

# Contents

<b>2</b>	<b>Structures de données et interfaces graphiques</b>	<b>3</b>
2.1	Structures de données . . . . .	3
2.1.1	Structures fréquentes . . . . .	3
2.1.2	Ecrire des classes génériques . . . . .	6
2.1.3	Structurations logiques . . . . .	7
2.1.4	Algorithmes sur les structures fréquentes . . . . .	8
2.2	Interfaces graphiques . . . . .	9
2.2.1	Exemple . . . . .	9
2.2.2	Mises en page . . . . .	10
2.2.3	Que mettre dans les mises en pages . . . . .	12
2.2.4	Evènements . . . . .	12
2.3	Exercices . . . . .	13



## Cours 2

# Structures de données et interfaces graphiques

Dans le chapitre précédent nous avons vu une méthodologie pour concevoir des classes avec une architecture modulaire. Dans ce chapitre nous détaillons plus la manière d'écrire le corps des méthodes. Nous nous attachons en particulier au fait de pouvoir parcourir différentes structures de données.

## 2.1 Structures de données

### 2.1.1 Structures fréquentes

#### Les vecteurs

##### Manipulation d'un nombre arbitraire de données

Imaginons un carnet d'adresses représenté par la classe `CarnetAdresse`, contenant des instances de la classe `Personne`. Les personnes pourraient être stockées dans un tableau par exemple :

```
class CarnetAdresse {
    Personne [] listeDesPersonnes = new Personne[100];
    // . . .
}
```

Cependant si on rajoute des personnes on risque de dépasser la taille de départ du tableau `listeDesPersonnes`. Aussi si le nombre des éléments peut être arbitrairement grand il faut plutôt utiliser la classe `java.util.Vector` décrite ci-dessous.

##### La classe `java.util.Vector`

La classe `java.util.Vector` est une classe paramétrable (générique dans la terminologie Java). Elle est paramétrée par le type des données contenues. Ainsi pour l'exemple précédent il nous faut un vecteur de `Personne` ce qui se note `Vector<Personne>`. Voici le code java :

```
import java.util.Vector;

class CarnetAdresse {
    Vector<Personne> listeDesPersonnes = new Vector<Personne>();
    // . . .
}
```

##### Exemple complet

Voici l'exemple complet de la classe `CarnetAdresse` :

```
1
2 import java.util.Vector;
3
4 class CarnetAdresse {
5
6     public static class Personne {
7         public String prenom;
```

```

8         public String nom;
9         public String telephone;
10        public Personne(String p,String n, String t) {
11            prenom=p;
12            nom=n;
13            telephone=t;
14        }
15
16    }
17
18    Vector<Personne> listeDesPersonnes = new Vector<Personne>();
19    public CarnetAdresse () {}
20    public void ajoutePersonne(Personne p) {
21        listeDesPersonnes.add(p);
22    }
23    public void affiche() {
24        for (int i=0;i<listeDesPersonnes.size();++i) {
25            Personne p = listeDesPersonnes.elementAt(i);
26            System.out.println(p.prenom+ " "+p.nom+ " "+p.telephone);
27        }
28    }
29
30    public static void main(String [] argv) {
31        CarnetAdresse ca = new CarnetAdresse();
32        ca.ajoutePersonne(new Personne("Victor", "Hugo", "+33 01 23 45 67 89"));
33        ca.ajoutePersonne(new Personne("Alphred", "De Musset", "+33 02 34 56 78 90"));
34        ca.affiche();
35    }
36 }

```

### Aparté sur les génériques

#### Ecrire ses classes génériques

Tout comme la classe `Vector` qui est générique on peut définir de nouveaux types génériques. Ainsi on peut paramétrer le moteur d'une voiture en déclarant la classe `Voiture` de la façon suivante :

```

1  class Voiture <M> {
2      M moteur;
3
4      public Voiture(M m) {
5          moteur = m;
6      }
7      public M donneMoteur () {
8          return moteur;
9      }
10 }

```

On peut alors définir les classes suivantes `Voiture<MoteurDeSport>` ou encore `Voiture<MoteurPasCher>` si `MoteurDeSport` et `MoteurPasCher` sont des classes définies par ailleurs.

#### Sous typage

On appelle sous-type une classe dérivant d'une autre. Ainsi `java.lang.Integer` dérive de `java.lang.Object` car `java.lang.Integer` hérite de `java.lang.Number` qui hérite lui-même de `java.lang.Object`.

Or le type `Vector<Integer>` n'est pas un sous-type de `Vector<Object>`. Ainsi les lignes suivantes :

```

Vector<Integer> v1;
Vector<Object> v2 = v1; // erreur à la compilation

```

Provoquent l'erreur suivante à la compilation :

```
incompatible types
found   : java.util.Vector<java.lang.Integer>
required: java.util.Vector<java.lang.Object>
Vector<Object> v2 =v1;
```

Ce comportement n'est pas très intuitif.

### Egalité des types génériques

Une autre particularité est que deux spécialisations du même type générique sont égaux. Ainsi le programme suivant affiche true :

```
1 import java.util.Vector;
2 class SousTypage {
3     public static void main(String [] argv) {
4         Vector<Integer> v1 = new Vector<Integer>();
5         Vector<Object> v2 = new Vector<Object>();
6         System.out.println(v1.getClass()==v2.getClass());
7     }
8 };
```

Ainsi donc on constate ici que v1 et v2 ont le même type mais que l'opération d'affectation v2=v1 est impossible comme dans l'exemple précédent.

### Jokers

Les types jokers (wildcard) permettent d'utiliser des types génériques sans les nommer. Ainsi la déclaration de la méthode java.util.Collections.disjoint pour tester si deux séquences n'ont pas d'argument en commun est :

```
class Collections {
    static boolean disjoint(Collection<?> c1, Collection<?> c2);
    ...
}
```

Ce qui est équivalent à :

```
class Collections {
    static <T> boolean disjoint(Collection<T> c1, Collection<T> c2);
    ...
}
```

Le fait de ne pas nommer le type générique T mais d'utiliser à la place ? donne une écriture plus lisible et permet de ne pas utiliser le nom T si on en a besoin ailleurs.

### Jokers contraints

On peut également contraindre les types génériques "?". Imaginons que nous voulions effectuer une opération interdite comme-ci dessous :

```
Vector<Integer> v1;
Vector<Object> v2 = v1; // erreur à la compilation
```

La sémantique de ce que nous voulons faire est quelque chose dans le genre :

Recopie une liste de type Vector<T> dans Vector<U> où T hérite de U.

On peut alors coder cette sémantique en écrivant le type "? extends T" :

```
1 import java.util.Vector;
2 class MaClasse {
3     public <T> void copy(Vector<T> dest, Vector<? extends T> src)
4     { /* ... */ }
5 }
```

Il existe également la syntaxe duale "? super T" pour décrire un type dont T est un sous type ou T lui même.

## Les containers associatifs

### Motivation

Un autre type de container souvent utilisé est le container associatif : il s'agit d'une liste de correspondance non bornée en taille.

Imaginons par exemple que l'on souhaite dans une entreprise maintenir une liste des personnes avec le numéro de bureau dans laquelle elle travaille. Il faudrait une "table de correspondance" qui prend en entrée un nom en renvoie un entier en sortie.

Ici le nom sert de clé pour se repérer dans la table.

#### HashMap

Il existe plusieurs classes réalisant cette tâche. `HashMap` est une implémentation souvent utilisée pour cela. Il s'agit d'une classe générique (comme `Vector`) mais avec deux paramètres : un pour la clé et un autre pour la valeur.

Ainsi pour l'exemple précédent on peut définir l'instance `annuaireBureau` qui représente l'attribution des bureaux.

```
java.util.HashMap<String,Integer> annuaireBureau =
    new java.util.HashMap<String,Integer>();
annuaireBureau.put("Robert",123);
annuaireBureau.put("Anne",234);
annuaireBureau.put("Sylvie",345);
```

Pour obtenir le bureau dans lequel est Anne on peut alors invoquer la méthode `get` :

```
Integer bureauDeAnne = annuaireBureau.get("Anne");
```

La méthode `get` renvoie `null` si aucune valeur correspondante n'a été rentrée dans la table.

### 2.1.2 Ecrire des classes génériques

#### Pourquoi des classes génériques

L'apparition des génériques en Java date de 2004 alors que le langage s'est principalement développé au milieu des années 1990. En l'absence de génériques la classe `java.util.Vector` ne renvoyait que des objets de type `java.lang.Object` et il fallait donc faire un "cast" c'est à dire expliciter le type attendu :

```
// ancienne syntaxe de java.util.Vector sans les génériques
Vector v = new Vector();
v.add(new Foo());
Foo foo = (Foo) v.elementAt(0);
UneAutreClasse instance = (UneAutreClasse) v.elementAt(0); // erreur à
// l'exécution
```

L'inconvénient d'une telle construction est qu'il faut se souvenir vers quoi il faut "caster" `v.elementAt(0)`. Si on se trompe, l'erreur n'est détectée qu'à l'exécution, ce qui nécessite beaucoup tester son code si on veut être sûr de ne pas avoir laissé de bugs.

En revanche avec les génériques si il y a une erreur, cette dernière est détectée à tous les coups à la compilation :

```
// syntaxe de java.util.Vector avec les génériques :
Vector<Foo> v = new Vector<Foo>();
v.add(new Foo());
Foo foo = v.elementAt(0);
UneAutreClasse instance = v.elementAt(0); // erreur à la compilation
```

Les génériques permettent donc de diminuer le nombre d'erreurs à l'exécution tout en facilitant la réutilisation de code.

#### Syntaxe et sémantique

La syntaxe décrit une suite cohérente de lexèmes (un nom, une parenthèse, ...). La sémantique est de sens de l'opération. Ainsi d'un point de vue syntaxique `v.sort()` est l'appel de la méthode `sort` sur l'instance `v` mais rien de plus : on ne sait pas ce que fait la méthode `sort`. D'un point de vue sémantique on peut penser que `v.sort()` va trier l'instance `v`, mais c'est impossible à dire sans voir le code de `sort`.

Souvent quand on programme on se laisse influencer par le nom des méthodes pour donner une sémantique (c'est à dire un sens) alors que le compilateur lui ne voit que la syntaxe.

Il faut bien comprendre que les génériques sont bien sûr syntaxiques et non sémantiques. Considérons l'exemple ci-dessous :

```
class Test <T> {
    static T min(T t1, T t2) {
        if(t1.inferieurA(t2)) return t1;
        return t2;
    }
}
```

Il semble logique de dire que la méthode `min` renvoie le minimum entre `t1` et `t2`. Cette impression est vraie si `t1.inferieurA(t2)` renvoie `true` si `t1` est plus petit que `t2`. Si en fait `t1.inferieurA(t2)` renvoie `true` si `t2` est plus petit que `t1` alors la méthode `min` renvoie en fait le maximum.

En résumé : les classes génériques décrivent de manière syntaxique une suite d'opérations, mais la sémantique finale va en fait dépendre de la classe avec laquelle elle va être instanciée.

### 2.1.3 Structurations logiques

L'idée de cette section est de dégager la notion d'interface et comment elle permet de structurer le cadre "Collection" de la librairie Java.

#### Motivation

Reprenons la structure de la classe `CarnetAdresse` donnée précédemment en exemple :

```
// ...
class CarnetAdresse {
    Vector<Personne> listeDesPersonnes = new Vector<Personne>();
    // ...
}
```

Nous conservons la liste des personnes dans un objet de type `Vector`. Si par exemple notre annuaire devient beaucoup plus volumineux il serait judicieux de pouvoir sortir rapidement la liste des noms commençant par "B" par exemple. Il faudrait donc conserver une liste ordonnée. Or il est difficile dans `Vector` d'insérer un élément au milieu des autres. Ceci est mieux fait avec `java.util.LinkedList`.

Du coup partout dans le code où apparaît `Vector<Personne>` il faut écrire `LinkedList<Personne>` alors que nous allons faire des opérations presque identiques. Ainsi la méthode :

```
public void ajoute(Vector<Personne> liste, Personne p) {
    liste.add(p);
}
```

Doit être remplacée par :

```
public void ajoute(LinkedList<Personne> liste, Personne p) {
    liste.add(p);
}
```

On souhaiterait avoir des abstractions de plus haut niveau correspondant à ces comportements, indépendamment de l'implémentation (`Vector` ou `LinkedList`). On a envie dans l'exemple ici de dire quelque chose dans le genre : "mon objet se comporte comme une liste à laquelle je peux ajouter un objet". Dans le cas présent une telle abstraction existe et se nomme `List`.

On peut ainsi écrire :

```
public void ajoute(List<Personne> liste, Personne p) {
    liste.add(p);
}
```

et passer à la méthode `ajoute` indifféremment un `Vector<Personne>` ou un `LinkedList<Personne>`.

On dit que les classes `Vector` et `LinkedList` implémentent l'interface `List`.

C'est confirmé par la visite du site `java.sun.com` :

```
java.util
Class Vector<E>
```

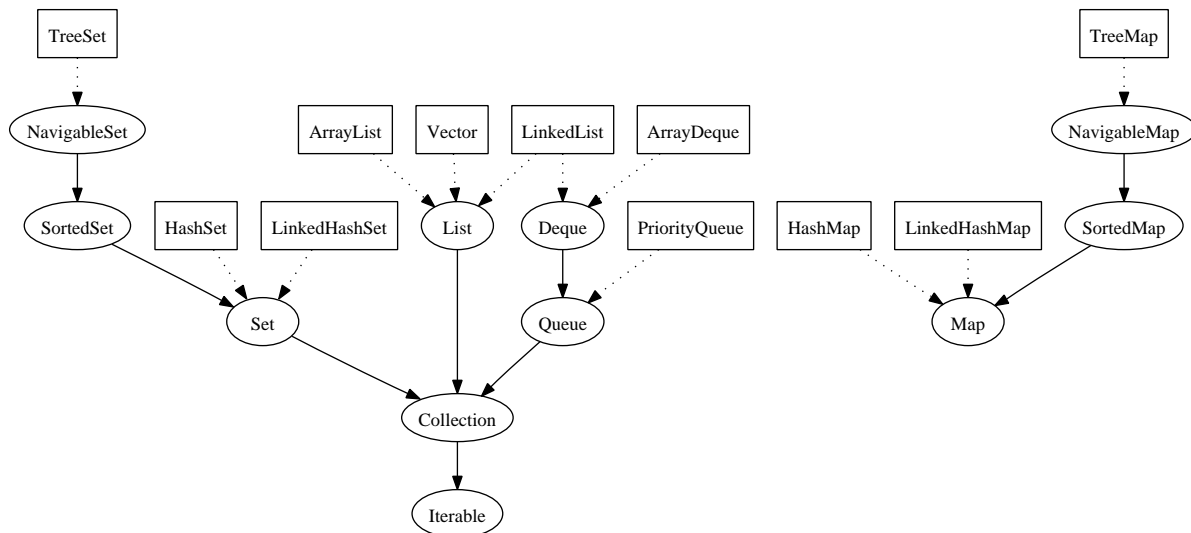
```
java.lang.Object
  extended by java.util.AbstractCollection<E>
    extended by java.util.AbstractList<E>
      extended by java.util.Vector<E>
```

All Implemented Interfaces:  
 Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:  
 Stack

### Structure du cadre Collection

Voici une image partielle des interfaces et des implémentations du “Java Collection Framework”. Les interfaces sont dans des ovales, les implémentations dans des carrés.



Interfaces et instances.

Deux racines sont présentes dans les interfaces : `Collection` et `Map`.

**Collection** L'interface `Collection` se raffine en :

- `Set`, une collection non ordonnée
- `List`, une collection ordonnée où on peut précisément manipuler la position de chaque élément,
- `Queue`, permet de gérer une collection d'éléments avec notions de type FIFO (First In First Out = Premier rentré premier sortit).

#### Map

L'interface `Map` se raffine en simplement en une autre interface `SortedMap` qui permet de conserver les paires (clé, valeur) triées clé ce qui facilite certaines opérations.

### 2.1.4 Algorithmes sur les structures fréquentes

#### Le tri

La méthode `java.util.Collections.sort` permet de trier un objet implémentant l'interface `java.util.List`, comme en témoigne le programme ci-dessous :

```

1  import java.util.*;
2
3  public class Sorting {
4      public static void main(String[] args) {
5          List<String> list = new Vector <String>();
6          list.add("Un element");
7          list.add("Un autre");
8          list.add("et un troisieme.");
9          Collections.sort(list);
10         System.out.println(list);
11     }
12 }

```

L'exécution du programme affiche les éléments dans l'ordre lexicographique :

```
[Un autre, Un element, et un troisieme.]
```

La classe `java.util.Collections` contient des méthodes statiques permettant d'agir sur les collections. Il ne faut pas la confondre avec l'interface `java.util.Collection`.

## 2.2 Interfaces graphiques

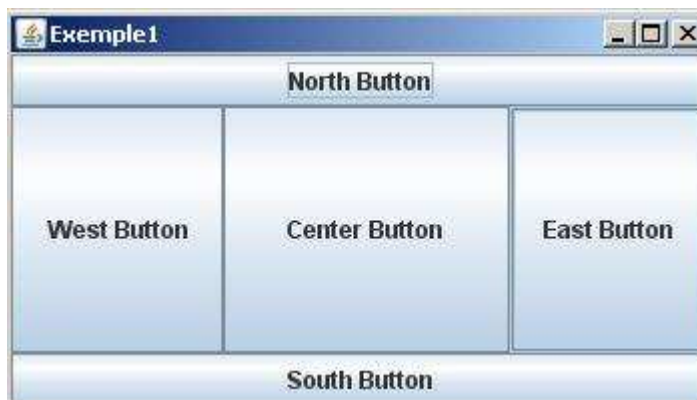
### 2.2.1 Exemple

Nous souhaitons réaliser une application avec les 5 boutons suivants :



Une interface à 5 boutons.

Nous souhaitons également que les boutons changent automatiquement de taille quand la fenêtre est modifiée :



Une interface à 5 boutons de taille variable.

Nous allons pour cela prendre un panneau (ou `JPanel`) dans lequel nous allons pouvoir “coller” des composants (boutons, listes ou autre).

La structure de la classe est donc la suivante :

```

package exemple1;

public class Exemple1

```

```

    extends javax.swing.JPanel
{
    public Exemple1() {
        // code à écrire
    }
    public static void main(String [] argv) {
        // code à écrire
    }
}

```

Dans le constructeur nous allons devoir créer une mise en page (layout en anglais) qui définit les zones nord, sud, est, ouest et centre. Heureusement il existe déjà une classe qui fait ce travail : `java.awt.BorderLayout`. Aussi il suffit pour le constructeur d'écrire le code suivant :

```

public Exemple1() {
    super(new java.awt.BorderLayout());
    JButton boutonC = new JButton("Center Button");
    add (boutonC, java.awt.BorderLayout.CENTER);
    JButton boutonN = new JButton("North Button");
    add (boutonN, java.awt.BorderLayout.NORTH);
    JButton boutonS = new JButton("South Button");
    add (boutonS, java.awt.BorderLayout.SOUTH);
    JButton boutonE = new JButton("East Button");
    add (boutonE, java.awt.BorderLayout.EAST);
    JButton boutonW = new JButton("West Button");
    add (boutonW, java.awt.BorderLayout.WEST);
}

```

On remarque que les ajouts à la mise en page sont effectués par la méthode `javax.swing.JPanel.add( truc à ajouter , endroit où l'ajouter )`.

Dans la fonction `main` il faut maintenant mettre ce panneau dans une fenêtre "normale". La classe représentant une fenêtre "normale" est `javax.swing.JFrame`. La méthode `main` s'écrit alors :

```

public static void main(String [] argv) {
    javax.swing.JFrame frame = new javax.swing.JFrame("Exemple1");
    frame.setDefaultCloseOperation( javax.swing.JFrame.EXIT_ON_CLOSE );
    Exemple1 e = new Exemple1();
    frame.setContentPane( ( javax.swing.JComponent ) e );
    frame.pack();
    frame.setVisible(true);
}

```

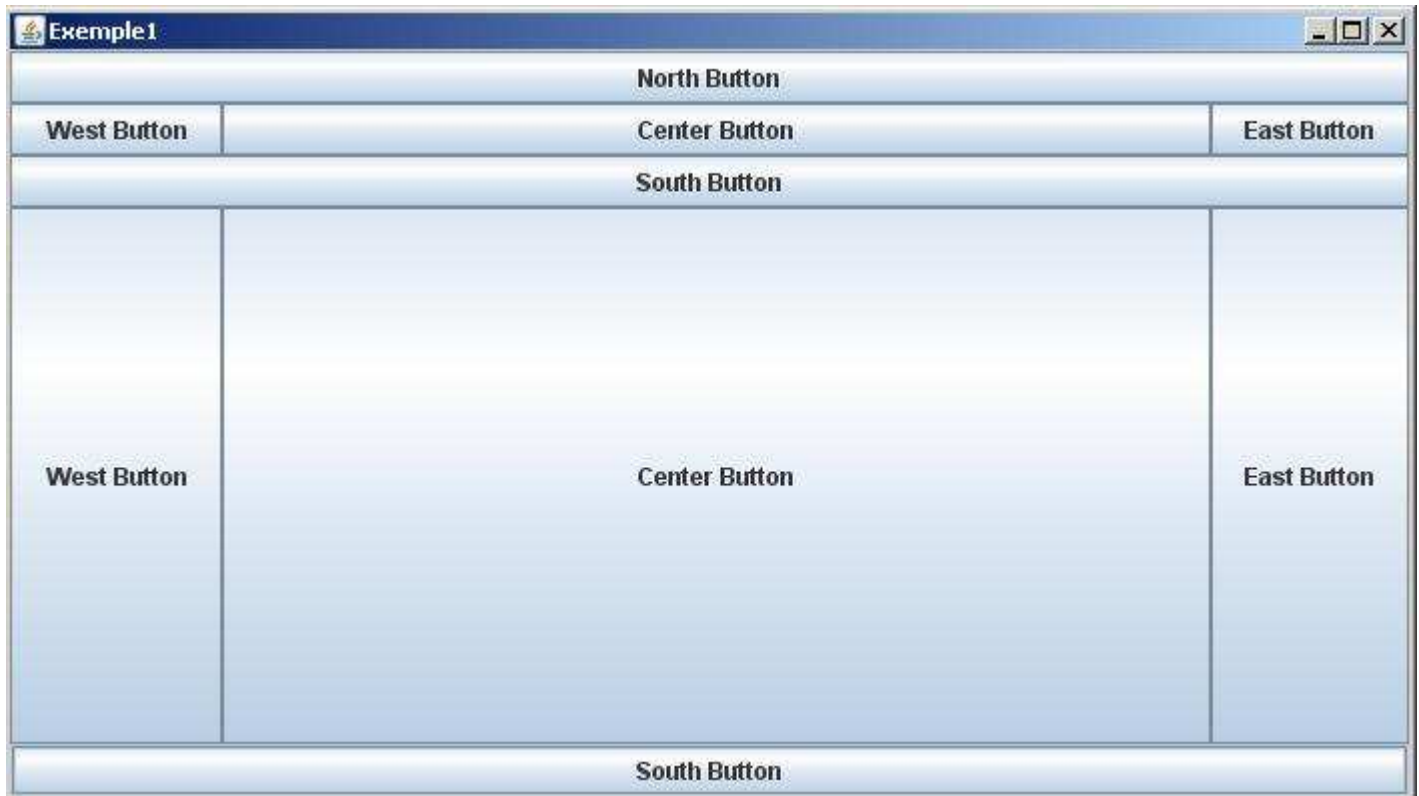
L'exemple complet est téléchargeable à l'url :

[http://www.derepas.com/java\\_int/demo\\_3\\_1.jar](http://www.derepas.com/java_int/demo_3_1.jar)

## 2.2.2 Mises en page

### Combinaison de mise en page

Nous avons vu la mise en page `java.awt.BorderLayout` dans l'exemple précédent. On peut utiliser à nouveau cette mise en page à la place d'un bouton existant pour obtenir l'agencement ci-dessous :



Une interface à 10 boutons.

Il suffit dans la méthode main de rajouter l'instruction

```
d.add(new Exemple1(), java.awt.BorderLayout.NORTH);
```

D'où le code complet suivant :

```
package cours3;

import javax.swing.*;

public class Exemple1
    extends javax.swing.JPanel
{
    static final long serialVersionUID = 1234531;
    public Exemple1() {
        super(new java.awt.BorderLayout());
        JButton buttonC = new JButton("Center Button");
        add (buttonC, java.awt.BorderLayout.CENTER); `w
        JButton buttonN = new JButton("North Button");
        add (buttonN, java.awt.BorderLayout.NORTH);
        JButton buttonS = new JButton("South Button");
        add (buttonS, java.awt.BorderLayout.SOUTH);
        JButton buttonE = new JButton("East Button");
        add (buttonE, java.awt.BorderLayout.EAST);
        JButton buttonW = new JButton("West Button");
        add (buttonW, java.awt.BorderLayout.WEST);
    }
    public static void main(String [] argv) {
        javax.swing.JFrame frame = new javax.swing.JFrame("Exemple1");
        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        Exemple1 d = new Exemple1();
```

```

        d.add(new Exemple1(), java.awt.BorderLayout.NORTH);
        frame.setContentPane(( javax.swing.JComponent )d);
        frame.pack();
        frame.setVisible(true);
    }
}

```

L'exemple complet est téléchargeable à l'url :

[http://www.derepas.com/java\\_int/demo\\_3\\_2.jar](http://www.derepas.com/java_int/demo_3_2.jar)

### Un autre type de mise en page

Il existe de nombreux autres types de mise en page que `java.awt.BorderLayout`. Mentionnons simplement `java.awt.GridLayout` qui permet d'afficher les composants contenus dans un rectangle.

Un exemple d'utilisation est proposé dans le fichier à télécharger pour l'exercice 2 :

[http://www.derepas.com/java\\_int/ex\\_3\\_2.jar](http://www.derepas.com/java_int/ex_3_2.jar)

## 2.2.3 Que mettre dans les mises en pages

### Des boutons

Comme dans l'exemple précédent on peut ajouter des boutons qui sont des instances de `JButton` :

```

JButton boutonC = new JButton("Center Button");
add ( boutonC, java.awt.BorderLayout.CENTER);

```

### Des labels

On peut également ajouter des `javax.swing.JLabel` qui supportent des images ou du texte :

```

JLabel labelAvecTexte = new JLabel("Bonjour!");
JLabel labelAvecImage = new JLabel(new ImageIcon("monImage.png"));
add ( labelAvecTexte, java.awt.BorderLayout.NORTH);
add ( labelAvecImage, java.awt.BorderLayout.CENTER);

```

## 2.2.4 Evènements

### Principe

Tout objet souhaitant recevoir des informations de la souris doit implémenter l'interface `java.awt.MouseListener`.

### Exemple

Ainsi dans le jeu de puissance 4 nous allons placer des cases qui héritent de `JLabel` pour afficher des images et qui en plus vont réagir aux clics de la souris :

```

package cours3;

import java.awt.event.MouseEvent;

import javax.swing.*;

public class Case extends JLabel
    implements java.awt.event.MouseListener
{
    // ...

    public Case(int c, Humain h) {

```

```

        // ...
    }

    public void fixeJeton(Jeton jeton) {
        // ...
    }

    //
    // méthodes pour l'implémentation de la classe mouse listener
    //

    public void mouseClicked(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
    public void mousePressed(MouseEvent e) {
        if (getBounds().contains(e.getX(),e.getY())) {
            System.out.println("Il y a un clic dans la colonne "
                +colonne+" aux coordonnées "
                +e.getX()+" "+e.getY());
            humain.joue(colonne);
        }
    }
    public void mouseReleased(MouseEvent e) {
    }
}

```

Dès lors les instance de Case doivent être enregistrées auprès d'un JPanel pour que les informations de la souris leur soit fournies :

```

// extrait de la classe afficheur :
package cours3;

import javax.swing.*;

public class Afficheur extends JPanel {
    // ...
    Afficheur(Humain humain) {
        // ...
        cases[i][j]=new Case(j, humain);
        addMouseListener(cases[i][j]);
    }
    // ...
}

```

## 2.3 Exercices

### Ex 2.1 \* Carnet d'adresses

On se donne les spécifications textuelles suivantes :

On souhaite réaliser une application modélisant un carnet d'adresse. Il contient pour chaque nom de famille un unique numéro de téléphone. On souhaite être capable de :

- ajouter une personne,

- supprimer une personne,
- obtenir le numéro de téléphone d'une personne.
- pour un numéro de téléphone obtenir le nom.

Exemple de carnet d'adresse :

Alphonse	+1 567-123-234
Bernadette	06 42 28 39 40
Charles	01 23 34 92 47
Delphine	+32 432 55 78

**Q1** Ecrire le squelette de la classe `CarnetAdresse` réalisant les spécifications ci-dessous.

**Q2** Remplir le corps des méthodes de la classe `CarnetAdresse`.

**Q3** Réaliser un petit script testant chaque méthode publique.

**Ex 2.2 \* Application à 5 boutons**

Considérons l'image ci-dessous :



Application à 5 boutons.

Utiliser la classe `Exemple1` de la section 2.2.1 pour réaliser l'image ci-dessous.

**Ex 2.3 \*\* Faire jouer l'ordinateur**

Une méthode a été rajoutée dans la classe `Tableau` : `evaluate`, elle permet d'évaluer si une situation est positive ou non pour un joueur. Ainsi pour évaluer le fait de jouer dans la colonne `i` on peut appeler la méthode `evaluate` :

```
int evaluation = tableau.evaluate(i, maCouleur);
```

où `maCouleur` est donne la couleur à jouer comme par exemple :

```
Jeton maCouleur = new Jeton(Jeton.JAUNE);
```

Plus la valeur renvoyée est impotante plus il est avantageux de jouer dans la colonne `i`.

Le but de l'exercice est de compléter la méthode `joue` dans la classe `Ordinateur`. Pour cela importer dans un nouveau projet le fichier

[http://www.derepas.com/java\\_int/ex\\_2\\_2.jar](http://www.derepas.com/java_int/ex_2_2.jar)

**Q1** Dans la méthode `joue` évaluer le fait de jouer dans chaque colonne.

**Q2** Sélectionner au hasard l'une des plus grandes valeur et jouer dans la colonne correspondante.

### Ex 2.4 \*\* Ordre de comparaison

Le but de l'exercice est de montrer comment on peut choisir l'ordre avec lequel les éléments sont comparés.

**Q1** Reprendre l'exemple de la section 2.1.4 et vérifier le résultat.

**Q2** On souhaite maintenant afficher les chaînes de caractère dans l'ordre lexicographique inverse. Créer pour cela créer une classe comparateur implémentant `java.util.Comparator<String>` et utiliser une des méthodes `java.util.Collections.sort`.

### Ex 2.5 \*\* Puissance 4

Le but de cet exercice est de rajouter une interface graphique au puissance 4 réalisé dans l'exercice 2.3.

Pour cela importez le fichier jar suivant dans un nouveau projet :

[http://www.derepas.com/java\\_int/ex\\_3\\_2.jar](http://www.derepas.com/java_int/ex_3_2.jar)

On a rajouté les classe `Afficheur` et `Case`. La classe `Case` qui hérite de `JLabel` est un composant permettant d'afficher soit une croix soit un rond.

Pour avoir un puissance 4 qui marche il suffit de compléter le constructeur de la classe `Afficheur`.