Une méthode rapide ludique et efficace pour enseigner les base de la programmation par le codage du jeu Pong sous Processing

Damien MUTI

Prof. de physique-chimie-informatique Lycée Saint Exupéry – Marseille – Février 2017

1 Table des matières

Introduction

Pourquoi le jeu Pong ?

Ce document a pour but de présenter une méthode rapide, ludique et efficace pour faire comprendre et manipuler les notions de base de la programmation par le codage du jeu « Pong » sous Processing [1].La thèse que je propose est que l'enseignement de la programmation pour des novices n'est jamais plus efficace que lorsqu'elle s'appuie sur un rendu graphique et ludique. En effet, les dernières recherches en neurosciences [3] montrent que l'assimilation des connaissances et compétences dépend conjointement de l'attention et du temps passé aux apprentissages. Le jeu est un des moyens les plus efficaces de capter l'attention, ainsi que le rendu graphique.

Pour qui ?

Cette méthode est le fruit de cinq années d'expérimentation sur l'enseignement de l'informatique sur un public de designer issu d'une filière d'art appliqué n'ayant aucune formation poussée en science. Elle a d'abord été proposée à des étudiants en DSAA (Diplôme Supérieur d'Arts Appliqués), puis en BTS Design graphique en première et deuxième année depuis trois ans, et enfin à des élèves de lycée en terminale S spécialité ISN (Informatique et Science du Numérique) ainsi qu'en enseignement d'exploration ICN (Informatique et Création Numérique) en seconde.

Quel contenu ?

Compte tenu du faible nombre d'heures d'enseignement des sciences dont les étudiants disposent (1h/semaine) et des grandes attentes des étudiants en termes de rendu graphique, j'ai dû développer une méthode efficace pour transmettre le maximum de compétences en un minimum de temps. Les notions abordées autour du jeu PONG sont plus ciblées, moins exhaustives et en rapport direct avec les productions futures des étudiants :

- Prise en main de Processing :
    - dessins de formes élémentaires (cercles, rectangles, etc…).
    - zone de dessin : notion de pixel, de repère, d'image numérique.
    - Notion de couleur numérique (RVB, HSV, codage hexadécimal) via un outil pratique (colorselector)
- Structure d'un programme.
- Notion de boucle et de tests conditionnels.
- Notion de variables et de types associés.
- Notion de tableaux.
- Notion de fonction.
- Initiation à la programmation orientée objet :
    - Notion de classe d'objet (attributs, constructeur, méthodes),
    - Instanciation d'un objet.
    - Présentation de diverses librairies permettant l'interactivité et la manipulation d'objets puissants (vidéo, son, typographie, …)
- Conduite de projets individuels ou en groupes :
    - De l'idée à la réalisation concrète d'une œuvre interactive.
    - Répartition des taches dans un groupe

La notion de classe et de programmation orientée objet est fondamentale dans l'enseignement de l'informatique pour les étudiants et les élèves, car elle permet la manipulation de toutes les librairies développées sous Processing (et d'autre langages de manière générale) pour la manipulation de la vidéo, du son, de la typographie, le dialogue avec les cartes microcontrôleurs Arduino [2], la Kinect [9] etc. La manipulation de ces librairies offre des possibilités remarquables en termes de rendu graphique et d'interactivité.

Quels objectifs ?

Dans la suite du document, je propose une progression qui vise à amener une personne débutante en programmation à un niveau lui permettant de comprendre, manipuler et manipuler les notions de base de la programmation (boucles, tests conditionnels, et.) ainsi que des notions avancées de programmation orientée objet. Cependant, il est tout à fait possible de s'arrêter à n'importe quel moment de la progression en fonction du public concerné (seconde, terminale, BTS, DSAA).

Comment se servir de ce document ?

Le document présenté ici comporte un grand nombre de pages car tous les programmes présentés y sont incorporés. Il est construit de telle sorte que le lecteur soit guidé dans la modification d'un programme initial pour le faire évoluer. Chaque modification est présentée dans une couleur différente que le noir pour être visuellement attractive

A la fin de chaque partie, le programme final est aussi proposé. Il suffit d'effectuer un copier-coller du code et de le placer dans les différents onglets de l'interface de développement de Processing, en plaçant aussi les différents fichiers utiles au programme (sons, images, etc…) pour que ce dernier fonctionne.

1 Prise en main de Processing

1.1 Présentation du logiciel

1.1.1 Historique : un logiciel créé par les artistes, pour les artistes

Processing est un environnement de développement simple qui a été créé par les artistes, pour les artistes. Le projet initialement créé au MIT (« design by numbers ») en 2001 s'est développé considérablement depuis sa création. En effet, le logiciel est libre de droit et open-source, si bien que d'innombrables designer-programmeurs ont contribués à son essor en proposant des programmes générant une multitude d'objets graphiques, de nouvelles fonctionnalités (librairies) abordant aussi bien la typographie, que du matériel comme la kinect [9].

1.1.2 L'interface de développement (IDE)

Le langage de programmation utilisé est aussi très simple et s'apparente à du Java. L'interface de développement est aussi très intuitive et, dans les dernières versions permet une correction des lignes de codes en temps réels ainsi qu'un outil de débogage performant (voir *figure 1*).

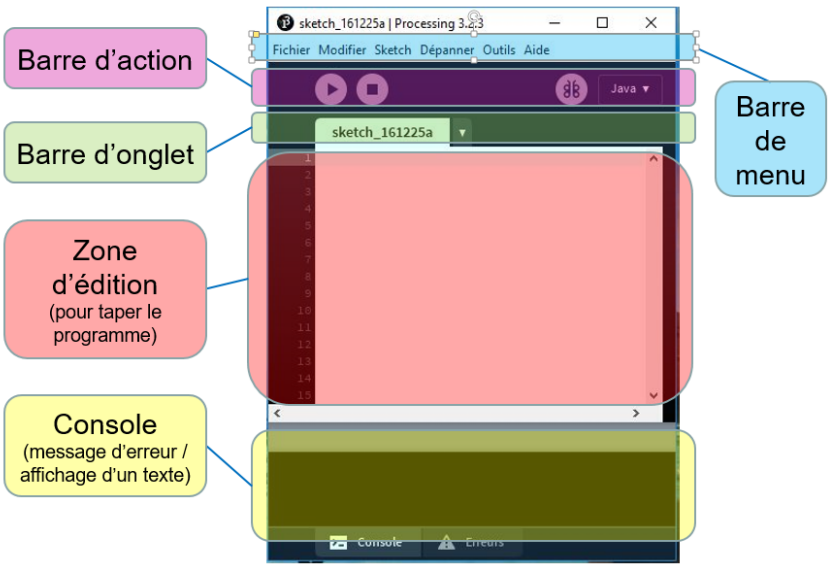
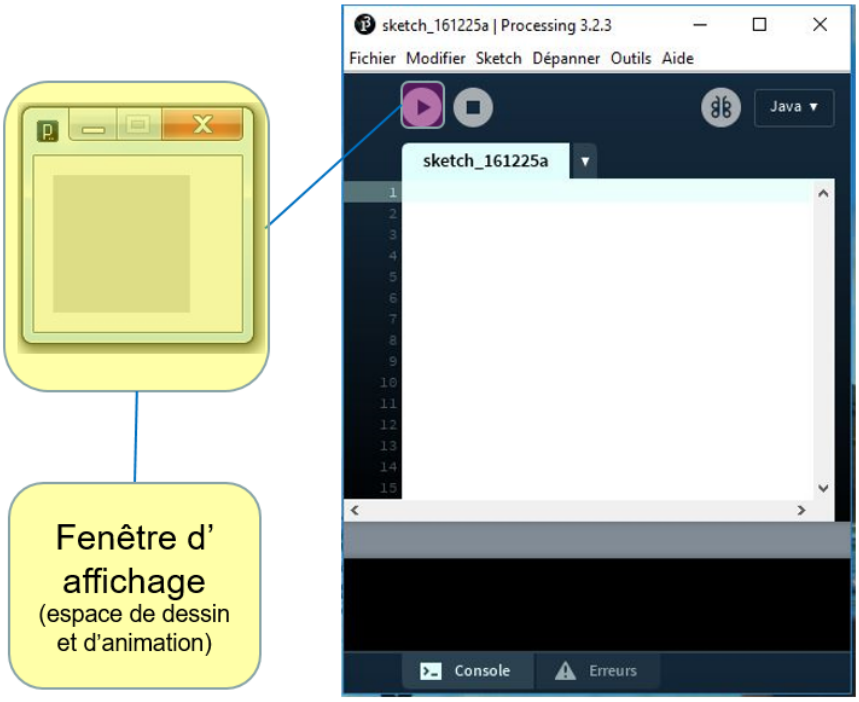


Figure 1- interface de développement de Processing

1.1.3 Un fonctionnement simple

Le rendu du programme s'obtient simplement en cliquant sur le bouton « play » :  .

Même si la zone d'édition est vide, le logiciel produit un rendu visuel : une image de 100×100 pixels sur un fond gris que l'on appellera « fenêtre de visualisation » :

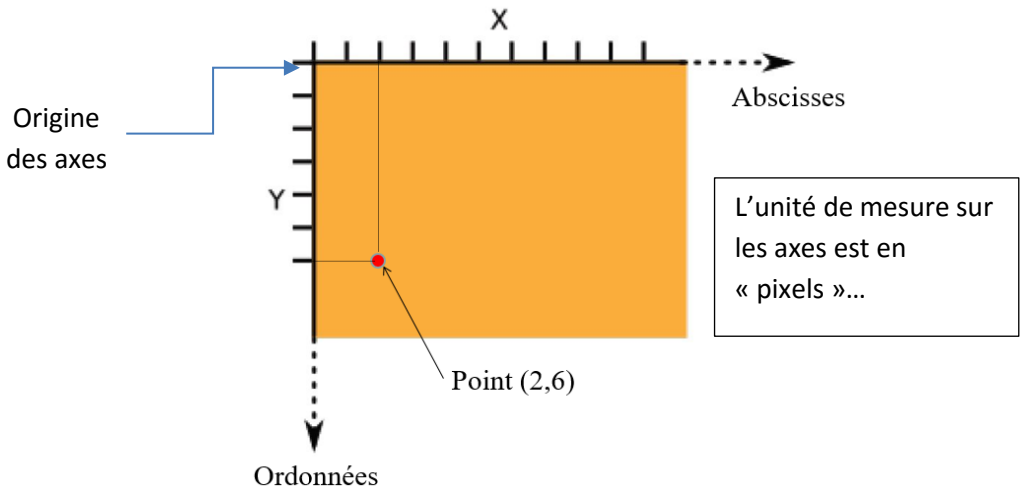


1.2 L'espace de dessin de la fenêtre d'affichage : structure d'une image numérique

Dans un premier temps, la fenêtre d'affichage représente une image numérique en couleur dans laquelle il est possible de dessiner toute sorte de formes qui se superposent comme lorsqu'on dessine sur feuille de papier ou sur un tableau. Il s'agit d'un espace en 2 dimensions.

Il est ensuite possible d'animer le contenu de la fenêtre d'affichage en modifiant le contenu de la fenêtre d'affichage 30 fois par seconde de la même manière qu'un film ou qu'un dessin animé.

Pour repérer la position des différentes formes dans l'image, il est nécessaire d'associer un repère (système d'axes et origine) à l'image numérique que constitue la fenêtre d'affichage. Par convention, l'origine se situe en haut à gauche de la fenêtre d'affichage, les abscisses sont orientées de la gauche vers la droite, et les ordonnées du haut vers le bas :



Remarque 1 : L'expérience montre que les élèves sont relativement perturbés par l'orientation de l'axe des ordonnées vers le bas, qui ne correspond pas à l'orientation vers le haut qui est généralement utilisé en Mathématique et Physique…

Remarque 2 : La fenêtre d'affichage permet de dessiner et d'animer des formes en 3D, mais nous nous contenterons ici d'utiliser l'espace 2D.

Remarque 3 : Il est ici possible de faire un rappel sur les images numériques : Echantillonnage spatial, matrice CCD, Image numérique en niveau de gris représenté par une matrice (tableau de nombres).

1.3 Premières instructions et premier dessin

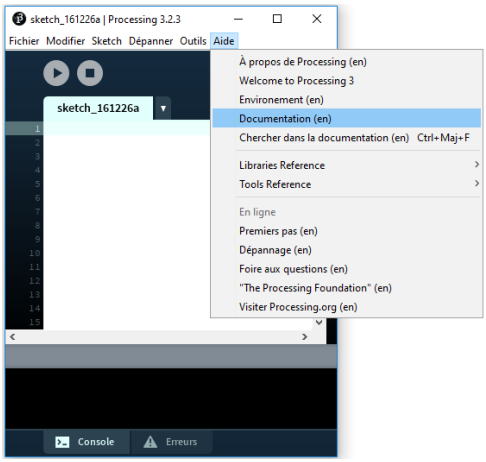
1.3.1 Objectifs

Les objectifs de cette partie sont les suivants :

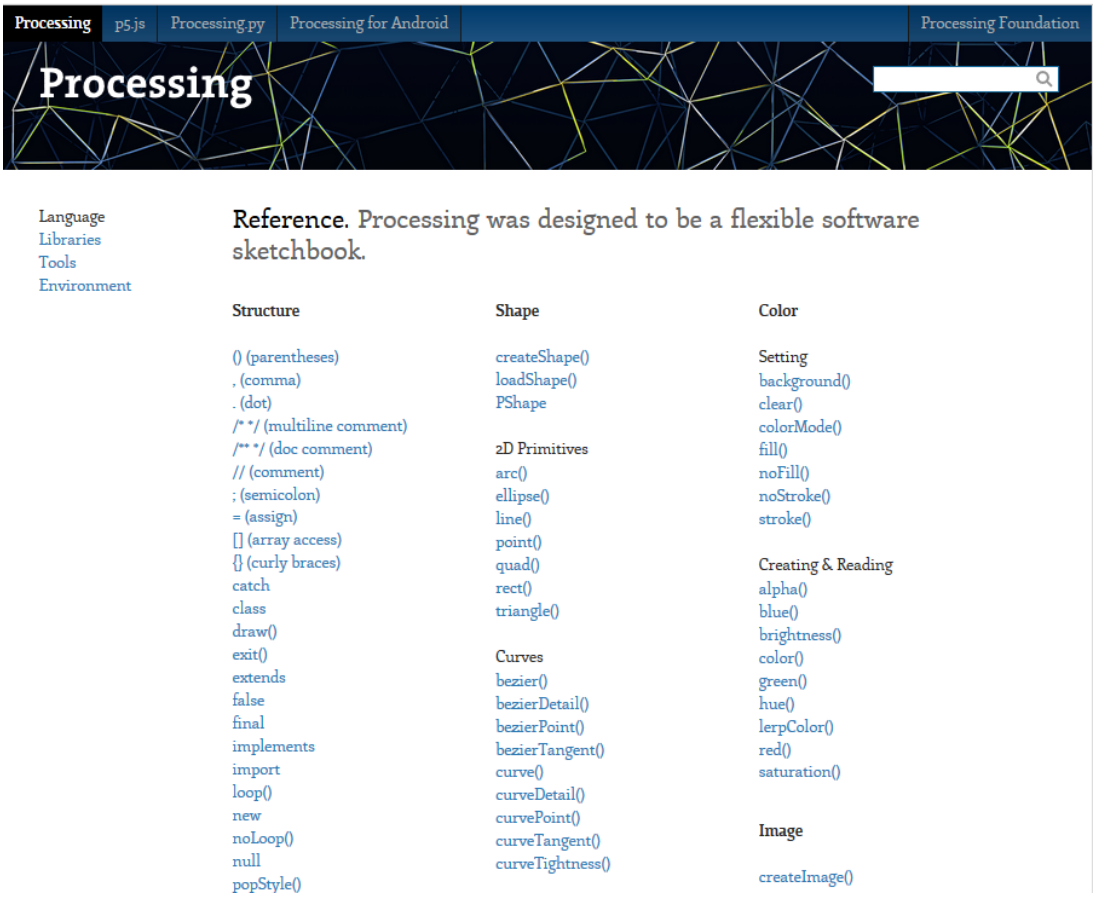
- comprendre l'intérêt et le fonctionnement de la documentation,
- modifier la taille de la fenêtre d'affichage,
- faire dessiner quelques formes simples dans la fenêtre d'affichage, en particulier un cercle qui modélise la balle de PONG,
- remplir la/les forme/s avec de la couleur,
- rappeler la façon dont est codée la couleur dans une image numérique.

1.3.2 Découverte de la documentation

Ouvrir la documentation de l'IDE : Aide> Documentation



Une page s'ouvre dans un navigateur internet :

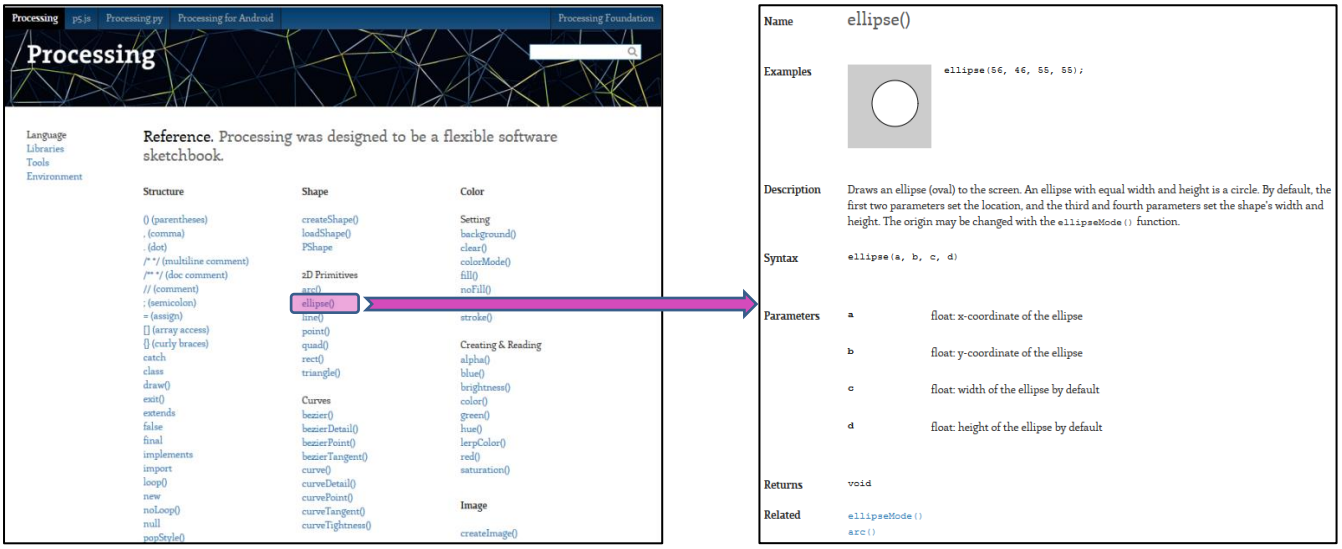


Montrer les différentes thématiques de la documentation qui est en anglais (en science l'anglais n'est pas une langue étrangère !!):

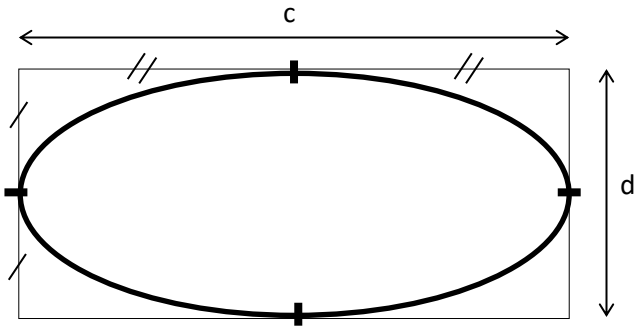
- Structure
- Shape
- Color
- Image
- Environnement
- Input
- Output
- Rendering
- Data
- Typography
- Maths
- Contol
- Transform
- Lights, Camera
- Constans

1.3.3 Utiliser la documentation en vue de dessiner une balle

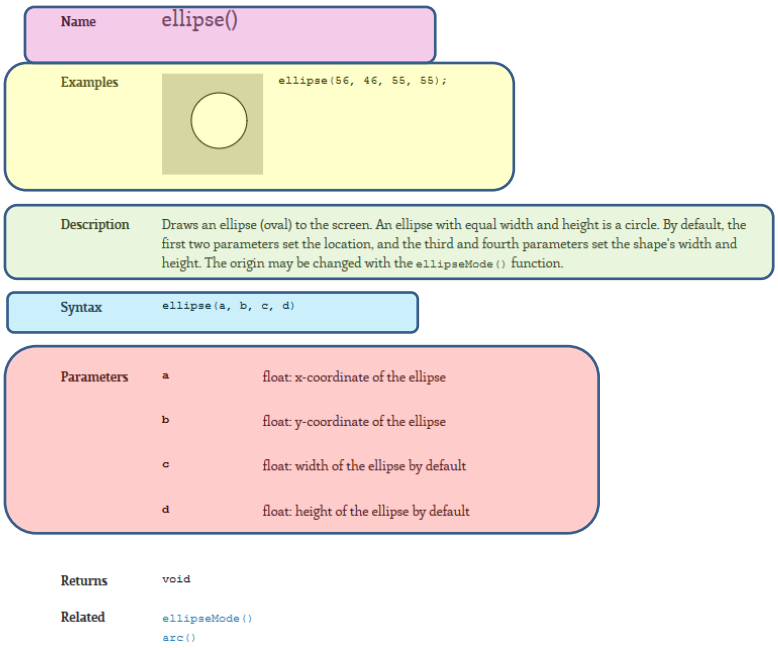
En vue de dessiner une balle, les élèves doivent dessiner un cercle. Amener les élèves à trouver dans la documentation les méthodes permettant de dessiner des formes simples (cercles, ellipses, rectangles, etc.) En particulier faire rechercher la méthode permettant de dessiner un cercle. Il s'agit de la méthode *ellipse()* dans Shape>2D Primitives :



Remarque 1 : On peut ici faire une remarque sur la manière de construire une ellipse. Il s'agit, par exemple, d'une courbe inscrite dans un rectangle de largeur c et de hauteur d. La courbe est tangente au milieu de chaque segment du rectangle.



Remarque 2 : On remarque que toutes les méthodes présentées dans la documentation sont construites sur le même modèle et comportent 7 parties : **Name, Exemples, Description, Syntax, Parameters, Returns, Related.**Pour l'instant, les parties les plus importantes sont les Exemples, la Description, la syntaxe et les paramètres d'entrée de la méthode :



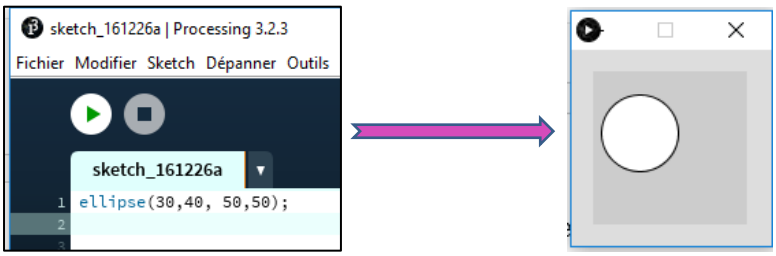
A ce stade, les élèves sont donc capables d'aller rechercher dans la documentation et comprendre le fonctionnement de toutes les méthodes de base proposées par Processing.

1.3.4 Dessiner une balle

Demander aux élèves de dessiner une balle dont les caractéristiques sont, par exemple les suivantes :

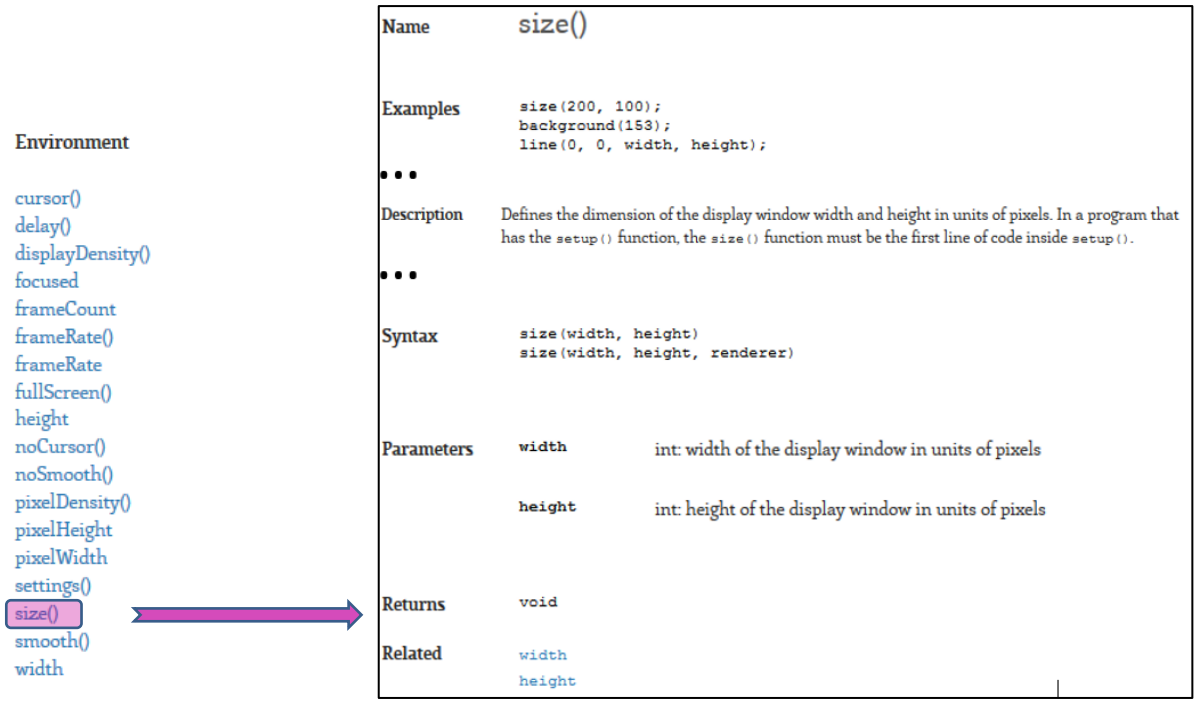
- diamètre 50 pixels,
- x = 30 pixels,
- y = 40 pixels.

L'instruction est la suivante :

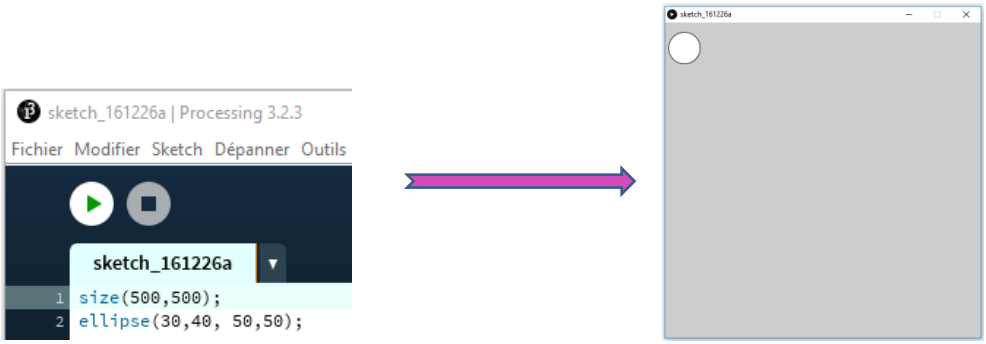


1.3.5 Modifier la taille de la fenêtre d'affichage

Faire rechercher dans la documentation les différentes fonctions appelées « méthodes » qui permettent de modifier la taille de la fenêtre d'affichage. Cette méthode se situe dans la partie « **Environnement** » dont les différentes méthodes permettent de gérer les paramètres d'affichage. Il s'agit de la méthode *size()* :



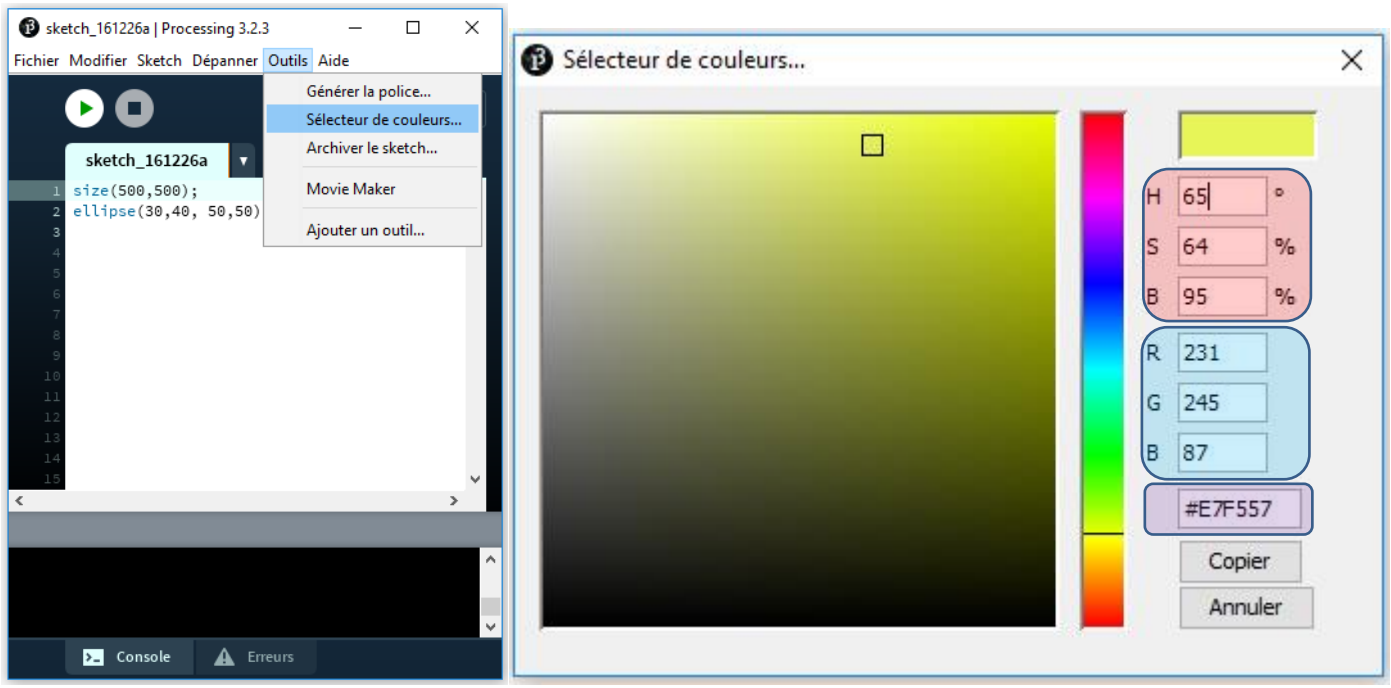
Dans toute la suite, nous travaillerons sur une fenêtre d'affichage de taille 500 × 500 pixels :



1.4 Gestion de la couleur

1.4.1 Le « Sélecteur de Couleur » : un outil pratique

Processing est doté d'un outil très pratique pour sélectionner les couleurs et comprendre ses différentes représentations numériques. Il s'agit du « Sélecteur de Couleur » :



Cet outil permet de visualiser très intuitivement les représentations de l'espace tridimensionnel des couleurs :

- (R,G,B) : Red, Green, Bleu. Voir encadré bleu ciel.
- (H, S, B) : Hue, Saturation, Brightness – Teinte, Saturation, Valeur. Voir encadré rouge.
- Le codage HTML représenté par 6 caractères hexadécimaux. Voir encadré violet.

Remarque 1 : On remarque que dans la représentation (R, G, B), chaque terme est compris entre 0 et 255. Chaque canal de couleur est donc codé sur un octet !!

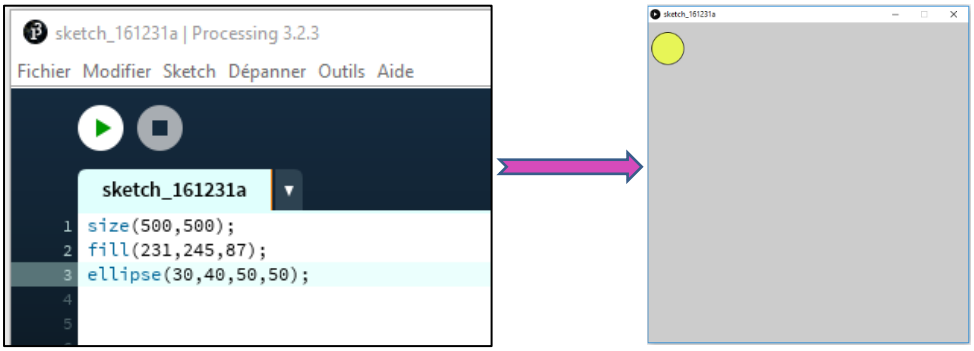
Remarque 2 : Le format hexadécimal qui code la couleur est compact et est représenté par un mot de 6 caractères. Les deux premier caractères, ici E7, représentent la valeur du rouge, soit R = 231. Les deux suivants, ici F5, représentent la valeur du vert, soit G = 245. Enfin, les deux derniers, ici 57, représentent la valeur du bleu, soit B = 87.

Remarque 3 : Il est ici possible d'effectuer un rappel sur :

- la représentation binaire des nombres,
- la notion de bit et d'octet,
- sur le fait qu'avec un octet il est possible de représenter 256 états possibles, c'est-à-dire de compter de 0 à 255,
- sur le passage de la base décimale à la base binaire et vice versa,
- sur la représentation hexadécimale des nombres.

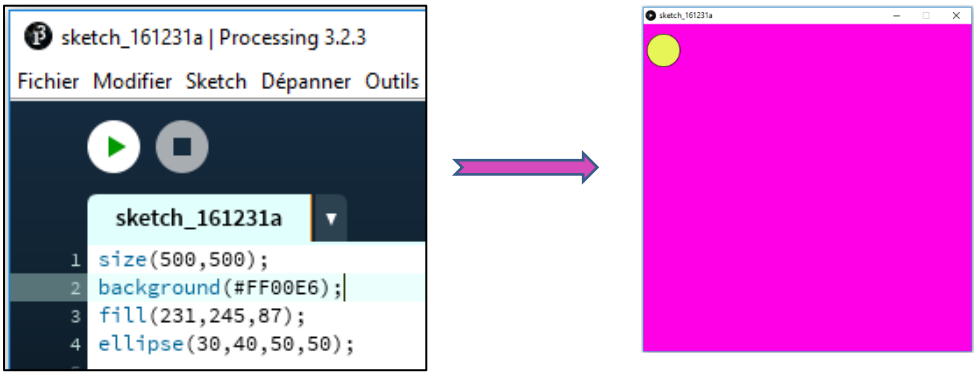
1.4.2 Modifier la couleur d'un objet

Faire rechercher dans la documentation la méthode qui permet de remplir une forme avec une couleur particulière.Ontrouve la méthode *fill()* qui doit être placée avant de dessiner la forme (voir dans la doc.).



1.4.3 Modifier la couleur du fond

Faire rechercher dans la documentation la méthode qui permet de remplir le fond de la fenêtre d'affichage avec une couleur particulière. On trouve la méthode *background()* qui doit être placée avant de dessiner la forme et avant l'appel à la méthode *fill()* (voir dans la doc.).



Copier/coller :

```
size(500,500) ;

background(#FF00E6) ;

fill(231,245,87) ;

ellipse(30,40,50,50) ;
```

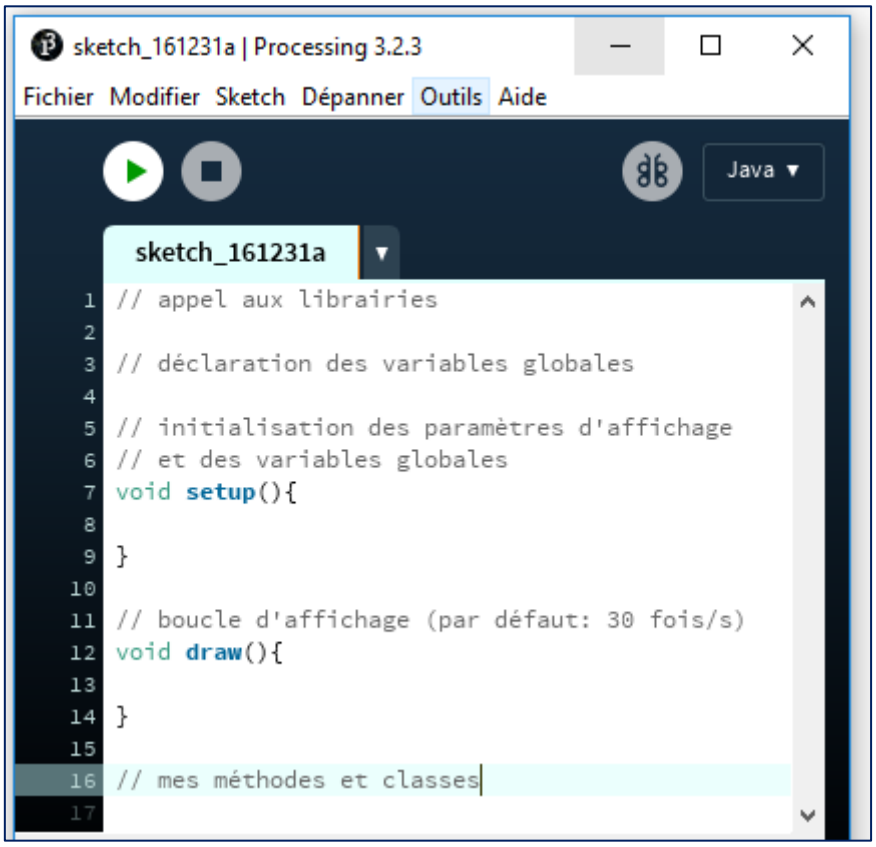
Remarque : Ici, on a utilisé la représentation hexadécimale de la couleur : #FF00E6. En utilisant le sélecteur de couleur on remarque que la teinte correspond aux valeurs suivantes (RVB) : R = 255, V = 0, B=230.

2 Mettre la balle en mouvement

Pour faire apparaître le mouvement de la balle il faut utiliser les méthodes de base proposées par Processing : *setup()* et *draw()*. Il convient alors de respecter la structure générale d'un programme sous Processing.

2.1 Structure générale d'un programme sous Processing

A partir de maintenant, il convient de respecter systématiquement la structure générale d'un programme sous Processing :



Copier/coller :

```
// appel aux librairies

// déclaration des variables globales

void setup(){  //initialisation des paramètres d'affichage et des variables globales

}

void draw(){// boucle d'affichage (par défaut: 30 fois/s)

}

// mes méthodes et classes
```

Remarque 1 : Nous définirons la notion de variable ultérieurement.

Remarque 2 : Tout le texte qui suit « // » est du commentaire et ne sera pas compilé (exécuté). Il est aussi possible de mettre du commentaire entre « /* » et « */ ».

2.1.1 *Méthode* setup()

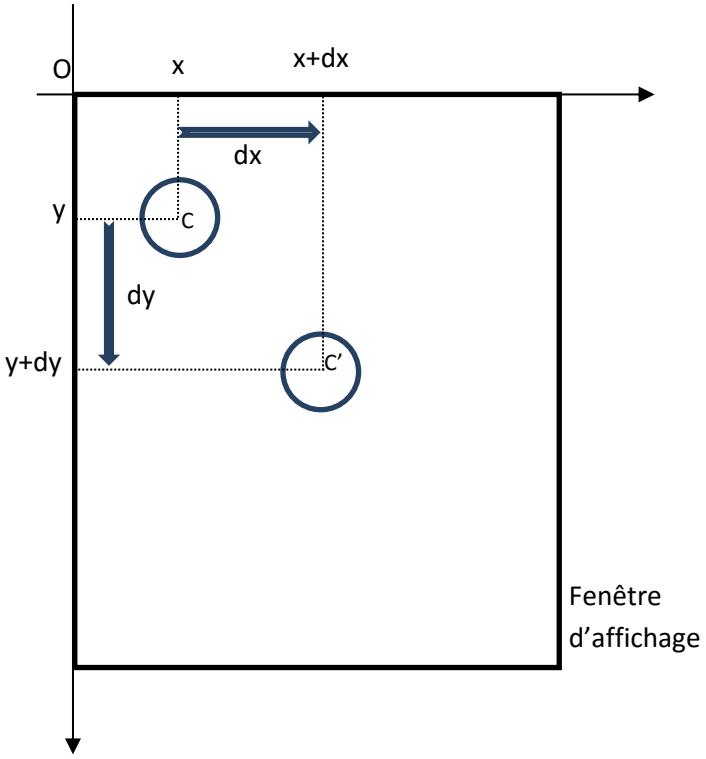
Cette méthode est exécutée une seule fois. Elle permet d'initialiser les paramètres d'affichage et d'initialiser les variables globales.

2.1.2 Méthode draw()

La méthode *draw()* est exécutée continuellement en boucle tant que le sketch s'exécute. Toutes les instructions contenues dans l'encapsulation {} de cette méthode sont exécutées de façon itérative par défaut 30 fois par seconde. Il est possible de régler la vitesse d'exécution grâce à la méthode *frameRate()* qui doit être appelée dans la méthode *setup()*.

2.2 Comment caractériser la balle ?

Pour pouvoir gérer de façon automatique la balle, il convient de définir l'ensemble de ses caractéristiques que l'on appellera par la suite « attributs » de la balle. Faire réfléchir les élèves sur la manière de caractériser une balle.
Dessiner au tableau : à un instant t, et à t+dt (l'image/frame suivante)



Attributs de la balle :

- Couleur : c
- Diamètre : d
- Position :
  - Abscisse : x
  - Ordonnée : y
- Déplacement :
  - Déplacement suivant x : dx
  - Déplacement suivant y : dy
- ~~Forme : cercle~~
  (Pour l'instant, nous n'allons pas considérer la forme comme un paramètre variable)

Remarque 1 : On peut aussi définir un vecteur position $\vec{p} = \begin{vmatrix} x \\ y \end{vmatrix}$ ainsi qu'un vecteur déplacement $\vec{d} = \begin{vmatrix} dx \\ dy \end{vmatrix}$. On considère dans un premier temps deux paramètres scalaires au lieu d'un vecteur.

Remarque 2 : Nous définissons volontairement les paramètres de la balle comme « attributs » de la balle en prévision de la partie concernant la Programmation Orientée Objet, dans laquelle nous allons définir les attributs de la classe « balle ». L'assimilation et la compréhension du concept d'attribut est ainsi facilité.

2.3 Notion de variable

L'ensemble des attributs de la balle {c, r, x,y, dx, dy} est susceptible d'évoluer au fur et à mesure de l'animation et de la vie de la balle. Il faut donc que le programme puisse manipuler et faire varier chacun de ces paramètres au fur et à mesure de son exécution.

Pour effectuer cette manipulation, en programmation, on utilise des **variables**. Il s'agit d'un espace mémoire dans l'ordinateur attribué à chacun des attributs de la balle. On peut assimiler cet espace à une boite dans laquelle on place la valeur de l'attribut :

La boite en question porte **un nom** pour la repérer de manière unique en mémoire (ici : x) et prend une valeur (ici 5).

En physique, chaque grandeur a intrinsèquement une unité. Par exemple :

- ma masse m est en kg,
- la vitesse v est en $m.s^{-1}$.
- Etc.

De la même manière, en programmation, chaque variable a un **type**. Dans le langage Processing, hérité du JAVA, il existe des **types primitifs** :

- ***int***: nombre entier codé sur 4 octets
- ***float*** : nombre à virgule, codé sur 4 octets
- ***char*** : caractère ascii
- ***boolean*** : booléen (vrai ou faux), codé sur un bit. Il s'agit de la variable occupant le moins de place mémoire.
- etc.

Les types primitifs de Java se résument dans le tableau suivant :

Type	*description*	*taille*	*ensemble de valeurs*	*val. init.*
Nombres entiers				
`byte`	Octet	8 bits	de -128 à 127	0
`short`	Entier court	16 bits	de $-32\,768$ à $32\,767$	
`int`	Entier	32 bits	de $-2\,147\,483\,648$ à $2\,147\,483\,647$	
`long`	Entier long	64 bits	de -2^{63} à $2^{63}-1$	
Nombres flottants				
`float`	Flottant simple précision	32 bits	de $-3,4028235\times10^{+38}$ à $-1,4\times10^{-45}$, 0 et de $1,4\times10^{-45}$ à $3,4028235\times10^{+38}$	0.0
`double`	Flottant double précision	64 bits	de $-1,7976931348623157\times10^{+308}$ à $-4,9\times10^{-324}$, 0 et de $4,9\times10^{-324}$ à $1,7976931348623157\times10^{+308}$	
Autres types				
`char`	Caractère	16 bits	Tous les caractères Unicode	`'\u0000'`
`boolean`	Valeur booléenne	1 bit	`false`, `true`	`false`

Il existe aussi des **types complexes** permettant de manipuler des objets tels que la vidéo, le son, les chaines de caractères. La place mémoire occupée par ces variables complexe dépend du contenu de la variable.

Bilan : Chaque variable est caractérisée par :

- **un nom**
- **un type**
- **une place mémoire**

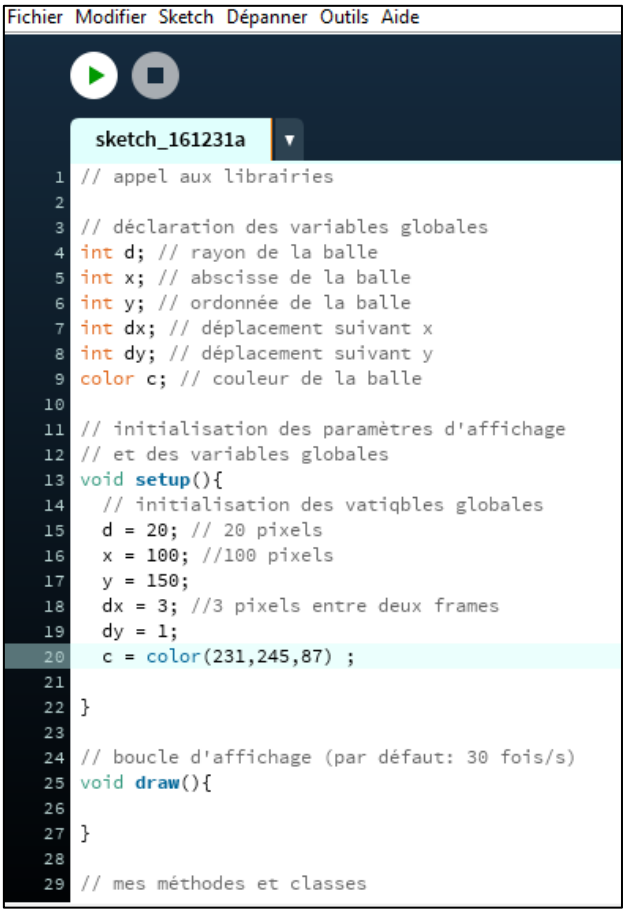
Remarque : En mathématique le concept de variable est bien connu lorsqu'on étudie les fonctions du type y=f(x). Les variables x et y sont alorsintroduites.

2.4 Les variables associées aux attributs de la balle

Les variables associées aux attributs de la balle et leur type peuvent être résumées dans ce tableau :

Nom	Variable	Type
Diamètre	d	*int*
position	x	*int*
	y	*int*
déplacement	dx	*int*
	dy	*int*
couleur	c	*color*

Dans la programmation Processing, il convient tout d'abord de déclarer les attributs de la balle en tant **que variables globales**, et ensuite de les initialiser dans la méthode *setup()*. Le début du programme est donc le suivant :



Copier/coller :*Programme 1*

```
// appel aux librairies


// déclaration des variables globales
// attributs de la balle
int d; // rayon de la balle
int x; // abscisse de la balle
int y; // ordonnée de la balle
int dx; // déplacement suivant x
intdy; // déplacement suivant y
color c; // couleur de la balle


void setup(){// initialisation des paramètres d'affichage et des variables globales
  // initialisation des variables globales
  d = 20; // 20 pixels
  x = 100; //100 pixels
  y = 150;
  dx = 3; //3 pixels entre deux frames
dy = 1;//1 pixels entre deux frames
  c = color(231,245,87) ;
}
// boucle d'affichage (par défaut: 30 fois/s)
void draw(){
}
// mes méthodes et classes
```

Remarques : Les valeurs des attributs ont été prises au hasard.

Remarque 2 : la méthode *color()* permet d'initialiser la couleur. Voir dans la documentation.

2.5 Quelles sont les actions associées à la balle ?

2.5.1 Discussion

Dans cette partie, nous devons discuter avec les élèves des différentes actions que peut réaliser la balle. Faire émerger les actions suivantes :

- **Afficher** : Cela paraît trivial, mais il est tout d'abord nécessaire de dessiner la balle pour qu'elle apparaisse sur la fenêtre d'affichage. Il faut alors un cercle de diamètre d, à la position (x,y), rempli avec la couleur c.
- **Avancer ou Bouger** : Faire avancer la balle de dx suivant l'axe x et de dy suivant l'axe y, entre deux frames consécutives.
- **Rebondir ou Test de collision**: Faire rebondir la balle sur les bords de la fenêtre d'affichage. Par la suite, nous ferons évoluer cette action en permettant la collision avec une raquette. Nous pourrons aussi rajouter du son à chaque collision.

Nous allons maintenant programmer ces trois actions en complétant le *programme 1* précédent.

2.5.2 Afficher la balle

On réinvestit le programme vu dans la section 1.4.3 précédente :

- On place les lignes de codes relatives aux paramètres d'affichage (*size()* et *background()*) dans la méthode *setup()*.
- On place les lignes de code relative au dessin de la balle dans la méthode *draw()* qui gère le dessin dans la fenêtre d'affichage et l'animation.

Copier/coller :

```
// appel aux librairies
// déclaration des variables globales
// attributs de la balle
int d; // rayon de la balle
int x; // abscisse de la balle
int y; // ordonnée de la balle
int dx; // déplacement suivant x
intdy; // déplacement suivant y
color c; // couleur de la balle
void setup(){// initialisation des paramètres d'affichage et des variables globales
// paramètresd'affichage
size(500,500) ;
background(#FF00E6) ;
// initialisation des variables globales
d = 20; // 20 pixels
x = 100; //100 pixels
y = 150;
dx = 3; //3 pixels entre deux frames
dy = 1;//1 pixels entre deux frames
c = color(231,245,87) ;
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
// affichage de la balle
fill(231,245,87) ;
   ellipse(x,y,d,d) ;
}
// mes méthodes et classes
```

2.5.3 Faire avancer la balle

Entre chaque frame la balle doit se déplacer de dx suivant l'axe x et dy suivant l'axe y. Il faut donc rajouter dans la boucle *draw()* les instructions suivantes :

x ← x+dx

y ← y+dy

la méthode *draw()* devient donc :

```
// boucle d'affichage (par défaut: 30 fois/s)
void draw(){
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
  // faire avancer la balle
  x = x + dx ;
  y = y + dy
}
```

En lançant le sketch nous obtenons :

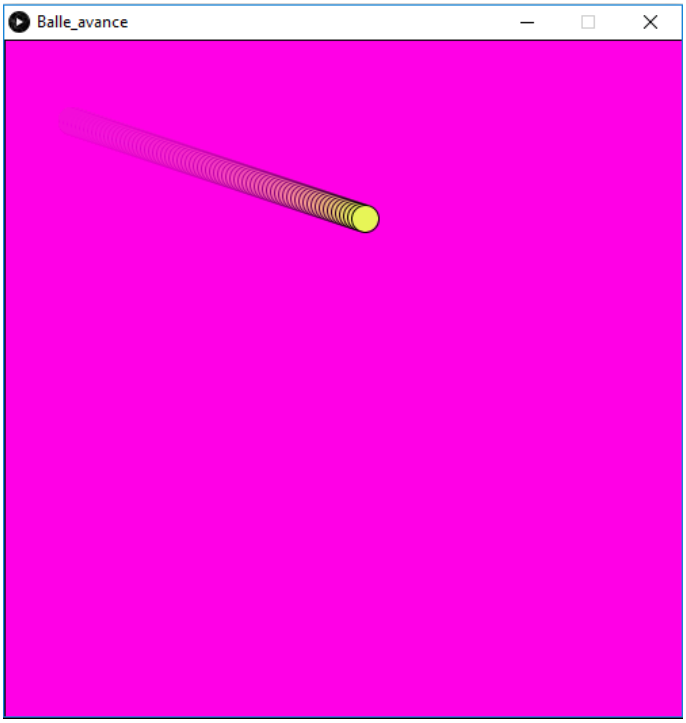
Remarque 1 : Nous remarquons que la balle laisse la trace de toutes les positions successive au cours du mouvement. Pour effacer les traces, il faut dessiner un rectangle de la taille de la fenêtre (*width, height*), remplie avec la couleur du fond (ici c = #FF00E6) avant de dessiner la nouvelle balle. Le dessin d'un rectangle s'effectue avec la méthode *rect()*, (voir la doc.) Il est aussi possible de rajouter de la transparence (par exemple 10%) au rectangle pour que la trace

s'efface progressivement et que l'effet visuel soit plus esthétique. Ceci s'effectue en rajoutant un paramètre spécifique (canal alpha) dans la méthode *fill()*, (voir la doc.)

Nous rajoutons donc les lignes de code suivant dans la méthode *draw()* :

```
// boucle d'affichage (par défaut: 30 fois/s)
void draw(){
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
// faire avancer la balle
  x = x + dx ;
  y = y + dy
}
```

On obtient l'effet suivant :



Remarque 3 : Il existe des opérateurs d'assignation qui permettent de simplifier l'écriture x = x+dx et y=y+dy. Ils se résument dans le tableau suivant :

Attribution/ modification d'une valeur	incrémentation	décrémentation	Incrémenter et décrémenter de 1	Multiplication ou division par un facteur
=	+=	-=	++ et --	*= ou /=

La partie « faire avancer la balle » du code précédent peut se remplacer par :

```
  // faire avancer la balle
x += dx ;
y += dy ;
}
```

Remarque 2 : La balle ne rebondit pas encore sur les bords. C'est l'objet de la partie suivante…

2.6 Rebond sur les bords : Tests conditionnel « if »

2.6.1 Analyse du problème et algorithme de rebond pour le bord vertical droit(x = width)

Convention de déplacement suivant x

D'après l'orientation des axes (voir partie 1.2) :

- si la balle se déplace de gauche à droite, alors dx est positif,
- si la balle se déplace de droite à gauche, alors dx est négatif,

Condition de rebond

L'algorithme est simple :

> ***Si*(la balle touche le bord)**
>
> ***alors***
>
> **(le sens de déplacement change de signe)(on peut dire que la balle va en sens inverse)**

Mathématiquement, le changement de signe de dx revient à multiplier cette valeur par -1. On écrit :

dx ← dx * (-1)

La ligne de code peut s'écrire de deux manières :

```
dx = dx *(-1) ;
```

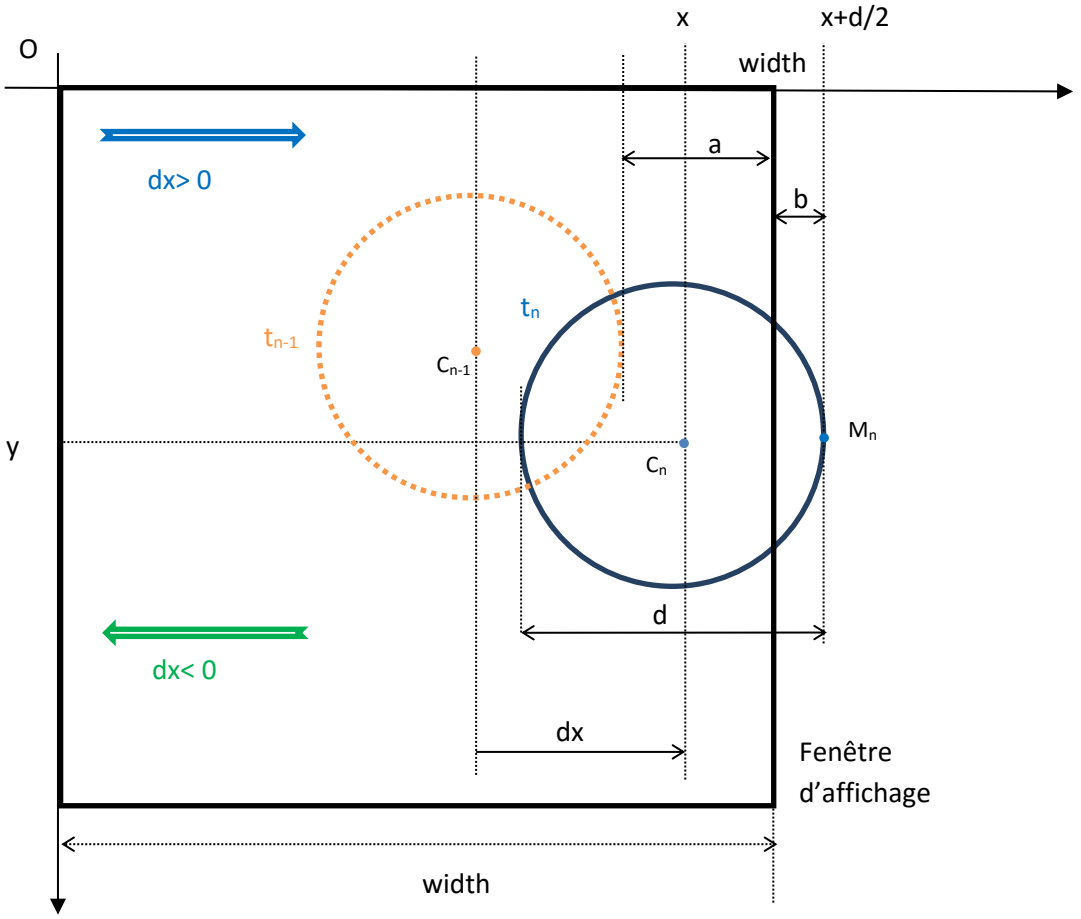
ou bien :

```
dx*= (-1) ;
```

Il Reste à trouver à quel moment la balle touche le bord. En réalité, le bord de la balle ne pourra quasiment jamais être rigoureusement tangent au bord vertical droit de la fenêtre d'affichage. En effet, comme le déplacement dx est de quelques pixels :

- à la frame d'avant la collision (instant t_{n-1}), le bord droit de la balle (point M_{n-1}) se situe à une distance a du bord de la fenêtre d'affichage, telle que $0 \leq a < dx$,
- et à l'instant d'après la collision (instant t_n), le bord droit de la balle (point M_n) se situe à une distance b du bord de la fenêtre d'affichage, telle que $0 \leq b < dx$.

Ceci est résumé dans la figure suivante :



Dans la figure suivante, on remarque qu'à l'instant t_n, lors de la collision, le point M_n d'abscisses x+d/2 dépasse le bord droit de la fenêtre d'affichage représenté par la droite d'équation x=*width*. **On considère que la balle entre en collision avec le bord lorsque la condition :**

(x+d/2 >width)

est vérifiée.

Algorithme de la collision sur le bord vertical droit de la fenêtre d'affichage

L'algorithme de collision sur le bord droit est donc le suivant :

***Si* la condition (x+d/2 >width) est vraie**
alors
dx ← dx * (-1)

Programme Processing

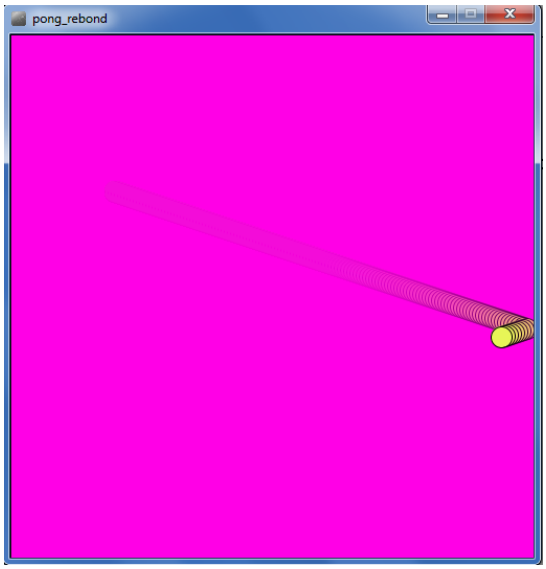
On peut alors programmer cet algorithme dans la méthode *draw()* à la suite de l'action « afficher la balle » et « faire avancer la balle » :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
// faire avancer la balle
x+= dx ;
y+= dy ;
  // Test de collision suivant x
  if ( x+d/2 >width){
dx  *= (-1) ; // changer de sens
  }
}
```

Remarque 1: Dans le test « if », la condition (x+d/2 >width) est évaluée et renvoie un **booléen** qui peut prendre la valeur « Vrai » ou « Faux ». Si la condition évaluée renvoie « vrai », alors un ensemble d'instructions encapsulées entre « { » et « } » sont effectuées. Ici, une seule instruction est réalisée : dx = dx*(-1).

Remarque 2: Pour permettre une meilleure lecture du code, le bloc d'instruction dans le test *if* doit toujours être bien **indenté**, c'est-à-dire décalé d'un espace notable par rapport à la position du *if*.

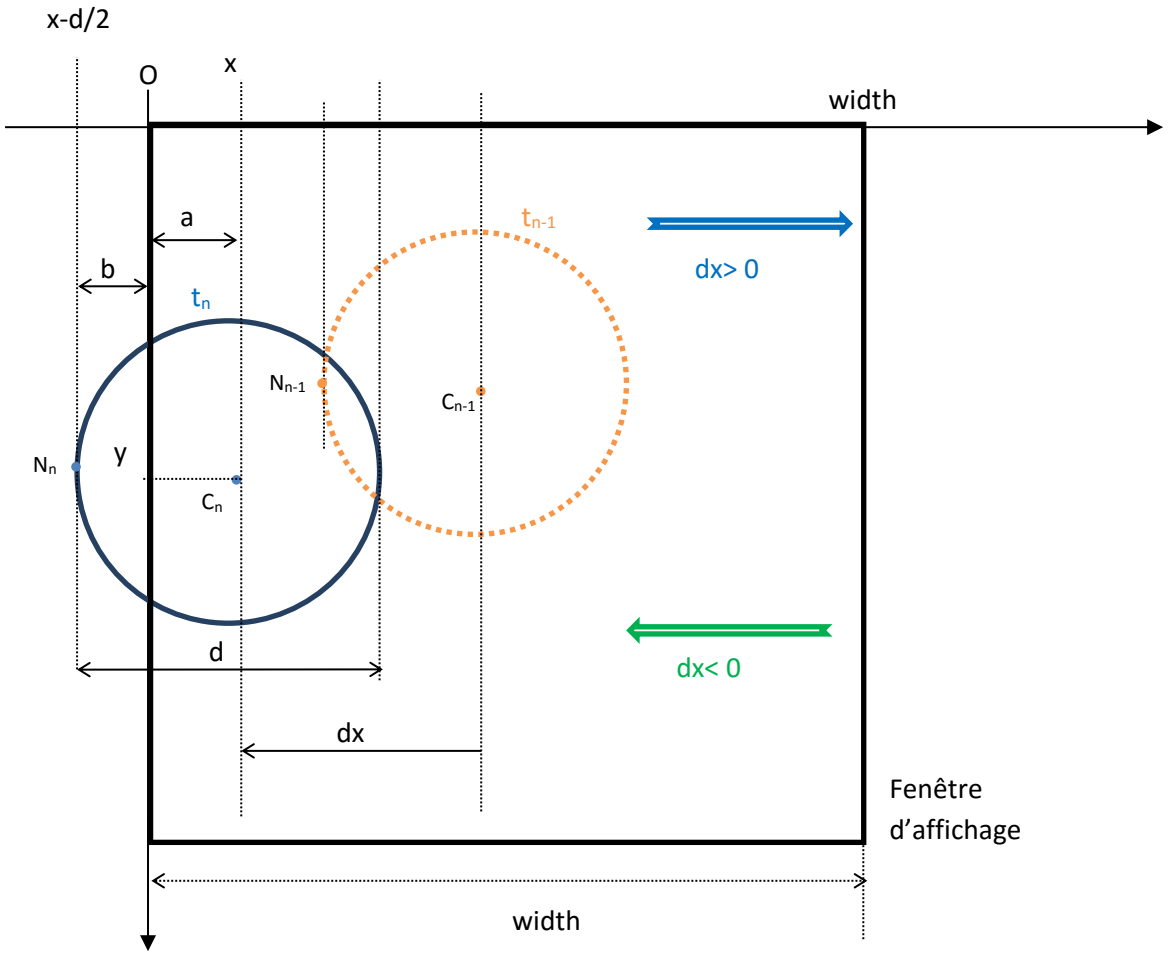
On obtient l'animation suivante :



2.6.2 Analyse du problème et algorithme de rebond pour le bord vertical gauche (x = 0)

Comme pour la collision sur le bord droit :

- à la frame d'avant la collision (instant t_{n-1}), le bord gauche de la balle (point N_{n-1}) se situe à une distance a du bord de la fenêtre d'affichage, telle que $0 \leq a < dx$,
- et à l'instant d'après la collision (instant t_n), le bord gauche de la balle (point N_n) se situe à une distance b du bord de la fenêtre d'affichage, telle que $0 \leq b < dx$.



Dans la figure suivante, on remarque qu'à l'instant t_n, lors de la collision, le point N_n d'abscisses x-d/2 dépasse le bord gauche de la fenêtre d'affichage représenté par la droite d'équation x=0. **On considère que la balle entre en collision avec le bord lorsque la condition :**

(x-d/2 <0)

est vérifiée.

Algorithme de la collision sur le bord vertical gauche de la fenêtre d'affichage

L'algorithme de collision sur le bord gauche est donc le suivant :

Si **la condition (x - d/2 < 0) est vraie**
alors
dx ← dx * (-1) (changement de sens)

Algorithme final de la collision sur les bords verticaux gauches et droits de la fenêtre d'affichage

On peut regrouper dans le test conditionnel if les deux conditions de collision sur le bord droit et gauche :

Si **les conditions(x+d/2 >width)** ***ou*** **(x - d/2 < 0)sont vraies**
alors
dx ← dx * (-1) (changement de sens)

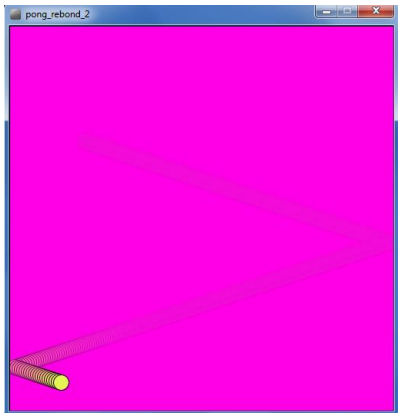
Programme Processing

Pour coder cet algorithme, il suffit de modifier le test conditionnel *if* dans le programme précédent, en y rajoutant une deuxième condition. Sous Processing (Java), le « ou logique » se note « || » :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
// faire avancer la balle
x+= dx ;
y+= dy ;
```

```
  // Test de collision suivant x
if ( (x+d/2 >width) || (x - d/2 < 0)){
dx  *= (-1) ;// changer de sens
}
}
```

On obtient l'animation suivante :



2.6.3 Rebond sur les bords horizontaux

Tout le raisonnement effectué pour la collision sur les bords verticaux se transpose intégralement par analogie sur les bords horizontaux. Dans les deux lignes de code correspondant à l'action « Test de collision suivant x », il suffit d'effectuer les changements de variable suivants :

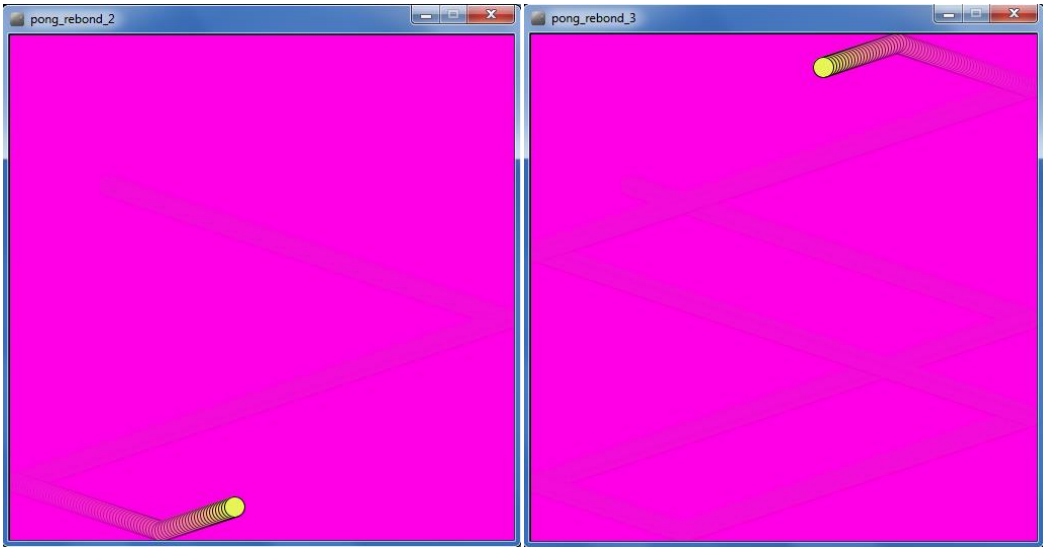
Axe x		Axe y
x	↔	y
dx	↔	dy
width	↔	height

Le code Processing correspondant au test de collision suivant y est donc le suivant :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
// faire avancer la balle
x+= dx ;
y+= dy ;
```

```
  // Test de collision suivant x
if ( (x + d/2  > width ) || ( x - d/2 < 0 ) ) {
dx  *= (-1) ; // changer de sens
  }
// Test de collision suivant y
if ( (y + d/2  > height ) || ( y - d/2 < 0 ) ) {
dy  *= (-1) ; // changer de sens
  }
}
```

On obtient l'animation suivante :



Le programme final est le suivant : **Programme 1**

```
// appel aux librairies
// déclaration des variables globales
// attributs de la balle
int d; // rayon de la balle
int x; // abscisse de la balle
int y; // ordonnée de la balle
int dx; // déplacement suivant x
intdy; // déplacement suivant y
color c; // couleur de la balle
void setup(){// initialisation des paramètres d'affichage et des variables globales
```

```
// paramètres d'affichage
size(500,500) ;
background(#FF00E6) ;
// initialisation des variables globales
d = 20; // 20 pixels
x = 100; //100 pixels
y = 150;
dx = 3; //3 pixels entre deux frames
dy = 1;//1 pixels entre deux frames
c = color(231,245,87) ;
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
  // affichage de la balle
fill(231,245,87) ;
  ellipse(x,y,d,d) ;
// faire avancer la balle
x+= dx ;
y+= dy ;
  // Test de collision suivant x
if ( (x + d/2  > width ) || ( x - d/2 < 0 ) ) {
dx  *= (-1) ; // changer de sens
  }
    // Test de collision suivant y
if ( (y + d/2  > height ) || ( y - d/2 < 0 ) ) {
dy  *= (-1) ; // changer de sens
  }
}
```

2.6.4 Bilan sur les opérateurs permettant les tests conditionnels

Lors des différents tests conditionnels, nous avons introduits un certain nombre **d'opérateurs de comparaison** (relationnels) qui peuvent se résumer dans le tableau suivant :

Comparaison entre deux variables de même type (renvoie un booléen)				
supérieur	Inférieur	Supérieur ou égale	Inférieur ou égale	Égalité
>	<	>=	<=	==

Nous avons introduits aussi un certain nombre d'opérateurs arithmétique :

Addition	Soustraction	Multiplication	Division	Modulo (reste d'une division euclidienne)
+	-	*	/	%

Pour plus d'information sur l'opérateur modulo « % », voir la documentation de Processing ou la documentation en ligne : https://processing.org/reference/modulo.html .

Enfin nous avons introduits des opérateurs d'assignation :

Attribution/ modification d'une valeur	incrémentation	décrémentation	Incrémenter et décrémenter de 1	Multiplication ou division par un facteur
=	+=	-=	++ et --	*= ou /=

Pour plus d'information sur ces opérateurs, voir la documentation de Processing ou la documentation en ligne.

3 Evolution du code vers un jeu multi-balle : notion de méthode et de tableau

3.1 Présentation du problème : pourquoi utiliser des méthodes, des tableaux et des boucles itératives ?

Supposons qu'on souhaite maintenant faire évoluer plusieurs balles dans la fenêtre d'affichage. Chaque balle doit avoir un ensemble d'attributs et de variables associées qui lui est propre. Actuellement, chaque balle dispose de 6 attributs qui la caractérisent de manière unique. Si on décide de faire évoluer 10 balles, cela revient à manipuler 6*10 = 60 variables différentes. De même, chaque balle doit avoir des actions propres (afficher, avancer, test de collision). Il faudrait donc dupliquer ces actions pour chaque balle.

Une méthode grossière consiste à copier chaque partie du code réalisé pour la première balle et à l'adapter pour chaque balle. Cela conduirait à un code très long, répétitif et fastidieux à lire.

Une des solutions pour remédier à ce problème est :

- l'utilisation de « fonctions d'action » que l'on nommera « **méthodes** d'action »,
- l'utilisation des **tableaux** de variables.

Nous introduirons enfin la **« boucle *for*** » qui permet d'effectuer de manière récursive des taches répétitives qui doivent s'exécuter pour chaque balle.

Nous introduisons dans un premier temps la notion de méthode pour la balle unique créée dans la partie 2. Nous introduirons ensuite la notion de tableaux et de boucle for pour gérer les balles multiples.

3.2 Création des méthodes pour les différentes actions d'une balle unique

3.2.1 Les méthodes associées à une balle unique

Lorsque la balle évolue dans la fenêtre d'affichage, il serait pratique que pour chaque itération de la boucle *draw()* le programme exécute des actions élémentaires du type :

Pour chaque itération de la boucle draw() :
Afficher la balle
Avancer
Tester la collision

Il suffit pour cela d'incorporer les différentes lignes de codes écrites pour chaque actions dans la boucle *draw()* dans des méthodes indépendantes qui peuvent être appelées à n'importe quel instant.

De manière générale, une méthode peut avoir des paramètres en entrée et renvoyer des paramètres en sortie. Dans le cas présent, dans le cas d'une balle unique, on pourra définir les méthodes :

- afficher() : permet de dessiner la balle
- avancer() : permet de faire avancer la balle suivant x et y.
- testCollision() : test la collision sur les bords.

A priori, chacune de ces méthodes ne prend aucun paramètre en entrée et ne renvoie aucun paramètre en sortie.

Remarque : Les méthodes suivantes :

- void setup() et
- void draw(),

ne prennent aussi aucun paramètre en entrée et ne retournent aucun paramètre en sortie. Elles permettent de d'exécuter un certain nombre d'actions d'initialisation ou d'animation. Le terme « *void* », qui veut dire « vide » en anglais, signifie que la fonction ne renvoie aucun paramètre en sortie.

3.2.2 Construction des méthodes *afficher(), avancer() et testCollision()*

Toutes les méthodes personnelles sont écrites en dessous de la méthode *draw()*, dans la partie intitulée « Mes méthodes et classes personnelles ». Par exemple, pour la méthode *afficher()* :

- On écrit la méthode *void afficher() { }*
- On déplace les lignes de codes correspondant à la partie « affichage de la balle », à l'intérieur des accolades d'encapsulation de la méthode *afficher().*

On effectue le même processus pour les méthodes *avancer() et testCollision()* :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
   // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
   // affichage de la balle
fill(231,245,87) ;
   ellipse(x,y,d,d) ;
// faire avancer la balle
x += dx ;
y += dy ;
   // Test de collision suivant x
if ( (x + d/2  > width ) || ( x - d/2 < 0 ) ) {
dx  *= (-1) ; // changer de sens
   }
     // Test de collision suivant y
if ( (y + d/2  > height ) || ( y - d/2 < 0 ) ) {
dy  *= (-1) ; // changer de sens
   }
}
// Mes méthodes et classes personnelles
void afficher(){

}
void avancer(){

}
void testCollision(){

}
```

Le code final est le suivant : **Programme 2**

```
// appel aux librairies

// déclaration des variables globales

// attributs de la balle

int d; // rayon de la balle

int x; // abscisse de la balle

int y; // ordonnée de la balle

int dx; // déplacement suivant x

intdy; // déplacement suivant y

color c; // couleur de la balle

void setup(){// initialisation des paramètres d'affichage et des variables globales

// paramètres d'affichage

   size(500,500) ;

   background(#FF00E6) ;

// initialisation des variables globales

d = 20; // 20 pixels

x = 100; //100 pixels

y = 150;

dx = 3; //3 pixels entre deux frames

dy = 1;//1 pixels entre deux frames

c = color(231,245,87) ;

}

void draw(){// boucle d'affichage (par défaut: 30 fois/s)

   // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle

fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)

rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine

// affichage de la balle

afficher() ;

// faire avancer la balle

avancer() ;
```

```
  // Test de collision sur les bords
testCollision() ;
}
// Mes méthodes et classes personnelles
void afficher(){
// affichage de la balle
  fill(231,245,87) ;
  ellipse(x,y,d,d) ;
}
void avancer(){
// faire avancer la balle
x += dx ;
y += dy ;
}
void testCollision(){
// Test de collision suivant x
   if ( (x + d/2  >width ) || ( x - d/2 < 0 ) ) {
   dx  *= (-1) ; // changer de sens
  }
    // Test de collision suivant y
  if ( (y + d/2  >height ) || ( y - d/2 < 0 ) ) {
  dy  *= (-1) ; // changer de sens
  }
}
```

3.3 Gestion d'un jeu multi-balles : introduction des tableaux, des listes et de la boucle « for »

Nous allons maintenant faire évoluer le programme de sorte qu'il puisse faire évoluer un nombre N de balles. Pour cela, il est nécessaire d'introduite la notion de tableau de données. Dans la suite, un tableau unidimensionnel s'appelle une « liste ».

Nous allons tout d'abord créer, puis manipuler 3 balles dont la gestion sera effectuée par l'utilisation de tableaux unidimensionnels. Nous créerons ensuite un grand nombre de balles et introduiront la boucle itérative « for » pour pouvoir les gérer.

3.3.1 Utilité des tableaux et des listes

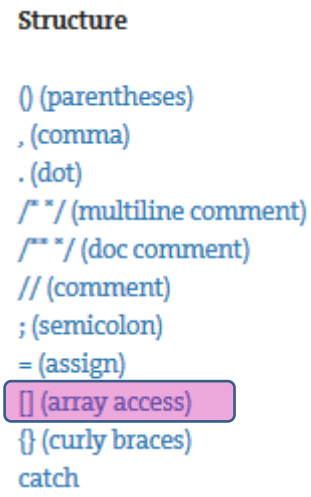
Comme nous l'avons vu précédemment, chaque balle dispose de 6 attributs qui la caractérisent de manière unique. Si on décide de faire évoluer N=10 balles, cela revient à manipuler 6*10 = 60 variables différentes, ce qui est très lourd.

La solution est de définir un tableau unidimensionnel (ou liste) de valeurs pour chaque attribut de la balle. Si nous manipulons N balles, chaque tableau comportera N valeurs de même type. Chaque tableau est alors considéré comme une variable unique qui permet de manipuler N valeurs.

Quel que soit le nombre N de balles, l'utilisation des tableaux permet alors de ne manipuler uniquement que 6 variables correspondant à chaque attribut d'une balle.

3.3.2 Comment coder un tableau sous Procesing ?

Demander aux élèves de rechercher dans la documentation la manière de coder un tableau. Le terme « tableau » se traduit par « Array » en anglais. On trouve dans la doc : Structure > [ ] (arrayacces).



L'exemple de la documentation est le suivant :

```
int[] numbers = new int[3];

numbers[0] = 90;

numbers[1] = 150;

numbers[2] = 30;

int a = numbers[0] + numbers[1]; // Sets variable 'a' to 240

int b = numbers[1] + numbers[2]; // Sets variable 'b' to 180
```

Dans la ligne en rouge précédente, un tableau d'entiers nommé « numbers » est déclaré par :

int [] numbers

Ce tableau qui comporte trois éléments (qui représentent des nombres entiers) est créé par :

numbers = new int[3] ;

L'instruction « *new* » va créer l'espace mémoire suffisant, dans l'ordinateur, pour contenir 3 entiers (c'est-à-dire 3*4 =12octets). Par défaut, la valeur des trois entiers du tableau est 0.

Il est possible ensuite d'accéder à un élément particulier du tableau en précisant, à l'intérieur des crochets qui suivent le nom du tableau (ici : « *numbers* »), l'indice de l'élément : voir les lignes en bleu et en vert. Ainsi, les trois valeurs du tableau « *numbers* » sont initialisées par les instructions en bleu :

numbers[0] = 90;

numbers[1] = 150;

numbers[2] = 30;

Remarque importante : Dans **tous les tableaux gérés par Processing, l'indice du premier élément du tableau commence à 0. Par conséquent, si le tableau comporte N éléments, l'indice du dernier élément du tableau est N-1.**

3.3.3 Application à la programmation de balles multiples

Dans un premier temps, afin d'éviter l'introduction prématurée de la boucle itérative « *for* », nous n'allons créer que 3 balles que nous allons manipuler « à la main ».

3.3.3.1 Déclaration des tableaux

Nous allons d'abord transformer en tableaux les variables associées à chaque attribut de la balle. Comme pour une variable simple, nous allons d'abord déclarer chaque tableau dans la partie « variable globale » du programme et ensuite l'initialiser dans la méthode setup(). Enfin, pour gérer les N balles, il est nécessaire de définir une variable globale N de type *int* :

```
// appel aux librairies
// déclaration des variables globales
int d; // rayon de la balle
int x; // abscisse de la balle
int y; // ordonnée de la balle
int dx; // déplacement suivant x
intdy; // déplacement suivant y
color c; // couleur de la balle
```

```
// appel aux librairies
// déclaration des variables globales
int[]d; // rayon de la balle
int[] x; // abscisse de la balle
int[] y; // ordonnée de la balle
int[]dx; // déplacement suivant x
int[]dy; // déplacement suivant y
color[] c; // couleur de la balle
int N = 3; // nombre de balles
```

3.3.3.2 Création des tableaux

Il faut ensuite créer les tableaux dans la méthode *setup()* :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales
// paramètres d'affichage
size(500,500) ;
background(#FF00E6) ;
```

```
// création des tableaux à N éléments relatifs aux attributs de la balle
d = new int[N];
  x = new int[N];
  y = new int[N];
dx = new int[N];
dy = new int[N];
  c = new color[N];
…
}
```

3.3.3.3 Initialisation des tableaux

Puis, il faut initialiser (avec les valeurs quelconques) les tableaux, à la suite de ces dernières lignes de code, dans la méthode *setup()* :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales
….
// initialisation des variables globales
d[0] = 10; // 20 pixels ---- diamètre des balles
d[1] = 15; // 15 pixels
d[2] = 20; // 20 pixels
x[0] = 100; // 100 pixels---- abscisses des balles
x[1] = 100; // 100 pixels
x[2] = 100; // 100 pixels
y[0] = 100; // 20 pixels  ---- ordonnées des balles
y[1] = 125; // 25 pixels
y[2] = 130; // 30 pixels
dx[0] = 5; //5 pixels entre deux frames ---- déplacement suivant x des balles
dx[1] = 4; //4 pixels entre deux frames
dx[2] = 3; //3 pixels entre deux frames


dy[0] = 1;//1 pixels entre deux frames---- déplacement suivant y des balles
dy[1] = 2;//2 pixels entre deux frames
```

```
dy[2] = 3;//3 pixels entre deux frames
c[0] = color(231,245,87) ; ---- couleur des balles
c[1] = color(#EF18F0) ;
c[2] = color(#71F018) ;
}
```

3.3.3.4 *Modification des méthodes liées aux actions de la balle*

Pour que les balles s'animent, il faut que les méthodes *afficher()*, *avancer()* et *testCollision()* puissent s'appliquer à une balle particulière. Il faut donc modifier ces trois méthodes qu'elles puissent prendre en paramètre d'entrée l'indice de la $i^{ème}$ balle traitée. Il faut ensuite préciser dans chaque tableau x, y, dx, dy, c et d, l'indice i de la balle concerné.

```
// Mes méthodes et classes personnelles
void afficher(int i){
// affichage de la balle
fill(231,245,87) ;
ellipse(x[i],y[i],d[i],d[i]) ;
}
void avancer(int i){
// faire avancer la balle
x[i] += dx[i] ;
y[i] += dy[i] ;
}
void testCollision(int i){
// Test de collision suivant x
if ( (x[i] + d[i]/2  >width ) || ( x[i]– d[i]/2 < 0 ) ) {
dx[i]  *= (-1) ; // changer de sens
   }
// Test de collision suivant y
if ( (y[i] + d[i]/2  >height ) || ( y[i]– d[i]/2 < 0 ) ) {
dy[i]  *= (-1) ; // changer de sens
   }
}
```

3.3.3.5 Appel aux méthodes d'action pour chaque balle

Nous allons maintenant appeler les méthodes d'actions pour chaque balle dans la méthode *draw()*. L'appel à une méthode particulière s'effectue ici directement par une ligne de code.

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
// affichage de les balles
afficher(0) ;
afficher(1) ;
afficher(2) ;
// faire avancer les balles
avancer(0) ;
avancer(1) ;
avancer(2) ;
  // Test de collision sur les bords pour chaque balle
testCollision(0) ;
testCollision(1) ;
testCollision(2) ;
}
```

3.3.3.6 Critique de la méthode

Questions aux élèves : Pensez-vous qu'il soit possible de coder directement de cette manière par exemple une cinquantaine de balles ?

Réponse : Il est clair que cette méthode est fastidieuse. Il n'est pas vraiment possible d'éviter l'écriture directe de la déclaration et de la création des tableaux d'attributs de la balle. En revanche, l'initialisation des tableaux ainsi que l'appel aux méthodes d'action des balles peut s'effectuer de manière automatique en utilisant la boucle itérative « *for* ».

3.3.4 Simplification et contraction du code : introduction de la boucles itérative « *for* »

La boucle itérative « for » permets l'exécution d'un bloc d'instructions un certain nombre de fois, en faisant généralement varier un indice entre une valeur initiale et une valeur finale avec un pas d'incrémentation.

Demander aux élèves de rechercher dans la documentation la manière de coder la boucle « *for* ». On trouve dans la partie Control > Itération :

Control

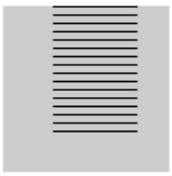
Relational Operators
!= (inequality)
< (less than)
<= (less than or equal to)
== (equality)
> (greater than)
>= (greater than or equal to)

Iteration
for
while

Name for

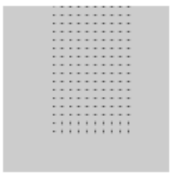
Examples

```
for (int i = 0; i < 40; i = i+1) {
  line(30, i, 80, i);
}
```



```
for (int i = 0; i < 80; i = i+5) {
  line(30, i, 80, i);
}
```

```
for (int i = 40; i < 80; i = i+5) {
  line(30, i, 80, i);
}
```



```
// Nested for() loops can be used to
// generate two-dimensional patterns
for (int i = 30; i < 80; i = i+5) {
  for (int j = 0; j < 80; j = j+5) {
    point(i, j);
  }
}
```

Expliquons le deuxième exemple :

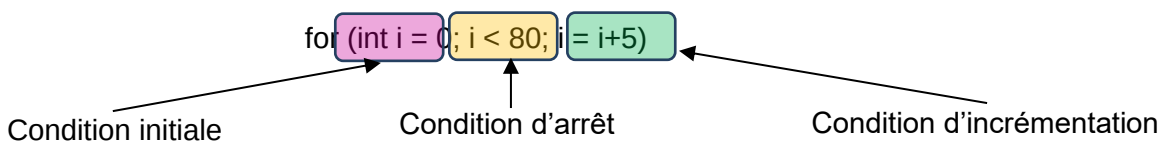
```
for (int i = 0; i < 80; i = i+5) {
line(30, i, 80, i);
}
```

L'algorithme correspondant est le suivant :

Pour un indice entier i allant de 0 jusqu'à 80 (exclu), avec un pas de 5, réaliser les instructions suivante :

Dessiner une ligne entre les point de coordonnées (30,i) et (80,i)

La syntaxe est la suivante :



3.3.5 Le code final avec l'utilisation des boucles « for »

Il est maintenant possible d'appliquer la boucle « for » pour simplifier l'écriture des taches itératives :

- Pour l'initialisation des différents tableaux dans le setup() : d, x, y, dx, dy, c.
- Pour l'appel aux différentes méthodes appliquées à chaque balle : affiche(i), avancer(i) et testCollision(i).

Pour faciliter l'initialisation, il est possible d'utiliser une méthode de génération de nombre aléatoires : ***random()*** (voir documentation). La méthode ***random()*** renvoie un nombre flottant (c'est-à-dire, à virgule) qui ne pourra être placé dans un tableau d'entier. Il faudra donc prendre la partie entière de ce nombre grâce à la méthode ***floor()*** (voir documentation).

D'autre part, pour générer des couleurs aléatoirement, il est plus facile de se placer dans une représentation HSV (teinte, saturation, valeur). En effet, la teinte n'est alors géré que par un paramètre : H, alors qu'il faut modifier trois paramètre si le mode de représentation des couleurs est basé sur la base Rouge, Vert, Bleu. Il faut donc changer de mode de couleur dans le setup() : ***colorMode(HSB)*** .

Le programme final pour par exemple 30 balles, est le suivant :

```
// appel aux librairies

// déclaration des variables globales

int[ ] d; // rayon de la balle

int[ ] x; // abscisse de la balle

int[ ] y; // ordonnée de la balle

int[ ] dx; // déplacement suivant x

int[ ] dy; // déplacement suivant y

color[ ] c; // couleur de la balle

int N = 30; // nombre de balles

void setup(){// initialisation des paramètres d'affichage et des variables globales

// paramètres d'affichage

size(500,500) ;

background(#FF00E6) ;

colorMode(HSB);

// création des tableaux à N éléments relatifs aux attributs de la balle

d = new int[N];

x = new int[N];

y = new int[N];

dx = new int[N];
```

```
dy = new int[N];
 c = new color[N];
// initialisation des variables globales
for (int i = 0 ; i<N ; i++) {
d[i] = floor(random(10,50)); // diamètre aléatoire entre 10 et 50 pixels ---- diamètre des balles
x[i] = floor(random(50,width -50));// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels
y[i] = floor(random(50,width -50));  // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
dx[i] = floor(random(1,5)); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant x des balles
dy[i] = floor(random(1,5)); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant y des balles
c[i] = color(random(255),255,255) ; // ---- couleur aléatoire des balles en représentation HSB
}
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
   // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
for (int i=0 ;i<N ; i++){
// affichage de les balles
afficher(i) ;
// faire avancer les balles
avancer(i) ;
// Test de collision sur les bords pour chaque balle
testCollision(i) ;
}
}
// Mes méthodes et classes personnelles
void afficher(int i){
// affichage de la balle
fill(231,245,87) ;
   ellipse(x[i],y[i],d[i],d[i]) ;
```

```
}

void avancer(int i){
// faire avancer la balle
x[i] += dx[i] ;
y[i] += dy[i] ;
}
void testCollision(int i){
// Test de collision suivant x
if ((x[i] + d[i]/2  >= width)  ||  (x[i] - d[i]/2 <= 0 )) {
dx[i]  *= (-1) ; // changer de sens
  }
// Test de collision suivant y
if ( (y[i] + d[i]/2  >height ) || ( y[i]- d[i]/2 < 0 ) ) {
dy[i]  *= (-1) ; // changer de sens
  }
}
```

4 Programmation Orientée Objet (POO)

4.1 Qu'est-ce que la programmation orientée objet ?

Dans cette partie, nous introduisons la Programmation Orientée Objet en définissant un objet « balle » qui va évoluer et « vivre » comme une entité individuelle.

De manière générale, chaque objet est construit sur un modèle appelé « **classe** ». Ce modèle doit comporter :

- Les paramètres caractérisant l'objet qui sont nommés « **attributs** »,
- Les actions que peut réaliser l'objet, qui sont nommés « **méthodes** »,
- Un « **constructeur »** qui permet de créer l'espace mémoire nécessaire à la vie de l'objet, ainsi que de donner des valeurs particulières à chaque attribut de l'objet, le rendant ainsi unique. Par définition, le constructeur est une méthode qui porte le même nom que la classe.

Toutes les **classes** auront la même architecture : **Attributs / Constructeur / Méthodes**.

Une fois la classe définie, il est possible de créer de multiples objets de même type. Le nom du type est le nom de la classe. L'action de créer un objet à partir d'une classe est appelé : « **instanciation** ».

Lorsqu'on crée un objet en instanciant une classe, on fait appel au constructeur de la classe en précisant des valeurs particulières à chaque attribut de la classe.

Une fois l'objet instancié, il est possible d'accéder à ses attributs et à ses méthodes pour le faire vivre.

Nous allons maintenant appliquer cette théorie pour créer un objet « balle ».

4.2 Définition de la classe balle

La réflexion précédente sur la manière de caractériser une balle (partie 2.2) va ici être mise à profit. En effet, nous avons déjà définie dans la partie 2 .2, 2.4 et 2.5 les attributs (caractéristiques) et les méthodes (actions) de la balle.

Nous pouvons le résumer dans le schéma suivant :

Classe « balle »

PARAMETRES

Attributs de la balle :

- Couleur : c
- Diamètre : d
- Position :
  - Abscisse : x
  - Ordonnée : y
- Déplacement :
  - Déplacement suivant x : dx
  - Déplacement suivant y : dy

CONSTRUCTEUR

balle()

ACTIONS

Méthodes de la balle :

- afficher()
- avancer()
- testCollision()

Instanciation

balle1

- c = rouge
- d = 20 pix
- x = 100 pix
- y =150 pix
- dx = +3 pix
- dy = -1pix

Balle2

- c = vert
- d = 35 pix
- x = 110 pix
- y =250 pix
- dx = 1 pix
- dy = 2 pix

Balle3

- c = bleu
- d = 20 pix
- x = 200 pix
- y =200 pix
- dx = 2 pix
- dy = 1 pix

…

balleN

- c = …
- d = … pix
- x = … pix
- y = … pix
- dx = … pix
- dy = … pix

Remarque 1 : L'écriture de la classe « balle » va être grandement facilité par le fait que les attributs (*c, d, x, y, dx, dy*), et les méthodes *afficher()*, *avancer()* et *testCollision()* ont déjà été codées dans le **Programme 2** de la partie **2.3.3**.

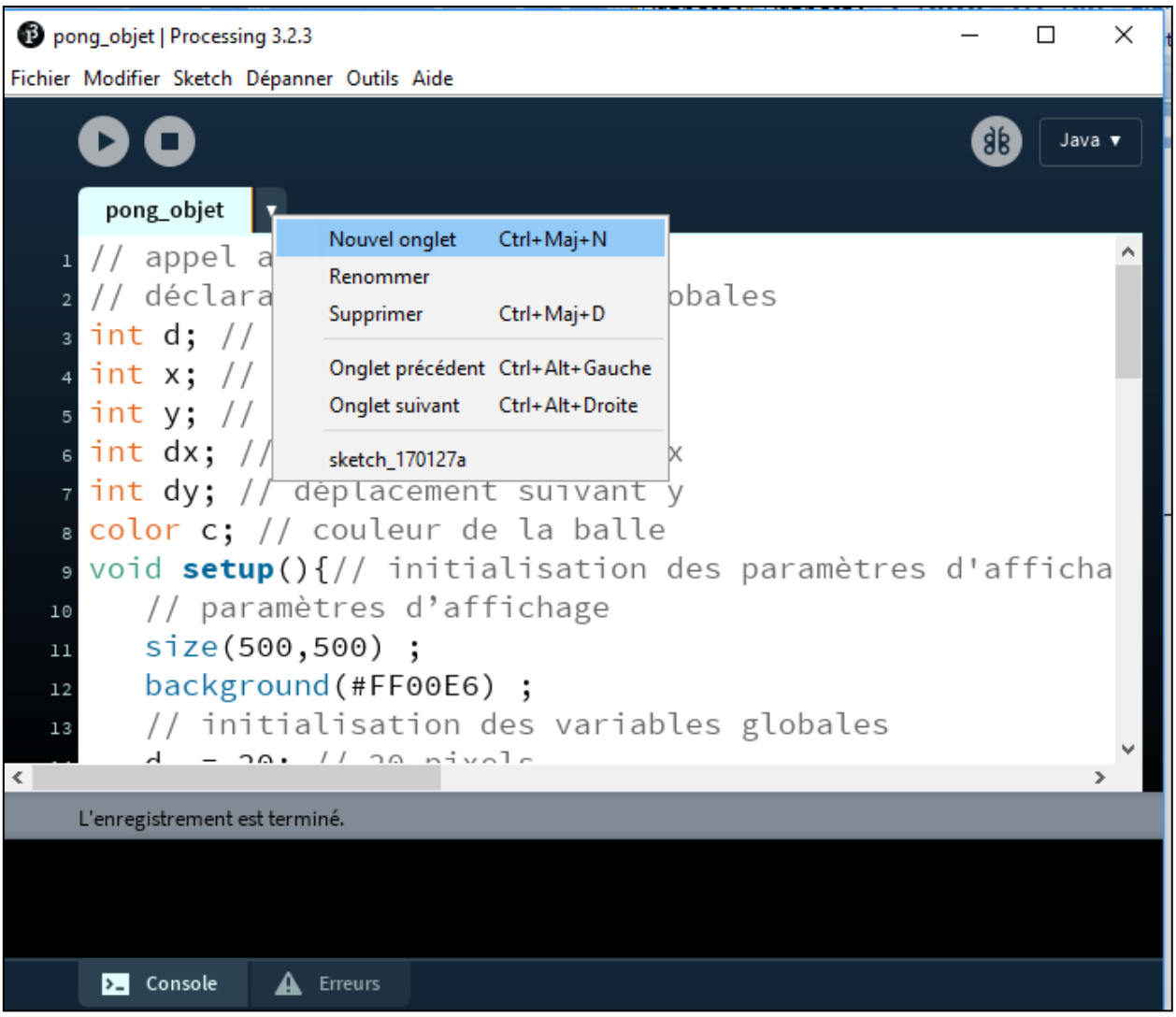
4.3 Ecriture de la classe balle

4.3.1 Architecture de la classe

La classe « balle » respecte l'architecture globale d'une classe :

```
class balle {

// Attributs

…

// Constructeur

…

// Méthodes

…

}
```

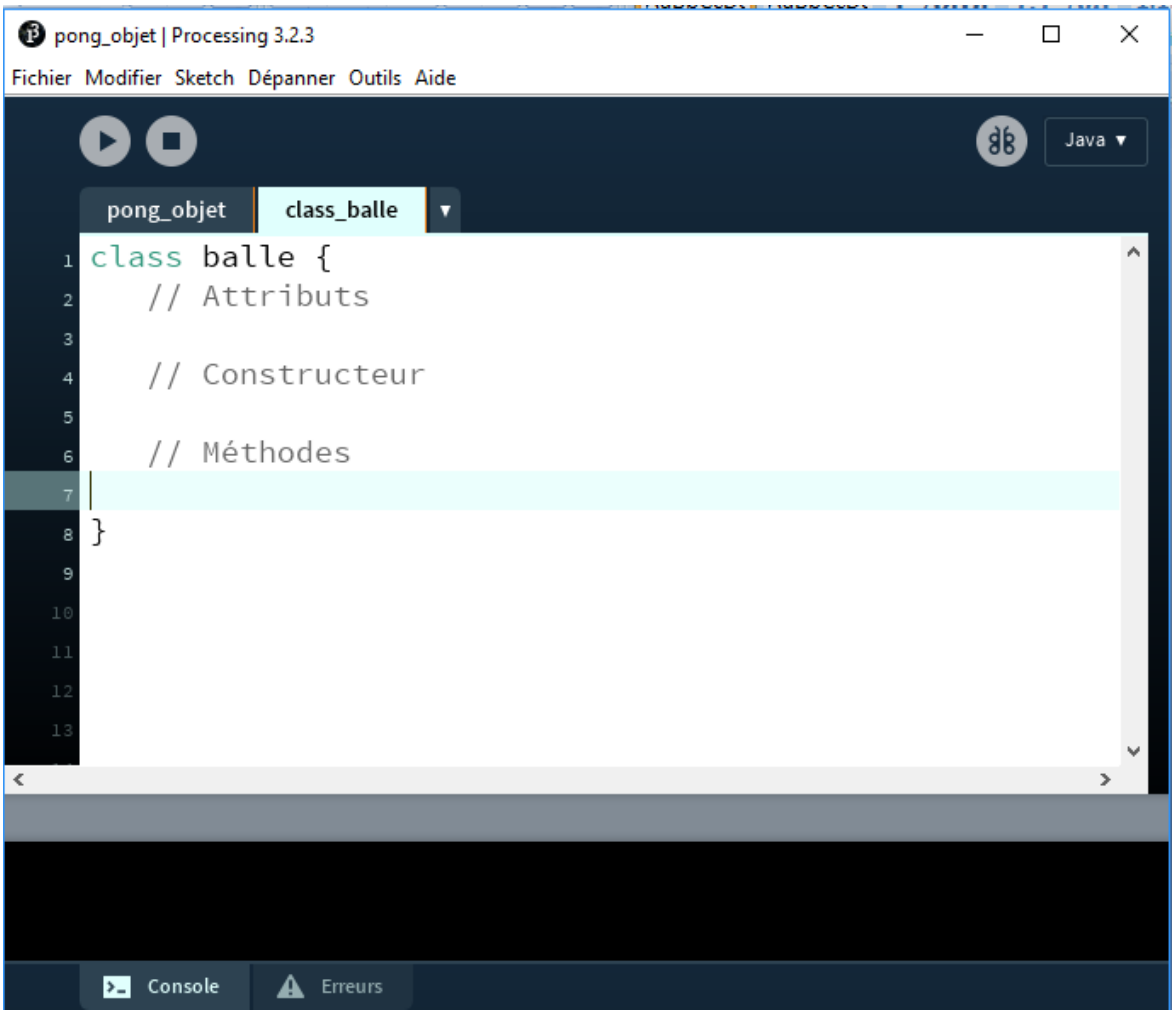
Partons du **programme 2** de la partie **2.3.2**. Pour permettre une lecture plus simple, nous allons écrire la classe « balle » dans un nouvel onglet nommé « class_balle », de l'interface de développement :



Remarque 1 : il est possible d'écrire la classe « balle » à la suite de la méthode draw() sur la même page, mais le programme comptabilisera alors un grand nombre de lignes de code les unes à la suite des autres. Ceci rend la lecture du programme difficile. Un roman peut se lire sur une page unique enroulée sur elle-même comme un parchemin, mais il est plus pratique de découper l'histoire en plusieurs pages pour former un livre. De la même manière, il est plus pratique de placer chaque nouvelle méthode dans un nouvel onglet.

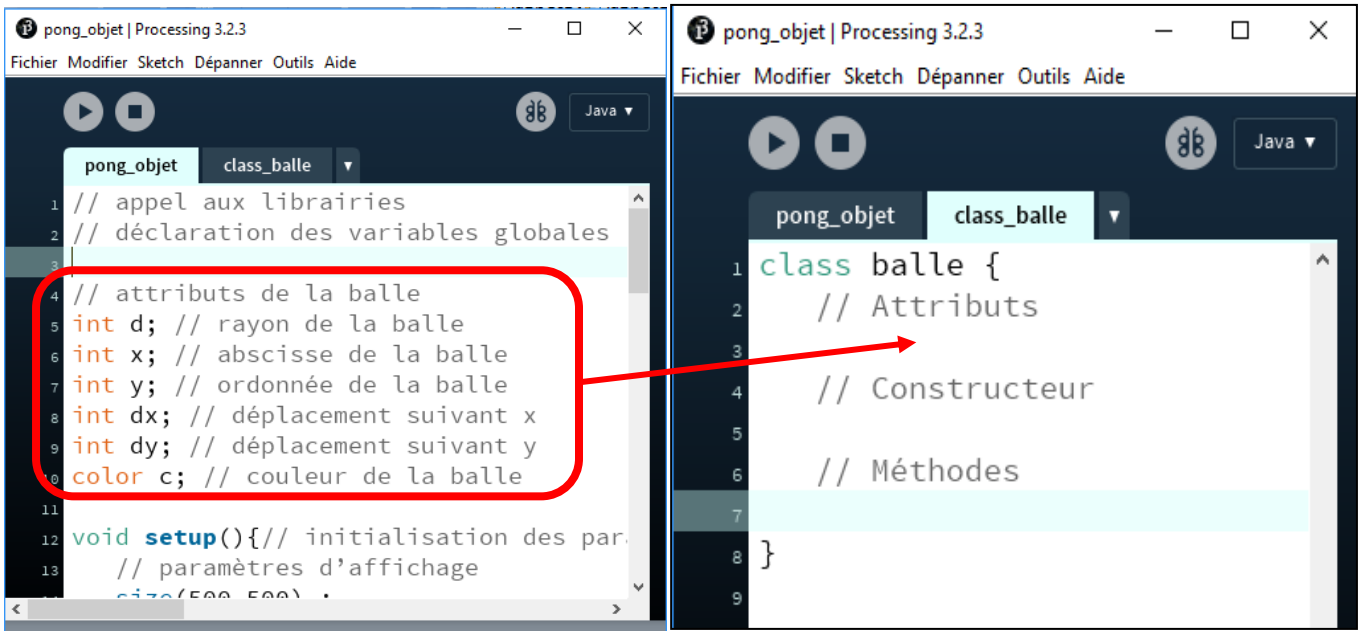
Parchemin -> pages d'un livre

On obtient donc :



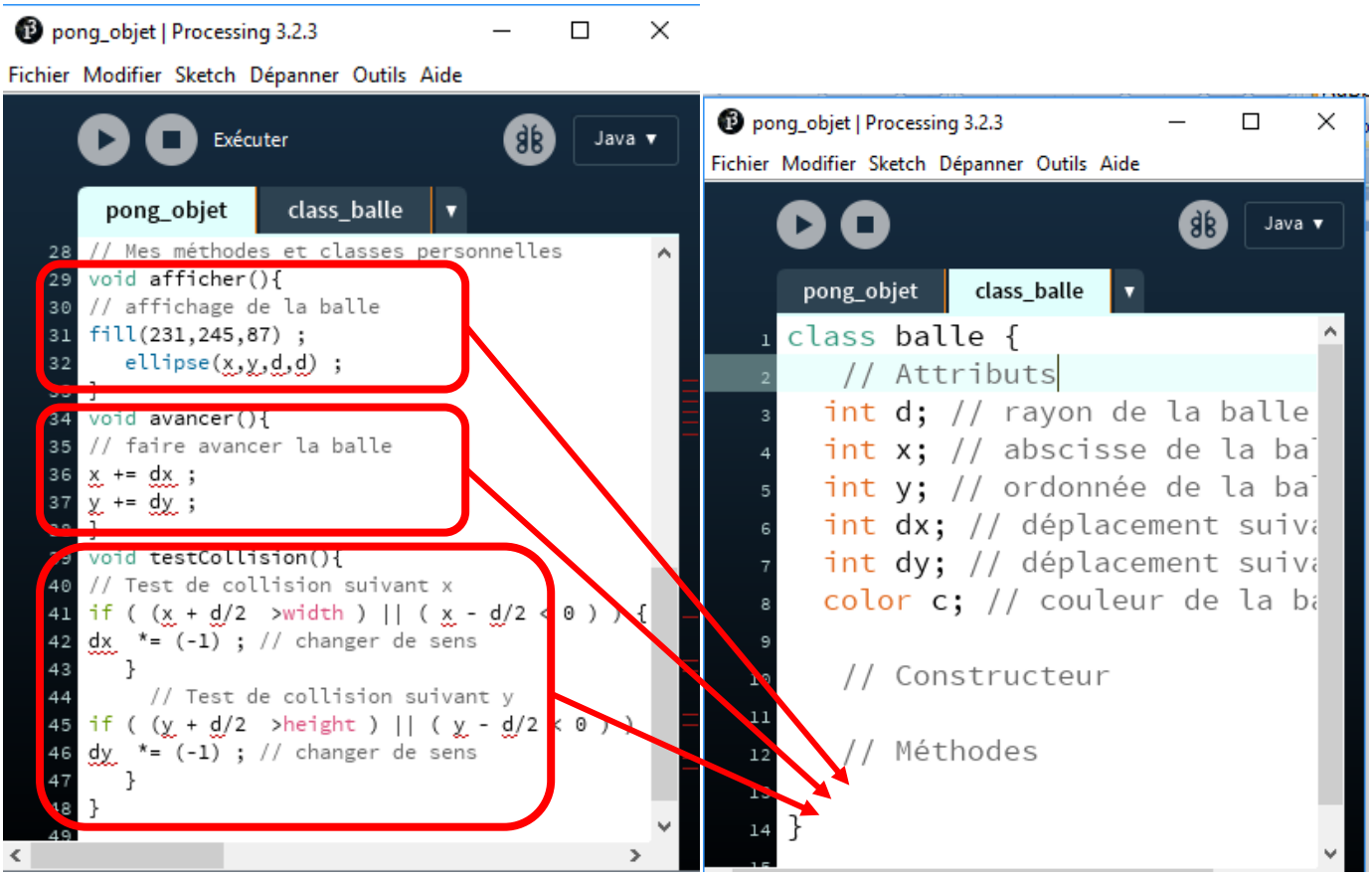
4.3.2 Ecriture des attributs

Pour remplir les attributs de la classe « balle », il suffit de couper, dans la partie « // déclaration des variables globales » du **programme2**,les lignes de code correspondant aux « //attributs de la balle », et de les coller, dans l'onglet « class_balle » dans la partie correspondante.



4.3.3 Ecriture des méthodes

Pour remplir les méthodes de la classe, il suffi de procéder de la même manière que pour les attributs, c'est-à-dire couper les méthodes *afficher()*, *avancer()* et *testCollision()* situées dans le programme 2 et les coller dans la partie « //méthodes » de la classe « balle ».



La classe balle est donc la suivante :

```
class balle {
  // Attributs
 int d; // diamètre de la balle
 int x; // abscisse de la balle
 int y; // ordonnée de la balle
 int dx; // déplacement suivant x
 int dy; // déplacement suivant y
 color c; // couleur de la balle

  // Constructeur

  // Méthodes
void afficher(){
  // affichage de la balle
   fill(231,245,87) ;
   ellipse(x,y,d,d) ;
 }
void avancer(){
  // faire avancer la balle
  x += dx ;
  y += dy ;
 }
void testCollision(){
  // Test de collision suivant x
if ( (x + d/2  >width ) || ( x - d/2 < 0 ) ) {
```

```
dx  *= (-1) ; // changer de sens

    }

     // Test de collision suivant y

if ( (y + d/2  >height ) || ( y - d/2 < 0 ) ) {

dy  *= (-1) ; // changer de sens

    }

 }

}
```

4.3.4 Ecriture du constructeur

Le rôle du **constructeur** est :

- de créer l'espace mémoire nécessaire à la vie de l'objet, c'est-à-dire un espace suffisant pour contenir à la fois les attributs et les méthodes.
- d'initialiser chaque attribut de l'objet, avec des valeurs particulières, le rendant ainsi unique.

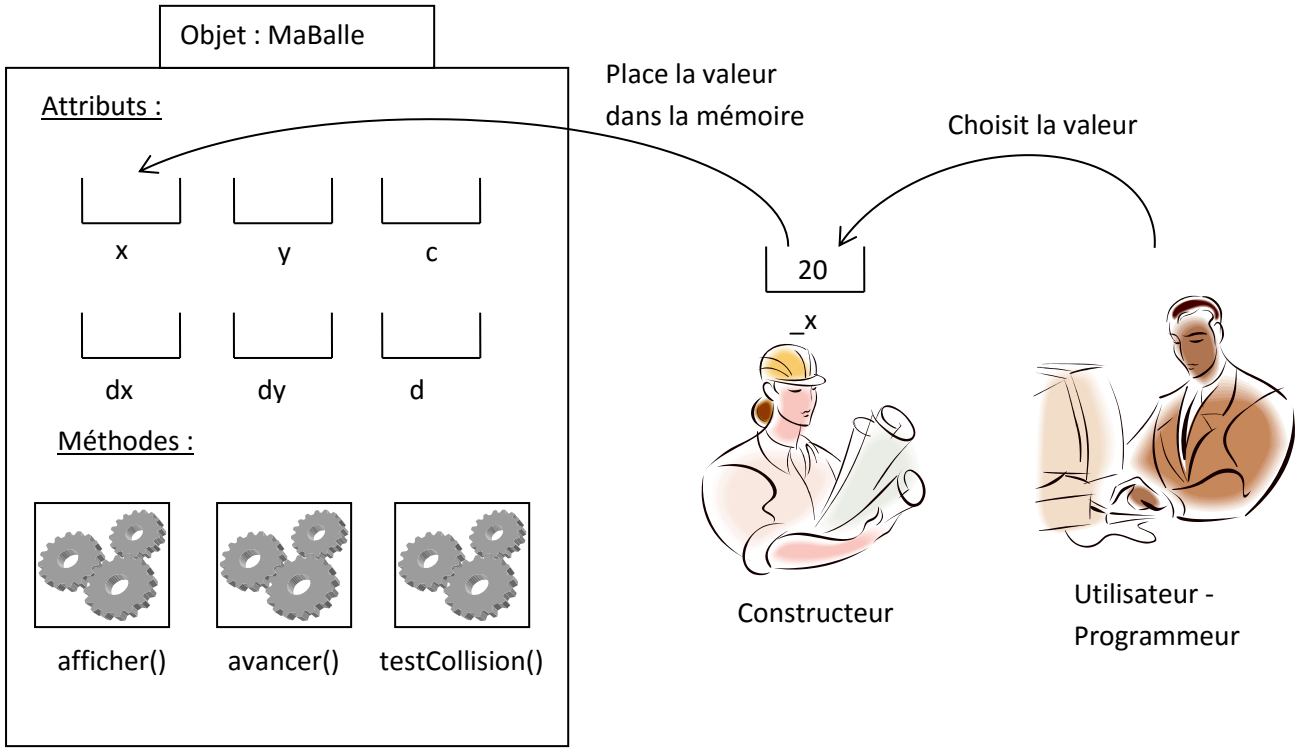
Par définition, le constructeur est une méthode qui porte le même nom que la classe. La différence avec une méthode de la classe est que le constructeur ne revoit aucun type. Son écriture n'est précédée par rien.

Pour lui permettre d'initialiser les attributs de la classe, le constructeur doit comporter autant de paramètres d'entrée (appelés « arguments ») que d'attributs. Les arguments du constructeur ne peuvent pas porter le même nom que l'attribut correspondant. Dans le cas contraire, le compilateur de Processing sera incapable de différencier les deux variables.

Les noms des arguments sont résumés dans le tableau suivant :

Type	Attributs de la classe	Arguments du constructeur
int	x	_x
int	y	_y
int	dx	_dx
int	dy	_dy
int	d	_d
color	c	_c

Schéma de l'initialisation des attributs lors de l'instanciation :



L'écriture du constructeur est donc la suivante :

```
class balle {
  // Attributs
 int d; // rayon de la balle
 int x; // abscisse de la balle
 int y; // ordonnée de la balle
 int dx; // déplacement suivant x
 int dy; // déplacement suivant y
 color c; // couleur de la balle
  // Constructeur
   balle(int _x, int _y, int _dx, int _dy, int _d, color _c){
   x = _x;
   y = _y;
  dx = _dx;
   dy = _dy;
```

```
    d = _d;
    c = _c;
   }
  // Méthodes
  void afficher(){
   // affichage de la balle
    fill(231,245,87) ;
    ellipse(x,y,d,d) ;
  }
  void avancer(){
   // faire avancer la balle
   x += dx ;
   y += dy ;
  }
  void testCollision(){
   // Test de collision suivant x
   if ( (x + d/2  >width ) || ( x - d/2 < 0 ) ) {
       dx  *= (-1) ; // changer de sens
     }
      // Test de collision suivant y
   if ( (y + d/2  >height ) || ( y - d/2 < 0 ) ) {
       dy  *= (-1) ; // changer de sens
     }
  }
}
```

5 Création et vie d'un objet balle

5.1 Déclaration d'un objet de type « balle »

Pour créer et manipuler un objet balle il faut lui attribuer une variable de type « *balle* » que nous devons déclarer, dans le corps principal du programme, en tant que variable globale. Appelons, par exemple, cette variable « maBalle » :

```
// appel aux librairies

// déclaration des variables globales

balle maBalle; // objet « balle »
```

5.2 Instanciation de la classe balle

La création d'un objet de type « *balle* » s'effectue par **l'instanciation** de la classe balle qui se réalise en faisant **appel au constructeur** de la classe. Cette étape constitue l'initialisation de la variable « *maBalle* » dans laquelle on précise la valeur de tous les attributs de la classe. Par exemple :

- c = rouge (R=255,G=0,B=0)
- d = 20 pix
- x = 100 pix
- y =150 pix
- dx = 1 pix
- dy = 2 pix

Comme pour l'initialisation de toutes les variables globales, la création de l'objet « *maBalle* » par instanciation de la classe « *balle* » s'effectue, sous Processing, dans la méthode *setup()* en précédant l'appel au constructeur de l'instruction « *new* », et en précisant la valeurs des attributs de l'objet dans l'ordre où ils ont été déclarés dans le constructeur :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales

// paramètres d'affichage

  size(500,500) ;

  background(#FF00E6) ;

// Instanciation de l'objet "balle"

  maBalle = new balle( 100, //x = 100 pixels

                       150, //y = 150 pixels

                       1, //dx = 1 pixels entre deux frames

                       2, //dy = 2 pixels entre deux frames

                       20,// d = 20 pix de diamètre
```

```
                color(255,0,0)) ; // couleur rouge
}
```

Remarque: Dans le programme permettant la gestion des balles multiples (section 3.3), nous avions déclaré des tableaux d'entier (par exemple int[] x). La création de ces tableaux dans le setup() est effectué aussi grâce à l'instruction « *new* ». Par conséquent, par construction, les tableaux sont considérés comme des objets ayant des attributs et des méthodes.

5.3 Faire vivre l'objet « maBalle »

Comme nous l'avons effectué dans la méthode draw() du **programme 2** (partie **2.3.2**), pour faire « vivre » la balle « *maBalle* », il faut appeler les méthodes *affiche()*, *avancer()* et *testCollision()* associé à l'objet « *maBalle* ».

De manière générale, il est possible d'accéder à un attribut ou à une méthode de l'objet via l'opérateur « . ». La modification de la méthode *draw()* :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
// affichage de la balle
  maBalle.afficher() ;
// faire avancer la balle
  maBalle.avancer() ;
   // Test de collision sur les bords
  maBalle.testCollision() ;
}
```

5.4 Programme final

```
// appel aux librairies
// déclaration des variables globales
balle maBalle; // objet « balle »
void setup(){// initialisation des paramètres d'affichage et des variables globales
// paramètres d'affichage
```

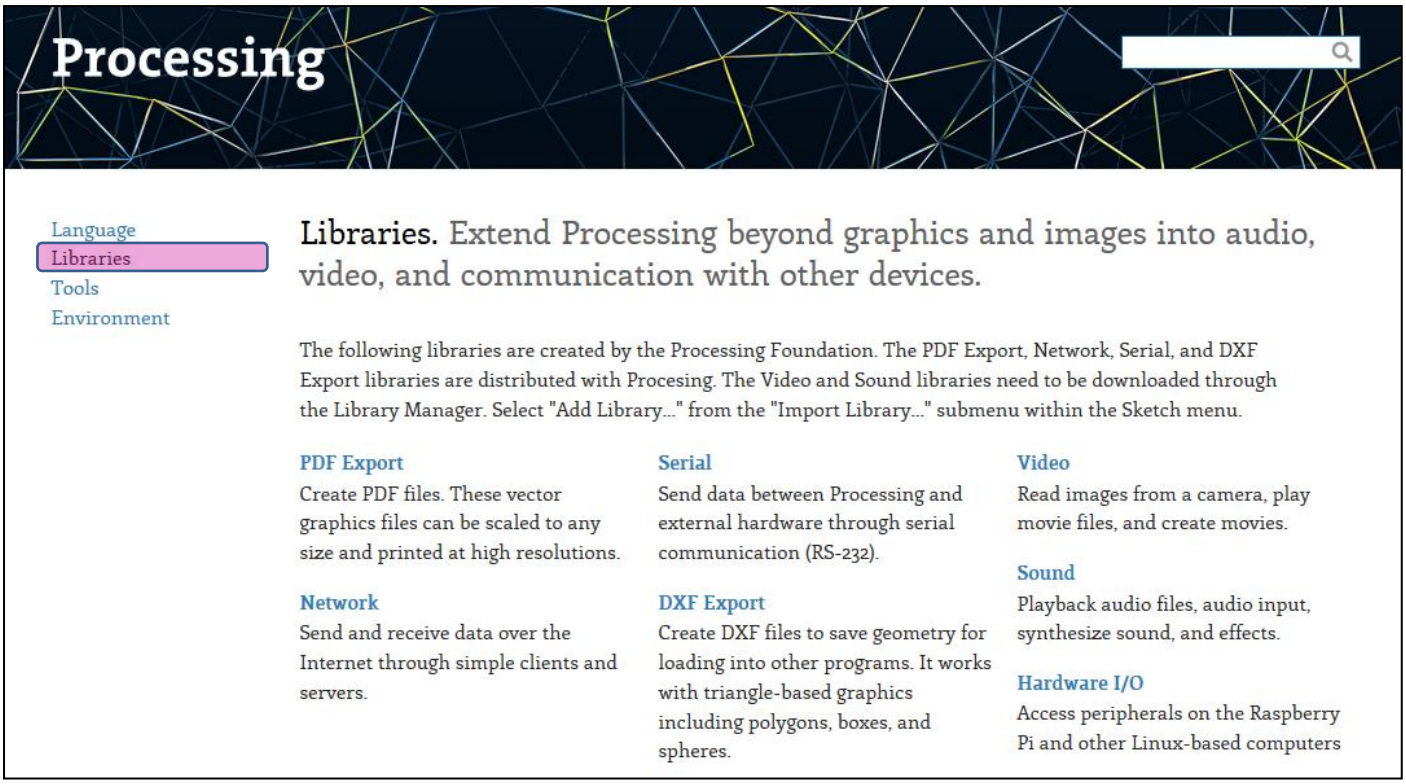
```
  size(500,500) ;
  background(#FF00E6) ;
// instanciation de l'objet "balle"
  maBalle = new balle( 100, //x = 100 pixels
                       150, //y = 150 pixels
                       1, //dx = 1 pixels entre deux frames
                       2, //dy = 2 pixels entre deux frames
                       20,// d = 20 pix de diamètre
                       color(255,0,0)) ; // couleur rouge
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
// affichage de la balle
 maBalle.afficher() ;
// faire avancer la balle
 maBalle.avancer() ;
  // Test de collision sur les bords
 maBalle.testCollision() ;
}
```

6 Utilisation des librairies : vidéo, son, typographie, etc…

6.1 Présentation des références

Toutes les librairies disponibles sous Processing sont construites sur des classes d'objets. Pour les manipuler convenablement, il est impératif de bien assimiler la manière dont est construite une classe pour être capable d'identifier les différents attributs qui caractérisent l'objet et les différentes méthodes de la classe, c'est-à-dire les différentes fonctionnalités que propose la librairie.

Généralement, les librairies standards sont bien documentées. Analysons par exemple la librairie « Video » permettant la gestion des vidéos et de la webCam et la librairie « Sound » qui permet la gestion du son. Elle sont accessibles sur le lien « Libraries » de la documentation :

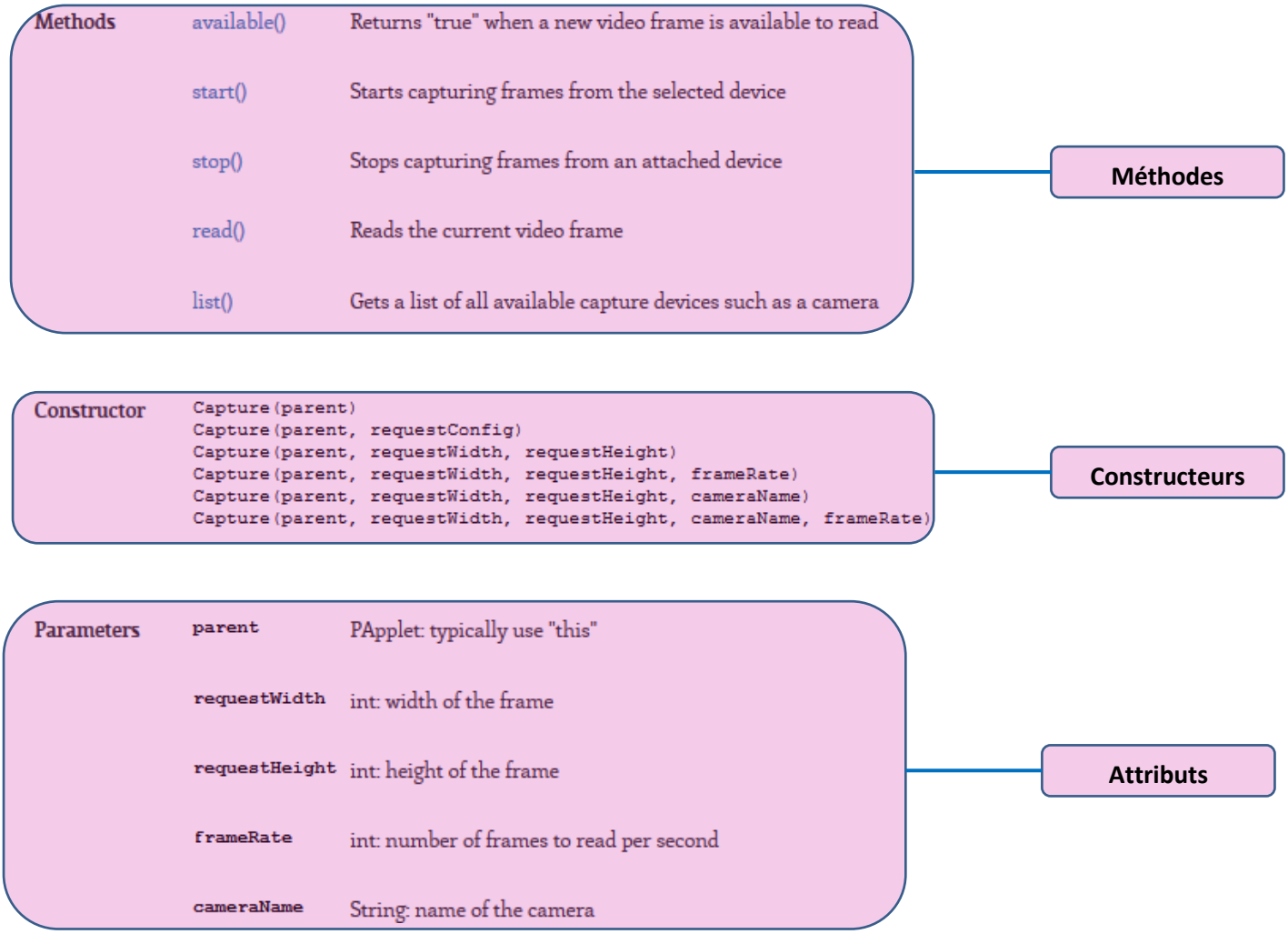


6.1.1 Librairie « Video »

La librairie « **Video** » comporte deux classes :

- La classe « **Movie** » permettant de charger des vidéos et de les lire de n'importe quelle manière incluant la lecture en boucle (loop), la pause et le changement de vitesse.
- La classe « **Capture** » permettant de capter les images issues d'un dispositif de capture comme une webCam ou une Camera.

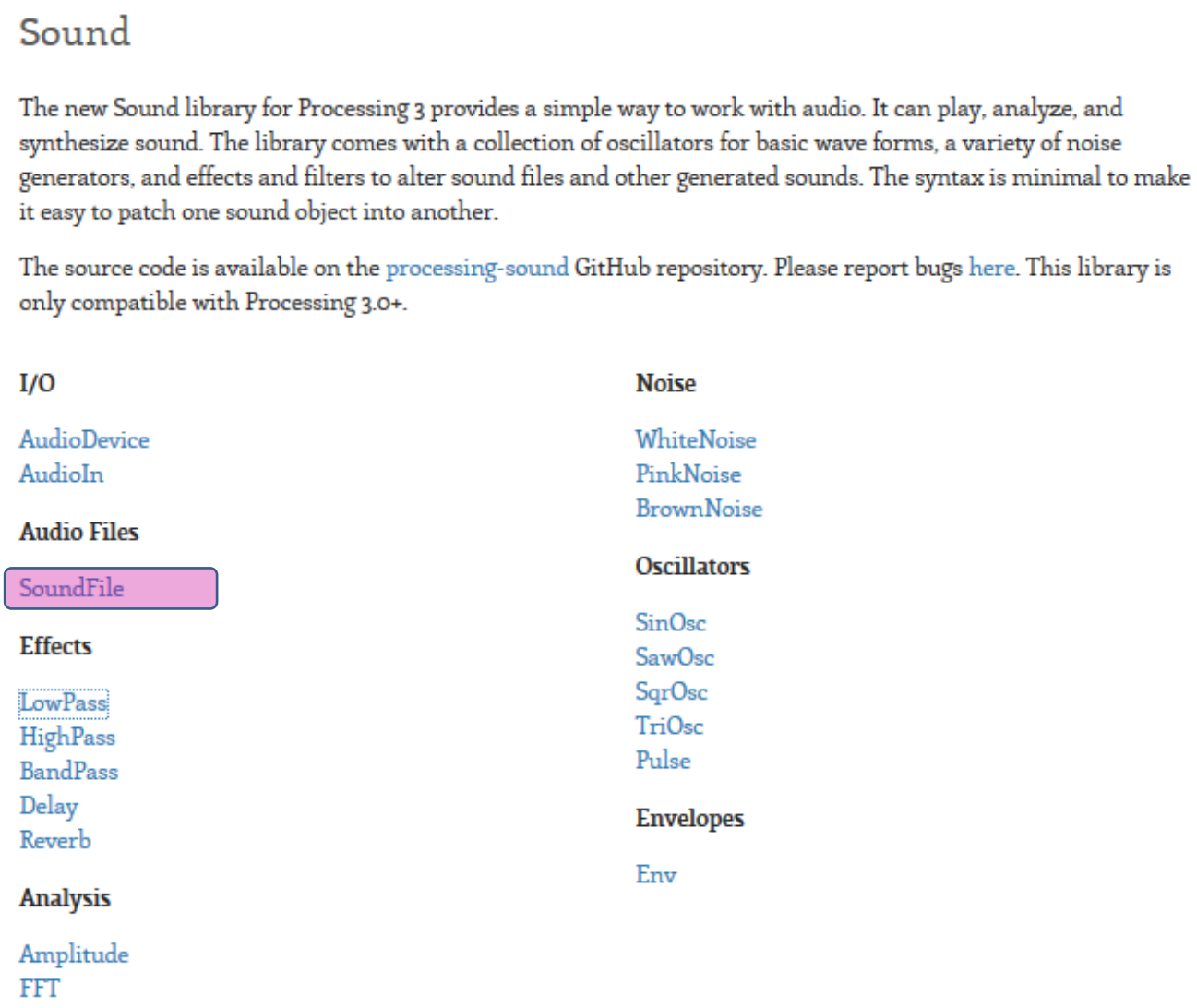
Détaillons la classe « **Capture** ». Nous retrouvons les trois champs principaux définissant les classes : les attributs, les méthodes et le/les constructeurs :



Remarque : Dans cette classe, nous remarquons qu'il existe plusieurs constructeurs qui diffèrent par le nombre d'arguments qu'ils comportent. On dit que la classe est **polymorphe** (poly : plusieurs, morphe : formes). Le **polymorphisme** peut aussi s'appliquer aux différentes méthodes d'une classe sur le même principe.

6.1.2 Librairie « Sound »

La librairie « **Sound** » comporte un grand nombre de classes :



Sound

The new Sound library for Processing 3 provides a simple way to work with audio. It can play, analyze, and synthesize sound. The library comes with a collection of oscillators for basic wave forms, a variety of noise generators, and effects and filters to alter sound files and other generated sounds. The syntax is minimal to make it easy to patch one sound object into another.

The source code is available on the processing-sound GitHub repository. Please report bugs here. This library is only compatible with Processing 3.0+.

I/O

AudioDevice
AudioIn

Audio Files

SoundFile

Effects

LowPass
HighPass
BandPass
Delay
Reverb

Analysis

Amplitude
FFT

Noise

WhiteNoise
PinkNoise
BrownNoise

Oscillators

SinOsc
SawOsc
SqrOsc
TriOsc
Pulse

Envelopes

Env

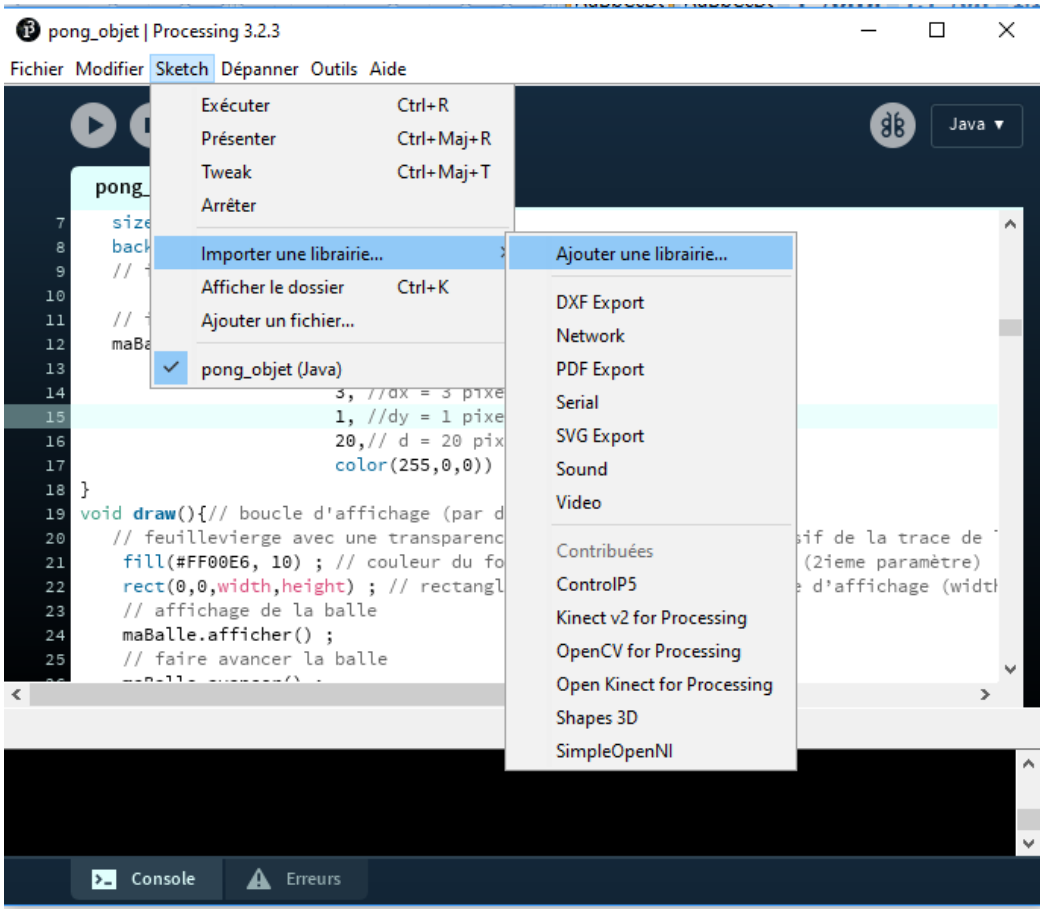
Analysons particulièrement la classe « SoundFile » :

Methods	frames()	Returns the number of frames/samples of the sound file.
	sampleRate()	Returns the sample rate of the soundfile.
	channels()	Returns the number of channels in the soundfile.
	duration()	Returns the duration of the the soundfile.
	play()	Starts the playback of a soundfile. Only plays the soundfile once.
	loop()	Starts the playback of a soundfile to loop.
	jump()	Jump to a specific position in the file while continuing to play.
	cue()	Cues the playhead to a fixed position in the soundfile. Note that the time parameter supports only integer values.
	set()	Set multiple parameters at once
	pan()	Move the sound in a stereo panorama, only supports Mono Files
	rate()	Change the playback rate of the soundfile.
	amp()	Changes the amplitude/volume of the player.
	add()	Offset the output of the player by given value
	stop()	Stops the player
Constructor	`SoundFile(theParent, path)`	

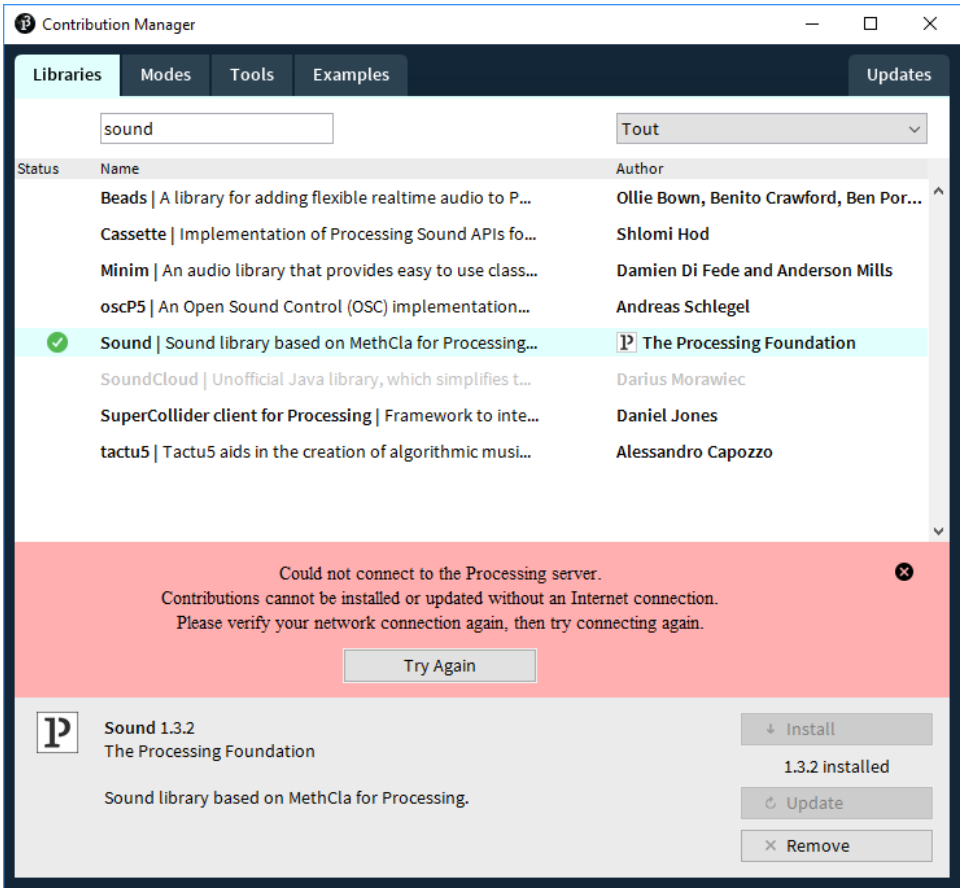
Elle comporte un grand nombre de méthodes (dont les plus importantes pour la suite seront *play()*, *loop()*, *stop()*) un constructeur unique (la classe n'est pas polymorphe), et aucun attribut.

6.2 Téléchargement et installation de la librairie

Toutes les librairies s'installent facilement via l'interface d'installation :



Il suffit alors de taper dans le moteur de recherche le nom de la librairie souhaitée (ici « *Sound* ») et d'appuyer sur « Install » :



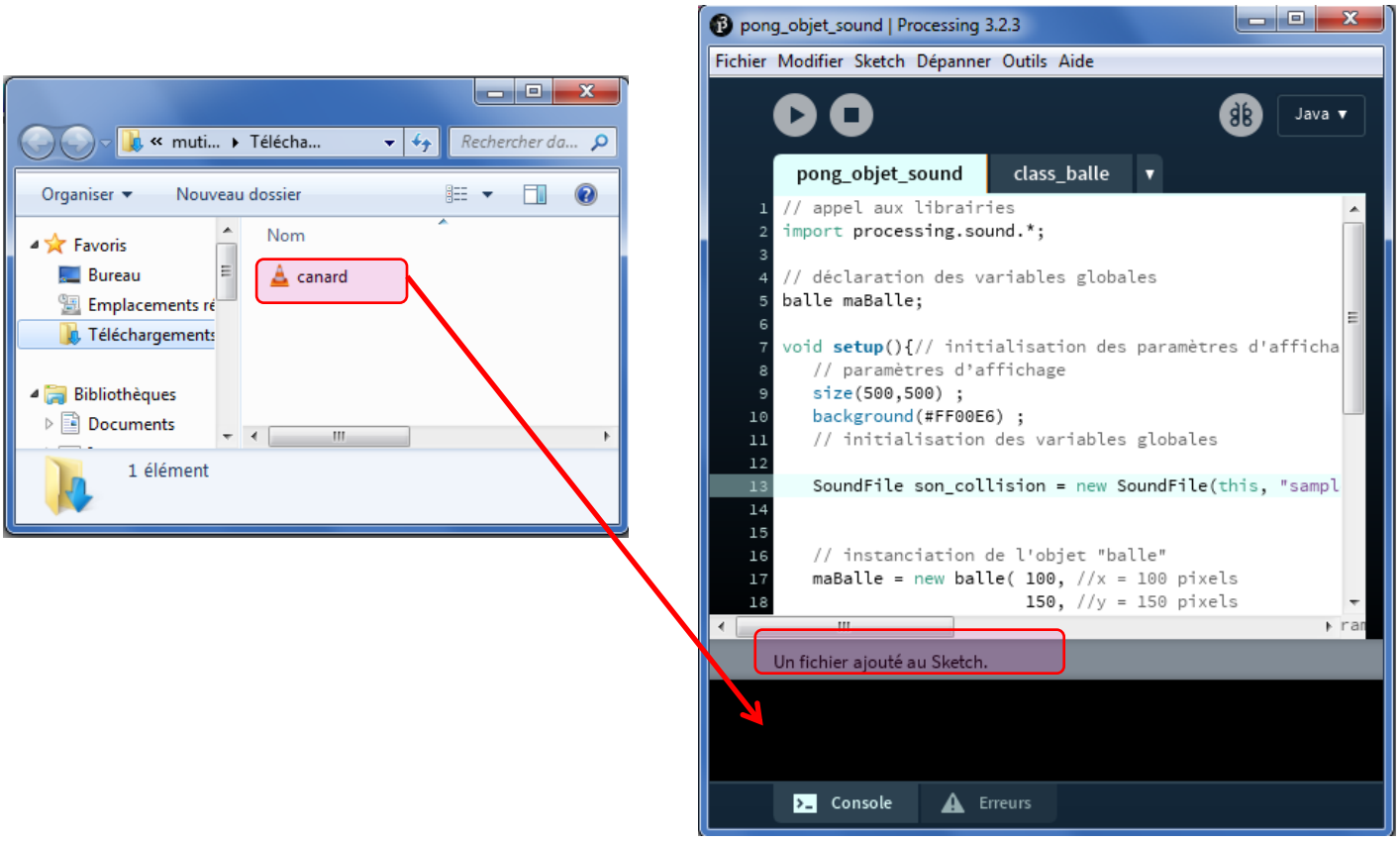
6.3 Inclure un fichier son dans le programme

Des bibliothèques de sons sont disponibles sur internet :

- http://www.universal-soundbank.com/
- http://lasonotheque.org/
- Etc.

Pour pouvoir utiliser un sample sonore, il faut l'inclure dans le dossier « Data » du sketch en cours :

1) Télécharger le son souhaité,
2) Glisser le fichier son dans la fenêtre de l'IDE de Processing.
3) Le message « Un fichier ajouté au sketch » apparait dans la console si l'inclusion du fichier son s'est bien déroulée.



6.4 Utilisation de la classe « Sound »

Nous pouvons ajouter un son à chaque collision de la balle avec les bords ou un autre objet (raquette ou autre balle). Considérons que ce son est une caractéristique de la balle. Il faut alors rajouter un attribut à la classe « balle » qui représente ce son.

6.4.1 Importation de la librairie Sound

Pour instancier un objet de la classe « *Sound* » il faut d'abord importer les fonctionnalités de la librairie en entête de programme :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle maBalle; // objet « balle »
```

Le signe « * » signifie qu'on importe **toutes** les fonctionnalités de la librairie.

6.4.2 Ajout d'un attribut de son à la classe « balle »

Pour attribuer un son spécifique à la balle, on ajoute un attribut de type « *SoundFile* » à la classe balle. On peut par exemple l'appeler « *son* » :

```
// Attributs

  int d; // rayon de la balle

  int x; // abscisse de la balle

  int y; // ordonnée de la balle

  int dx; // déplacement suivant x

  int dy; // déplacement suivant y

  color c; // couleur de la balle

SoundFile son ; // son associé à la balle
```

6.4.3 Modification du constructeur pour initialiser l'attribut « son »

L'initialisation du nouvel attribut « son » nécessite de modifier le constructeur en y rajoutant un argument « _son » de type SoundFile :

```
balle(int _x, int _y, int _dx, int _dy, int _d, color _c, SoundFile _son){

    x = _x;

    y = _y;

dx = _dx;

    dy = _dy;

    d = _d;

c = _c;

son = _son ;

  }
```

6.4.4 Modification de la méthode *testCollision()* pour inclure le son à chaque collision

Pour qu'un son soit émis à chaque collision avec les bords de la fenêtre d'affichage, il faut enrichir le bloc d'instruction suivant les tests de collision de la balle, dans la méthode *testCollision()* :

```
void testCollision(){
// Test de collision suivant x
if ( (x + d/2  >width ) || ( x - d/2 < 0 ) ) {
        dx  *= (-1) ; // changer de sens
        son.play() ; // lancer le son
  }
    // Test de collision suivant y
if ( (y + d/2  >height ) || ( y - d/2 < 0 ) ) {
        dy  *= (-1) ; // changer de sens
        son.play() ; // lancer le son
  }
}
```

6.4.5 Instanciation d'un nouvel objet « balle » prenant en compte le son à chaque collision

Dans le setup(), il faut d'abord charger le son dans le programme. On utilise pour cela une variable locale appelée « son_collision » de type SoundFile :

```
        void setup(){// initialisation des paramètres d'affichage et des variables globales
        // paramètres d'affichage
          size(500,500) ;
          background(#FF00E6) ;
        // initialisation des variables globales
          // importantion d'un son
          SoundFile son_collision = new SoundFile(this, "sample.mp3");// importantion d'un son
```

Ensuite, il faut appeler ce son dans le constructeur de la classe balle :

```
        // instanciation de l'objet "balle"
          maBalle = new balle( 100, //x = 100 pixels
                               150, //y = 150 pixels
                               3, //dx = 3 pixels entre deux frames
                               1, //dy = 1 pixels entre deux frames
                               20,// d = 20 pix de diamètre
```

```
                    color(255,0,0), // couleur rouge
                    son_collision) ; // son de la collision
```

Dans la méthode *draw(),* la méthode *testColision()* qui est appelée a été modifiée de sorte que le son soit lancé à chaque collision de la balle avec les bords.

6.4.6 Programme final

Le sketch global est donc le suivant :

```
class balle {
        // Attributs
        int d; // diamètre de la balle
        int x; // abscisse de la balle
        int y; // ordonnée de la balle
        int dx; // déplacement suivant x
        int dy; // déplacement suivant y
        color c; // couleur de la balle
        SoundFile son ; // son associé à la balle
        // Constructeur
        balle(int _x, int _y, int _dx, int _dy, int _d, color _c, SoundFile _son){
          x = _x;
          y = _y;
         dx = _dx;
          dy = _dy;
          d = _d;
         c = _c;
          son = _son;
        }
        // Méthodes
        void afficher(){
         // affichage de la balle
          fill(231,245,87) ;
          ellipse(x,y,d,d) ;
        }
        void avancer(){
```

```
            // faire avancer la balle
            x += dx ;
            y += dy ;
          }
           void testCollision(){
            // Test de collision suivant x
            if ( (x + d/2  >width ) || ( x - d/2 < 0 ) ) {
            dx  *= (-1) ; // changer de sens
              }
                // Test de collision suivant y
            if ( (y + d/2  >height ) || ( y - d/2 < 0 ) ) {
            dy  *= (-1) ; // changer de sens
              }
           }
}
```

Et le corps du programme :

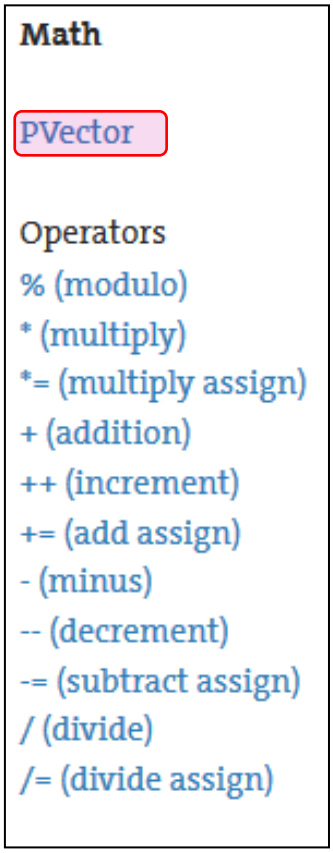
```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle maBalle;
void setup(){// initialisation des paramètres d'affichage et des variables globales
        // paramètres d'affichage
          size(500,500) ;
          background(#FF00E6) ;
        // initialisation des variables globales


          // importantion d'un son
        SoundFile son_collision = new SoundFile(this, "sample.mp3");
        // instanciation de l'objet "balle"
          maBalle = new balle( 100, //x = 100 pixels
                               150, //y = 150 pixels
```

```
                        3, //dx = 3 pixels entre deux frames
                        1, //dy = 1 pixels entre deux frames
                        20,// d = 20 pix de diamètre
                        color(255,0,0), // couleur rouge
                        son_collision) ; // son de la collision
}

void draw(){// boucle d'affichage (par défaut: 30 fois/s)
          // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
           fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
           rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
           // affichage de la balle
           maBalle.afficher() ;
           // faire avancer la balle
           maBalle.avancer() ;
             // Test de collision sur les bords
           maBalle.testCollision() ;
}
```

7 Vecteurs position et déplacement : classe PVector

7.1 Présentation de la classe PVector

La documentation sur la classe *PVector* se trouve dans la partie « **Maths**» des références de Processing :



La classe PVector se compose de 3 attributs (x, y, z), de 22 méthodes permettant la manipulation et le calcul vectoriel (somme de vecteurs, produit scalaire, calcul de distances, norme de vecteur, etc…), et 3 constructeurs permettant la création de vecteurs dans un espace de dimension 2 ou 3 :

Fields	x	The x component of the vector
	y	The y component of the vector
	z	The z component of the vector
Methods	set()	Set the components of the vector
	random2D()	Make a new 2D unit vector with a random direction.
	random3D()	Make a new 3D unit vector with a random direction.
	fromAngle()	Make a new 2D unit vector from an angle
	copy()	Get a copy of the vector
	mag()	Calculate the magnitude of the vector
	magSq()	Calculate the magnitude of the vector, squared
	add()	Adds x, y, and z components to a vector, one vector to another, or two independent vectors
	• • •	

Constructor	`PVector()`
	`PVector(x, y, z)`
	`PVector(x, y)`	

7.2 Modification de la classe « balle » pour utiliser des vecteurs « position » et « déplacement »

Les deux attributs de position *x* et *y* peuvent se regrouper dans une seule variable « *pos* » de type *PVector* représentant le vecteur position. Il en est de même pour les deux attributs de déplacement *dx* et *dy* qui peuvent se regrouper dans dans une variable « *dep* » de type *PVector* représentant le vecteur déplacement.

L'intérêt d'utiliser des variables de type PVector est multiple :

- Cela permet de diminuer le nombre d'attribut de la classe qui devient plus concise.
- Cela permettra par la suite d'utiliser toutes les méthodes permettant la manipulation des vecteurs, notamment si l'on souhaite gérer la collision entre les balles dans un jeu multi-balles. En effet, le calcul de la distance entre le centre des balles s'effectue très rapidement.

7.2.1 Modification des attributs

On définit les vecteurs positions et déplacement comme attributs de la classe balle :

```
class balle {
  // Attributs
PVector pos; // Vecteur position du centre de la balle
 PVector dep; // Vecteur déplacement du centre de la balle
 color c; // couleur de la balle
 int d; // diamètre de la balle
 SoundFile son ; // son associé à la balle
```

La classe balle ne comporte plus que 5 attributs au lieu de 7.

7.2.2 Modification du constructeur

L'initialisation des vecteurs « position » et « déplacement » s'effectue dans le constructeur :

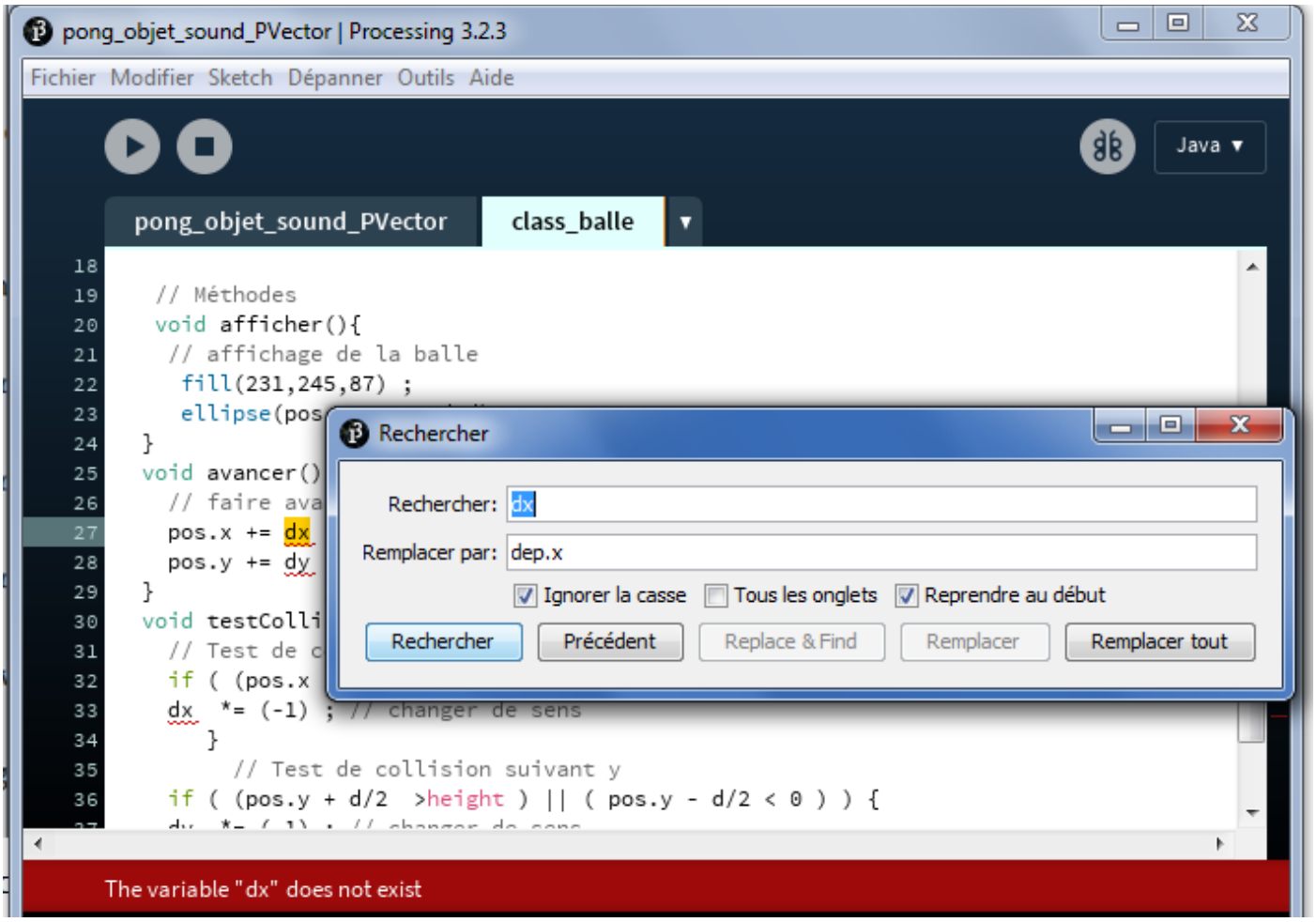
```
// Constructeur
  balle(PVector _pos, PVector _dep, int _d, color _c, SoundFile _son){
pos = _pos;
dep = _dep;
   d = _d;
   c = _c;
son = _son;
  }
```

7.2.3 Modification des méthodes

Dans chacune des méthodes *afficher()*, *avancer()* et *testCollision()*, il faut effectuer les changements d'écriture suivants :

- *x* → *pos.x*
- *y* → *pos.y*
- *dx* → *dep.x*
- *dy* → *dep.x*

Pour accélérer la modification, il est possible d'utiliser la fenêtre de recherche et de remplacement :



On obtient les méthodes suivantes :

```
// Méthodes
void afficher(){
 // affichage de la balle
  fill(231,245,87) ;
  ellipse(pos.x,pos.y,d,d) ;
}
void avancer(){
  // faire avancer la balle
  pos.x += dep.x ;
  pos.y += dep.y ;
}
void testCollision(){
  // Test de collision suivant x
```

```
        if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
                dep.x  *= (-1) ; // changer de sens
                son.play();
        }
              // Test de collision suivant y
           if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
                dep.y  *= (-1) ; // changer de sens
                son.play();
              }
}
```

Remarque : l'écriture actuelle utilisant dep.x au lieu de simplement x, etc. est plus lourde que précédemment. C'est un sacrifice que nous devons accepter pour utiliser les méthodes de la classe PVector…

7.3 Instanciation des vecteurs « position » et « déplacement »

Afin d'instancier une balle avec le nouveau constructeur prenant en compte les vecteurs « position » et « déplacement », il faut d'abord instancier ces deux vecteurs en tant que variable locale de la méthode *setup()* :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales
        // paramètres d'affichage
          size(500,500) ;
          background(#FF00E6) ;
        // initialisation des variables globales
          // vecteur position
          PVector p = new PVector(100,150); //x = 100 pixels et y = 150 pixels
          // vecteur déplacement
          PVector d = new PVector(3,1); //dx = 3 pixels entre deux frames et dy = 1 pixels entre deux frames
          // importantion d'un son
          SoundFile son_collision = new SoundFile(this, "canard.mp3");
```

7.4 Instanciation de la nouvelle classe balle

L'instanciation de la classe« *balle* » avec le nouveau constructeur prenant en compte les vecteurs « position » et « déplacement » s'effectue dans la méthode *setup()* à la suite du code précédent :

```
        // instanciation de l'objet "balle"
          maBalle = new balle( p, //vecteur position
                               d, // vecteur déplacement
                               20,// d = 20 pix de diamètre
                               color(255,0,0), // couleur rouge
                               son_collision) ; // son de la collision
}
```

Dans la méthode *draw(),* les méthodes *afficher(), avancer()* et*testColision(),* sont appelées pour faire « vivre » la balle. Elles ont été modifiées pour prendre en compte les vecteurs « position » et « déplacement ».

7.5 Programme final

Le programme final est le suivant :

```
class balle {
          // Attributs
          PVector pos; // Vecteur position du centre de la balle
          PVector dep; // Vecteur déplacement du centre de la balle
          color c; // couleur de la balle
          int d; // rayon de la balle
          SoundFile son ; // son associé à la balle
          // Constructeur
          balle(PVector _pos, PVector _dep, int _d, color _c, SoundFile _son){
            pos = _pos;
        dep = _dep;
            d = _d;
            c = _c;
        son = _son;
          }
          // Méthodes
          void afficher(){
           // affichage de la balle
```

```
            fill(231,245,87) ;
            ellipse(pos.x,pos.y,d,d) ;
          }
          void avancer(){
            // faire avancer la balle
            pos.x += dep.x ;
            pos.y += dep.y ;
          }
          void testCollision(){
            // Test de collision suivant x
        if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
        dep.x  *= (-1) ; // changer de sens
              son.play();
             }
              // Test de collision suivant y
           if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
              dep.y  *= (-1) ; // changer de sens
              son.play();
             }
          }
}
```

Le corps du programme est le suivant :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle maBalle;
void setup(){// initialisation des paramètres d'affichage et des variables globales
        // paramètres d'affichage
           size(500,500) ;
           background(#FF00E6) ;
        // initialisation des variables globales
```

```
        // vecteur position
        PVector p = new PVector(100,150); //x = 100 pixels et y = 150 pixels
        // vecteur déplacement
        PVector d = new PVector(3,1); //dx = 3 pixels entre deux frames et dy = 1 pixels entre deux frames
        // importantion d'un son
        SoundFile son_collision = new SoundFile(this, "canard.mp3");
        // instanciation de l'objet "balle"
        maBalle = new balle( p, //vecteur position
                             d, // vecteur déplacement
                             20,// d = 20 pix de diamètre
                             color(255,0,0), // couleur rouge
                             son_collision) ; // son de la collision
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
        // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
         fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
        rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
        // affichage de la balle
        maBalle.afficher() ;
        // faire avancer la balle
        maBalle.avancer() ;
          // Test de collision sur les bords
        maBalle.testCollision() ;
}
```

8 Liste d'objets « balle » et tableaux dynamiques ArrayList

8.1 Liste d'objets « balle »

Nous avons déjà étudié les listes et tableaux dans la section 3.3. Nous allons ici une liste d'objet de type « balle » et nous allons « faire vivre » chacun des objets de la liste.

8.1.1 Déclaration d'un tableau de balle dans les variables globales

Dans les variables globales, remplaçons la variable « *maBalle* » de type « *balle* » par un tableau de balle que nous nommons « *mesBalles* ». Comme dans la section 3.3.3.1, il est nécessaire ici de déclarer une variable globale

permettant de gérer le nombre de balles. Nous déclarons donc une variable *N* de type *int* qui prend par exemple la valeur 10 (10 balles):

```
// appel aux librairies

import processing.sound.*;

// déclaration des variables globales

balle[] mesBalles; // tableau de balle

int N = 10 ; // nombre de balles
```

8.1.2 Création du tableau de balle dans le *setup()*

Il convient maintenant de créer le tableau de N balles dans le setup() :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales

// paramètres d'affichage

   size(500,500) ;

   background(#FF00E6) ;

// initialisation des variables globales

//création du tableau de balles

   mesBalles = new balle[N];
```

8.1.3 Initialisation du tableau de balle dans le *setup()*

Comme dans la section 3.3.4, nous allons utiliser les boucles « for » pour faciliter l'initialisation des balles. Nous allons pour cela utiliser la méthode ***random()*** pour générer des positions et couleurs aléatoires. Dans un premier temps, nous associons le même son à chaque balle. Le code est le suivant :

```
//initialisation des balles

   for (int i = 0; i<mesBalles.length ; i++){

      // vecteur position

      PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,

floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels

      // vecteur déplacement

      PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant x des balles,

                              floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant y des balles

      // importantion d'un son

      SoundFile son_collision = new SoundFile(this, "canard.wav");

      // instanciation de l'objet "balle"

      mesBalles[i] = new balle( p, //vecteur position
```

```
                    d, // vecteur déplacement
floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels ---- diamètre des balles,
color(random(255),255,255), // ---- couleur aléatoire des balles en représentation HSB,
                    son_collision) ; // son de la collision
  }
```

Remarque : Dans la boucle « ***for*** » précédente nous avons utilisé l'attribut « ***length*** » du tableau « ***mesBalles*** »qui vaut ici N = 10.

8.1.4 Faire vivre chaque balle dans le *draw()*

Dans la méthode *draw(),* il faut maintenant appliquer les méthodes *afficher(),avancer()* et *testCollision()* à chaque objet balles du tableau *mesBalles*. De même que précédemment, il faut utiliser une boucle « *for* » :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
   fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
   rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
// faire "vivre" chaque balle
   for (int i = 0; i<mesBalles.length ; i++){
// affichage de la balle
mesBalles[i].afficher() ;
      // faire avancer la balle
mesBalles[i].avancer() ;
        // Test de collision sur les bords
mesBalles[i].testCollision() ;
   }
}
```

Remarque 1 : Le code actuel est finalement beaucoup plus simple que celui effectué à la section 3.3 car il ne nous a pas été nécessaire de modifier les méthodes *afficher(), avancer()* et *testCollision()* en leur incorporant en argument l'indice de la balle. Chaque balle a ses méthodes propres et vit de façon autonome.

8.1.5 Programme final

Le corps du programme permettant de gérer les listes de balles est le suivant :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle[] mesBalles;
```

```
int N = 10; // nombre de balles
void setup(){// initialisation des paramètres d'affichage et des variables globales
// paramètres d'affichage
  size(500,500) ;
  background(#FF00E6) ;
//colorMode(HSB) ;
  // initialisation des variables globales
  //création du tableau de balles
  mesBalles = new balle[N];
//initialisation des balles
  for (int i = 0; i<mesBalles.length ; i++){
// vecteur position
    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
    floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
    // vecteur déplacement
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant x des balles,
     floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant y des balles
    // couleur
    color c = color(random(255),100,100); // teinte aléatoire en mode HSB
    // importantion d'un son
    SoundFile son_collision = new SoundFile(this, "canard.wav");
    // instanciation de l'objet "balle"
    mesBalles[i] = new balle( p, //vecteur position
                    d, // vecteur déplacement
                    floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels ---- diamètre des balles,
                    c, // ---- couleur aléatoire des balles en représentation HSB,
                    son_collision) ; // son de la collision
  }
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
```

```
    fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
    rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine


// faire "vivre" chaque balle
    for (int i = 0; i<mesBalles.length ; i++){
// affichage de la balle
        mesBalles[i].afficher() ;
        // faire avancer la balle
        mesBalles[i].avancer() ;
          // Test de collision sur les bords
        mesBalles[i].testCollision() ;
    }
}
```

8.1.6 Discussion sur les deux boucles« *for* » utilisées

Nous avons utilisé deux boucles « for » pour initialiser et faire vivre les balles. Dans chacune des boucles, nous avons utilisé une écriture de « bas niveau » qui implique l'utilisation d'un indice entier « *i* » qui varie de la valeur initiale 0 à la valeur finale N =10.

L'algorithme d'initialisation des balles peut se lire de la manière suivante :

Pour chaque balle i, i allant de 0 à N :

- *Créer :*
  - *Un vecteur position aléatoire,*
  - *Un vecteur déplacement aléatoire*
  - *Un diamètre aléatoire*
  - *Une couleur aléatoire*
- *Instancier la $i^{ème}$ balle du tableau en utilisant les variables précédemment créée.*

De même, l'algorithme permettant la vie des balles est le suivant :

Pour chaque balle i, i allant de 0 à N :

- *Afficher la balle i,*
- *Faire avancer la balle i*
- *Tester la collision de la balle i.*

On peut remarquer qu'à chaque balle, les mêmes méthodes sont appliquées, et que l'indice i intervient uniquement pour préciser le numéro de la balle considérée, mais n'intervient pas directement dans les méthodes utilisées. Il existe une écriture différente de ces algorithme impliquant une vision plus « haut niveau » de la programmation, qui n'utilise plus l'indice i. C'est l'objet de la section suivante.

8.1.7 Une nouvelle écriture élégante de la boucle for

Il existe une écriture plus élégante de la boucle for qui ne fait plus intervenir l'indice i de la balle. Seul l'algorithme d'initialisation peut se réécrire de la manière suivante :

Pour chaque balle b du tableau de balles :

- *Afficher la balle b,*
- *Faire avancer la balle b*
- *Tester la collision de la balle b.*

Le code Processing correspondant est le suivant :

```
// faire "vivre" chaque balle
  for (balle b : mesBalles){
   // affichage de la balle
b.afficher() ;
    // faire avancer la balle
b.avancer() ;
      // Test de collision sur les bords
b.testCollision() ;
  }
```

8.2 Un nombre variable de balles : tableaux dynamiques et ArrayList

8.2.1 Evolution de l'algorithme de « vie » des balles

Il serait intéressant de partir d'un nombre N de balles et de faire évoluer ce nombre. Il faudrait alors gérer des tableaux de balles dont le nombre d'éléments qui le compose est variable. On parle alors de **tableaux dynamiques**.

Voici les nouvelles fonctionnalités qu'il est possible d'incorporer au programme :

- *Lorsque l'utilisateur clique sur la touche ENTER : Faire apparaitre une nouvelle balle.*
- *Lorsque l'utilisateur clique sur la touche ESPACE : Faire disparaitre une balle.*

8.2.2 Présentation de la classe ArrayList

En JAVA et Processing, il est possible de gérer les tableaux (ou listes) dynamiques d'objet quelconque grâce à la classe ***ArrayList*.**La documentation Processing fournit un exemple précis, mais ne détaille pas les différents attributs et méthodes de cette classe :

Name ArrayList

Examples

```
// These are code fragments that show how to use an ArrayList.
// They won't compile because they assume the existence of a Particle class.

// Declaring the ArrayList, note the use of the syntax "<Particle>" to indicate
// our intention to fill this ArrayList with Particle objects
ArrayList<Particle> particles = new ArrayList<Particle>();

// Objects can be added to an ArrayList with add()
particles.add(new Particle());

// Particles can be pulled out of an ArrayList with get()
Particle part = particles.get(0);
part.display();

// The size() method returns the current number of items in the list
int total = particles.size();
println("The total number of particles is: " + total);

// You can iterate over an ArrayList in two ways.
// The first is by counting through the elements:
for (int i = 0; i < particles.size(); i++) {
  Particle part = particles.get(i);
  part.display();
}

// The second is using an enhanced loop:
for (Particle part : particles) {
  part.display();
}

// You can delete particles from an ArrayList with remove()
particles.remove(0);
println(particles.size()); // Now one less!

// If you are modifying an ArrayList during the loop,
// then you cannot use the enhanced loop syntax.
// In addition, when deleting in order to hit all elements,
// you should loop through it backwards, as shown here:
for (int i = particles.size() - 1; i >= 0; i--) {
  Particle part = particles.get(i);
  if (part.finished()) {
    particles.remove(i);
  }
}
```

Un détail précis de la classe est toutefois fourni dans la JAVADoc :

Description An `ArrayList` stores a variable number of objects. This is similar to making an array of objects, but with an `ArrayList`, items can be easily added and removed from the ArrayList and it is resized dynamically. This can be very convenient, but it's slower than making an array of objects when using many elements. Note that for resizable lists of integers, floats, and Strings, you can use the Processing classes IntList, FloatList, and StringList.

An ArrayList is a resizable-array implementation of the Java List interface. It has many methods used to control and search its contents. For example, the length of the `ArrayList` is returned by its `size()` method, which is an integer value for the total number of elements in the list. An element is added to an `ArrayList` with the `add()` method and is deleted with the `remove()` method. The `get()` method returns the element at the specified position in the list. (See the above example for context.)

For a list of the numerous `ArrayList` features, please read the Java reference description.

Constructor

```
ArrayList<Type>()
ArrayList<Type>(initialCapacity)
```

Parameters

`Type`	Class Name: the data type for the objects to be placed in the ArrayList.
`initialCapacity`	int: defines the initial capacity of the list; it's empty by default

Cette page est accessible à l'adresse suivante: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html.

java.util

Class ArrayList<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

Dans cette classe, l'attribut que nous allons utiliser est le suivant :

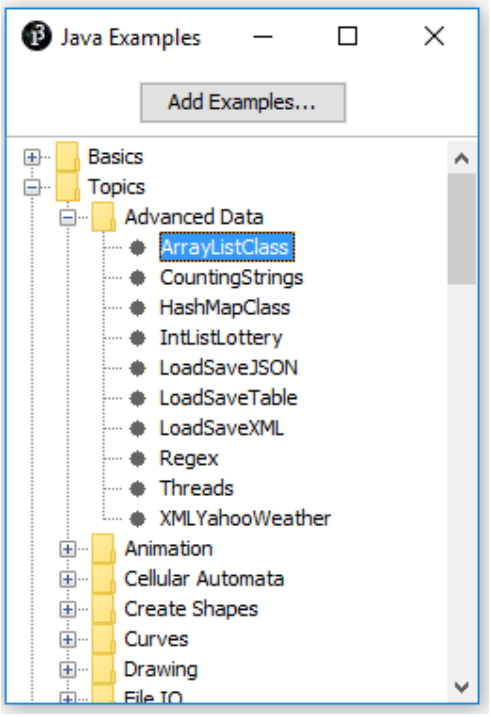
- *« **size** »* qui précise la taille de l'ArrayList

Les méthodes utilisées sont les suivantes :

- ***get(int i)*** qui permet de sélectionner le i^{ème} élément de la liste.
- ***add(E o)***qui rajoute un objet « o » de type E à la liste.

L'exemple de Processing sur les ArrayList est simple et bien expliqué et mérite de s'y attarder :

- https://www.openprocessing.org/sketch/95796
- Dans l'IDE de Processing : Fichier/Exemple/Topics/Advanced Data/ArrayList



8.2.3 Mise en place du tableau dynamique de balles

8.2.3.1 Déclaration du tableau dynamique de type ArrayList

La déclaration du tableau dynamique de type *ArrayList, contenant des objets de type* ***<balle>,*** s'effectue dans les variables globales :

```
// appel aux librairies

import processing.sound.*;

// déclaration des variables globales

ArrayList<balle> mesBalles;

int N = 10; // nombre de balles
```

8.2.3.2 Création du tableau dynamique dans le setup()

On crée un crée un tableau dynamique qui ne comporte aucun objet pour le moment. Chaque objet est de type **<balle>** :

```
// initialisation des variables globales

  //création de l'ArrayList

mesBalles = new ArrayList<balle>();
```

8.2.3.3 Création et initialisation des balles et remplissage du tableau dynamique dans le setup()

Dans cette étape, on crée N balles (dont la position, le déplacement, le diamètre et la couleur sont aléatoire comme précédemment), et on remplit le tableau dynamique grâce à la méthode *add()* de la classe *ArrayList* :

```
//création et initialisation des N balles et remplissage de l'ArrayList avec les balles créées

for (int i = 0; i<N ; i++){

// vecteur position

    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,

     floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels

    // vecteur déplacement

    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels suivant x des balles,

    floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels suivant y des balles

    // importantion d'un son

    SoundFile son_collision = new SoundFile(this, "canard.wav");

    // instanciation de l'objet "balle"

    balle b = new balle( p, //vecteur position

                         d, // vecteur déplacement

                         floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels --- diamètre des balles,
```

```
                    color(random(255),255,255), // ---- couleur aléatoire des balles en représentation HSB,
                    son_collision) ; // son de la collision


// ajout d'un objet balle à l'ArrayList
     mesBalles.add(b);
 }
```

A la fin de cette étape, le tableau dynamique *mesBalles* comporte N balles.

8.2.3.4 Création d'une méthode pour créer une balle d'attributs aléatoires

Pour simplifier l'écriture du programme et permettre plus facilement la création de balles, il est possible de créer une méthode spécifique pour la création de balle dont les attributs sont aléatoires :

Nom : creerBalle

Paramètre d'entrée : rien

Paramètre de sortie : objet de type balle

Fonction :

- *Créer :*
  - *Un vecteur position aléatoire,*
  - *Un vecteur déplacement aléatoire*
  - *Un diamètre aléatoire*
  - *Une couleur aléatoire*
- *Instancier la $i^{ème}$ balle du tableau en utilisant les variables précédemment créée.*

Pour créer cette méthode il suffit simplement de copier le code en bleu de la section précédente et de le coller dans le corps de la méthode *creerBalle()* :

```
balle creerBalle(){
  // vecteur position
    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
    floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
    // vecteur déplacement
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels suivant x des balles,
     floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels suivant y des balles
    // importantion d'un son
    SoundFile son_collision = new SoundFile(this, "canard.wav");
    // instanciation de l'objet "balle"
    balle b = new balle( p, //vecteur position
```

```
                    d, // vecteur déplacement
                    floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels --- diamètre des balles,
                    color(random(255),255,255), // ---- couleur aléatoire des balles en représentation HSB,
                    son_collision) ; // son de la collision
return b; // renvoyer la balle b créée
}
```

Ainsi, dans la méthode *draw(),* le code permettant le remplissage du tableau dynamique est grandement simplifié en faisant appel à la méthode *creerBalle()* :

```
//initialisation des N balles et remplissage de l'ArrayList avec les balles créées
for (int i = 0; i<N ; i++){
// Création d'une nouvelle balle d'attributs aléatoires
    balle b = creerBalle();
    // ajout d'un objet balle à l'ArrayList
    mesBalles.add(b);
  }
```

Remarque : En toute rigueur, la méthode *creerBalle()* n'est pas réellement nécessaire. En effet, il aurait suffi de créer un constructeur prenant en argument la seule variable que l'utilisateur doit définir, c'est-à-dire le nom du son associé aux collisions. La classe balle serait alors polymorphe.

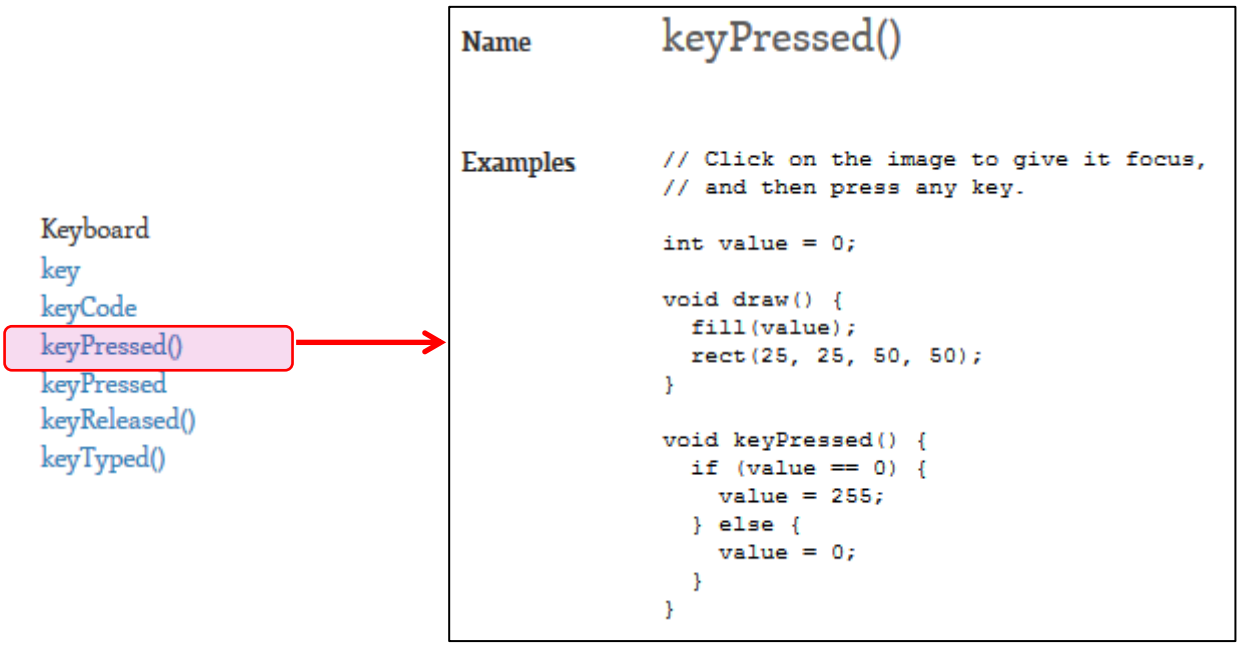
8.2.3.5 Vie de chaque balle dans le draw()

A ce stade, la boucle for permettant d'appeler les méthodes *afficher()*, *avancer()* et *testCollision()* reste inchangée car les ArrayList peuvent se traiter comme des tableaux classiques.

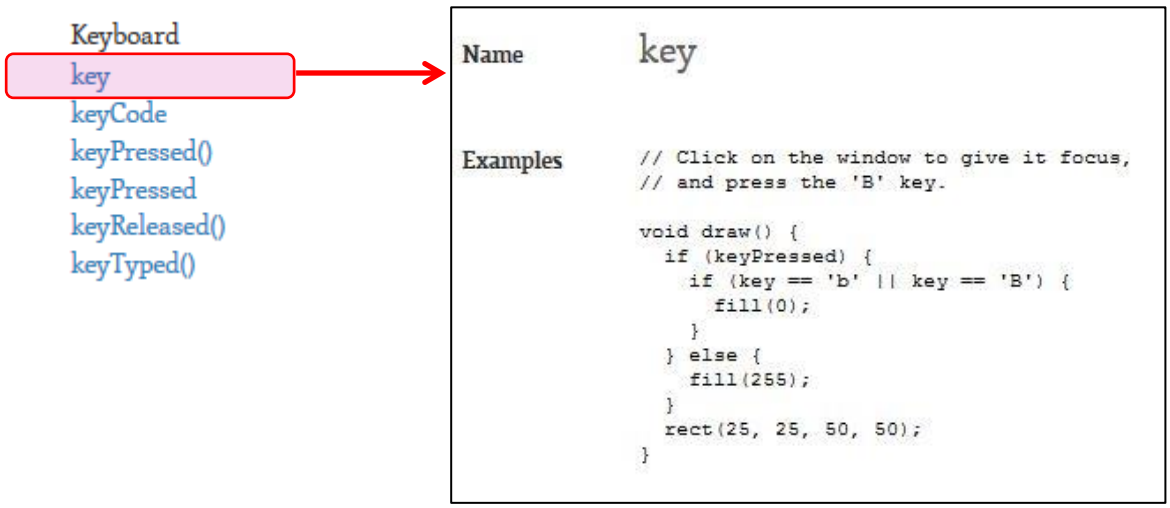
Il convient de rajouter au programme les deux nouvelles fonctionnalités suivantes:

- *Lorsque l'utilisateur clique sur la touche ENTER : Faire apparaitre une nouvelle balle.*
- *Lorsque l'utilisateur clique sur la touche ESPACE : Faire disparaitre une balle.*

Pour cela il faut faire appel à méthode ***keyPressed()*** de Processing. La documentation relative à cette fonction est disponible dans la section « keyboard » de la partie « Input » :



Il faudra faire appel ensuite à la variable ***key*** de Processing permettant de gérer les touches « espace » et « ENTER » :



La variable ***key*** de Processing permet aussi de gérer les touches classiques telles que « a », « b », etc… Les touches *BACKSPACE, TAB, ENTER, RETURN, ESC,* et *DELETE,* sont accessibles via la variable ***keyCode***.

8.2.3.5.1 Création d'une balle

Le code correspondant à l'algorithme :

- *Lorsque l'utilisateur clique sur la touche ENTER : Faire apparaitre une nouvelle balle,*

est le suivant :

```
void keyPressed(){
   if (key == ENTER){
     //création d'une nouvelle balle dont les attributs sont aléatoires
      balle b = creerBalle();
      // ajout d'un objet balle à l'ArrayList mesBalles
      mesBalles.add(b);
   }
}
```

8.2.3.5.2 Elimination d'une balle

Le code correspondant à l'algorithme :

- *Lorsque l'utilisateur clique sur la touche ESPACE : Faire disparaitre une balle.*

est le suivant :

```
void keyPressed(){
   if (key == ENTER){
     //création d'une nouvelle balle dont les attributs sont aléatoires
      balle b = creerBalle();
      // ajout d'un objet balle à l'ArrayList
mesBalles.add(b);
   }
if (key == ' '){ // touche espace appuyée
     if(mesBalles.size()>0){
          mesBalles.remove(0);
     }
   }
}
```

8.2.4 Programme final

Le code final est le suivant :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
ArrayList<balle> mesBalles;
int N = 1; // nombre de balles
void setup(){// initialisation des paramètres d'affichage et des variables globales
// paramètres d'affichage
  size(500,500) ;
  background(#FF00E6) ;
//colorMode(HSB);
  // initialisation des variables globales
  //création de l'ArrayList
  mesBalles = new ArrayList<balle>();
  //initialisation des N balles et remplissage de l'ArrayList avec les balles créées
for (int i = 0; i<N ; i++){
// création d'une nouvelle balle d'attributs alléatoires
    balle b = creerBalle();
    // ajout d'un objet balle à l'ArrayList
    mesBalles.add(b);
  }
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
   fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
   rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine

   // faire "vivre" chaque balle
   for (balle b : mesBalles){
    // affichage de la balle
```

```
        b.afficher() ;
        // faire avancer la balle
        b.avancer() ;
          // Test de collision sur les bords
        b.testCollision() ;
    }
}
balle creerBalle(){
  // vecteur position
    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
     floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
    // vecteur déplacement
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels suivant x des balles,
    floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels suivant y des balles
    // importantion d'un son
    SoundFile son_collision = new SoundFile(this, "canard.wav");
    // instanciation de l'objet "balle"
    balle b = new balle( p, //vecteur position
                         d, // vecteur déplacement
                         floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels --- diamètre des balles,
                         color(random(255),255,255), // ---- couleur aléatoire des balles en représentation HSB,
                         son_collision) ; // son de la collision
    return b;
}
void keyPressed(){
   if (key == ENTER){
     //création d'une nouvelle balle dont les attributs sont aléatoires
      balle b = creerBalle();
      // ajout d'un objet balle à l'ArrayList
      mesBalles.add(b);
   }
```

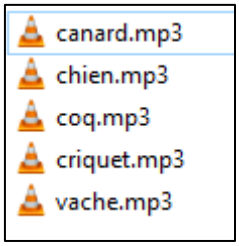
```
    if (key == ' '){ // touche espace appuyée
      if(mesBalles.size()>0){
mesBalles.remove(0); // on élimine le 1er objet balle du ArrayList
      }
    }
}
```

9 Lecture d'un fichier texte pour importer une liste de sons

9.1 Présentation du problème

Jusqu'à présent, nous n'avons utilisé qu'un seul son lors de la collision des balles avec les bords. Supposons que nous souhaitions maintenant attribuer un son spécifique à chaque balle, choisi aléatoirement parmi un ensemble de sons préalablement chargé dans le dossier Data du sketch en cours. Il suffit pour cela :

1) Enregistrer une série de son dans un **dossier « sons » placé dans le dossier « Data » du sketch** :



2) D'écrire les noms des fichiers sons dans un fichier texte nommé par exemple « nomDesSons.txt », placé dans le dossier « Data » :

```
Canard.mp3

Chien.mp3

Coq.mp3

Criquet.mp3
```

3) De lire les noms des fichiers sons dans le sketch et de les mémoriser dans une liste de chaines de caractère de type *String*. Cette opération peut s'effectuer grâce à la méthode *LoadStrings()* de Processing.
4) D'utiliser la liste des noms des sons pour charger aléatoirement les fichiers sonores.

9.2 Codage de l'algorithme

9.2.1 Déclaration de la liste des noms des sons dans les variables globales

Il faut tout d'abord déclarer liste des noms des sons dans les variables globales. Appelons cette liste « *nomDesSon* » :

```
// appel aux librairies

import processing.sound.*;

// déclaration des variables globales
```

```
ArrayList<balle> mesBalles;
int N = 1; // nombre de balles
String[] nomDesSons; // liste des noms des sons
```

9.2.2 Chargement des noms des fichiers sons dans le setup()

Il faut ensuite charger les noms des fichiers sons à partir du fichier « nomDesSons.txt » placé dans le dossier « Data » :

```
// initialisation des variables globales
// chargement des noms des fichiers sons
  nomDesSons = loadStrings("nomDesSons.txt");
  //création de l'ArrayList
  mesBalles = new ArrayList<balle>();
…
```

9.2.3 Modification de la méthode *creerBalle()* pour permettre le chargement aléatoire des sons

Dans la méthode *creerBalle()* qui permet l'instanciation d'une balle avec des attributs aléatoires, il convient maintenant de permettre l'importation d'un son aléatoire dans l'ensemble des sons disponibles :

```
// Importation d'un son aléatoire dans l'ensemble des sons disponibles
    int i = floor(random(0,nomDesSons.length)); // indice aléatoire entre 0 et le nombre de fichiers sons
    String nomDuSon = "sons/"+ nomDesSons[i]; // chemin complet pour aller chercher le fichier son
    SoundFile son_collision = new SoundFile(this, nomDuSon); // chargement du son
```

9.2.4 Programme final

Le programme final est le suivant :

```
// Appel aux librairies
import processing.sound.*;
// Déclaration des variables globales
ArrayList<balle> mesBalles;
int N = 1; // nombre de balles
String[] nomDesSons; // liste des noms des sons
void setup(){// initialisation des paramètres d'affichage et des variables globales
  // Paramètres d'affichage
  size(500,500) ;
  background(#FF00E6) ;
  // Initialisation des variables globales
  // Chargement des noms des fichiers sons
```

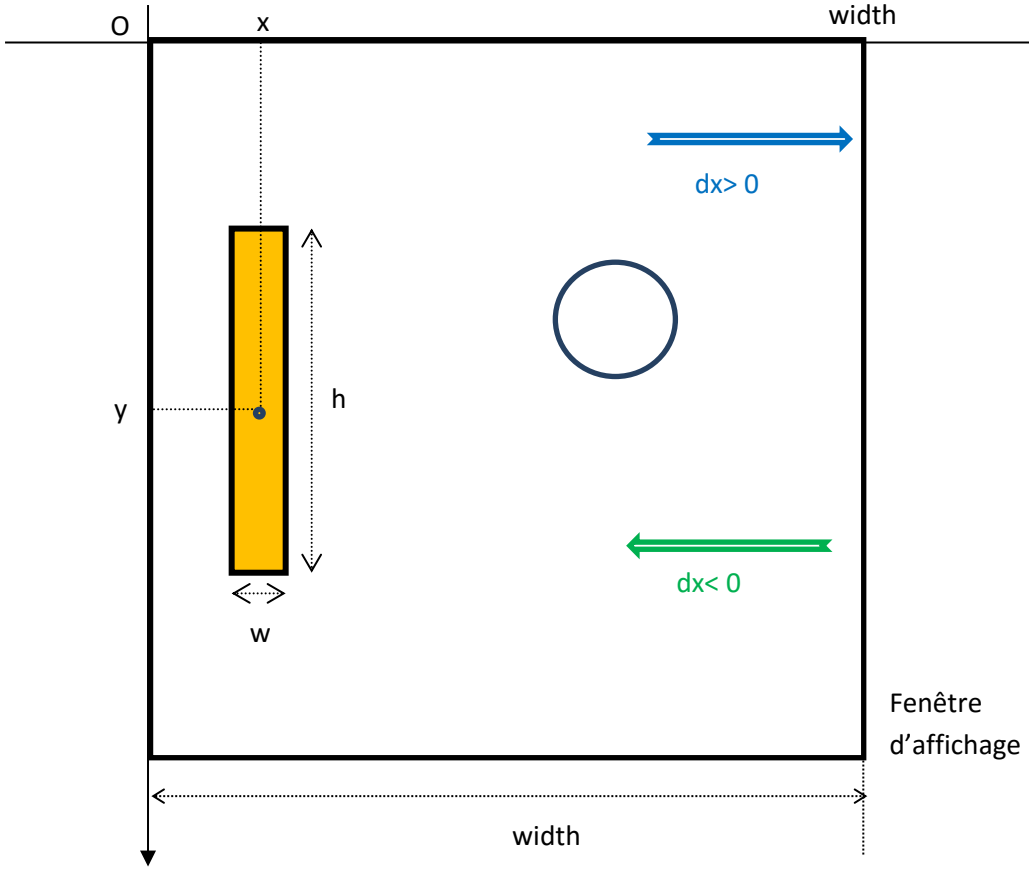
```
   nomDesSons = loadStrings("nomDesSons.txt");
   //création de l'ArrayList
   mesBalles = new ArrayList<balle>();
   //initialisation des N balles et remplissage de l'ArrayList avec les balles créées
for (int i = 0; i<N ; i++){
// création d'une nouvelle balle d'attributs aléatoires
     balle b = creerBalle();
     // ajout d'un objet balle à l'ArrayList
     mesBalles.add(b);
   }
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
   // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
    fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
    rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
    // faire "vivre" chaque balle
    for (balle b : mesBalles){
      // affichage de la balle
        b.afficher() ;
        // faire avancer la balle
        b.avancer() ;
          // Test de collision sur les bords
        b.testCollision() ;
    }
}
balle creerBalle(){
  // vecteur position
     PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
    floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
     // vecteur déplacement
```

```
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels suivant x des balles,
     floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels suivant y des balles
        // Importation d'un son aléatoire dans l'ensemble des sons disponibles
    int i = floor(random(0,nomDesSons.length)); // indice aléatoire entre 0 et le nombre de fichier son
    String nomDuSon = "sons/"+ nomDesSons[i]; // chemin complet pour aller chercher le fichier son
    SoundFile son_collision = new SoundFile(this, nomDuSon); // chargement du son
    // Instanciation de l'objet "balle"
    balle b = new balle( p, //vecteur position
                         d, // vecteur déplacement
                         floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels --- diamètre des balles,
                         color(random(255),255,255), // ---- couleur aléatoire des balles en représentation HSB,
                         son_collision) ; // son de la collision
return b;
}
void keyPressed(){
   if (key == ENTER){
//création d'une nouvelle balle dont les attributs sont aléatoires
      balle b = creerBalle();
      // ajout d'un objet balle à l'ArrayList
      mesBalles.add(b);
   }
   if (key == ' '){ // touche espace appuyée
     if(mesBalles.size()>0){
       mesBalles.remove(0);
     }
   }
}
```

10 Création et utilisation d'un objet « Raquette »

10.1 Description de l'objet « raquette »

On peut considérer la raquette comme un rectangle dont le centre est situé à la position (*x,y*), de largeur *w* (pour *width*) et de hauteur *h* (pour *height*).



La distance de la raquette par rapport à l'axe (Oy) doit rester fixe, par exemple 20 pixels. Dans un premier temps, il sera plus facile de gérer la position verticale grâce à la sourie. Pour faire déplacer la raquette, il suffira donc d'afficher la raquette à la même position que la sourie.

Dans la classe « Raquette », il ne sera pas nécessaire de prévoir une méthode pour gérer la collision avec les balles. En effet, le test de collision avec la raquette peut directement s'intégrer dans la méthode *testColision()* de chaque balle.

L'architecture de la classe « Raquette » est la suivante :

Classe « Raquette »

PARAMETRES

Attributs de la Raquette :

- Couleur : c
- largeur : w
- hauteur : h
- Position :
  - Abscisse : x
  - Ordonnée : v

CONSTRUCTEUR

Raquette()

ACTIONS

Méthodes de la Raquette :

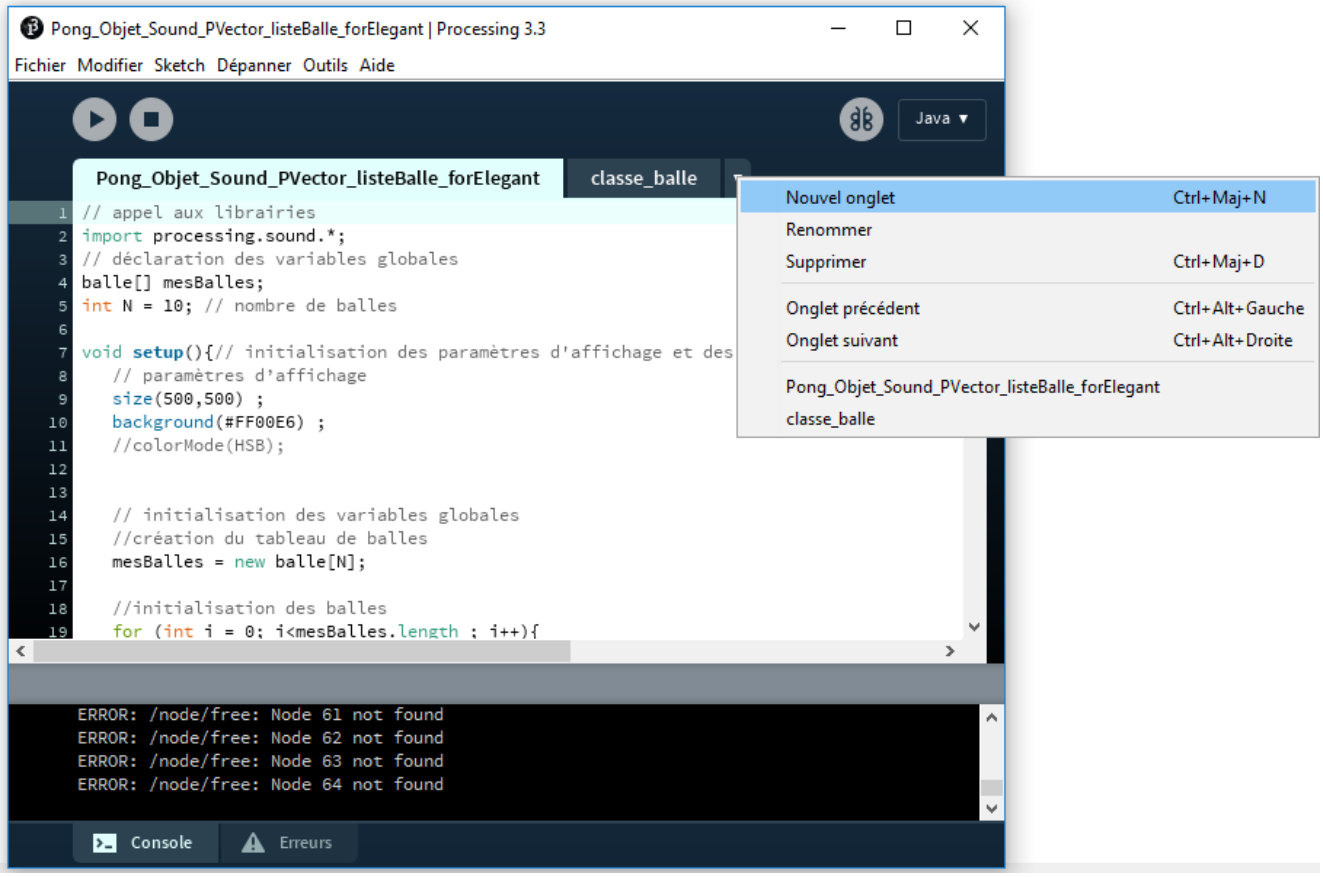
- afficher()

Nous pouvons remarquer cette classe ne comporte (pour le moment) ni d'attributs de déplacement *(dx, dy)*, ni de méthodes *avancer()* et *testCollision()*.

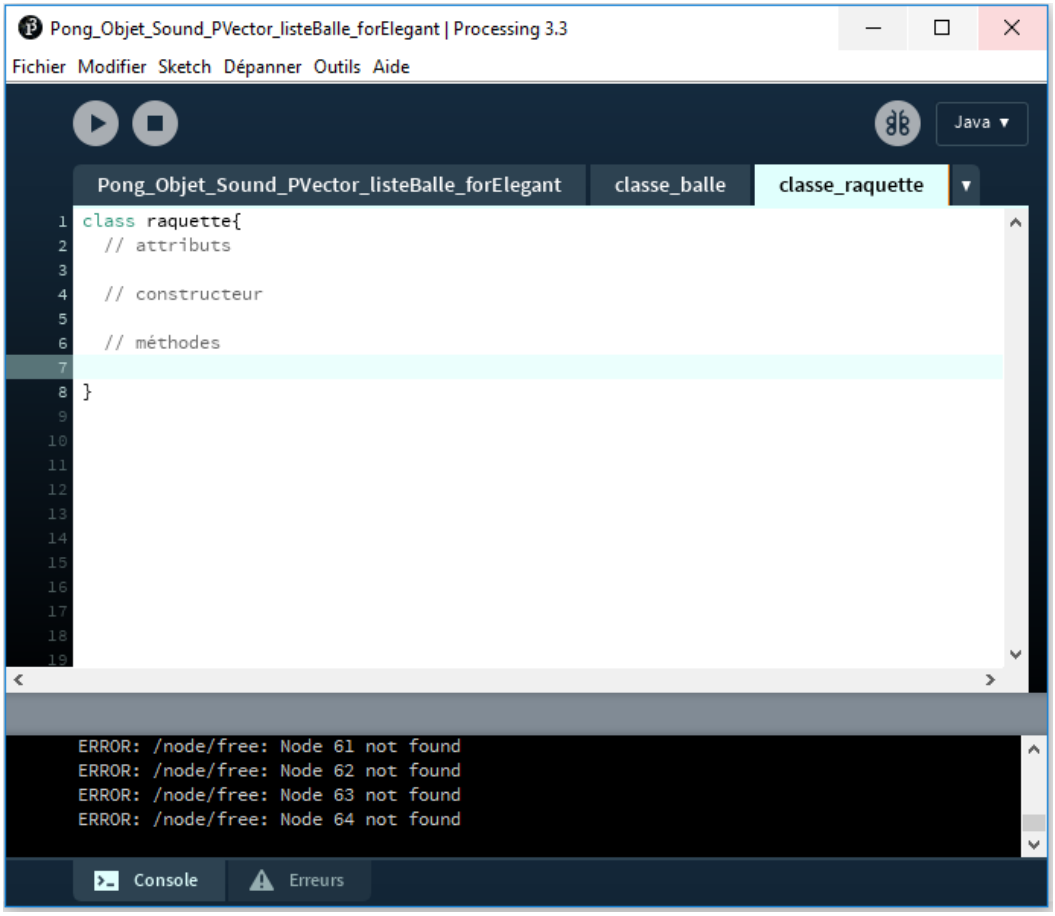
10.2 Ecriture de la classe raquette

En se basant sur l'architecture décrite précédemment, nous pouvons construire la classe « raquette » en respectant l'architecture globale d'une classe : attributs / constructeur / méthodes.

Il est plus pratique d'écrire cette classe dans un nouvel onglet nommé « classe_raquette » :



Architecture globale :



10.2.1 Attributs de la classe « raquette »

Pour simplifier la manipulation, nous choisissons de manipuler deux variables x et y indépendantes pour la position de la raquette :

```
class raquette{
  // attributs
  int x; // absisses du centre de la raquette
  int y; // ordonnée du centre de la raquette
  int w; // largeur
  int h; // hauteur
  color c; // couleur de la raquette
  // constructeur
  // méthodes
}
```

10.2.2 Constructeur de la classe « raquette »

Le constructeur sert à créer l'espace mémoire disponible pour une instance de la raquette et éventuellement à initialiser « manuellement » les attributs :

```
// constructeur 1
  raquette(int _x, int _y, int _w, int _h, color _c){
    x = _x;
    y = _y;
    w = _w;
    h = _h;
    c = _c;
  }
```

10.2.3 Deuxième constructeur de la classe « raquette » : polymorphisme

Pour éviter d'avoir à rentrer manuellement les valeurs des attributs de la raquette lors de l'instanciation de la classe raquette, nous pouvons créer un deuxième constructeur qui ne prend aucun argument comme paramètre d'entrée :

```
// constructeur 2
raquette(){
    x = 20; // 20pix du bord
    y = width/2; // au milieu de la fenêtre d'affichage
    w = 20; // 20 pixels de largeur
    h = 100; // 100 pixels de hauteur
    c = color(255,0,0); // rouge
  }
```

Remarque 1 : A la compilation, les deux constructeurs 1 et 2 seront différenciés par le nombre et le type des arguments qui les compose. Pour le constructeur 1 : int, int, int, int, color. Pour le constructeur 2 : aucun argument.

Remarque 2 : Le fait que la classe raquette est composée de deux constructeurs s'appelle le **polymorphisme** (*ploy* : plusieurs, *morphisme* : forme). On dit que la classe est polymorphe.

Remarque 3 : Les méthodes peuvent aussi être polymorphes. Voir par exemple la méthode *fill()* dans la documentation.

10.2.4 Méthodes de la classe « raquette »

La classe « raquette » ne comporte qu'une seule méthode : *affiche()*. Son but est de dessiner un rectangle de couleur *c* de largeur *w*, de hauteur *h*, de couleur *c*. Ceci est réalisé par la méthode rect() qui dessine un rectangle dont le point de référence est le point supérieur gauche. Pour que la raquette soit centrée sur la sourie, il faut donc que le coin supérieur gauche soit situé verticalement à une distance h/2 de l'ordonnée de la sourie :

```
// méthodes
void affiche(){
   fill(c);
  y = mouseY –h/2 ;
   rect(x,y,w,h);
 }
```

10.2.5 Ecriture finale de la classe « raquette »

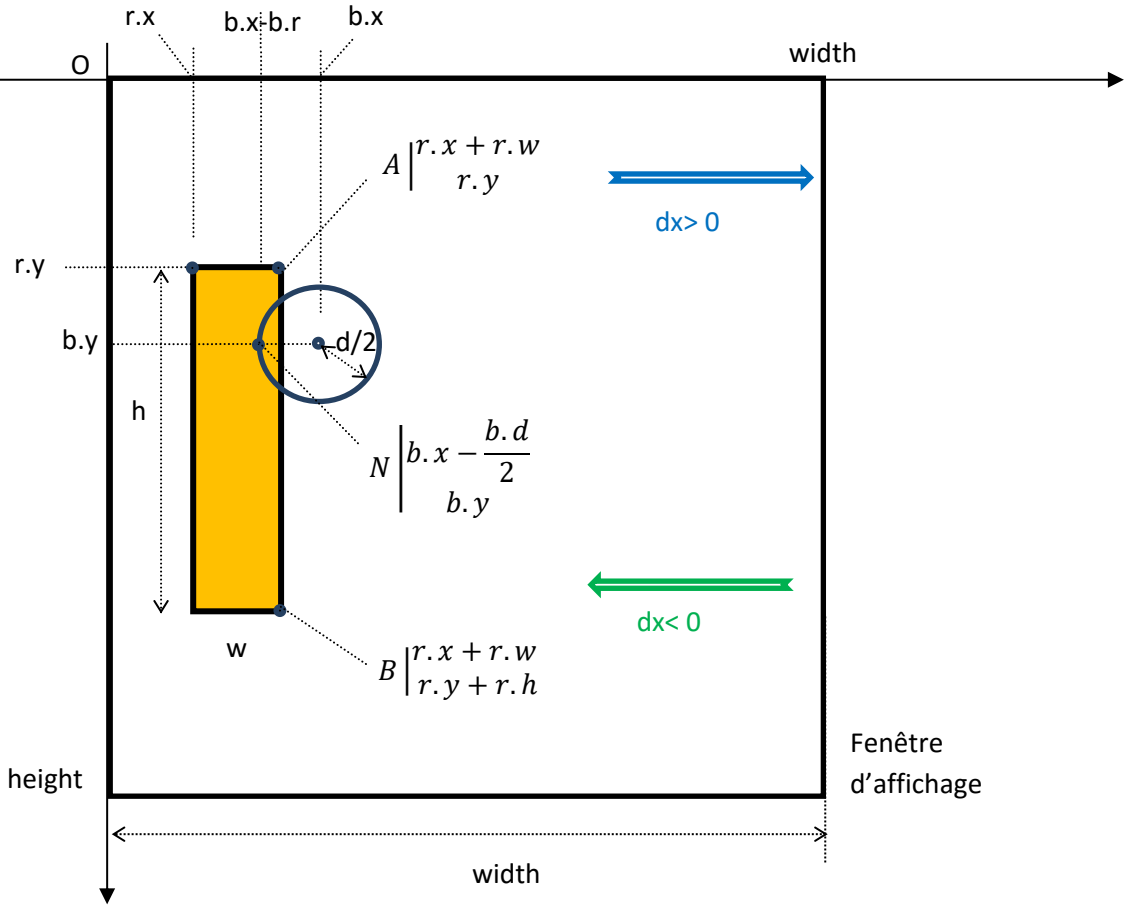
```
class raquette{
 // attributs
 int x; // absisses du centre de la raquette
 int y; // ordonnée du centre de la raquette
 int w; // largeur
 int h; // hauteur
 color c; // couleur de la raquette
 // constructeur 1
 raquette(int _x, int _y, int _w, int _h, color _c){
   x = _x;
   y = _y;
   w = _w;
   h = _h;
   c = _c;
 }
 // constructeur 2
 raquette(){
   x = 20; // 20pix du bord
   y = width/2; // au milieu de la fenêtre d'affichage
   w = 20; // 20 pixels de largeur
   h = 100; // 100 pixels de hauteur
```

```
    c = color(255,0,0); // rouge
  }
  // méthodes
  void affiche(){
    fill(c);
    y = mouseY –h/2; // affichage de la raquette à la position verticale de la raquette
    rect(x,y,w,h);
  }
}
```

10.3 Collision entre une balle et la raquette

Nous proposons ici de **modifier la méthode *testCollision()* de la classe « balle »**. Comme pour la collision entre la balle et les bords de la fenêtre d'affichage, il est nécessaire de faire un schéma pour bien comprendre la situation.

10.3.1 Schéma du problème



Pour qu'il y ait collision entre la balle et la raquette, il faut que le point N de la balle soit tangent ou traverse légèrement le bord droit de la raquette. Les conditions suivantes doivent **simultanément** être respectées :

Suivant x : l'abscisse du point N doit être inférieure à celle du point A et B, soit,

b.x – b.d/2< r.x+r.w

Suivant y : L'ordonnée du centre de la balle doit être comprise entre l'ordonnée du point A et l'ordonnée du point B, soit,

r.y<b.y <r.y + r.h .

Algorithmiquement, il faut donc que la condition **(r.y <b.y) et** que la condition **(b.y <r.y + r.h) et** que la condition **(b.x – b.d/2< r.x+r.w) soient vérifiées simultanément.** En programmation Processing, le « et logique » s'écrit « **&&** ».

Remarque : « b. » fait référence à un attribut de la balle et « r. » fait référence à un attribut de la raquette.

10.3.2 Modification de la méthode *testCollision()* de la classe « balle »

Dans un premier temps, nous proposons de conserver les collisions avec les bords de la fenêtre d'affichage et de rajouter la possibilité pour une balle de rebondir sur la raquette. Nous proposons aussi de supprimer le son lors de la collision entre les bords et une balle et d'introduire un son uniquement à la collision avec la raquette.

Pour qu'une balle puisse connaître la position de la raquette, il est nécessaire de fournir à la balle une **référence** sur la raquette. Nous allons donc créer une deuxième méthode *testCollision()* (polymorphe de la première) prenant en argument une variable de type *raquette*. Cet argument permettra à la balle d'accéder aux attributs *x*, *y*, *h*, et *w* de la raquette.

Cette nouvelle méthode *testCollision()* s'écrit de la manière suivante :

```
void testCollision(raquette r){
  // Test de collision suivant x
  if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
     dep.x  *= (-1) ; // changer de sens
     }
  }
  // Test de collision suivant y
  if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
     dep.y  *= (-1) ; // changer de sens
     }
  }
  // Test de collision avec la raquette
  if ( (pos.x - d/2 <r.x+r.w) &&// suivant x
(r.y<pos.y) && (pos.y<r.y+r.h)) { // suivant y
     dep.x  *= (-1) ; // changer de sens
     if (son != null){ // jouer un son
       son.play();
```

```
    }
  }
}
```

Remarque : en langage C, la référence d'une variable est nommée « pointeur ». Il s'agit de l'adresse de la variable dans la mémoire de l'ordinateur.

10.4 Instanciation de la classe raquette

Pour plus de simplicité, nous allons faire appel au constructeur 2 de la raquette qui nous évite de rentrer à la main les différentes valeurs des attributs de la raquette.

10.4.1 Déclaration d'une variable de type *raquette* dans les variables globales

Il faut d'abord déclarer une variable nommée « raq », de type *raquette* dans les variables globales :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle[] mesBalles;
int N = 10; // nombre de balles
// raquette
raquette raq;
```

10.4.2 Instanciation de la raquette dans le *setup()*

Il faut ensuite instancier la raquette dans le *setup()* :

```
void setup(){// initialisation des paramètres d'affichage et des variables globales
  // paramètres d'affichage
  size(500,500) ;
  background(#FF00E6) ;
// instanciation de la raquette
  raq = new raquette(); // appel au constructeur 2
…
```

Remarque : Pour pouvoir « customiser » la raquette avec une autre taille et une autre couleur, il faut faire appel au constructeur 1.

10.4.3 Faire vivre les balles et la raquette dans le *draw()*

Il faut ici afficher la raquette à la position verticale *mouseY*, et appeler la deuxième méthode *testCollision()* prenant en argument une référence sur la raquette :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
```

```
    fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
    rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
    // affichage de la raquette à la position MouseY
    raq.affiche();
    // faire "vivre" chaque balle
    for (int i = 0; i<mesBalles.length ; i++){
      // affichage de la balle
        mesBalles[i].afficher() ;
        // faire avancer la balle
        mesBalles[i].avancer() ;
          // Test de collision sur les bords
mesBalles[i].testCollision(raq) ;
    }
}
```

10.5 Programme final

Corps du programme :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle[] mesBalles;
int N = 10; // nombre de balles
// raquette
raquette raq;
void setup(){// initialisation des paramètres d'affichage et des variables globales
   // paramètres d'affichage
   size(500,500) ;
   background(#FF00E6) ;
     // instanciation de la raquette
   raq = new raquette();
   // initialisation des variables globales
   //création du tableau de balles
```

```
  mesBalles = new balle[N];
  //initialisation des balles
  for (int i = 0; i<mesBalles.length ; i++){
    // vecteur position
    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
                floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
    // vecteur déplacement
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant x des balles,
                floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant y des balles
    // couleur
    color c = color(random(255),100,100); // teinte aléatoire en mode HSB
    // importantion d'un son
    SoundFile son_collision = new SoundFile(this, "canard.wav");
    // instanciation de l'objet "balle"
    mesBalles[i] = new balle( p, //vecteur position //<>//
                d, // vecteur déplacement
                floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels ---- diamètre des balles,
                c, // ---- couleur aléatoire des balles en représentation HSB,
                son_collision) ; // son de la collision
  }
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
   fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
  rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
   // affichage de la raquette à la position MouseY
   raq.affiche();
   // faire "vivre" chaque balle
   for (int i = 0; i<mesBalles.length ; i++){
```

```
      // affichage de la balle
        mesBalles[i].afficher() ;
        // faire avancer la balle
        mesBalles[i].avancer() ;
          // Test de collision sur les bords
        mesBalles[i].testCollision(raq) ;
    }
}
```

Classe balle :

```
class balle {
   // Attributs
  PVector pos; // Vecteur position du centre de la balle
  PVector dep; // Vecteur déplacement du centre de la balle
  color c; // couleur de la balle
  int d; // diamètre de la balle
  SoundFile son ; // son associé à la balle
   // Constructeur 1
   balle(PVector _pos, PVector _dep, int _d, color _c){
     pos = _pos;
     dep = _dep;
     d = _d;
     c = _c;
   }
 // Constructeur 2
   balle(PVector _pos, PVector _dep, int _d, color _c, SoundFile _son){
     pos = _pos;
     dep = _dep;
     d = _d;
     c = _c;
     son = _son;
```

```
 }
 // Méthodes
 void afficher(){
  // affichage de la balle
   fill(231,245,87) ;
   ellipse(pos.x,pos.y,d,d) ;
}
void avancer(){
  // faire avancer la balle
  pos.x += dep.x ;
  pos.y += dep.y ;
}
void testCollision(){
  // Test de collision suivant x
  if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
      dep.x  *= (-1) ; // changer de sens
      if (son != null){
        son.play();
      }
     }
       // Test de collision suivant y
  if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
      dep.y  *= (-1) ; // changer de sens
      if (son != null){
        son.play();
      }
     }
}
void testCollision(raquette r){
  // Test de collision suivant x
  if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
      dep.x  *= (-1) ; // changer de sens
```

```
    }
    // Test de collision suivant y
    if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
        dep.y  *= (-1) ; // changer de sens
    }
    // Test de collision avec la raquette
    if ( (pos.x - d/2 < r.x+r.w) && // suivant x
          (r.y<pos.y) && (pos.y<r.y+r.h) ) { // suivant y
        dep.x  *= (-1) ; // changer de sens
        if (son != null){ // jouer un son
          son.play();
        }
    }
  }
}
```

Classe raquette :

```
class raquette{
  // attributs
  int x; // absisses du centre de la raquette
  int y; // ordonnée du centre de la raquette
  int w; // largeur
  int h; // hauteur
  color c; // couleur de la raquette
  // constructeur 1
  raquette(int _x, int _y, int _w, int _h, color _c){
    x = _x;
    y = _y;
    w = _w;
    h = _h;
    c = _c;
  }
```

```
  // constructeur 2
  raquette(){
    x = 20; // 20pix du bord
    y = width/2; // au milieu de la fenêtre d'affichage
    w = 20; // 20 pixels de largeur
    h = 100; // 100 pixels de hauteur
    c = color(255,0,0); // rouge
  }
```

```
  // méthodes
  void affiche(){
      fill(c);
      y = mouseY - h/2;
      rect(x,y,w,h);
  }
}
```

11 Collision entre les balles

11.1 Modèle physique simplifié

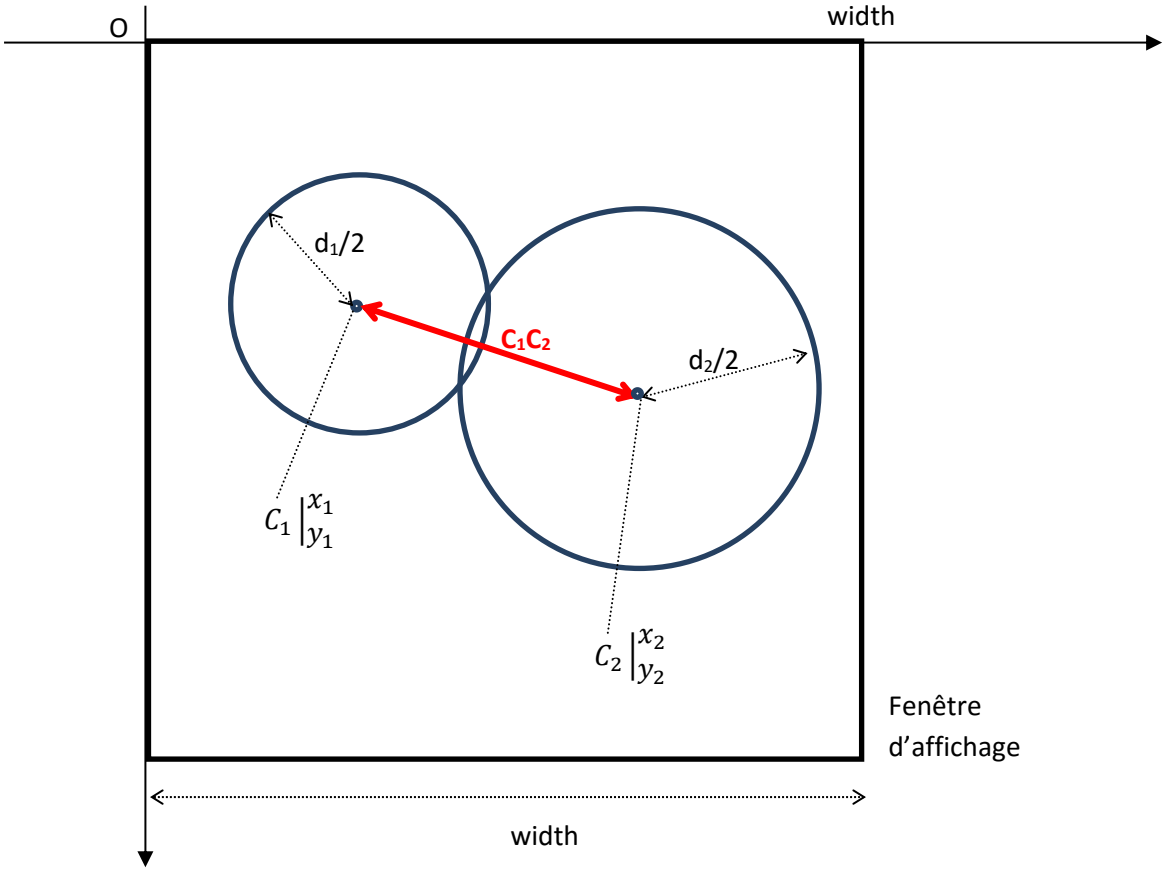
Pour rendre compte d'une réalité physique il faudrait ajouter la **masse** comme caractéristique de la balle. Le modèle physique qui traduit la collision entre objets n'est pas simple. Une première approche est de considérer que les chocs s'effectuent sans perte d'énergie. On parle alors de **choc élastique**.

D'après Wikipaedia : « *L'angle entre les directions des deux particules après le choc dépend de leurs masses et du choix du référentiel. Dans le référentiel où une des deux particules est initialement au repos, si les masses sont égales, les deux vitesses résultantes sont à angle droit l'une de l'autre, dans le cas où la particule incidente a une masse plus faible, cet angle est supérieur à l'angle droit, et dans le cas où la particule incidente est de masse plus grande, l'angle est inférieur à l'angle droit.* »

Pour simplifier le modèle, on **considèrera que lorsque deux balles entrent en collision, tout se passe comme si chacune des balles entre en collision avec un mur vertical**. Nous connaissons bien ce cas et l'avons traité dans les exemples précédents.

11.2 Schéma de la situation

Il y a collision entre deux balles lorsque la distance entre les centres des deux balles $\left\|\overrightarrow{C_1C_2}\right\| = C_1C_2$ est inférieure à la somme des rayons des deux balles : $\boldsymbol{C_1C_2 < \frac{d_1}{2} + \frac{d_2}{2}}$



Mathématiquement, la distance C_1C_2 est égale à :

$$\boldsymbol{C_1C_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

Par construction, les centres des balles sont repérés par des vecteurs positions $\overrightarrow{p_i}$. Sous Processing, il s'agit de variables de type *PVector* dont la classe associé comporte une méthode permettant de calculer la distance entre deux points : ***dist()***.

Il est possible de retrouver la documentation de cette méthode dans les références de Processing :

Class PVector

Name dist()

Examples

```
PVector  v1 = new PVector(10, 20, 0);
PVector  v2 = new PVector(60, 80, 0);
float d = v1.dist(v2);
println(d);  // Prints "78.10249"
```

```
PVector  v1 = new PVector(10, 20, 0);
PVector  v2 = new PVector(60, 80, 0);
float d = PVector.dist(v1, v2);
println(d);  // Prints "78.10249"
```

Description Calculates the Euclidean distance between two points (considering a point as a vector object).

Syntax

```
.dist(v)
.dist(v1, v2)
```

Parameters

v	PVector: the x, y, and z coordinates of a PVector
v1	PVector: any variable of type PVector
v2	PVector: any variable of type PVector

Returns float

11.3 Algorithme de la gestion des collisions entre balles

Les *N* balles sont mémorisées dans un tableau de balles qui permet de ne manipuler qu'une seule variable au lieu de *N*. Précédemment, nous avons appelé ce tableau « *mesBalles* ». L'algorithme permettant la gestion des collisions entre balles est le suivant :

Pour une balle i du tableau *mesBalles*, i allant de 1 à N :

Pour toutes les balles j, j allant de 1 à N, avec j ≠ i :

Calculer la distance d entre la balle i et la balle j ;

Si la distance d est inférieure à la somme des rayons des deux balles alors :

Les balles i et j changent de sens suivant l'axe x ;

Concrètement, le changement de sens s'écrira simplement de la manière suivante :

$dx_i \leftarrow (-1)*dx_i$

$dx_j \leftarrow (-1)*dx_j$

11.4 Modification de la méthode *testCollision()* de la classe « balle »

Nous allons intégrer l'algorithme précédent dans la méthode *testCollision()* de la classe balle prenant déjà en compte la collision avec la raquette. Comme dans la partie précédente traitant de la collision entre une balle et la raquette, pour que la balle considérée puisse connaitre la position des autres balles, il est nécessaire de lui donner une référence sur le tableau de balles. Il faudra alors introduire plusieurs argumentsdans la méthode *testCollision()* :

- Une variable étant un tableau de balles : *balle[] tabBalles*
- Une variable permettant de mémoriser le numéro de la balle considérée dans le tableau : *int k*.

La méthode *testCollision()* modifiée est la suivante :

```
void testCollision(raquette r, balle[] tabBalles, int k){
  // Test de collision suivant x
  if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
     dep.x  *= (-1) ; // changer de sens
  }
  // Test de collision suivant y
  if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
     dep.y  *= (-1) ; // changer de sens
  }
  // Test de collision avec la raquette
  if ( (pos.x - d/2 < r.x+r.w) && // suivant x
       (r.y<pos.y) && (pos.y<r.y+r.h) ) { // suivant y
     dep.x  *= (-1) ; // changer de sens
     if (son != null){ // jouer un son
       son.play();
     }
  }
  // test collision entre balles
  for(int i = 0; i<tabBalles.length; i++){
    if (i!=k){ // toutes les balles sauf la balle considérée d'indice k
      float d = PVector.dist(tabBalles[i].pos, this.pos); // distance entre la balle i et la balle k
```

```
        if (d<tabBalles[i].d/2+this.d/2){ // si la distance d entre les centres est plus petit que la somme des rayons
          tabBalles[i].dep.x *= (-1); // la ième part en sens inverse suivant x
          this.dep.x *= (-1); // la balle actuelle part en sens inverse suivant x
        }
      }
    }
  }
```

Remarque : Dans la partie concernant le test de collision entre les balles ci-dessus, nous avons utilisé la variable « ***this*** » qui est une référence sur l'objet actuellement traité (ici, la balle d'indice k passé en argument de la méthode). L'utilisation de la référence « ***this*** » est facultative mais permet de bien se rendre compte que les attributs auquel elle se réfère concernent bien l'objet actuellement traité.

11.5 Appel à la nouvelle méthode *testCollision()* dans le *draw()*

Il faut appeler maintenant la nouvelle méthode *testCollision()* dans le *draw()* :

```
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
  // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
   fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
   rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine

   // affichage de la raquette à la position MouseY
   raq.affiche();
   // faire "vivre" chaque balle
   for (int i = 0; i<mesBalles.length ; i++){
     // affichage de la balle
       mesBalles[i].afficher() ;
       // faire avancer la balle
       mesBalles[i].avancer() ;
         // Test de collision sur les bords
mesBalles[i].testCollision(raq,mesBalles,i) ;
   }
}
```

11.6 Programme final

Corps du programme :

```
// appel aux librairies
import processing.sound.*;
// déclaration des variables globales
balle[] mesBalles;
int N = 10; // nombre de balles
// raquette
raquette raq;
void setup(){// initialisation des paramètres d'affichage et des variables globales
  // paramètres d'affichage
  size(500,500) ;
  background(#FF00E6) ;
 // instanciation de la raquette
  raq = new raquette();
  // initialisation des variables globales
  //création du tableau de balles
  mesBalles = new balle[N];
  //initialisation des balles
  for (int i = 0; i<mesBalles.length ; i++){
    // vecteur position
    PVector p = new PVector(floor(random(50,width -50)),// génération d'abscisses aléatoires entre 50 pixels et width – 50 pixels,
                            floor(random(50,width -50))); // génération d'ordonnées aléatoires entre 50 pixels et width – 50 pixels
    // vecteur déplacement
    PVector d = new PVector(floor(random(1,5)), // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant x des balles,
                            floor(random(1,5))); // déplacement aléatoire entre 1 et 5 pixels ---- déplacement suivant y des balles
    // couleur
    color c = color(random(255),100,100); // teinte aléatoire en mode HSB
    // importantion d'un son
    SoundFile son_collision = new SoundFile(this, "canard.wav");
    // instanciation de l'objet "balle"
```

```
      mesBalles[i] = new balle( p, //vecteur position //<>//
                              d, // vecteur déplacement
                              floor(random(10,50)), // diamètre aléatoire entre 10 et 50 pixels ---- diamètre des balles,
                              c, // ---- couleur aléatoire des balles en représentation HSB,
                              son_collision) ; // son de la collision
   }
}
void draw(){// boucle d'affichage (par défaut: 30 fois/s)
   // feuille vierge avec une transparence pour l'effacement progressif de la trace de la balle
    fill(#FF00E6, 10) ; // couleur du fond avec 10% de transparence (2ieme paramètre)
    rect(0,0,width,height) ; // rectangle de la taille de la fenêtre d'affichage (width, height) dont le coin supérieur gauche est à l'origine
    // affichage de la raquette à la position MouseY
    raq.affiche();
    // faire "vivre" chaque balle
    for (int i = 0; i<mesBalles.length ; i++){
      // affichage de la balle
       mesBalles[i].afficher() ;
       // faire avancer la balle
       mesBalles[i].avancer() ;
         // Test de collision sur les bords
       mesBalles[i].testCollision(raq,mesBalles,i) ;
    }
}
```

Classe Balle :

```
class balle {
   // Attributs
  PVector pos; // Vecteur position du centre de la balle
  PVector dep; // Vecteur déplacement du centre de la balle
  color c; // couleur de la balle
  int d; // diamètre de la balle
  SoundFile son ; // son associé à la balle
```

```
  // Constructeur 1
  balle(PVector _pos, PVector _dep, int _d, color _c){
    pos = _pos;
    dep = _dep;
    d = _d;
    c = _c;
  }
// Constructeur 2
  balle(PVector _pos, PVector _dep, int _d, color _c, SoundFile _son){
    pos = _pos;
    dep = _dep;
    d = _d;
    c = _c;
    son = _son;
  }
  // Méthodes
  void afficher(){
   // affichage de la balle
    fill(231,245,87) ;
    ellipse(pos.x,pos.y,d,d) ;
 }
 void avancer(){
   // faire avancer la balle
   pos.x += dep.x ;
   pos.y += dep.y ;
 }
 void testCollision(){
   // Test de collision suivant x
   if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
       dep.x  *= (-1) ; // changer de sens
       if (son != null){
         son.play();
```

```
      }
     }
      // Test de collision suivant y
  if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
     dep.y  *= (-1) ; // changer de sens
     if (son != null){
       son.play();
     }
    }
}
void testCollision(raquette r, balle[] tabBalles, int k){
  // Test de collision suivant x
  if ( (pos.x + d/2  >width ) || ( pos.x - d/2 < 0 ) ) {
     dep.x  *= (-1) ; // changer de sens
  }
  // Test de collision suivant y
  if ( (pos.y + d/2  >height ) || ( pos.y - d/2 < 0 ) ) {
     dep.y  *= (-1) ; // changer de sens
  }
  // Test de collision avec la raquette
  if ( (pos.x - d/2 < r.x+r.w) && // suivant x
       (r.y<pos.y) && (pos.y<r.y+r.h) ) { // suivant y
     dep.x  *= (-1) ; // changer de sens
     if (son != null){ // jouer un son
       son.play();
     }
  }
  // test collision entre balles
  for(int i = 0; i<tabBalles.length; i++){
    if (i!=k){ // toutes les balles sauf la balle considérée d'indice k
      float d = PVector.dist(tabBalles[i].pos, this.pos); // distance entre la balle i et la balle k
      if (d<tabBalles[i].d/2+this.d/2){ // si la distance d entre les centres est plus petit que la somme des rayons
```

```
        tabBalles[i].dep.x *= (-1); // la ième part en sens inverse suivant x
        this.dep.x *= (-1); // la balle actuelle part en sens inverse suivant x
       }
      }
    }
  }
}
```

Classe Raquette : (inchangée)

```
class raquette{
  // attributs
  int x; // absisses du centre de la raquette
  int y; // ordonnée du centre de la raquette
  int w; // largeur
  int h; // hauteur
  color c; // couleur de la raquette
  // constructeur 1
  raquette(int _x, int _y, int _w, int _h, color _c){
    x = _x;
    y = _y;
    w = _w;
    h = _h;
    c = _c;
  }
  // constructeur 2
  raquette(){
    x = 20; // 20pix du bord
    y = width/2; // au milieu de la fenêtre d'affichage
    w = 20; // 20 pixels de largeur
    h = 100; // 100 pixels de hauteur
```

```
    c = color(255,0,0); // rouge
  }
  // méthodes
  void affiche(){
     fill(c);
     y = mouseY - h/2;
     rect(x,y,w,h);
  }
}
```

Le résultat est très satisfaisant.

12 Conclusion générale

Ce document est un bilan des cinq années passées en DSAA dans lesquelles j'ai pu développer l'apprentissage des notions fondamentales de programmation autour de l'élaboration du jeu PONG.

Vous remarquerez que le jeu PONG n'est finalement pas très abouti ce qui est volontaire. En effet, à la fin du document, le lecteur pourra faire évoluer les différents bouts de code pour peaufiner le jeu. Mon objectif était surtout de présenter les notions importantes autour d'une application graphique attrayante. Chaque modification de code entraine un résultat graphique et sonore immédiat.

Un lecteur expérimenté (un enseignant) pourra bien sûr choisir un chemin différent du mien dans la conduite des notions abordées.

J'ai pu tester cette méthode en seconde option ICN, en spécialité ISN de la terminale S, en BTS Design Graphique 1ère et 2ième année, ainsi qu'en Diplôme Supérieur d'Arts Appliqués (DSAA) 1ère et 2ième année. J'ai été surpris par l'attrait important que cette méthode a suscité chez les personnes concernées par ce cours et qui avaient entre 15 et 27 ans.

Cette méthode marche quel que soit le niveau et l'âge du public. Il suffit d'adapter les notions théoriques au public.

Un travail supplémentaire serait plutôt dédié à un public scientifique qui pourrait étudier plus précisément les collisions entres les balles dans un modèle de choc élastique… Les possibilités de cette méthode sont grandes !

13 Références

[1] : http://www.processing.org

[2] : http://www.arduino.org

[3] : « *Libérez votre cerveau ! Traité de neurosagesse pour changer l'école et la société* », Idriss Aberkane. Editions Robert Laffont, 2016.

[4]: https://www.**raspberrypi**.org et **raspberrypi**.fr

[5] : http://fr.openclassrooms.com/ > Arduino pour bien commencer en électronique et en programmation.

[6] : « Démarrer avec Arduino, principe de base et premiers montages », Massimo Banzi, ETSF.

[7] : « Arduino, Maîtrisez sa programmation et ses cartes d'interface (shield) », Christian Tavernier, DUNOD.

[8] : http://www.tonerkebab.fr/wiki/doku.php

[9] : http://makingthingssee.com/