Travaux dirigés de JAVA

Programmation par objets

Classes - encapsulation - objets
Héritage - méthodes virtuelles - classes abstraites
Généricité - Interfaces

Interfaces homme-machine

Fenêtrage - réaction aux événements

Programmation parallèle

Processus (**`Thread`**)
Exclusion mutuelle - attentes explicites - moniteurs

Documents HTML

Applets - animation, images, son

Applications réparties

Appel de méthodes d'objets distants (**`RMI`**)

Ouvrages de références :

Java in a Nutshell - David Flanagan - O'Reilly & Associates Inc.
Thinking in Java - Bruce Eckel - http://www.EckelObjects.com

Travaux dirigés de JAVA

Programmation par objets

classes - encapsulation

```
class Decompteur {
public static final int initParDefaut= 100;   constante
private int i;                                 variable d'état
                                               initialisations (constructeurs)
public Decompteur(int valInit) {i=valInit;}
public Decompteur() {i=initParDefaut;}
                                               opérations (méthodes)
public void dec() { if (i!=0) {i--;} }
public boolean estNul() {return(i==0);}
}
```

objets

déclaration d'identificateur

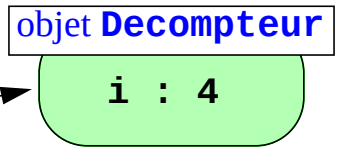
```
Decompteur reveil;
```

référence

réveil | null

création d'objet

```
reveil = new Decompteur(4);
```



réveil

déclaration + création

```
Decompteur reveil = new Decompteur(4);
```

usage d'un objet

```
reveil.dec();
```

instance courante : `reveil`

citation explicite : `this` `i ≡ this.i`

Travaux dirigés de JAVA

Programmation par objets

composants statiques : non liés à un objet

```
class Decompteur {
                                        donnée statique
public static final int initParDefaut= 100;
private int i;
...                                     procédure statique
public static Decompteur max(Decompteur d1, Decompteur d2) {
  if (d1.i>d2.i) {return d1;} else {return d2;}
}
}
```

utilisation d'un composant statique

```
int k = Decompteur.initParDefaut;
Decompteur.max(reveil1, reveil2).dec();
```

Travaux dirigés de JAVA

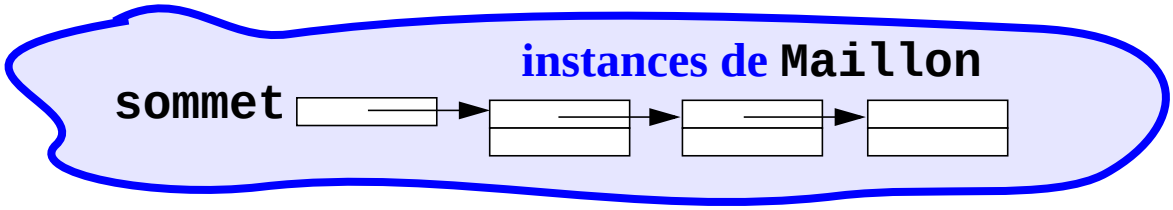
Programmation par objets

Classe interne

aspect statique : modularité

peut être rendue invisible depuis l'extérieur par **`private`**

exemple : pile réalisée au moyen de maillons chaînés




```
class PileEnt {
private static class Maillon {
  int elt; Maillon suivant;
  Maillon(int e,Maillon s) {elt=e; suivant=s;}
}
private Maillon sommet;
public PileEnt() {sommet=null;}
public void empiler(int e) {
  sommet=new Maillon(e,sommet);
}
public void depiler() {sommet=sommet.suivant;}
public int sommet() {return sommet.elt;}
}
```

Travaux dirigés de JAVA

Classe interne

aspect dynamique : classe liée à un objet

accès aux composants de l'objet courant de la classe englobante
⇒ ***une nouvelle classe pour chaque objet de la classe englobante***

exemple : pile dotée de moyens de parcours

```
class PileEnt {
private int s; private int[] P = new int[100];
public PileEnt() {s=-1;}
public void empiler(int e) {s=s+1; P[s]=e;}
public void depiler() {s=s-1;}
public int sommet() {return P[s];}

public class Parcours {
 private int courant;
 public Parcours() {courant=s;}
 public int element() {return P[courant];}
 public void suivant(){courant--;}
 public boolean estEnFin(){return courant==-1;}
}}
```

deux parcours sur p

```
 PileEnt p= new PileEnt(); ...
 PileEnt.Parcours parc1= p.new Parcours();
 PileEnt.Parcours parc2= p.new Parcours();
 parc1.suivant(); parc2.suivant(); ...
```

Traitements d'exception

`throw-try-catch`

Pour les ***cas exceptionnels***, ou les ***cas d'erreurs***
sans ce mécanisme : les procédures doivent rendre des résultats supplémentaires, qu'il faut tester ... ⟶ alourdit la programmation

En Java, une exception peut être ***déclenchée***

- par l'interpréteur du langage :

`ArithmeticException`	*division par 0*
`NullPointerException`	*accès à* **`null`**
`IndexOutOfBoundsException`	*accès tableau hors bornes*
`IOException`	*erreur d'entrée-sortie*
`NumberFormatException`	*erreur de format*

- par une instruction : `throw e;`
  `e` : objet de classe de type **`Exception`**,
  défini par le programmeur, destiné à préciser l'exception

Une procédure génératrice d'exception doit être déclarée avec une rubrique `throws` :

```
int ppp() throws xxxException { ... }
```

Traitements d'exception

Pour utiliser une procédure génératrice d'exception ***il faut*** :

`q()` : susceptible de déclencher une exception

soit ***capter et traiter l'exception*** : *bloc* **`try`**

```
void p() {
       ...
       try {... q() ...}
       catch(xxException e){... traitement ...}
       ...
}
```

soit ***propager l'exception*** : *profil* **`throws`**

```
void p() throws xxException {
       ...
       q()
       ...
}
```

Traitements d'exception

l'objet de type **`Exception`** transmis par **`throw e ;`**
est reçu lors du captage de l'exception :
`catch (MachinException e) { ...` traite **`e ...}`**
il sert à transmettre un ***diagnostic*** sur la cause de l'exception

définition d'un type **`Exception`** : doit ***hériter*** du type **`Exception`**

```
class MachinException extends Exception {
   typeAnomalie quoiQuiVaPAs;
   MachinException(typeAnomalie x) {
       quoiQuiVaPas=x;
   }
}
```

déclenchement d'exception :

```
void q() throws MachinException {
  ... throw new MachinException(çaVaPas) ...
}
```

captage d'exception :

```
void p() { ...
  try {... q() ...}
  catch(MachinException e){
       traitement(e.quoiQuiVaPas)
  }
}
```

Traitements d'exception

voici un exemple simple pour illustrer le mécanime :

la fonction **`decodeCouleur`** est génératrice d'exceptions

```
class CouleurException extends Exception { type exception
  public String mauvaiseCouleur;
  public CouleurException(String s){mauvaiseCouleur=s;}
}

class TestExceptions {

  static final int NOIR=1; static final int ROUGE=2;

  static int decodeCouleur(String s) throws CouleurException {
  // résultat : couleur représentée par la chaîne s
  // "noir" pour NOIR, "rouge" pour ROUGE exception sinon
    if (s.equals("noir")) {return NOIR;}
    else if (s.equals("rouge")) {return ROUGE;}
    else {throw new CouleurException(s);}
  }

  public static void main(String[] n){
    int couleurLue; boolean couleurCorrecte=false;
    while(!couleurCorrecte){
      try {
        couleurLue=decodeCouleur(Lecture.chaine());
        couleurCorrecte=true;
      }
      catch(CouleurException ce){
        System.out.println(
          ce.mauvaiseCouleur+" n'est pas une couleur");
      }
    }
    System.out.println("couleur correcte");
  }
}
```

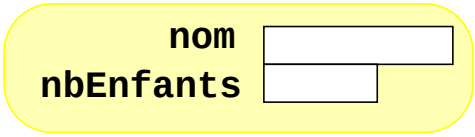
Traitements d'exception

Programmation par objets

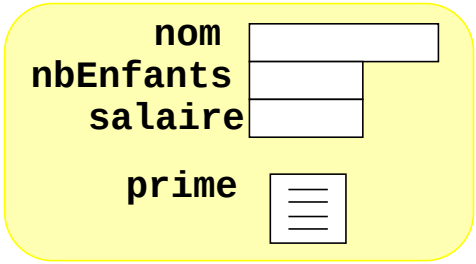
Héritage

```
class Personne {                    classe de base
String nom;
int nbEnfants;
Personne(String n) {
nom=n; nbEnfants=0;
}
}

                          classe dérivée

class Salarie extends Personne {
int salaire;
int prime() {
  return 5*salaire*nbEnfants/100;
}
Salarie (String n, int s){
  super(n); salaire=s;
}
}
```

une personne



un salarié



compatibilité entre types

B hérite de **A** ⇨ un objet de type **B** ***est*** également de type **A**

⇨ toute opération de classe **A** est applicable à un **B**

accessibilité des composants

aucun attribut : par les classes du même paquetage
`public` : par toutes les classes
`protected` : par les classes ***dérivées*** et les classes du même paquetage
`private` : par aucune autre classe

Traitements d'exception

test de classe

test de type au moment de l'exécution :

e **`instanceof`** *c*

rend **`true`** si l'expression ***e*** désigne un objet de classe ***c***
rend **`false`** sinon

☛ **usage pratique mais détourné de l'héritage : union de types**

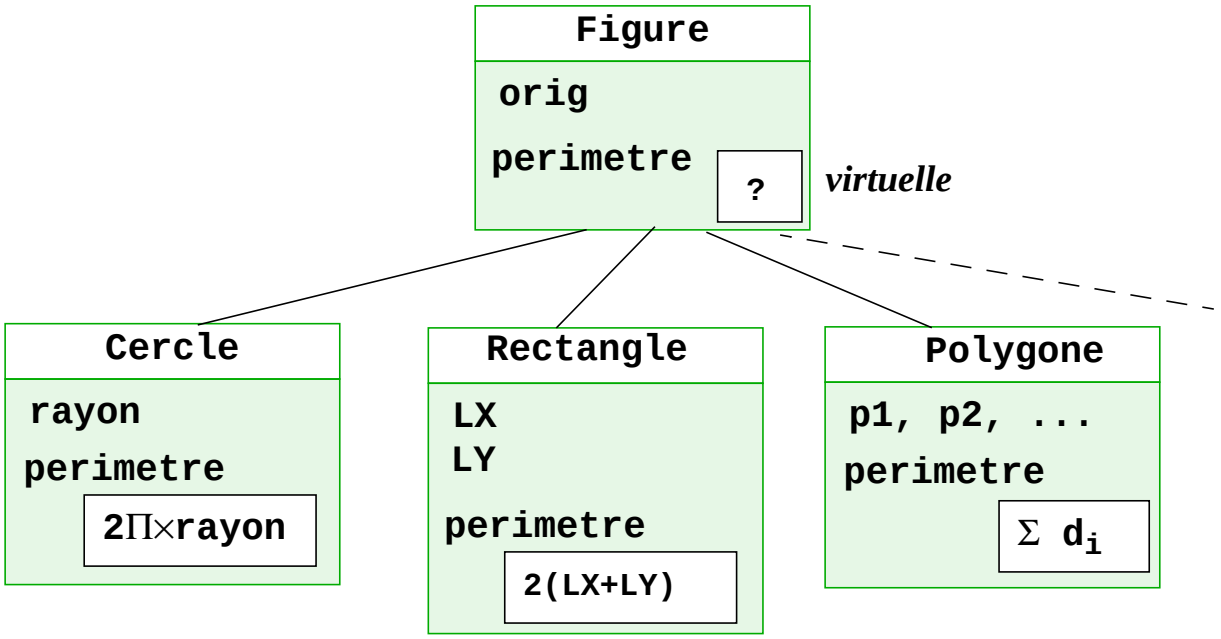
type de base = union des types dérivés

Traitements d'exception

Programmation par objets

méthodes virtuelles - classes abstraites

prévoir des ***opérations similaires***
sur des objets d'***espèces différentes***



toute méthode est potentiellement virtuelle (sauf attribut **`final`**)

méthode abstraite : méthode annoncée mais non définie

classe abstraite : possède des méthodes abstraites

Traitements d'exception

Programmation par objets

méthodes virtuelles - classes abstraites

```
abstract class Figure {  classe abstraite
  Point orig;
  Figure(Point o) {orig=new Point(o);}
  abstract float perimetre();  méthode abstraite
}



class Cercle extends Figure {
  private static final float pi=3.141592;
  float rayon;
  Cercle(Point centre, float r) { super(centre); rayon=r;}
  float perimetre() {return 2*pi*rayon;}
}


class Rectangle extends Figure {
  float LX; float LY;
  Rectangle(Point coin, float lx, float ly) {
    super(coin); LX=lx; LY=ly;
  }
  float perimetre() {return 2*(LX+LY);}
}


class Polygone extends Figure {
  ...
  float perimetre() { ... }
}
```

Traitements d'exception

Programmation par objets
Structures de données génériques

type paramétré par d'autres types
`Pile(T)` piles d'éléments de type quelconque **`T`**
instanciations : **`Pile(int)`**, **`Pile(char)`**, **`Pile(Figure)`**

réalisation en Java : **`class A<T1,T2..> { … }`**

```
class Pile<T> {          pile générique d'éléments de type T
  private int  s;
  private Object[] P= new Object[100];
  public Pile() {s=-1;}
  public void empiler(T e) { s++; P[s]=e;}
  public void depiler() { s--;}
  public T sommet() {return (T) P[s];}
  public boolean estVide() {return s==-1;}
}
```

```
    Etudiant toto = new Etudiant("toto",...);

    Pile<Etudiant> pEtu = new Pile<Etudiant>();
    pEtu.empiler(toto);
```

la version 1.5 de java de permet pas de créer des tableaux de type générique
On est obligé d'écrire `Object[] P= new Object[100]`
au lieu de `T[] P= new T[100]`
et pratiquer un cast `(T) P[s]` lors de l'extraction de données de la pile

Traitements d'exception

Méthodes génériques

Exemple : fonction générique qui rend en résultat la représentation en clair des éléments d'un tableau de type quelconque :

```
static <TypeElt> String enClair(TypeElt[] tab){
  if (tab.length==0){return "";}
  String resul=""+tab[0];
  for(int i=1;i<tab.length;i++){
    resul=resul+" "+tab[i];
  }
  return resul;
}
```

usage fréquent : ***fonctions statiques de classes génériques***

Java ne permet pas d'utiliser les paramètres de généricité dans les fonctions statiques. On rédige alors une fonction statique générique
Exemple : classe générique **`Paire`** dotée d'une fonction statique **`aPartirDe(Tp p,Ts s)`** qui rend en résultat une paire constituée des éléments **`p`** et **`s`** :

```
class Paire<T1,T2> {
  private T1 premier; private T2 second;
  private Paire(T1 p,T2 s){premier=p; second=s;}
  public T1 getPremier() {return premier;}
  public T2 getSecond() {return second;}
  public static <Tp,Ts> Paire<Tp,Ts> aPartirDe(Tp p, Ts s){
            return new Paire(p,s);
  }
}
```

Généricité et types primitifs - *Autoboxing*

En java, les paramètres de généricité sont ***nécessairement des classes*** non des types primitifs (**`char`**, **`int`**, **`long`**, **`float`**, **`double`**...)

Pour des structures de données génériques d'éléments d'un type primitif, il faut ***encapsuler*** ce type primitif dans une classe. Par exemple, pour une pile d'entiers, il faut utiliser une classe **`Integer`** :

```
class Integer {
  private int v;
  public Integer(int i) {v=i;}
  public int intValue() {return v;}
}
```

et utiliser une **`Pile`** de **`Integer`** :

```
Pile<Integer> p = new Pile<Integer>();
```

L'utilisation d'une telle pile est cependant un peu lourde :

```
  p.empiler(new Integer(12));
  p.empiler(new Integer(14));

  while (!p.estVide()) {
    int i = p.sommet().intValue();
    System.out.print(" "+ i);
    p.depiler();
  }
```

Traitements d'exception

Généricité et types primitifs - *Autoboxing*

java offre l'***autoboxing*** (enveloppement automatique) :

classes d'objets correspondant aux types primitifs :
`Integer`, **`Long`**, **`Short`**, **`Byte`**, **`Float`**, **`Double`**, **`Character`**, **`Boolean`**

Une expression ***e*** d'un type primitif ***t*** est permise partout où son correspondant objet ***T*** est attendu.
Cette expression est considérée comme équivalente à **`new`** ***`T(e)`***

Exemple :
on peut écrire **`p.empiler(12)`**
le compilateur le traduit en **`p.empiler(new Integer(12))`**.

Partout où une expression de type primitif ***t*** est permise, on peut utiliser une expression ***E*** du type d'objet correspondant.
Cette expression est considérée comme équivalente à ***`E.t`*****`Value(`**

Exemple :
on peut écrire **`int i= p.sommet()`**
le compilateur le traduit en **`int i= p.sommet().intValue()`**

Traitements d'exception

Algorithmes génériques - interfaces

algorithme générique : qui peut s'appliquer à des paramètres de types divers dotés de certaines opérations
exemple simple et classique : le tri d'un tableau
le type des éléments doit disposer d'une fonction qui indique si un élément est ***inférieur*** à un autre.

Java offre la notion d'***interface*** pour exprimer le besoin d'existence de certaines opérations.

Interface :

- collection de déclarations de méthodes
- cas limite de classe abstraite :
  aucun corps de méthode, ni aucun attribut de données

rôle principal :
établir un "contrat" ou encore un "cahier des charges" que des types d'objets devront satisfaire pour pouvoir participer à certains traitements.

Traitements d'exception

Définition d'interface

exemple : ***parcoureur de collection***
un tel objet doit se comporter comme un "curseur" qui repère un élément au sein d'une collection et qui permet de la parcourir

```
interface Parcours<TypeElement> {

  public void tete();
  // effet : positionne this en début de collection
  // ou en fin de parcours si la collection est vide

  public void suivant();
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément suivant

  public boolean estEnFin();
  // résultat : indique si this est en fin de parcours
  // (au delà du dernier)

  public TypeElement elementCourant();
  // prérequis : this n'est pas en fin de parcours
  // résultat : l'élement désigné par this
}
```

Une interface peut être générique. C'est le cas ici : le type des éléments est indiqué par le paramètre de généricité **`TypeElement`**. Ainsi l'interface **`Parcours`** pourra être mise en œuvre pour des collections d'éléments de type quelconque

Traitements d'exception

Mise en œuvre d'interface

une classe peut ***mettre en œuvre*** une interface : **`implements`**
Exemple : mise en œuvre de **`Parcours`** pour parcourir des tableaux

```
class ParcoursDeTableau<TypeElement>
                        implements Parcours<TypeElement> {

  private TypeElement[] T; // tableau objet du parcours
  private int i; // indice courant

  public ParcoursDeTableau(TypeElement[] T) {
  // parcours sur T initialisé à l'indice 0
    this.T=T; i=0;
  }
  public void tete(){
  // effet : positionne this en début de collection
    i=0;
  }
  public void suivant(){
  // prérequis : this n'est pas en fin de parcours
  // effet : positionne this sur l'élément suivant
    i++;
  }
  public boolean estEnFin(){
  // résultat : indique si this est en fin de parcours
  // (au delà du dernier)
    return i==T.length;
  }
  public TypeElement elementCourant(){
  // prérequis : this n'est pas en fin de parcours
  // résultat : l'élement désigné par this
    return T[i];
  }
}
```

Mise en œuvre d'interface

autre mise en œuvre de **`Parcours`** : pour les listes

```
public class Liste<T> {

  private static class Maillon<TE> {
    Maillon<TE> suivant; TE element;
    Maillon(Maillon<TE> s, TE e) {suivant=s;element=e;}
  }

  private Maillon<T> tete;
  private Maillon<T> queue;

  //========= parcoureur de liste ====================
  public class ParcoursDeliste implements Parcours<T>{
    private Maillon<T> courant;
    public ParcoursDeListe() {courant=tete;}
    public void tete() {courant=tete;}
    public void suivant() {courant=courant.suivant;}
    public boolean estEnFin() {return courant==null;}
    public T elementCourant() {return courant.element;}
  }//================================================

  public Liste() { // liste vide
    tete=null; queue=null
  }

  public Parcours<T> nouveauParcours() {
  // résultat : nouveau parcours initialisé au début
    return new ParcoursDeListe();
  }
  public void ajouteEnQueue(T nouvelElement) {
  // effet : ajoute nouvelElement en queue de this
    ...
  }
}
```

Utilisation d'interface

Une interface ***I*** s'utilise comme une classe abstraite :

- on peut déclarer des variables, des résultats ou des paramètres de type ***I***
- toute instance d'une classe qui implémente ***I*** est compatible avec ces variables, résultats ou paramètres

Une classe peut mettre en œuvre un nombre quelconque d'interfaces :

`class A implements` ***Interface1, Interface2...*** **`{ ... }`**

ceci remplace en partie l'héritage multiple qui n'existe pas en Java

les algorithmes rédigés en termes d'interfaces sont très généraux

Exemple :
affichage d'une collection sans connaître son type précis
il suffit de pouvoir la parcourir

```
static <TE> void afficheCollection(Parcours<TE> p){
  p.tete();
  while(!p.estEnFin()){
    System.out.println(p.elementCourant());
    p.suivant();
  }
}
```

Traitements d'exception

Algorithmes génériques : interfaces pour exprimer des exigences sur les paramètres

exemple: algorithme de tri de tableau.
le type des éléments peut être quelconque, à condition qu'on puisse décider si un élément est ***inférieur*** à un autre
Pour exprimer cette contrainte sur le type des éléments on définit une interface

```
interface Ordonnable<T> {
  public boolean inferieur(T x);
  // résultat : indique si this est inférieur à x
}
```

l'algorithme de tri exige que le type des éléments mette en œuvre cette interface :

```
public static <T extends Ordonnable<T>> void trier(T[] tab){
// effet : tri tab par ordre croissant selon l'ordre
// défini par la méthode inferieur des éléments de tab
  for (int i=tab.length-1; i>=0; i--) {
    for (int j=1; j<=i; j++) {
      if (tab[j].inferieur(tab[j-1])) {
        T x=tab[j-1]; tab[j-1]=tab[j]; tab[j]=x;
      }
    }
  }
}
```

Remarque : **`T extends Ordonnable<T>`** et non **`implements`**

Traitements d'exception

Algorithmes génériques

Pour trier un tableau d'éléments de type **`Personne`**,
la classe **`Personne`** doit mettre en œuvre l'interface **`Ordonnable`**
choix (arbitraire) : l'ordre alphabétique de leur nom.

```
class Personne implements Ordonnable<Personne>{
  public String nom; public int age; public int poids;
  public Personne(String n, int a, int p){
    nom=n; age=a; poids=p;
  }
  public String toString(){
    return "<"+nom+","+age+" ans, "+poids+" kg>";
  }
  public  boolean inferieur(Personne p){
  // résultat : indique si this est avant y par
  // ordre alphabétique de leurs noms
    return nom.compareTo(p.nom)<0;
  }
}
```

exemple d'utilisation :

```
  Personne[] peuple = { new Personne("toto", 25, 80),
                        new Personne("tutu", 53, 65),
                        new Personne("tata", 15, 47),
                        new Personne("jojo", 12, 30)
                       };
  System.out.println("peuple = "+enClair(peuple));
  trier(peuple);
  System.out.println("peuple trié selon leur nom = "
                                    +enClair(peuple));
```

Algorithmes génériques

L'exigence d'une relation d'ordre est un besoin fréquent :
une interface **`Comparable`** est définie dans la bibliothèque standard

```
Interface Comparable<T> {
  public int compareTo(T x);
  // résultat : <0, =0 ou >0 selon que this est
  // respectivement inférieur, égal ou supérieur à x
}
```

Les classes **`Integer`**, **`Double`**, **`String`**... mettent en œuvre l'interface **`Comparable`**.

Algorithmes génériques
interfaces pour passer des fonctions en paramètre

moyen plus général d'exprimer des algorithmes génériques

exemple : pour trier un tableau selon plusieurs critères, il faut passer en paramètre la fonction de comparaison
une interface permet d'abstraire la fonction de comparaison :

```
interface Comparaison<T> {
  public boolean inferieur(T x, T y);
 // résultat : indique si x est inférieur à y
}
```

nouvelle version du tri : un paramètre supplémentaire **op** donne accès à la fonction de comparaison

```
public static <T> void trier(T[] tab, Comparaison<T> op) {
// effet : tri tab par ordre croissant selon l'ordre
// défini par la méthode inferieur de op
  for (int i=tab.length-1; i>=0; i--) {
    for (int j=1; j<=i; j++) {
      if (op.inferieur(tab[j],tab[j-1])) {
        T x=tab[j-1]; tab[j-1]=tab[j]; tab[j]=x;
      }
    }
  }
}
```

Algorithmes génériques
interfaces pour passer des fonctions en paramètre

pour trier les personnes selon leur âge :

```
class CompareAge implements Comparaison<Personne>{
  public boolean inferieur(Personne x, Personne y){
    return x.age<y.age;
  }
}
```

pour trier les personnes selon leur poids :

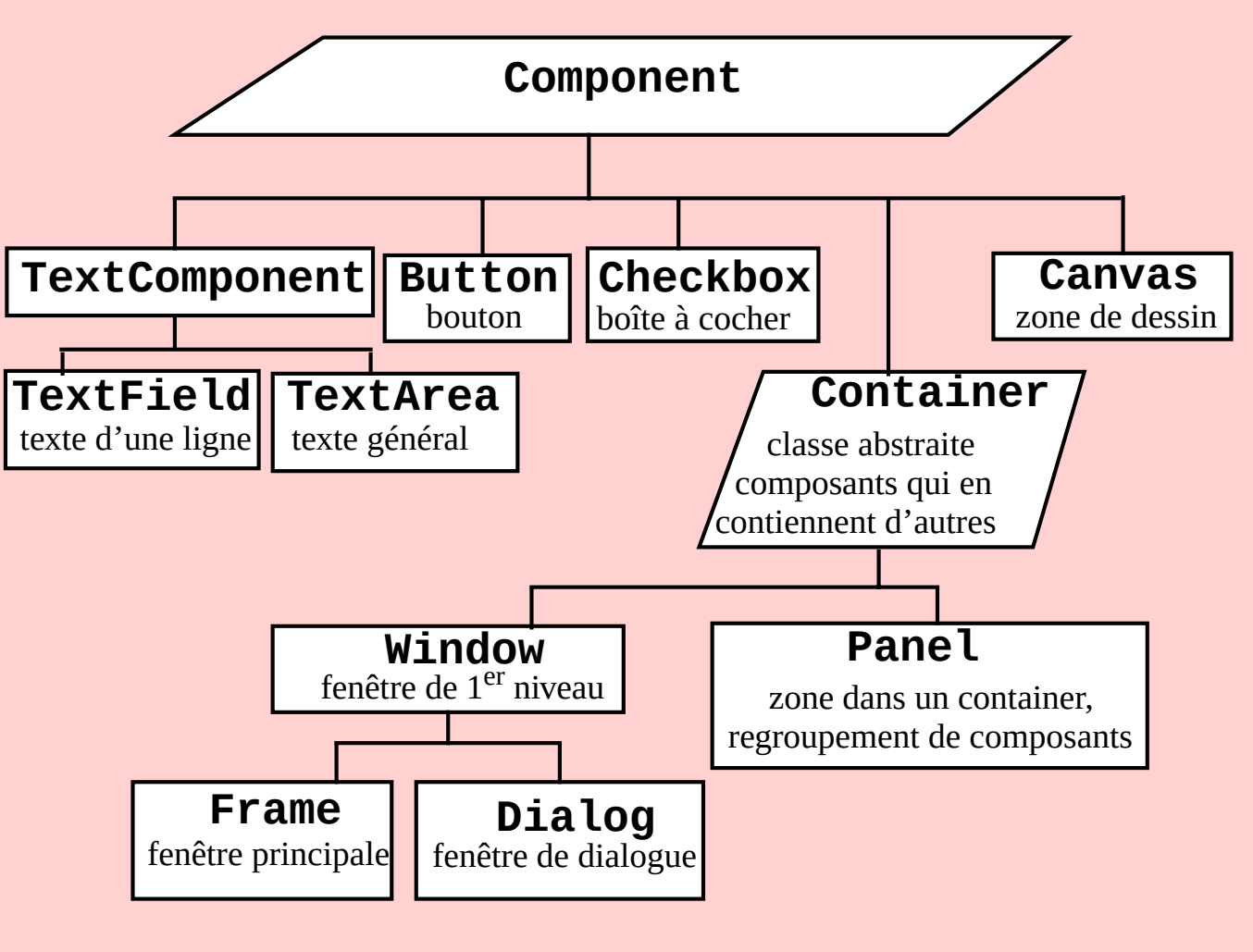
```
class ComparePoids implements Comparaison<Personne>{
  public boolean inferieur(Personne x, Personne y){
    return x.poids<y.poids;
  }
}
```

utilisation :

```
  ...
  trier(peuple,new ComparePoids());
  System.out.println("peuple trié selon le poids = "
                                      +enClair(peuple));
  trier(peuple,new CompareAge());
  System.out.println("peuple trié selon l'âge = "
                                     +enClair(peuple));
```

Remarque : **`ComparAge`** (de même **`ComparPoids`**) est une pure fonction. Ses instances sont identiques. Pour éviter de créer une instance à chaque utilisation, on peut utiliser toujours la même, créée une seule fois.

Interfaces homme-machine (AWT)



Exemple

Traitements d'exception

```
import java.awt.*; import java.awt.event.*;
1  class FenetreCompteur {
   int compteur;

   Frame f;
2  Button boutonIncr=    new Button("+");
   Button boutonDecr=    new Button("-");
   Button boutonQuit=    new Button("quit");

3  TextField affichageCompteur = new TextField(7);

   class ActionIncr implements ActionListener {
   public synchronized void actionPerformed(ActionEvent e)
       {compteur ++; afficherCompteur();}
   }

4  class ActionDecr implements ActionListener {
   public synchronized void actionPerformed(ActionEvent e)
       {compteur --; afficherCompteur();}
   }

   class ActionQuit implements ActionListener {
   public synchronized void actionPerformed(ActionEvent e)
       {System.exit(0);}
   }

   void afficherCompteur() {
     affichageCompteur.setText(String.valueOf(compteur));
   }

   public FenetreCompteur(String nom) {
     f=new Frame("compteur "+nom); compteur=0;
5    Placement.p(f,boutonIncr,1,1,1,1);
     Placement.p(f,boutonDecr,1,2,1,1);
     Placement.p(f,boutonQuit,1,3,1,1);
     Placement.p(f,affichageCompteur,2,1,1,2);
6    boutonIncr.addActionListener(new ActionIncr());
     boutonDecr.addActionListener(new ActionDecr());
     boutonQuit.addActionListener(new ActionQuit());
7    f.pack(); f.setVisible(true);
     afficherCompteur();
   }}
```

composants

réactions aux événements

placement des composants

connexion des réactions aux événements

Interfaces homme-machine (AWT)

Gestion des événements

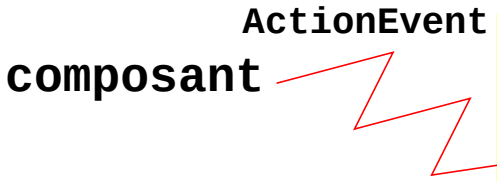
classe d'événement	composant générateur	signification
`ActionEvent`	`Button` :	cliquage
	`TextField` :	touche *Enter*
`MouseEvent`	`Component` :	mouvements et cliquage de souris
`KeyEvent`	`Component` :	enfoncement et relâchement de touche
`FocusEvent`	`Component` :	entrée et sortie du curseur de souris
`TextEvent`	`TextField`, `TextArea` :	modification du texte
`WindowEvent`	`Window` :	iconification, desiconification

événement ***`xxx`*`Event`**

↓

interface ***`xxx`*`Listener`**

```
ActionListener   :  actionPerformed(ActionEvent)
MouseListener    :  mouseClicked(MouseEvent)
                    mouseDragged(MouseEvent)
                    mouseMoved(MouseEvent)
KeyListener      :  keyPressed(KeyEvent)
                    keyReleased(KeyEvent)
                    keyTyped(KeyEvent)
```




```
class ReactionComposant
  implements ActionListener {
actionPerformed() { ... }
}
```

```
composant.addActionListener(new ReactionComposant())
```

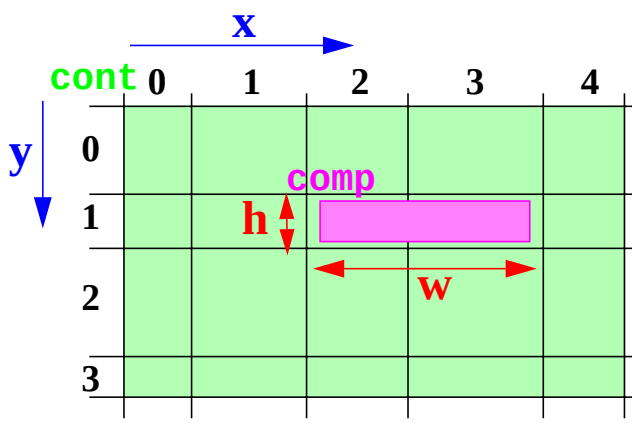
Interfaces homme-machine (AWT)

Placement des composants

gestionnaire de placement

exemple :

`GridBagLayout`




```
class Placement {

static GridBagLayout placeur= new GridBagLayout();
static GridBagConstraints c = new GridBagConstraints();

      // procedure generale de placement
public static void p( Container cont, Component comp,
                      int x, int y, int w, int h,
                      int cadrage, int t, int l, int b, int r,
                      double wx, double wy, int fill) {
  cont.setLayout(placeur);
  c.gridx=x; c.gridy=y; c.gridwidth=w; c.gridheight=h;
  c.fill=fill; c.anchor=cadrage;
  c.weightx=wx; c.weighty=wy;
  c.insets = new Insets(t,l,b,r);
  placeur.setConstraints(comp, c); cont.add(comp);
}
}
```

Programmation parallèle : Thread

Processus

programmation de processus : objet qui hérite de la classe **Thread**

méthode virtuelle **run()** programme principal du processus
méthode **start()** lance effectivement le processus

```
class A extends Thread {  processus impression de A1 A2...
  public void run() {
    for(int i=0; i<8; i++) { System.out.print("A"+i+" ");
      try {sleep(100);} catch (InterruptedException e) {};
    }
  }
}
class B extends Thread {  processus impression de B1 B2...
  public void run() {
    for(int i=0; i<8; i++) { System.out.print("B"+i+" ");
      try {sleep(200);} catch (InterruptedException e){}
    }
  }
}
public class TestThread1 {
  static public void main(String[] arg) {
    new A().start(); new B().start();
  }
}
```

création *lancement*

A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7

Traitements d'exception

Programmation parallèle : `Thread`

classe **`Thread`** : modèle de processus

plusieurs processus sur le même modèle

```
class Impr extends Thread {  modèle de processus impression
  String txt; int periode;
  public Impr(String t, int p){txt=t; periode=p;}
  public void run() {
    for(int i=0; i<8; i++) { System.out.print(txt+i+" ");
      try {sleep(periode);}catch(InterruptedException e){}
    }
  }
}

public class TestThread {
  static public void main(String[] arg) {
   new Impr("A",100).start(); new Impr("B",200).start();
  }
}
```

constructeur : paramètres de création qui particularisent les exemplaires

A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7

Programmation parallèle : Thread

Exclusion

limiter le parallélisme
pour qu'un objet ne subisse pas en même temps
plusieurs séquences d'actions

```
class Impr extends Thread {
String txt;
public Impr(String t){txt=t;}
public void run() {
  for(int j=0; j<2; j++) {
    for(int i=0; i<txt.length(); i++) {
      try {sleep(100);} catch (InterruptedException e) {};
      System.out.print(txt.charAt(i));
    };
  };
}}

public class TestThread {
static public void main(String[] arg) {
 new Impr("BONJOUR ").start(); new Impr("AU REVOIR ").start();
}}
```

impressions non coordonnées

BAOUN JROEUVRO IBRO NAJUO URRE VOIR

Programmation parallèle : `Thread`

Exclusion : `synchronized`

en Java : ***tout objet est susceptible d'être un verrou d'exclusion***

`synchronized(cetObj){` ... *instructions* ... **`}`**

retarde l'exécution tant qu'il y a une autre exécution en cours sous la coupe d'un **`synchronized(cetObj)`**

```
class Exclusion {}; pour créer de purs verrous d'exclusion

class PusImpr extends Thread {
static Exclusion exclusionImpression = new Exclusion();
String txt;
public PusImpr(String t){txt=t;}

public void run() {
for(int j=0; j<2; j++) {
  synchronized(exclusionImpression){ impression en exclusion
    for(int i=0; i<txt.length(); i++) {
      try {sleep(100);} catch(InterruptedException e){};
      System.out.print(txt.charAt(i));
    }
  }
};
```

verrou d'exclusion

`BONJOUR AU REVOIR BONJOUR AU REVOIR`

Traitements d'exception

Programmation parallèle : `Thread`

Moniteurs : **encapsule les opérations à réaliser sous exclusion**
objet d'exclusion : l'objet courant
exclusion sur l'exécution d'une méthode

```
class A {
     ...
     synchronized ... P(...) { ... }
     ...
}
```

équivalent à `... P(...) {synchronized(this){ ... }}`

modèle de moniteur

```
class MoniteurImpression {
  synchronized void imprTexte(String txt) {
  for(int i=0; i<txt.length(); i++) {
    try {Thread.currentThread().sleep(100);}
      catch(InterruptedException e) {};
      System.out.print(txt.charAt(i));
    }
  }
}
```

```
class Impr extends Thread {
  static MoniteurImpression m1 = new MoniteurImpression();
  String txt;
  public Impr(String t){txt=t;}
  public void run() {
    for(int j=0; j<2; j++) { m1.imprTexte(txt);};
  }
}
```

moniteur

appel du moniteur

Traitements d'exception

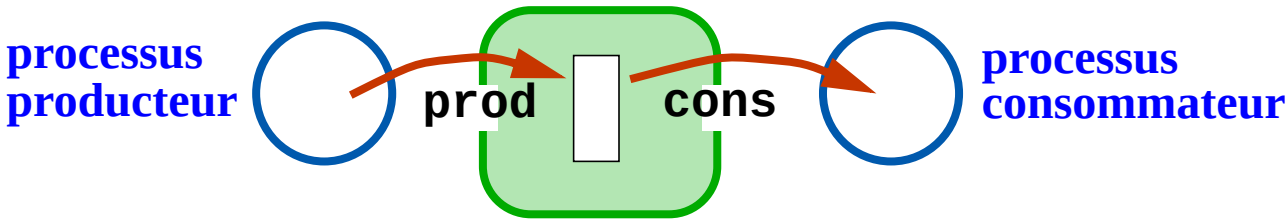
Attente explicite : `wait` - `notify`

Attente explicite

sous exclusion **`synchronized (obj)`**

`wait()` attente qu'une condition soit satisfaite
relâche l'exclusion sur **`obj`** afin que d'autres processus le réveillent

`notifyAll()` réveille ***tous les*** processus en attente sur **`obj`**
`notify()` réveille ***un*** processus en attente



tampon de production-consommation pour un seul producteur et un seul consommateur

```
class MoniteurProdCons {
String tampon; boolean estVide=true;

synchronized void prod(String m) {
  if(!estVide){
    try {wait();} catch(InterruptedException e) {};
  }
  tampon=m; estVide=false; notify();
}

synchronized String cons() {
  if(estVide){
    try {wait();} catch(InterruptedException e) {};
  }
  String resul=tampon; estVide=true; notify();
  return resul;
}}
```

attend tampon vide

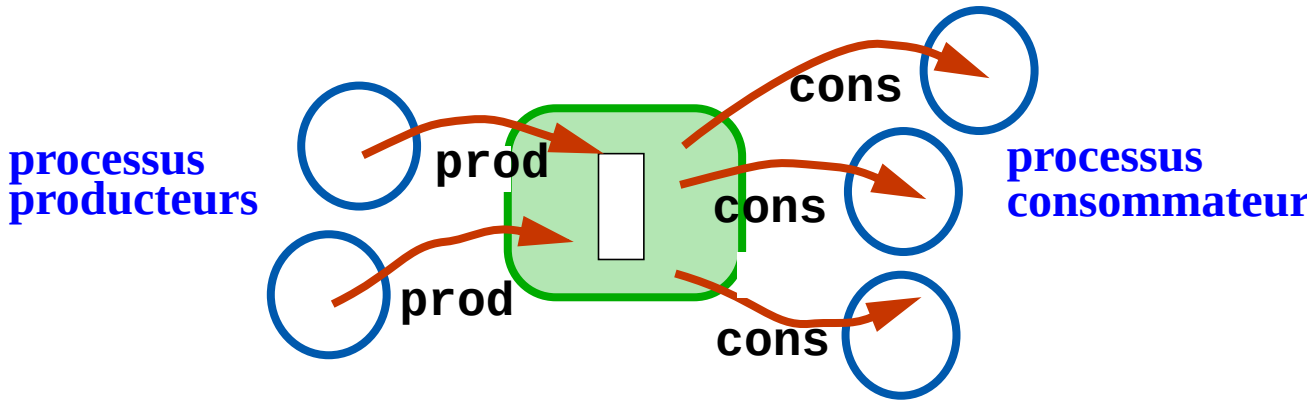
signale tampon non vide

attend tampon non vide

signale tampon vide

Traitements d'exception

Programmation parallèle : Thread




```
class MoniteurProdCons {  tampon de production-consommation
  String tampon; boolean estVide=true;

  synchronized void prod(String m) {
    while(!estVide){      attend tampon vide
      try {wait();} catch(InterruptedException e) {};
    }
    tampon=m; estVide=false; notifyAll();
  }                                          signale tampon non vide

  synchronized String cons() {
    while(estVide){ attend tampon non vide
      try {wait();} catch(InterruptedException e) {};
    }
    String resul=tampon; estVide=true; notifyAll();
    return resul;                              signale tampon vide
  }
}
```

Analogie avec les Moniteurs de Hoare

moniteur de Hoare plusieurs conditions c_1, c_2

méthododologie : prédicat **P** , condition c_p associée

aux endroits où **P** doit être vrai pour poursuivre :

... si (non P) alors attendre(c_p) fsi ; /* P est vrai */ ...

aux endroits où **P** devient vrai :

... /* P est vrai ici */ reprendre(c_p) ...

moniteurs de Java ***une seule condition***

☞ re-tester **P** après chaque attente
se remettre en attente si **P** est faux

```
... while (!P) {wait();} /* P est vrai */ ...
```

aux endroits où une assertion attendue devient vraie

```
/* P ou Q ou R*/  notifyAll(); ...
```

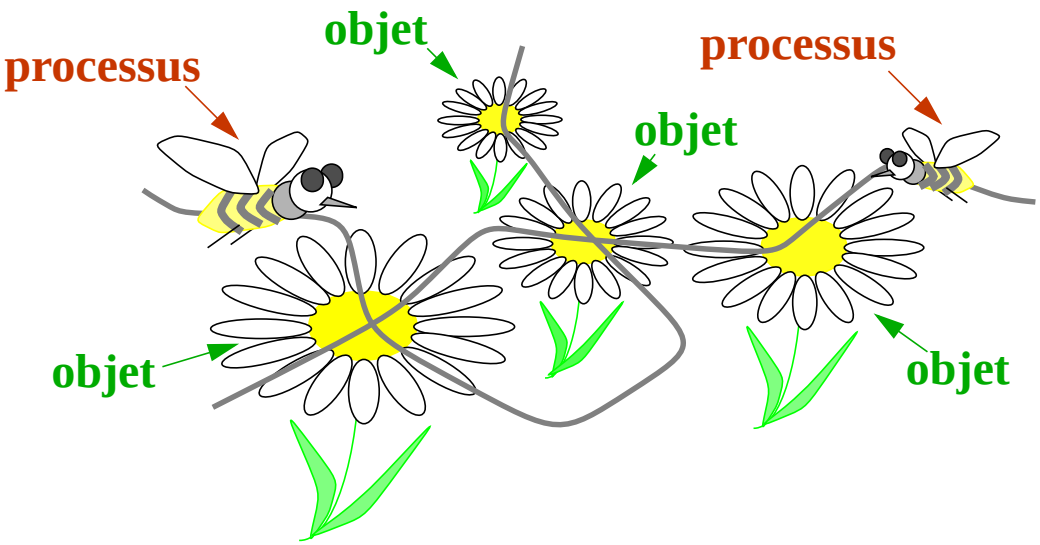
ou `notify()` dans certains cas

Traitements d'exception

Nature des processus

un processus n'est ***pas un objet*** : c'est une ***exécution***, une activité

objet associé de classe **`Thread`** : point de départ du processus



processus courant (l'abeille)

`... Thread.currentThread() ...`

rend l'objet **`Thread`** du processus qui l'exécute

l'objet courant (la fleur)

`this` l'objet courant sur lequel a lieu l'exécution

méthodes de la classe **`Thread`**

```
sleep(int t)          join()                stop()
int getPriority()     setPriority(int p)    ...
```

méthodes de la classe **`Object`**

```
wait()          notify()  notifyAll()
```

Traitements d'exception

Arrêt d'un processus depuis un autre processus

premières versions de Java : primitive **`stop()`**

p*`.stop()`** provoque la fin du processus ***p

quelque soit l'état dans lequel il se trouve

versions plus récentes : primitive **`stop()`** ***obsolète*** (*deprecated*)

bien que pratique, elle peut conduire à des programmes erronés

exemple : processus détruit alors qu'il est dans un moniteur

technique plus sûre : variable d'état testée par le processus en des points convenables de sa programmation

pour rendre ce principe systématique,

classe "processus stoppable" :

```
class StoppableThread extends Thread {
  private boolean stop;
  public StoppableThread(){stop=false;}
  public synchronized void ordreStop() {
     stop=true;
  }
  public synchronized boolean testeStop() {
     return stop;
  }
}
```

Traitements d'exception

Documents HTML : `Applet`

référencé depuis un document HTML
téléchargé et exécuté par un navigateur Web
dérive de la classe **`Applet`**

```
class Applet extends Panel {
```

méthodes à définir

`void init()`	*initialisations*
`void start()`	*le navigateur (r)ouvre le document*
`void stop()`	*le navigateur quitte le document*
`void paint(Graphic g)`	*effets visuels sur* **`Panel`** *au sein du document HTML*

méthodes à utiliser

```
void repaint()
URL getCodeBase();
URL getDocumentBase();
AudioClip getAudioClip(URL u, String nom);
Image getImage(URL u, String nom);
...
}
```

Traitements d'exception

fonctions graphiques ***de bas niveau***

```
class Graphics {
    void drawLine(int x1,int y1,int x2,int y2)
    void drawRect(int x,int y,int w,int h)
    void drawOval(int x,int y,int w,int h)
    void drawString(String s, int x,int y)
    void setColor(Color c)
    Color getColor()
    void setFont(Font f)
    Font getFont()
    boolean drawImage(...)
}
```

images (`.gif` ou `.jpg`)

```
Image im =
       getImage(getDocumentBase(),"vue.jpg");
g.drawImage(im,...);
```

sons (`.au`)

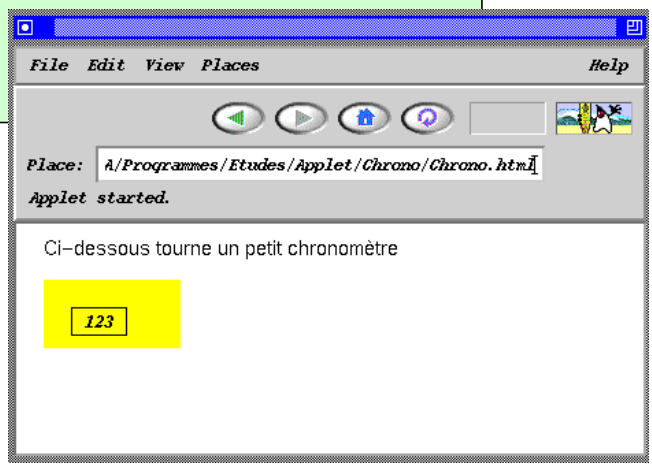
```
AudioClip m =
    getAudioClip(getDocumentBase(),"son.au");
m.play()
```

Traitements d'exception

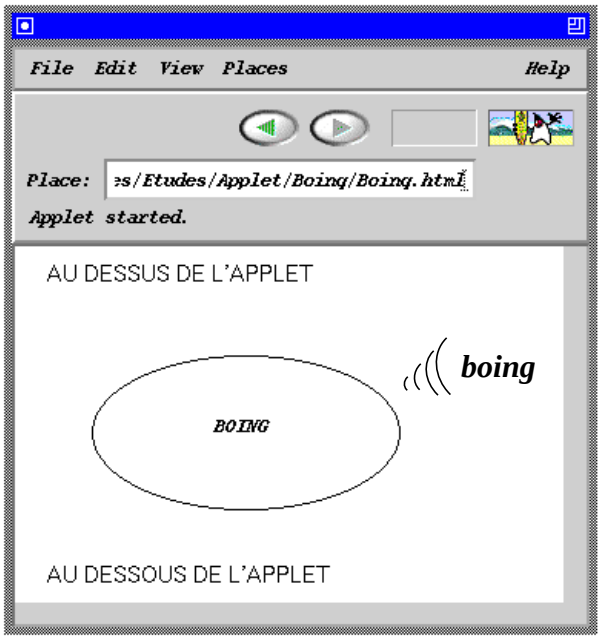
```
public class Chrono extends Applet {
int date;
class IncrementDate extends StoppableThread {
  public void run() {
    while (!testeStop()) {try {sleep(1000);}catch(... e){}
                    date++; Chrono.this.repaint();
      }
}}
IncrementDate incrDate;

public void init () {date=0;}
public void start () {incrDate = new IncrementDate();
                                        incrDate.start();}
public void stop () {incrDate.ordreStop();}

public void paint (Graphics g) {
   setBackground(Color.yellow); g.drawRect(20,20,40,20);
   g.drawString(String.valueOf(date),30,35);
}}
```

```
<HTML>
<HEAD> <TITLE> CHRONO </TITLE> </HEAD>
<BODY>
<P> Ci-dessous tourne un petit chronomètre </P>
<APPLET CODE= Chrono.class WIDTH=100 HEIGHT=50>
</APPLET>
</BODY>
</HTML>
```

document HTML



Exemple d'applet



Périodiquement, l'applet affiche une ellipse aplatie pendant 0,5 seconde puis une ellipse plus enflée tout en prononçant "boing" au moyen d'un fichier de son **`boing.au`**

Traitements d'exception

corrigé

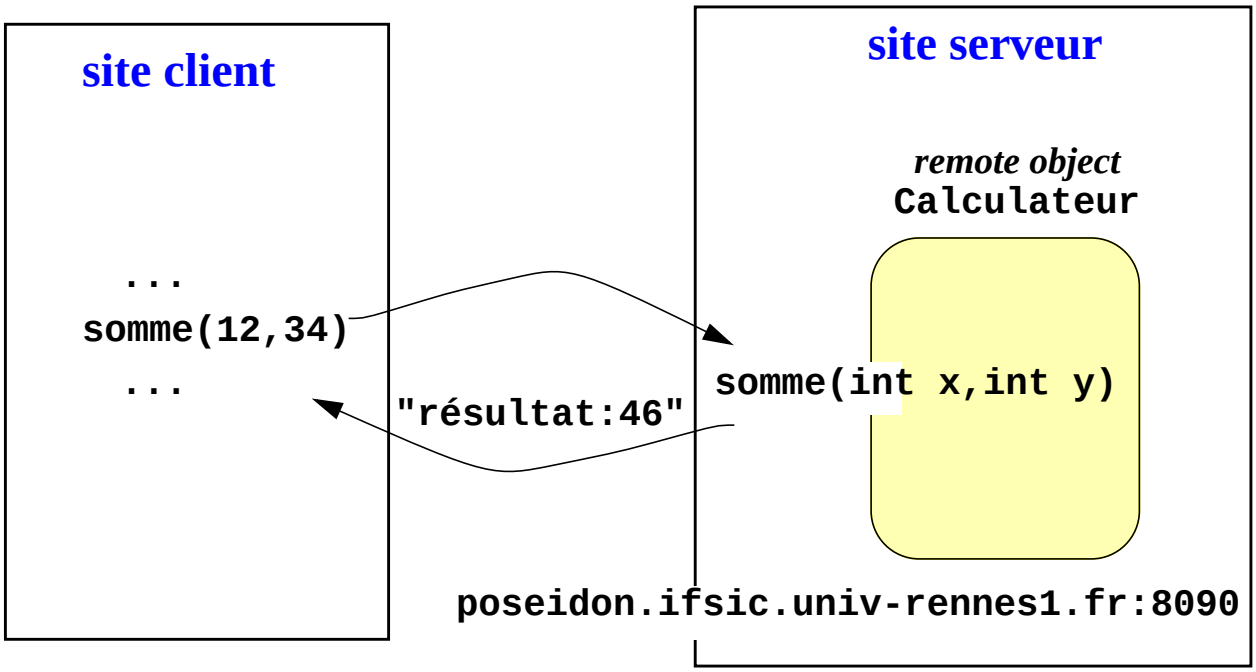
```
public class Boing extends Applet {

  static final int BOING=1; static final int PASBOING=2;
  int etat;// vaut altenativement BOING et PASBOING

  AudioClip boing; // element sonore

  Animation anim; // processus qui réalise l'animation

  class Animation extends StoppableThread {
     public void run() {
       while (!testeStop()) {
         try {sleep(500);} catch (Int..Exception e) {}
         if (etat==PASBOING) {etat=BOING; boing.play();}
         else {etat=PASBOING;}
         Boing.this.repaint();
       }
     }}

  public void init () {
     etat = PASBOING;
     boing=getAudioClip(getDocumentBase(),"Boing.au");
  }

  public void start () {
   anim = new Animation(); anim.start();
  }

  public void stop () { anim.ordreStop();}

  public void paint (Graphics g) {
    if (etat==PASBOING) {g.drawOval(30,60,200,30);}
    else{ g.drawOval(30,30,200,100);
          g.drawString("BOING",110,80);
    }
  }
}
```

Applications réparties : `rmi`

`rmi` : ***Remote Method Invocation***

objet "distant" (***remote object***)
appel de méthodes comme si l'objet était local

exemple



Traitements d'exception

Applications réparties : rmi

1/ Définition de l'interface d'un objet distant

```
public interface Calculateur extends Remote {
public String somme(int x,int y)
                    throws RemoteException;
}
```

2/ Définition d'une classe qui implémente l'objet distant

```
class ServeurCalculateur
                extends UnicastRemoteObject
                implements Calculateur {
public String somme(int x, int y) {
    return ("resultat : " + (x+y));
}}
```

3/ Initialisation d'un site serveur,
enregistrement de l'adresse externe d'un objet distant

4/ Connexion d'un client à un objet distant par son nom externe

5/ Compilation des souches et des squelettes : `rmic`
`stubs & skeletons`

Traitements d'exception

Applications réparties : rmi

3/ Initialisation d'un site serveur
enregistrement de l'adresse externe d'un objet distant

```
class TestServeurClaculateur {
public static void main(String[] argv) {
try {
    LocateRegistry.createRegistry(8090);
    ServeurCalculateur gaston = new ServeurCalculateur();
    Naming.bind(
    "//e103c04.ifsic.univ-rennes1.fr:8090/additionneur",
    gaston);
  }
  catch(Exception e) { System.out.println("erreur");}
}
}
```

processus d'enregistrement des associations
adresse externe - objet distant

`LocateRegistry.createRegistry(port)`

enregistrement d'une association

`Naming.bind(adresse externe, objet)`

Applications réparties : rmi

4/ Connexion d'un client à un objet distant par son nom externe

```
import java.rmi.*; import java.rmi.registry.*;
import es.*;

public class ClientCalcul {
public static void main(String[] argv){
  try {
    Calculateur calc =
     (Calculateur) Naming.lookup(
        "//e103c04.ifsic.univ-rennes1.fr:8090/additionneur");
    while (Lecture.unCarCmde() != '.') {
      int x = Lecture.unEntier();
      int y = Lecture.unEntier();
      System.out.println(calc.somme(x,y));
    }
  }
  catch(Exception e){
    System.out.println("erreur");
  }
}}
```

obtention de la référence à un objet distant
connu par son adresse externe

Naming.lookup()

utilisation de la référence rendue comme si l'objet était local

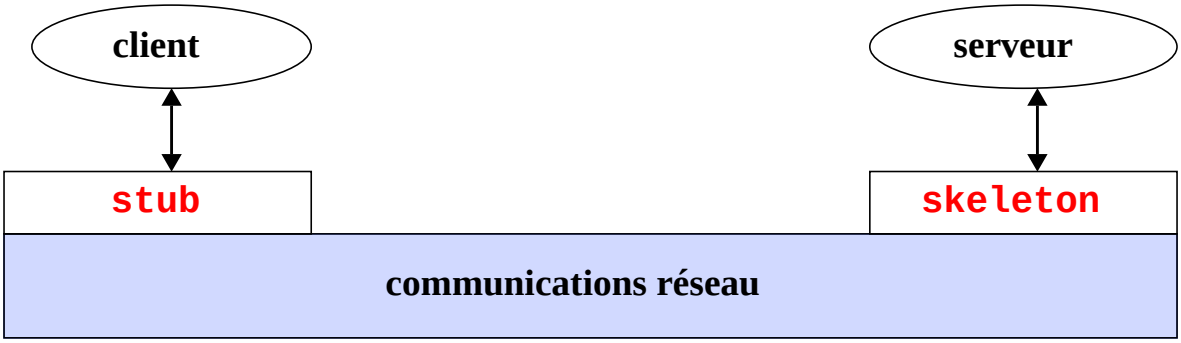
Applications réparties : rmi

5/ Compilation des souches et des squelettes : rmic

Traitements d'exception

stubs & skeletons

deux classes supplémentaires pour chaque classe d'objet distant :
la souche (**stub**) côté client :
émet les paramètres et récupère les résultats
le squelette (**skeleton**) côté serveur :
reçoit les paramètres et retourne le résultat



totalement transparent à la programmation
mais exige une compilation supplémentaire

```
rmic ServeurCalculateur
```

crée deux fichiers
ServeurCalculateur_Skel.class
ServeurCalculateur_Stub.class

Depuis **java 5** cette commande est devenue inutile :
les classes **_Skel** et **_Stub** sont
générées automatiquement lors du chargement

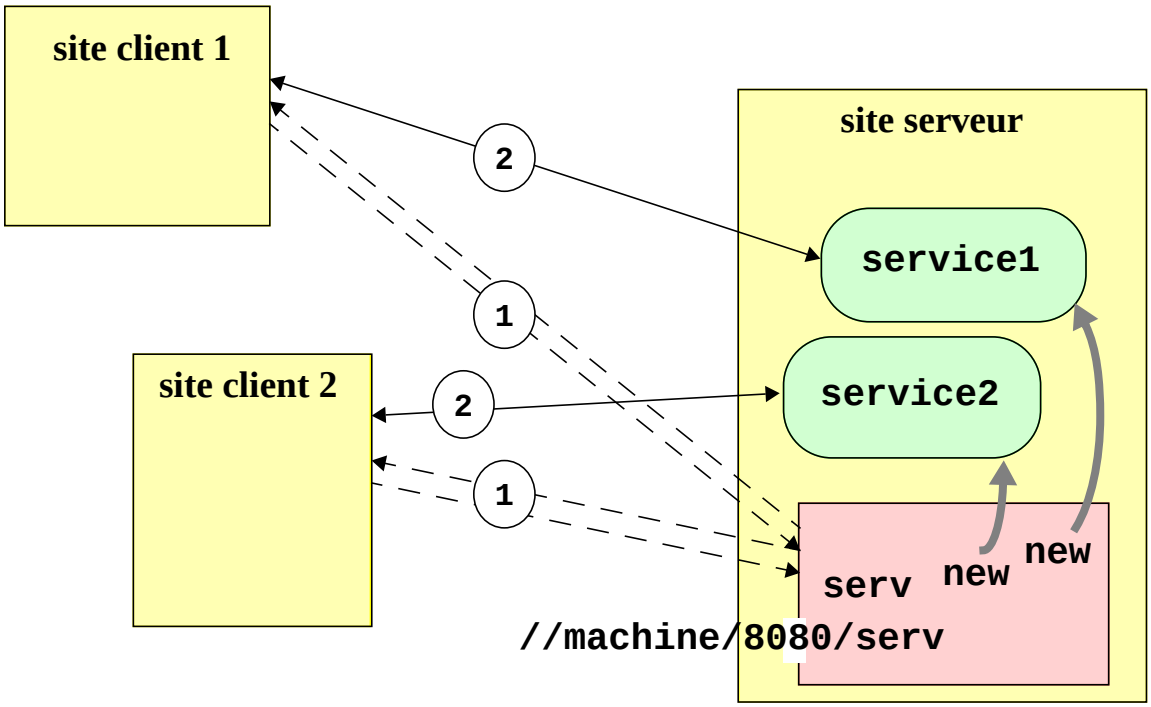
Traitements d'exception

Applications réparties : `rmi`

Communication de références d'objets distants

Cas général :
un ou quelques objets "serveurs" connus par une adresse externe
germe d'activités qui créent dynamiquement
des objets et se les communiquent par paramètres ou résultats

les objets distants
peuvent être passés en paramètre ou rendus en résultat
de méthodes d'objets distants



Traitements d'exception

Applications réparties : `rmi`

passage des paramètres et rendu de résultat

type de base (scalaire) : passage **par valeur**

type classe d'***objet distant*** (dérivé de **`Remote`**) : par référence à l'objet
type classe ***non distant*** : un **clone de l'objet** est réalisé en local

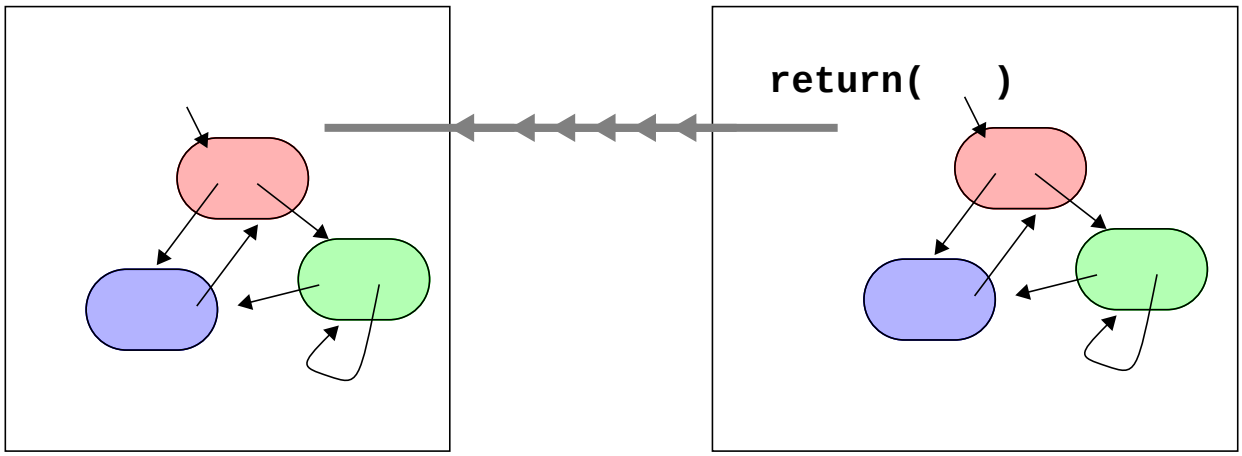
pour que le clonage soit possible :

```
interface Serializable {}
```

interface vide (il n'y a rien à rédiger de façon standard)

```
class UnObjetTransmissible
                implements Serializable {...}
```

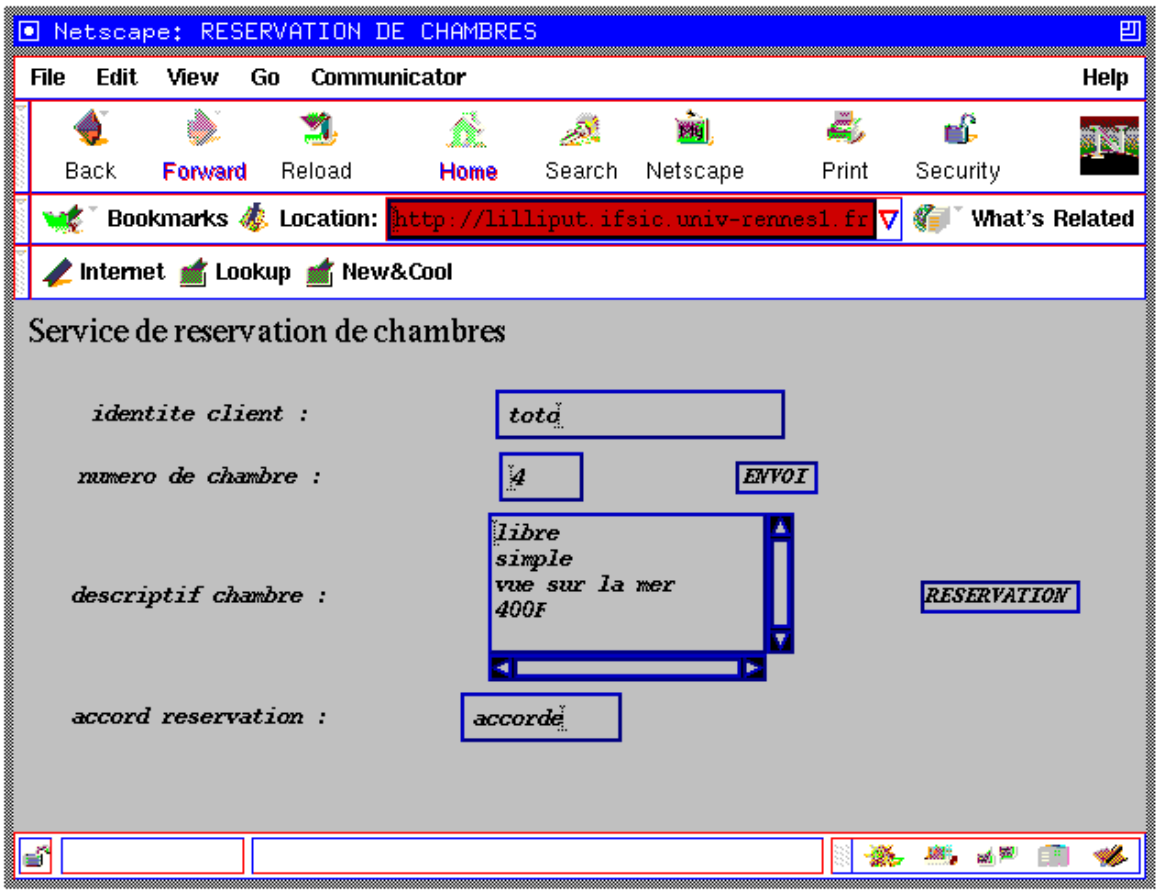
clonage astucieux : pour un réseau d'objets **`Serializable`** qui se référencent mutuellement, le graphe d'objets est transmis, ***en respectant la structure du graphe***



Traitements d'exception

Applications réparties : rmi

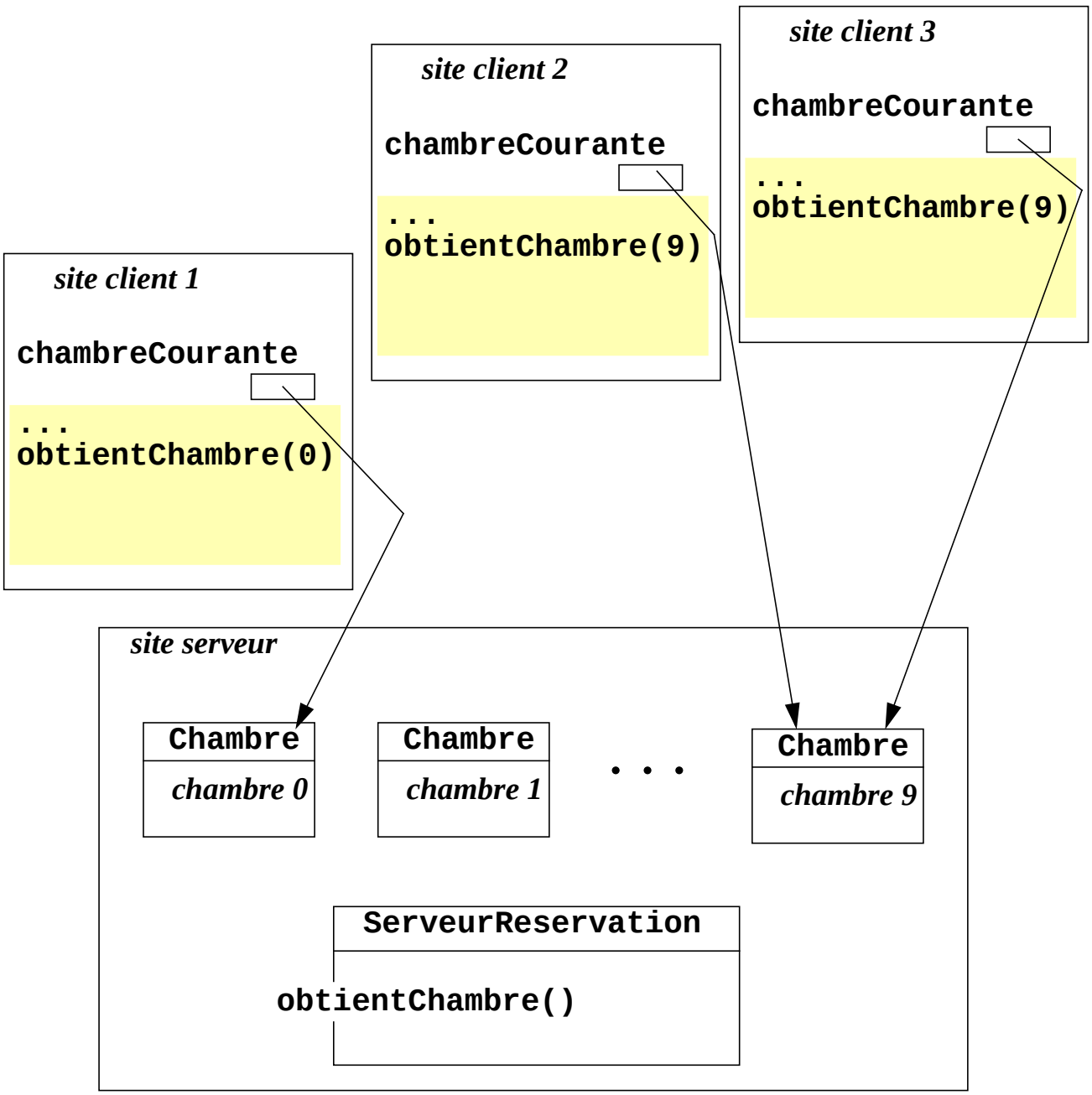
Exemple : réservation de chambres d'hôtel par internet




```
public interface InterfaceServeurReservation extends Remote{
public InterfaceChambre obtientChambre(int numChambre)
                                    throws RemoteException;
```

```
public interface InterfaceChambre extends Remote {
public String descriptif() throws RemoteException;
public boolean tentativeReservation(String identClient)
                                    throws RemoteException;
}
```

Traitements d'exception



Traitements d'exception

Le serveur

Objet unique
connu par nom externe **"//machine:port/leServeur"**

```
public class ServeurReservation extends UnicastRemoteObject
                implements InterfaceServeurReservation {

public static Chambre[] chambre = new Chambre[10];

public  ServeurReservation() throws RemoteException {
 chambre[0]= new Chambre("simple", "vue sur la rue", 200);
 chambre[1]= new Chambre("double", "vue sur la rue", 300);
    ...
    ...
 chambre[9]= new Chambre("simple", "vue sur la mer", 350);
}

public InterfaceChambre obtientChambre(int numeroChambre) {
    try {return chambre[numeroChambre];}
    catch (ArrayIndexOutOfBoundsException e) {return null;}
}

public static void main(String[] arg) {
   try {
     LocateRegistry.createRegistry(8090);
     ServeurReservation leServeur= new ServeurReservation();
     Naming.bind(
       "//florence.irisa.fr:8090/leServeur",leServeur);
   }
   catch(Exception e) {System.out.println("erreur");}
}
}
```

Traitements d'exception

Les chambres

Objets distants captés par références rendues en résultat

```
class Chambre extends UnicastRemoteObject
                  implements InterfaceChambre {
private final String capacite;
private final String vue;
private int tarif;
String etatReservation="libre";

public Chambre(String capacite, String vue, int tarif)
                                      throws RemoteException {
  this.capacite=capacite; this.vue=vue; this.tarif=tarif;
}

public String descriptif() throws RemoteException {
   return
     etatReservation+"\n"+capacite+"\n"+vue+"\n"+tarif+"F";
}

public synchronized boolean
              tentativeReservation(String identClient)
                            throws RemoteException {
  if(etatReservation.equals("libre")) {
     etatReservation="occupe par " + identClient;
     return true;
  }
  else {return false;}
}
}
```

Traitements d'exception

Les clients

```
public class ClientReservation extends Applet {

TextField identClient = new TextField(20);
TextField numChambre = new TextField(4);

Button boutonEnvoiNumChambre = new Button("ENVOI");

class ActionEnvoiNumChambre implements ActionListener {
  public synchronized void actionPerformed(ActionEvent e) {
    int numChambre;
    try {
     numChambre = Integer.parseInt(numChambre.getText());
     chambreCourante=
             serveurReservation.obtientChambre(numChambre);
     descriptif.setText(chambreCourante.descriptif());
   } catch(Exception ex){};
   accordReservation.setText("");
}}

TextArea descriptif = new TextArea(5,20);
Button boutonReservation = new Button("RESERVATION");
class ActionReservation implements ActionListener {
  public synchronized void actionPerformed(ActionEvent ev) {
    if (chambreCourante == null) {return;}
    try {
     boolean ok = chambreCourante.tentativeReservation
                                    (identClient.getText());
     if (ok) {accordReservation.setText("accorde");}
     else {accordReservation.setText("refuse");}
   } catch(Exception e){};
}}
```

Les clients (suite)

```
TextField accordReservation = new TextField(10);

InterfaceServeurReservation serveurReservation;

InterfaceChambre chambreCourante;

public void init(){
  Placement.p(this,...);
     ...
  Placement.p(this,...);

  try {
    serveurReservation = (InterfaceServeurReservation)
      Naming.lookup("//florence.irisa.fr:8090/leServeur");
  } catch(Exception e){System.out.println ("erreur");}

  boutonEnvoiNumChambre.addActionListener
                        (new ActionEnvoiNumChambre());
  boutonReservation.addActionListener
                        (new ActionReservation());
}
}
```