Introduction à Processing

Un environnement de programmation

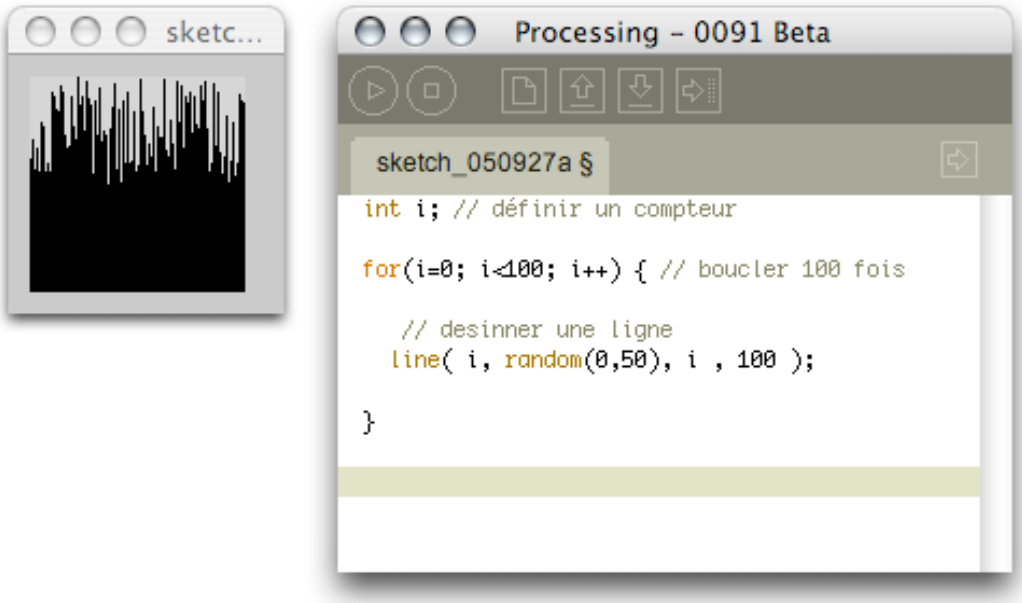
conçu par des artistes, pour les artistes.

Par Douglas Edric Stanley
Adapté par Frédéric Durieu

Table des matières

1. Introduction

Processing est un environnement de programmation conçu par des artistes, pour les artistes. Il a été imaginé dès le départ comme un environnement d'apprentissage pour des gens souhaitant rapidement apprendre à générer des images sur ordinateur. En conséquence, il protège l'utilisateur de tout un tas de complexités qu'on aurait normalement à apprendre pour générer un programme. L'environnement de Processing est archi simple : un champ de texte pour écrire le programme, un bouton "play" pour jouer l'image.



Quand on appuie sur "play", notre programme se visualise sous nos yeux.

Ce cours utilisera cet environnement pour apprendre les éléments les plus importants de la programmation d'images interactives, génératives, ou tout simplement algorithmiques. Processing peut être un programme d'initiation pour des ateliers mais peut être aussi utilisé pour créer de réelles œuvres online ou offline. Plusieurs exemples d'œuvres créées avec Processing se trouvent en ligne ici : http://www.processing.org/exhibition/index.html

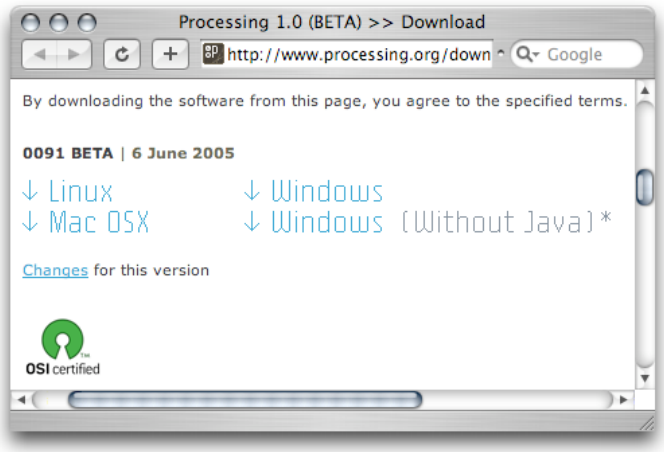
Processing est un logiciel libre (open-source) totalement gratuit. Il marche sur les plateformes Mac, Windows et Linux. Il existe également des versions pour téléphones portables et circuits électroniques.

Processing a été conçu au laboratoire Aesthetics + Computation Group du MIT Media Lab. Processing est plus ou moins le prolongement du projet Design By Numbers, créé par le directeur du laboratoire, le célèbre artiste programmeur John Maeda. M. Maeda a écrit un livre sur son langage – Design By Numbers de John Maeda – où il explique la philosophie de son travail à travers des sortes de Haikus où il fait l'éloge de la simplicité et de l'économie dans la programmation d'images. Beaucoup de traces de ce premier projet sont visibles dans l'environnement Processing : la simplicité de l'interface, la priorité donnée à l'expérimentation et l'apprentissage, ainsi que les nombreuses fonctions que les deux environnements partagent. Les concepteurs de Processing, Casey Reas et Ben Fry, ne cachent pas cet héritage – ils ont de toute façon commencé ce projet alors qu'ils étaient encore étudiants au ACG.

2. Configuration

2.1 Utiliser Processing

La configuration de Processing sur votre ordinateur est facile. On commence par télécharger le logiciel sur le site de http://www.processing.org/download/.

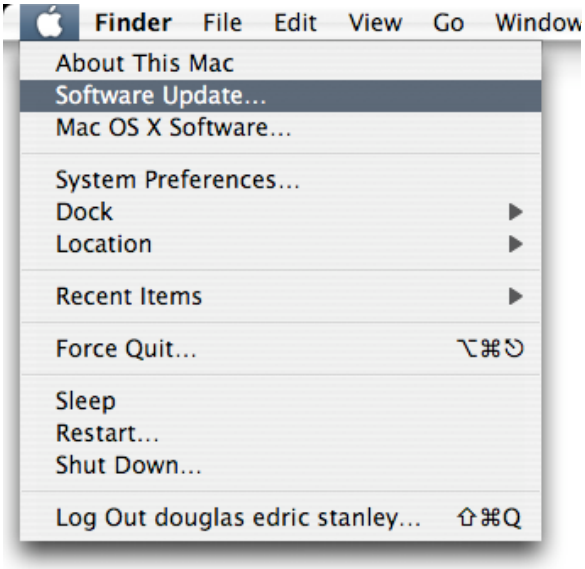


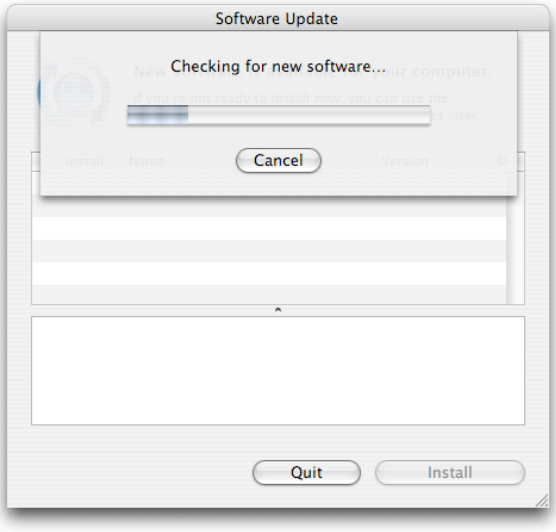
Si on est sur la plateforme Windows, "(without Java)", offre un téléchargement plus rapide, mais nécessite une installation au préalable de la bonne version de Java.

2.2 Java

Processing est en réalité une interface simplifiée de la plateforme de développement Java. Si Processing a du mal à fonctionner, il se peut que vous ayez besoin de mettre à jour Java sur votre machine.

Sur Macintosh, rien de plus simple : allez dans le **menu pomme > mise à jour des logiciels**.

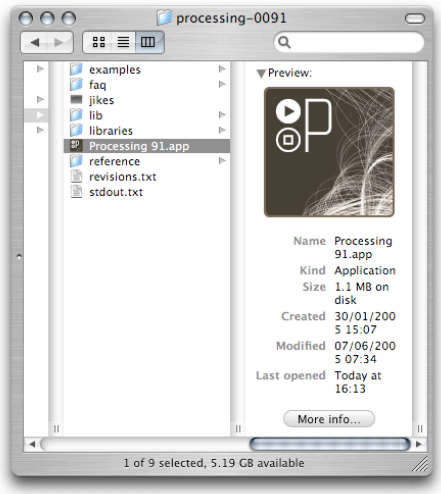




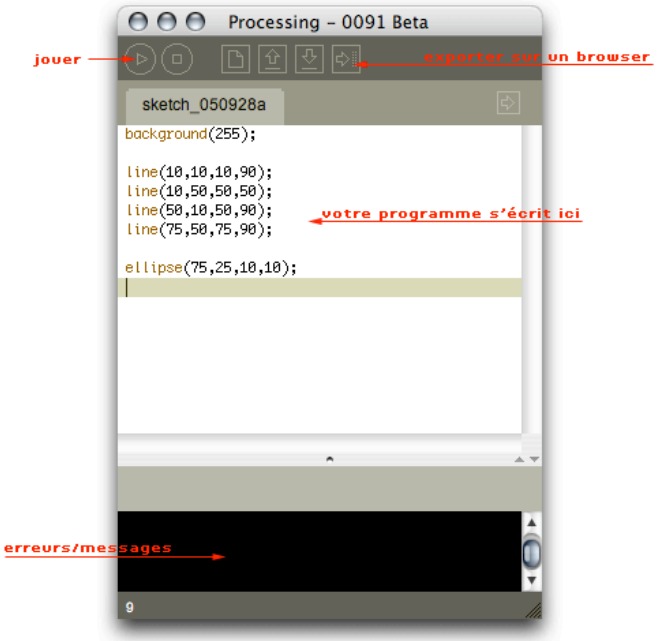
Si votre Macintosh est à jour, vous avez la dernière version de Java installée sur votre machine. Apple fournit Java gratuitement — il est installé sur toutes les versions de Mac OS X par défaut.

2.3 L'environnement

Une fois Processing téléchargé sur votre machine, rangez-le dans le dossier **Applications**, puis cliquez deux fois sur l'icône de l'application pour le démarrer.



L'interface graphique de Processing s'ouvre. Vous remarquerez tout de suite que l'environnement a été simplifié à l'extrême.



Il y a quatre espaces fondamentaux dans l'environnement Processing. D'abord le **champ de texte** où on écrit le programme. C'est ici que l'on travaille.

Ensuite, on fait fonctionner notre programme en appuyant sur le bouton **jouer** (le triangle jaune). Si nous avons mal écrit notre programme, en ne respectant pas la syntaxe par exemple, un message d'erreur apparaît en bas de la fenêtre, dans la **console de communication**. La console est très importante car elle permet à Processing de vous signaler les erreurs de votre programme. Elle affiche aussi les informations que vous lui demandez de vous écrire (par code). Enfin, il reste le bouton **Export** qui permet d'exporter l'animation en "Applet". Cet Applet pourra être directement intégré dans un site web.

2.4 Notre premier programme

Par tradition, on apprend toujours un nouveau langage de programmation informatique en commençant par écrire "Hello World !" sur l'écran. C'est la preuve que nous pouvons communiquer à partir de notre application avec le monde, mais aussi que nous traitons notre programme comme le début d'une entité autonome qui parlerait en son nom propre. Comme Processing est conçu tout d'abord pour générer des images, je propose de suivre la tradition, mais en l'adaptant selon la logique de l'environnement.

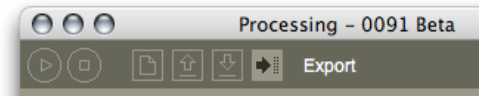
Tapez les lignes suivantes dans le champ de texte, puis jouez.

```
// bonjour : les 2 barres obliques indiquent que cette ligne est un commentaire

background(255);

line(10,10,10,90);
line(10,50,50,50);
line(50,10,50,90);
line(75,50,75,90);

ellipse(75,25,10,10);

println("Hello World!");

// fin du programme
```

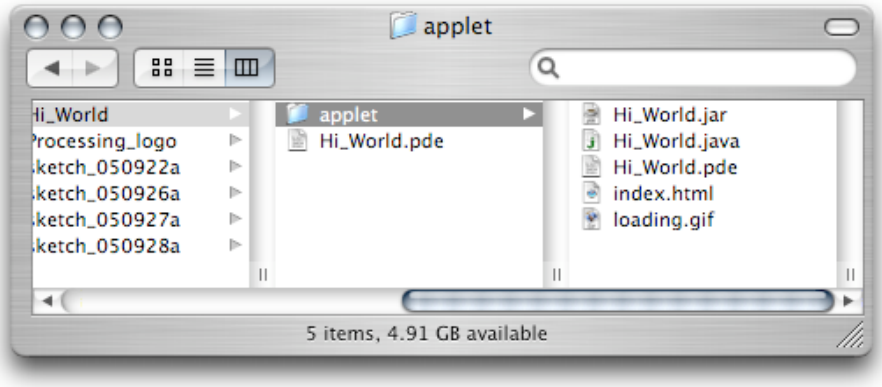
Une fois remis de l'émotion d'avoir créé un premier dessin algorithmique, vous allez l'exporter pour que le reste de l'univers puisse en profiter. Il faut d'abord donner un nom à ce qui, pour l'instant, n'était qu'un "sketch" (croquis) nommé avec la date actuelle. Pour cela cliquez sur **File > Save As...** et donnez un nouveau nom à votre programme. Processing créera automatiquement un dossier du même nom que votre programme et il placera votre programme dans ce dossier.

2.5 Dites bonjour au World Wide Web

Nous allons maintenant exporter notre programme pour le rendre compatible avec internet. Cliquez pour cela sur le bouton **Export**.

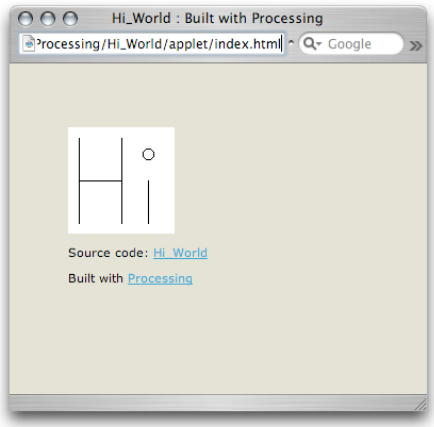


Processing va maintenant compresser votre programme dans ce qu'on appelle un Applet, accompagné d'un fichier HTML + icône GIF pour intégrer facilement l'animation dans votre site. Une fois la compression terminée, le nouveau dossier de votre Applet (le fichier.jar) s'ouvre sur le bureau.



Le seul fichier en réalité dont vous avez besoin est le fichier se terminant en ".jar". Un .jar représente une archive Java et contient tout le code, plus d'éventuelles données supplémentaires (images, polices, vidéo, ...) nécessaires au fonctionnement de votre animation.

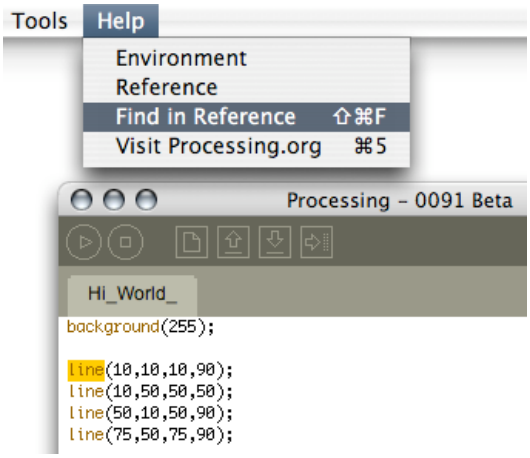
Cliquez sur le fichier "index.html" pour voir votre animation à l'œuvre dans un browser.



Dans le dossier Applet, vous trouverez aussi le fichier .pde qui vous permet de partager votre code avec la communauté des artistes programmeurs. Sur le site de Processing Exhibition (http://www.processing.org/exhibition/) vous remarquerez, par exemple, que la plupart des artistes préfèrent partager le code de leurs programmes. Si cela vous semble étrange, n'oubliez pas que l'économie de l'art n'est pas celle de l'industrie des logiciels, et que le projet Processing est proposé en tant que logiciel libre avec les codes sources visibles pour tout le monde. Il n'y a qu'un pas pour imaginer que vos créations pourraient également être régies par la même ouverture. Ceci dit, le choix de partager vos codes est personnel.

2.6 Aide en ligne

Un point important de l'environnement Processing : son système d'aide en ligne. Si vous voulez avoir plus d'informations sur les mots-clés dans Processing, ou que vous avez oublié dans quel ordre il faut écrire les chiffres dans une fonction, il suffit de cliquer sur un mot dans votre programme et sélectionner **Help > Find in Reference**. Processing vous amènera directement sur le mode d'emploi de cette fonction sur le site de Processing, où on trouvera de nombreux exemples que l'on peut copier-coller pour mieux apprendre la fonction.

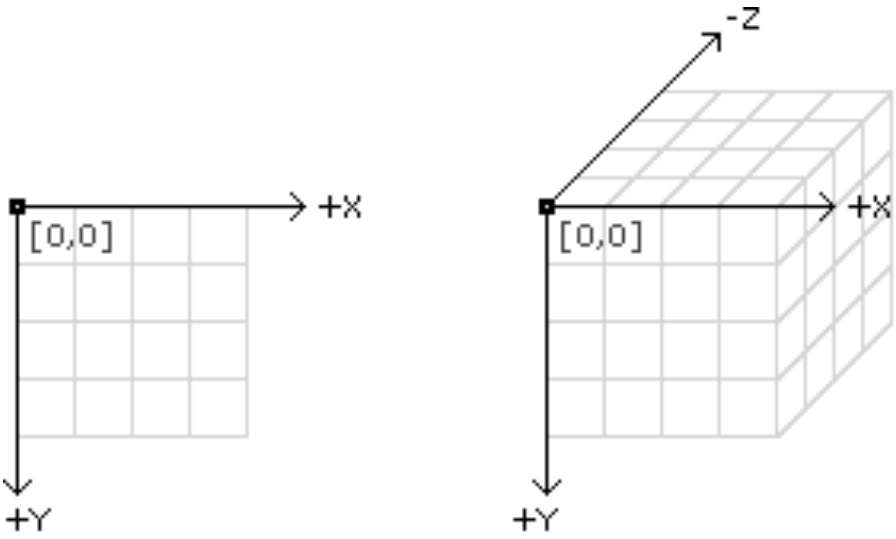


3. Dessiner

Processing vient avec un ensemble très riche de méthodes écrites à l'avance pour faire du dessin. Il contient également de nombreuses méthodes pour générer d'autres formes d'images : images à base de pixels, formes 3d, formes typographiques, images en mouvement, etc. Dans ce chapitre, nous ferons un tour rapide des fonctions de dessin, car ce sont celles qui permettent le mieux d'apprendre la logique de programmation, voire même de la visualiser. Mais avant d'attaquer la programmation à proprement parler, donnons-nous quelques éléments de base.

3.1 Les axes de référence

Dans Processing comme dans presque tous les environnements, on commence à compter en haut à gauche. Le point `[0,0]` est le coin supérieur gauche de la fenêtre de travail. L'axe des X est horizontal vers la droite et l'axe des Y est vertical vers le bas. En 3D, l'axe des Z sort de l'écran vers vous.



3.2 Lignes : line(x1,y1,x2,y2);

Nous avons déjà vu qu'on peut facilement créer une ligne dans Processing. Il suffit d'écrire **line**, d'ouvrir les parenthèses, écrire quatre nombres, fermer les parenthèses et ajouter un point-virgule. Par exemple, le code

```
line(10,10,90,90);
```

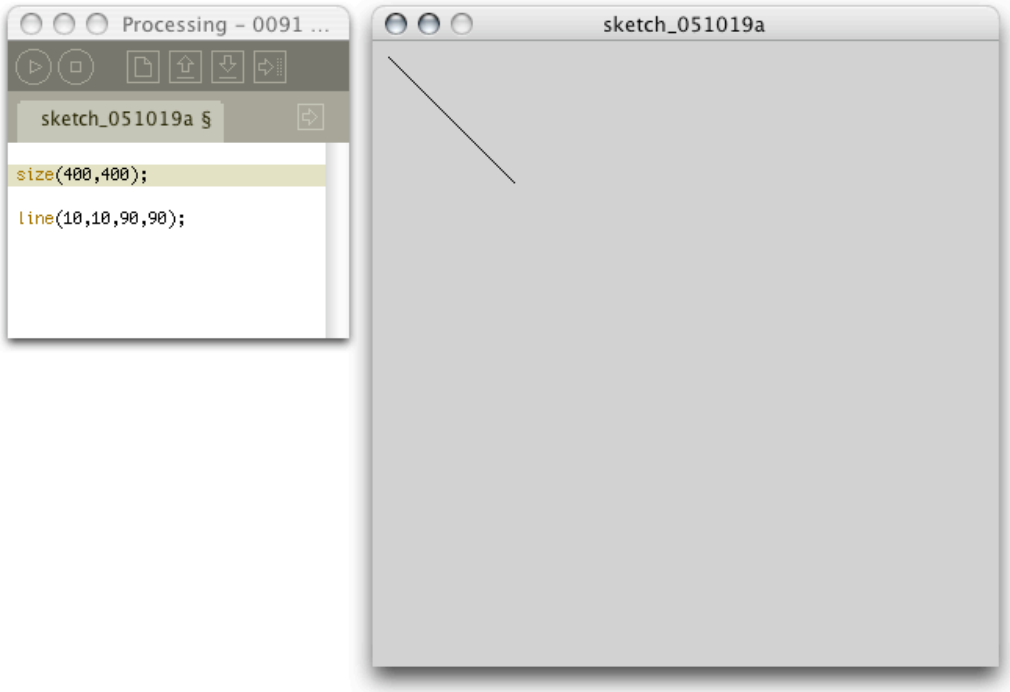
veut dire : commencer la ligne à 10 pixels de la gauche du cadre, à 10 pixels du haut du cadre, puis dessiner cette ligne jusqu'à 90 pixels de la gauche du cadre, et à 90 pixels du haut du cadre.

3.3 Taille de l'espace de dessin : size(l,h);

Si vous n'aimez pas faire des dessins dans un tout petit espace de 100 pixels sur 100 pixels, vous pouvez changer la taille du cadre de votre dessin par la méthode **size** :

```
size(400,400);
```

Ceci créera un cadre de 400 pixels par 400 pixels, dans lequel vous pouvez faire votre dessin.



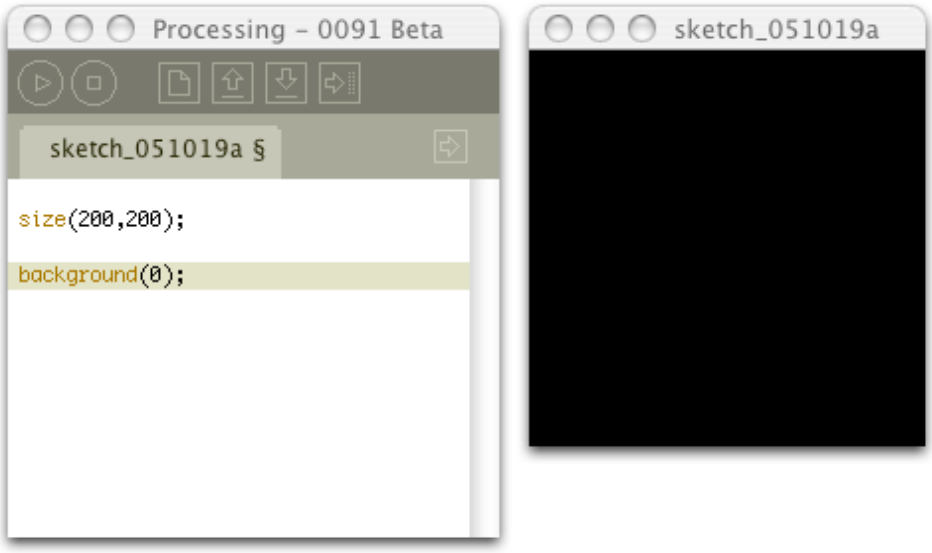
3.4 Couleur du fond : background(…);

Si nous voulons changer la couleur du fond, nous avons une méthode, comme pour line et size, qui s'appelle **background**.

background peut être écrit de deux façons, selon le type de couleur de fond que vous voulez générer. Si vous écrivez ceci :

```
background(0);
```

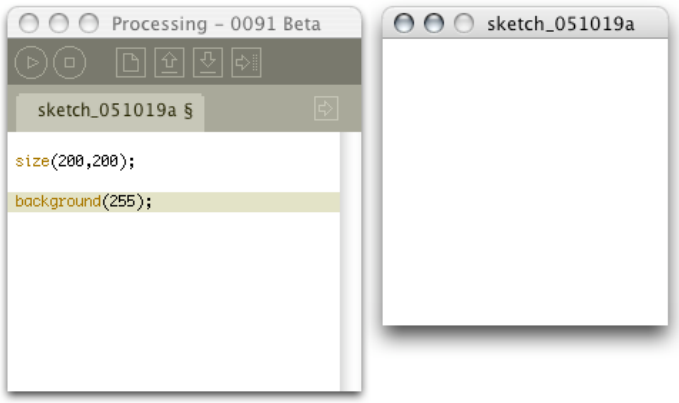
Vous obtiendrez un fond noir :



Et si vous écrivez ceci :

```
background(255);
```

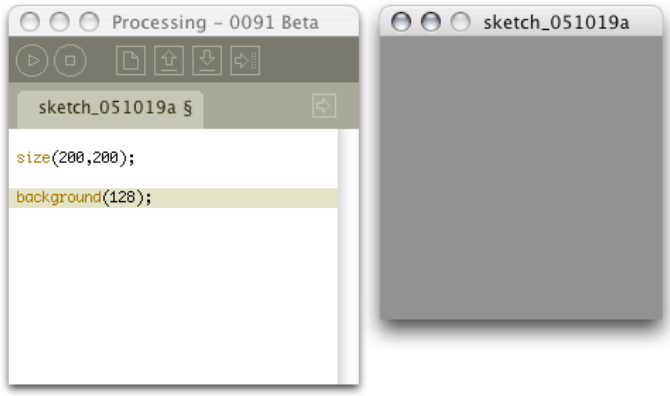
Vous obtiendrez un fond blanc :



Et pour finir, essayez ceci :

```
background(128);
```

...pour obtenir un fond gris :



C'est-à-dire que le chiffre (unique) que vous mettez dans les parenthèses de la méthode **`background();`** contrôle uniquement les intensités allant de noir à blanc en passant par les différents niveaux de gris possibles (entre 0 et 255 il y a 254 différents niveaux de gris, pour être exact).

Il y a une première chose bizarre pour quelqu'un qui apprend à programmer : nous ne comptons pas toujours de 0 à 100 mais de 0 à 255. C'est une habitude qui vient des origines de l'informatique. Dans notre cas, écrire 0 équivaut à intensité minimum alors que 255 équivaut à intensité maximum.

Pour les autres couleurs, nous aurons besoin de 3 valeurs allant de 0 à 255. La première valeur pour le rouge, la deuxième pour le vert et la troisième pour le bleu. Par exemple :

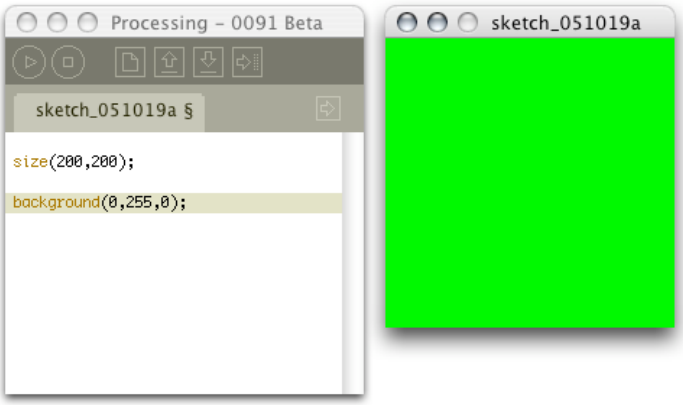
```
background(255,0,0);
```

Cela nous donnera 100% d'intensité pour la couleur rouge, 0% d'intensité pour la couleur verte, et 0% d'intensité pour la couleur bleue.

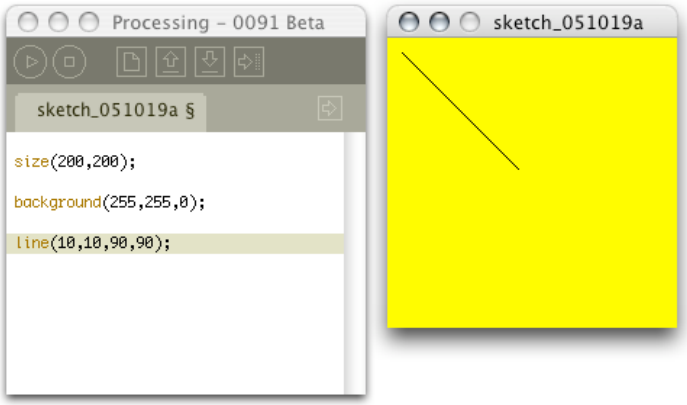
Si nous écrivons :

```
background(0,255,0);
```

Nous aurons un fond de couleur verte :



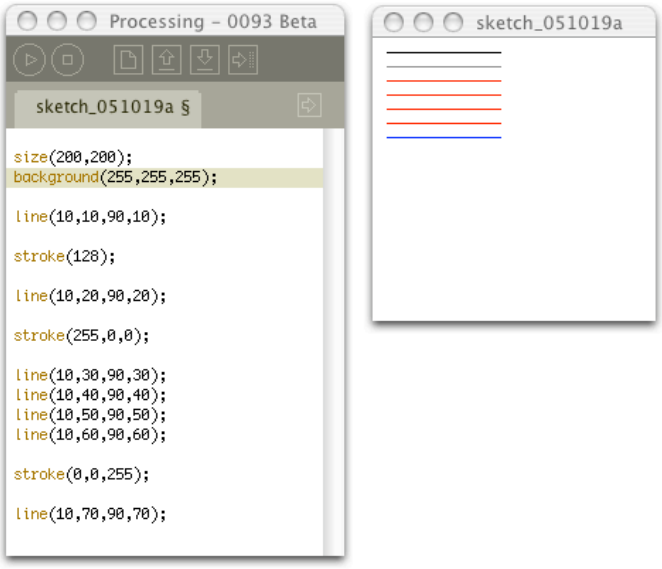
Et si nous voulons mélanger les deux couleurs, par exemple un mélange de maximum rouge avec maximum vert, nous obtiendrons...



...un fond jaune, sur laquelle nous avons ajouté ici notre ligne noire.

3.5 Des traits en couleur : `stroke(…);`

Nous pouvons également changer la couleur des lignes que nous dessinons dans notre cadre. La méthode s'appelle **`stroke();`**. Comme pour le fond, nous pouvons écrire avec 1 ou 3 nombres. Tant qu'on ne change pas la méthode **`stroke`**, nous aurons des dessins de cette couleur :

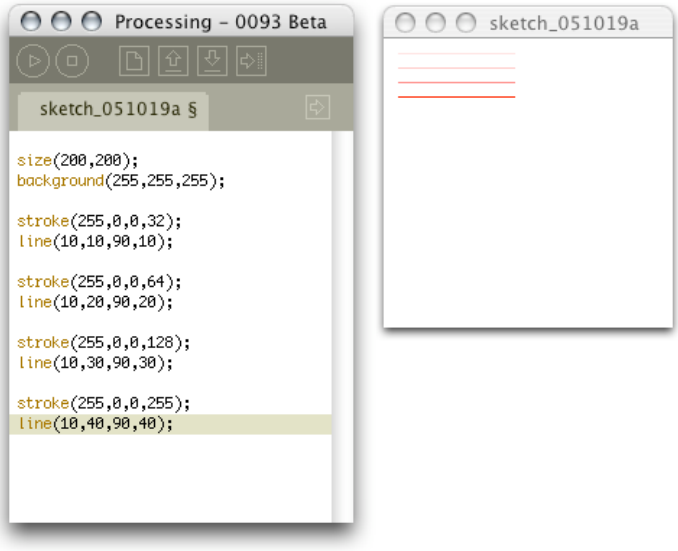


Notez que vous avons obtenu 4 lignes de couleur rouge ; c'est normal puisqu'il y a eu un changement de couleur qu'après les quatre méthodes **line();**. Ceci veut dire que la méthode **stroke();** ne dessine pas directement telle ou telle couleur, elle change simplement la couleur du crayon qui dessinera les prochaines lignes.

3.6 Transparence des traits

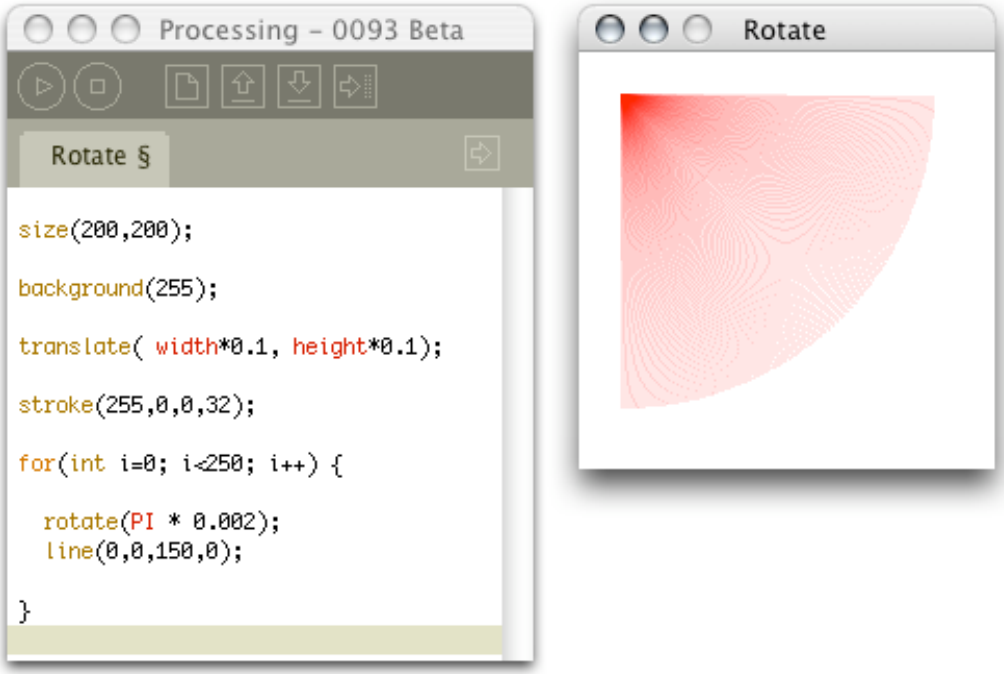
En rajoutant un quatrième nombre dans les arguments de **stroke**, on peut contrôler la transparence.

```
stroke(255,0,0,32);
line(10,10,90,10);

stroke(255,0,0,64);
line(10,20,90,20);

stroke(255,0,0,128);
line(10,30,90,30);

stroke(255,0,0,255);
line(10,40,90,40);
```



Il s'agit du célèbre terme alpha comme dans la couche alpha de Photoshop.

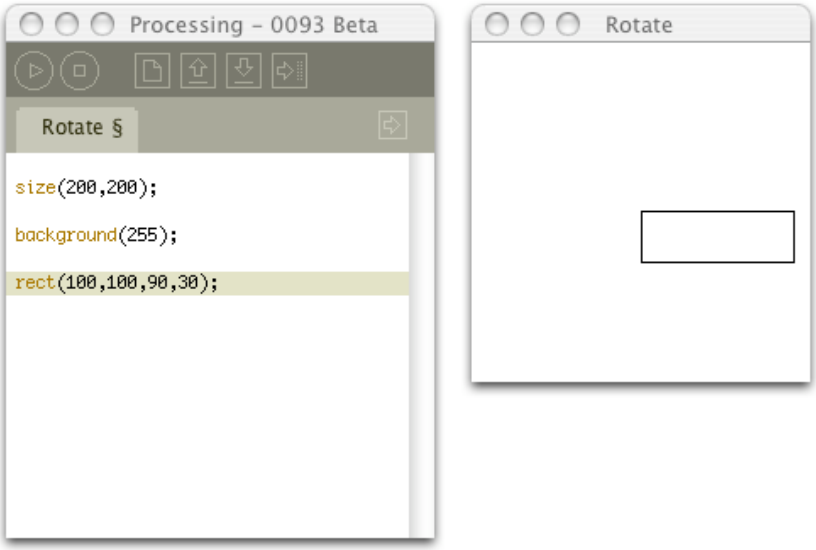
Un des trucs souvent utilisés dans les dessins algorithmiques actuels consiste à dessiner beaucoup de lignes avec de la transparence.



3.7 Rectangles : rect(x,y,l,h);

La méthode **rect** dessine un rectangle

```
size(200,200);

background(255);

rect(100,100,90,30);
```



Contrairement à la méthode **line**, **rect** ne dessine pas un rectangle à partir d'un point x1, y1 vers un autre point x2, y2. La méthode **rect(x,y,l,h)** dessine un rectangle à partir d'un point x, y, de largeur l et de hauteur h. Dans notre exemple, nous dessinons à partir du point 100,100, c'est-à-dire le milieu de notre cadre. Et à partir de ce point, nous avons dessiné un rectangle de 90 pixels de largeur et 30 pixels de hauteur.

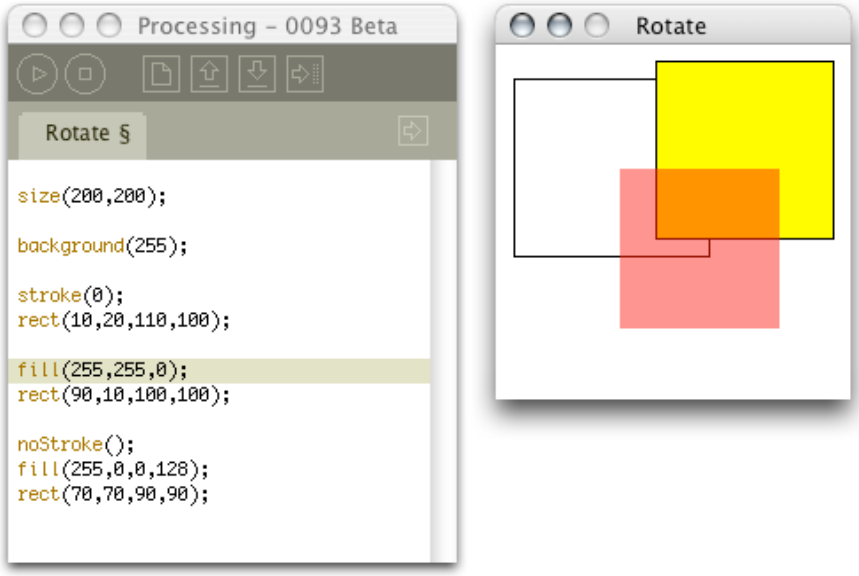
On peut remarquer que le rectangle est vide à l'intérieur.

3.8 Le remplissage : fill(…);

Processing, comme beaucoup d'environnement de programmation fait une distinction entre le trait qui fait le tour d'une forme, et la couleur qui remplit l'intérieur.

La méthode **fill();** colorie l'intérieur des formes. Elle fonctionne comme la méthode **stroke();** (on peut utiliser 1, 3, ou 4 paramètres), Voici un exemple pour montrer les différentes possibilités d'interaction entre **stroke()** et **fill()** :

```
size(200,200);
background(255);

stroke(0);
rect(10,20,110,100);

fill(255,255,0);
rect(90,10,100,100);

noStroke();
fill(255,0,0,128);
rect(70,70,90,90);
```



Processing fait donc une différence entre les bords d'une forme et son intérieur. Notez également que l'on peut dessiner un rectangle sans bord :

```
noStroke();
fill(255,0,0);
rect(10,10,80,80);
```

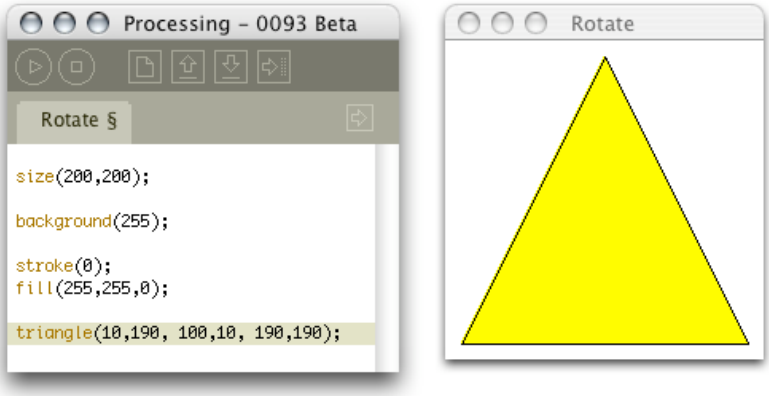
On peut aussi dessiner un rectangle avec l'intérieur transparent:

```
noFill();
stroke(255,0,0);
rect(10,10,80,80);
```

3.9 Les triangles : triangle(x1,y1, x2,y2, x3,y3);

Dessiner un triangle est plus logique que dessiner un rectangle ; il n'a pas besoin de paramètres de largeur ni de hauteur, il suffit de donner les coordonnées de trois points : x1,y1, x2,y2, x3,y3.

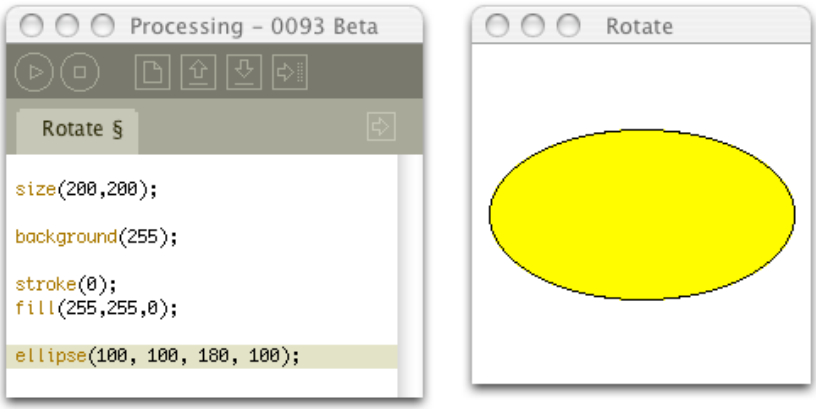
```
size(200,200);

background(255);

stroke(0);
fill(255,255,0);

triangle(10,190, 100,10, 190,190);
```



3.10 Les cercles : ellipse(x,y,l,h);

Vous n'êtes pas condamné à ne dessiner que des formes droites dans Processing. Vous pouvez dessiner des cercles, ou plus précisément des ellipses. Une ellipse est dessinée de la même façon qu'un rectangle – un point x, y suivi de deux valeurs pour la largeur et la hauteur. Attention, contrairement à la méthode **rect()**, l'ellipse est centrée sur le point x,y.

```
size(200,200);
background(255);
stroke(0);
fill(255,255,0);
ellipse(100, 100, 180, 100);
```



3.11 Les points : point(x,y);

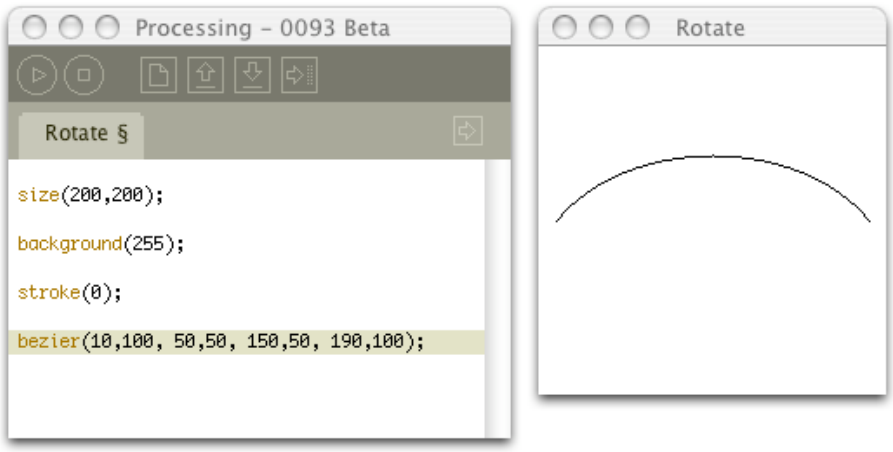
La méthode **point(x,y);** permet de dessiner la forme la plus fondamentale sur un écran d'ordinateur : un pixel.

```
stroke(255,0,0);
point(100, 100);
```

3.12 Les courbes : bezier(…);

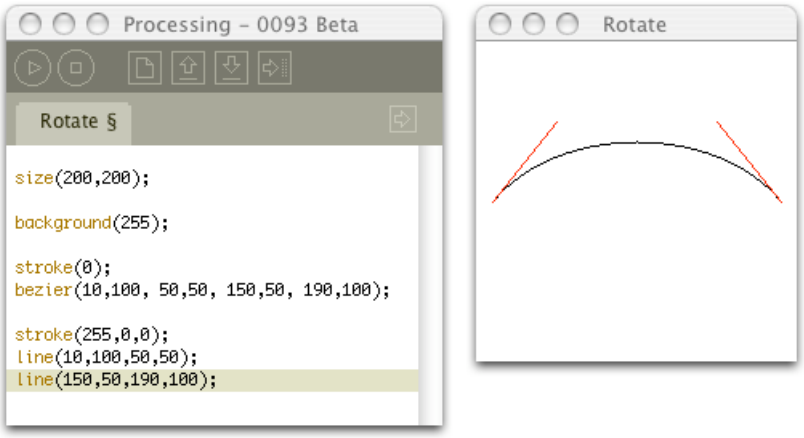
Il y a beaucoup d'autres possibilités, mais pour terminer ce rapide tour d'horizon des fonctions de dessin, regardons comment dessiner une courbe de Bézier. Pour dessiner une courbe, il faut définir quatre points x,y. Le premier et le dernier point définissent le début et la fin de la courbe. Les deux points intermédiaires définissent les tangentes c'est à dire la direction vers laquelle la courbe doit partir pour rejoindre l'autre point.

```
size(200,200);
background(255);
stroke(0);

bezier(10,100, 50,50, 150,50, 190,100);
```



Pour donner plus de lisibilité à notre courbe de Bézier, ajoutons deux lignes rouges tangentes à la courbe : la première allant du début de la courbe jusqu'au premier point de contrôle, la deuxième ligne allant du deuxième point de contrôle jusqu'à la fin de la courbe.

```
size(200,200) ;
background(255) ;
stroke(0) ;

bezier(10,100, 50,50, 150,50, 190,100) ;
stroke(255,0,0) ;
line(10,100,50,50) ;
line(150,50,190,100) ;
```



3.13 Exercice

Dessinez quelque chose !

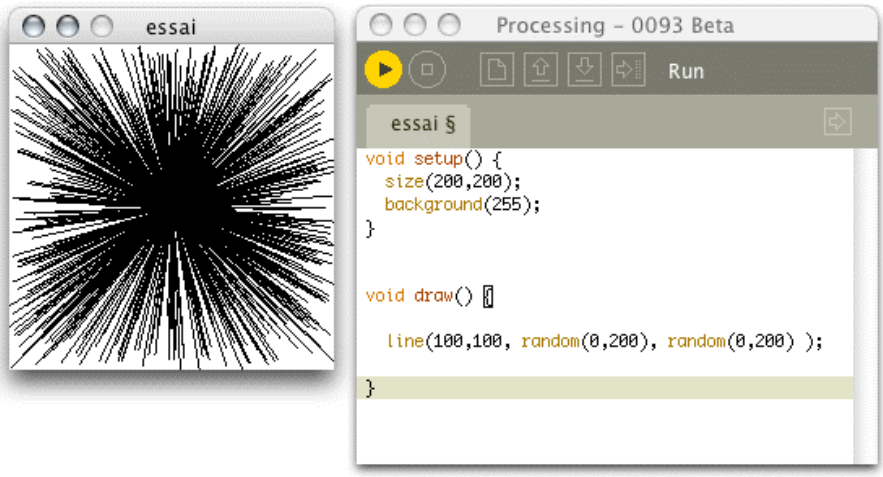
4. Le mouvement

Dessiner quelques lignes, c'est chouette, mais on nous a parlé d'interactivité, de mouvement, de dessins génératifs. Alors, comment ajouter le mouvement ?

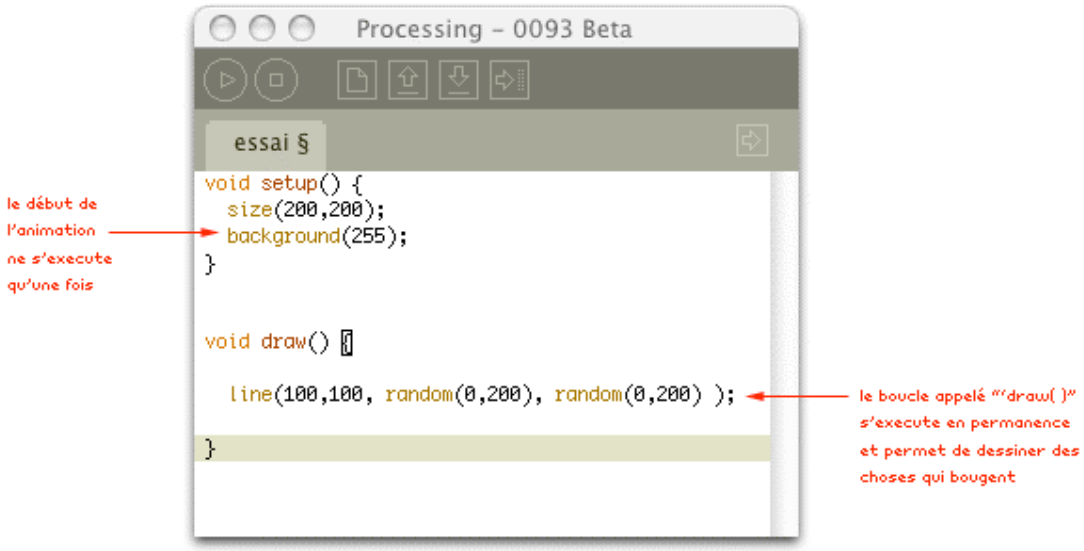
4.1 Animation

Tapons le programme suivant dans la fenêtre Processing :

```
void setup() {
 size(200,200);
 background(255);
}

void draw() {
 line(100,100, random(0,200), random(0,200) );
}
```



Nous avons remplacé notre code écrit ligne par ligne par un autre genre de programme. Il contient de nouveaux mots : **void setup() {…}** et **void draw() {…}.**
Ces deux termes correspondent à l'initialisation de l'animation (= ce qui ne s'exécute qu'une fois) et au mouvement de l'animation (= ce qui s'exécute en permanence).



Au début de l'animation, nous déterminons la taille de la fenêtre de l'animation et sa couleur de fond (255 = blanc). Ces deux instructions sont incluses dans les accolades qui déterminent ce qui doit être fait au début de l'animation, c'est-à-dire quand nous appuyons sur le bouton **lecture**. Par contre, ce qui doit se dessiner en permanence, c'est-à-dire nos lignes qui se dessineront de façon aléatoire du centre de l'écran vers le reste de la fenêtre, est inclus à l'intérieur des accolades du **draw()**. En résumé : **setup()** détermine ce qui doit être fait quand on appuie sur le bouton "lecture", alors que **draw()** détermine ce qui se passe par la suite, et jusqu'à ce qu'on arrête l'animation.

4.2 L'aléatoire : random(min,max);

La méthode **random(min,max)** permet de choisir un nombre au hasard entre min et max. Dans l'exemple précédent, le résultat de **random** est placé dans le troisième et le quatrième argument de **line()**. Chaque fois que le programme joue la méthode **draw()** (il le fait en permanence), il choisira deux nouveaux nombres entre 0 et 200 qui détermineront le x et le y de la fin de la ligne.

Au fait, chaque fois que vous voulez remplacer un chiffre fixe, comme le chiffre 50, 99, 666, ou 13 par un chiffre choisi au hasard, remplacez ce chiffre dans votre programme avec l'expression **random({valeur minimum},{valeur maximum});**. Par exemple, on pourrait faire un programme très simple, comme celui-ci, où nous ne changeons que le fond de l'écran :

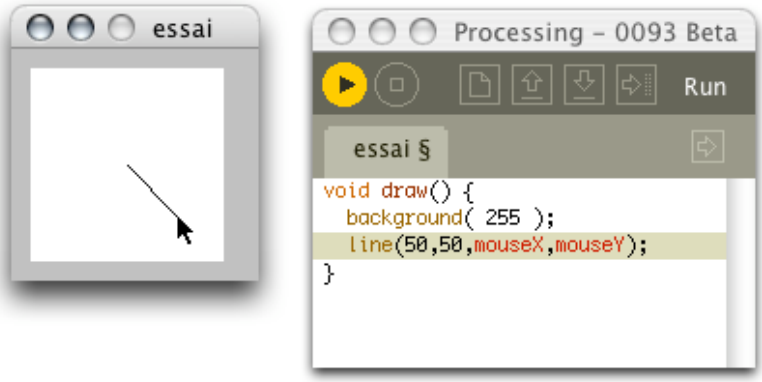
Il s'agit d'un programme qui choisit au hasard une nouvelle couleur de fond entre noir et blanc.

4.3 La souris : mouseX et mouseY

Essayons cet exemple

```
void draw() {
 background( 255 );
 line(50,50,mouseX,mouseY);
}
```

Nous avons maintenant une ligne qui suit la position de votre souris à l'intérieur de la fenêtre :



Comme pour l'animation aléatoire, nous avons placé le dessin de notre ligne à l'intérieur d'une méthode d'animation qui dessine en permanence. Nous avons ensuite remplacé les expressions **random()** par deux nouvelles expressions **mouseX** et **mouseY** (attention de toujours respecter les majuscules et minuscules). Comme vous l'avez peut-être deviné, il s'agit de mots qui seront remplacés pendant la lecture de l'animation par les valeurs actuelles des positions horizontales et verticales de la souris.

Notez également que notre dessin ne fait pas autant de lignes qu'avant. Le fond de la scène reste clair. C'est parce que **background(255);** est placé à l'intérieur de la méthode de dessin **draw().** A peu près 30 fois par seconde, l'animation remplit la fenêtre de blancs, puis dessine une ligne qui va de **50,50** vers les positions **mouseX, mouseY** de la souris. Cela revient à nettoyer l'ancien dessin avant d'en dessiner un nouveau.

4.4 Exercice

Faites un dessin qui dessine en permanence une forme quelconque influencée par les positions de la souris et qui change de couleur et de transparence.

5. Les méthodes

Il y a quatre techniques fondamentales utilisées dans presque tous les langages de programmation. Elles sont :

- Les méthodes
- Les variables
- Les boucles
- Les bifurcations

Pour ceux qui n'ont jamais programmé ni vu un programme, sachez que c'est la combinaison de ces 4 techniques qui constitue l'art de la programmation. Presque sans limite, on peut empiler et emboîter ces 4 techniques les unes après les autres, les unes dans les autres, et construire une infinité de résultats.

Les méthodes, aussi connues sous le nom de fonction, procédure ou routine, est le premier concept à maîtriser.

5.1 Définition

Une méthode est une liste d'instructions à appliquer chaque fois que le nom de la méthode est invoqué. Dans un langage plus simple, on peut dire que les méthodes sont des mots magiques : chaque fois que l'ordinateur rencontrera ce mot dans notre programme, il ira faire l'action décrite par ce mot magique. Cela permet de faire toute une série de choses en ne tapant qu'un seul mot.

Nous avons déjà vu un certain nombre de méthodes dans Processing. Par exemple, quand vous tapez :

```
line(20,30,70,10);
```

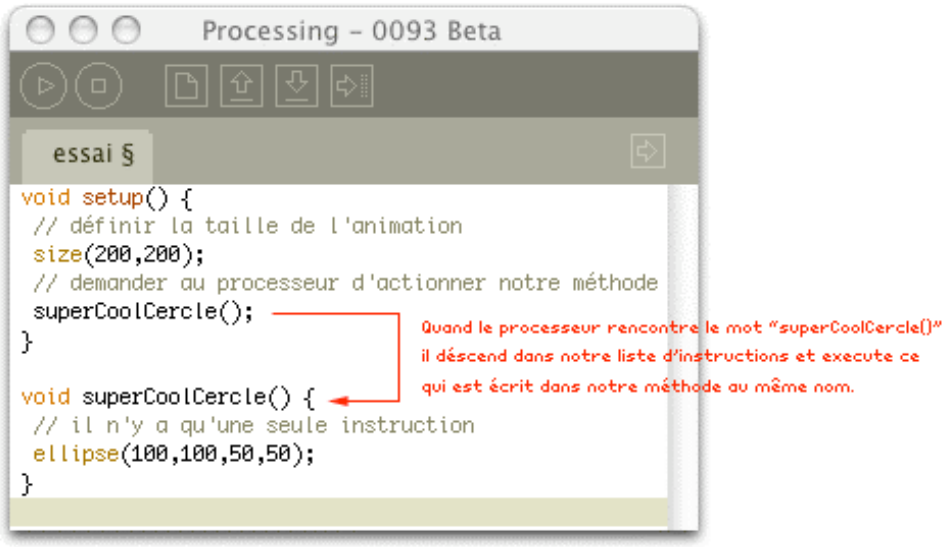
vous êtes en réalité en train de faire appel à une méthode préexistante, écrite pour vous par les concepteurs de Processing pour vous faciliter la tâche.

Une autre méthode que nous avons vue s'appelle **println();.** Cette méthode permet d'afficher en bas de votre fenêtre de programmation des informations sur votre programme.

Au lieu de faire appel à des méthodes préexistantes, nous pouvons créer des méthodes nous-mêmes et les appeler dans nos programmes. Voici un exemple :

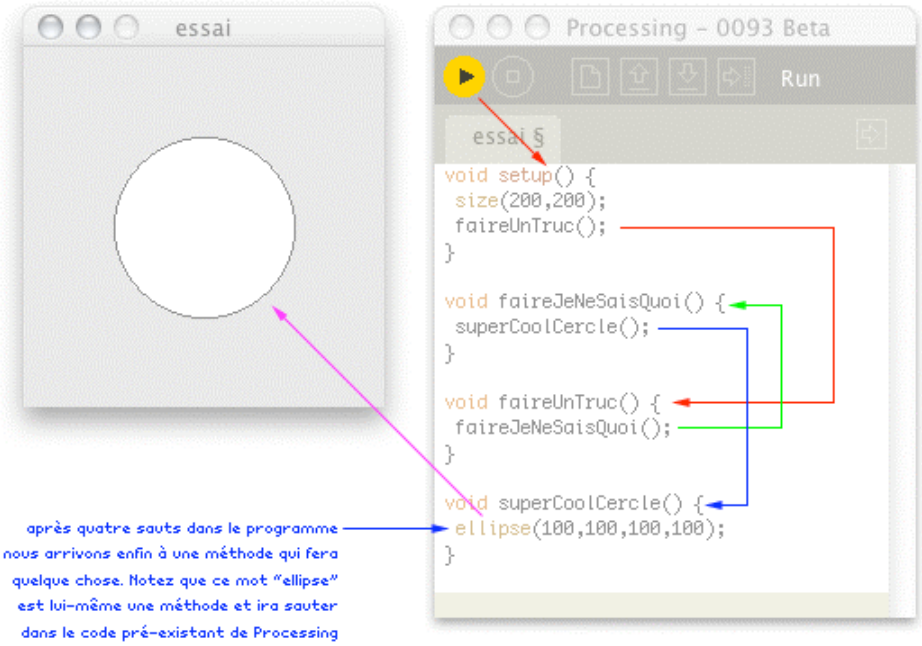
```
void setup() {
 // définir la taille de l'animation
 size(200,200);
 // demander au processeur d'appeler notre méthode
 superCoolCercle();
}

void superCoolCercle() {
 // pour l'instant il n'y a qu'une seule instruction dans notre méthode
 ellipse(100,100,50,50);
}
```

Nous venons d'inventer une méthode, que nous avons nommé **superCoolCercle**. Chaque fois que le processeur rencontre ce nom, il exécutera les instructions contenues dans notre méthode. Si on écrit 3 fois notre méthode, le processeur l'exécutera 3 fois.



Voici notre programme avec quelques méthodes en plus :

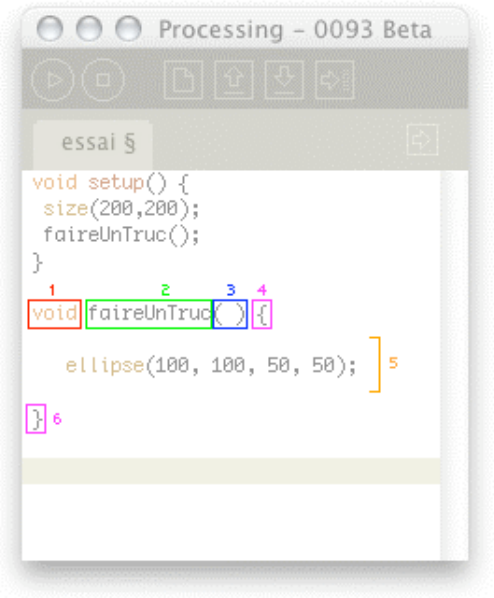
```
void setup() {
 size(200,200);
 faireUnTruc();
}

void faireJeNeSaisQuoi() {
 superCoolCercle();
}

void faireUnTruc() {
 faireJeNeSaisQuoi();
}

void superCoolCercle() {
 ellipse(100,100,100,100);
}
```



Il n'est pas important que les méthodes suivent l'ordre dans lequel elles seront activées dans notre programme.

Pour fabriquer nos propres méthodes, il faut juste avoir :

void « le nom de la méthode » des parenthèses (pour l'instant vides)
  une accolade ouvrante
  les instructions de notre méthode
  une accolade fermante



Nous pouvons créer autant de méthodes que nous voulons, ce qui plus tard nous permettra de faire des programmes complexes à partir de petites méthodes qui seront comme des modules simplifiés, emboîtés les uns dans les autres. On construira un programme de façon plus explicite si on utilise des noms simples pour les méthodes.

On peut aussi remarquer que **setup()** a la structure d'une méthode.

Au fait, il existe un certain nombre de méthodes que Processing exécutera à votre place : vous n'avez pas besoin de les appeler pour qu'elles soient exécutées. On pourrait dire qu'elles sont des méthodes automatiques qui sont appelées automatiquement à votre place. **setup()** et **draw()** sont dans ce cas.

```
void setup() {
 // choisir une couleur de fond au hasard
 background( random(0,255), random(0,255), random(0,255) );
}
```

C'est comme si nous avions appelé la méthode **setup()** directement en appuyant sur le bouton **lecture** et donc que le bouton **lecture** contenait le code **setup();**

Ensuite, on peut écrire les mêmes instructions à l'intérieur d'une autre méthode automatique, ce qui donnera évidemment un autre comportement :

```
void draw() {
 // choisir une couleur de fond au hasard
 background( random(0,255), random(0,255), random(0,255) );
}
```

Et puis, pour terminer, prenons une méthode automatique très pratique : il s'agit d'une méthode que le processeur appellera automatiquement chaque fois que la souris change de position...

```
void mouseMoved() {
 // choisir une couleur de fond au hasard
 background( random(0,255), random(0,255), random(0,255) );
}
```

5.2 Résumé

Une méthode contient une liste d'actions à exécuter chaque fois qu'elle est appelée. Nous pouvons créer nos propres méthodes et les appeler dans notre code en écrivant tout simplement leur nom. Il existe aussi des méthodes automatiques, qui seront appelées par le processeur au démarrage de l'animation, par exemple, ou quand l'utilisateur bougera sa souris. Ces méthodes sont décrites de la même manière que nos méthodes que nous avons écrites nous-mêmes mais nous laisserons à la machine le soin de les appeler.

5.3 Exercice

Essayer de modifier un de vos anciens programmes, en remplaçant une partie du programme à l'intérieur d'une méthode avec le nom de votre choix.

6. Les variables

Le deuxième concept fondamental de la programmation est celui de la **variable**.

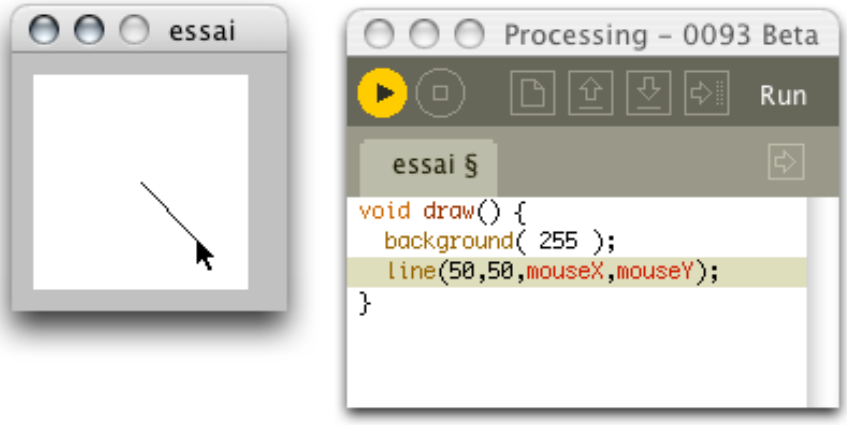
6.1 Définition

Une **variable** est une donnée, elle contient de l'information. Autrement dit, une variable peut contenir une valeur quelconque (5, 13, 666, "Hello World!", 3.145, etc.). Cette variable est identifiée par un nom sans espaces (x, y, premiereCouleur, deuxiemeCouleur, etc). Chaque fois que vous utilisez le nom d'une variable, vous utiliserez la valeur qu'elle contient.

6.2 Quelques exemples de variables

Nous utilisons tous les jours des variables sur un ordinateur sans forcément le savoir. La position de la fenêtre de votre navigateur. Le texte que cette page contient est également une variable : une longue variable qui contient tous les caractères du texte. Et chacune des images sur cette page est également contenue à l'intérieur de variables. Des variables sont partout dans les programmes informatiques.

Comme dans notre concept précédent, nous avons déjà eu affaire à une variable : **mouseX** et **mouseY**. Chaque fois que nous tapons l'expression **mouseX** dans un programme, le processeur va chercher la valeur actuelle de la position horizontale de la souris et utilisera cette valeur à la place du mot **mouseX**. Là où nous voyons un mot, l'ordinateur voit une valeur.

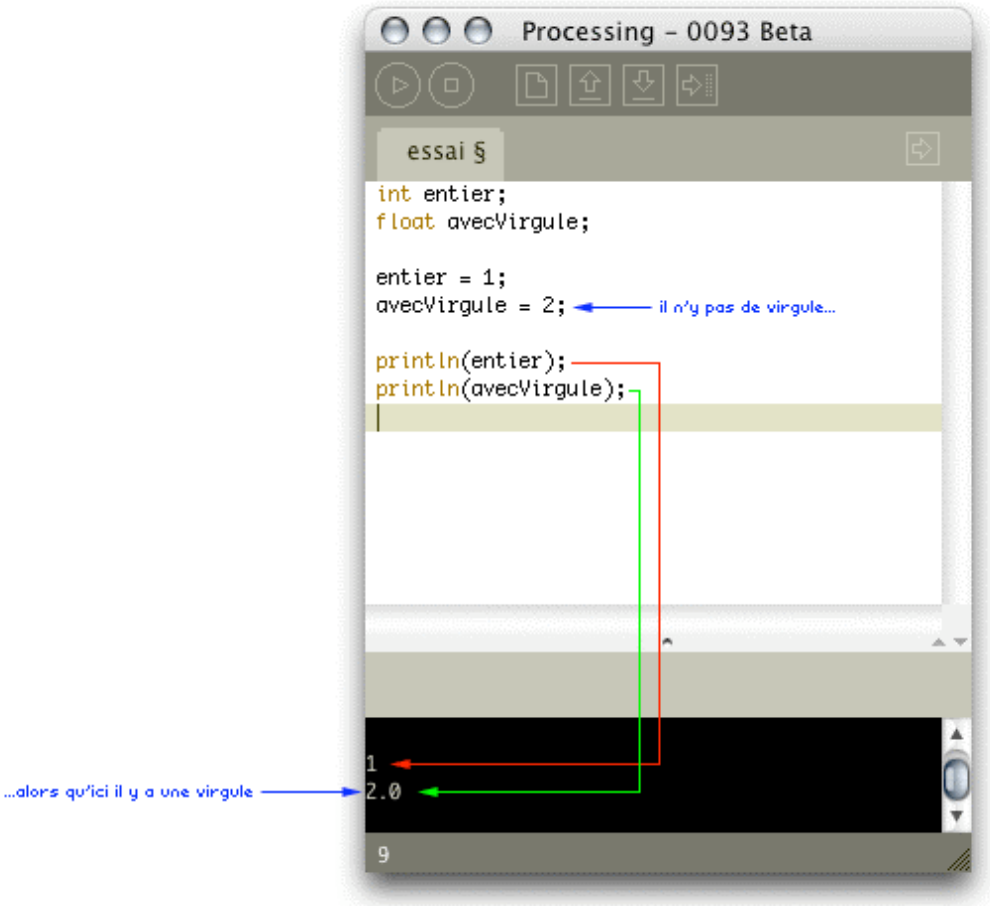


Dans cet exemple du chapitre précédent, nous voyons qu'au lieu de taper l'expression **line(50,50,150,175),** ou quelque chose dans le genre, nous avons remplacé les deux nombres fixes (150 et 175) par des variables (**mouseX** et **mouseY**). Pour créer une variable, il faut d'abord la définir. Lors de cette définition, il faut déclarer le ***type*** de votre variable. Pour des raisons techniques, un ordinateur a besoin de savoir combien de place chaque variable prendra, et c'est à la définition de notre variable que nous allons lui dire combien de place il lui faudra. D'habitude votre programme contiendra deux *types* de variables : des nombres *entiers* et des nombres *avec virgule*. Les nombres entiers sont appelés *integers* en anglais, ce qui s'écrit dans Processing avec l'expression **int**. Les nombres avec virgule sont appelés *floating-point numbers*, ou **float** dans le langage de Processing.

Commençons par un exemple. Voici le code qu'il faut écrire dans Processing :

```
int entier;
float avecVirgule;

entier = 1;
avecVirgule = 2;

println(entier);
println(avecVirgule);
```

Et son résultat lorsque vous appuyez sur le bouton **lecture** :



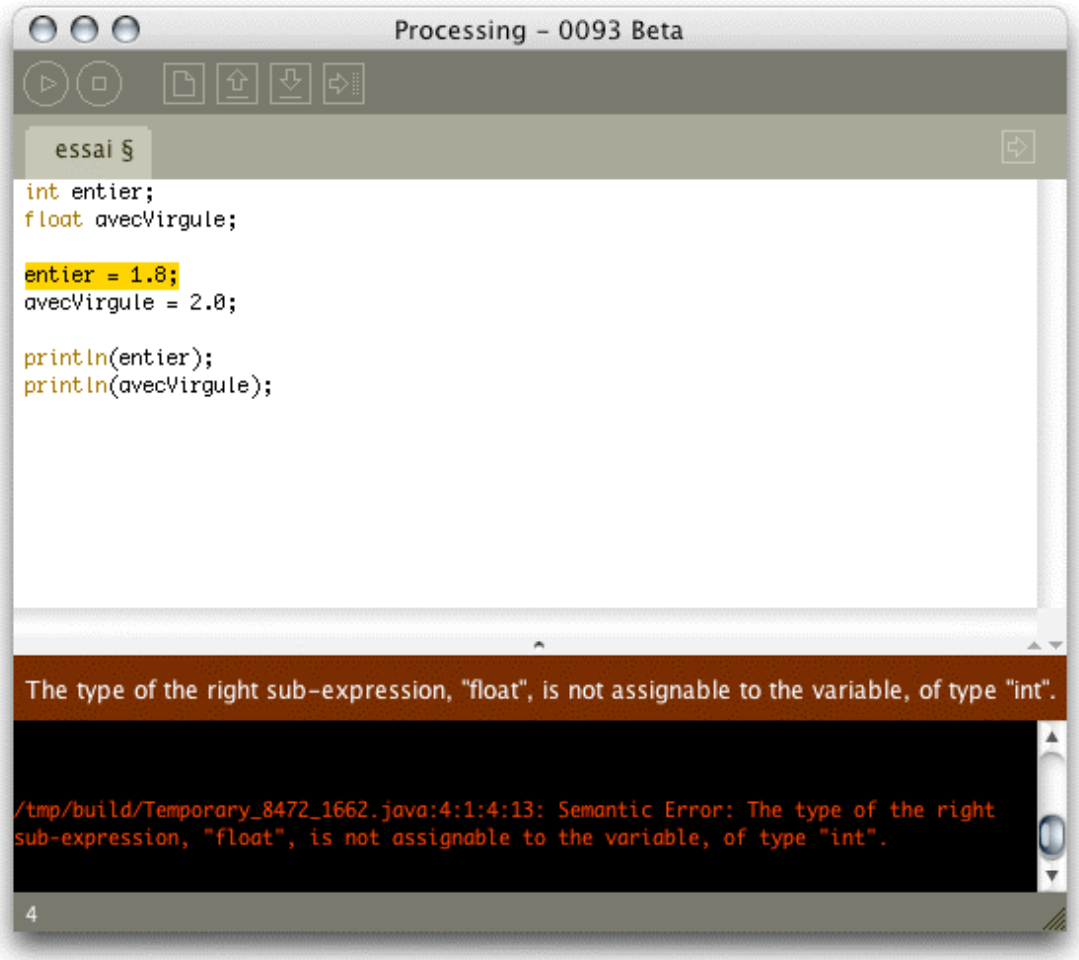
Notez d'abord que nous avons largement simplifié notre exemple en enlevant toutes les méthodes de notre programme : les expressions sont traitées les unes après les autres, ce qui simplifie la compréhension du programme. Ensuite, notez que nous avons défini deux variables : la première de type `int` et la seconde de type `float`. Nous avons ensuite affecté deux valeurs à ces variables :

```
entier = 1;
```

et `avecVirgule = 2;`

Mais quand nous appuyons sur **lecture**, nous constatons qu'une des variables a changé de valeur : au lieu d'afficher la valeur de départ 2, Processing l'a transformée en 2.0.

En plus, si nous essayons d'aller dans l'autre sens, à savoir donner une valeur de *type float* à une variable de *type int*, Processing nous déclare carrément une erreur :



Visiblement Processing n'est pas très content. Mais heureusement, Processing fait l'effort de nous décrire le problème : en anglais, il indique que la valeur à droite du signe **=** est de type **float** et ne peut pas être affectée à la variable à gauche, de type **int**.
Apparemment, le processeur arrive assez facilement à ajouter un **.0** à une valeur de type **int**, mais n'arrive pas à forcer une valeur de type **float** dans une variable de type **int**. Si on voulait faire cela, il faudrait en réalité écrire le programme suivant :

```
int entier;
float avecVirgule;

entier = (int) 1.8;
avecVirgule = 2.0;

println(entier);
println(avecVirgule);
```

Ce qui donnera le résultat suivant :

En écrivant **(int)** avant notre valeur de type **float**, nous forçons la valeur 1.8 à enlever la valeur après la virgule, et à ne garder que la valeur 1 d'avant la virgule. Cette transformation peut être très pratique par la suite.

6.3 Différentes utilisations des variables

Nous venons de voir que les variables doivent être définies dès le départ avec leur type. Regardons maintenant comment on se sert de ces variables dans nos programmes.

On peut utiliser les variables à beaucoup d'endroits différents dans un programme, mais pour l'instant on va parler de trois types d'usages :

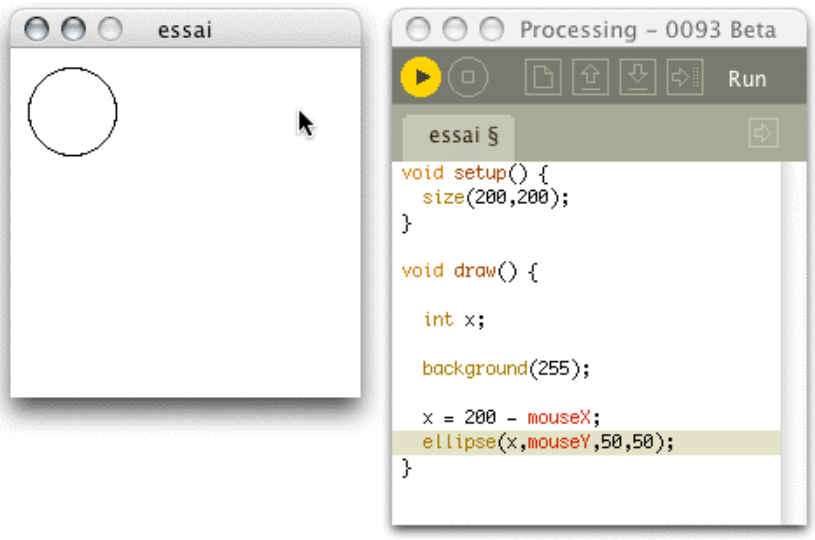
- des variables à l'intérieur d'une méthode = variables locales
- des variables utilisables de façon globale à l'intérieur de toutes les méthodes en même temps = variables globales
- des variables qui sont "passées" à une méthode (à l'intérieur de ses parenthèses = paramètres)

6.3.1 Variables locales

Dans l'exemple suivant, nous allons définir une variable qui va récupérer la valeur de la souris, enlèvera cette valeur de la valeur 200, pour ensuite l'appliquer dans le positionnement d'une ellipse :

```
void setup() {
 size(200,200);
}
```

```
void draw() {
 int x;
 background(255);
 x = 200 - mouseX;
 ellipse(x,mouseY,50,50);
}
```

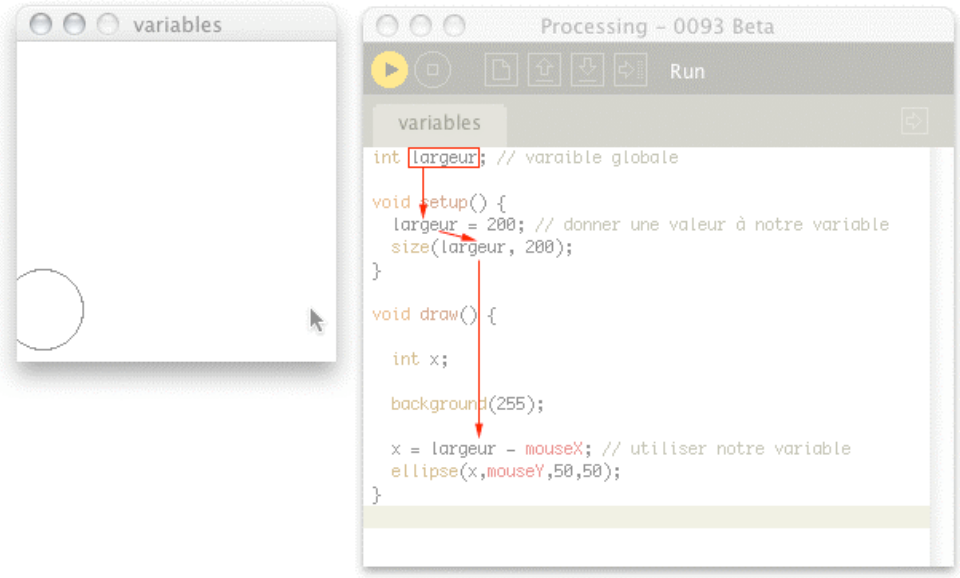


Dans cet exemple, nous voyons qu'à chaque exécution de la méthode **draw()** le processeur définit une nouvelle variable de *type* **int**, donne une nouvelle valeur à cette variable **(200 - mouseX)**, puis utilise cette variable dans le positionnement de notre ellipse. A la fin de chaque exécution de la méthode **draw()**, notre variable disparaîtra de la mémoire de l'ordinateur. Notez également que nous avons utilisé directement la valeur de la variable **mouseY**, sans la modifier avec une variable intermédiaire. C'est pour cette raison que l'ellipse suit la position verticale, mais s'inverse sur le plan horizontal.

6.3.2 Variables globales

Si nous voulons que notre variable soit utilisable pour tout le programme, et qu'elle ne disparaisse pas entre chaque exécution de la méthode **draw()**, nous allons devoir la déclarer comme une variable **globale**, c'est-à-dire en début du programme avant toutes les méthodes :

```
int largeur; // variable globale

void setup() {
 largeur = 200; // donner une valeur à notre variable
 size(largeur, 200);
}

void draw() {

 int x;

 background(255);

 x = largeur - mouseX; // utiliser notre variable
 ellipse(x,mouseY,50,50);
}
```



Une fois que notre variable est définie en début du programme, elle devient accessible à toutes les méthodes contenues dans ce programme. Ces méthodes peuvent lui donner de nouvelles valeurs, elles peuvent faire des additions, soustractions, multiplications, divisions, etc. La valeur sera respectée entre une méthode et une autre, parce que la variable fait maintenant partie de la mémoire de l'ordinateur jusqu'à ce qu'on arrête le programme.

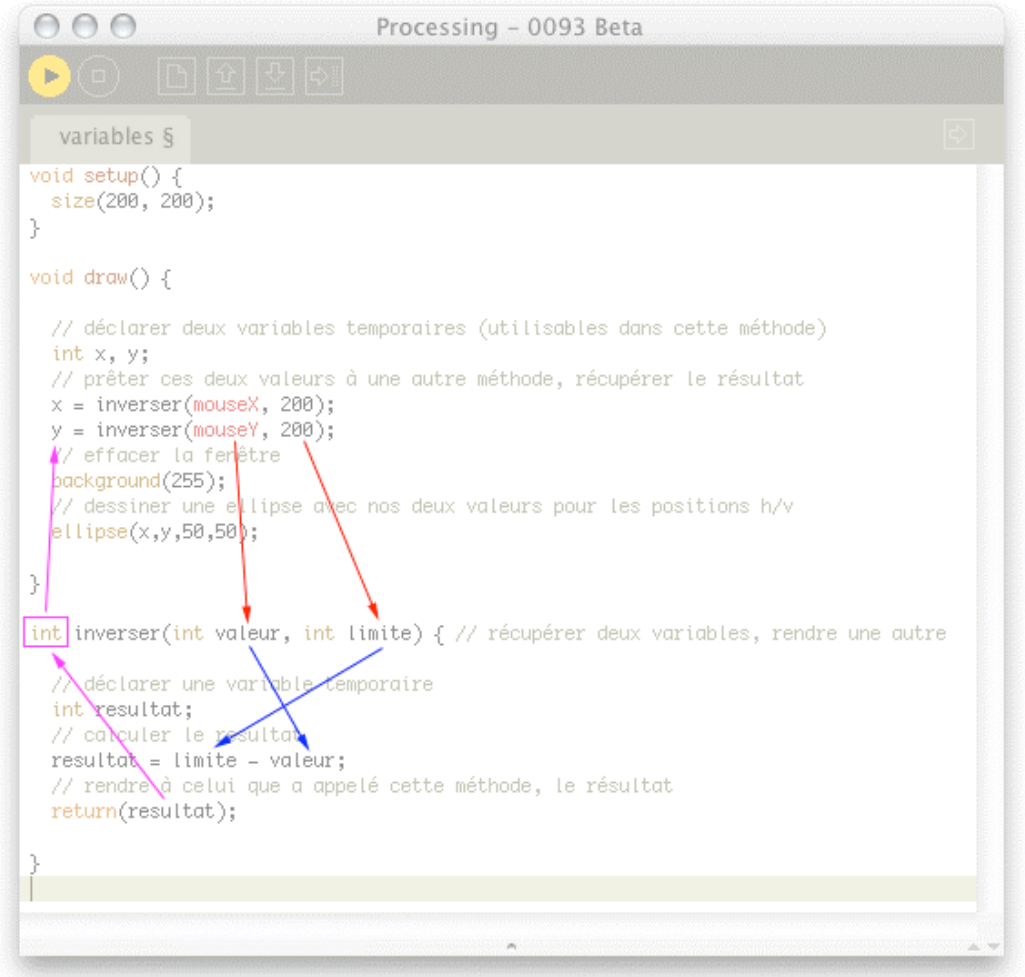
6.3.3 Variables passées comme paramètre aux méthodes

Nous allons maintenant terminer notre tour rapide de l'usage des variables, avec sa forme la plus complexe (mais très puissante).

Avez-vous remarqué qu'avec chaque méthode, nous avons deux signes étranges : le **void** et les parenthèses vides **()**. Savez-vous à quoi servent ces deux éléments ? Lisez le programme suivant et essayez de comprendre ce qui se passe (nous avons ajouté quelques commentaires pour vous aider) :

```
void setup() {
 size(200, 200);
}

void draw() {
 // déclarer 2 variables temporaires (utilisables dans cette méthode)
 int x, y;
 // prêter ces 2 valeurs à une autre méthode, récupérer le résultat
 x = inverser(mouseX, 200);
 y = inverser(mouseY, 200);
 // effacer la fenêtre
 background(255);
 // dessiner une ellipse avec nos 2 valeurs pour les positions h et v
 ellipse(x,y,50,50);
}

int inverser(int valeur, int limite) { // récupérer 2 variables, rendre une autre
 // déclarer une variable temporaire
 int resultat;
 // calculer le resultat
 resultat = limite - valeur;
 // rendre le résultat au programme qui a appelé cette méthode
 return(resultat);
}
```

Ce programme est complexe, mais il a l'avantage de nous expliquer l'histoire du **void** et les parenthèses vides **()**.
On pourrait presque dire qu'une méthode est une sorte d'usine. Elle peut prendre des données, traiter ces données, et donner le résultat en retour. C'est un principe très puissant de la programmation : la possibilité de prendre beaucoup de choses complexes et de les enfermer dans une boîte noire qui nous cache toute cette complexité. Par exemple, nous ne connaissons pas tous les détails sur : comment Processing arrive à dessiner une ligne sur un écran ? Ce que l'on veut juste c'est : lui donner les points de notre ligne et qu'il nous la dessine. C'est pareil pour n'importe quel programme : les méthodes sont des assistants, des boîtes noires, bref des usines où on peut traiter des choses complexes de façon simple, avec un mot magique suivi de quelques paramètres.
Mais pour avoir cette usine, il faut avoir un moyen de passer à l'usine des valeurs, des paramètres, c'est-à-dire des spécificités, comment on veut dessiner la ligne ou colorier le carré. Il faut aussi avoir un moyen — comme dans le cas de notre code d'inversement — pour que notre usine traite nos paramètres et nous donne un résultat.
Dans les deux cas : l'entrée des paramètres dans la boîte noire et la récupération du résultat, il nous faut une variable. Et comme pour n'importe quelle variable, il faut donner le *type* de la variable, à la fois pour les paramètres et pour la valeur de retour :



Quand on appelle la méthode **inverser(),** on lui "passe" 2 valeurs : **mouseY** et **200**. La méthode prend ces deux valeurs et les attribue aux 2 variables respectives : « valeur » et « limite ». A l'intérieur de la méthode, elle se sert de ces valeurs comme n'importe quelle variable. Par exemple, ici, elle soustrait ce que vaut la variable « valeur » de ce que vaut la variable « limite ». Ensuite, le résultat de cette soustraction (la variable « resultat ») est repassé à la ligne qui a appelé la méthode, et elle est utilisée pour donner une valeur à la variable **y**.

Dans notre exemple, notre "usine" s'appelle **inverser()** et donne en retour un résultat de type **int**. C'est pour cette raison que nous écrivons **int inverser()** : parce que la méthode donne en retour une valeur de ce type. Par contre, dans les exemples précédents, comme par exemple pour la méthode **faireUnTruc()**, nous n'avions pas besoin d'une valeur de retour, d'où le mot **void** juste devant la méthode : **void faireUnTruc() {…}**. Les mots **int, float** ou **void** explique le type de valeur de retour d'une méthode. **void** signifie qu'il n'y a pas de valeur à retourner.

6.4 Résumé

Nous avons vu qu'une variable sert à créer une donnée (une valeur quelconque) et à pouvoir se servir de cette donnée dans notre programme. Nous avons vu que les variables peuvent :

- venir de **Processing** (comme dans le cas de **mouseX**)
- être créées à l'intérieur des méthodes
- être définies en début du programme (variables globales)
- être "passées" comme *paramètre* à l'intérieur de la définition d'une méthode.

Nous avons également vu que chaque variable doit être définie par son type (nous n'avons vu, pour l'instant, que les types **int** et **float**).

Il est intéressant de noter que nous avons déjà commencé à mélanger les deux premiers concepts de programmation : **méthode** et **variable**, surtout dans le cas de passage des variables à l'intérieur des parenthèses d'une méthode.

6.5 Exercice

Prenez un de vos programmes existant, mais cette fois-ci ajouter quelques variables supplémentaires.

7. Les boucles

Le troisième concept fondamental de la programmation est le concept de boucles.

7.1 Définition

Une **boucle** est une liste d'instructions qui est exécutée de façon répétitive. Ce code s'exécute en boucle jusqu'à ce qu'une condition soit vérifiée.

7.2 Economie de code

Imaginons que vous voulez dessiner un rectangle qui changerait de couleur de façon progressive, allant du blanc au noir. Comment faire ?

Vous pourriez essayer la méthode suivante, en dessinant des centaines de lignes, les unes après les autres, à la main :

```
size(300,300);
background(255);

stroke(0);
line(10,10,290,10);

stroke(1);
line(10,11,290,11);

stroke(2);
line(10,12,290,12);

stroke(3);
line(10,13,290,13);

// continuer comme ça pendant une heure

stroke(254);
line(10,264,290,264);

stroke(255);
line(10,265,290,265);
```

Au début, il est normal de se laisser prendre à ce jeu-là. On peut facilement perdre une heure pour faire quelque chose qui, ironiquement, sera exécuté par l'ordinateur en moins d'une milliseconde. C'est donc pour cela que la boucle a été inventée. Pour transformer ce code en boucle, il faudra définir :

1. un compteur (une variable) qui comptera jusqu'à une valeur définie
2. une liste de tâches que le processeur doit effectuer en boucle à chaque valeur du compteur.

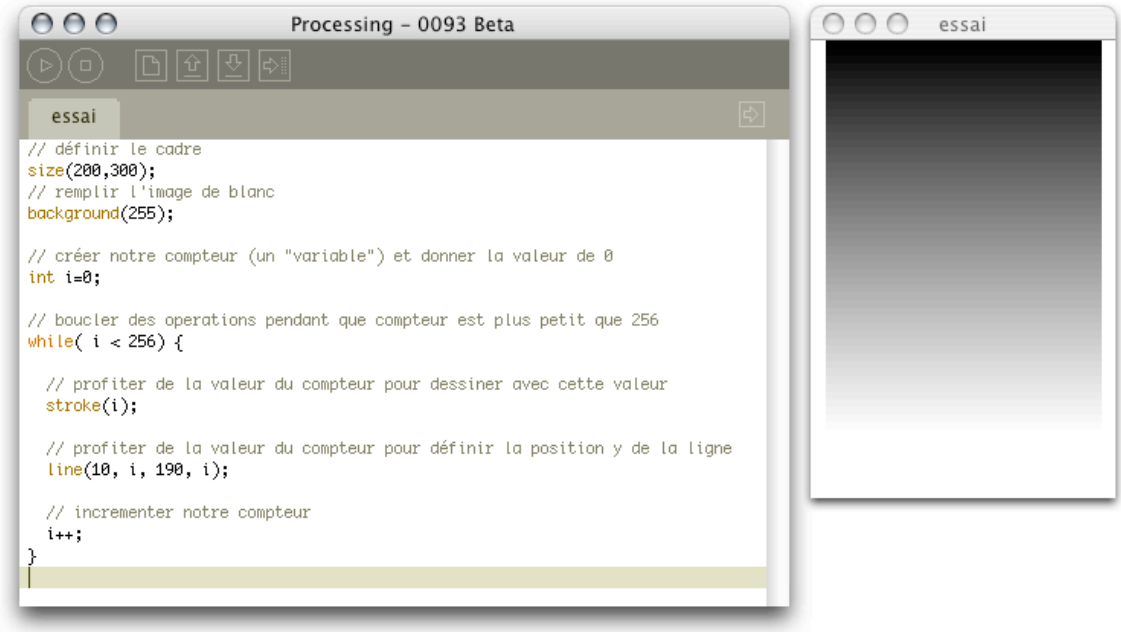
7.3 La boucle simple : while()

Une première sorte de boucle utilise le mot clé **while()**

Exemple :

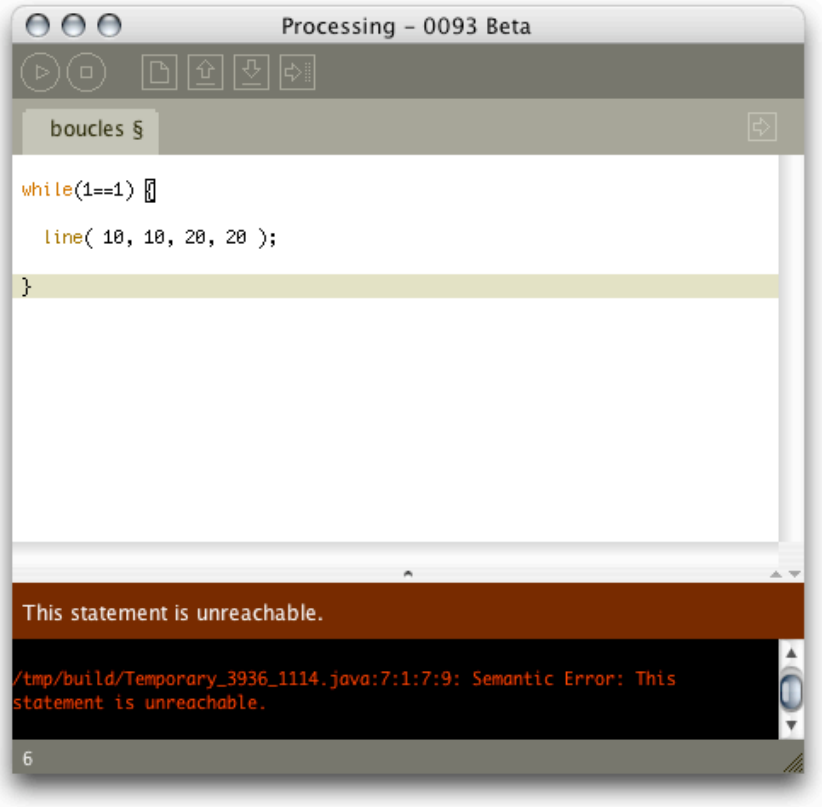
```
size(200,300);
background(255);

// définir le compteur
int i=0;

// la boucle
while( i < 256) { // exécuter tant que 'i' est plus petit de 256
 // utiliser le compteur pour changer la couleur
 stroke(i);
 // dessiner une ligne
 line(10, i, 190, i);
 // augmenter la variable de 1
 i++;
}
```

Ce code donnera le résultat suivant:



La boucle **while(){…}** tourne en rond tant que ce qui est entre parenthèses reste vrai. Dans notre exemple, dès que la variable **i** n'est plus inférieure à **256**, l'ordinateur sort de la boucle et exécute les lignes qui suivent.

La boucle **while(){…}** est très pratique, mais peut aussi être dangereuse. Si vous ne faites pas attention, votre boucle peut enfermer votre programme dans ce qu'on appelle une **boucle infinie**. C'est le cas dans le code suivant car la valeur 1 est toujours égale à 1. Et tant que la valeur 1 est égale à 1, la boucle va boucler.



Quand nous exécutons ce programme, Processing affiche une erreur parce qu'il n'arrive pas à "accéder" à la méthode. C'est une autre façon de dire que Processing est enfermé dans une boucle infinie dont il n'arrive pas à s'extraire, et que du coup, il ne lui reste plus de ressources (de temps) pour dessiner notre ligne.

Notez également qu'au lieu d'écrire **1 = 1,** nous écrivons **1 == 1**. Il y a une différence dans la programmation entre le signe **=** et le signe **==**.

Aussi, notez la formule **i++;**. Cela veut dire que la valeur actuelle de **i** sera augmentée de 1. Si, par exemple, **i** contient la valeur de 5, **i++;** augmentera **i** de 1 et vaudra 6. Il existe également une autre formule similaire : **i--;** pour diminuer la valeur d'une variable de 1.

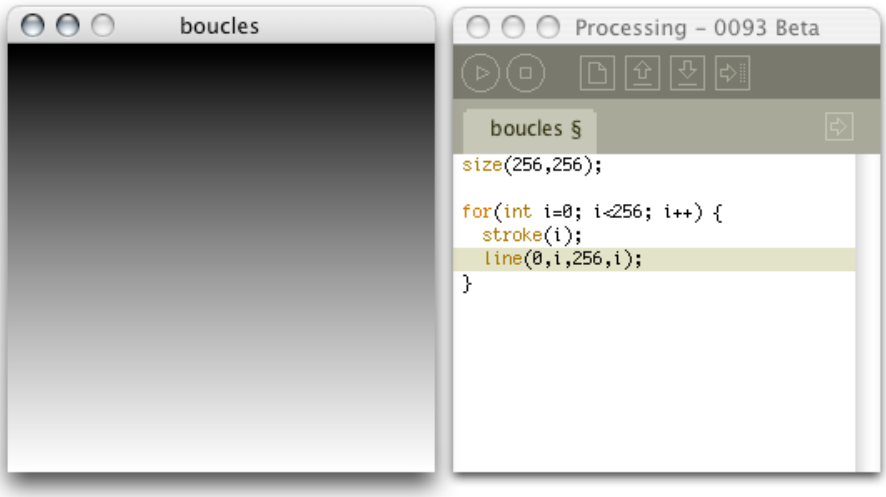
7.4 La boucle sophistiquée : for()

Il y a une deuxième façon de faire des boucles qui comptent de 1 à 256. Ces boucles s'appellent des boucles **for()** et sont plus compliquées à lire que les boucles **while().** Par contre, elles sont plus simples à manier, une fois qu'on est habitué à leur syntaxe bizarre.

Voici le même programme que précédemment avec une boucle **for():**

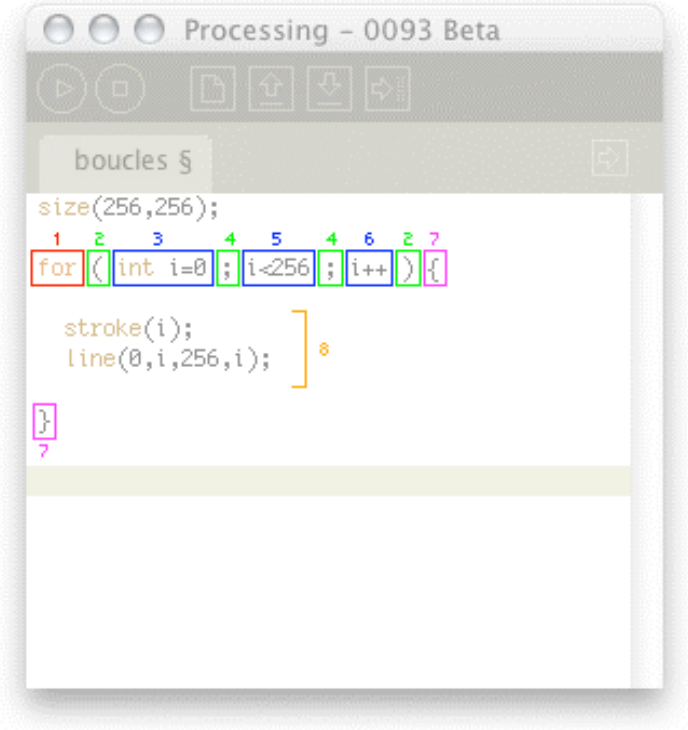
```
size(256,256);

for(int i=0; i<256; i++) {
 stroke(i);
 line(0,i,256,i);
}
```

Ce qui devrait donner comme résultat ceci :



Comme dans l'exemple précédent, la boucle **for()** exécute les actions entre les accolades de façon répétitive. L'avantage de la boucle **for()** par rapport à la boucle **while()**, c'est que l'on peut contrôler beaucoup plus précisément le déroulement du compteur dans le boucle.

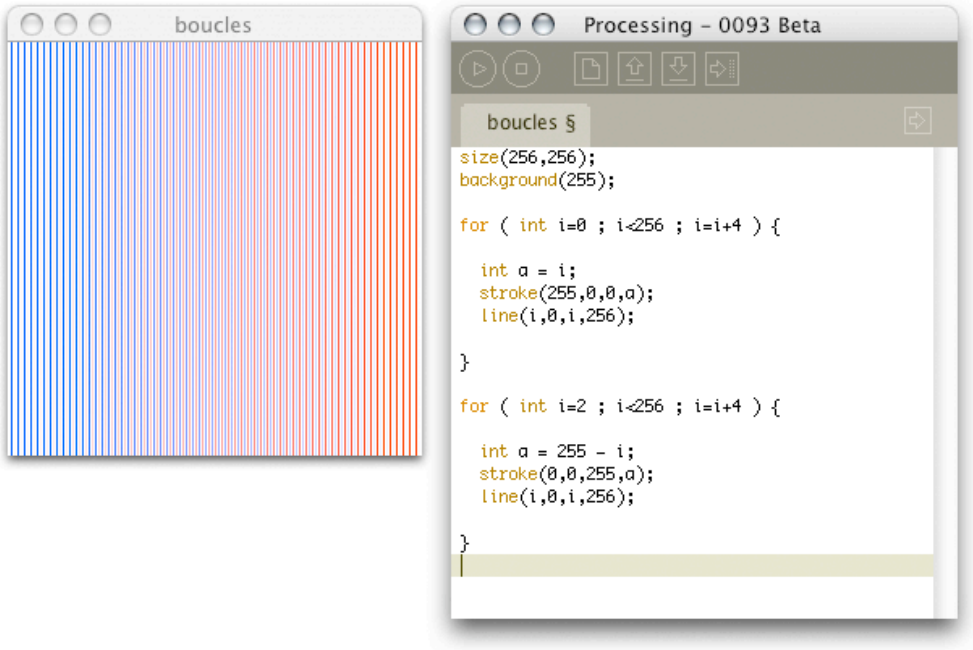
Par contre, la boucle **for()** est presque illisible pour quelqu'un qui ne l'a jamais rencontrée. Alors, décryptons-la :



Comme nous l'indique l'illustration, une boucle **for()** se compose de plusieurs éléments. Cela ressemble beaucoup à une méthode, avec les accolades et les parenthèses. Par contre, à l'intérieur des parenthèses, les choses se compliquent. Il y a trois choses qu'il nous faut pour définir la boucle : le nom du compteur, jusqu'à combien il faut compter, et finalement par combien il faut augmenter la variable à chaque passage. Voici de façon plus schématique la description de ces différentes composantes, suivant l'illustration :

1. l'expression `for`
2. les parenthèses
3. la déclaration (s'il faut) de notre variable compteur, et sa valeur de départ
4. des points-virgules pour séparer les trois informations
5. la condition pour rester dans la boucle (ici que la variable `i` reste inférieure à 256)
6. l'incrémentation du compteur (d'habitude `i++`)
7. les accolades pour définir ce qui doit être bouclé
8. les instructions de la boucle (ce que la boucle doit faire)

Voici un autre dessin qui utilise la boucle **`for()`**, mais en changeant, l'incrémentation de la variable `i` lors de chaque bouclage :



Notez également que le départ des deux boucles ne commencent par au même endroit. Dans la première boucle (rouge), la variable **`i`** commence avec la valeur 0, alors que dans la deuxième boucle (bleu), la variable **`i`** commence avec la valeur 2. C'est ce départ décalé qui donne le motif du dessin. Les deux `i` augmentent de 4 lors de chaque bouclage, mais comme les deux boucles ne commencent pas à compter au même endroit nous avons deux suites de nombres différents (0, 4, 8, 12, 16, ...contre 2, 6, 10, 14, 18, ...)

7.5 Exercice

Modifier les exemples de ce chapitre et essayer de comprendre ce qui se passe.

8. Les bifurcations

Enfin nous arrivons au dernier concept fondamental : les bifurcations.

8.1 Définition

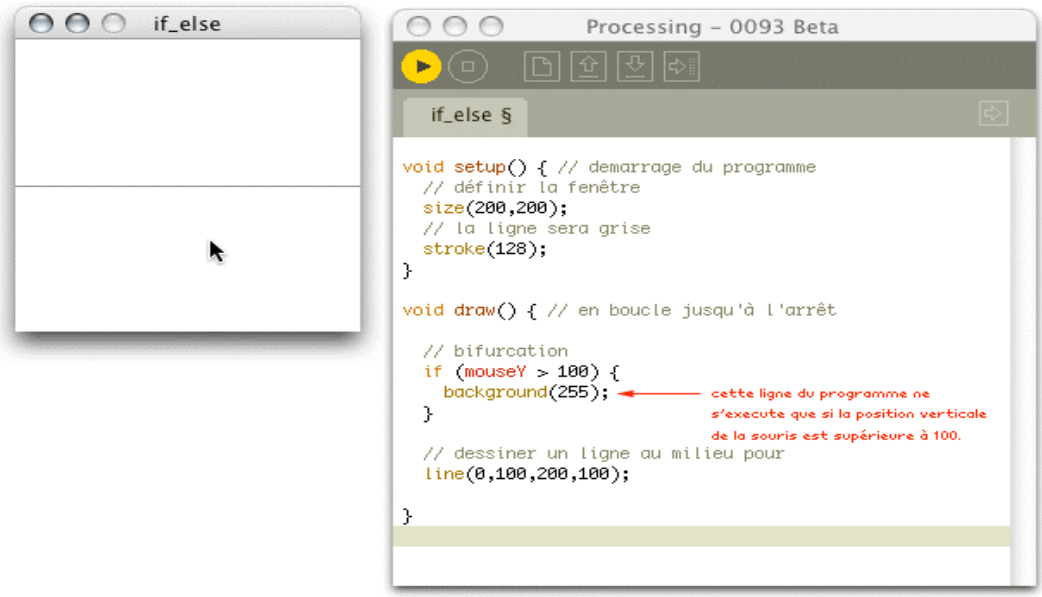
Les bifurcations permettent de changer l'ordre suivant lequel des parties de code du programme s'exécutent en fonction de critères spécifiques. Autrement dit, une bifurcation permet de changer le cours du programme — ce que le programme fait — la plupart du temps en fonction de données extérieures, c'est-à-dire en fonction d'interactions extérieures. Une bifurcation permet au programme de *moduler* son comportement *pendant qu'il s'exécute*. C'est un pas fondamental vers l'interactivité.

8.2 if...else

La plus connue des bifurcations (et la plus simple) est connue sous le nom **"if...else"**. C'est un des schémas les plus anciens de la programmation informatique : l'ordinateur fait quelque chose, mais si et seulement si un critère quelconque est vrai (**if**), sinon il ne fait rien ou fait autre chose à la place (**else**).

Comme pour les méthodes et les boucles, une bifurcation **if()** s'écrit avec des parenthèses et des accolades :

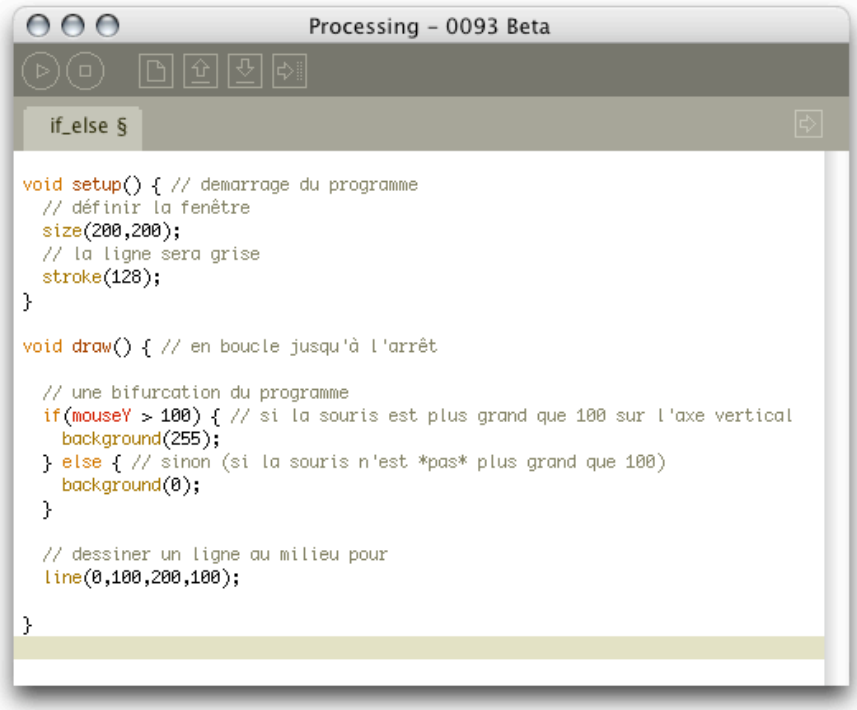
```
void setup() {
 size(200,200);
 stroke(128);
}

void draw() {
 if (mouseY > 100) {
   background(255);
 }
 line(0,100,200,100);
}
```



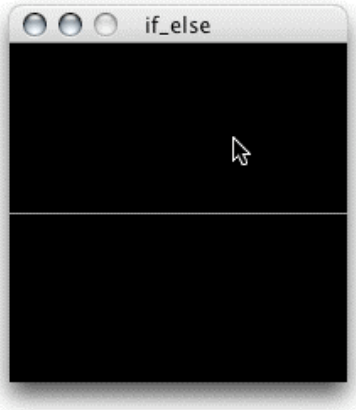
Ce programme dessine un fond gris, mais quand la souris est en dessous de la ligne du milieu (quand mouseY est plus grand que 100 sur l'axe vertical) le fond devient blanc. Notez que le fond ne change plus de couleur dès que le fond est devenu blanc. C'est parce que nous n'avons pas de cas contraire : au début de l'animation, le fond est gris (toujours le cas par défaut), puis dès qu'on passe en dessous de la ligne, le fond devient blanc.

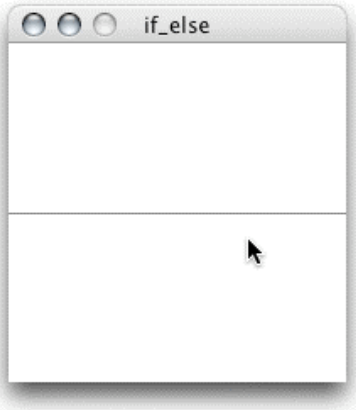
On peut aussi changer légèrement notre programme pour dire ce que le programme doit faire dans le cas *contraire*, c'est-à-dire quand la souris n'est *plus* en dessous de la ligne. Du coup, l'animation peut alterner entre deux états.

```
void setup() {
 size(200,200);
 stroke(128);
}

void draw() {

 if(mouseY > 100) {
   background(255);
 } else {
   background(0);
 }

 line(0,100,200,100);

}
```



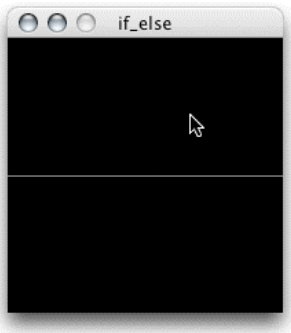
Voici le résultat de ce programme. Si on place la souris au-dessus de la ligne grise, le fond devient noir. Si on place la souris en dessous de la ligne grise, le fond devient blanc.

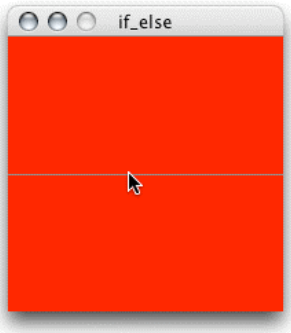


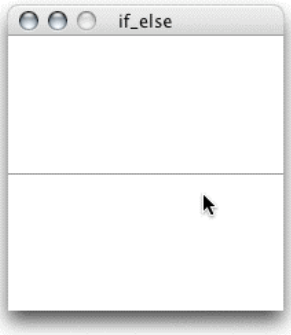


Cet exemple nous montre le *choix* que l'ordinateur peut faire à votre place sur le fonctionnement de votre programme. C'est comme si notre programme contenait deux sous-programmes potentiels : un premier programme quand la position verticale de la souris est plus grande que 100 pixels, et un autre programme dans le cas contraire. Attention (et vous l'avez peut-être déjà remarqué) : notre programme dessine une couleur de fond quand on est *plus grand* que 100, et une autre couleur *dans le cas contraire*. Ce qui veut dire que le fond est toujours noir lorsqu'on est exactement sur la ligne. Il n'y a que deux choix ici : au-dessus, puis tout le reste. C'est le sens de **else** : tout le contraire de ce qui est marqué entre les parenthèses du **if().**
Si cela vous semble un peu complexe, méditez sur le prochain exemple qui donne trois états. Un état par défaut (rouge), puis deux états possibles (un état au-dessus de la ligne, un autre état en dessous de la ligne).

```
void setup() {
 size(200,200);
}

void draw() {

 background(255,0,0);

 if (mouseY > 100) {
   background(255);
 } else if (mouseY < 100) {
   background(0);
 }

 stroke(128);
 line(0,100,200,100);

}
```







Comme on peut voir dans l'illustration, nous avons maintenant trois possibilités : fond noir quand on est au-dessus de la ligne, fond rouge quand on est sur la ligne, et fond blanc quand on est en dessous de la ligne. Ces trois possibilités viennent du fait qu'avant de choisir entre noir ou blanc, nous dessinons un fond rouge. Du

coup, seulement dans les cas où on n'est plus sur la ligne (supérieur à 100 ou inférieur à 100) on remplace ce fond rouge avec le blanc ou le noir. Cela revient à faire ce qu'on appelle souvent en informatique un choix "par défaut".

Au fait, on aurait pu écrire cette animation d'une autre façon. Voici une alternative pour arriver au même résultat :

```
void setup() {
 size(200,200);
}

void draw() {

 if (mouseY == 100) {
   background(255,0,0);
 }

 if (mouseY > 100) {
   background(255);
 }

 if (mouseY < 100) {
   background(0);
 }

 stroke(128);
 line(0,100,200,100);

}
```

Deux programmes peuvent donc donner le même résultat.

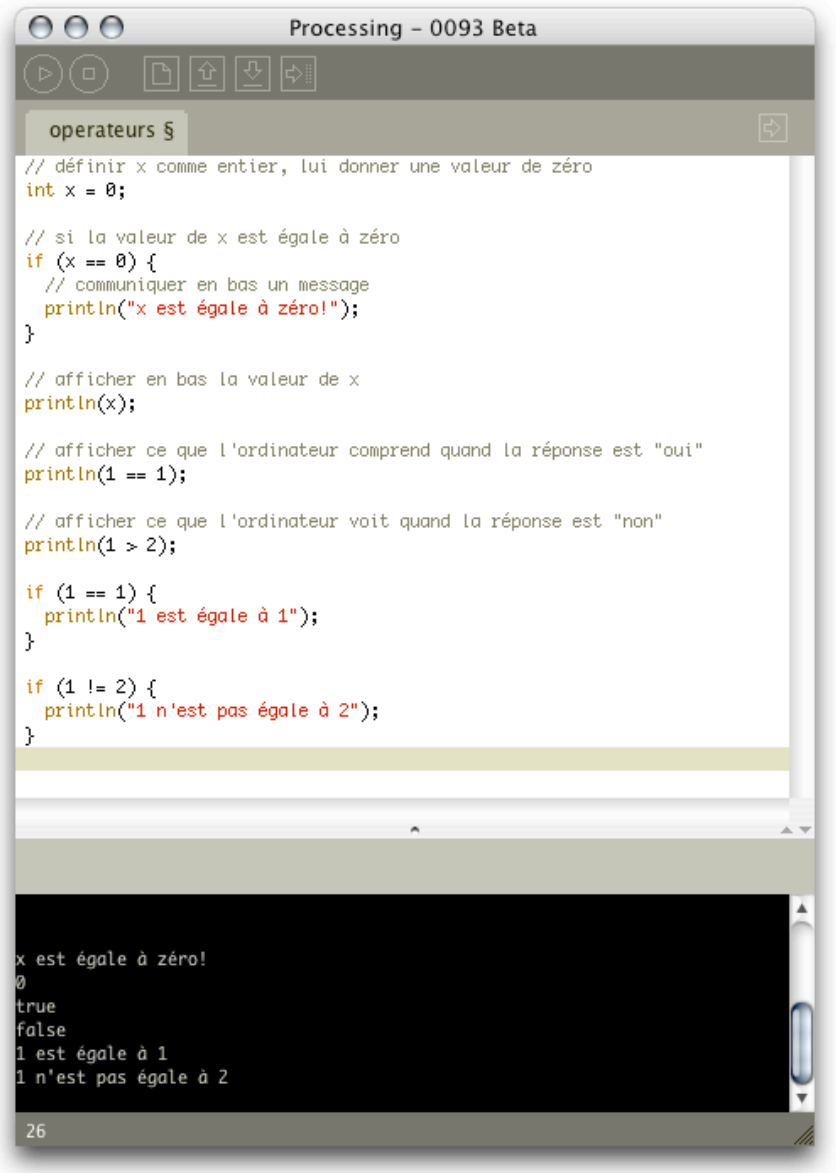
8.3 Les opérateurs, les comparaisons

Vous avez peut-être remarqué un signe un peu particulier dans notre dernier exemple. Au lieu d'écrire un seul signe *égal à* (**=**), nous en avons écrit deux (**==**). Il s'agit d'un "opérateur" qui ne fait pas un calcul arithmétique, mais une comparaison "logique".

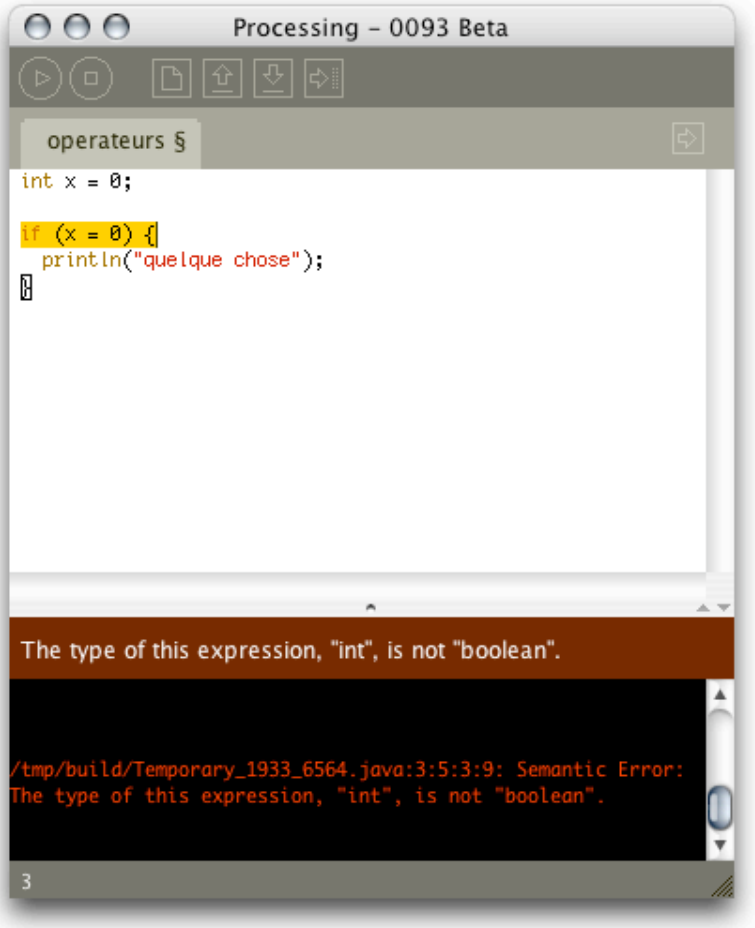
Qu'est-ce qu'une comparaison logique ?

Sans entrer dans les détails, disons qu'il faut mettre à l'intérieur des parenthèses d'une bifurcation **if()** ou **else(),** quelque chose qui expliquera à l'ordinateur ce que vous recherchez comme condition. Autrement dit, il faut expliquer ce qui doit être *vrai*, pour que l'ordinateur bifurque et exécute votre programme optionnel. En réalité, l'ordinateur est en train de transformer votre question (est-ce que **mouseY** est plus grand que 100 ?) en une "opération" qui donnera le résultat **true** ou **false** : c'est le fameux 0 et 1 de l'ordinateur.

Pour expliquer de manière plus explicite, demandons au processeur de nous montrer comment il voit toutes ces choses. Voici un petit exemple pour démontrer la façon dont un ordinateur utilise le signe == et ce qu'il s'attend à voir à l'intérieur des parenthèses pour exécuter votre code. Nous utiliserons la méthode **println()** pour voir avec la "vision" du processeur.



Comme le montre l'exemple (regardez bien en bas de la fenêtre), l'ordinateur *pose une question* quand il voit le code **x == 0**. Par contre, il *attribue* une valeur quand il voit le code **x = 0**. Le signe *égal à* (=) *donne* une valeur, alors que le signe « *est-il égal à* » (==) *compare* les valeurs et donne un résultat "vrai" (**true**) ou "faux" (**false**). C'est donc un *opérateur* de comparaison. C'est un *opérateur* parce qu'il donne des résultats, mais ces résultats ne sont pas des valeurs normales : ces résultats sont des valeurs **true** ou **false**. D'ailleurs, si on essaie de mettre un *égal à* classique (=) à l'intérieur des parenthèses, Processing signalera une erreur:



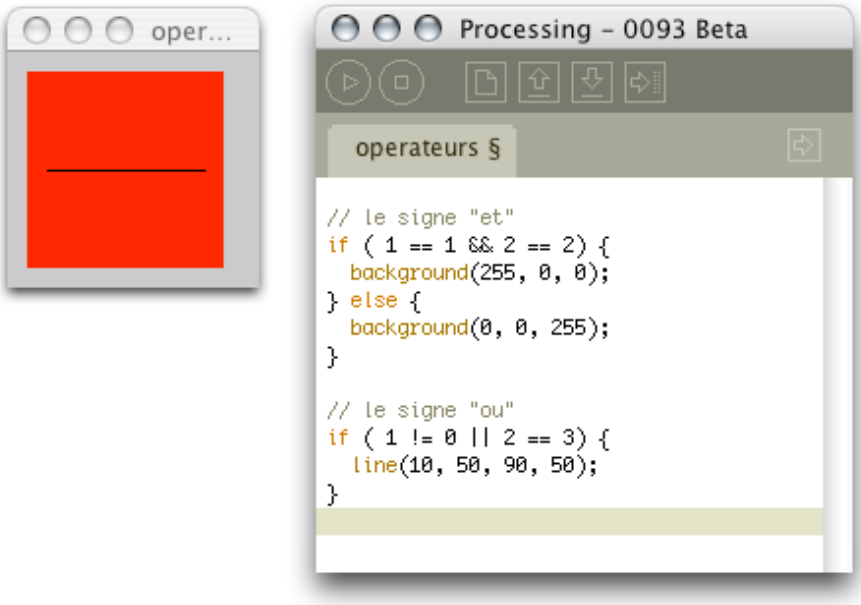
Il nous informe que ce qui se trouve entre les parenthèses n'est pas un "**boolean**". **boolean** est un nouveau *type* de variable qui ne peut avoir que deux valeurs différentes : **true** ou **false**

Voici quelques opérateurs "logiques" :

```
==          est égal à
!-          n'est pas égal à
>           est plus grand que
<           est plus petit que
```

On peut aussi poser deux questions à la fois, du genre : est-ce que ceci est vrai *et* est-ce que cela est vrai aussi ?

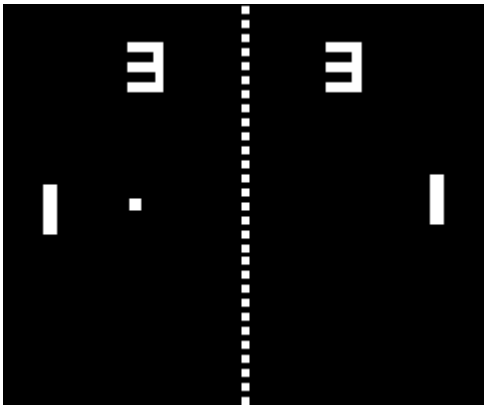
```
//  && est le signe "et"
if ( 1 == 1 && 2 == 2) {
 background(255, 0, 0);
} else {
 background(0, 0, 255);
}

// || est le signe "ou"
if ( 1 != 0 || 2 == 3) {
 line(10, 50, 90, 50);
}
```



En utilisant le signe && et le signe || on peut soit demander si deux questions sont toutes les deux vraies, ou demander si au moins une des deux questions est vraie. C'est ce qu'on appelle les signes "et" et "ou".

9. Un classique : le jeu Pong

Voici un exemple pour mettre en pratique les concepts de programmation que nous venons d'apprendre.
Nous allons commencer par reproduire la fameuse balle qui rebondit du premier jeu video Tennis For Two (Higinbotham, 1958), repris quinze ans plus tard dans le premier jeu vidéo commercialisé : Pong !



9.1 Réactif + automatique = jeu

Jusqu'ici, nous avons créé des dessins et des animations réactifs. Une animation réactive, comme son nom l'indique, réagit à l'utilisateur, mais elle a tendance à réagir d'une manière passive : l'utilisateur bouge à droite / le dessin bouge à droite, l'utilisateur bouge à gauche / le dessin bouge à gauche ; et ainsi de suite.
Le contraire du réactif est le programme automatique. Le programme est passif d'une autre manière : il réagit aux conditions internes du programme, sans prendre en compte les mouvements de l'utilisateur.
La rencontre de ces deux couches donnera naissance au jeu vidéo. Si je contrôle trop l'ensemble des éléments du programme, je n'ai plus de jeu — je suis dans un système réactif. Si le programme contrôle trop l'ensemble de ses éléments, le programme devient purement automatique. Mais si je crée la rencontre de ces deux couches, où je contrôle un élément (réactif) dans un milieu hostile (automatique), je suis du coup dans un jeu. Le jeu Pong ! repose totalement sur ce principe, il est même presque le degré zéro de cette rencontre.

9.2 Automatique

La partie qui nous intéresse pour l'instant est la couche automatique. Nous savons déjà comment faire réagir le programme à l'utilisateur, il faut maintenant voir comment faire tourner un élément du programme de façon automatique.

Pour faire rebondir une balle, nous avons besoin de trois choses essentielles :

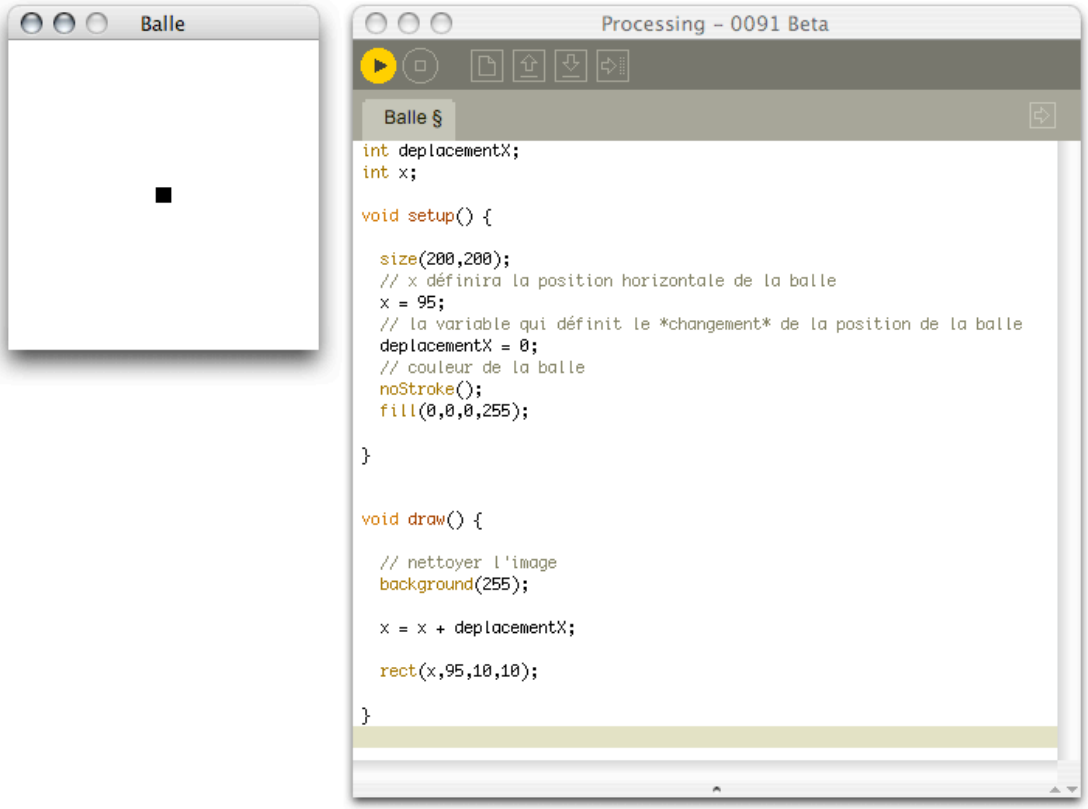
1. Deux variables pour définir la position (horizontale + verticale) de la balle
2. Deux variables pour diriger le déplacement (horizontal + vertical) de la balle entre chaque cycle de la méthode **draw()**
3. Une interrogation (**if()**) sur la position de la balle : si la balle touche un des bords de l'image, changer la variable de déplacement correspondante.

9.3 Déplacement

Nous allons commencer par la définition des variables de position et de mouvement. Pour comprendre la logique, essayez l'exemple suivant dans votre copie de Processing :

```
int deplacementX;
int x;

void setup() {
 size(200,200);
 // x définira la position horizontale de la balle
 x = 95;
 // la variable qui définit le *changement* de la position de la balle
 deplacementX = 0;
 // couleur de la balle
 noStroke();
 fill(0,0,0,255);
}

void draw() {
 // nettoyer l'image
 background(255);

 // changer la position de la balle
 x = x + deplacementX;

 rect(x,95,10,10);
}
```

Et voici le résultat. Notez que notre magnifique balle carrée ne bouge pas encore :

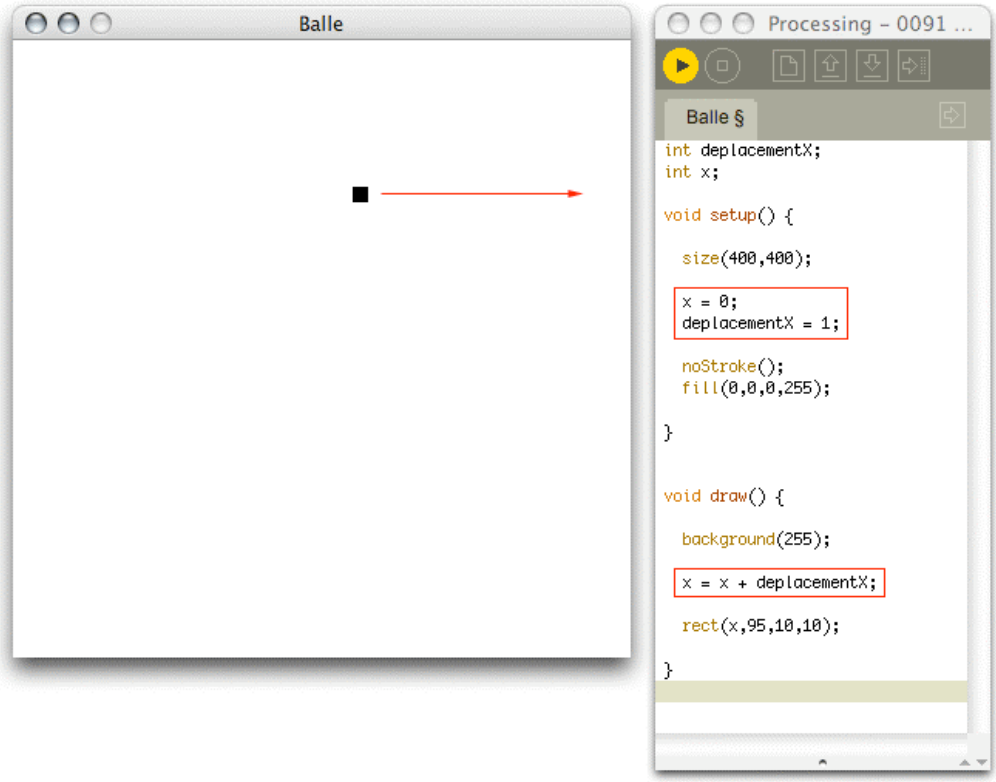


Pourquoi la balle ne bouge pas ? Et bien, parce que 30 fois par seconde, nous demandons à la variable **x** de changer sa valeur en ajoutant la valeur d'une autre

variable. Malheureusement, celle-ci est actuellement à 0 (**deplacementX = 0;**). Si j'ajoute la valeur 0 à 95, par exemple, j'aurais toujours 95. Par contre, si nous ajoutons une valeur positive, notre variable x changera de position :

```
int deplacementX;
int x;

void setup() {
 size(400,400);
 // x définira la position horizontale de la balle
 x = 0;
 // la variable qui définit le *changement* de la position de la balle
 deplacementX = 1;
 // couleur de la balle
 noStroke();
 fill(0,0,0,255);
}

void draw() {
 // nettoyer l'image
 background(255);
 x = x + deplacementX;
 rect(x,95,10,10);
}
```

Regardez comment la variable **deplacementX** maintenant arrive à bouger le carré d'un pixel lors de chaque cycle dans la méthode **draw()**.



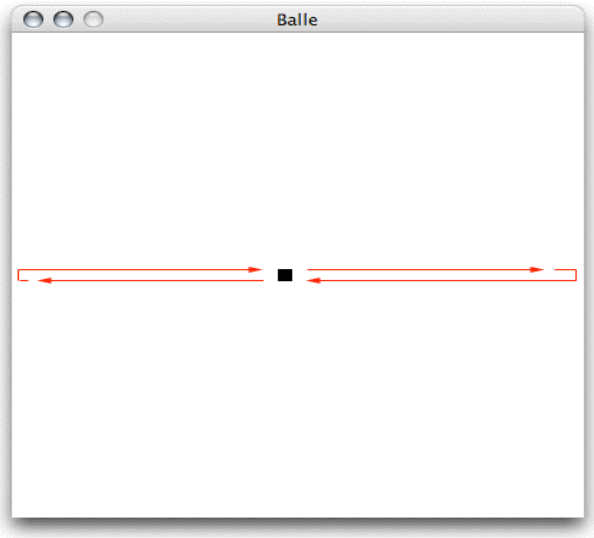
Lors de chaque cycle de la méthode **draw()**, la variable **deplacementX** augmente la valeur de la variable **x**. Puisqu'on dessine, 30 fois par seconde, un rectangle à la position de cette variable **x**, chaque fois que la valeur de **x** change, la position du rectangle change.

9.4 Rebondissements

D'accord, nous avons du mouvement, mais l'action est toujours la même, l'histoire est un peu décevante. Ce qu'il nous faut, c'est un rebondissement.
Pour faire "rebondir" notre balle, nous avons besoin d'interroger sa position lors de chaque cycle : quand la valeur de **x** sera trop grande, nous inverserons le sens du mouvement en donnant une valeur *négative* à la variable **deplacementX** :
Nous ferons la même chose quand la valeur sera trop petite :

```
int deplacementX;
int x;

void setup() {
 size(400,400);
 x = 0;
 deplacementX = 1;
 noStroke();
 fill(0,0,0,255);
}


void draw() {
 background(255);
 x = x + deplacementX;

 if (x > width) {
   deplacementX = -1;
 }

 if (x < 0) {
   deplacementX = 1;
 }

 rect(x,195,10,10);
}
```

Ce qui donne un mouvement d'aller-retour de la balle :

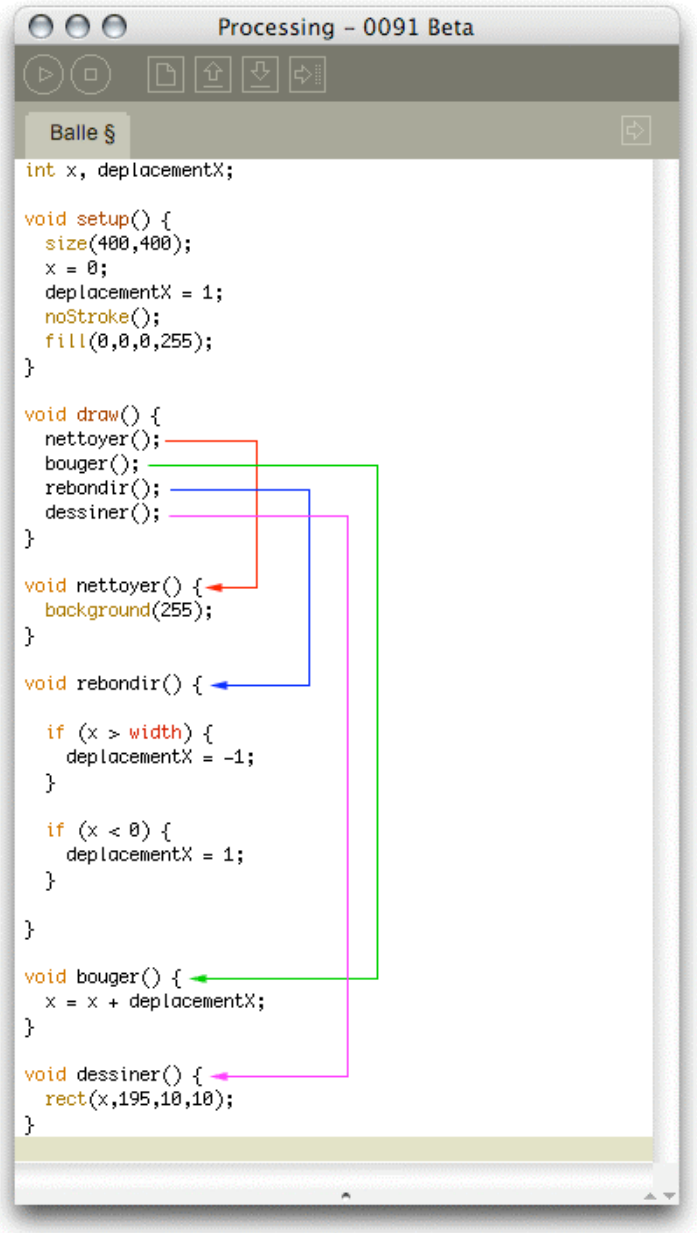


9.5 Mieux organiser le programme

Nous connaissons les *actions* de notre programme :
effacer tout, incrémenter/décrémenter **x,** vérifier la position, dessiner le carré.

Ce serait bien si on pouvait mieux lire ces actions en lisant le programme. Dans l'exemple suivant, nous avons justement pris toutes nos actions, et nous les avons rassemblées dans des méthodes distinctes :

```
int deplacementX;
int x;

void setup() {
 size(400,400);      // taille image
 x = 0;              // position horizontale
 deplacementX = 1;   // déplacement
 noStroke();         // pas de trait
 fill(0,0,0,255);    // remplissage noir
}

void draw() {
 nettoyer();
 bouger();
 rebondir();
 dessiner();
}

void nettoyer() {
 background(255);
}

void rebondir() {
 if (x > width) {
   deplacementX = -1;
 }
 if (x < 0) {
   deplacementX = 1;
 }
}

void bouger() {
 x = x + deplacementX;
}

void dessiner() {
 rect(x,195,10,10);
}
```



Bien sûr, il s'agit juste d'une question de style. Néanmoins, la méthode **draw()** est maintenant beaucoup plus lisible – en tout cas plus lisible pour un être humain. Regardez bien, il utilise maintenant 4 verbes, lisibles par nous, plutôt que des noms de méthodes avec des paramètres et des variables bizarroïdes. Même si on ne savait pas programmer, on arriverait probablement à lire la méthode **draw()** :

```
void draw() {
 nettoyer();
 bouger();
 rebondir();
 dessiner();
}
```

En un seul coup d'œil, nous savons maintenant que, 30 fois par seconde, le processeur va **nettoyer()** l'image, **bouger()** la valeur de **x**, vérifier si **x** doit rebondir, puis **dessiner()** le rectangle à la position de **x**.
Rien n'a changé dans le résultat sur l'écran, mais notre programme est maintenant prêt à être modifié sans devenir illisible. On peut, par exemple, maintenant ajouter le mouvement/rebondissement vertical, voire même ajouter une notion de vitesse, sans créer une bouillie de code dans notre fenêtre de programmation.

9.6 Vertical

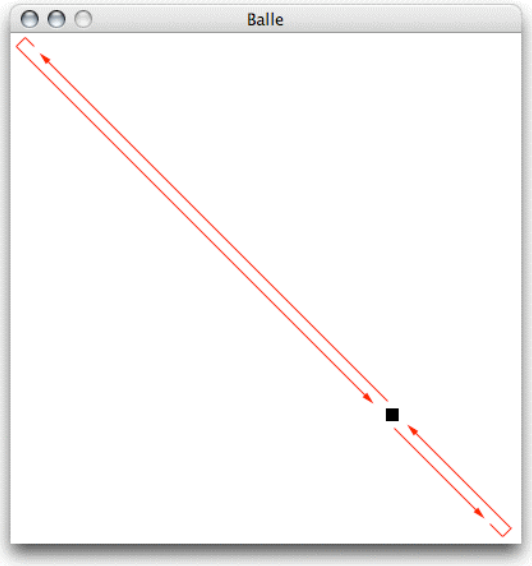
Pour ajouter un mouvement vertical, il faut ajouter 2 variables (une pour la position, une autre pour le déplacement). Comme notre programme est maintenant sectionné en petites parties, il devient relativement plus facile de savoir où ajouter ces éléments :

```
int deplacementX, deplacementY; // deux variables de déplacement
int x, y; // deux variables de position

void setup() {
 size(400,400);
 x = 0;
 y = 0;              // position verticale
 deplacementX = 1;
 deplacementY = 1;   // deplacement vertical
 noStroke();
 fill(0,0,0,255);
}

void draw() {
 nettoyer();
 bouger();
 rebondir();
 dessiner();
}

void nettoyer() {
 background(255);
}

void rebondir() {
 if (x > width) {
   deplacementX = -1;
 }

 if (x < 0) {
   deplacementX = 1;
 }

 if (y > width) {      // vérifier rebondissement en bas
   deplacementY = -1;
 }

 if (y < 0) {          // vérifier rebondissement en haut
   deplacementY = 1;
 }
}

void bouger() {
 x = x + deplacementX;
 y = y + deplacementY;  // bouger sur l'axe vertical
}

void dessiner() {
 rect(x,y,10,10);      // dessiner la position selon x et y
}
```

Du coup notre balle bouge et rebondit sur l'axe horizontal et vertical. Le changement du programme n'était pas aussi douloureux qu'il aurait été si on avait écrit tout à l'intérieur du bloc **draw().**

9.7 Vitesse

Vous avez noté quelque chose d'étrange avec le mouvement ?



La balle fait toujours une ligne droite, comme avant, mais en diagonale. Pourquoi ? Parce que nous augmentons **x** en même temps que nous augmentons **y**. En gros, la valeur de **x** a la même valeur que **y**. Quand **x=5, y=5** et quand **x=150, y=150**. La balle rebondit donc en faisant des trajectoires en diagonale.
Pour changer cela, il faut changer la valeur de déplacement. Au lieu d'être toujours 1 ou -1, il faudrait que le déplacement horizontal soit, par exemple, 5 alors que le déplacement vertical serait -3. Il s'agit de l'introduction de la notion de vitesse dans le mouvement : au lieu de toujours se déplacer de 1 pixel à chaque cycle, on se déplacera de *n* pixels :

```
int deplacementX, deplacementY;
int x, y;

void setup() {
 size(400,400);
 x = 200;  // commençons au centre de l'écran
 y = 200;
 deplacementX = 5;
 deplacementY = -3;
 noStroke();
 fill(0,0,0,255);
}

void draw() {
 nettoyer();
 bouger();
 rebondir();
 dessiner();
}

void nettoyer() {
 background(255);
}

void rebondir() {

 if (x > width) {
   deplacementX = -5;
 }
```

```
 if (x < 0) {
   deplacementX = 5;
 }

 if (y > width) {
   deplacementY = -3;
 }

 if (y < 0) {
   deplacementY = 3;
 }
}

void bouger() {
 x = x + deplacementX;
 y = y + deplacementY;
}

void dessiner() {
 rect(x,y,10,10);
}
```

9.8 Ajouter de la souplesse

Notre balle bouge maintenant d'une manière plus variée. Mais il reste un dernier point un peu problématique : la balle fera toujours les *mêmes* mouvements : ce sera toujours à la vitesse de 5 et de –5 sur l'axe horizontal, ou de 3 et de –3 sur l'axe vertical. En plus, si on veut changer ces valeurs en haut de programme...

```
deplacementX = 1;
deplacementY = -2;
```

...on est aussi obligé de changer les valeurs en bas du programme :

```
 if (x > width) {
   deplacementX = -1;
 }

 if (x < 0) {
   deplacementX = 1;
 }

 if (y > width) {
   deplacementY = -2;
 }

 if (y < 0) {
   deplacementY = 2;
 }
```

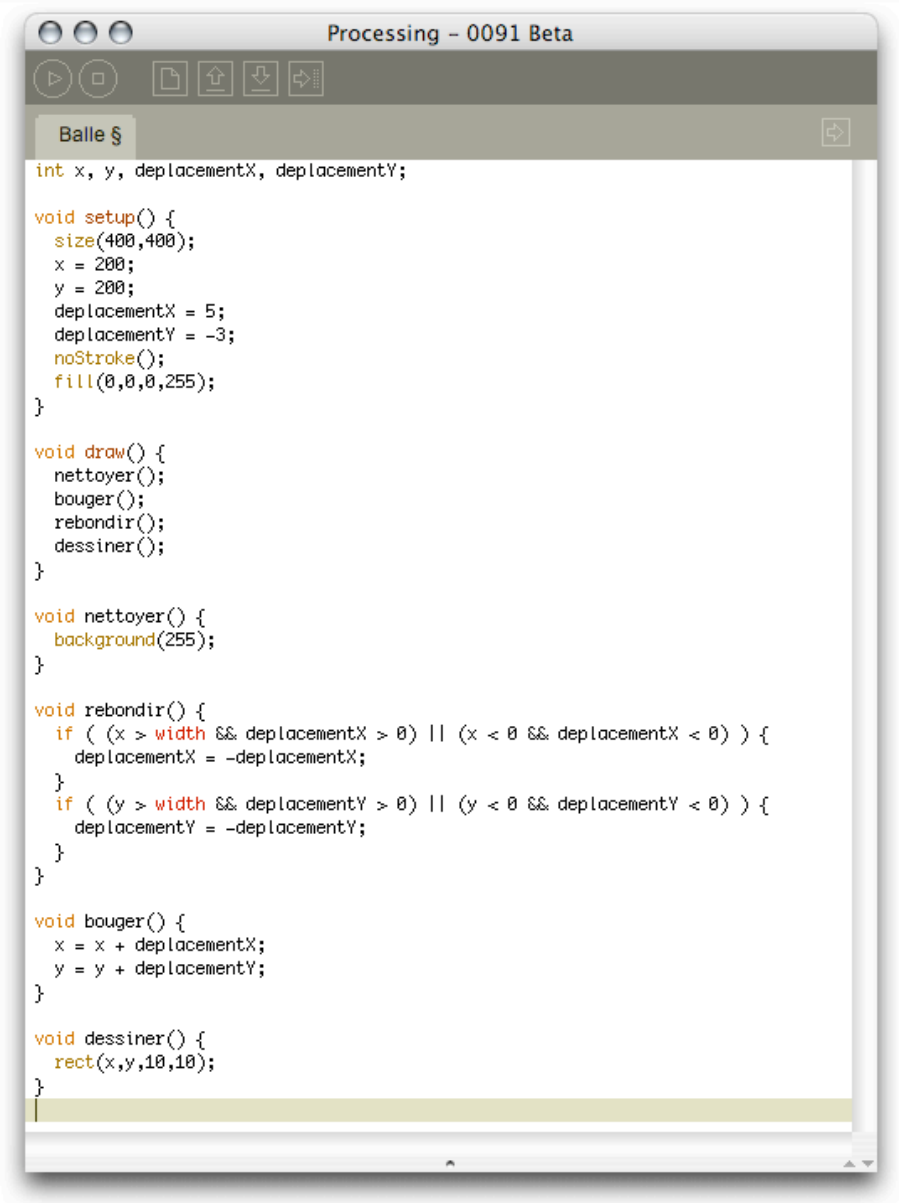
Devoir changer des valeurs partout dans un programme peut être fatiguant et nous amener à faire des erreurs.
On pourrait ajouter donc une petite sophistication, en expliquant à la méthode **rebondir()** comment s'inverser, *quelle que soit sa valeur*.
Du coup, **rebondir()** respectera la valeur du déplacement, mais inverserait le sens (+ / - (= le signe)) de cette valeur.

Voici la version plus sophistiquée de la méthode **rebondir()** :

```
void rebondir() {

 // si on est trop à droite ET si le déplacement horizontal est positif
 if (x > width && deplacementX > 0) {
   deplacementX = -deplacementX; // inverser la valeur quelle qu'elle soit)
 }

 // si on est trop à gauche ET si le déplacement horizontal est négatif
 if (x < 0 && deplacementX < 0) {
   deplacementX = abs(deplacementX); // abs() enlève le signe - de la valeur
 }

 // si on est trop bas ET si le déplacement vertical est positif
 if (y > width && deplacementY > 0) {
   deplacementY = -deplacementY; // rendre négative la valeur
 }

 // si on est trop haut ET si le déplacement vertical est négatif
 if (y < 0 && deplacementY < 0) {
   deplacementY = abs(deplacementY); // rendre positive cette valeur
 }
}
```

Du coup, nous pouvons changer la valeur de départ de **deplacementX** ou de **deplacementY** et le programme fonctionnera toujours.
Mais notre code est quand même un peu long. Peut-être pourrait-on le rendre *encore plus sophistiqué*. Voici donc une version encore plus courte qui nous donnera le même comportement.

```
void rebondir() {

 // si on est (trop à droite ET positif) OU si on est (trop à gauche ET négatif)
 if ( (x > width && deplacementX > 0) || (x < 0 && deplacementX < 0) ) {
   deplacementX = -deplacementX;
 }

 // si on est (trop bas ET positif) OU si on est (trop haut ET négatif)
 if ( (y > width && deplacementY > 0) || (y < 0 && deplacementY < 0) ) {
   deplacementY = -deplacementY;
 }

}
```



9.9 Exercice

Créez un programme qui dessine automatiquement en laissant des traces derrière le mouvement de la balle.

10. Les images photographiques

Cela fait pas mal de temps que l'on est sur Processing et peut-être que vous vous êtes posé la question : est-ce qu'on est condamné à ne dessiner qu'avec des formes géométriques dans une fenêtre ? Peut-on utiliser Processing pour faire autre chose que des dessins à la John Maeda ou des tableaux cinétiques comme on en trouverait au musée Vasarely ?
Une des choses qui peut nous faire sortir du dessin géométrique est l'image photographique. Autrement dit, des images composées de pixels, importées par exemple à partir d'un appareil photo numérique.

10.1 Importation

Processing est capable de générer évidemment des dessins à partir de rien, ou du moins à partir de rien d'autre que vos instructions. Mais pour afficher une image, il a besoin de l'importer.
Il y a deux possibilités pour importer une image dans Processing. Soit l'image se trouve à l'extérieur du projet, soit elle se trouve dans le dossier "data" du projet. Dans les deux cas, les instructions seront les mêmes, mais l'adresse du fichier sera légèrement différente.

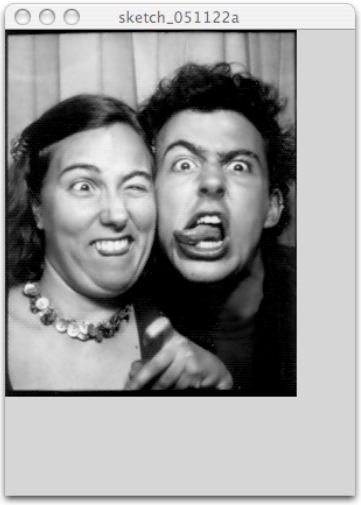
Commençons avec une image importée du wikimedia :

Ce programme importe le fichier "MakingAFace.jpg" qui se trouve à l'adresse "http://upload.wikimedia.org/wikipedia/commons/4/44/MakingAFace.jpg".
Processing importe cette image directement dans la mémoire du logiciel avec la ligne suivante :

```
photo = loadImage("http://upload.wikimedia.org/wikipedia/commons/4/44/MakingAFace.jpg");
```

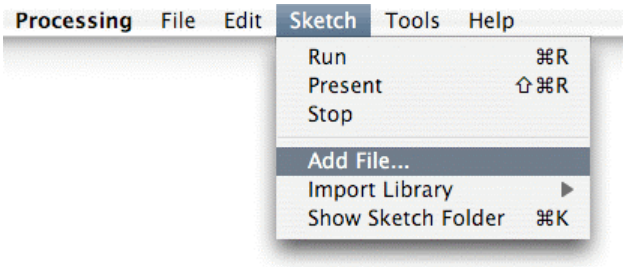
Une fois cette image importée, nous l'affichons sur l'écran avec la commande :

```
image(photo, 0, 0);
```

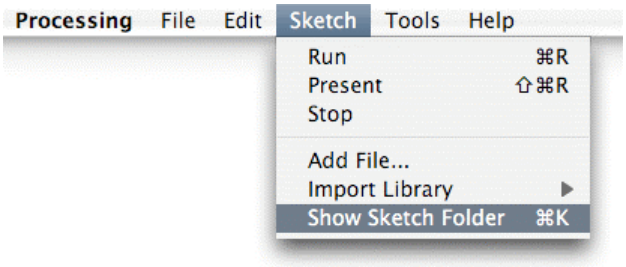


Si on écrit autre chose que 0,0, l'image s'affichera à partir de ce point-là. Les deux chiffres qui suivent le nom de la variable (photo) contenant l'image indiquent l'emplacement x et y de l'image, comme pour les rectangles.

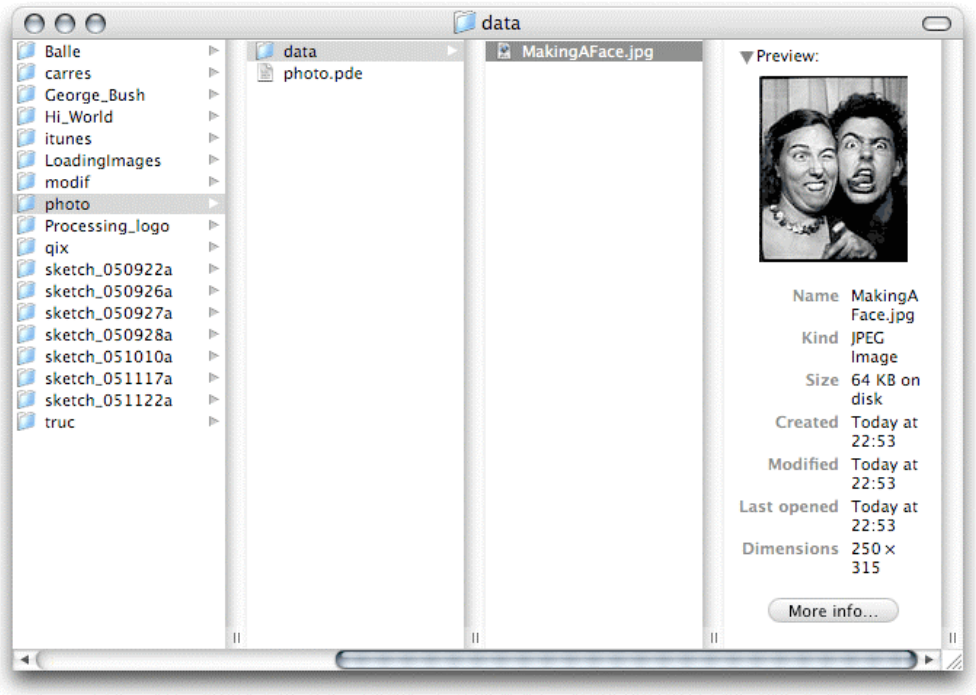
Si on veut maintenant intégrer une image directement dans notre programme, il faut l'ajouter au dossier "**data**". Sélectionnez ***Sketch > Add File*** du menu **Processing** :



Ensuite, choisissez l'image que vous voulez importer dans votre programme. Une fois l'image importée, vous pouvez vérifier son emplacement en sélectionnant ***Sketch > Show Sketch Folder***



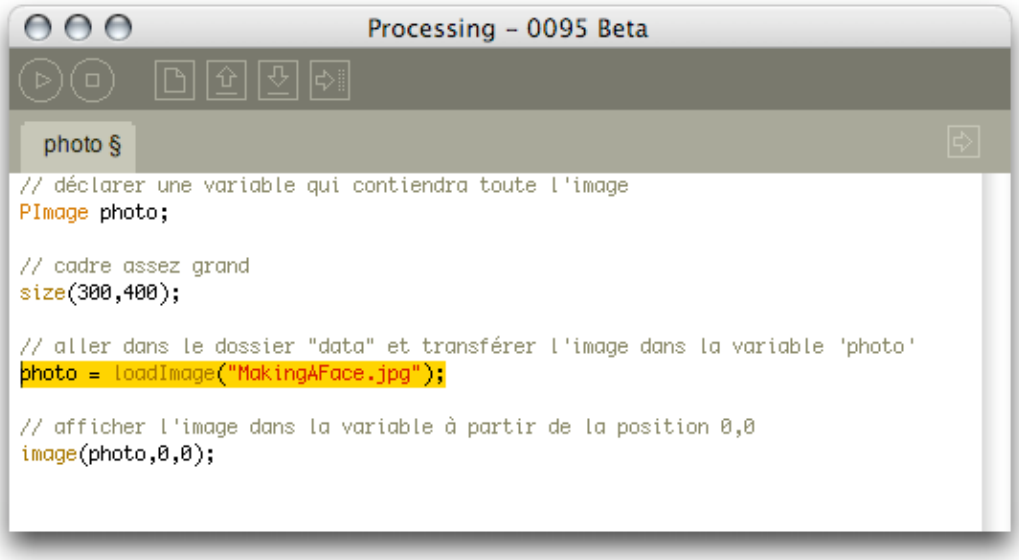
Ce qui vous affichera dans le Finder le dossier "**data**" qui contient maintenant votre image.



Dans le programme, il y a juste une ligne à modifier pour que ce programme fonctionne. On doit maintenant seulement écrire...

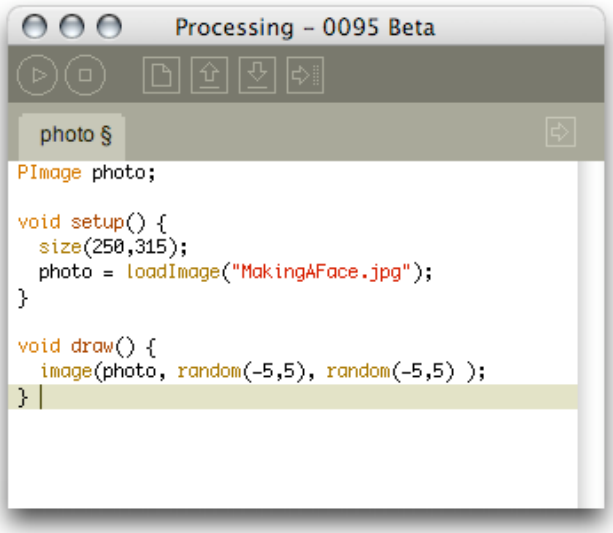
```
photo = loadImage("MakingAFace.jpg");
```

...pour que notre image fonctionne dans notre programme. Quand nous compilons ensemble notre projet pour le mettre sur le web, tout ce qui se trouve dans le dossier ***data*** sera compressé dans le fichier web. Attention donc à n'avoir dans ce dossier que les données absolument nécessaires au bon déroulement de votre programme. Voici une illustration du code final :



10.2 Image en mouvement

On peut maintenant facilement faire bouger l'image, en utilisant tous les subterfuges que nous avons appris dans les chapitres précédents. Par exemple, nous pouvons maintenant remplacer la position 0,0 de l'image avec quelque chose de plus intéressant. Pour ce faire, il faut par contre mettre l'ensemble dans les boucles **setup()** et **draw()** :



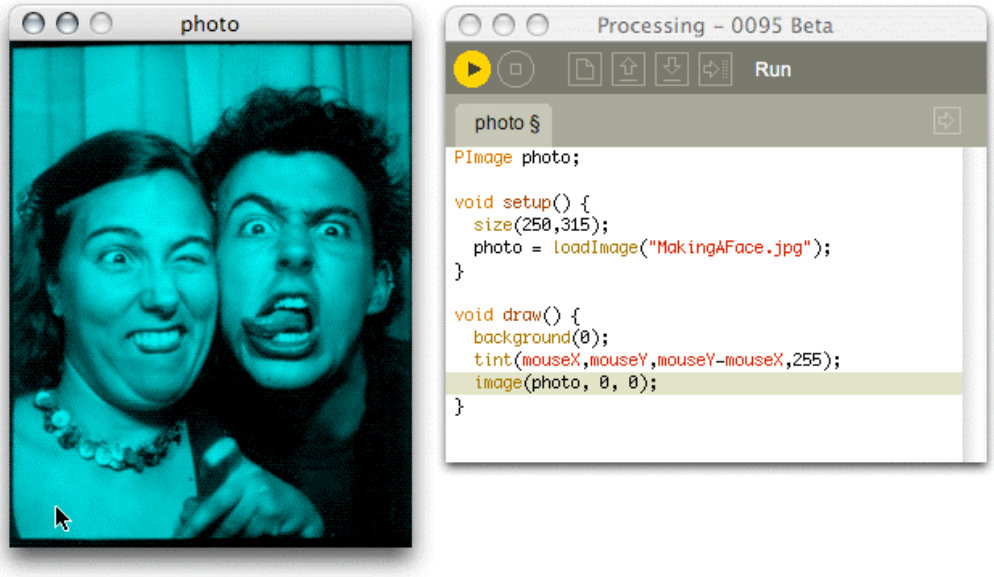
Avec ce code, notre programme tournera maintenant en boucle, et bougera aléatoirement notre image.

10.3 Resize

Ce serait bien de changer la taille de l'image dynamiquement, non ? Et bien, il suffit d'ajouter deux chiffres après la position **x, y** et votre image se redimensionnera à ces deux valeurs ***width*** et ***height*** :

10.4 Teinter l'image

Les concepteurs de Processing ont ajouté également la possibilité de teinter l'image avant qu'elle ne s'affiche sur l'écran. Il suffit – comme pour la couleur de l'encre **stroke()** et le pot de peinture **fill()** – de choisir trois couleurs + la transparence, pour déterminer la teinte de l'image avec la méthode **tint().**



10.5 Résumé

Il y a de multiples possibilités avec les images. Nous avons seulement montré les bases. Il existe plein d'exemples de toutes les possibilités sur le site de Processing.

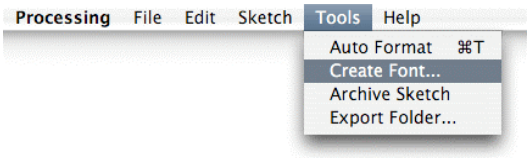
10.6 Exercice

Trouver une image dans Google, récupérez son *url* (c'est-à-dire son adresse http://...) et faites un programme qui anime d'une façon ou d'une autre cette image.

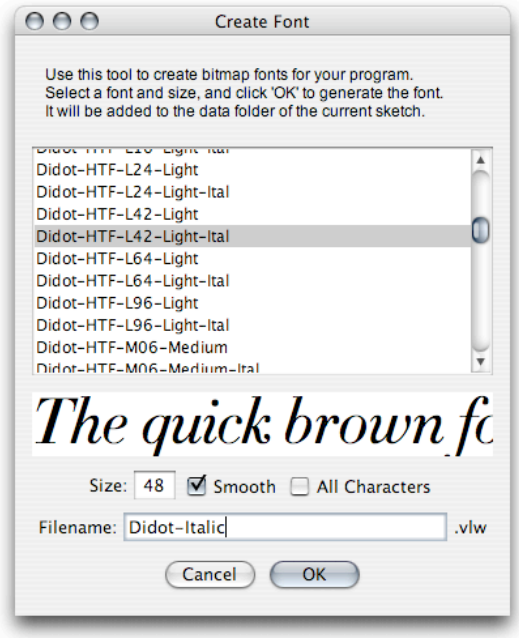
11. Typographie

Travailler avec des signes typographiques dans Processing ressemble beaucoup à ce que nous venons d'apprendre dans le chapitre précédent. Mais au lieu d'importer une image dans Processing, il faut importer une police.

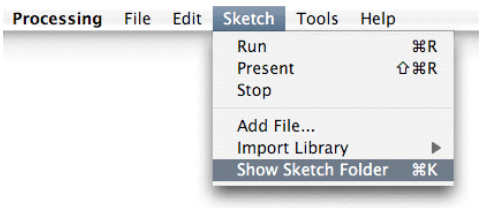
Comme pour l'importation des images, Processing vous aide dans l'importation des polices. Il suffit de choisir dans le menu ***Tools > Create Font***



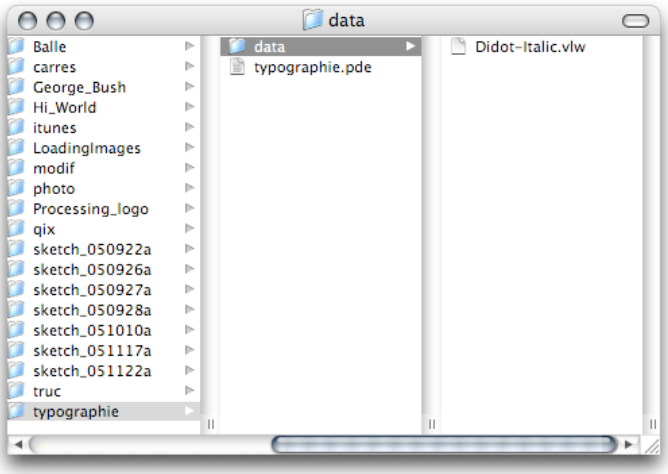
Processing vous propose ensuite une liste de toutes les polices sur votre ordinateur.



Choisissez-en une et faites attention de lui donner un nom un peu plus facile à retenir que le nom offert par Processing au départ. Ce nom correspond au fichier dans le dossier ***data*** où résidera dorénavant votre police. Comme dans le chapitre précédent, vous pouvez demander de voir le dossier ***data*** de votre programme.



Dans ce dossier, vous trouverez enfin votre **fichier.vlw** tel que vous l'avez nommé lors du processus de création.

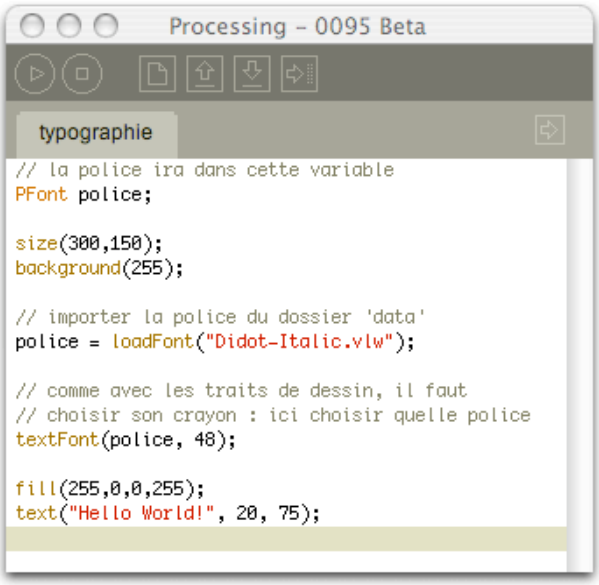


11.1 Importer, sélectionner, écrire

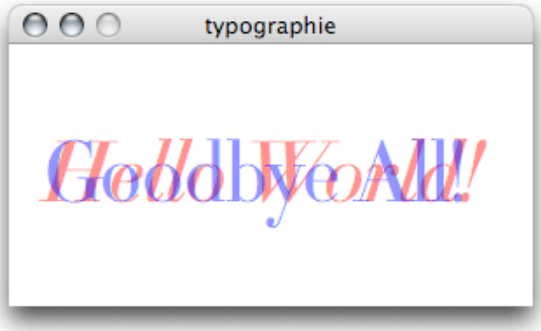
Comme pour les images, nous allons enfin importer notre police et nous en servir dans notre programme. Voici un petit programme simple, avec le célèbre "**Hello World !**" que tape tout programmeur à ses débuts de programmation :

Le code est bien commenté, regardez-le bien. Il faut non seulement importer une police avant d'écrire, mais il faut la choisir, notamment pour définir la taille de la police lors de l'écriture.

Si nous voulons colorier la police, il suffit d'ajouter une indication de couleur, comme pour les rectangles ou les ellipses :



Et si nous voulons écrire avec deux formes typographiques différentes, il suffit de charger deux polices et de passer à l'un et à l'autre comme s'il s'agissait d'un changement de couleur de crayon :



10.2 Machine à écrire

Nous avons déjà vu comment utiliser les positions **x** et **y** de la souris pour moduler nos dessins. Dans le chapitre sur la typographie, ce serait peut-être le bon moment pour introduire les touches du clavier !

Dans Processing il y a deux approches principales pour détecter si une touche est pressée.

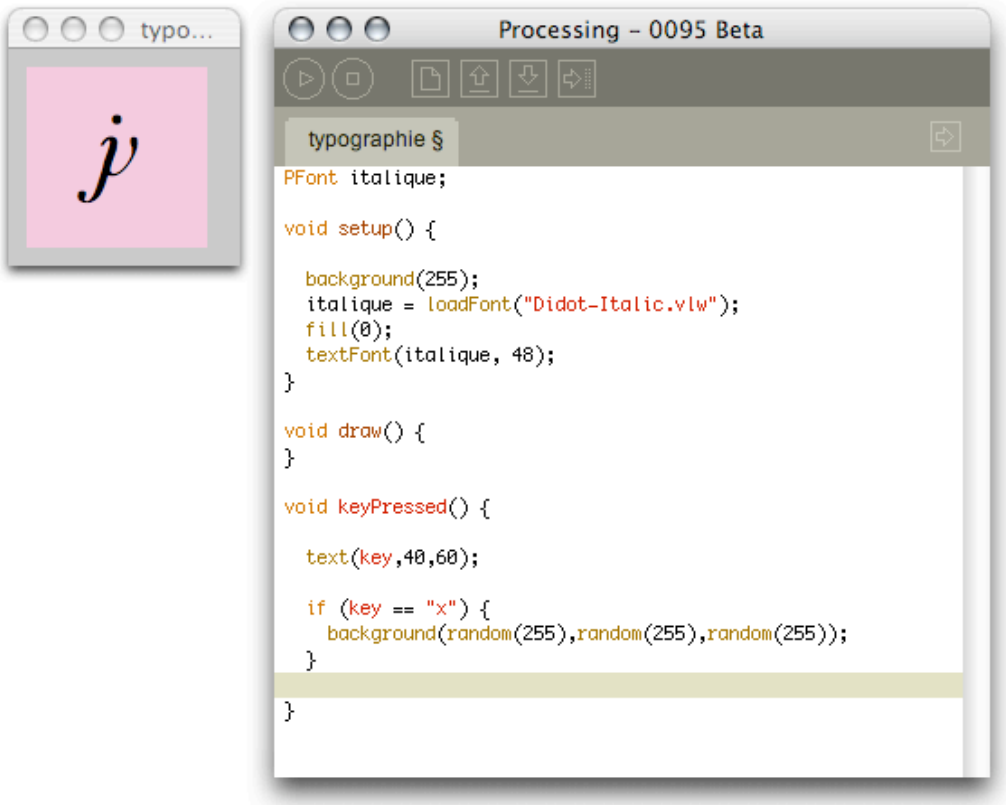
Dans la première approche, nous vérifions si une touche est actuellement en train d'être pressée, et si c'est le cas nous regardons quelle est la dernière touche enregistrée par la machine.

Dans la deuxième approche, nous demandons à Processing de nous informer chaque fois que l'utilisateur appuie de nouveau sur des touches.

Pour simplifier notre machine à écrire, nous allons utiliser la deuxième approche, parce qu'elle est plus facile à comprendre.

Comme les méthodes **setup()** et **draw()**, il existe une méthode que Processing exécutera automatiquement quand l'utilisateur appuie sur une touche : **keyPressed()**.

On peut ensuite savoir quelle touche il a pressé en utilisant la variable **key** :



Enfin, si vous voulez connaître la seconde approche, elle consiste à interroger en permanence la machine pour voir si une touche est active, comme ceci :

```
void draw() {

 if (keyPressed) {
   background(random(255),random(255),random(255));
 }
}
```

Vous pouvez, comme dans l’illustration précédente, voir s’il s’agit de telle ou telle touche. Par contre, le désavantage de cette approche, c’est qu’elle est appelée en permanence, ce qui peut créer un peu de chaos, et du coup être plus difficile à manier pour les débutants.

10.3 Exercice

Dessiner "Hello World" sur l’écran ! Et si vous êtes motivés, faites une petite machine à écrire.