Meetup Code-Algorithm-Processing-Dream-Paris

https://www.meetup.com/fr-FR/Code-Algorithm-Processing-Dream-Paris/

Atelier « migration »
de Processing à P5.js

le 16 janvier 2017

au Friends Vaugirard ☺

Table des matières

1. Introduction

1.1 Préambule

L'idée de cet atelier est partie d'une constatation personnelle, à savoir que la migration de sketch de Processing vers P5 peut être tantôt assez facile, tantôt très difficile, selon la complexité du sketch Processing à convertir.

Pour rappel, Processing s'appuie sur le langage Java, tandis que P5.js s'appuie sur le langage Javascript. Ces 2 langages présentent quelques similitudes en termes de syntaxe, mais aussi de grosses différences, notamment au niveau du typage des données et du modèle objet. Ces différences peuvent être à l'origine d'incompréhensions, et de découragement.

Au travers de divers scénarios de migration, ce dossier propose une assistance aux développeurs qui en éprouvent le besoins.

1.2 Ressources sur le portage de Processing vers P5.js

Quelques documentations en ligne intéressantes pour faciliter le processus de migration :

http://gerard.paresys.free.fr/Methodes/Methode-Processing-p5.html

https://github.com/processing/p5.js/wiki/Processing-transition

Un convertisseur en ligne :

http://faculty.purchase.edu/joseph.mckay/p5jsconverter.html

Ce convertisseur dégrossit un peu le travail de migration, mais la conversion est loin d'être parfaite et il y a souvent un gros travail de retouche à faire derrière.

1.3 Un point sur Processing.js

Processing.js est un projet parallèle à P5.js, on peut même considérer qu'il est un concurrent direct de P5.js.

Processing.js a été initié par John Resig, un développeur chevronné qui est aussi l'initiateur d'un autre projet bien connu : jQuery. John Resig travaille actuellement pour la Khan Academy, et Processing.js est utilisé par ce site comme support de certains cours d'initiation à la programmation.

Processing.js propose deux modes de fonctionnement :

- Conversion à la volée d'un sketch Processing, réalisée au sein même de la page HTML où le sketch s'exécute
- Préconversion d'un sketch Processing en sketch Javascript (compatible uniquement avec Processing.js et pas avec P5.js)

La conversion à la volée est très pratique, mais pour des raisons de performance on pourra préférer le second mode.

L'insertion d'un sketch Processing (non converti en JS) dans une page web se fait de la façon suivante :

```
<script src="
https://cdnjs.cloudflare.com/ajax/libs/processing.js/1.6.3/processing.min.js"></script>
<canvas data-processing-sources="mon_sketch.pde"></canvas>
```

L'outil de préconversion en Javascript est accessible ici :

http://processingjs.org/tools/processing-helper.html

Voici un sketch Processing fourni avec l'IDE de Processing, comme exemple de sketch :

```
// Brightness, by Rusty Robison.
int barWidth = 20;
int lastBar = -1;
void setup() {
  size(640, 360);
  colorMode(HSB, width, 100, width);
  noStroke();
  background(0);
}
void draw() {
  int whichBar = mouseX / barWidth;
  if (whichBar != lastBar) {
    int barX = whichBar * barWidth;
    fill(barX, 100, mouseY);
    rect(barX, 0, barWidth, height);
    lastBar = whichBar;
  }
}
```

Voici le sketch de la page précédente converti en Javascript par Processing.js :

```
// this code was autogenerated from PJS
(function($p) {

    var barWidth = 20;
    var lastBar = -1;

    function setup() {
        $p.size(640, 360);
        $p.colorMode($p.HSB, $p.width, 100, $p.width);
        $p.noStroke();
        $p.background(0);
    }
    $p.setup = setup;
    setup = setup.bind($p);

    function draw() {
        var whichBar = $p.mouseX / barWidth;
        if (whichBar != lastBar) {
            var barX = whichBar * barWidth;
            $p.fill(barX, 100, $p.mouseY);
            $p.rect(barX, 0, barWidth, $p.height);
            lastBar = whichBar;
        }
    }
    $p.draw = draw;
    draw = draw.bind($p);
})
```

On constate que Processing.js effectue un excellent travail de conversion. L'ensemble des fonctions propres à Processing sont préfixées par le symbole $p (le dollar est peut être un clin d'œil de John Resig par rapport au projet jQuery).

Sur la page suivante, on retrouvera le même sketch, mais adapté cette fois à P5.js.

1.4 Installation de P5.js

P5.js est un projet porté par la Fondation Processing. Soutenu par Casey Reas et Ben Fry, les chefs de file du projet Processing originel, son développement est assuré par Lauren Mc Carthy et un collectif de développeurs. Daniel Shiffman anime sur youtube de nombreux cours vidéos autour de Processing, et de P5.js, dont il est un ardent supporter.

P5.js est écrit en Javascript, et est conçu dans l'optique d'exécuter des sketchs écrits également en Javascript.

Pour pouvoir exécuter avec P5.js un sketch écrit initialement pour Processing, il convient de procéder à un certain nombre de modifications que nous allons étudier dans les chapitres suivants.

Le téléchargement de P5.js sur le site officiel se fait très facilement, sous la forme d'un fichier compressé contenant le code source JS du projet, et quelques libraires de code complémentaires.

L'installation de P5.js dans une page HTML se fait très simplement :

```
<script type="text/javascript" src="lib/p5.js"></script>
```

L'insertion d'un sketch P5.js dans une page web se fait de la façon suivante :

```
<script type="text/javascript" src="monsketch.js"></script>
```

On peut bien évidemment insérer son sketch directement à l'intérieur d'une page, en le plaçant entre les balises <script> et </script>.

2. Cinq exemples de conversion de Processing vers P5.js

2.1 Brighness

Voici le même sketch « Brigthness » adapté manuellement en Javascript, pour fonctionner directement avec P5.js. On trouve à droite le sketch Processing d'origine, et à gauche le sketch modifié pour fonctionner avec P5.js :

Version P5	Version Processing (d'origine)

`// Brightness, by Rusty Robison.`
`var barWidth = 20;`
`var lastBar = -1;`
 	
`function setup() {`
`  createCanvas(640, 360);`
`  colorMode(HSB, width, 100, width);`
`  noStroke();`
`  background(0);`
`}`
 	
`function draw() {`
`  var whichBar = mouseX / barWidth;`
`  if (whichBar != lastBar) {`
`  var barX = whichBar * barWidth;`
`  fill(barX, 100, mouseY);`
`  rect(barX, 0, barWidth, height);`
`  lastBar = whichBar;`
`  }`
`}`
	`// Brightness, by Rusty Robison.`
	`int barWidth = 20;`
	`int lastBar = -1;`
	
	`void setup() {`
	`  size(640, 360);`
	`  colorMode(HSB, width, 100, width);`
	`  noStroke();`
	`  background(0);`
	`}`
	
	`void draw() {`
	`  int whichBar = mouseX / barWidth;`
	`  if (whichBar != lastBar) {`
	`  int barX = whichBar * barWidth;`
	`  fill(barX, 100, mouseY);`
	`  rect(barX, 0, barWidth, height);`
	`  lastBar = whichBar;`
	`  }`
	`}`

Dans un cas comme celui-ci, on voit que les adaptations (indiquées en rouge) sont minimes, pour obtenir un code immédiatement opérationnel avec P5.js.

Les appels de fonction Java (mot clé « void ») doivent être remplacés par le mot clé « function » en Javascript.

Les variables, qui sont fortement typées en Java, doivent être remplacées par des variables (via le mot clé « var ») dont le typage est dynamique en Javascript (on parle de « typage faible », ou encore de langage « faiblement typé », caractéristique que Javascript partage avec PHP notamment).

Nous allons poursuivre avec d'autres exemples de conversion vers P5.js.

2.2 Les nuages de Paul

Nous allons effectuer ce second exemple de conversion en utilisant un exemple de code proposé par Paul Orlov, dans son livre « Programming for Artists ».

NB : Ce livre en russe est librement téléchargeable sur le blog de Paul Orlov.

Code P5	Code Processing d'origine

`var a = new Array(500) ;`
 	
`function setup() {`
`  createCanvas(700, 500);`
`  var i, j ;`
`  for (i = 0; i < 500; i++){`
`  a[i] = new Array(2);`
`  for (j = 0; j < 2; j++){`
`  a[i][j] = random(10,490);`
`  }`
`  }`
`}`
 	
`function draw() {`
`  smooth();`
`  noStroke();`
`  background(0);`
`  var i, eDist, eSize, eColor, cx, cy;`
`  for (i = 0; i < a.length; i++){`
`  eDist = dist(mouseX , mouseY , a[i][0],`
`  a[i][1]);`
`  eSize = map(eDist , 0, 200, 5, 100);`
`  eColor = map(eDist , 0, 200, 50, 255);`
`  fill(eColor , 200);`
 	
`  cx = noise(mouseX)*10 + a[i][0];`
`  cy = noise(mouseY)*10 + a[i][1];`
 	
`  ellipse(cx, cy , eSize , eSize);`
`  }`
`}`
	`float [][] a = new float[500][2];`
	
	`void setup() {`
	`  size(700, 500);`
	`  for(int i = 0; i < a.length; i++){`
	`  for(int j = 0; j < a[i].length; j++){`
	`  a[i][j] = random(10,490); 9`
	`  }`
	`  }`
	`}`
	
	`void draw() {`
	`  smooth();`
	`  noStroke();`
	`  background(0);`
	`  for(int i = 0; i < a.length; i++){`
	`  float eDist = dist(mouseX, mouseY,`
	`  a[i][0], a[i][1]);`
	`  float eSize = map(eDist, 0, 200, 5, 100);`
	`  float eColor = map(eDist, 0, 200, 50,`
	`  255);`
	`  fill(eColor, 200);`
	
	`  float cx = noise(mouseX)*10 + a[i][0];`
	`  float cy = noise(mouseY)*10 + a[i][1];`
	
	`  ellipse(cx, cy, eSize, eSize);`
	`  }`
	`}`

Dans cet exemple, la difficulté se situe essentiellement autour du tableau « a » qui est un tableau à 2 dimensions. La syntaxe de création d'un tableau en Java est différente de celle d'un tableau en Javascript. Hormis, cette difficulté, il y a beaucoup de similitudes dans la manipulation des tableaux, dans les 2 langages.

On constate que la déclaration des variables de type « integer » et « float » est effectuée en Javascript par le même mot clé « var ». Une variable pourrait donc changer de type « en cours de route » si l'on n'y prend pas garde.

Si en Java, une variable de type « integer » reçoit le résultat d'un calcul et que ce calcul est de type « nombre à virgule flottante », alors le résultat du calcul sera

automatiquement adapté au type de la variable réceptrice (integer). Javascript n'aura pas ce type de comportement et la variable réceptrice deviendra un « nombre à virgule flottante ». Cela peut avoir des conséquences sur la précision des calculs et donc des résultats ou effets obtenus. On pourra contourner ce problème en Javascript en utilisant la fonction Javascript parseInt() qui aura pour effet de renvoyer un « integer » quoi qu'il arrive.

Lors de la conversion de code Java vers Javascript, il convient d'être prudent concernant le « scope » (ou « portée ») des variables. Par exemple, la variable « i » dans la boucle Javascript « for » ci-dessous n'est pas circonscrite à la boucle « for » :

```
for (var i = 0; i < a.length; i++){
   // traitement spécifique
}
```

En effet, en Javascript, la présence du mot clé « var » devant la variable peut laisser croire que la portée de la variable « i » est circonscrite à la boucle, or il n'est en rien. Dans l'exemple de la page précédente, la variable « i » de la fonction « setup » est visible à tous les niveaux de la fonction, idem pour la fonction « draw ». C'est la raison pour laquelle, pour éviter toute confusion, j'ai choisi lors de la conversion de déclarer la variable en amont de la boucle.

Il est intéressant de noter que l'on peut optimiser la boucle prédédente de la façon suivante :

```
for (var i = 0, imax = a.length ; i < imax ; i++){
   // traitement spécifique
}
```

Dans l'exemple ci-dessus, la variable « imax » est initialisée en même temps que la variable « i », avant le première point virgule. De cette manière, le comptage du nombre d'éléments du tableau « a » n'est pas réévalué à chaque itération.

Un point important à noter également : la création de variable en Javascript se fait via le mot clé « var ». Si on oublie d'utiliser ce mot clé lors de la déclaration d'une variable, cette variable se trouvera automatiquement rattachée à l'objet « window » qui est un objet « piloté » par le navigateur et qui contient beaucoup de propriétés et méthodes dont le navigateur a besoin pour fonctionner. On peut obtenir une vue détaillée de cet objet via un console.log(window). On peut contrer ce comportement permissif de Javascript en utilisant au début de son sketch l'instruction suivante :

```
"use strict";
```

Cette instruction fonctionne comme un « drapeau » avertissant l'interpréteur Javascript qu'on ne l'autorise plus à utiliser une variable non déclarée. Cela peut être un excellent moyen de sécuriser votre travail, lors de l'adaptation d'un sketch de Processing vers P5.js. Et je vous encourage à l'utiliser dans un contexte plus large, notamment pour du

développement web professionnel. On notera qu'il est possible d'utiliser cette instruction « use strict » à l'intérieur de fonctions Javascript, de façon à limiter la « portée » de l'instruction. Cela peut être utile si l'on est obligé de composer avec des librairies Javascript développées par des tiers.

2.3 Cent balles bondissantes

A partir d'un sketch Processing emprunté à un site japonais, sketch dont le principe consiste à afficher 100 balles rebondissant sur les 4 côtés de l'écran, nous allons aborder le problème de la conversion de classes Java en classes Javascript.

Le lien vers le script d'origine est le suivant.

https://github.com/p5aholic/p5codeschool/blob/master/samples/Chapter15_3/sketch07/sketch07.pde

Le code source du sketch Processing étant plus volumineux que les précédents, nous présenterons d'abord le sketch d'origine, puis dans un second temps le sketch converti pour P5.js.

Code source initial en Processing :

```
Ball[] balls = new Ball[100];

void setup() {
  size(750, 350);

  // Générer 100 objets de classe Ball
  for (int i = 0; i < balls.length; i++) {
    balls[i] = new Ball();
  }
}

void draw() {
  background(255);

  // Exécuter les méthodes de mise à jour et d'affichage pour tous les objets Ball
  for (int i = 0; i < balls.length; i++) {
    balls[i].update();
    balls[i].display();
  }
}
```

```
class Ball {
  float x, y;
  float vx, vy;
  int radius;
  color c;
  // constructeur
  Ball() {
    this.radius = (int)random(10, 20);
    this.x = random(this.radius, width-this.radius);
    this.y = random(this.radius, height-this.radius);
    this.vx = random(-5, 5);
    this.vy = random(-5, 5);
    this.c = color(random(255), random(255), random(255), random(255));
  }
  // méthode de mise à jour
  void update() {
    this.x += this.vx;
    this.y += this.vy;
    if (this.x-this.radius <= 0 || this.x+this.radius >= width) {
      this.vx *= -1;
    }
    if (this.y-radius <= 0 || this.y+this.radius >= height) {
      this.vy *= -1;
    }
  }
  // méthode d'affichage
  void display() {
    noStroke();
    fill(c);
    ellipse(x, y, 2*radius, 2*radius);
  }
}
```

La conversion des fonctions setup() et draw() ne présente pas de difficulté, mais la conversion de la classe Ball de Java vers Javascript peut se faire de différentes manières :

- Soit par une réécriture en Javascript selon la norme ECMAScript 5 (souvent abrégée en ES5)
- Soit par une réécriture en Javascript selon la norme ECMAScript 6 (souvent abrégée en ES6, ou quelquefois ES2015, en référence à son année de publication).

La norme ES6 introduit une nouvelle manière d'écrire des classes en Javascript, avec une syntaxe plus proche de la syntaxe Java. Nous allons dans ce chapitre étudier cette approche, car elle est plus facile à aborder pour un développeur ne maîtrisant pas toutes les subtilités des objets en Javascript. Nous verrons dans le dernier chapitre deux manière alternatives d'écrire des classes Java (qui fonctionnent aussi bien avec ES5 que ES6).

Pour convertir la classe Ball de Java vers Javascript, il nous faut retravailler un peu le code cette classe, pour qu'elle soit conforme aux principes de la norme ES6 :

```
class Ball {
  constructor() {
    this.radius = random(10, 20);
    this.x = random(this.radius, width-this.radius);
    this.y = random(this.radius, height-this.radius);
    this.vx = random(-5, 5);
    this.vy = random(-5, 5);
    this.c = color(random(255), random(255), random(255), random(255));
  }
  update() {
    this.x += this.vx;
    this.y += this.vy;
    if (this.x-this.radius <= 0 || this.x+this.radius >= width) {
      this.vx *= -1;
    }
    if (this.y-radius <= 0 || this.y+this.radius >= height) {
      this.vy *= -1;
    }
  }
  display() {
```

```
    noStroke();
    fill(c);
    ellipse(x, y, 2*radius, 2*radius);
  }
}
```

On peut intégrer le code de la page précédente à notre sketch, cela fonctionnera avec les navigateurs les plus récents, sous réserve qu'ils soient à jour (c'est le cas de Chrome, Firefox, Safari, etc..). Mais les navigateurs plus anciens ne supporte pas ES6, aussi si l'on souhaite bénéficier d'une portabilité maximale, on aura intérêt à utiliser un outil comme BabelJS pour convertir notre code ES6 en ES5 :

https://babeljs.io/repl/

Lorsque vous effectuerez la conversion de votre code avec BabelJS, vous constaterez que BabelJS ajoute quelques fonctions spécifiques, qui lui permettent d'effectuer plus simplement le portage du code vers ES5. Ces fonctions doivent être insérées avec la classe dans votre sketch pour que ce dernier fonctionne. Voici en particulier le code qui sera ajouté pour notre sketch en cours d'étude (le code de la classe Ball se trouve page suivante) :

```
"use strict";

var _createClass = function () { function defineProperties(target, props) { for (var
i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumerable =
descriptor.enumerable || false; descriptor.configurable = true; if ("value" in
descriptor) descriptor.writable = true; Object.defineProperty(target, descriptor.key,
descriptor); } } return function (Constructor, protoProps, staticProps) { if
(protoProps) defineProperties(Constructor.prototype, protoProps); if (staticProps)
defineProperties(Constructor, staticProps); return Constructor; }; }();

function _classCallCheck(instance, Constructor) { if (!(instance instanceof
Constructor)) { throw new TypeError("Cannot call a class as a function"); } }
```

Voici le code de la classe Ball converti en ES5 par BabelJS :

```
var Ball = function () {
  // constructeur
  function Ball() {
    _classCallCheck(this, Ball);
    this.radius = random(10, 20);
    this.x = random(this.radius, width - this.radius);
    this.y = random(this.radius, height - this.radius);
    this.vx = random(-5, 5);
    this.vy = random(-5, 5);
    this.c = color(random(255), random(255), random(255), random(255));
  }
  _createClass(Ball, [{
    key: "update",
    value: function update() {
      this.x += this.vx;
      this.y += this.vy;
      if (this.x - this.radius <= 0 || this.x + this.radius >= width) {
        this.vx *= -1;
      }
      if (this.y - radius <= 0 || this.y + this.radius >= height) {
        this.vy *= -1;
      }
    }
  }, {
    key: "display",
    value: function display() {
      noStroke();
      fill(c);
      ellipse(x, y, 2 * radius, 2 * radius);
    }
  }]);
  return Ball;
}();
```

2.4 Sauts dans l'hyper-espace

Le très beau sketch "Moving through space", de Konrad Junger (découvert sur le site openprocessing.org), a servi de test pour ce quatrième exemple de conversion.

Pour l'occasion, je vous propose deux versions :

- Une version dans laquelle la classe « Star » a été écrite en « full » ES6, et laissée telle quelle, sans conversion via BabelJS.
- Une version équivalente pour laquelle la classe Star a été convertie en ES5 avec BabelJS

La première version constitue un excellent moyen de tester la compatibilité de votre navigateur avec ES6. Si vous utilisez un navigateur ancien, elle a toutes les chances de « planter ». En revanche la version ES5 devrait fonctionner sur la grande majorité des navigateurs (sauf ceux – vraiment trop anciens - qui ne supportent pas l'API Canvas de la norme HTML5).

Le code source de la classe « star » - après conversion en ES6 - est le suivant :

```
class star {
  constructor() {
    this.x = random(width);
    this.y = random(height);
    this.speed = random(0.2, 5);
    this.wachsen = parseInt(random(0, 2));
    if (this.wachsen == 1) {
      this.d = 0;
    } else {
      this.d = random(0.2, 3);
    }
    this.age = 0;
    this.sizeIncr = random(0,0.03);
  }
  render() {
    this.age++;
    if (this.age < 200){
      if (this.wachsen == 1){
        this.d += this.sizeIncr;
        if (this.d > 3 || this.d < -3) {
```

```
            this.d = 3;
        }
      } else {
        if (this.d > 3 || this.d < -3) {
           this.d = 3;
        }
        this.d = this.d + 0.2 - 0.6 * noise(this.x, this.y, frameCount);
      }
    } else {
      if (this.d > 3 || this.d <- 3) {
          this.d = 3;
      }
    }
    ellipse(this.x, this.y,
        this.d*(map(noise(this.x, this.y, 0.001 * frameCount),0,1,0.2,1.5)),
        this.d*(map(noise(this.x, this.y, 0.001 * frameCount),0,1,0.2,1.5))
    );
  }
  move() {
    this.x = this.x - map(mouseX, 0, width, -0.05 * this.speed, 0.05 * this.speed) *
              (w2 - this.x);
    this.y = this.y - map(mouseY, 0, height, -0.05 * this.speed, 0.05 * this.speed) *
              (h2 - this.y);
  }
}
```

Quelques explications : par rapport à la classe « star » d'origine, j'ai effectué les modifications suivantes :

- J'ai préfixé toutes les propriétés de la classe par « this »
- J'ai supprimé le mot clé « void » devant les méhodes « render » et « move » (on notera que le mot clé « function » ne doit pas être utilisé dans la déclaration des classes ES6)
- J'ai renommé la méthode « star » en « constructor » pour être conforme avec la syntaxe ES6

2.5 Attrapeur de rêves

Je vous propose de finir cet exercice de conversion avec le très beau sketch d'Oggy qui s'intitule « Dream Catcher » (également découvert sur le site openprocessing.org).

Nous avons vu que nous pouvons convertir des classes Java en Javascript, avec l'utilisation de la nouvelle syntaxe des classes propre à la norme ES6 (avec et sans le soutien de BabelJS). Nous n'avons en revanche pas vu comment réaliser la même chose en « pur ES5 ». Je vais vous montrer deux versions différentes du même sketch, avec deux manières différentes de créer des objets sous ES5. Ces deux manières fonctionnent toutes deux très bien, dès lors que l'on n'a pas besoin d'utiliser de notion d'héritage. Avec la notion d'héritage, cela devient plus complexe sous ES5, et cela nécessite une maîtrise plus avancée de Javascript. Si vous avez besoin de la notion d'héritage, je vous encourage dans ce cas à utiliser ES6 (avec ou sans BabelJS).

Le sketch d'Oggy utilise deux objets différents, MyColor et SeeWead. Dans la première version du sketch pour P5.js, j'ai créé les deux objets en utilisant la même technique que nous allons étudier maintenant. L'objet MyColor étant le plus simple des deux (en termes de nombre de lignes), je vais me focaliser sur lui dans un premier temps. Je vous explique comment construire cet objet en « pas à pas », avant de vous fournir le code source complet.

Les objets en Javascript ont quelques caractéristiques particulières, que l'on ne retrouve pas dans les autres langages (comme Java et PHP par exemple).

Première caractéristique essentielle : toute fonction en Javascript est par définition un objet. Cet objet peut être utilisé comme une simple fonction (au sens où on l'entend dans des langages comme PHP par exemple), ou être utilisé comme « constructeur » d'un objet.

Si l'on souhaite utiliser une fonction comme constructeur d'un objet on écrira quelque chose du genre :

```
var monObjet = new MyColor() ;
```

Pour que cela fonctionne, il faut que les méthodes et propriétés publiques de cet objet soient préfixées par « this » dans le code source de la fonction MyColor().

Nous commençons par créer une fonction MyColor qui fait appel à une méthode interne « init » (on sait qu'il s'agit d'une méthode du fait de la présence du préfixe « this ») :

```
var MyColor = function () {
    this.init();
}
```

Nous définissons ensuite le code de la méthode « init » en l'associant à l'objet « prototype » de la fonction MyColor :

```
MyColor.prototype.init = function () {
    this.minSpeed = .6;
    this.maxSpeed = 1.8;
    // etc.. je ne détaille pas toute la méthode ici, sachant que vous
    // trouverez le code source complet plus loin dans ce chapitre
};
```

C'est quoi cet objet « Prototype » ?

Quelle que soit son origine (fonction ou autre), tout objet Javascript a pour objet parent un objet « prototype », qu'il est possible de modifier en lui ajoutant des méthodes et propriétés spécifiques. Pour faire « court », disons que ces méthodes et propriétés « redescendront » par héritage sur tous les objets instanciés à partir de la fonction considérée (en l'occurrence la fonction MyColor).

Le mieux, pour bien comprendre le principe, c'est de le tester sur un cas concret, aussi voici le code source complet de l'objet MyColor :

```
var MyColor = function () {
    this.init();
}
MyColor.prototype.init = function () {
    this.minSpeed = .6;
    this.maxSpeed = 1.8;
    this.minR = 200;
    this.maxR = 255;
    this.minG = 20;
    this.maxG = 120;
    this.minB = 100;
    this.maxB = 140;
    this.R = random(this.minR, this.maxR);
    this.G = random(this.minG, this.maxG);
    this.B = random(this.minB, this.maxB);
    this.Rspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
    this.Gspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
    this.Bspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
};
```

```
MyColor.prototype.update = function () {
    this.Rspeed = (((this.R += this.Rspeed) > this.maxR) || (this.R < this.minR)) ?
                  -this.Rspeed : this.Rspeed;
    this.Gspeed = (((this.G += this.Gspeed) > this.maxG) || (this.G < this.minG)) ?
                  -this.Gspeed : this.Gspeed;
    this.Bspeed = (((this.B += this.Bspeed) > this.maxB) || (this.B < this.minB)) ?
                  -this.Bspeed : this.Bspeed;
};
MyColor.prototype.getColor = function () {
    return color(this.R, this.G, this.B);
};
```

Au début de ce chapitre, j'indiquais qu'il était possible de créer nos objets MyColor et SeeWead selon une autre méthode. La voici :

```
var MyColor_Proto = {
    minSpeed : .6,
    maxSpeed : 1.8,
    minR : 200,
    maxR : 255,
    minG : 20,
    maxG : 12,
    minB : 100,
    maxB : 140
};
var MyColor = function () {
    var that = Object.create(MyColor_Proto);
    that.R = random(that.minR, that.maxR);
    that.G = random(that.minG, that.maxG);
    that.B = random(that.minB, that.maxB);
    that.Rspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
    that.Gspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
    that.Bspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
```

```
    that.update = function () {
        that.Rspeed = (((that.R += that.Rspeed) > that.maxR) ||
                (that.R < that.minR)) ? -that.Rspeed : that.Rspeed;
        that.Gspeed = (((that.G += that.Gspeed) > that.maxG) ||
                (that.G < that.minG)) ? -that.Gspeed : that.Gspeed;
        that.Bspeed = (((that.B += that.Bspeed) > that.maxB) ||
                (that.B < that.minB)) ? -that.Bspeed : that.Bspeed;
    };
    that.getColor = function () {
        return color(that.R, that.G, that.B);
    };
    return that ;
}
```

J'ai mis en rouge les éléments qui me semblaient essentiels. On commence par créer un objet que j'ai appelé « that » (par convention, et pour éviter tout risque de confusion avec « this »). Pour créer cet objet « that » j'ai utilisé la méthode « create » de l'objet « Object ». C'est un objet standard de Javascript, qui est en quelque sorte le « papa » de tous les autres objets.

La présence du « return that » à la fin de la fonction est essentielle, sans cela nous ne serons pas en mesure de récupérer l'objet généré par la fonction, et nous ne pourrons pas utiliser ses propriétés et méthodes. Je vous invite à mettre cette ligne en commentaire, et à regarder ce qui se passe.

Avant de conclure, je dois vous signaler quelques différences que j'ai constatées entre Processing et P5.js, au niveau de la manipulation des objets de type « vecteur ».

La création d'objets de type "vecteur" avec Processing se fait de la façon suivante :

```
PVector mouse = new PVector(mouseX, mouseY);
```

Avec P5.js, la création d'objets de type vecteur peut se faire de 2 manières qui sont équivalentes :

```
var mouse = new p5.Vector(mouseX, mouseY);
```

ou encore :

```
var mouse = createVector(mouseX, mouseY);
```

Il y a quelques différences entre Processing et P5.js, concernant le fonctionnement des objets de type vecteur :

- La méthode « get() » sous Processing devient « copy() » sous P5.js
  - `Avec Processing : PVector vect2 = vect1.get();`
  - `Avec P5.js :      var vect2 = vect1.copy();`

- La méthode « dist() » fonctionne différemment (cf. exemple ci-dessous).

Avec Processing, pour déterminer la distance entre 2 vecteurs, on utilise la méthode dist() de la façon suivante :

```
float d = PVector.dist(vect1, vect2);
```

Sous P5.js on peut écrire la même chose de 2 manières différentes :

```
var d = p5.Vector.dist(vect1, vect2);  !!! Attention : ne pas utiliser « new » !!!
var d = vect1.dist(vect2);
```

Je n'ai pas résisté, avec ce dernier sketch, à l'envie d'ajouter quelques fonctionnalités, telles que :

- Un petit effet sonore, avec le projet Pizzicato.js
  - https://alemangui.github.io/pizzicato/
- La possibilité de modifier le nombre d'algues, via les touches 1 à 9 (à chaque fois on multiplie par 10 la valeur correspondant à la touche pressée, et on regénère les algues)
- La possibilité de stopper l'animation, via la touche X
- La possibilité de sauvegarder les images au fil de l'eau, via la touche S
- *La possibilité d'activer/désactiver le bruit (effet de perturbation) était déjà implémentée dans la version initiale du sketch*
- *La possibilité de modifier le rendu de l'image (points ou traits pleins) en pressant la plupart des autres touches, était également implémentée dans la version initiale du sketch*

Vous pouvez personnaliser à loisir les actions au clavier, en modifiant la fonction keyPressed(), dont vous trouverez le code source ci-dessous :

```
function keyPressed() {
    if (key > '0' && key <= '9') {
        noiseOn = true;
        nbWeeds = parseInt(key) * 10;
        weeds = new Array(nbWeeds);
        // réinitialisation du canvas
        setup();
        return;
    } else {
        if (key == 'x' || key == 'X') {
            noLoop();
            return;
        } else {
            if (key == 's' || key == 'S') {
                saveCanvas("DreamCatcher-" + frameCount + ".png", 'png');
                return;
            } else {
                if (key == 'n' || key == 'N') {
                    noiseOn = !noiseOn;
                    return;
                } else {
                    mode = (mode + 1) % 2;
                    return;
                }
            }
        }
    }
}
```

3. Références

Lien vers ce support et le code source des exemples sur Github :

https://github.com/gregja/p5Migration

Slide d'introduction à l'animation avec P5.js (avec le code source sur Github) :

http://backstages.gregphplab.com/circlecirclemeetup/

https://github.com/gregja/circleCircleMeetup

BabelJS, convertisseur JS de ES5 vers ES6 :

https://babeljs.io/repl/

Tableau comparatif du support de ES6 par plateformes et navigateurs

http://kangax.github.io/compat-table/es6/

Tutoriel pour la création de classes ES6

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes

Site officiel P5.js :

http://p5js.org/

Exemples de sketchs Processing intéressants à convertir (pour se faire la main) :

https://www.openprocessing.org/sketch/169537

https://github.com/PaulOrlov/processing-1

http://ptahi.ru/2016/03/25/processing-sketch-based-on-histograms-of-iterated-chaotic-functions/

http://sketchpad.cc/Yvmsqf9O8N

Javascript n'est pas multi-thread. Pour répondre à des besoins de parallélisation de traitement (par exemple pour des calculs un peu « lourds »), on peut se tourner vers l'API HTML5 WebWorkers :

https://developer.mozilla.org/fr/docs/Utilisation_des_web_workers

http://www.w3schools.com/html/html5_webworkers.asp