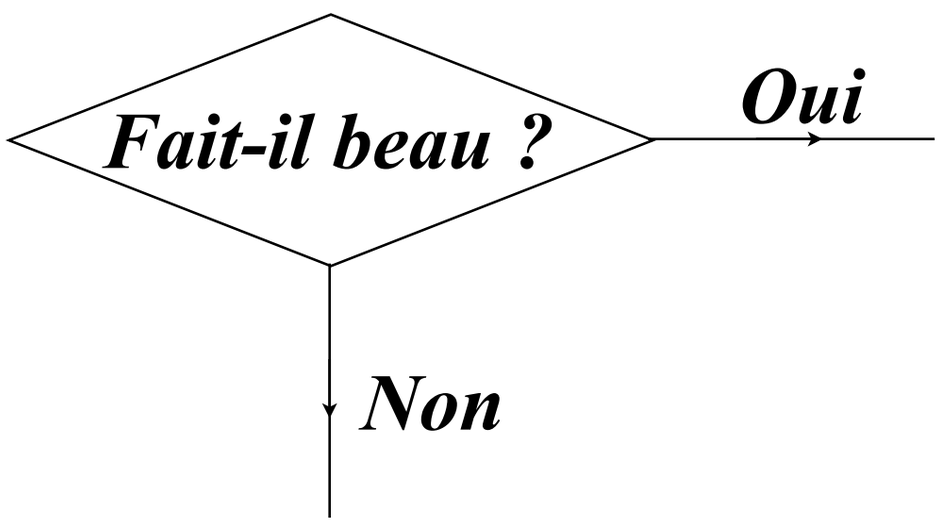
Programmation par Objets - Printemps 2010
Frédéric Mallet & Jean-Paul Roy
L1 Info - Faculté des Sciences

http://deptinfo.unice.fr/~roy

La Prise de Décision



Les comparaisons numériques

- Les **opérateurs de comparaison** usuels fonctionnent sur les nombres entiers et approchés :

Expression — *a pour valeur*

`3 > 5` → `false`

`3 < 5` → `true`

`5 < 2 + 1` → `false` *(+ est prioritaire sur <)*

En cas de doute :
5 < (2 + 1)

- Les valeurs constantes `true` et `false` sont les **valeurs booléennes**. On peut afficher une valeur booléenne.

```
print(3 < log(2));
```

Run →

```
false
```

- Une **expression booléenne** est une expression dont la valeur est booléenne.

ATTENTION !

- Les signes $\leq$ et $\geq$ des maths se notent <= et >= en programmation !

a pour valeur

```
3 >= PI          ⟶   false
3 <= 2 + 1       ⟶   true
```

- Le signe $=$ des maths se note == en Processing/Java/C lorsqu'il sert à vérifier qu'une égalité est vraie.

```
pow(2,10) == 1024.0          ⟶   true
PI == 3.1416                 ⟶   false
PI == 3.1415927              ⟶   true
```

- Mais ne vous fiez pas au dernier exemple ! En principe **il est illusoire de demander l'égalité <u>exacte</u> entre deux nombres <u>approchés</u>** !

```
0.1+0.1+0.1+0.1+0.1+0.1+0.1 == 0.7    ⟶   false
```

Distinguer = (affectation) et == (égalité)

- C'est l'une des erreurs majeures du débutant !

```
int x = 2, y = 3, z = 6;      // déclarations + initialisations
z = z + 1;                    //  z devient égal à z + 1, donc à 7
y == x + 1                    // est-ce que y vaut x + 1, c'est à dire 3 ?
```

→ `true`

```
x == y + 1                    // est-ce que x vaut y + 1, c'est à dire 4 ?
```

→ `false`

```
x + 2 == y - 1                // est-ce que x+2 vaut y-1, c'est à dire 4==2 ?
```

→ `false`

- Le résultat de x == y est bien un **booléen**.

L'expression conditionnelle if ... else ...

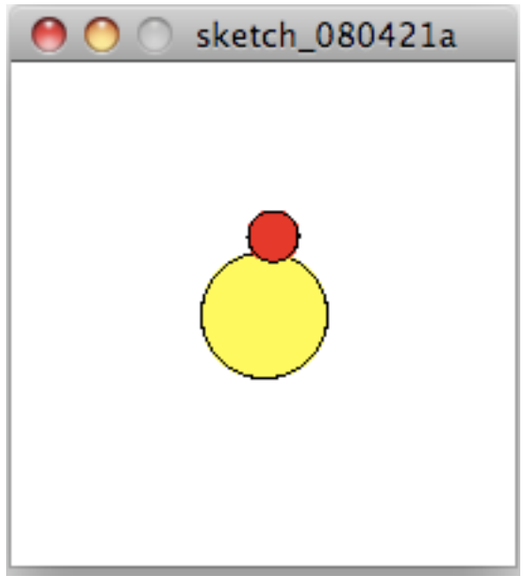
- Vitale, elle permet de prendre une décision à un moment donné.

SI ... ALORS ... SINON ...

if (...) { ... } else { ... }

- Exemple. Une **balle vibrante** rouge part du centre du canvas, elle suit un mouvement aléatoire.

- On utilise la méthode `random(n)` dont la signature est `float random(float n)` et dont le résultat est un nombre approché aléatoire de `[0;n[`.
Par exemple, `random(6)` ∈ `[0.0;6.0[` au petit bonheur la chance...



- Version 1. On laisse la balle aller où elle veut...

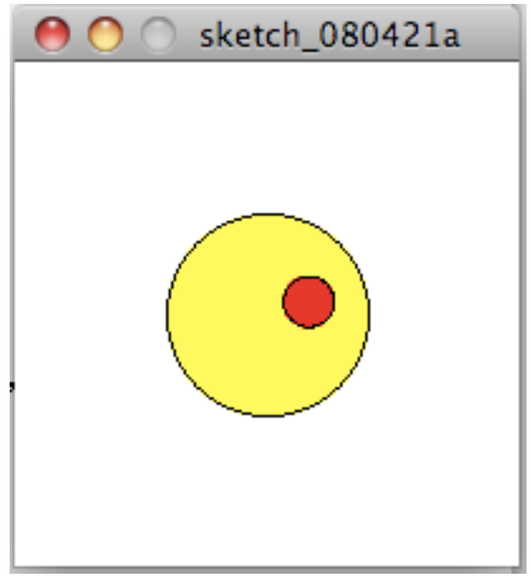
```
int largeur = 200, hauteur = 200;          // dimensions du canvas
int rayonBalle = 10;                       // rayon de la balle rouge
int rayonCercle = 50;                      // rayon du cercle jaune
float x, y;                                // position de la balle

void setup() {
  size(largeur, hauteur);
  x = largeur/2;                           // position initiale
  y = hauteur/2;                           // de la balle
}

void draw() {
  background(255,255,255);                 // couleur de fond du canvas
  fill(255,255,0);                         // remplissage du cercle en jaune
  ellipse(largeur/2, hauteur/2, 2 * rayonCercle, 2 * rayonCercle);
  fill(255,0,0);                           // remplissage de la balle en rouge
  ellipse(x, y, 2 * rayonBalle, 2 * rayonBalle);
  float dx = random(6) - 3, dy = random(6) - 3;      // déplacement aléatoire
  x = x + dx;                              // et mise à jour de la position
  y = y + dy;
}
```

- Version 2. Je veux **empêcher son centre de sortir du cercle** jaune.

Je teste la distance des centres et la compare au rayon du cercle :




```
void draw() {
  background(255,255,255);
  fill(255,255,0);
  ellipse(largeur/2, hauteur/2,2*rayonCercle,2*rayonCercle);
  fill(255,0,0);
  ellipse(x,y,2*rayonBalle,2*rayonBalle);
  float dx = random(6)-3, dy = random(6) - 3;
  float xSuiv = x + dx;              // la position suivante
  float ySuiv = y + dy;
  if (distance(xSuiv,ySuiv,largeur/2,hauteur/2) < rayonCercle) {
    x = xSuiv;                       // si la position suivante est encore dans
    y = ySuiv;                       //     le cercle, je peux mettre à jour la position
  }                                  // sinon je ne fais rien...
}

float distance(float x1, float y1, float x2, float y2) {
  return sqrt(pow(x1-x2,2) + pow(y1-y2,2));
}
```

une fonction auxilliaire

- Dans l'exemple précédent, nous avons utilisé la forme :

SI ... ALORS ...

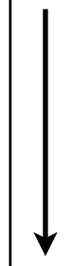
qui s'exprime en Java sous la présentation suivante :

```
if (condition) {
   ...
}
```

- Notez que la condition est une expression booléenne entre **parenthèses**, et que les instructions qui seront exécutées si la condition est vérifiée sont entre **accolades**.

- Une suite d'instructions entre accolades se nomme un **bloc**.

```
if (condition) {
   instruction1;
   instruction2;
   ...
}
```



- Mais dans la plupart des cas, nous devrons prévoir ce qui se passe si la condition **n'est pas** vérifiée.

SI ... ALORS ... SINON ...

qui s'exprimera cette fois sous la présentation avec deux blocs :

```
if (condition) {
   instruction1;
   instruction2;
   ...
} else {
   instruction8;
   instruction9;
   ...
}
```

- Vous aurez bien noté que le mot "then" est sous-entendu !

• Exemple. Je veux que le contour du cercle soit électrifié ! Si la balle **touche le cercle**, elle reçoit une **décharge** et se retrouve brutalement au centre !

```
if (distance(xSuiv,ySuiv,largeur/2,hauteur/2) < rayonCercle) {
  x = xSuiv;
  y = ySuiv;
} else {
  println("Aïe !");
  x = largeur/2;
  y = hauteur/2;
}
```

• L'oubli du `else` est aussi une erreur fréquente dans les programmes !

• Les instructions dans chaque bloc sont décalées par-rapport à la marge, pour bien montrer à quel niveau elles se trouvent. On dit qu'elles sont indentées. **Respectez l'indentation !**

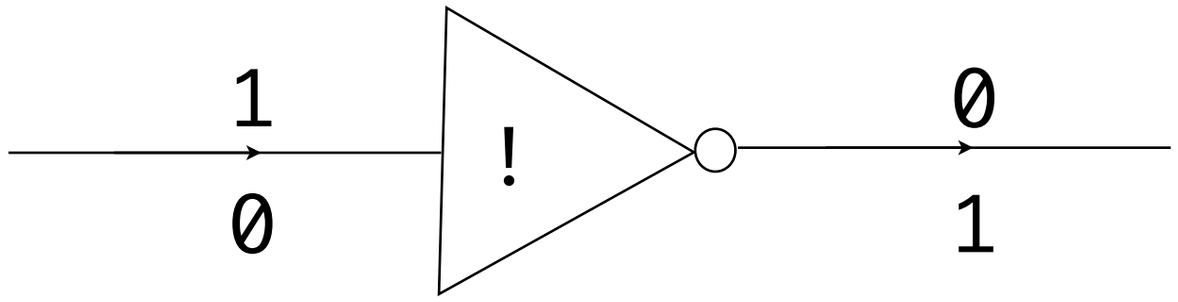
Auto Format !

Les opérateurs booléens ! && ||

- La **négation** d'une expression booléenne E se note !(E). Par exemple, la négation de x == y se note !(x == y), dont l'abréviation est x != y

- Attention, la négation de x < y est x >= y.

- Si E vaut true, alors !E vaut false, et inversement. Les électroniciens disent que le circuit not est un ***inverseur***.



la porte "NOT"

- Exemple : si le nombre entier n n'est pas premier :

```
if (!premier(n)) {
   ...
}
```

avec

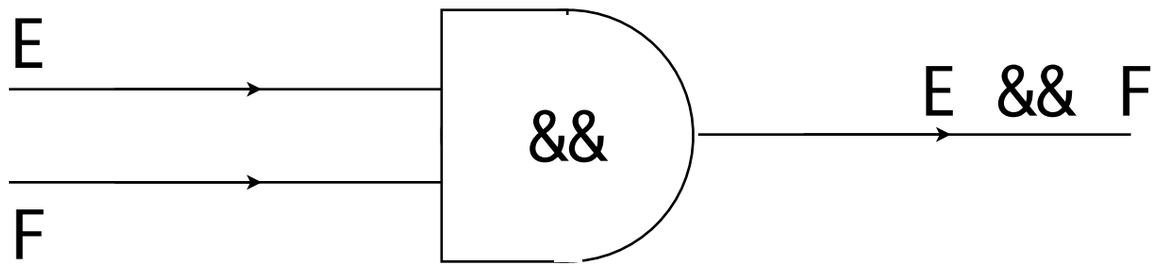
```
boolean premier (int n) {
  ...
}
```

• La **conjonction** $E \; et \; F$ de deux expressions booléennes `E` et `F` se note en Java `E && F`. Par exemple, pour exprimer que `n` est premier et que `n` est plus grand que 100, la condition s'écrira :

```
premier(n) && (n > 100)
```

• Une expression booléenne `E && F` vaut `true` si et seulement si `E` vaut `true` et `F` vaut `true`. Elle vaut `false` dans tous les autres cas.

• Les électroniciens schématisent un circuit `ET` de la manière suivante :



la porte "AND"

• COURT-CIRCUIT ! Attention, dans l'expression `E && F`, l'expression `E` est évaluée en premier. Si elle vaut `false`, le résultat du `&&` est `false` sans avoir besoin d'évaluer `F`. Ceci est TRES IMPORTANT !

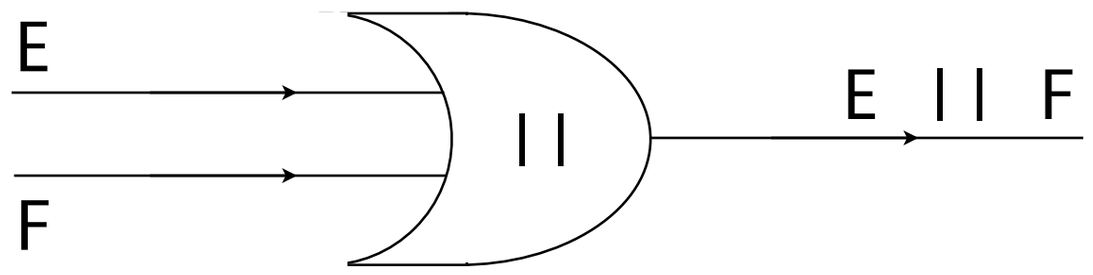
```
if ((x != 0) && (1/x > y)) ...
```

• La **disjonction** $E \; ou \; F$ de deux expressions booléennes `E` et `F` se note en Java `E || F`. Par exemple, pour exprimer que `n` est premier ou qu'il est plus grand que 100, la condition s'écrira :

```
premier(n) || (n > 100)
```

• Une expression booléenne `E || F` vaut `false` si et seulement si `E` vaut `false` et `F` vaut `false`. Elle vaut `true` dans tous les autres cas.

• Les électroniciens schématisent un circuit `OU` de la manière suivante :

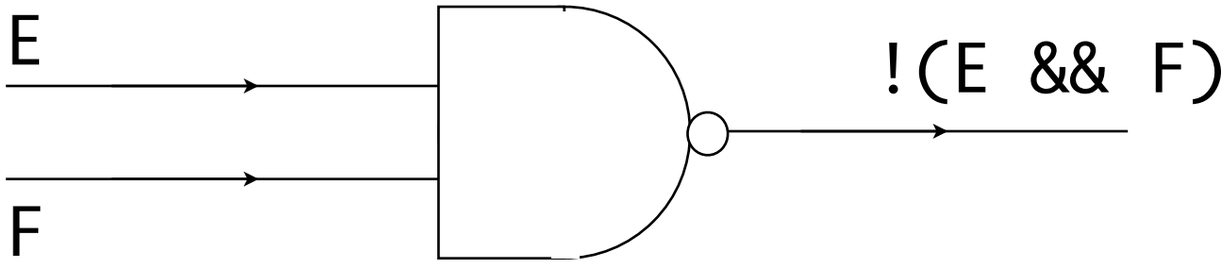


la porte "OR"

• COURT-CIRCUIT ! Attention, dans l'expression `E || F`, l'expression `E` est évaluée en premier. Si elle vaut `true`, le résultat du `||` est `true` sans avoir besoin d'évaluer `F`. Ceci est TRES IMPORTANT !

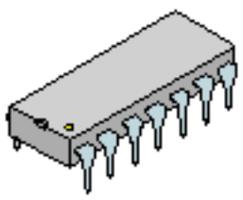
```
if ((x > 0) || (x < -2)) ...
```

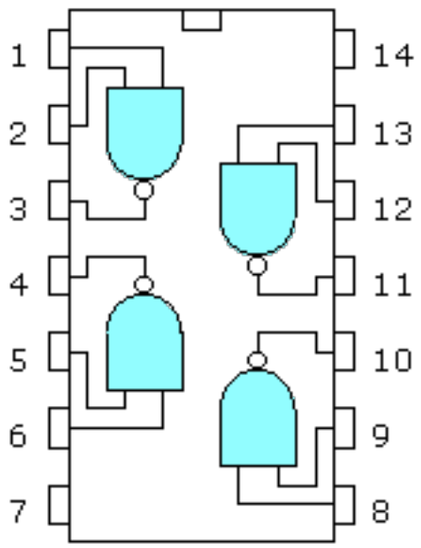
• On peut montrer [cf TD] que **la seule porte NAND** [la négation du AND] **suffit à engendrer les autres portes** ! Cependant elle n'existe pas en Java ! Les électroniciens la connaissent bien.

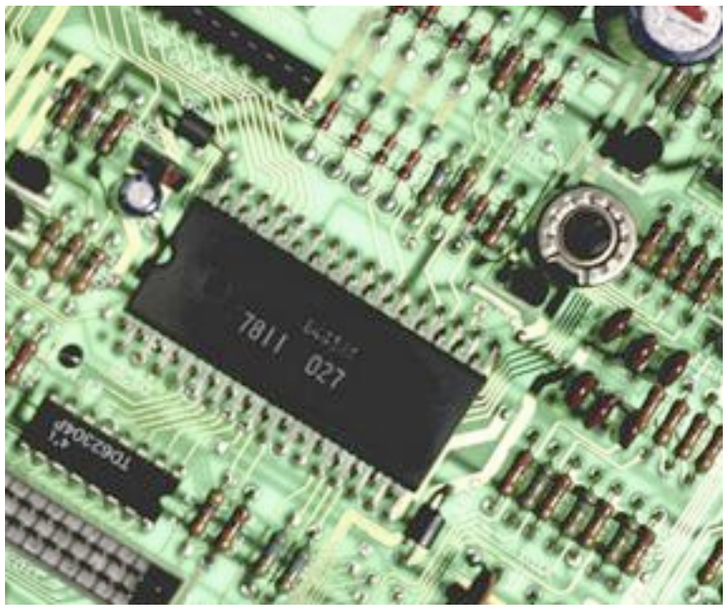


la porte "NAND"

• Les **portes logiques** sont de petits circuits électroniques qui forment, avec les transistors, le coeur de nos ordinateurs :







Les Méthodes

- Les langages objets ont des **méthodes**, les autres ont des **fonctions**.

- Une méthode peut **avoir ou non un résultat** ! Une méthode sans résultat se contente d'effectuer une action (afficher du texte, déplacer une balle dans le canvas, modifier une variable, etc).

```
void méthode(int x)
```

↑ `int` → ∅

pas de résultat, seulement une action !

```
float méthode(int x)
```

↑ `int` → `float`

un (seul) résultat, de type float.

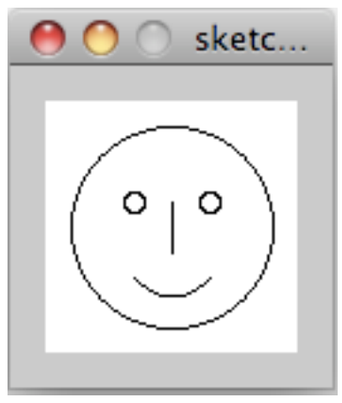
- Par exemple, les méthodes `print(...)`, `line(...)`, `background(...)` n'ont pas de résultat.
- Par contre, les méthodes `log(...)`, `sin(...)`, `sqrt(...)` retournent un résultat.
- Notre méthode `distance(...)` de la page 7 retourne un résultat.
- L'ordre des méthodes n'a pas d'importance !

Les fonctions sans résultat

- Une méthode sans résultat se contente d'**effectuer une action**.
- Les plus célèbres en Processing sont `setup()` et `draw()`. La méthode `setup()` initialise l'animation, tandis que la méthode `draw()` dessine l'image courante et effectue les actions de mise à jour pour passer à l'image suivante.
- Voici une méthode qui affiche un *smiley* à l'écran, sans résultat.

Aucune animation, donc seulement la fonction `setup()` :

```
void smiley() {
  fill(255,255,255);
  ellipse(50,50,80,80);
  ellipse(35, 40, 8, 8);
  ellipse(65, 40, 8, 8);
  line(50,40,50,60);
  bezier(35,70,45,80,55,80,65,70);
}
```




```
void setup() {
  size(100,100);
  background(255,255,255);
  smiley();
}
```

Une méthode ne fait rien si on ne l'invoque (appelle) pas !

Les fonctions avec résultat

- Une méthode avec résultat va **effectuer un calcul** et **retourner le résultat** de ce calcul à celui qui l'a appelée.
- Le retour du résultat à l'appelant se fait avec l'instruction **return** qui abandonne immédiatement la méthode, avec le résultat sous le bras !

```
float foo(int x) {
  float res;
  if(x > 0) {
    res = x;
  } else {
   res = (x + sin(x)) / 2;
  }
 return res;
}
```

signature

déclaration *du résultat (bien typé)*

calcul *du résultat*

retour *du résultat*

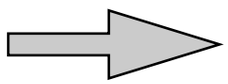
```
void setup() {
  println("foo(5) = " + foo(5));
  println("foo(-5) = " + foo(-5));
}
```

Run

```
foo(5) = 5.0
foo(-5) = -2.0205379
```

- On peut parfois simplifier le schéma précédent :

```
float moyenne(float x, float y) {
  float res;
  res = (x + y) / 2;
  return res;
}
```



```
float moyenne(float x, float y) {
  return (x + y) / 2;
}
```

- Une factorielle `fac(n)` par **récurrence sur n** :

```
int fac(int n) {
  if (n <= 0) {
    return 1;
  } else {
    return n * fac(n - 1);
  }
}
```

$$0! = 1$$
$$n! = n \times (n\text{-}1)! \quad si\ n > 0$$

```
void setup() {
  println("fac(10) = " + fac(10));
  println("fac(17) = " + fac(17));
}
```


```
fac(10) = 3628800
fac(17) = -288522240
```

!

Variables et méthodes

3 types de variables

des ***champs***

des ***paramètres***

```
int largeur = 100, hauteur = 100;

void cercle_inscrit(int gauche, int droite, int haut, int bas) {
  float x = moyenne(gauche, droite);
  float y = moyenne(haut, bas);
  ellipse(x, y, gauche - droite, haut - bas);
}

void setup() {
  size(largeur, hauteur);
  cercle_inscrit(0, largeur, 0, hauteur);
}
```

des ***variables locales***

Portée et durée de vie d'une variable

La portée (visibilité) d'une variable est la partie du code source où l'on peut accéder à sa valeur.

- La portée des champs est la totalité des méthodes définies.
- La portée des paramètres et des variables locales est le bloc de déclaration (i.e. méthode ou accolades les plus proches)

La durée de vie d'une variable correspond à la période pendant laquelle elle subsiste avant sa destruction.

- Les champs gardent leur valeur au cours de la vie du programme.
- Les paramètres et les variables locales ont une durée de vie limitée au seul appel du bloc (durée de l'appel de la méthode).