Cours 8

Exceptions (fin)

Généricité

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

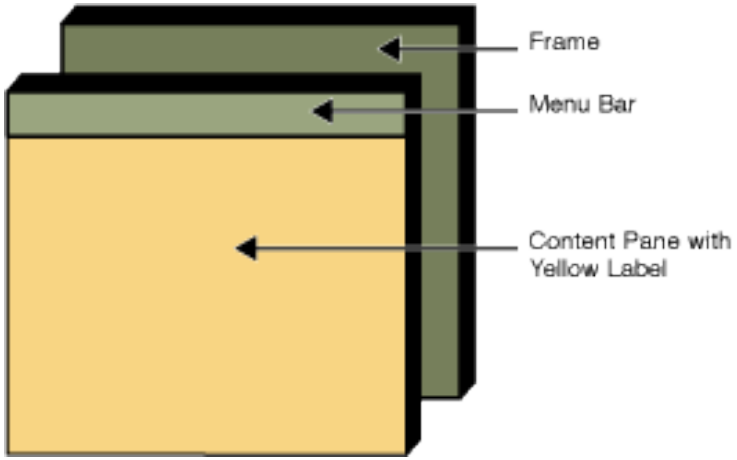
- Dans une interface graphique, le programme réagit aux interactions avec l'utilisateur
- Les interactions génèrent des événements
- Le programme est dirigé par les événements (event-driven)

Afficher…

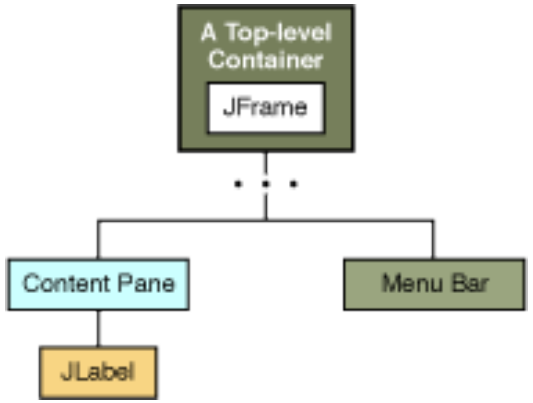
- Pour pouvoir être affiché, il faut que le composant soit dans un top-level conteneur:

  (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();
         }
      });
   }
}
```

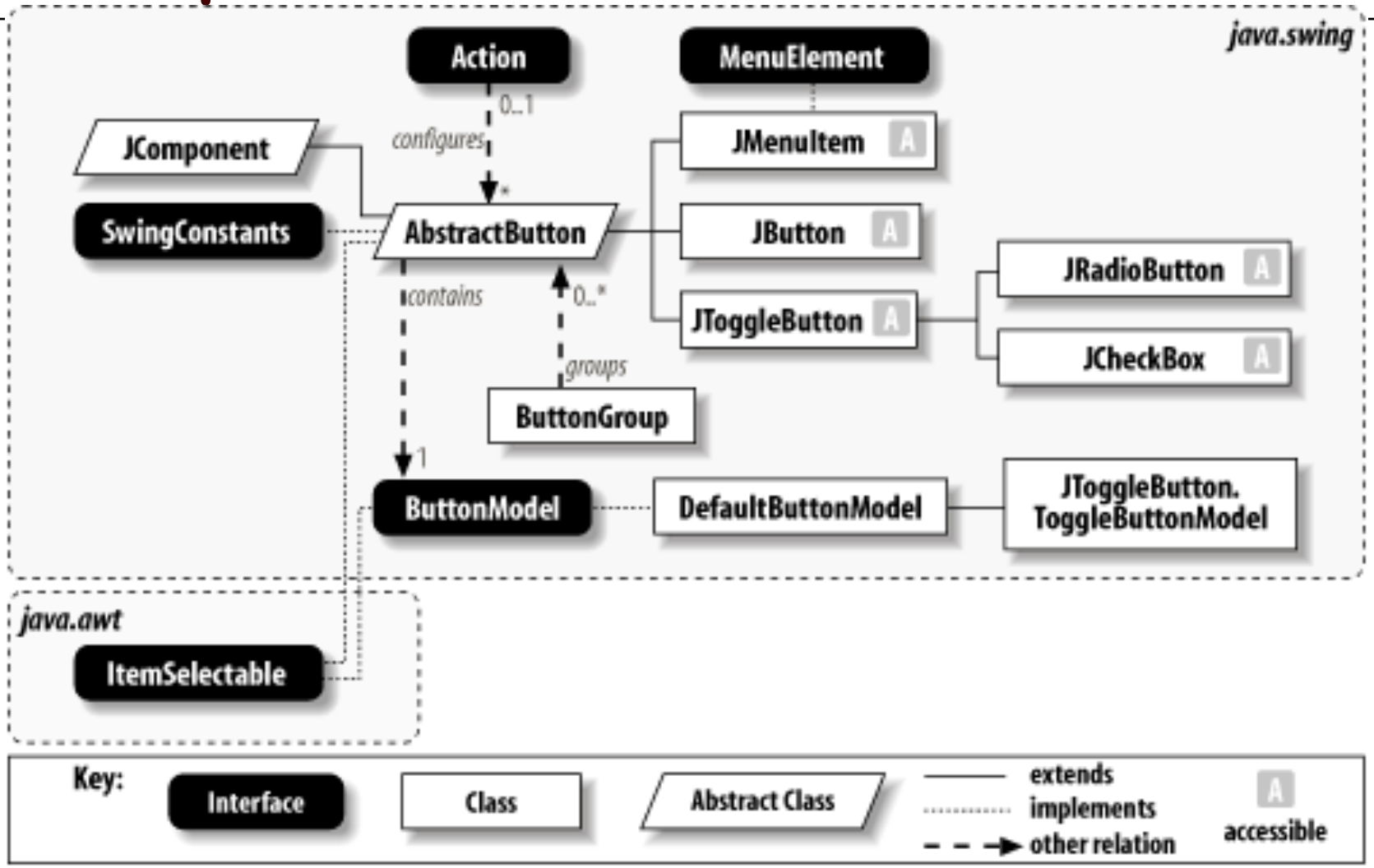
Evénements: principes

- 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

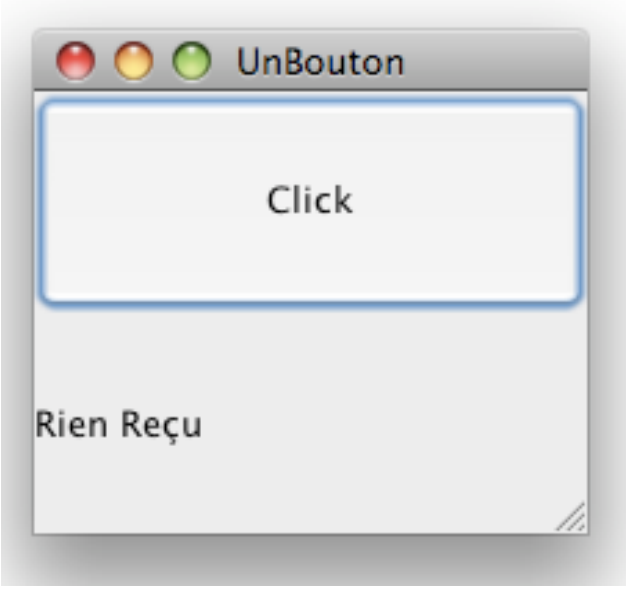
- Chaque composant génère des événements:
  - Presser un JButton génère un ActionEvent (système d'interface graphique)
    - Cet ActionEvent contient des infos (quel bouton, position de la souris, modificateurs…)
  - Un event listener (implémente ActionListener)
    - définit une méthode actionPerformed
    - 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");
  }
```

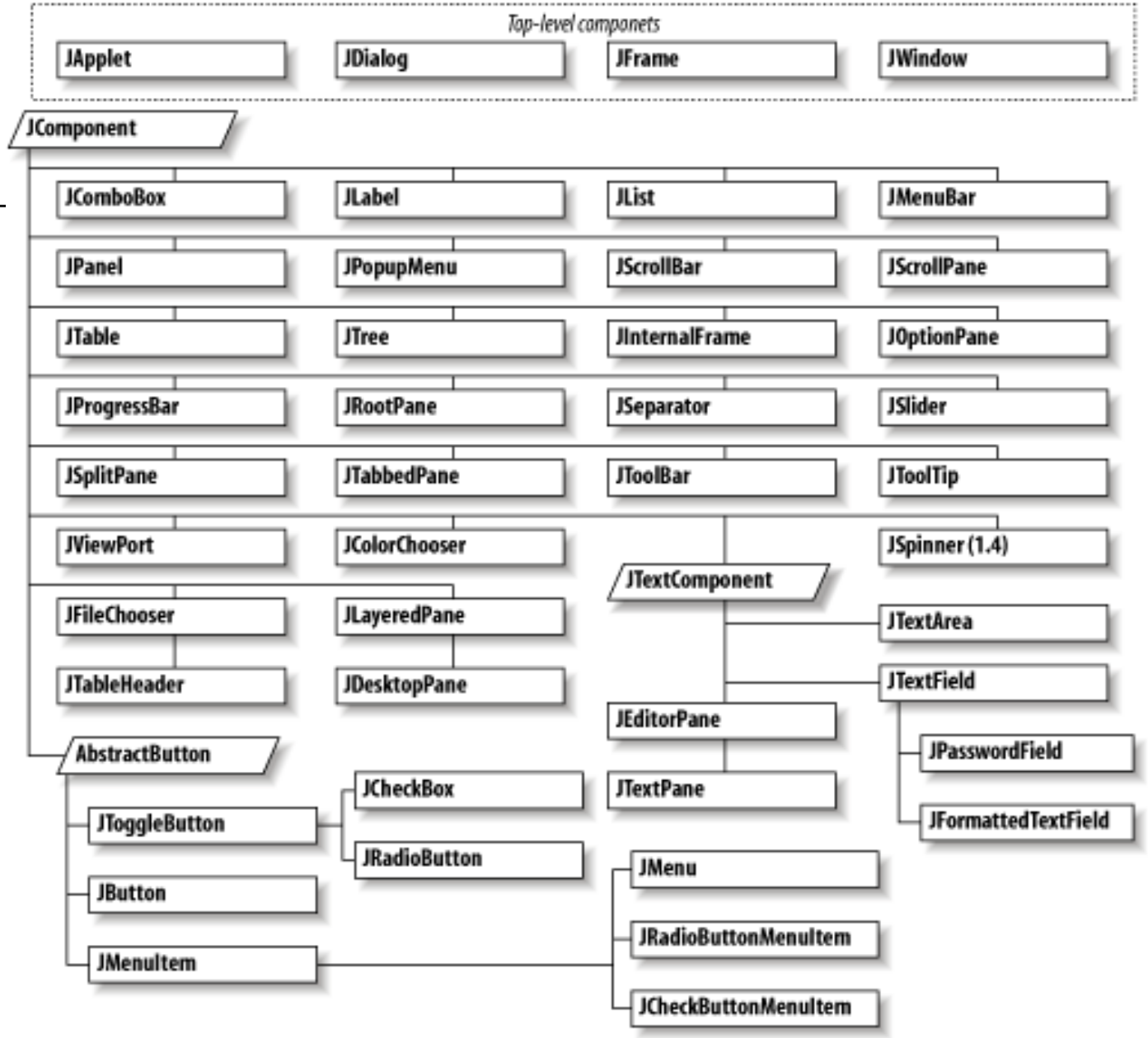
Code

```
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

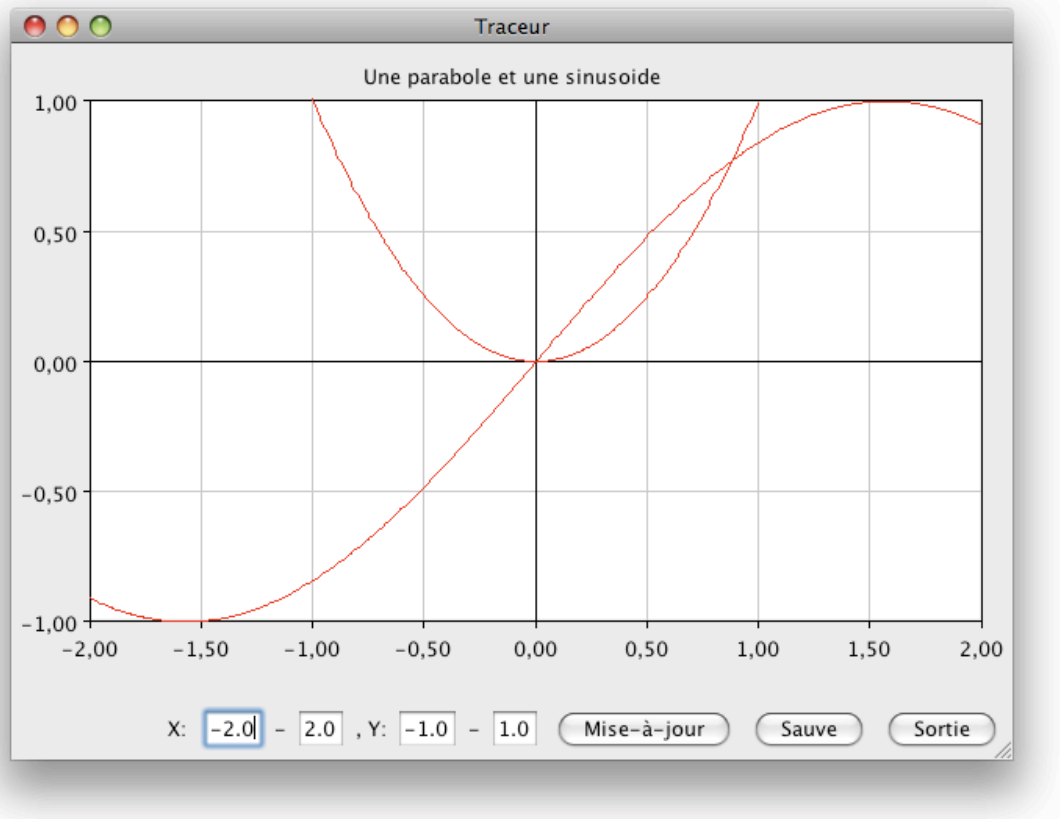
```
public class UnBoutonBis extends JPanel {
//…
 bouton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Toolkit.getDefaultToolkit().beep();
                label.setText("clické " + (++cmp) + " fois");
            } });
        }
//…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

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
AbstractButton
JToggleButton
JCheckBox
JRadioButton
JButton
JMenuItem
JTextComponent
JTextArea
JTextField
JEditorPane
JTextPane
JPasswordField
JFormattedTextField
JMenu
JRadioButtonMenuItem
JCheckButtonMenuItem

Un exemple

- Un traceur de fonctions
  - Une interface graphique swing



Organisation

- GrapheSwing contient unGraphePanel 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

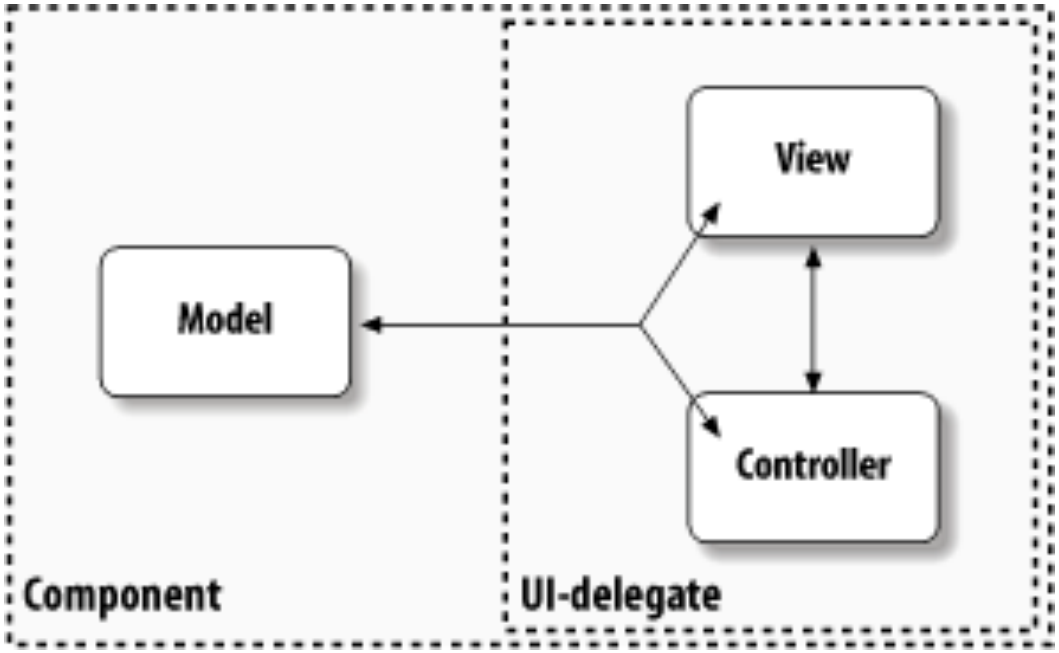
Le main

```
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 dessinés 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

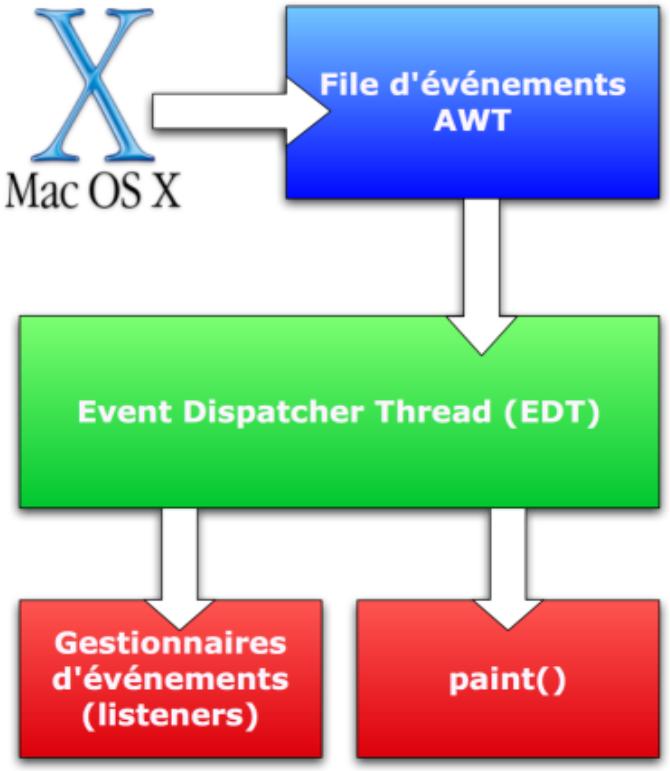
- Swing prend en charge la gestion des composants qui sont dessinés en code Java (lightweight)
- Les composants AWT sont eux liés aux composants natifs (heavyweight)
- Swing dessine le composants dans un canevas AWT et utilise le traitement des événements de AWT

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();
```

Paramètre type et invocation

Invocation et type 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 (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 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 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 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 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 est 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 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 paramètre type: pas de new T() ou de new T[]
  - on ne peut pas utiliser `instanceof` pour une instance de type paramétrée
  - 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)
}
```