[] dst, int dstBegin)

Chaînes et tableaux de byte

- les chaines contiennent des caractères codés en UTF-16.
- On peut convertir ces caractères en byte suivant un codage
- De même on peut coder des bytes en caractères unicode.
- (par exemple un byte Latin-1 se code en Unicode en ajoutant un octet de 0)

Charset

- la classe Charset permet de faire correspondre des séquences d'unicode et des bytes. En standard:
  - US-ASCII
  - ISO-8859-1
  - UTF-8
  - UTF-16BE
  - UTF-16LE
  - UTF-16

tableau de byte

- constructeurs:
  - `String(byte[] bytes)` (conversion suivant le jeu de caractère par défaut)
  - `String(byte[] bytes, int offset, int length)`
  - `String(byte[] bytes, String charsetName)` (suivant le charset)
  - `String(byte[] bytes, int offset, int length, String charsetName)` (suivant le charset)
- méthodes
  - `byte[]getBytes()`
  - `byte[]getBytes(String charsetName)`

Exemple:

```
 try {
    // Conversion Unicode en x-MacRoman
    String string = "abcéçùà\u563b";
    System.out.println(string);
    byte[] mac = string.getBytes("x-MacRoman");
    System.out.println(new String(mac));
    // Conversion x-MacRoman vers Unicode
    string = new String(utf8, "x-MacRoman");
    System.out.println(string);
} catch (UnsupportedEncodingException e) {
}
```

- abcéçùà?
- abcŽ??^?
- abcéçùà?

Exemples:

```
for(String nom: Charset.availableCharsets().keySet())
        System.out.println(nom);
```

affichera la liste des jeux de caractères:

- EUC-JP
- EUC-KR
- GB18030
- GB2312
- GBK
- IBM-Thai
- IBM00858
- IBM01140
- IBM01141
- IBM01142
- IBM01143
- IBM01144
- IBM01145
- IBM01146
- IBM01147
- IBM01148
- …

Exemple

```
Charset charset = Charset.forName("ISO-8859-1");
CharsetDecoder decoder = charset.newDecoder();
CharsetEncoder encoder = charset.newEncoder();
try {
    // Convertit une string vers ISO-LATIN-1
    ByteBuffer bbuf =
        encoder.encode(CharBuffer.wrap("une chaîne"));

    // Convertit ISO-LATIN-1 bytes en string.
    CharBuffer cbuf = decoder.decode(bbuf);
    String s = cbuf.toString();
} catch (CharacterCodingException e) {
}
```

Exemple:

```
try {
        // Conversion vers ISO-LATIN-1 de bytes
        ByteBuffer bbuf =
  encoder.encode(CharBuffer.wrap("une chaîne"));

        // Conversion de ISO-LATIN-1 vers bytes
        CharBuffer cbuf = decoder.decode(bbuf);
        String s = cbuf.toString();
        System.out.println(s);
    } catch (CharacterCodingException e) {
    }
```

StringBuilder

- La classe `StringBuilder` peut contenir des chaînes qui peuvent être modifiées.
- Il s'agit d'une structure de données dynamique: la taille de la chaîne est augmentée automatiquement
- La taille initiale peut être précisée dans le constructeur (16 par défaut).
- (Il existe aussi une classe `StringBuffer` qui est "thread-safe")
- méthodes insert, append, delete

Chapitre IX

Entrées-sorties

Principes généraux

- entrées sorties
  - streams dans **java.io**
  - channels dans **java.nio** ("n" pour non-blocking)

- streams: séquences de données ordonnées avec une source (stream d'entrée) ou une destination (stream de sortie)
- channels et buffer. Buffer contient les données et le channel correspond à des connections

Streams

- Deux grandes sortes de Stream
  - character Streams contiennet des caractères UTF-16
  - byte Streams contiennent des octets
- Character versus byte
  - imput ou output stream pour les byte
  - reader, writer pour les character
  - deux hiérarchies qui se correspondent

Hiérarchie

- java.io.**InputStream** (implements java.io.*Closeable*)
  - java.io.**ByteArrayInputStream**
  - java.io.**FileInputStream**
  - java.io.**FilterInputStream**
    - java.io.**BufferedInputStream**
    - java.io.**DataInputStream** (implements java.io.*DataInput*)
    - java.io.**LineNumberInputStream**
    - java.io.**PushbackInputStream**
  - java.io.**ObjectInputStream** (implements java.io.*ObjectInput*, java.io.*ObjectStreamConstants*)
  - java.io.**PipedInputStream**
  - java.io.**SequenceInputStream**
  - java.io.**StringBufferInputStream**

Hiérarchie

- java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)
  - java.io.**ByteArrayOutputStream**
  - java.io.**FileOutputStream**
  - java.io.**FilterOutputStream**
    - java.io.**BufferedOutputStream**
    - java.io.**DataOutputStream** (implements java.io.DataOutput)
    - java.io.**PrintStream** (implements java.lang.Appendable, java.io.Closeable)
  - java.io.**ObjectOutputStream** (implements java.io.ObjectOutput, java.io.ObjectStreamConstants)
  - java.io.**PipedOutputStream**

Hiérachie…

- java.io.**Reader** (implements java.io.Closeable, java.lang.Readable)
  - java.io.**BufferedReader**
    - java.io.**LineNumberReader**
  - java.io.**CharArrayReader**
  - java.io.**FilterReader**
    - java.io.**PushbackReader**
  - java.io.**InputStreamReader**
    - java.io.**FileReader**
  - java.io.**PipedReader**
  - java.io.**StringReader**

Hiérarchie

- java.io.**Writer** (implements java.lang.Appendable, java.io.Closeable, java.io.Flushable)
  - java.io.**BufferedWriter**
  - java.io.**CharArrayWriter**
  - java.io.**FilterWriter**
  - java.io.**OutputStreamWriter**
    - java.io.**FileWriter**
  - java.io.**PipedWriter**
  - java.io.**PrintWriter**
  - java.io.**StringWriter**

Streams d'octets

- Class InputStream
  (toutes ces méthodes peuvent lancer IOEXception)
  - abstract int **read**() lit un octet
  - int **read** (byte[] b) lit et écrit dans b, (le nombre d'octets lus dépend de b et est retourné
  - int **read**(byte[] b, int off, int len)
  - long **skip**(long n)
  - int **available**() n d'octets pouvant être lus
  - void **mark**(int readlimit) et void **reset**() pose d'une marque et retour à la marque
  - void**close**()

Stream d'octets

- OutputStream

  (toutes ces méthodes peuvent lancer IOEXception)

  - abstract void **write**(int b) écrit un octet
  - void **write**(byte[] b)
  - void **write**(byte[] b, int off, int len)
  - void**close**()
  - void**flush**()

Exemples

```
public static int compteIS(String st)
        throws IOException{
    InputStream in;
    int n=0;
    if (st==null)in=System.in;
    else
        in= new FileInputStream(st);
    for(; in.read() != -1;n++);
    return n;
 }
```

Exemples

```
public static void trOS(String f, char from, char to){
    int b;
    OutputStream out=null;
    try{
      if(f==null) out=System.out;
       else
        out=new FileOutputStream(f);
        while((b= System.in.read())!=-1)
            out.write(b==from? to:b);
        out.flush();
        out.close();
    }catch(IOException ex){
            ex.printStackTrace();
    }
}
```

Reader-Writer

- Reader et Writer sont les deux classes abstraites pour les streams de caractères:
  - le read de InputStream retourne un byte comme octet de poids faible d'un int alors que le read de Reader retourne un char à partir de 2 octets de poids faible d'un int

Reader

- essentiellement les mêmes méthodes que pour InputStream:
  - int **read**()
  - int **read**(char[] cbuf)
  - abstract int **read**(char[] cbuf, int off, int len)
  - int **read**(CharBuffer target)
  - boolean **ready**() si prêt à lire
  - void**reset**()
  - void**mark**(int readAheadLimit)
  - boolean**markSupported**()
  - long**skip**(long n)

Writer

- similaire à OutputStream mais avec des caractères au lieu d'octets.
  - void **write**(char[] cbuf)
  - abstract void **write**(char[] cbuf, int off, int len)
  - void **write**(int c)
  - void **write**(int c)
  - void **write**(String str)
  - void **write**(String str, int off, int len)
  - Writer **append**(char c)
  - Writer **append**(CharSequence csq)
  - Writer **append**(CharSequence csq, int start, int end)
  - abstract void **close**()
  - abstract void **flush**()

Exemples

```
public static int compteR(String st)
        throws IOException{
   Reader in;
   int n=0;
   if (st==null)
    in=new InputStreamReader(System.in);
   else
    in= new FileReader(st);
   for(; in.read() != -1;n++);
   return n;
}
```

Exemples

```
public static void trWr(String f, char from, char to){
    int b;
    Writer out=null;
    try {
        if(f==null)
            out= new OutputStreamWriter(System.out);
        else
            out=new FileWriter(f);
        while((b= System.in.read())!=-1)
                out.write(b==from? to:b);
        out.flush();
        out.close();
   }catch(IOException e){System.out.println(e);}
}
```

Remarques

- les streams standard System.in et System.out sont des streams d'octets
- InputStreamReader permet de passer de InputStream à Reader
- System.in et System.out sont des PrintStream (obsolète à remplacer par la version caractère PrintWriter)

InputStreamReader et OutputStreamWriter

- conversion entre caractères et octets, la conversion est donnée par un charset
- Constructeurs:
  - InputStreamReader(InputStream in)
  - InputStreamReader(InputStream in, Charset cs)
  - InputStreamReader(InputStream in, String charsetName)
  - OutputStreamWriter(OutputStream out)
  - OutputStreamWriter(OutputStream out, Charset cs)
  - OutputStreamWriter(OutputStream out, CharsetEncoder enc)
  - OutputStreamWriter(OutputStream out, String charsetName)

Autres classes

- FileInputStream
- FileOutputStream
- FileReader
- FileWriter
- Ces classes permettent d'associer une stream à un fichier donné par son nom (String) par un objet File ou un objet FileDescriptor (un mode append peut être défini pour les ecritures)

Quelques autres classes

- Les filtres
  - FilterInputStream
  - FilterOutputStream
  - FilterReader
  - FilterWriter
- Buffered
  - BufferedInputStream
  - BufferedOutputStream
  - BufferedReader
  - BufferedWriter
- Tubes
  - PipedInputStream
  - PipedOutputStream
  - PipedReader
  - PipedWriter

Exemple d'un filtre

```
class conversionMaj extends FilterReader{
    public conversionMaj(Reader in){
        super(in);
    }
    public int read() throws IOException{
        int c=super.read();
        return(c==-1?c:Character.toUpperCase((char)c));
    }
    public int read(char[] buf, int offset, int count)throws
    IOException{
        int nread=super.read(buf,offset,count);
        int last=offset+nread;
        for(int i=offset; i<last;i++)
            buf[i] = Character.toUpperCase(buf[i]);
        return nread;
    }
}
```

Exemple suite:

```
StringReader source=new StringReader("ma chaîne");
FilterReader filtre=new conversionMaj(source);
int c;
try{
  while((c=filtre.read())!= -1)
      System.out.print((char)c);
  }catch(IOException e){
  e.printStackTrace(System.err);
 }
```

Pipe (tube)

- Un tube associe une entrée et un sortie: on lit à une extrémité et on écrit à l'autre.

Exemple pipe

```
class PipeExemple extends Thread{
    private Writer out;
    private Reader in;
    public PipeExemple(Writer out,Reader in){
        this.out=out;
        this.in=in;
    }
    public void run(){
        int c;
        try{
            try{
                while ((c=in.read())!=-1){
                    out.write(Character.toUpperCase((char)c));
                }
            }finally{out.close();}
        }catch(IOException e){e.printStackTrace(System.err);}
    }
}
```

Suite

```
PipedWriter out1=new PipedWriter();
PipedReader in1=new PipedReader(out1);
PipedWriter out2=new PipedWriter();
PipedReader in2=new PipedReader(out2);
PipeExemple pipe=new PipeExemple(out1, in2);
pipe.start();
try{
    for(char c='a';c<='z';c++) {
        out2.write(c);
        System.out.print((char)(in1.read()));
    }
}finally {
    out2.close();
}
```

ByteArray, String…

- java.io.**InputStream** (implements java.io.Closeable)
  - java.io.**ByteArrayInputStream**
  - java.io.**StringBufferInputStream**
- java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)
  - java.io.**ByteArrayOutputStream**
- java.io.**Reader** (implements java.io.Closeable, java.lang.Readable)
  - java.io.**CharArrayReader**
  - java.io.**StringReader**
- java.io.**Writer** (implements java.lang.Appendable, java.io.Closeable, java.io.Flushable)
  - java.io.**CharArrayWriter**
  - java.io.**StringWriter**

Print Streams

- java.io.**PrintStream** (implements java.lang.Appendable, java.io.Closeable)
- java.io.**PrintWriter**
  - méthode println()

Streams pour les données

- DataInput: interface pour lire des bytes d'une stream binaire et les transformer en données java de type primitif
- DataOutput: interface pour convertir des valeurs de type primitif en stream binaire.

Sérialisation

- sauvegarder des objets java en flot d'octets:
  - objet vers byte: sérialisation
  - byte vers objet: désérialisation
- java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
- java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)

Sauver des objets

- la serialisation-désérialisation doit permettre de sauvegarder et restituer des objets.
- sauver et restituer des objets n'est pas si simple. Pourquoi?
- interface serializable
- (alternative XML)

Manipuler des fichiers…

- java.io.**File** (implements java.lang.Comparable<T>, java.io.Serializable)

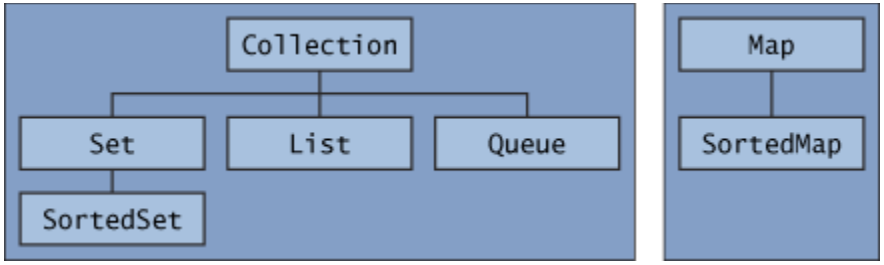
nio

- entrée sorties de haute performance avec contrôle sur les buffers.

Collections

Collections

- types de données
  - interfaces
  - implémentations
  - algorithmes
- Interfaces:



Collections: les interfaces

Les collections sont des interfaces génériques

- Collection<E>: add, remove size toArray…
  - Set<E>: éléments sans duplication
    - SortedSet<E>: ensembles ordonnés
  - List<E>: des listes éléments non ordonnés et avec duplication
  - Queue<E>: files avec tête: peek, poll (défiler), offer (enfiler)
- Map<K,V>:association clés valeurs
- SortedMap<K,V> avec clés triées

Certaines méthodes sont *optionnelles* (si elles ne sont pas implémentées UnsupportedOperationException).

En plus:

- Iterator<E>: interface qui retourne successivement les éléments next(), hasNext(), remove()
- ListIterator<E>: itérateur pour des List, set(E) previous, add(E)

Collection

```
public interface Collection<E> extends Iterable<E> {
    // operations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optionnel
    boolean remove(Object element); //optionnel
    Iterator<E> iterator();

    // operations des collections
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optionnel
    boolean removeAll(Collection<?> c);        //optionnel
    boolean retainAll(Collection<?> c);        //optionnel
    void clear();                              //optionnel

    // Array
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Collection

- Les collections sont génériques
- Parcours:
  - On peut parcourir les éléments par « for »:

```
for (Object o : collection)
    System.out.println(o);
```

  - Ou avec un Iterator:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Collection

- On peut convertir une collection en tableau
  - En tableaux de Object
  - En tableaux d'objet du type paramètre de la collection
- Il existe aussi une *classe* Collections qui contient des méthodes statiques utiles

Set

- Interface pour contenir des objets différents
  - Opérations ensemblistes
  - SortedSet pour des ensembles ordonnés
- Implémentations:
  - HashSet par hachage (preformances)
  - TreeSet arbre rouge-noir
  - LinkedHashSet ordonnés par ordre d'insertion

Set

```
public interface Set<E> extends Collection<E> {
    // opérations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optionnel
    boolean remove(Object element); //optionnel
    Iterator<E> iterator();

    // autres
    boolean containsAll(Collection<?> c); // sous-ensemble
    boolean addAll(Collection<? extends E> c); //optionnel- union
    boolean removeAll(Collection<?> c);        //optionnel- différence
    boolean retainAll(Collection<?> c);        //optionnel- intersection
    void clear();                              //optionnel

    // Array
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Exemple:

```
public static void chercheDoublons(String ... st){
       Set<String> s = new HashSet<String>();
       for (String a : st)
           if (!s.add(a))
               System.out.println("Doublon: " + a);

       System.out.println("il y a "+s.size() + " mots différents: " + s);
}
public static void chercheDoublonsbis(String st[]){
       Set<String> s=new HashSet<String>();
       Set<String> sdup=new HashSet<String>();
       for(String a :st)
           if (!s.add(a))
               sdup.add(a);
       s.removeAll(sdup);
       System.out.println("Mots uniques:    " + s);
       System.out.println("Mots dupliqués: " + sdup);
 }
```

Lists

- En plus de Collection:
  - Accès par position de l ' élément
  - Recherche qui retourne la position de l' élément
  - Sous-liste entre deux positions
- Implémentations:
  - ArrayList
  - LinkedList

List

```
public interface List<E> extends Collection<E> {
    // accès par position
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // recherche
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // sous-liste
    List<E> subList(int from, int to);
}
```

Itérateur pour listes

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

Exemple

```
public static <E> void swap(List<E> a, int i, int j) {
        E tmp = a.get(i);
        a.set(i, a.get(j));
        a.set(j, tmp);
}
public static void melange(List<?> list, Random rnd) {
        for (int i = list.size(); i > 1; i--)
            swap(list, i - 1, rnd.nextInt(i));
}
```

Suite…

```
public static <E> List<E> uneMain(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
}
public static void distribuer(int nMains, int nCartes) {
        String[] couleurs = new String[]{"pique","coeur","carreau","trèfle"};
        String[] rank = new String[]
        {"as","2","3","4","5","6","7","8", "9","10","valet","dame","roi"};
        List<String> deck = new ArrayList<String>();
        for (int i = 0; i < couleurs.length; i++)
            for (int j = 0; j < rank.length; j++)
                deck.add(rank[j] + " de " + couleurs[i]);
        melange(deck,new Random());
        for (int i=0; i < nMains; i++)
            System.out.println(uneMain(deck,nCartes));
    }
```

Map

- Map associe des clés à des valeurs
  - Association injective: à une clé correspond exactement une valeur.
  - Trois implémentations, comme pour set
    - HashMap,
    - TreeMap,
    - LinkedHashMap
  - Remplace Hash

Map

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Exemples

```
 public static void mapFreq(String ... t) {
        Map<String, Integer> m = new HashMap<String,
                                              Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Exemples

```
 public static void mapFreq(String ... t) {
        Map<String, Integer> m = new TreeMap<String,
                                              Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Exemples

```
 public static void mapFreq(String ... t) {
        Map<String, Integer> m = new LinkedHashMap<String,
                                                  Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Queue

- Pour représenter une file (en principe FIFO):
  - Insertion: offer -add
  - Extraction: poll - remove
  - Pour voir: peek –element
  - (retourne une valeur - exception
- PriorityQueue implémentation pour une file à priorité

Interface Queue

```
public interface Queue<E> extends
   Collection<E> {
     E element();
     boolean offer(E e);
     E peek();
     E poll();
     E remove();
}
```

Exemple

```
public static void compteur(int n)
            throws InterruptedException {
        Queue<Integer> file = new
                    LinkedList<Integer>();
        for (int i = n; i >= 0; i--)
            file.add(i);
        while (!file.isEmpty()) {
            System.out.println(file.remove());
            Thread.sleep(1000);
        }
    }
```

Exemple

```
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();
    while (!queue.isEmpty())
        result.add(queue.remove());
    return result;
}
```

Des implémentations

- HashSet<E>: implémentation de Set comme table de hachage. Recherche/ ajout suppression en temps constant
- TreeSet<E>: SortedSet comme arbre binaire équilibré O(log(n))
- ArrayList<E>: liste implémentée par des tableaux à taille variable accès en O(1) ajout et suppression en O(n-i) (i position considérée)
- LinkedList<E>: liste doublement chaînée implémente List et Queue accès en O(i)
- HashMap<K,V>: implémentation de Map par table de hachage ajout suppression et recherche en O(1)
- TreeMap<K,V>: implémentation de SortedMap à partir d'arbres équilibrés ajout, suppression et recherche en O(log(n))
- WeakHashMap<K,V>: implémentation de Map par table de hachage
- PriorityQueue<E>: tas à priorité.

Comparaisons

- Interface Comparable<T> contient la méthode
  - public int compareTo(T e)
  - "ordre naturel"
- Interface Comparator<T> contient la méthode
  - public int compare(T o1, T o2)

Quelques autres packages

- System méthodes static pour le système:
  - entrée-sorties standard
  - manipulation des propriétés systèmes
  - utilitaires "Runtime" exit(), gc() …

Runtime, Process

- Runtime permet de créer des processus pour exécuter des commande: exec
- Process retourné par un exec méthodes
  - destroy()
  - exitValue()
  - getInputStream()
  - getOutputStream()
  - getErrorStream()

Exemple

- exécuter une commande (du système local)
  - associer l'entrée de la commande sur System.in
  - associer la sortie sur System.out.

Exemple

```
class plugTogether extends Thread {
    InputStream from;
    OutputStream to;
    plugTogether(OutputStream to, InputStream from ) {
        this.from = from; this.to = to;
    }
    public void run() {
        byte b;
        try {
            while ((b= (byte) from.read()) !=  -1) to.write(b);
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            try {
                to.close();
                from.close();
            } catch (IOException e) {
                System.out.println(e);
            }
        }
    }
}
```

Exemple suite

```
public class Main {
    public static Process userProg(String cmd)
    throws IOException {
      Process proc = Runtime.getRuntime().exec(cmd);
      Thread thread1 = new plugTogether(proc.getOutputStream(), System.in);
      Thread thread2 = new plugTogether(System.out, proc.getInputStream());
      Thread thread3 = new plugTogether(System.err, proc.getErrorStream());
      thread1.start(); thread2.start(); thread3.start();
      try {proc.waitFor();} catch (InterruptedException e) {}
        return proc;
    }
    public static void main(String args[])
    throws IOException {
       String cmd = args[0];
       System.out.println("Execution de: "+cmd);
       Process proc = userProg(cmd);
    }
}
```

Chapitre X

threads

Threads

- threads: plusieurs activités qui coexistent et partagent des données
  - exemples:
    - pendant un chargement long faire autre chose
    - coopérer
    - processus versus threads
  - problème de l'accès aux ressources partagées
    - verrous
    - moniteur
    - synchronisation

Principes de base

- extension de la classe Thread
  - méthode `run` est le code qui sera exécuté.
  - la création d'un objet dont la superclasse est `Thread` crée la thread (mais ne la démarre pas)
  - la méthode `start` démarre la thread (et retourne immédiatement)
  - la méthode `join` permet d'attendre la fin de la thread
  - les exécutions des threads sont asynchrones et concurrentes

Exemple

```
class ThreadAffiche extends Thread{
    private String mot;
    private int delay;
    public ThreadAffiche(String w,int duree){
        mot=w;
        delay=duree;
    }
    public void run(){
        try{
            for(;;){
                System.out.println(mot);
                Thread.sleep(delay);
            }
        }catch(InterruptedException e){
        }
    }
}
```

Suite

```
public static void main(String[] args) {
      new ThreadAffiche("PING", 10).start();
      new ThreadAffiche("PONG", 30).start();
      new ThreadAffiche("Splash!",60).start();
 }
```

Alternative: Runnable

- Une autre solution:
  - créer une classe qui implémente l'interface `Runnable` (cette interface contient la méthode `run`)
  - créer une `Thread` à partir du constructeur `Thread` avec un `Runnable` comme argument.

Exemple

```
class RunnableAffiche implements Runnable{
    private String mot;
    private int delay;
    public RunnableAffiche(String w,int duree){
        mot=w;
        delay=duree;
    }
    public void run(){
        try{
            for(;;){
                System.out.println(mot);
                Thread.sleep(delay);
            }
        }catch(InterruptedException e){
        }
    }
}
```

Suite

```
  public static void main(String[] args) {
      Runnable ping=new RunnableAffiche("PING", 10);
      Runnable pong=new RunnableAffiche("PONG", 50);
      new Thread(ping).start();
      new Thread(pong).start();
}
```

Synchronisation

- les threads s'exécutent concurremment et peuvent accéder concurremment à des objets:
  - il faut contrôler l'accès:
    - thread un lit une variable (R1) puis modifie cette variable (W1)
    - thread deux lit la même variable (R2) puis la modifie (W2)
    - R1-R2-W2-W1
    - R1-W1-R2-W2 résultat différent!

Exemple

```
class X{
    int val;
}
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        i=x.val;
        System.out.println("thread:"+nom+" valeur x="+i);
        try{
        Thread.sleep(10);
        }catch(Exception e){}
        x.val=i+1;
        System.out.println("thread:"+nom+" valeur x="+x.val);
    }
}
```

Suite

```
public static void main(String[] args) {
        X x=new X();
        Thread un=new Concur("un",x);
        Thread deux=new Concur("deux",x);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch (InterruptedException e){}
        System.out.println("X="+x.val);
    }
```

donnera (par exemple)

- thread:un valeur x=0
- thread:deux valeur x=0
- thread:un valeur x=1
- thread:deux valeur x=1
- X=1

Deuxième exemple

```
class Y{
    int val=0;
    public int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
class Concur1 extends Thread{
    Y y;
    String nom;
    public Concur1(String st, Y y){
        nom=st;
        this.y=y;
    }
    public void run(){
        System.out.println("thread:"+nom+" valeur="+y.increment());
    }
}
```

Suite

```
  public static void main(String[] args) {
        Y y=new Y();
        Thread un=new Concur1("un",y);
        Thread deux=new Concur1("deux",y);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch (InterruptedException e){}
        System.out.println("Y="+y.getVal());
    }
-----------
```

- thread:un valeur=1
- thread:deux valeur=1
- Y=1

Verrous

- à chaque objet est associé un verrou
  - *synchronized*(*expr*) {*instructions*}
    - *expr* doit s'évaluer comme une référence à un objet
    - verrou sur cet objet pour la durée de l'exécution de *instructions*
  - déclarer les méthodes comme synchronized: la thread obtient le verrou et le relâche quand la méthode se termine

synchronised(x)

```
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        synchronized(x){
            i=x.val;
            System.out.println("thread:"+nom+" valeur x="+i);
            try{
                Thread.sleep(10);
            }catch(Exception e){}
            x.val=i+1;
            System.out.println("thread:"+nom+" valeur x="+x.val);
        }
    }
}
```

Méthode synchronisée

```
class Y{
    int val=0;
    public synchronized int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
------------
```

- thread:un valeur=1
- thread:deux valeur=2
- Y=2

Mais…

- la synchronisation par des verrous peut entraîner un blocage:
  - la thread un (XA) pose un verrou sur l'objet A et (YB) demande un verrou sur l'objet B
  - la thread deux (XB) pose un verrou sur l'objet B et (YA) demande un verrou sur l'objet A
  - si XA –XB : ni YA ni YB ne peuvent êter satisfaites -> blocage
- (pour une méthode synchronisée, le verrou concerne l'objet globalement et pas seulement la méthode)

Exemple

```
class Dead{
    Dead partenaire;
    String nom;
    public Dead(String st){
        nom=st;
    }
    public synchronized void f(){
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        System.out.println(Thread.currentThread().getName()+
                " de "+ nom+".f() invoque "+ partenaire.nom+".g()");
        partenaire.g();    }
    public synchronized void g(){
        System.out.println(Thread.currentThread().getName()+
                " de "+ nom+".g()");
    }
    public void setPartenaire(Dead d){
        partenaire=d;
    }
}
```

Exemple (suite)

```
        final Dead un=new Dead("un");
        final Dead deux= new Dead("deux");
        un.setPartenaire(deux);
        deux.setPartenaire(un);
        new Thread(new Runnable(){public void run(){un.f();}
        },"T1").start();
        new Thread(new Runnable(){public void run(){deux.f();}
        },"T2").start();
------------
```

- T1 de un.f() invoque deux.g()
- T2 de deux.f() invoque un.g()

Synchronisation…

- wait, notifyAll notify
  - attendre une condition / notifier le changement de condition:

```
synchronized void fairesurcondition(){
    while(!condition){
        wait();
        faire ce qu'il faut quand la condition est vraie
        }
------------------
synchronized void changercondition(){
    … changer quelque chose concernant la condition
    notifyAll(); // ou notify()
}
```

Exemple (file: rappel Cellule)

```
public class Cellule<E>{
   private Cellule<E> suivant;
   private E element;
   public Cellule(E val) {
      this.element=val;
   }
   public Cellule(E val, Cellule suivant){
      this.element=val;
      this.suivant=suivant;
   }
   public E getElement(){
      return element;
   }
   public void setElement(E v){
      element=v;
   }
   public Cellule<E> getSuivant(){
      return suivant;
   }
   public void setSuivant(Cellule<E> s){
      this.suivant=s;
   }
```

File synchronisées

```
class File<E>{
    protected Cellule<E> tete, queue;
    private int taille=0;

    public synchronized void enfiler(E item){
        Cellule<E> c=new Cellule<E>(item);
        if (queue==null)
            tete=c;
        else{
            queue.setSuivant(c);
        }
        c.setSuivant(null);
        queue = c;
        notifyAll();
 }
```

File (suite)

```
public synchronized E defiler() throws InterruptedException{
       while (tete == null)
           wait();
       Cellule<E> tmp=tete;
       tete=tete.getSuivant();
       if (tete == null) queue=null;
       return tmp.getElement();
}
```