PRÉSENTATION DU LANGAGE

Historique du Langage

1987 James Gosling travaille sur NeWS (un concurrent du Système X Window).

$\longrightarrow$ Echec de News

1990 Bill Joy dépose les premières bases du Langage Java ("*le réseau est l'ordinateur*"
Création de *Sun Aspen Smallworks*

1992 James Gosling rejoint Bill Joy à Aspen.

* Création de First Person, filiale de Sun.
* Développement de *C++ minus minus*.
* Développement de `Oak`
  - petit,
  - robuste,
  - indépendant de l'architecture,
  - orienté objet.

1993 Explosion de l'Internet et du Web.

* Oak devint Java

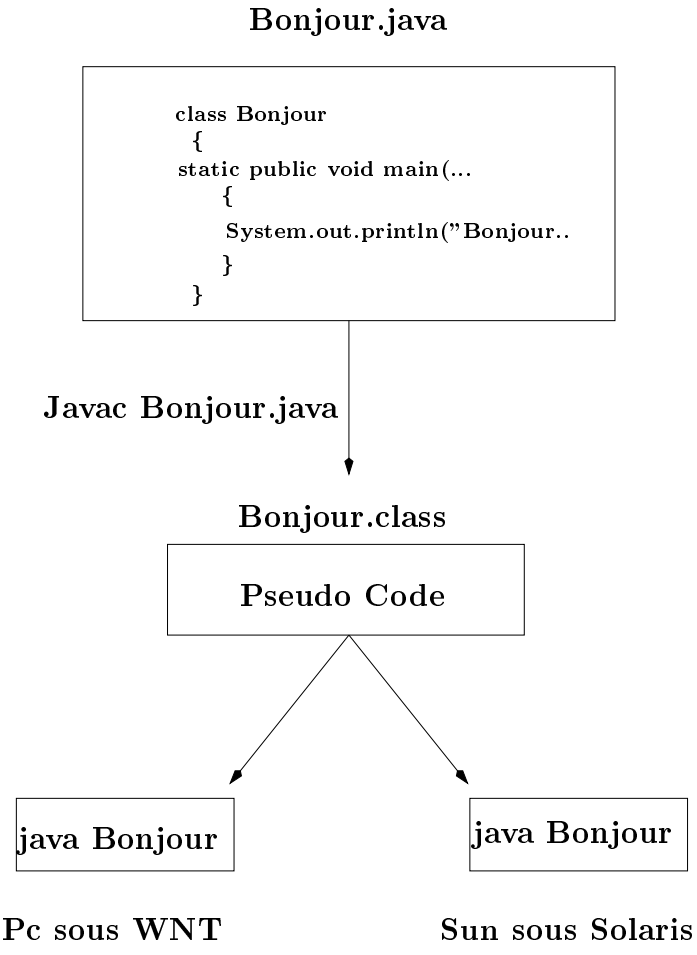
Caractéristiques de Java

- **Java est un langage orienté objet**
  - Objets et classes;
  - Héritage;
  - Hiérarchie de classe;
    - Une seule racine `class Object`
    - Chaque classe est représentée par un objet `class Class`
- **Java est fortement typé**
- **Java intègre un ramasse-miettes**
- **Java est multi-processus**
- **Java permet le chargement dynamique**
- **Java est parfaitement adapté a Internet**
  - Robustesse;
  - Sécuritaire;
  - Neutre et portable;
- **Java possède une syntaxe simple**

Environnement de développement

- Un compilateur `javac`
  A partir d'un fichier en langage Java, le compilateur génère un fichier interprétable par la machine virtuelle Java.
- Un Interpréteur `java`
  L'interpréteur a en charge d'exécuter le code sur la machine virtuelle Java
- Un générateur de documentation `javadoc`
  A partir du code source, il est possible de générer une documentation au format HTML
- Un intégrateur de code C `javah`
  Il est possible d'utiliser en Java des portions de code écrites en Langage C.
- Un désassembleur `javap`
- Un débogueur de code `jdb`
- Un visualisateur d'Applet `appletviewer`
  La possibilité d'inclure du code Java dans des fichiers HTML.

Compilation et Interprétation



Le premier programme Java

Le fichier `Bonjour.java`

```
public class Bonjour
{
  public static void main(String [] argv)
    {
      System.out.println(" Bonjour");
    }
}
```

La Compilation

Le fichier `Bonjour.java` est compilé en utilisant la commande

javac Bonjour.java

Un fichier `Bonjour.class` résulte de cette compilation.

L'exécution Le fichier `Bonjour.class` est exécuter en utilisant la commande

java Bonjour

Le mot Bonjour s'affiche sur le terminal.

Un apperçu du débogueur

```
>>: jdb Bonjour
Initializing jdb...
0x40776030:class(Bonjour)

> stop in Bonjour.main
Breakpoint set in Bonjour.main

> run
run Bonjour
running ...
main[1]
Breakpoint hit: Bonjour.main (Bonjour:5)

main[1] list
1          public class Bonjour
2          {
3            public static void main(String [] argv)
4              {
5       =>       System.out.println(" Bonjour");
6              }
7          }

main[1] where
  [1] Bonjour.main (Bonjour:5)
  [2] sun.tools.debug.MainThread.run (MainThread:55)

main[1] print System.out
System.out = java.io.PrintStream@80ee838
main[1] cont
 Bonjour
Bonjour exited
```

Java et HTML

On peut intégrer dans des pages Web des applications interactives écrites en Java. Le terme *Applet* désigne une portion de code Java appelable depuis un "browser HTML" ou depuis appletviewer.

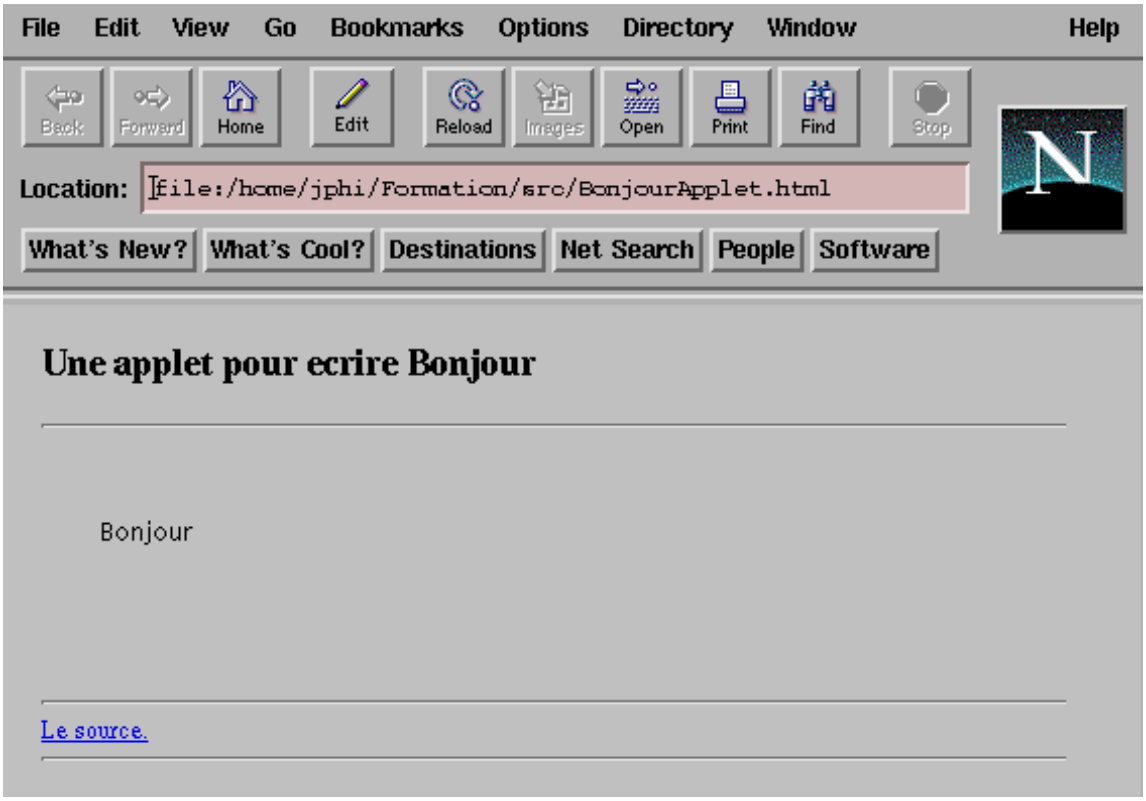
Le fichier BonjourApplet.java

```
import java.applet.*;
import java.awt.*;

public class BonjourApplet extends Applet
{
  public void paint(Graphics g)
    {
      g.drawString(" Bonjour", 25, 50);
    }
}
```

Le fichier BonjourApplet.html

```
<title> Applet Bonjour </title>
<h1> Une applet pour ecrire Bonjour </h1>
<hr>
<applet code=BonjourApplet.class width=120 height=120>
</applet>
<hr>
<a href="BonjourApplet.java"> Le source.</a>
<hr>
```

L'applet visualisée avec `netscape BonjourApplet.html`



L'applet visualisée avec `appletviewver BonjourApplet.html`

SYNTAXE DE BASE DU LANGAGE

Les mots clefs du langage Java

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronised
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto	private	transient
const	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

Les identificateurs en Java

Les identificateurs sont utilisés pour nommer des:

- Variables;
- Étiquette;
- Classes;
- Méthodes (au sens large);
- Packages;

Un identificateur commence soit :

- par une lettre majuscule ou minuscule
- par un $ ou un _

Le reste de l'identificateur doit être:

- une lettre;
- un chiffre;
- les symboles $ ou un _;
- un caractère unicode supérieur à 192;

Un identificateur ne peut être un mot clef du langage.

Exemple d'identificateur

_variable;

$System;

ValeurIndent_9;

Allouée;

Convention de nommage Java

Les identificateurs des classes commencent par une majuscule; Les autres identificateurs commencent par une minuscule; La coupure est matérialisée par la première lettre en majuscule;

Exemple

définitionDeMethode;

NouvelleClasse;

uneVariableEstDéfinie;

Véhicule;

VéhiculeQuatreRoues;

avancerVéhicule;

Les types de bases en Java

Types	Contenu	Valeur par défaut	taille
`boolean`	{ true, false }	false	1 bit
`char`	Caractère Unicode	\ u0000	16 bits
`byte`	entiers signés	0	8 bits
`short`	entiers signés	0	16 bits
`int`	entiers signés	0	32 bits
`long`	entiers signés	0	64 bits
`float`	flottant	0.0	32 bits
`double`	flottant	0.0	64 bits

Les types de bases ne sont pas des classes au sens de Java.

Les littéraux en Java

Les booléens (**boolean**)

```
boolean flag = true;
flag = false;
```

Les entiers (**byte, short, int, long**)

On peut les noter en :

- décimal;
- octal, en commençant par un zéro;
- hexadécimal, en commençant par `0x` ou `0X`;
- distinction entre `int` et `long` s'effectue en terminant l'entier par un `L` ou un `l`

```
int  i = 12;  // decimal
int  i = 014; // octal
int  i = 0x0c; // hexadecimal
long i = 12L; // de type long
byte i = 100000; //Incompatible type for declaration
```

Les réels (**float, double**) On peut les noter en :

- notation décimale;
- notation scientifique;
- distinction entre `float` et `double` s'effectue en terminant le réel soit par un `F` soit par `D`

```
double  f = 0.12; // decimal
double  f = .12; // decimal
double  f = 0.012e+1; //scientifique
double  f = 1.2E-1;  //scientifique
float   f = 0.12F; // de type float
double  f = .12D ; // de type double
```

Les caractères(**char**)

On peut les noter soit:

- entre apostrophe '
- en notation Unicode \uxxxx
- en octal
- en hexadécimal

```
char c = 0377;
char c =  '\'';
char c = 0xff;
char c = '\u00ff';
```

Les chaînes de caractères (**String**)

Il existe dans Java une classe `String` implémentant les chaînes de caractères. Bien qu'étant une classe, le type `String` en Java, est considéré comme un type *primitif* par le compilateur.

1. Le contenu d'une `String` n'est pas modifiable. Utiliser alors le type `StringBuffer`.
2. Il n'existe pas de symbole de fin de chaîne.
3. `length()`, `charAt()`, `equals()`, `compareTo()`

Le littéral chaîne de caractères existe en Java.

Une chaîne de caractères se note entre guillemets;

```
String s = "\"";
String s = "Il fait beau \n cette ete";
String s = "\u00ff";
```

La déclaration des variables

La déclaration d'une variable consiste à associer un type avec un nom d'identificateur. Il est possible d'initialiser une variable lors de sa déclaration. De plus on peut définir des constantes en utilisant le mot clef `final`. Dans ce cas, la variable ne peut être modifiée après sa déclaration.

```
int i;
long j = 3.4L;
final double pi = 3.14159;
```

Les tableaux en Java

On peut définir en Java des tableaux mono ou multidimensionnel.

```
int [] tab;
int tab [];
char tab [][];
```

A la différence de C, on ne fait que définir une variable. La taille du tableau ne fait pas partie de la définition, elle doit être explicitement allouée. Comme en C, les tableaux commencent à l'indice 0.

On peut obtenir la taille d'un tableau en utilisant la syntaxe suivante

```
tab.length // la taille du tableau.
```

```
int [] tab = new int[10];
// tab.length  est egale a 10
int tab [] = {0,1,2,3,4};
// tab.length  est egale a 5
char tab [][]; = {{'a','b','c'}, {'d','e','f','g','h'}}
// tab.length est egale a 2
// tab[0].lenght = 3
// tab[1].lenght = 5
char [][]tab = new tab[8][8];
long [][]tab = new tab[5][];
```

Les opérateurs en Java

Les opérateurs du langage Java sont pratiquement les mêmes que les opérateurs du langage C. Les différences sont l'opérateur `instanceof`(seulement en Java), l'opérateur `sizeof` (seulement en C), les pointeurs n'existent pas dans la syntaxe du langage Java. Comme en C les opérateurs du langage Java sont typés.

Priorité	Opérateur	Type Opérateur	Description
1	++, –	Arithmétique	Incrémentation et décrémentation
1	+, -	Arithmétique	Plus et moins unaire
1	~	Numérique	Complément à 1
1	!	Boolean	Complément logique
1	(*type*)	Tous	Conversion
2	*, /	Arithmétique	Multiplication, division
2	%	Arithmétique	Modulo
3	+, -	Arithmétique	Addition, soustraction
3	+	String	Concaténation de chaîne
4	<<	Numérique	Décalage à gauche
4	>>	Numérique	Décalage à droite (signe)
4	>>>	Numérique	Décalage à droite (sans signe)
5	<, <=, >, >=	Arithmétique	Comparaison
5	`instanceof`	Objet	Comparaison de type
6	== , !=	Primitive	Égalité et Inégalité par valeur
6	== , !=	Objet	Égalité et Inégalité par référence
7	&	Numérique	ET binaire
7	&	Boolean	ET booléen
8	^	Numérique	OU exclusif binaire
8	^	Boolean	OU exclusif booléen
9	|	Numérique	OU binaire
9	|	Boolean	OU booléen
10	&&	Boolean	ET conditionnel
11	||	Boolean	OU conditionnel
12	?:	N/A	Comparaison ternaire
13	=	Tous	Affectation
13	*=, /=, %=, +=, -=, —=, &=, >>=, <<=, >>>=		

Les instructions en Java

Bloc de code. Comme en C les instructions ou les expressions se situent à l'intérieur d'un bloc de code. Un bloc de code est délimité par une accolade ouvrante { et une accolade fermante } .

```
{
   int i = 3;
   int j = 5;
   {
     int a = 2*3;
         .....
   }
   i = i + j -1;
}
```

La portée d'une variable est limitée au bloc ou elle est définie. Une simple variable ne peut être vue à l'extérieur du bloc ou elle est définie. Cette propriétés doit être modulée pour les variables de classes ou les variables d'instances.

```
{
   int i = 3;
   int j = 5;
   {
     int a = 2*3;
         .........
   }
   i = i + a;
   //Erreur la variable a n'est plus visible.
}
```

Une variable d'un bloc supérieur peut être masquée par la définition d'une variable de même nom dans un bloc inférieur.

```
{
   final double pi = 3.14159;
   int j = 0;
   {
     double j;
     j = pi;
   }
}
```

Les structures de contrôle

- Il n'existe pas en Java une instructions similaire à la liste d'instructions du langage C.
- Les instructions `if/then/else, while, do/while, switch` sont les mêmes qu'en Langage C.
- L'instruction `for` est la même qu'en langage C, mais il possible de cumuler plusieurs instructions dans la partie *initialisation* et dans la partie *incrémentation* bien que la liste d'instructions n'existe pas.

Instruction Conditionnelle

Les instructions conditionnelles sont :

1. `if then else`
2. `switch case`

Le `if then else` se décline sous trois formes:

```
// premiere forme
if (condition)
    instruction;

// seconde forme
if (condition)
    instruction1;
else
    instruction2;

// troisieme forme

if (condition1)
    instruction1;
else if(condition2)
    instruction2;
else if (condition3)
     instruction3;
else
     instruction4;
```

Un exemple en Java

```
if (tab[0] < tab[1])
   System.out.println("Tableau ordonne");

if (tab[0] < tab[1])
   System.out.println("Tableau ordonne");
else
   {
     int tmp = tab[1];tab[1]=tab[0];tab[0]=tmp;
```

```
    }

if (tab[0] < tab[1])
   System.out.println("Tableau ordonne");
else if (tab[1] < tab[2])
   {
     int tmp = tab[1];tab[1]=tab[0];tab[0]=tmp;
   }
else
   {
     int tmp = tab[1];tab[1]=tab[0];tab[0]=tmp;
     tmp = tab[2];tab[2]=tab[1];tab[1]=tmp;
   }
```

Le **switch** est un aiguillage prenant en entrée une expression assimilable à un entier. Il compare séquentiellement la valeur de l'expression aux constantes définies par les clauses **case**. Lorsqu'une correspondance est faîte, les instructions sont ensuite exécutées les une après les autres à moins d'un échappement . Si aucune clauses ne convient les instructions de la clause **default** sont exécutées si elle est présente.

```
switch(value)
{
   case const1:
        instruction1;
   case const2:
        instruction2;
   default:
        instruction3;
}
```

```
static public void main (String [] argv)
{
   int tmp = Integer.parseInt(argv[0]);
   final int constante2 = 2;

   switch(tmp)
     {
      case 1:
        System.out.println("La valeur est" +  " 1");
        break;
      case constante2:
        System.out.println("La valeur est" +  " 2");
        break;
      default:
        System.out.println("ni une ni deux");
        break;
      }
}
```

Dans ce cas, les instructions **break** permettent de n'exécuter que la clause sélectionnée.

Les instructions itératives

Les instructions itératives sont:

1. **while**
2. **do/while**
3. **for**

Le **while** se présente sous la forme:

```
while(condition)
    instruction1;
```

Tant que **condition** est vrai, le bloc **instruction1** est évalué. Si dés le départ **condition** est **false** jamais **instruction1** n'est évaluée.

```
File f = new File(".");
String [] files = f.list();
int i = 0;

while(i < files.length)
{
   System.out.println(files[i]);
   i++;
}
```

Cette portion de code écrit les fichiers contenus dans le répertoire courant.

Le **do/while** se présente sous la forme:

```
do
{
   instruction1;
}while(condition);
```

On commence par évaluer le bloc d'instruction **instruction1**, ensuite **condition** est évaluée. Si la valeur de condition est égale à **true** on recommence **instruction1** jusqu'à que **condition** soit égale à **false**.

```
File f = new File(".");
String [] files = f.list();
int i = 0;


do {
   System.out.println(files[i]);
   i++;
}while(i < files.length)
```

Dans ce cas, une erreur peut se produire si `f.list()` retourne `null`.

L'instruction `for` se présente sous la forme suivante:

```
for(initialisation;condition;evolution)
    instruction1;
```

Cette forme peut se réécrire en

```
initialisation;
while(condition)
{
  instruction1;
  evolution;
}
```

On commence par évaluer `initialisation`, puis on évalue `condition`. Si `condition` est égale à `true` on exécute `instruction1` et `evolution`, on continue jusqu'à ce que `condition` soit égale à `false`.

Un exemple avec la liste d'instruction.

```
class syn
```

```
{
  static public void test()
  {
    int k;
    for(int i =0, k = 0;
        i + k < 3;
        i+=2, k--)
      System.out.println("k" + k + " i " + i);
  }
}
```

Dans ce cas, la portée de `i, k` est réduite à celle du `for`.

```
File f = new File(".");
String [] files;
int i;
for( files = f.list(),  i = 0;
     i < files.length;
     System.out.println(files[i++]))
  ;
```

Un autre exemple avec les tableaux:

```
import java.io.*;

class test
{
  static public void main (String [] argv)
   {
      int [][] tab = new int[8][];
      for(int i = 0; i < tab.length; i++)
        {
          tab[i] = new int[i+1];
```

```
          for(int k = 0; k <= i; k++)
            tab[i][k] = i;
         }
      for(int i = 0; i <  tab.length; i++)
         {
          for(int k = 0; k < tab[i].length; k++)
            System.out.print(" " + tab[i][k]);
          System.out.println("");
         }
    }

}
```

```
 0
 1 1
 2 2 2
 3 3 3 3
 4 4 4 4 4
 5 5 5 5 5 5
 6 6 6 6 6 6 6
 7 7 7 7 7 7 7 7
```

Identification d'instructions

Il est possible en Java d'étiqueter une instruction ou un bloc d'instructions. Cette étiquette peut ensuite être utiliser pour préciser un point de reprise pour une instruction d'échappement (**break, continue**). L'étiquetage d'instruction se présente sous les formes:

```
etiquette:
       instruction;
```

ou

```
etiquette:
{
       instruction;
}

public static void main (String [] argv)
{
  int k = 0;
 boucleExterne:
   for(int i = 10; i > 2 * k; i++, k++)
     {
       k = 2 * k;
     boucleInterne:
       {
         int j = 0;
         while(j++ < i)
           {
             System.out.println("i " + i +
                                " j " + j +
                                "k " + k);
           }
       }
     }
}
```

Instruction d'échappement

Les instructions d'échappement sont:

- `return` donne le contrôle à la fonction appelante;
- `break` sort du bloc courant;

utiliser dans les instructions (`while, for,switch,do/while`).

- `continue` se repositionne avant l'évaluation de la condition; utiliser dans les instructions (`while, for, do/while`).
- `throws` tirer par les cheveux (voir les exceptions).

```
static public void main(String [] param)
 {
  for(int j = 1; j < 10; j++)
   {
     if( j % 2 == 0)
          continue;
      System.out.println(j);
   }
 }
```

```
static public void main(String [] param)
 {
  for(int j = 1; j < 10; j++)
   {
     if( j % 2 == 0)
          break;
      System.out.println(j);
   }
 }
```

Les instructions d'échappement peuvent reprendre à une étiquette;

```
static public void main(String [] param)
 {
  int k = 0;
  boucle:
  for(int i = 1; i < 10; i++)
```

```
    for(int j = 1; j < 10; j++)
     {
       if( j % 2 == 0)
          continue boucle;
       System.out.println(k++);
     }
  }

static public void main(String [] param)
 {
  int k = 0;
  boucle:
  for(int i = 1; i < 10; i++)
    for(int j = 1; j < 10; j++)
     {
       if( j % 2 == 0)
          break boucle;
       System.out.println(k++);
     }
  }
```

PROGRAMMATION ORIENTÉE OBJET

CONCEPTS DE BASES DE PPO

Définition de la PPO

Définition 1 *La POO est une méthode d'implémentation dans laquelle des programmes sont organisés comme des ensembles coopérants d'objets, chacun représentant une instance d'une certaine classe et toutes les classes sont membres d'une hiérarchie de classe unifiée par une relation d'héritage.*

$$\text{PPO} = \begin{cases} \text{Objets} \\ + \\ \text{Classe} \\ + \\ \text{Héritage} \end{cases}$$

La Classe

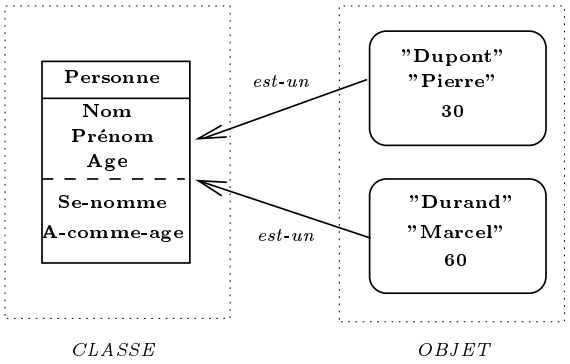
Classe: La classe représente les structures et les comportements communs à un ensemble d'objets. Elle permet de gérer une famille d'objets.

1. **Définition ensembliste**: Une classe est un ensemble d'objets ayant des caractéristiques et des comportements similaires.
2. **Définition conceptuelle**: La classe est une entité abstraite, appelé *type abstrait de données*. Ce type abstrait définit une structure (un type) et un ensemble de méthodes autorisées sur ce type. La classe est un concept, matérialisé par les *instances* représentant de la classe (objet et instance ont la même signification).

La classe a pour finalité de servir de “moule”. Elle représente une définition statique des objets.

L’instanciation

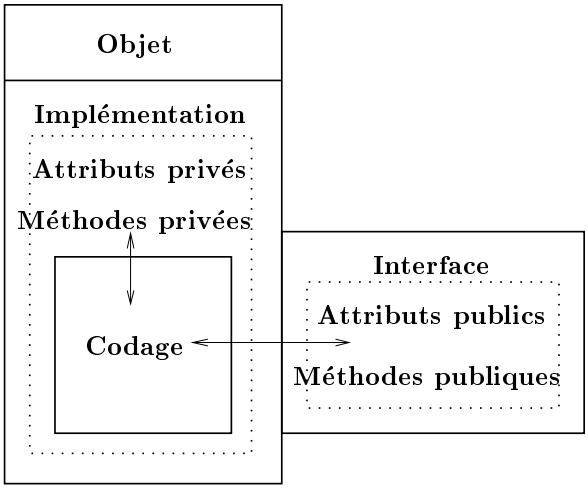
Instanciation: d’une classe consiste à créer une instance à partir de la description d’une classe. Le lien entre une classe et une de ses instances est un lien dénommé “*est-une-instance*”



L’encapsulation

Encapsulation: Capacité à masquer les informations internes à un objet (attributs et méthodes). L’encapsulation permet la séparation entre l’*interface* (qui est la manière dont l’extérieur perçoît un objet) et l’*implémentation* (qui représente la manière interne dont l’objet est codé). Seule l’interface doit être visible de l’extérieur.

Dans l’absolu, l’interface peut contenir aussi bien des attributs que des méthodes. Parfois, la présence d’attributs dans l’interface publique peut se révéler une erreur de conception.



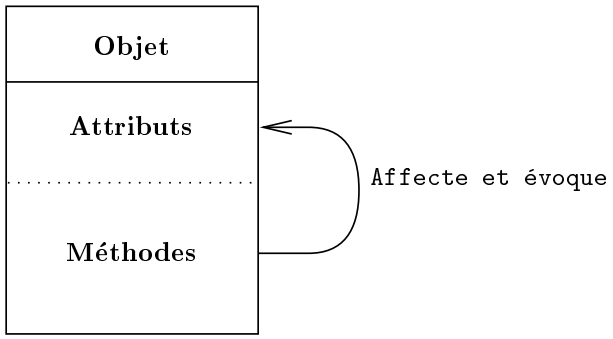
Objets

L'**objet** est l'entité logique de construction de l'architecture d'une application.

Un **objet** est une donnée composée d'un ensemble de **champs** organisés en:

- un regroupement de données encore appelées *attributs*, ou *variables d'instance*, et constituant la *structure* de l'objet;
- un regroupement de fonctions encore appelées *méthodes*, et constituant son *comportement*.

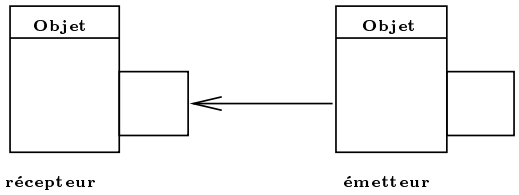
$$\text{OBJET} = \left\{ \begin{array}{c} \text{État (dynamique)} \\ + \\ \text{Comportement (statique)} \end{array} \right.$$



Communication par message

Envoie de message. L'envoi de message a pour conséquence l'activation dans l'objet récepteur d'une méthode déclarée dans son interface. Dans le cadre d'une encapsulation pure et dure, l'envoie de message assure l'intégrité de l'objet, car seules les méthodes de l'objet peuvent manipuler ses attributs.

- Transmission synchrone ou asynchrone des messages
- Traitement des messages de manière concurrentes
- Sélection dynamique de la méthode à activer



Héritage

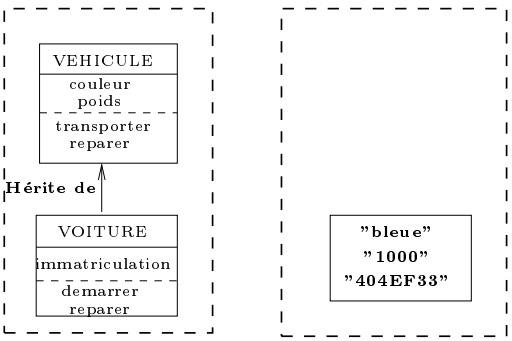
Héritage:Définit une *relation hiérarchique* entre une sous-classe (la classe qui hérite) et une surclasse (la classe dont elle hérite). La sous-classe peut:

- définir des attributs ou des méthodes supplémentaires

– redéfinir des attributs ou des méthodes hérités (principe de surcharge)

Toute instance d'une sous-classe possède par conséquent les propriétés déclarées au niveau de sa classe et les propriétés héritées de la surclasse.

Il faut retenir que la classe sous-classe est "*une sorte*" de surclasse. Tout ce qui caractérise la mère caractérise la fille.



Les avantages de l'héritage :

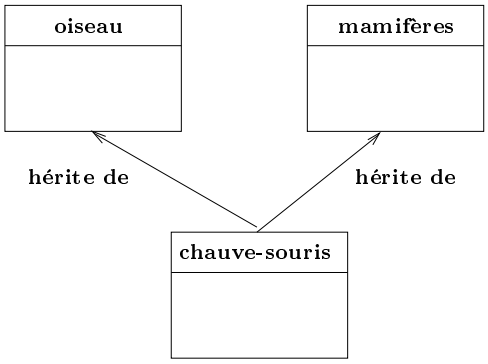
– Partage de ressources, il évite des duplications inutiles et diminue par conséquent la taille.
– Favorise la réutilisabilité, une nouvelle classe peut-être définie par spécialisation de classe existante.
– Permet le réordonnancement des définitions de classes.

Différent type d'héritage sont possibles:

– héritage simple ou héritage multiple

Héritage simple ou multiple

La hiérarchie induit un graphe sans cycle entre les classes. Dans le cas d'héritage simple, le graphe est en réalité un arbre. Dans le cas de l'héritage multiple, le graphe est alors un treillis.

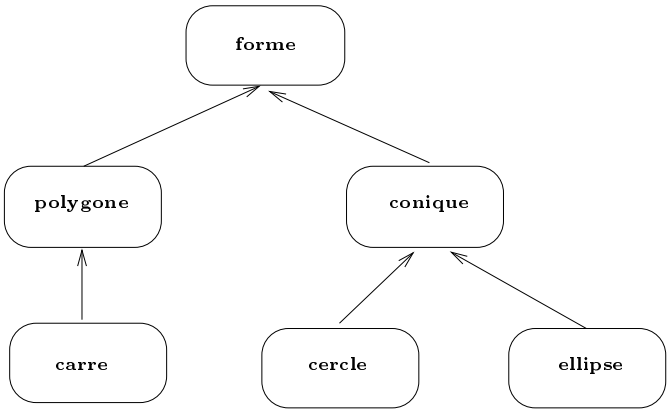


Lors de l'héritage multiple, des conflits peuvent se produire lorsque deux sur classes comportent des méthodes ou des attributs ayant le même nom.

Polymorphisme

Le polymorphisme est un concept dépendant de celui de l'héritage et de celui de la redéfinition. Il permet de manipuler de manière indistincte des objets appartenant à des classe différentes, toutes ces classes ayant une surclasse commune.

Du point de vue de l'implémentation, le polymorphisme est mis en oeuvre grâce à un mécanisme appelé liaison dynamique. Ce mécanisme permet de choisir lors de l'exécution (et non lors de la compilation) une parmi les différentes méthodes appelables. La résolution se fait en fonction du type réel de l'instance et non en fonction du type qui la référence.


forme
polygone
conique
carre
cercle
ellipse

LES CONCEPTS OBJETS EN JAVA

Le Lancement d'une application en Java

Il n'existe pas de contexte global en Java, il est obligatoire de créer systématiquement une classe, pour avoir un comportement.

Une classe `public` peut être défini dans un fichier java. Le nom de la classe `public` doit être le même que celui du fichier.

```
public class syn
{
 static public void main(String [] param)
  {
      System.out.printl("Hello World");
  }
}
```

Lorsqu'on exécute l'interpréteur java, `java file`, il vérifie dans la classe `file` qu'il existe une **méthode de classe** `static public void main(String [])`.

Il peut exister plusieurs méthodes `main` dans un ensemble de classe.

```
public class syn2
{
  static public void main(String [] param)
  {
    syn.main(param);
  }
}
```

Les classes en Java

```
public class <GENERALE>
{
        Variable de classes;
        Variables d'instances;

        Methodes de classes;

        Constructeur;
        Methodes;

}
```

1. Les *constructeurs* sont appelés lors de l'instanciation d'un objet. Ils servent à définir l'état initial d'un objet en attribuant une valeur aux variables d'instances. Un constructeur sert à structurer la mémoire attribuée par le ramasse-miette. Syntaxiquement un contructeur porte le même nom que la classe et il ne renvoie rien.

   ```
   class ClassTest
   {
       public ClassTest(){......};
       public ClassTest(String s){.....};
   }
   ```

2. *Variables d'instance*. Chaque objet possède ses propres variables d'instances, on ne peut y accéder qu'en utilisant un objet. La valeur des variables d'instances constitue l'état d'un objet. Syntaxiquement une variable d'instance, se déclare comme une variable appartenant au bloc de définition de la classe.

   ```
   class Point
   {
      int abscisse = 0;
      int ordonne  = 0;
      public Point();
   }
   ```

3. Les *methodes* permettent d'interagir avec un objet. Elles représentent son comportement. La sélection d'une méthode se fait obligatoirement en utilisant un objet. La méthode s'exécute toujours en utilisant l'objet récepteur du message. Si il n'existe pas d'ambigüité, on peut directement utiliser le nom d'une variable d'instance. Sinon il faut expliciter l'objet récepteur (identifié par le mot clef `this`). Syntaxiquement une méthode correspond à la définition d'une fonction à l'intérieur du bloc de définition de la classe.

```
class Point
{
   int abscisse = 0;
   int ordonne  = 0;

   void move(int abscisse,int ordonne)
   {
     this.abscisse = abscisse;
     this.ordonne  = ordonne;
   }

   void moveX(int x)
   {
     abscisse = x;
   }
}
```

4. *Variable de classes*. Il n'existe qu'une seule occurence de ces variables au sein de la classe. Toutes les instances de cette classe peuvent y accéder. On peut y accéder soit par le nom de la classe, soit par une instance de cette classe. Syntaxiquement la déclaration d'une variable de classe correspond à la déclaration

d'une variable d'instance précédée du mot clef `static`. On peut définir des constantes de classes en utilisant le mot clef `final` devant `static`.

```
class Point
{
   static int nombreInstance = 0;
   final static int origineX = 1;
   final static int origineY = 1;
}
```

5. Les *méthodes de classes*. Une méthode de classe est appelée en utilisant soit le nom de la classe, soit une instance de la classe. Contrairement au méthode, elles ne font référence à aucune instance particuliére. Il est donc impossible d'utiliser `this`. Syntaxiquement une méthode de classe correspond à la déclaration d'une méthode précédée du mot clef `static`.

```
class Point
{
   static void incrementNbInstance()
   {
      nombreInstance++;
   }
}
```

Un exemple

```
public class Test
{
  public static int nombreInstance = 0;

  int indice;

  public static int donneNombreInstance()
  {
```

```
    return nombreInstance;
  }
  public Test()
  {
    nombreInstance++;
    indice = donneNombreInstance();
  }
  public int donneIndice()
  {
    return this.indice; // equivalent return indice;
  }
}
```

Constructeur en Java (I)

Pour créer un nouvel objet en Java (instancier une classe) on utilise le mot clef réservé **new**. Cet appel demande au ramasse-miette de donner un nouvel espace mémoire au moins égal à la taille de l'objet. Ensuite cet espace mémoire est structuré en appelant un constructeur. Le constructeur est sélectionné en fonction des paramètres fournis.

```
class Point
{
   Point();
   Point(int x, int y);
   Point(Point);
}

Point p = new Point();
Point p = new Point(3,4);
Point p = new Point(q);
```

En java, si aucun construteur n'est défini, la langage fournit un constructeur sans argument. Si le programmeur d'une classe définit un constructeur avec arguments sans redéfinir le constructeur sans arguments, ce dernier n'est plus accessible.

```
class Essai
{
  private int vi;
  public Essai(int i)
  {
    vi = i;
  }
}
public class Ex1
{
  static public void main(String [] param)
  {
    Essai e = new Essai();
    // Ex1.java:16: No constructor
    // matching Essai() found in class Essai.
    // Essai e = new Essai();
  }
}
```

Constructeur en Java (II)

Pour éviter la duplication de code en java. On peut à l'intérieur d'un constructeur appelé un autre constructeur de la classe. Cela se fait à l'aide de `this(..,..)`. On doit impérativement commencer le code du constructeur avec cette première ligne.

```
class Essai
{
  private int vi;
  public Essai(int i)
  {
    vi = i;
  }

  public Essai()
  {
    this(1);
  }
}
public class Ex1
{
  static public void main(String [] param)
  {
    Essai e = new Essai();
  }
}
```

Destructeur en Java

Il n'existe pas à proprement parler de destructeur en Java. L'objet sera libéré par le ramasse-miettes. Il est possible de définir dans une classe une méthode `finalize()`. Cette méthode sera évoquée une seule fois, lors de la libération de l'objet par le ramasse-iettes.

La référence en Java

Il n'existe pas de pointeurs en Java. A l'exception des types primitifs tout est implémenté en Java avec des pointeurs.

Quand on écrit `Test t;`, on ne crée pas un objet `t` instance de `Test` mais simplement on déclare une variable `t` permettant de référencé un objet de type `Test`.

```
public class Point
{
  int abs;
  int ord;

  public Point(int x, int y)
  {
    abs = x;
    ord = y;
  }

  public void move(int x, int y);
  {
    abs = x;
    ord = y;
  }

  public void affiche()
  {
    System.out.print(" Abs : ");
    System.out.println(this.abs);
    System.out.print(" Ord : ");
    System.out.println(this.ord);
  }
}
```

Une utilisation de la Classe `Point`

```
public class Main
{
  static public void main(String [] Param)
    {
      Point p1;
      Point p2;

      p1 = new Point(0,1);
      p2 = p1;

      p2.move(10,10);
      p1.affiche();
      p2.affiche();
    }
}
```

Une autre utilisation de la Classe `Point`

```
public class Main
{
  static void move(Point p)
  {
    p.move(10,10);
  }

  static public void main(String [] Param)
    {
      Point p1;
      Point p2;

      p1 = new Point(0,1);
      p2 = p1;

      move(p2);
      p1.affiche();
      p2.affiche();
    }
}
```

Le passage de paramêtres se fait:

- Par **Valeur** pour les types de bases;
- Par **Référence** pour les autres types;

Les tableaux en Java (Retour)

Lorsqu'on déclare un tableau en Java, on n'instancie pas les élements du tableaux, on déclare seulement un ensemble contigü de références.

```
public class Main
{
   static public void main(String [] Param)
    {
      Point [] p1 = new Point[10];

        for(int i=0,j = 0; i < p1.length; i++, j++)
          p1[i] = new Point(i,j);


      for(int i = 0; i < p1.length; i++)
          p1[i].affiche();
    }
}
```

La surcharge en Java

La **surcharge** est la possibilité au sein d'un même contexte d'utiliser le même nom pour définir des fonctions ou des méthodes. La surcharge est autorisée si la signature des fonctions diffère soit par le nombre de paramètres soit par le type de paramètre à une position donnée. Le type de retour de la fonction n'est pas pris en compte dans la distinction de la surcharge.

```
class Point
{
   void move()
    {
      abscisse = ordonne = 0;
    }

   void move(int x, int y)
   {
    abscisse = x;
    ordonne = y;
   }

   void move(Point p)
   {
    abscisse = p.abscisse;
    ordonne  = p.ordonne;
   }
}
```

Dans un premier temps, le compilateur détermine quelles méthodes ont une signature compatible avec les paramêtres de l'appel. Si plusieurs méthodes sont canditates, la méthode qui se rapproche le plus est sélectionnée. Si plusieurs méthodes sont à égales distance, le compilateur génère un message d'erreurs.

Généralités sur l'héritage (I)

L'héritage définit une hiérarchie entre classes. La classe héritière est appelée la **sous-classe**, et la classe dont elle hérite est appelée la sur-classe.

- *Héritage simple*: Une sous-classe hérite directement d'une seule sur classe.
- *Héritage multiple*: Une sous-classe hérite directement d'une ou plusieurs classes.

Dans le cas d'héritage simple, la hiérarchie est représenté par un arbre. Dans le cas d'héritage multiple, la hiérarchie est représentée par un graphe dirigé sans cycle. L'héritage comporte deux propriétés:

1. Il est toujours possible de convertir une sous classe dans le type de la sous classe. Une sous classe contient nécessairement toutes les caractéristique de sa surclasse. Les parties `private` n'étant pas accessible.
2. L'interface publique de la sous-classe contient l'interface `public` de la sur classe.

Tout ce qui caractérise une surclasse caractérise une sous-classe. La sous classe ayant la possibilité de modifier ou d'étendre les caractéristiques de sa surclasse.

Héritage en Java (I)

On parle d'*héritage d'implémentation* lorsque une sous classe hérite du comportement de sa surclasse ou d'une partie de ces champs. C'est à dire que l'implémentation d'une sousclasse utilise l'implémentation de sa mère.

On parle d'*héritage d'interface* dans le cas contraire.

En java, il existe un **héritage simple d'implémentation** et un **héritage multiple d'interface**.

L'héritage simple d'implémentation en Java s'exprime en utilisant la primitive *extends* lors de la déclaration par exemple *class Fille extends Mere*. Dans ce cas, il ne peut y avoir qu'une seule surclasse à la suite de la clause `extends`.

L'héritage multiple d'interface s'exprime à l'aide de la clause *implements* . Dans ce cas, on peut écrire *class Fille implements mere1, mere2 extends mere 3*. La classe Fille, doit spécialiser les méthodes des classes `mere1, mere2` et peut utiliser les méthodes de la classe `mere3`.

Héritage en Java (II)

Cet exemple illustre la première propriété de l'héritage. On peut convertir le résultat de `new Fille` en une instance de la classe `Mere`. On peut aussi remarquer que lors de la construction d'une instance de la classe `Fille` le constructeur de la classe `Mere` est appelé.

```
class Fille extends Mere
{
  public Fille()
    {
      System.out.println(" Constructeur de la fille ");
    }
}

class Mere
{
  public Mere()
    {
      System.out.println(" Constructeur de la Mere ");
    }
}

public class Herit2
{
 static public void main(String [] argv)
   {
     Mere m = new Fille();
   }
}

//  Constructeur de la Mere
// Constructeur de la fille
```

Héritage en Java (III)

Cette exemple est une illustration de la deuxième propriété de l'héritage. On retrouve dans la classe **`fille`** toutes les possibilités publiques définies dans le mère.

```
class Fille extends Mere
{}

class Mere
{
  public int variable;

  public void m()
    {
      System.out.println("valeur de la variable " +
                         String.valueOf(variable));
    }

}

public class Herit1
{
  static public void main(String [] argv)
    {
      Fille f = new Fille();

      f.variable = 2;
      f.m();
    }
}
// valeur de la variable 2
```

Constructeur et Héritage en Java

On a vu précédement que pour construire une sousclasse, le constructeur de la surclasse est évoqué. Par défaut, la surclasse est construite en utilisant le constructeur sans argument. Cela pose un problème si la sur classe masque ce constructeur.

```
class Mere
{
  private int vi;

  public Mere(int i)
  {
    vi = i;
  }
}
class Fille extends Mere
{
  public Fille()
  {}
}
public class Ex1
{
  static public void main(String [] param)
  {
    Fille f = new Fille();
    // No constructor matching
    // Mere() found in class Mere
  }
}
```

Constructeur et Héritage en Java

En suivant un mécanisme similaire à celui de l'appel utilisant `this(...,..)`, on peut depuis le constructeur d'une sousclasse évoquer un constructeur d'une surclasse. On utilise alors l'instruction `super(....)`. Cette instruction doit être la première ligne du constructeur.

```
class Mere
{
  private int vi;

  public Mere(int i)
  {
    vi = i;
  }
}
class Fille extends Mere
{
  private String name;

  public Fille (String s)
  {
    super(2);
    name = s;
  }
  public Fille()
  {
    this("Sans nom");
  }
}
```

Liaison dynamique en Java(I)

La **redéfinition** d'une méthode est la possibilité, au sein d'une sousclasse, de spécialiser l'implémentation d'une méthode de la surclasse.

```
class Mere
{
 void uneMethode()
  {
    System.out.println("uneMethode dans Mere");
  }
}
class Fille extends Mere
{

  void uneMethode()
  {
    System.out.println("uneMethode Redefinie dans fille");
  }
  void uneMethode(int i)
  {
    System.out.println("uneMethode Surcharge dans fille");
  }
}
```

Liaison dynamique en Java(II)

Lorsqu'une sousclasse spécialise une méthode de sa surclasse, deux stratégies de résolution de la liaison sont envisageables:

1. Soit on résout au moment de la compilation (statiquement) en considérant le type de la variable référençant l'objet.
2. Soit on résout au moment de la compilation (dynamiquement) en considérant le type réél de l'objet et non le type de la variable.

```
class Mere
{
 void uneMethode()
  {
    System.out.println("uneMethode dans Mere");
  }
}
class Fille extends Mere
{

  void uneMethode()
  {
    System.out.println("uneMethode Redefinie dans fille");
  }
}

class Ex1
{
  static public void main(String [] param)
  {
    mere m = new Fille();
    m.uneMethode();
  }
  // liaison statique
  // uneMethode dans Mere
  // liaison dynamique
  // uneMethode Redefinie dans fille
}
```

En Java, il n'existe qu'un seul mécanisme de résolution, ce sont les liaisons dynamiques.

Polymorphisme

L'intéret des liaisons dynamiques est de permettre le **polymorphisme**. Le polymorsphisme utilise l'h'eritage pour unifier les sous-classes vers une surclasse de base. Et il utilise les liaisons dynamiques pour conserver à chaque instance son comportement initial.

```
class Forme
{
 void display()
  {
  }
}
class Carre extends Forme
{
  void display()
  {
    System.out.println("Un carre");
  }
}

class Cercle extends Forme
{
  void display()
  {
    System.out.println("Un cercle");
  }
}

class Triangle extends Forme
{
  void display()
  {
    System.out.println("Un triangle");
  }
}

class Ex1
{
  static public void main(String [] param)
  {
    Forme tabforme [] = { new Carre(),
                          new Cercle(),
                          new Triangle(),
                          new Cercle()
      };

    for(int i = 0; i < tabforme.length; i++)
      tabforme[i].display();

    // Un carre
    // Un cercle
    // Un triangle
    // Un cercle
  }
}
```

Méthode et classe finale

En java, le modificateur **`final`** permet d'indiquer

1. la présence d'une constante;

   ```
   final static float pi = 3.14159;
   ```

2. qu'une méthode n'a pas le droit dêtre redéfinie dans les sousclasses.

   ```
   final void methode()
   ```

3. qu'une classe ne peut avoir d'héritières.

   ```
   public final class SansFille
   {
   .........
   }
   ```

```
class mere
{
  final void methode()
    {
      System.out.println("Une methode finale");
    }
}

class fille extends mere
{
  void methode()
    {
      System.out.println("Il fallait essayer");
    }
//Final methods can't be overriden.
//Method void methode() is final in class mere.
}
```

Une utilisation possible concerne les problèmes de sécurité. Imaginons qu'il existe une classe `user` contenant la méthode `give_passwd()`, permettre la spécialisation de cette méthode peut s'avérer dangereux.

Algorithme Général de la sélection du comportement

La sélection procède en trois étapes:

1. *Sélection des fonctions compatibles.* La première étape consiste à sélectionner les méthodes compatibles. Il n'y a aucune distinction entre méthodes de classes et méthodes de classes. Si un objet est utilisé pour sélectionner le comportement la recherche est effectuée à partir du type qui le référence. Le comportement des surclasses peut être lui aussi pris en compte. Une méthode est compatible si:
   - Le nom de la méthode est le bon;
   - la méthode est accesssible dans le contexte;
   - Le nombre de paramêtre est correct;
   - Le type des paramètres est compatibles
2. *Résolution de la surcharge.* Si la première phase a élu plusieurs méthodes canditates. Le compilateur sélectionne celle qui convient le mieux. Si plusieurs méthodes sont aussi précises, une erreur est générée (ambigüité).
3. *Résolution dynamique.* Une fois la signature de la méthode sélectionnée, le mécanisme de liaisin dynamique est évoquée. Deux cas sont à distinguer:

   3.1. Le compilateur connaît la méthode:
   - On utilise un objet et sa référence est une classe finale.
   - On utilise on objet et la méthode est finale

– La méthode est static

3.2. Le compilateur génère le code de la sélection:

Surcharge et redéfinition

La surcharge ne tient pas compte du type réel des arguments pour sélectionner la méthode. Elle tient compte seulement du type identifiant l'instance.

```
class Mere
{
  void methode(Mere m)
  {
    System.out.println("Je suis la Mere");
  }
}

class Fille extends Mere
{
  void methode(Fille f)
  {
    System.out.println("Je suis la Fille");
  }
}

class Test
{
  static public void main(String [] argv)
  {
    Fille f = new Fille();
    Mere  m = f;

    f.methode(f); // Je suis la fille
    m.methode(m); // Je suis la Mere
  }
}
```

Un autre exemple

```
class A{}
class B extends A{}
class C extends B{}
class D extends C{}
class W
{
   void foo(D d){System.out.println("D");}
}
class X extends W
{
   void foo(A a){System.out.println("A");}
   void foo(B b){System.out.println("X.B");}

}
class Y extends X
{
   void foo(B b){System.out.println("Y.B");}
}
class Z extends Y
{
   void foo(C c){System.out.println("C");}
}
class Exemple
{
   public static void main(String [] argv)
   {
     Z z = new Z();
     ((X) z).foo(new C());
   }
}
```

Liaison dynamique et Méthode de classe

Les méthodes de classe sont héritées par les sous-classes. Mais elles ne sont pas liées dynamiquement.

```
class Mere
{
  static void fonction()
  {
    System.out.println("une mere");
  }

}

class Fille extends Mere
{
  static void fonction()
  {
    System.out.println("je suis dans la fille");
   }
}
class Test
{
  static public void main(String [] argv)
  {
    Mere m = new Fille();
    m.fonction();
  }
}
```

Les exceptions

Le but des exceptions et de permettre de traiter les dysfonctionnements d'un programme. Une exception peut être due à une erreur de programmation ou bien à une erreur d'utilisation. Par exemple, un utilisateur saisit un mauvais nom de fichier, il serait inacceptable que le programme s'arrête.

- Une instruction lève une exception;
- Soit
  - l'exception est prévue: on exécute le code associé et on continue le programme;
  - l'exception n'est pas prévue: on la communique à la fonction appelante. (On itère le processus);

Si aucun traitement d'exception n'est prévue le programme se termine par le traitement général de l'exception. Les exceptions sont des objets de Java, il est donc possible de les typer.

La syntaxe des Exceptions

```
try
{   Definition d'un bloc
    ou certaines instructions
    seront traites
}
catch(Type1 e)
{ traitement associe
  a l'exception de type1
}
catch(Type2 e)
{ traitement associe
  a l'exception de type2
}
finally
{
traitement commun
}
```

Lorsqu'une exception de type `T` est levée dans le bloc **`try`**, Java parcourt séquentiellement les clauses **`catch`**. La première exception compatible avec le type `T` est traitée. Si il existe une clause **`finally`** celle ci est exécutée.

Attention, comme seule la première clause compatible est traitée, l'ordre des clauses **`catch`** est important. Il faut aller du type le plus précis au type le plus général.

Un exemple de capture d'exception:

```
class ExceptionExample
{
  static public double divise(double x, double y)
   {
     try{
          return x/y;
        }
     catch(Exception e)
     {
           return 9E99D;
     }
    }

  static public void main(String [] S)
    {
      System.out.println(divise(1.,0.));
    }
}
```

Déclaration des exceptions

Java est très rigoureux, si une fonction utilise des instructions susceptibles de lever des exceptions alors soit

1. elles doivent être traitées par la fonction courante;
2. elles doivent être déclarées dans le prototype de la fonction;

La déclaration d'une exception dans le prototype d'une fonction est le suivant

```
type nomFonction throw except1, except2,....
{
  code
}
```

```
void lectureFichier(String name) throw IOException
```

Contrairement à C++, une méthode ou une fonction de classe qui ne donne aucune liste d'exceptions ne peut en lever aucune. De ce fait, le compilateur est capable de vérifier qu'une fonction ou méthode remplira correctement son contrat en ne levant que les exceptions déclarées dans sa signature.

```
public class Except1
{
  void f(){}
  void methode1()
    {
      except1 e;
      try
       {
         e = new except1();
       }
```

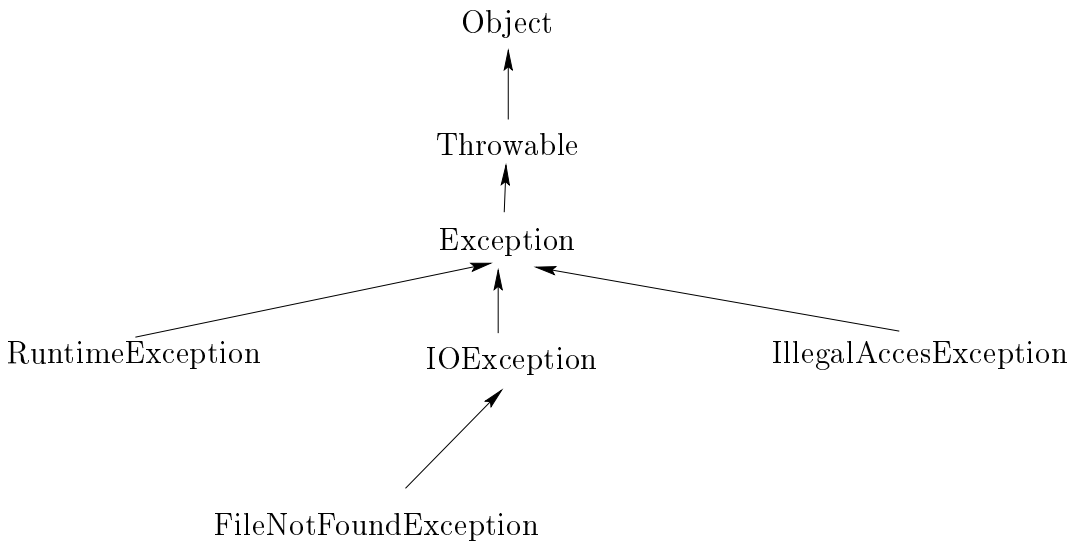
```
      catch(Exception exp){}
      e.f();
      // Variable e may not
      // have been initialized.
    }
}
```

La grande rigueur de java est ici illustrée. Bien que la variable `e` soit correctement créée, java déclenche une erreur car il ne peut être sur de l'initialisation. En effet, la variable `e` a été initialisée dans un bloc contenant un traitement d'exception, comme java ne sait pas si l'instruction `e = new except1();` a pu réellement être effectuée ou si une exception a créé un déroutement avant son exécution, il génère une erreur.

Les classes Exceptions

En langage Java, il existe une classe **`Throwable`** et une classe **`Exception`** qui hérite de **`Throwable`**. Pour qu'un nouveau type soit utilisable dans une clause **`throw`** il faut qu'il hérite au minimum de **`Throwable`**, il est préférable par souci des convenances de la faire héritée de **`Exception`**. La clause `catch(...)` de C++ peut dans ce cas être remplacée par une clause `catch(Exception e)`.

L'arbre d'héritage des Exceptions



Un exemple d'une nouvelle Exception

```
class MyException extends Exception
{
  MyException(String name, Object value)
    {
      super("My message" + name);
      ......
    }
}

class essai
{
  void methode1()
    {
      try
        {
          this.methode();
        }
      catch(MyException e){}
      catch(Exception e){}
    }
  void methode() throws MyException
    {
      throw new MyException("is in the class essai", this);
    }
}
```

Pour lever une exception, il est nécessaire de créer une instance de la classe représentant l'exception. Puis on utilise la clause **`throw`** pour lancer l'exception.

Héritage et exception

Du fait des liaisons dynamiques, la liste des exceptions d'une sous-classe doit être compatible avec la liste des exceptions de sur-classe. Elle peut être plus précise à condition que les exceptions de la sous-classe puissent être converties en des exceptions de la surclasse.

Package en Java

Un paquetage en Java, est un nom permettant de regrouper un certain nombre de classes ayant une sémantique commune. Physiquement, un paquetage Java est constitué de plusieurs ".java" qui produisent des fichiers ".class".

L'utilisation du paquetage `mypack` se fait par la déclaration `import mypack.*;` qui permet l'utilisation de toutes les classes définies dans ce paquetage.

L'appartenance des classe d'un fichier ".java" à un paquetage `mypack` se fait en utilisant la déclaration `package mypack;`

Un paquetage représente plus qu'une organisation du code, il crée certains accés privilégiés entre les classes appartenant à un même paquetage.

Par exemple, le fichier **entity1.java**

```
package mypack;

class entity1
{
}
```

Le fichier suivant **entity2.java**,

```
package mypack;

class entity2
{
   void methode1()
      {
        entity1 e;
      }
}
```

Bien que la classe **entity1** ne soit pas définie dans le même fichier que **entity2** et bien que la classe ne soit pas **public**. Il est possible d'évoquer des **entity1** à partir **entity2** car les 2 classes appartiennent au même paquetage.

Il n'existe pas de hiérarchie à l'interieur des paquetages en fonction de leur nom. Par exemple, **mypack.value** n'est pas un sous paquetage du paquetage **mypack**. Ces deux paquetages différents non aucune relation entre eux (si ce n'est une place sur la disque).

Lors de la compilation on peut attacher une classe à un paquetage en utlisant la syntaxe suivante:

```
javac -d mypack entity1.java
javac -d mypack entity2.java
```

Visibilité en Java

Chaque entité d'une classe peut avoir sa visibilité modifiée à l'aide des mots clefs `private, public, protected`. Si aucun de ces mots clefs n'est précisé, on dit que la visibilité est réduite au paquetage.

Situation	`public`	`protected`	`default`	private
mm̂e package	oui	oui	oui	non
package différent	oui	non, excepté les sous classes	non	non

Classes Virtuelles

Une classe virtuelle est une classe qui ne peut être instancier directement.

Deux types de classe virtuelle en Java:

- classe Abstraite;
- Interface;

Exemple de classe abstraite

```
abstract class Test
{
   abstract String entete();
   void affiche()
    {
      System.out.println("Je suis la classe" + entete();
    }
}

class Concrete extends Test
{
   String entete()
   {
    return "Concrete";
    }
}
```

Les classes abstraites sont des classes, elles ne peuvent être utilisées que pour l'héritage simple.

Les problèmes de l'héritage multiple.

L'héritage multiple en C++, pose de nombreux problèmes comme la sélection d'une méthode définie dans plusieurs classes soeurs, l'unicité des variables d'instances dans le cas des héritages en diamant. Ces problèmes sont résolus si les classes impliquées dans la partie supérieure du graphe d'héritage sont virtuelles pures et ne définissent aucun contexte. Dans ce cas, ces classes virtuelle pure ne servent qu'à définir des interfaces publiques. Les sous classes ayant en charge de donner une implémentation de cette interface si elle veulent être instanciable.

Interface en Java

Une classe virtuelle pure s'appelle en Java une *interface*. Dans la déclaration d'une classe, le mot clef `interface` remplace le mot clef `class`.

```
interface animal
{
  void deplace();
  void mange(int i);
}

interface mamifere extends animal
{
  void  allaite();
}

interface oiseau extends animal
{
  void vole();
}

class chauvesouris implements oiseau, mamifere
{
  public void deplace(){}
  public void mange(int i){}
  public void vole(){}
  public void  allaite(){}
}
```

Commentaires

Le type `animal` est représenter par une interface, il est impossible de créer des instances de `animal` en utilisant l'instruction `new animal()`. Les types `mamifere` et `oiseau` sont des extensions de l'interface `animal`, ils sont aussi représentés par des interfaces. Enfin la classe `chauvesouris` assure la concrétisation des interfaces `mamifere` et `oiseau` en définissant le code. Il s'agit dans ce cas, d'un héritage multiple, car les instances de `chauvesouris` peuvent être implicitement converties en instance de `mamifere` ou de `oiseau` (`mamifere = new chauvesouris()` ou `oiseau = new chauvesouris()`).

Composition des interfaces et des classes

```
interface A {}
interface B {}
interface C extends A,B {}
class D implements A{}
class E extends D, implements B{}
```

Une interface peut elle même héritée de plusieurs interfaces (cf. `interface C`). Il est possible à la fois d'hériter simplement de l'implementation d'une classe et aussi d'hériter de l'interface de plusieurs implementations (cf. `class E`).

Utilisation des interfaces

La grande utilisation des interfaces est de permettre aux applications d'être écrites en utilisant une interface. L'exemple le plus classique est l'écriture d'un environnement graphique. Tout le code de cette application est écrit en utilisant le type **`FormeGeometrique`** qui est une interface. L'extérieur à la possibilité d'étendre les fonctionnalités en créant diverses concrétisation de **`FormeGeometrique`** comme **`Carre, Cercle, ....`**

Une autre application possible peut être l'impression d'un texte.

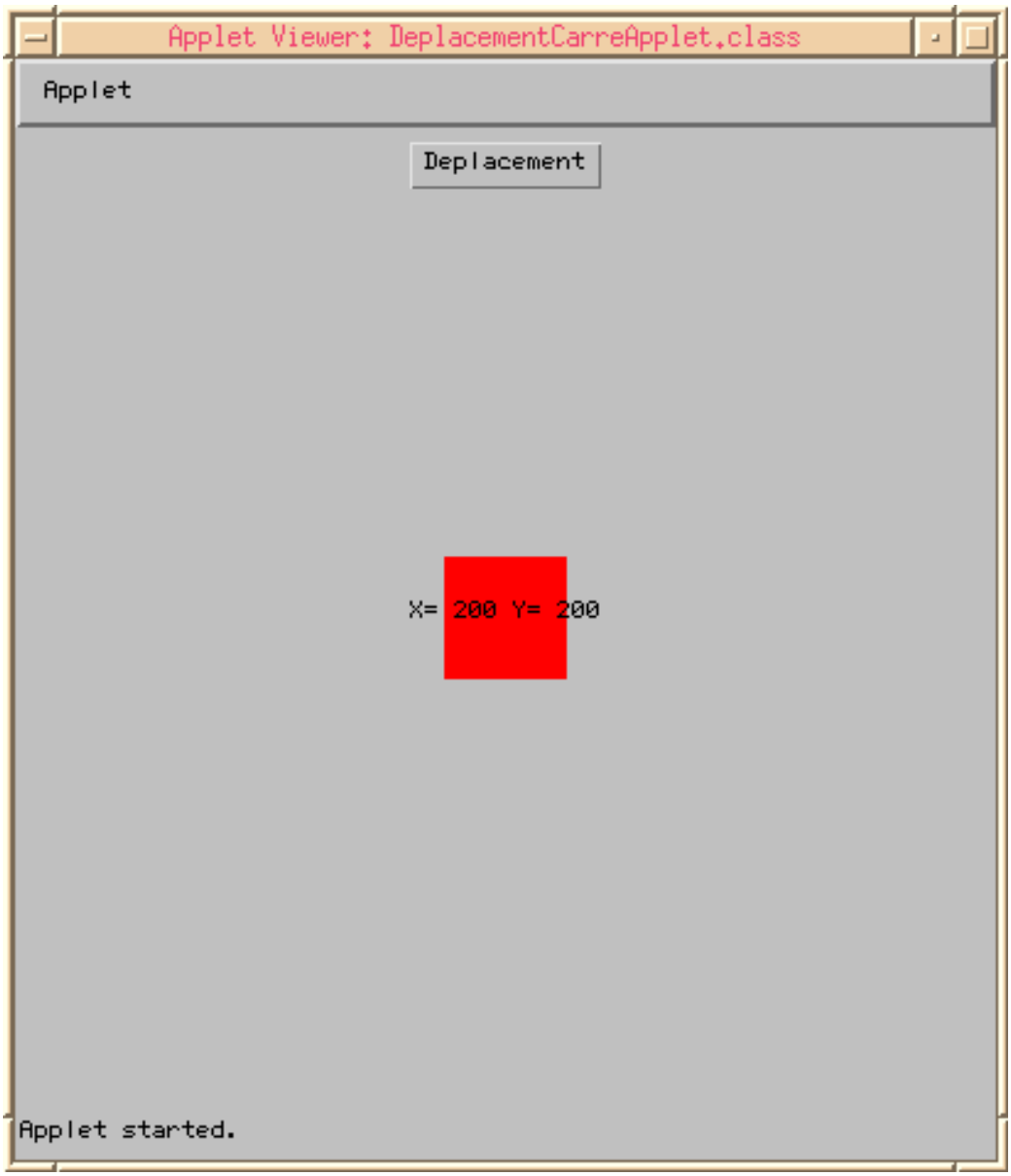
```
interface outputPeriph
{
  void writeString(String);
  void newline();
  void newpage();
}
class texte
{
   void write(outputPeriph P)
    {
       P.newpage();
       P.writeString();
    }
}
```

Sur cet exemple, si l'on concrétise l'interface **`outputPeriph`** en créant des imprimantes, des écrans, ... une instance de la classe **`texte`** pourra être affichée sur l'ensemble de ces périphériques sans aucune modification du code. Le polymorphisme peut être implémenté en Java, soit en utilisant des classes abstraites soit des classes concrètes.

UN EXEMPLE COMPLET

But de l'exercice

Ecrire une applet dans laquelle un carré se déplace aléatoirement lorsqu'on clique sur un bouton. De plus les coordonnées du centre du carré doivent être affichées.



La classe `Point`

```
public class Point
{
  int abscisse = 0;
  int ordonne = 0;

  public Point(){}
  public Point(int x, int y)
  {
    abscisse = x;
    ordonne = y;
  }
  public Point(Point p)
  {
    abscisse = p.abscisse;
    ordonne = p.ordonne;
  }
  public void move(int x, int y)
  {
    abscisse = x;
    ordonne = y;
  }
  public void move(Point p)
  {
    abscisse = p.abscisse;
    ordonne = p.ordonne;
  }
  public void translate(int x, int y)
  {
    abscisse += x;
    ordonne += y;
  }
  public void translate(Point p)
  {
    abscisse += p.abscisse;
    ordonne += p.ordonne;
  }
  public String toString()
  {
    return "X= " + abscisse + " Y= " + ordonne;
  }
  public double distance()
  {
    return Math.sqrt((abscisse * abscisse) +
                     (ordonne *ordonne));
  }
}
```

Sur cet exemple, les concepts mis en œvre sont:

- Déclaration d'une classe;
- Initialisation par défaut des variables;
- Surcharge
  - Constructeur;
  - méthodes `translate, move`
- Héritage implicite de la classe `Object` par la classe `Point`.
- Redéfinition de la méthode `toString` de `Object`

La classe **Carre**

```
import java.awt.*;
public class Carre
{
  Color  color = Color.red;
  Point centre = new Point();
  int   cote  = 50;

  Carre(){}
  Carre(Point ce, int cote, Color co)
  {
    color = co;
    this.cote = cote;
    centre = new Point(ce);
  }
  public void move(int x, int y)
  {
    centre.move(x,y);
  }
  public void move(Point p)
  {
    centre.move(p);
  }
  public void translate(int x, int y)
  {
    centre.translate(x,y);
  }
  public void translate(Point p)
  {
    centre.translate(p);
  }
  public void paint(Graphics g)
  {
    int leftCornerX = centre.abscisse - (cote / 2);
    int leftCornerY = centre.ordonne - (cote / 2);

    g.setColor(color);
    g.fillRect(leftCornerX, leftCornerY, cote, cote);

    g.setColor(Color.black);

    FontMetrics f = g.getFontMetrics();
    int      width = f.stringWidth(centre.toString());

    g.drawString(centre.toString(),
                 centre.abscisse - (width/2),
                 centre.ordonne);
  }
```

```
}
```

Les concepts mis en œuvre dans la classe carré sont :

- Déclaration d'un paquetage `import java.awt.*`
- Déclaration d'une classe;
- Initialisation par défaut des variables;
- Evocation de `this` pour éviter le masquage de `this.cote` par la variable locale `cote`.
- Surcharge
  - Constructeur;
  - méthodes `translate, move`
- Utilisation de l'objet `centre`
- Utilisation de la classe `Graphics` pour dessiner.

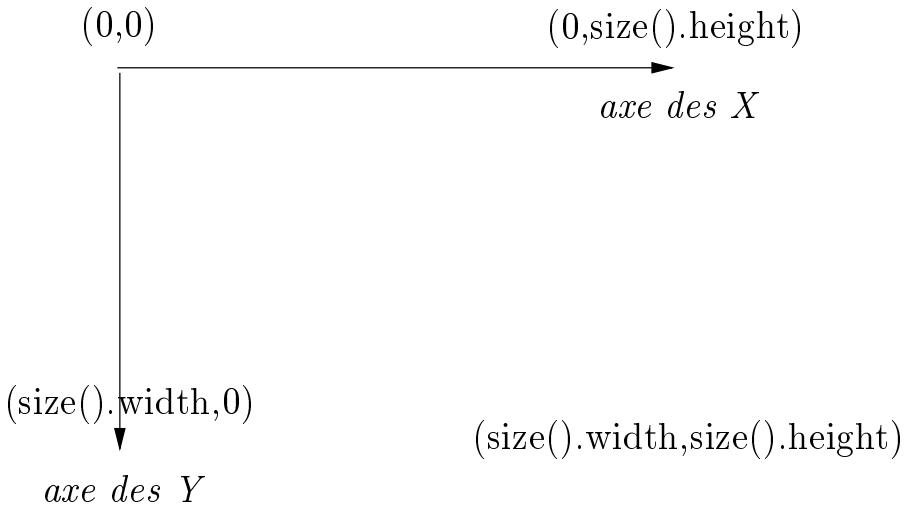
La classe Graphics

La classe `graphics` est utilisée pour dessiner dans une composante graphique. Elle permet de

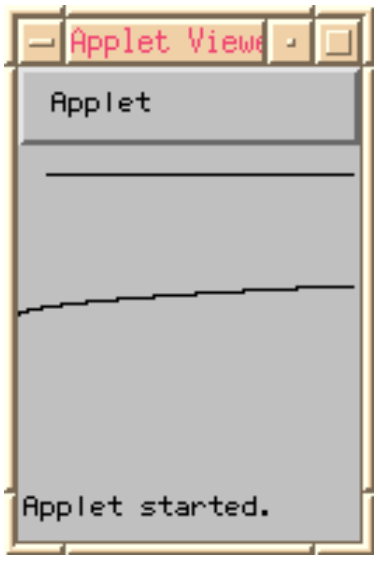
- De dessiner
  - des formes géométriques (Polygone, Ovale, Ligne,.....)
  - des chaînes de caractères.
  - des images;
- De changer la couleur du fond;
- De récupérer la police courante;
- d'identifier la composante;
- .....

Dessiner des formes géometriques

Chaque composante graphique possède son propre repère, on peut accéder à la tailler de la composante graphique en utilisant la méthode **`site()`** ou **`getSize()`** (jdk 1.1) qui retourne un objet de la classe **`Dimension`**.

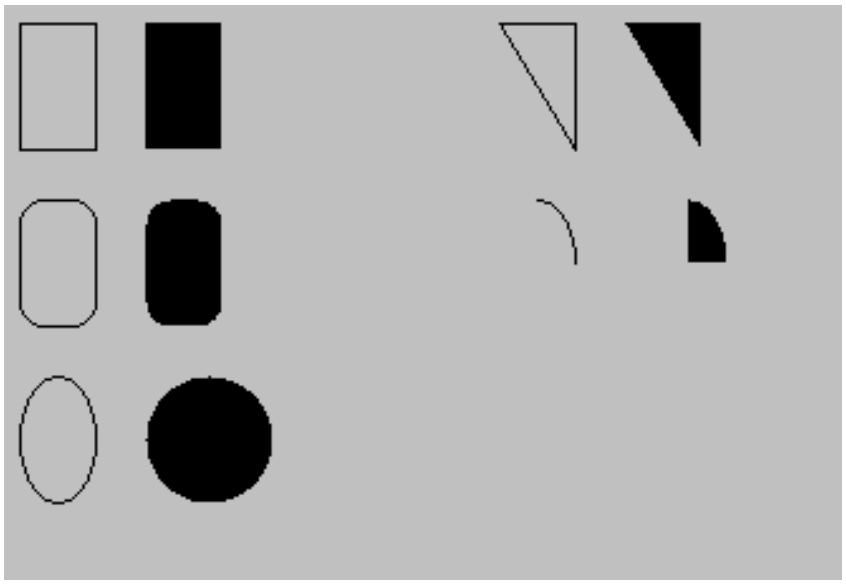


Utilisation de la primitive **`drawLine`**



```
public void paint(Graphics g)
 {
 g.drawLine(10,10, size().width, 10);

for(int i = 0; i < (size().width * size().width); i++)
{
  g.drawLine(i,
             (size().height) /2 - (int) Math.sqrt(i),
             i,
             (size().height /2)- (int) Math.sqrt(i+1));
}
```

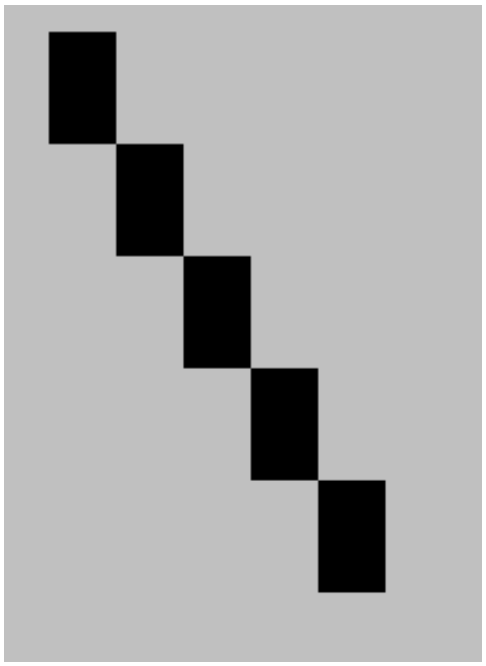
Utilisation des primitive de dessin



```
 public void paint(Graphics g)
 {
   g.drawRect(10,10, 30 ,50);
   g.fillRect(60,10, 30, 50);

   g.drawRoundRect(10,80, 30 ,50,20,20);
   g.fillRoundRect(60,80, 30, 50,20,20);

   g.drawOval(10,150, 30 ,50);
   g.fillOval(60,150, 50, 50);


   int [] coordX = { 200, 230, 230};
   int [] coordY = { 10, 10, 60};

   g.drawPolygon(coordX, coordY, coordX.length);
```

```
  int [] coordX1 = { 200 + 50, 230 + 50, 230 + 50};
  int [] coordY1 = { 10, 10, 60};

  g.fillPolygon(coordX1, coordY1, coordX1.length);
  g.drawArc(200,80, 30, 50, 0, 90);
  g.fillArc(260,80, 30,50, 0, 90);
}
```

Copier et Effacer



```
 g.fillRect(60,10, 30, 50);
 for(int i = 0; i < 5; i++)
   g.copyArea(60,10,30,50, 60 + 30*i, 10 + 50 *i);
  g.clearRect(60,10,30,50);
}
```

Ecrire du texte


```
g.setFont(new Font("Helvetica",Font.PLAIN,14));
g.drawString("Bonjour", 10 , 20);

g.setFont(new Font("Helvetica",Font.PLAIN,25));
g.drawString("Bonjour", 10 , 50);

g.setFont(new Font("Helvetica",Font.ITALIC,25));
g.drawString("Bonjour", 10 , 90);

g.setFont(new Font("TimesRoman",Font.ITALIC,25));
g.drawString("Bonjour", 10 , 130);
```

On peut obtenir des informations sur la nature de la police en utilisant les méthodes de la classe Font.

Mesure du texte

On peut à partir d'un objet de la classe **`Font`** instancier un objet de la classe **`FontMetrics`** afin d'obtenir des informations sur la taille de la police en pixel.

```
FontMetrics f = new FontMetrics(g.getFont());
int      width = f.stringWidth(centre.toString());
```

Utiliser la Couleur

La classe **`Color`** gère la couleur en java. Certaines couleurs sont prédéfinies par des variables de classe de **`Color`**. Par exemple

`Color.red` représente la couleur rouge. Les couleurs sont exprimées par un triplet `R,G,B`. Par exemple, `new Color(255,0,0)` représente la couleur rouge.

Une instance de la classe composante `Component` contient deux couleurs. La couleur du fond `background` et la couleur de dessin `foreground`. On peut changer la couleur du fond par `setBackground(Color)` , récuperer la couleur du fond par `getBackground()`.

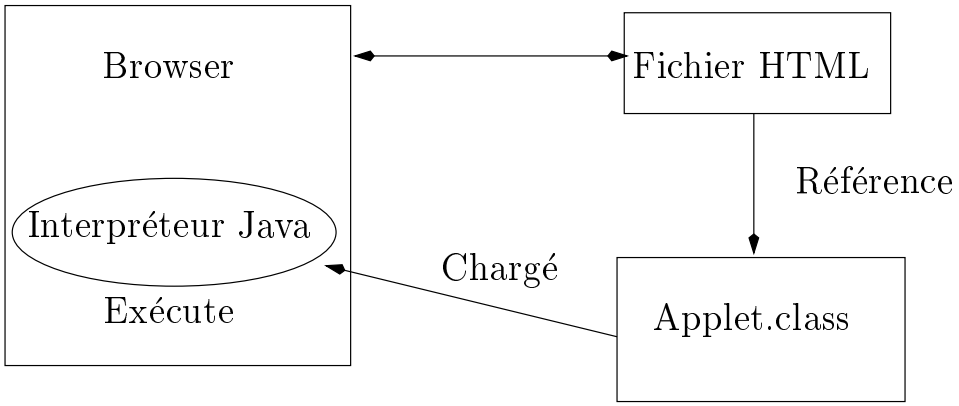
Au niveau de la classe `Graphics`

- on peut changer la couleur en utilisant la méthode `setColor(Color)`.
- on peut obtenir la couleur courante en utilisant la méthode `getColor()`


```
public void paint(Graphics g)
{

  g.setColor(new Color(10,10,10));

  g.setFont(new Font("Helvetica",Font.PLAIN,14));
  g.drawString("Bonjour", 10 , 20);

  g.setColor(new Color(0xff0000));
  g.setFont(new Font("Helvetica",Font.PLAIN,25));
  g.drawString("Bonjour", 10 , 50);

  g.setColor(new Color(0.0F,1.0F,0.0F));
```

```
  g.setFont(new Font("Helvetica",Font.ITALIC,25));
  g.drawString("Bonjour", 10 , 90);

  g.setColor(new Color(255,255,255));
  g.setFont(new Font("TimesRoman",Font.ITALIC,25));
  g.drawString("Bonjour", 10 , 130);
  }
}
```

Enfin La première Applet



Une applet est invoquée depuis un browser HTML en interprétant le tag `<APPLET>`. Le code de la machine est virtuelle est transférée sur la machine qui exécute le browser à travers le réseau. Une fois le code transféré, ce code est exécuter au moyen d'un interpréteur java (`hotjava, Netscape, ..`).

Un exemple de Fichier HTML

```
<title> Applet DeplacementCarre </title>
<h1> Une applet qui deplace un carre </h1>
<hr>
<applet code=DeplacementCarreApplet.class
        width=400 height=400>
</applet>
<hr>
    <a href="DeplacementCarreApplet.java"> Le source.</a>
<hr>
```

Contraintes Sécuritaires

Comme le code est exécuté sur la machine cliente, la sécurité est un élément prépondérant.

Une applet ne peut :

- Ne peut lancer de processus sur la machine cliente;
- Ne peut accéder au système de fichier local
  - lecture,écriture;
  - renommer, copier, effacer;
  - .....
- Ne peut ouvrir de canaux de communications avec des machines différentes du serveur. Avec Netscape aucune communication possible;
- Attendre de communications sur un port de la machine cliente.
- Charger du code dynamiquement;
- Modification ou création de propriétés;
- Quitter l'interpréteur java;

Echo Version applet

Dans un fichier `HTML` il est possible de définir des paramêtres consultable par l'applet.

Dans le fichier `HTML`

```
<PARAM NAME=..... VALUE=....>
```

Dans le code de l'applet

```
value  getParameter(String parameterName)
```

Un example:

```
<title> Applet ECHO </title>
<h1> On recopie la parametre </h1>
<hr>
<applet code=AppletEcho.class width=400 height=300>
<param  name="Parametre" value = "Bonjour le monde">
</applet>
<hr>
<a href="AppletEcho.java"> Le source.</a>
<hr>

public class AppletEcho extends Applet
{
  String valueParametre;

  public void init()
  {
        valueParametre = getParameter("Parametre");
  }

  public void paint(Graphics g)
    {
        g.drawString(valueParametre, 100,100);
    }

}
```

La classe `Applet`

```
Class java.applet.Applet

java.lang.Object
   |
   +----java.awt.Component ---> possibilites Graphique
            |
            +----java.awt.Container -----> Organisation des elements
                     |
                     +----java.awt.Panel
                              |
                              +----java.applet.Applet
```

Les Méthodes d'`Applet`

Les méthodes d'une `Applet` évoquées par le "browser".

- `init()` Cette méthode est appelée par le "browser" afin de d'informer l'applet de son chargement dans le système.
- `start()` Cette méthode est appelée par le "browser" pour informer l'applet qu'elle doit commencer son exécution. Elle est appelée après la méthode `init` ou à chaque fois qu'elle est revisitée par le "browser".
- `stop()` Cette méthode est appelée par le "browser", elle suspend l'exécution de l'applet. Elle peut être appelée juste avant la destruction de l'applet ou bien lorsque la page courante du "browser" change. On peut rependre le cours de l'exécution en invoquant la méthode `start()`.
- `destroy()` Cette méthode provoque la destruction de l'applet.

Une Applet est un `Component`

La méthode `paint()`

La méthode `paint()` est une méthode hérité de la classe `Component`, elle est utilisé pour dessiner à l'intérieur d'une instance de la classe `Component`. Cette méthode est évoquée à chaque fois que nécessaire par le gestionnaire d'écran (désiconification, recouvrement,....). Cette méthode est appelé lorsqu'un rafraichissement est demandé par l'extérieur.

La méthode `repaint()`

La méthode `repaint()` est une méthode qui à en charge la gestion du rafraichissement des composantes, elle appelle la méthode `update()` pour chacune des composantes. La méthode `update()` est responsable du rafraichissement de la composante. Par défault elle réaffiche le fond de l'image et ensuite appelle la méthode `paint()`

L' Applet `DeplacementCarreAppletComponent`

Le code de la classe `DeplacementCarreApplet`

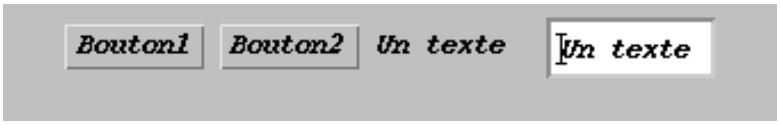
```
import java.awt.*;
import java.applet.*;

public class DeplacementCarreApplet extends Applet
{

  Carre carre;

  public void init()
  {
    carre = new Carre();
    carre.translate(200,200);
    add(new Button("Deplacement"));
  }

  public void paint(Graphics g)
  {
    carre.paint(g);
  }

  public boolean action(Event evt, Object arg)
  {
    int depx = (int) (Math.random()*5);
    int depy = (int) (Math.random()*5);

    depx = depx > 2 ? -depx: depx;
    depy = depy > 2 ? -depy: depy;

    carre.translate(depx, depy);
    repaint();
    return true;
  }
}
```

Une applet est un `Container`

Un `Container` est une identité graphique capable de contenir plusieurs composantes. Il a en charge la gestion d'un ensemble de composantes. Une partie de son interface offre donc les services attendus d'un ensemble

- `add(...)` avec ces diverses déclinaisons. Insère une composante dans le conteneur.
- `getComponent(..)` permet d'obtenir une coomponent en fonction d'un point, d'une position. On peut aussi obtenir la liste de toutes les composantes incluses dans un conteneur.
- `getComponentCount()` retourne le nombre de composantes à l'intérieur du conteneur.
- `remove(...)` avec ces diverses déclinaisons. Supprime une composante de conteneur.

Un exemple d'utilisation



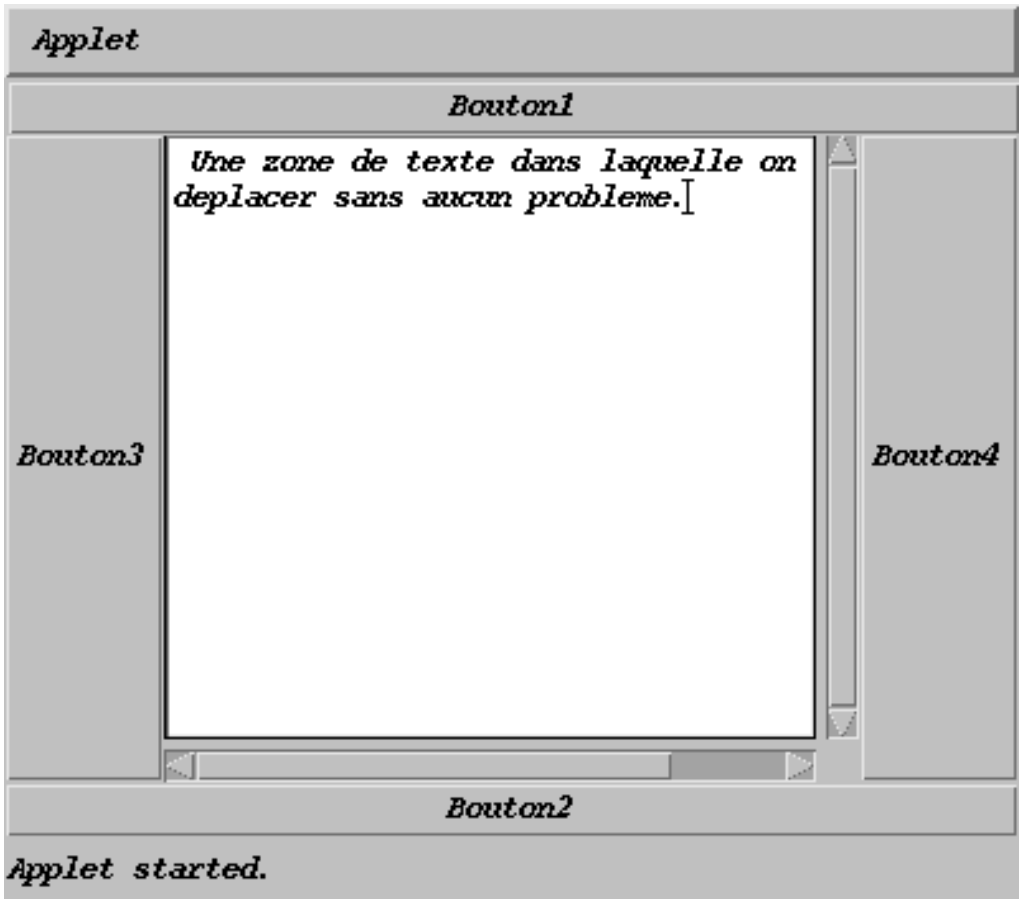

```
import java.awt.*;
import java.applet.*;
public class AppletConteneur extends Applet
{
  public void init()
  {

    add(new Button("Bouton1"));
    add(new Button("Bouton2"));

    add(new Label("Un texte"));
    add(new TextField("Un texte"));

  }
}
```

Un `Container` contient un `Layout`

Un "Layout" manager permet d'agencer les composantes à l'intérieur d'un conteneur. Il existe 5 types de "Layout Manager" avec leurs caractéristiques spécifiques.

1. `FlowLayout` les composantes sont placées de la droite vers la gauche.
2. `BorderLayout` les composantes sont placées dans 5 zones `North,South,East,West,Center`.
3. `GridLayout` les composantes sont placés dans une grille définie par le nombre de lignes et le nombre de colonnes. Toutes les composantes sont ajustées à la même taille.
4. `GridBagLayout` les composantes sont placés dans une grille définie par le nombre de lignes et le nombre de colonnes. Les composantes ne sont réajustées.
5. `CardLayout` les composantes sont placées les unes sur les autres et l'on peut les déplacer les unes sur les autres.

Une Utilisation d'un BorderLayout




```
import java.awt.*;
import java.applet.*;

public class AppletConteneur extends Applet
{
  public void init()
  {

    setLayout(new BorderLayout());
    add("North", new Button("Bouton1"));
    add("South", new Button("Bouton2"));
    add("West", new Button("Bouton3"));
    add("East", new Button("Bouton4"));
    add("Center",new TextArea("Un texte au centre"));
  }

}
```

Gestion des évènements en Java

Dans une section, on a vu qu'une Applet hérite du comportement d'une composante pour la partie graphique. Elle hérite aussi des méthodes de traitement des événements.

Certains évènements del' AWT version jdk1.0.*

- évènements souris
  - MOUSE_DOWN;
  - MOUSE_UP;
  - MOUSE_ENTER;
  - MOUSE_EXIT;
  - MOUSE_DRAG;
- évènements clavier
  - KEY_PRESS
  - KEY RELEASE
  - KEY_ACTION
  - KEY_ACTION_RELEASE
- évènements fenêtres;
  - GOT_FOCUS
  - LOST_FOCUS
  - WINDOW_DESTROY
  - WINDOW_ICONIFY
  - WINDOW_MOVED
  - WINDOW_EXPOSE

– ACTION_EVENT cet évènement est un des plus important évènement de l'AWT. Il est crée lorsqu'une interaction avec une composante se produit. Par exemple, appuyer sur un bouton, appuyer sur return dans une composante `TextField`, selection d'un item dans une instance de `CheckBox`,..... La variable `target` représente l'objet émetteur.

La manière la plus générale de traiter un évènement est de spécialiser la méthode `boolean handleEvent(Event evt)`. Dans ce cas, le traitement peut s'avérer fastidieux car il comporte un énorme `switch/case` qui associe à chaque type d'événement le traitement adéquat. L'AWT offre la possibilité de traiter séparément par des méthodes différentes le traitement des événements provenant

– des actions `boolean action(Event, Object)`, On identifie l'événement et l'objet dans lequel il a eu lieu.

– du clavier par exemple `boolean keyDown(Event)`;

– de la souris `mouseDown(Event, int, int)`;

Par exemple, le fait de cliquer sur le bouton provoque le déplacement du carrré.

```
public boolean action(Event evt, Object arg)
{
  int depx = (int) (Math.random()*5);
  int depy = (int) (Math.random()*5);

  depx = depx > 2 ? -depx: depx;
  depy = depy > 2 ? -depy: depy;

  carre.translate(depx, depy);
  repaint();
  return true;
}
```

L'applet a la possibilité de spécialiser les traitements de la souris et ainsi de ne s'interesser qu'à ceux-ci.

- mouseDown(Event, int, int) : Bouton de la souris enfoncé dans la composante;
- mouseDrag(Event, int, int) : Déplacement de la souris bouton enfoncé;
- mouseEnter(Event, int, int): La souris rentre dans l'applet.
- mouseExit(Event, int, int) : La souris sort de l'applet.
- mouseMove(Event, int, int) : La souris se déplace bouton non enfoncé
- mouseUp(Event, int, int) : Bouton de la souris relâché dans l'applet.

Tranformer une applet en application

```
import java.applet.Applet;
import java.awt.Event;
import java.awt.Frame;
import java.awt.Label;
import java.awt.event.*;

public class StarterCombined extends Applet
{
   private Label label;

   public static void main(String args[])
     {
       StarterCombinedFrame app = new
           StarterCombinedFrame("Starter Application");
       app.setSize(300,100);
       app.show  ();
       System.out.println("StarterCombinedFrame::main()");
   }
   public void init() {
       System.out.println("Applet::init()");
     }
   public void start() {
       System.out.println("Applet::start()");
       label = new Label("Starter");
       add(label);
   }
   public void stop() {
     System.out.println("Applet::stop()");
     remove(label);
   }
   public void destroy() {
      System.out.println("Applet::destroy()");
   }
}

class StarterCombinedFrame extends Frame {
      public StarterCombinedFrame(String frameTitle) {
          super(frameTitle);

        StarterCombined applet = new StarterCombined();
        applet.start();
        add  (applet, "Center");
        addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent event) {
                dispose();
                System.exit(0);
        }});
   }
}
```

LES THREADS

Java offre la possibilité de créer des processus pendant sont éxécutions.

Une `Thread` est un processus indépendant représenté par un objet. Cet objet est utilisé pour gérer, contrôler et synchroniser l'éxécution.

- Instruction courante;
- Environnement;
- Un état;
- Une priorité;

Pour créer un `Thread` on peut spécialiser la classe `Thread` ou bien implémenter l'interface `Runnable`. Si on implémente `Runnable` on encapsule le processus dans une `Thread`.

Tout `Thread` pour être actif doit éxécuter la méthode `run()`. On lancer l'éxécution d'un `Thread` par la méthode `start()` qui déclenche `run()`.

Un exemple Runnable

```
class ThreadRun implements Runnable
{
  String name;
  Thread monThread;

  ThreadRun(String s)
    {
      name      = s;
      monThread = new Thread(this);

      monThread.start();
    }

  public void run()
    {
      for(int i = 0; i < 100; i++)
         {
           try
            {
              Thread.sleep(10);
            }
          catch(Exception e){}
          System.out.println("proc " + name + " --- i = " + i);
         }
    }

  public static void main(String [] S)
    {
     ThreadRun tr1 = new  ThreadRun("P1");
     ThreadRun tr2 = new  ThreadRun("P2");
    }
}
proc P1 --- i = 0
proc P2 --- i = 0
proc P1 --- i = 1
proc P2 --- i = 1
proc P1 --- i = 2
proc P2 --- i = 2
```

Un exemple Thread

```
class ThreadThread extends Thread
{
  String name;
  Thread monThread;

  ThreadThread(String s)
    {
      name      = s;
      start();
    }

  public void run()
    {
      for(int i = 0; i < 100; i++)
       {
          try
            {
              Thread.sleep(10);
            }
          catch(Exception e){}
          System.out.println("proc " + name + " --- i = " + i);
       }
    }

  public static void main(String [] S)
    {
     ThreadThread   tr1 = new  ThreadThread("P1");
     ThreadThread   tr2 = new  ThreadThread("P2");
    }
}
```

Les Threads et les Applets

Sun Oct 26 13:35:48 GMT+03:30 1997

```
import java.applet.*;
import java.awt.*;
public class Horloge extends Applet implements Runnable
{
  Thread personnel;
  public void run()
    {
     while(true)
       {
        try {
            Thread.sleep(1000);
            }
         catch(Exception e){}
         repaint();
       }
     }
  public void start()
    {
      if (personnel == null)
       {
          personnel = new Thread(this);
          personnel.start();
        }
    }
  public void stop()
    {
      if (personnel != null)
        {
          personnel.stop();
          personnel = null;
        }
    }

  public void paint(Graphics g)
    {
      g.drawString(new java.util.Date().toString(), 10, 25);
    }

}
```

Déplacement du Carré

```
import java.awt.*;
import java.applet.*;
public class DeplacementCarreApplet extends Applet implements Runnable
{

  Carre carre;
  Thread personnel;

 public void run()
    {
      while(true)
        {
          try {
            Thread.sleep(2000);
          }
          catch(Exception e){}
          int depx = (int) (Math.random()*5);
          int depy = (int) (Math.random()*5);

          depx = depx > 2 ? -depx: depx;
          depy = depy > 2 ? -depy: depy;

          carre.translate(depx, depy);
          repaint();
        }
    }
  public void start()
    {
      if (personnel == null)
        {
          personnel = new Thread(this);
          personnel.start();
        }
    }

  public void stop()
    {
      if (personnel != null)
        {
          personnel.stop();
          personnel = null;
        }
    }
  public void init()
  {
    carre = new Carre();
  }
```

```
  public void paint(Graphics g)
  {
    carre.paint(g);
  }
}
```

La synchronisation des Threads

Quand deux Threads veulent accéder à la même information, il peut y avoid des problèmes. Comme cela peut se produire sur cette exemple.

```
class Test
{
    private int value = 0;

    public void Inc(int v)
    {
      value += v;
    }

    public void Dec(int v)
    {
      value -= v;
    }

    public int getValue()
    {
      return values;
    }
}
```

Le processus P1 commence a accéder à `value` pour l'incrementer. Mais il s'arrete avant de l'affecter. Le processus P2 consulte `value` et la décrémente. P1 reprend son exécution, il incremente value. Bénéfice pour le particulier.

Pour éviter ce genre de problème, il faut utiliser le mot clef `synchronized`. L'accés à chaque objet est géré par un moniteur. Un seul processus peut accéder à l'objet en utilisant une méthode `synchronized`.

```
   public synchronized void Inc(int v)
   { }
   public  synchronized void Dec(int v)
   {}
   public  synchronized int getValue()
   {}
}
```

Suivant le même principe, on peut contrôler l'accés à une variable. Sur cette exemple, personne ne peut évoquer value, tant que le bloc n'est pas terminé.

```
synchronized(value) {.....}
```

Les primitives `wait` et `notify`

- `wait()` le possesseur du verrou se met en attente d'une condition. Dans ce cas, il relâche le verrou. Lorsque la condition est remplie, il attend l'obtention du verrou pour continuer.

```
   synchronized void Contexte()
   {
      while(!condition)
        wait();
```

```
    suite
}
```

– `notify()` Prévient les objets en attente (`waiting`) qu'il y a eu une évolution. Le `notify()` conserve l'accés au verrou.