

5XSCL1b

SCRIPTS CLIENTS 25-26

Thibault Six

20 chapitres

SUPPORTS DE COURS

https://cours.tsix.be/cours/scripts-clients-25-26

Version 4 — Fiche UE 2026-2027 liée depuis la page d'accueil · 10/10/2026

PRÉSENTATION

SCRIPTS CLIENTS

Programme du cours

L'étudiant sera capable :

1. **Identification des éléments impliquant l'usage d'un script client** dans une page web.
2. **Analyse d'un script client** en termes de :
   - Définition des variables et des objets.
   - Structures conditionnelles et itératives.
   - Fonctions et de procédures.
   - Structures interactives (gestion des évènements,...).
   - ...
3. **Exploitation d'un script client** dans une page web.
4. **Modification et création d'un script** et intégration dans une page web.
5. **Description et caractérisation** d'objets, propriétés et méthodes.
6. **Détermination des événements** auxquels les éléments de la page doivent réagir.
7. **Traduction** des éléments nécessaires à la résolution d'un problème posé (structures procédurales, interactives, animations, objets...) sous forme de commentaires, de schémas, de dessins, etc.
8. **Mise en œuvre de la résolution** d'un problème posé en utilisant le langage client choisi.
9. **Exploitation du côté orienté objet** du langage choisi :
   - Les classes prédéfinies et leurs composants (window, document, cookie, ...).
   - La définition de classes et leur instanciation.
   - ...
10. **Utilisation des variables, des structures conditionnelles, des structures itératives, des tableaux, de l'affichage dans une page web**, etc. dans le langage choisi.
11. **Exploitation de la notion d'expression régulière** (validation de formulaires, ...).
12. **Exploitation de données structurées en XML** (Extensible Markup Langage), en JSON (JavaScript Object Notation), ... contenues dans un fichier externe.
13. **Description et mise en œuvre** des technologies utilisées dans le développement d'applications web dynamiques et animées telles que AJAX (Asynchronous Javascript and XML), ...
14. **Choix et exploitation** d'une bibliothèque tierce telle que jQuery, MooTools, ... pour le développement de scripts spécifiques pour RIA (interfaces riches), transmissions asynchrones, ...
15. **Identification et résolution** d'erreurs de programmation au moyen d'outils ou de techniques de débogage.
16. **Utilisation appropriée** de la documentation disponible.

Acquis d'apprentissage

Critères de réussite

Pour atteindre le seuil de réussite, l'étudiant sera capable, et au départ d'un cahier des charges contenant un projet de pages web interactives et animées :

1. **Création et exploitation** de scripts clients basés sur des classes prédéfinies.
2. **Création et exploitation** de ses propres classes.
3. **Création et exploitation** de scripts basés sur une bibliothèque tierce.

Critères de maîtrise

Pour la détermination du degré de maîtrise, il sera tenu compte des critères suivants :

1. **Design et Ergonomie du Site**
2. **Contenu**
3. **Qualité du Code**
4. **Fonctionnalités supplémentaires**

Évaluation

- Évaluation finale (projet) 100% de la note finale
- Fiche UE 2026-2027 (PDF) PDF
- Grille d'évaluation (PDF) PDF

Sommaire

- SCRIPTS CLIENTS

1 P5.js
- 1.1 Outils de dessin
- 1.2 Couleurs
- 1.3 Premières formes
- 1.4 Variables
- 1.5 Conditions
- 1.6 Boucles
- 1.7 Art Génératif
- 1.8 Intégration de p5.js avec Vite

2 JS
- 2.1 CLI
- 2.2 NodeJS et NPM
- 2.3 Exercices de révision
- 2.4 Vite
- 2.5 Objets et json
- 2.6 Méthodes des tableaux
- 2.7 Templating
- 2.8 Hébergement
- 2.9 Morpion
- 2.10 Snake

3 Examen

CHAPITRE 1

Introduction à p5.js

> 🎯 ***Objectif*** *: Comprendre ce qu'est p5.js, pourquoi on l'utilise dans ce cours, le mettre en place et découvrir l'interface. À la fin de cette introduction, tu seras capable de lancer ton premier programme.*



Réviser JavaScript en s'amusant

Dans ce cours, tu as déjà appris les bases de **JavaScript** : variables, conditions, boucles, fonctions... Mais écrire du code qui affiche du texte dans la console, avouons-le, c'est pas le truc le plus excitant du monde.

Et si on révisait tout ça en **dessinant** ? En créant des animations, des formes colorées, des programmes visuels ?

p5.js, c'est exactement ça : une **bibliothèque JavaScript** conçue pour rendre le code **visuel et créatif**. Tu écris du JavaScript, tu rafraîchis ta page, et tu vois directement le résultat à l'écran.

L'avantage ? Tu révises **toute la syntaxe JavaScript** que tu connais déjà (variables, `if`, `for`, fonctions...) mais avec des résultats immédiats et visuels. C'est le même langage, juste avec des super-pouvoirs graphiques !

Un peu d'histoire

p5.js est né du projet **Processing**, créé en **2001** au **MIT** (Massachusetts Institute of Technology) par **Ben Fry** et **Casey Reas**. Leur objectif : permettre aux **artistes**, designers et non-programmeurs de créer des œuvres visuelles avec du code, sans avoir besoin d'un diplôme en informatique.

Le projet a tellement bien marché qu'il a reçu le **Golden Nica** du prestigieux festival Ars Electronica en 2005.

En **2013**, **Lauren Lee McCarthy** a créé **p5.js**, une version de Processing qui tourne directement dans le **navigateur web** grâce à JavaScript. Aujourd'hui, p5.js est utilisé dans le monde entier pour l'art numérique, la visualisation de données, et surtout... **l'apprentissage de la programmation**.

Pourquoi p5.js dans un cours de Scripts Clients ?

Voici le secret : p5.js, c'est du **JavaScript pur**. Tout ce que tu as appris en JavaScript fonctionne directement dans p5.js.

JavaScript classique	p5.js
`let x = 100;`	`let x = 100;`
`if (x > 50) { }`	`if (x > 50) { }`
`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)`
`function maFonction() { }`	`function maFonction() { }`
`console.log("hello")`	`console.log("hello")`

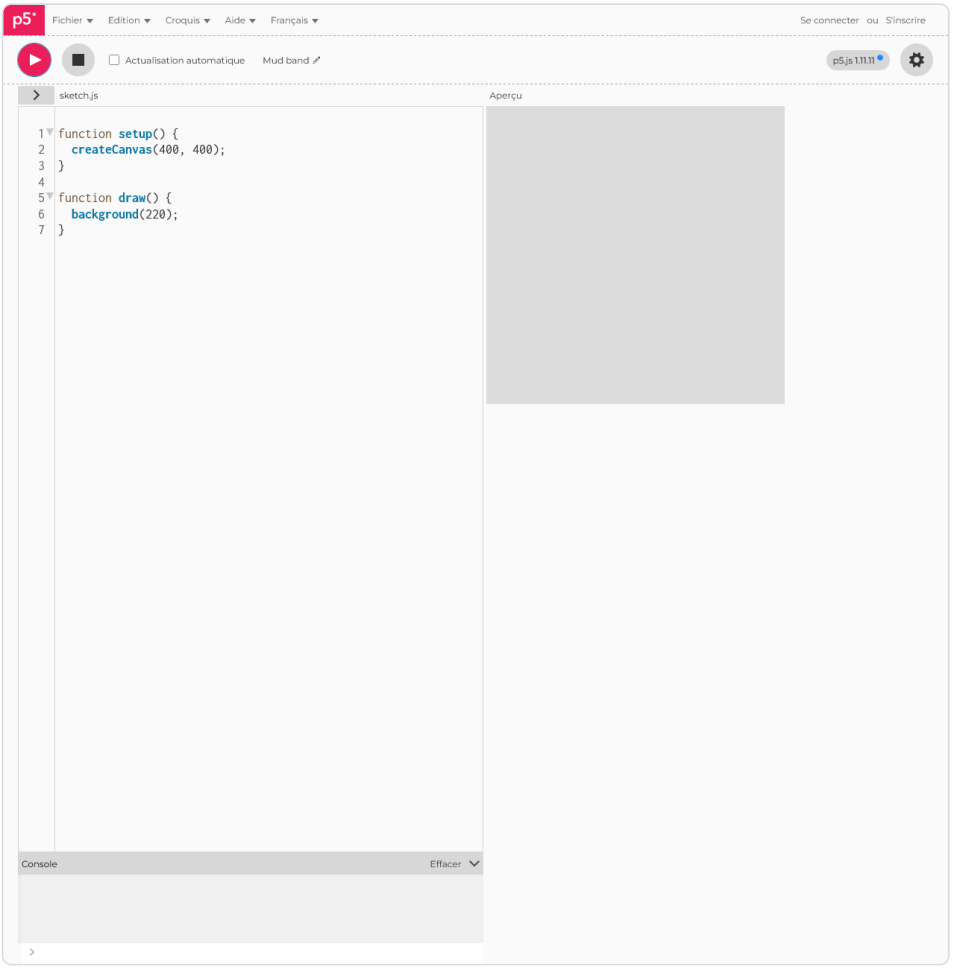
Tu vois ? C'est **exactement le même langage**. p5.js ajoute simplement des fonctions pour dessiner (`rect()`, `ellipse()`, `fill()`...) et une structure de base (`setup()` et `draw()`) pour organiser ton programme.

💡 **Fun fact** : p5.js peut interagir avec le DOM, gérer du son, de la vidéo, de la webcam, et même de la 3D — tout ça en JavaScript dans le navigateur !

Mettre en place p5.js

L'éditeur en ligne (le plus simple)

1. Va sur **https://editor.p5js.org**
2. Tu arrives directement sur un éditeur de code prêt à l'emploi
3. Clique sur ▶ **Play** pour lancer ton programme
4. C'est tout ! Pas besoin d'installer quoi que ce soit.



En local (dans ton projet)

Si tu préfères travailler en local, il suffit d'inclure la bibliothèque p5.js dans ton fichier HTML :

```xml
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdn.jsdelivr.net/npm/p5@1/lib/p5.min.js"></script>
</head>
<body>
  <script src="sketch.js"></script>
</body>
</html>
```

Puis tu crées un fichier `sketch.js` avec ton code p5.js. Ouvre le fichier HTML dans ton navigateur et c'est parti !

L'interface de l'éditeur en ligne

Quand tu ouvres l'éditeur p5.js, voici ce que tu vois :

Zone	Rôle
Zone de code (grande zone blanche)	C'est là que tu écris ton programme
Bouton ▶ Play (en haut à gauche)	Lance ton programme
Bouton ■ Stop	Arrête ton programme
Console (zone noire en bas)	Affiche les messages et les erreurs
Zone de prévisualisation	S'affiche à droite quand tu cliques sur Play — c'est ton canevas !

💡 **Astuce** : Si tu vois du texte rouge dans la console, c'est une **erreur**. Pas de panique ! Lis le message, il te dit souvent quelle ligne pose problème.

La structure d'un programme p5.js

Tout programme p5.js a la même structure de base :

```javascript
function setup() {
  // Code exécuté UNE SEULE FOIS au démarrage
}

function draw() {
  // Code exécuté EN BOUCLE (60 fois par seconde !)
}
```

Le parallèle avec JavaScript classique

Concept	JavaScript classique	p5.js	Rôle
Initialisation	`window.onload = ...`	`function setup() { }`	S'exécute une seule fois au début
Répétition	`setInterval(...)`	`function draw() { }`	S'exécute en boucle, encore et encore
Zone d'affichage	Le DOM / un `<canvas>`	Le **canevas** (canvas)	La zone où on dessine

Pour créer un canevas, on utilise `createCanvas()` dans le `setup()` :

```javascript
function setup() {
  createCanvas(400, 300);  // Crée un canevas de 400 pixels de large et 300 de haut
}

function draw() {
  // On dessinera ici
}
```

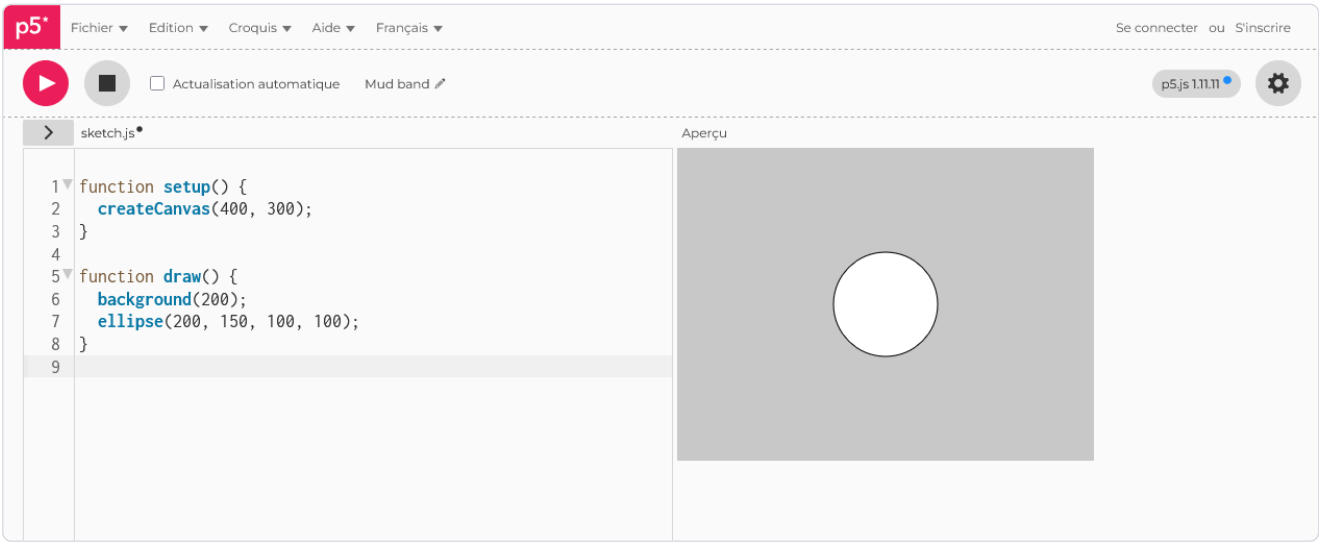
⚠️ **Attention** : `createCanvas()` doit toujours être dans `setup()`, idéalement comme **première instruction**.

Ton premier programme

Tape ce code dans l'éditeur p5.js et clique sur ▶ **Play** :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(200);
  ellipse(200, 150, 100, 100);
}
```



Tu devrais voir une fenêtre grise avec un cercle blanc au centre. Félicitations, tu viens d'écrire ton premier programme p5.js ! 🎉

Ne t'inquiète pas si tu ne comprends pas encore chaque ligne — on va tout décortiquer dans les prochains chapitres.

Ce qu'on va apprendre dans les prochains chapitres

Voici un aperçu de la suite. Chaque concept utilise du JavaScript que tu connais déjà :

Chapitre	Concept	En JavaScript classique	En p5.js
0b	**Outils de dessin**	Manipuler un `<canvas>`	`rect()`, `ellipse()`, `stroke()`, `fill()`
0c	**Couleurs**	`ctx.fillStyle = "red"`	`fill(255, 0, 0)`, `background()`
1	**Premières formes**	Canvas API complexe	Combiner formes et couleurs
2	**Variables**	`let x = 10;`	`let x = 10;` (identique !)
3	**Conditions**	`if (x > 5) { } else { }`	`if (x > 5) { } else { }` (identique !)
4	**Boucles**	`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)` (identique !)
5	**Fonctions**	`function dessiner() { }`	`function dessinerMaison(x, y) { }`
6	**Animation**	`requestAnimationFrame()`	Variable qui change dans `draw()`

Les types de données (un avant-goût)

En JavaScript, les variables sont déclarées avec `let` ou `const`. Contrairement à des langages comme Java ou C++, tu ne dois pas déclarer le type — JavaScript le détermine automatiquement. Mais il est important de connaître les types qui existent :

Type	Ce que ça stocke	Exemple
`number`	Un nombre (entier ou à virgule)	`let age = 25;` / `let prix = 9.99;`
`boolean`	**Vrai** ou **Faux**	`let estVisible = true;`
`string`	Du **texte**	`let nom = "Alice";`

💡 En JavaScript, pas besoin de préciser `int` ou `float` comme en Java/C++ — un `number` gère les deux :

```javascript
let x = 100;          // ✅ Nombre entier
let y = 3.14;         // ✅ Nombre à virgule
let nom = "Bonjour";  // ✅ Texte
let ok = true;        // ✅ Booléen
```

p5.js ajoute aussi la fonction `color()` pour manipuler les couleurs facilement :

```javascript
let rouge = color(255, 0, 0);
```

On verra tout ça en détail dans le chapitre sur les variables.

Pour aller plus loin

Si tu veux explorer p5.js par toi-même :

- **Documentation officielle** : https://p5js.org/reference — la liste complète de toutes les fonctions
- **The Coding Train** (YouTube) : Daniel Shiffman explique p5.js de façon fun et accessible
- **Nature of Code** (https://natureofcode.com) : un livre gratuit pour les plus avancés, mêlant code et simulation physique
- **Éditeur en ligne** : https://editor.p5js.org — code et teste directement dans le navigateur

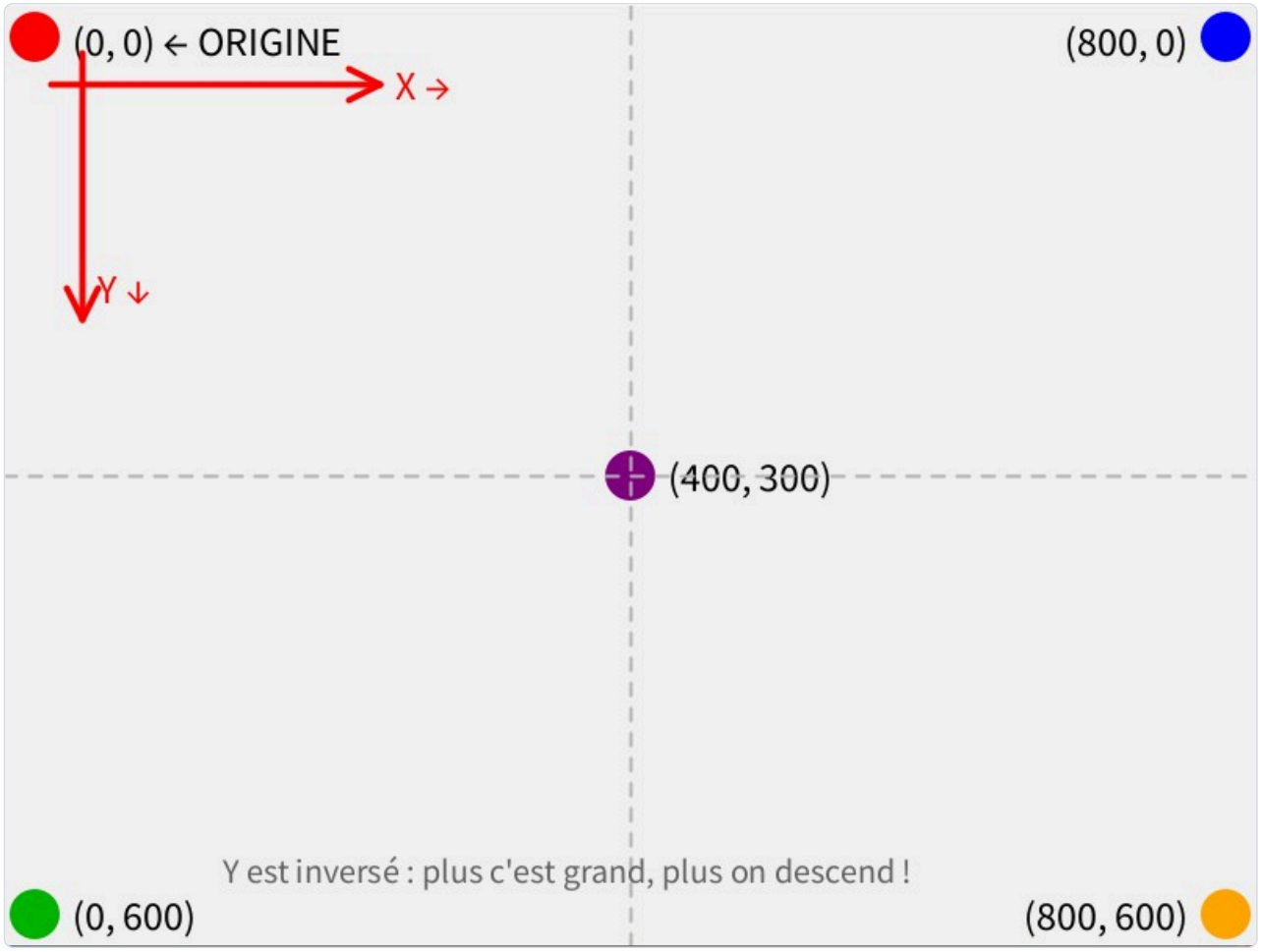
1.1 Les Outils de Dessin

> 🎯 ***Objectif*** *: Maîtriser le système de coordonnées, dessiner toutes les formes de base, et comprendre le fonctionnement du remplissage et du contour. À la fin de ce chapitre, tu seras capable de dessiner n'importe quelle combinaison de formes sur ton canevas.*

Le système de coordonnées

L'origine est en haut à gauche

C'est différent des maths ! En p5.js, le point **(0, 0)** est en **haut à gauche** du canevas. L'axe X va vers la **droite** et l'axe Y va vers le **bas**.



Quand on écrit `ellipse(100, 200, 50, 50)`, on dessine un cercle dont le centre est à **100 pixels depuis la gauche** et **200 pixels depuis le haut**.

⚠️ **Piège classique** : l'axe Y est **inversé** par rapport aux cours de maths. Plus la valeur de Y est grande, plus on **descend** !

Direction	Valeur qui augmente
→ Vers la droite	X augmente
↓ Vers le bas	Y augmente
← Vers la gauche	X diminue
↑ Vers le haut	Y diminue

Les formes de base

p5.js met à ta disposition plusieurs fonctions pour dessiner. Voici les principales :

Le point

```javascript
point(x, y);
```

Dessine un seul pixel à la position (x, y). Pas très visible ! Utilise `strokeWeight()` pour le rendre plus gros.

```javascript
strokeWeight(8);
point(200, 150);   // Un gros point au centre
```

La ligne

```javascript
line(x1, y1, x2, y2);
```

Trace une ligne entre le point (x1, y1) et le point (x2, y2).

```javascript
line(50, 50, 350, 250);  // Ligne diagonale
```

Le rectangle

```javascript
rect(x, y, largeur, hauteur);
```

Dessine un rectangle. Par défaut, **(x, y) est le coin supérieur gauche**.

```javascript
rect(100, 80, 200, 120);  // Rectangle à partir du coin (100, 80)
```

💡 Pour que (x, y) soit le **centre** du rectangle au lieu du coin, utilise :

```javascript
rectMode(CENTER);
rect(200, 150, 200, 120);  // (200, 150) est maintenant le CENTRE
```

L'ellipse (et le cercle)

```javascript
ellipse(x, y, largeur, hauteur);
```

Dessine une ellipse **centrée** sur (x, y). Si largeur = hauteur, c'est un **cercle**.

```javascript
ellipse(200, 150, 100, 100);  // Cercle de diamètre 100
ellipse(200, 150, 150, 80);   // Ellipse aplatie
```

Le triangle

```javascript
triangle(x1, y1, x2, y2, x3, y3);
```

Dessine un triangle en reliant trois points.

```javascript
triangle(200, 50, 100, 250, 300, 250);  // Triangle pointant vers le haut
```

L'arc (portion de cercle)

```javascript
arc(x, y, largeur, hauteur, angleDebut, angleFin);
```

Dessine un arc de cercle. Les angles sont en **radians**, pas en degrés.

Angle	Radians	Constante p5.js
0°	0	`0`
90°	π/2	`HALF_PI`
180°	π	`PI`
270°	3π/2	`PI + HALF_PI`
360°	2π	`TWO_PI`

```javascript
arc(200, 150, 160, 160, 0, PI);  // Demi-cercle (de 0° à 180°)
```

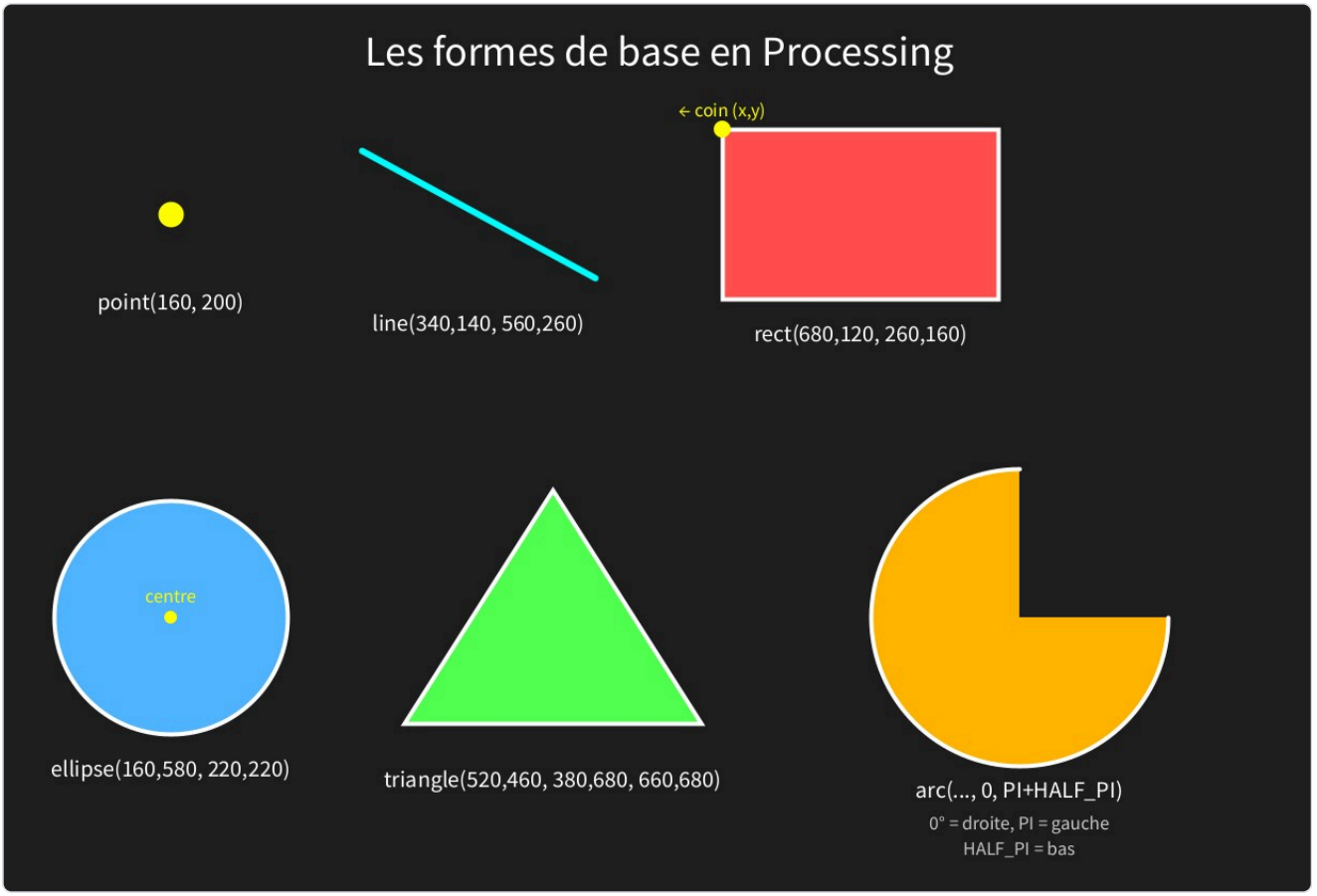


Tableau récapitulatif

Fonction	Paramètres	Dessine...
`point(x, y)`	Position	Un point
`line(x1, y1, x2, y2)`	Deux points	Une ligne
`rect(x, y, l, h)`	Coin + dimensions	Un rectangle

Fonction	Paramètres	Dessine...
`ellipse(x, y, l, h)`	Centre + dimensions	Une ellipse/cercle
`triangle(x1,y1, x2,y2, x3,y3)`	Trois points	Un triangle
`arc(x, y, l, h, début, fin)`	Centre + dimensions + angles	Un arc
`background(couleur)`	Couleur	Remplit tout le fond

Le remplissage et le contour

p5.js fonctionne comme si tu avais deux outils dans la main :

🎨 Le pot de peinture : `fill()`

`fill()` définit la couleur de **remplissage** (l'intérieur) de toutes les formes dessinées après.

```javascript
fill(200);              // Remplissage gris clair
rect(50, 50, 100, 80);  // Ce rectangle sera gris clair à l'intérieur
```

Pour désactiver le remplissage (forme "vide", transparente) :

```javascript
noFill();
```

✏️ Le stylo : `stroke()` et `strokeWeight()`

`stroke()` définit la couleur du **contour** (le trait autour des formes).

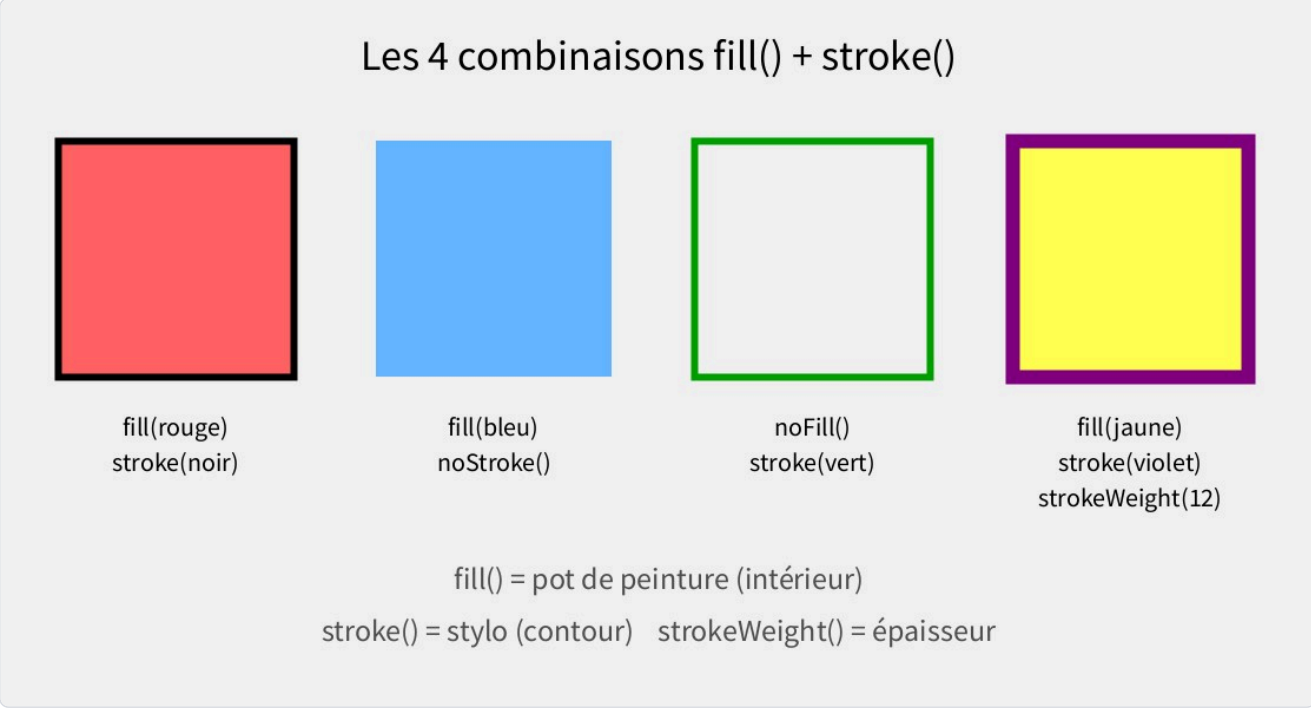
`strokeWeight()` définit l'**épaisseur** du trait en pixels.

```javascript
stroke(0);              // Contour noir
strokeWeight(4);         // Trait de 4 pixels d'épaisseur
rect(50, 50, 100, 80);  // Ce rectangle aura un gros contour noir
```

Pour désactiver le contour :

```javascript
noStroke();
```

Les 4 combinaisons possibles



Combinaison	Code	Résultat
Remplissage + contour	`fill(200); stroke(0);`	Forme grise avec bord noir
Remplissage seul	`fill(200); noStroke();`	Forme grise, pas de bord
Contour seul	`noFill(); stroke(0);`	Juste le contour, intérieur vide
Les deux personnalisés	`fill(200); stroke(0); strokeWeight(5);`	Forme grise avec gros bord noir

💡 Règle importante : l'effet "feutre"

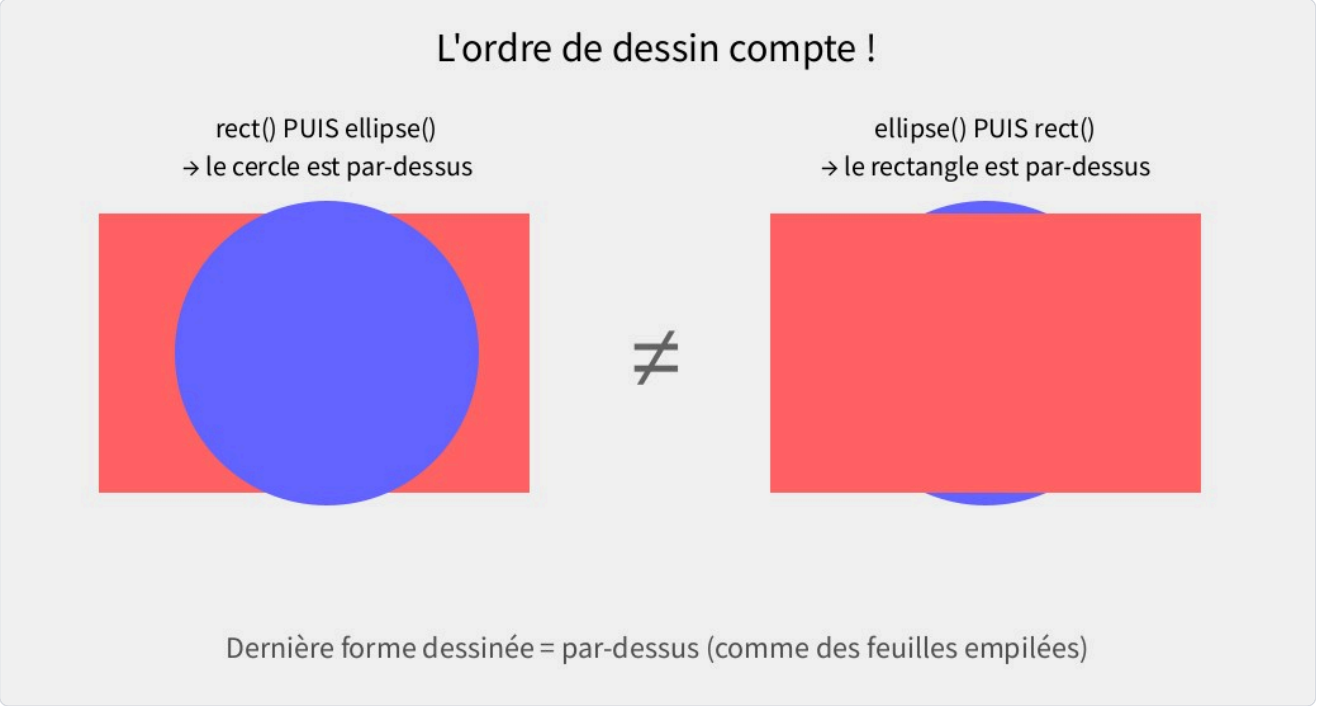
`fill()` et `stroke()` s'appliquent à **tout ce qui est dessiné après**, jusqu'au prochain appel. C'est comme si tu changeais de feutre : une fois que tu prends le rouge, tout ce que tu dessines est rouge jusqu'à ce que tu changes.

```javascript
fill(200);                  // "Je prends le feutre gris"
rect(20, 20, 80, 80);       // Rectangle gris
ellipse(200, 60, 80, 80);   // Cercle gris AUSSI !

fill(100);                  // "Je change pour le feutre gris foncé"
rect(300, 20, 80, 80);      // Rectangle gris foncé
```

L'ordre des instructions compte !

En p5.js, les formes sont dessinées **les unes par-dessus les autres**, dans l'ordre du code. La dernière forme dessinée est **par-dessus** toutes les autres.




```javascript
// Le cercle sera PAR-DESSUS le rectangle
fill(200);
rect(50, 50, 200, 100);     // Dessiné en premier → en dessous

fill(100);
ellipse(150, 100, 120, 120); // Dessiné en dernier → par-dessus
```

C'est comme quand tu empiles des feuilles de papier : la dernière posée est celle qu'on voit.

⚠️ **Conséquence** : si tu veux qu'un fond soit derrière tout, dessine-le **en premier** ! C'est le rôle de `background()` qu'on met au début du `draw()`.

Résumé

Outil	Fonction	Ce que ça fait
🎨 Remplissage	`fill(couleur)`	Couleur de l'intérieur des formes
🎨 Pas de remplissage	`noFill()`	Formes transparentes
✏️ Contour	`stroke(couleur)`	Couleur du trait/bord
✏️ Épaisseur	`strokeWeight(pixels)`	Épaisseur du trait
✏️ Pas de contour	`noStroke()`	Pas de trait autour des formes
📐 Mode rectangle	`rectMode(CENTER)`	Change le point de référence du rectangle
🖼️ Fond	`background(couleur)`	Remplit tout le canevas

💡 **Astuce finale** : quand tu te perds, rappelle-toi que chaque forme a besoin de **trois choses** :

1. Un **remplissage** (`fill()` ou `noFill()`)
2. Un **contour** (`stroke()` / `strokeWeight()` ou `noStroke()`)
3. Sa **position et taille** (les paramètres de la fonction)

1.2 Les Couleurs

> 🎯 ***Objectif*** *: Maîtriser le système de couleurs de p5.js, des niveaux de gris jusqu'aux couleurs RGB avec transparence. À la fin de ce chapitre, tu seras capable de colorier n'importe quelle forme avec la couleur exacte que tu veux.*

Les niveaux de gris

La façon la plus simple d'utiliser les couleurs en p5.js : **une seule valeur** entre 0 et 255.

Valeur	Couleur
`0`	⬛ Noir complet
`64`	Gris foncé
`128`	Gris moyen
`192`	Gris clair
`255`	⬜ Blanc complet

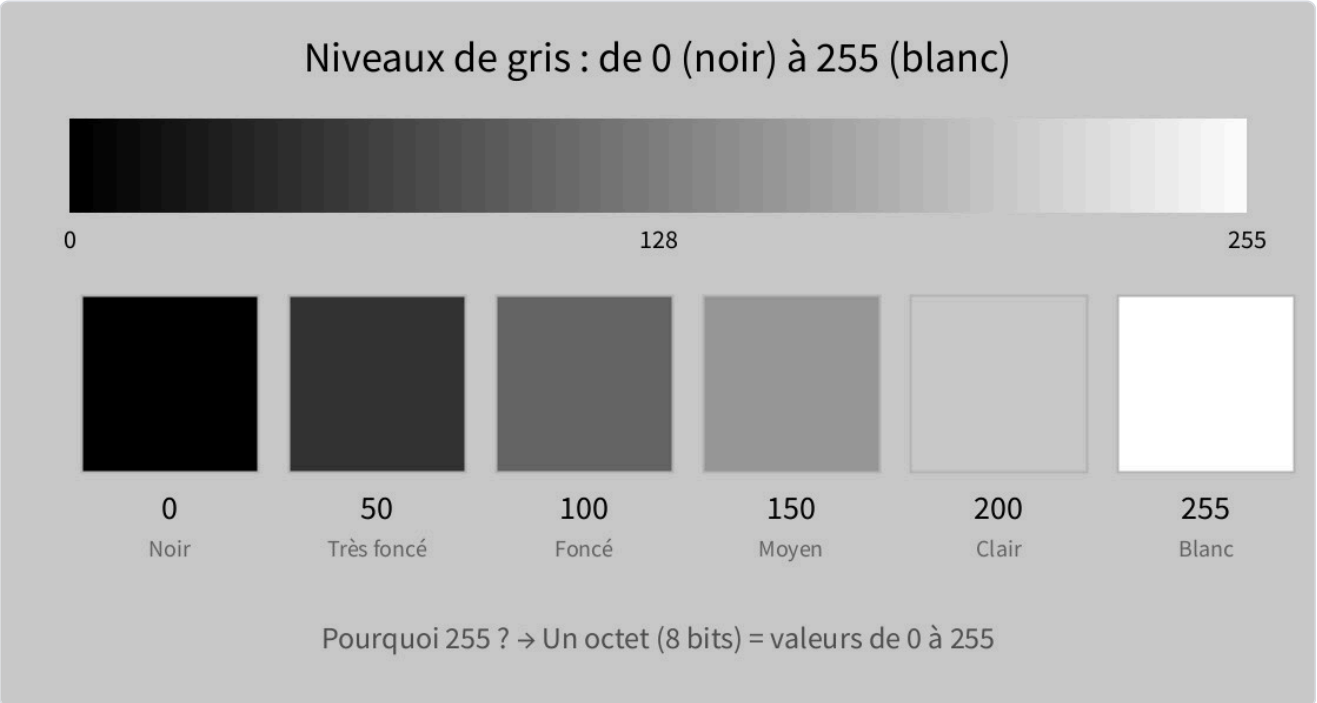
```javascript
background(240);        // Fond gris très clair

fill(0);                // Noir
rect(20, 20, 80, 80);

fill(128);              // Gris moyen
rect(120, 20, 80, 80);

fill(255);              // Blanc
rect(220, 20, 80, 80);
```

💡 **Pourquoi 255 et pas 100 ?** Parce qu'un octet (8 bits) peut stocker des valeurs de 0 à 255. C'est un standard en informatique que tu retrouveras partout : écrans, images, web... D'ailleurs, c'est exactement le même système que les couleurs CSS en `rgb()` !



Les couleurs RGB

Pour obtenir de vraies couleurs, on utilise le système **RGB** : **R**ouge, **V**ert (Green), **B**leu. Chaque composante va de 0 à 255.

```javascript
fill(rouge, vert, bleu);
```

Le principe est celui de la **synthèse additive** : on mélange de la lumière, pas de la peinture ! C'est comme les pixels de ton écran qui combinent trois petites LEDs (rouge, verte, bleue).

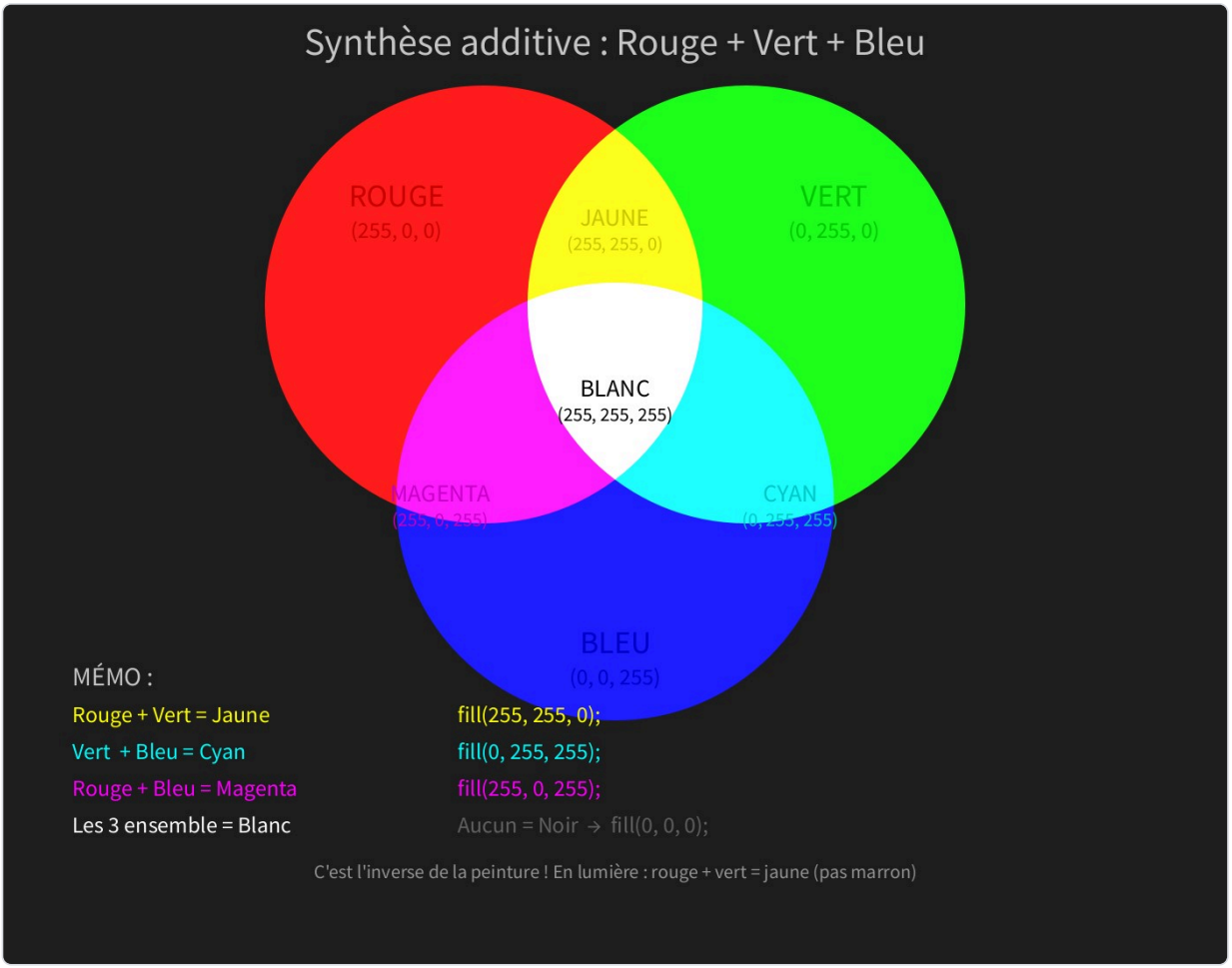
Les couleurs primaires

```javascript
fill(255, 0, 0);    // 🟥 Rouge pur — que du rouge, pas de vert, pas de bleu
fill(0, 255, 0);    // 🟩 Vert pur
fill(0, 0, 255);    // 🟦 Bleu pur
```

Les mélanges

```javascript
fill(255, 255, 0);  // 🟨 Rouge + Vert = Jaune
fill(0, 255, 255);  //    Vert + Bleu = Cyan
fill(255, 0, 255);  // 🟪 Rouge + Bleu = Magenta/Violet
fill(255, 255, 255); // ⬜ Les trois ensemble = Blanc
fill(0, 0, 0);      // ⬛ Aucun = Noir
```

⚠️ **Attention** : mélanger de la lumière, c'est l'inverse de la peinture ! En peinture, rouge + vert = marron. En lumière (RGB), rouge + vert = **jaune**. C'est surprenant mais c'est comme ça que fonctionnent tous les écrans.



Mémo des couleurs courantes

Voici les codes RGB des couleurs que tu utiliseras le plus souvent :

Couleur	Code RGB	Astuce pour retenir
⬛ Noir	`0, 0, 0`	Tout à zéro
⬜ Blanc	`255, 255, 255`	Tout au max
🟥 Rouge	`255, 0, 0`	Premier à fond
🟩 Vert	`0, 255, 0`	Deuxième à fond
🟦 Bleu	`0, 0, 255`	Troisième à fond
🟨 Jaune	`255, 255, 0`	Rouge + Vert
🟧 Orange	`255, 165, 0`	Rouge + un peu de Vert
🟪 Violet	`128, 0, 128`	Rouge + Bleu (à moitié)
Cyan	`0, 255, 255`	Vert + Bleu
🦩 Rose	`255, 192, 203`	Beaucoup de tout, surtout Rouge
🤎 Brun	`139, 69, 19`	Rouge dominant, peu de Vert
🪨 Gris	`128, 128, 128`	Les trois égaux (entre noir et blanc)

💡 **Astuce** : pas besoin de retenir tout ça par cœur ! Tu peux utiliser n'importe quel **color picker** en ligne, ou même celui intégré aux DevTools de ton navigateur. Tape "color picker" dans Google et il t'en affiche un directement.

Le lien avec CSS

Tu connais déjà les couleurs en CSS ? Bonne nouvelle, c'est le **même système RGB** :

CSS	p5.js	Résultat
`color: rgb(255, 0, 0);`	`fill(255, 0, 0);`	🟥 Rouge
`background-color: rgb(0, 0, 0);`	`background(0, 0, 0);`	⬛ Noir
`border-color: rgb(0, 0, 255);`	`stroke(0, 0, 255);`	🟦 Contour bleu

Les valeurs sont exactement les mêmes — seule la syntaxe change !

Où utiliser les couleurs ?

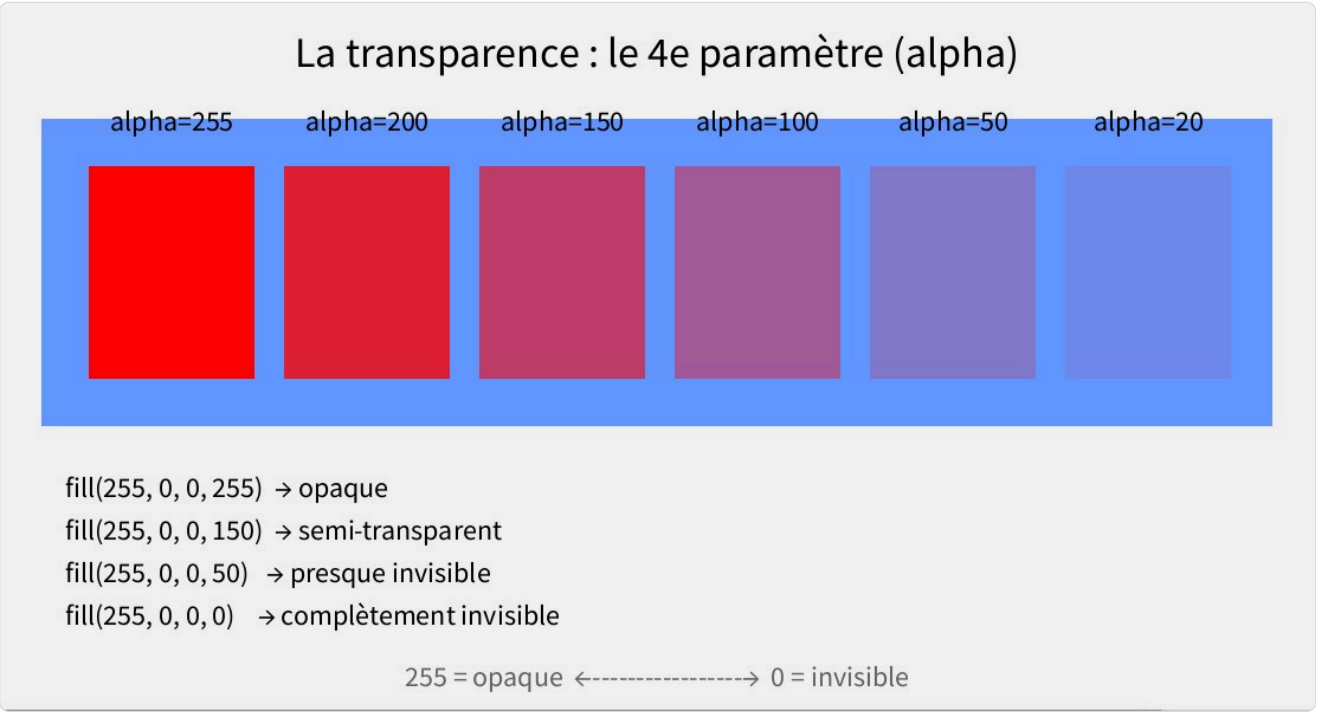
Les couleurs s'appliquent à trois endroits :

Fonction	Ce qu'elle colorie	Exemple
`background(r, g, b)`	Le **fond** du canevas	`background(135, 206, 235);` → ciel bleu
`fill(r, g, b)`	L'**intérieur** des formes	`fill(255, 0, 0);` → formes rouges
`stroke(r, g, b)`	Le **contour** des formes	`stroke(0, 0, 255);` → contour bleu

Chacune de ces fonctions accepte les mêmes formats :

```javascript
fill(128);            // 1 valeur  → niveau de gris
fill(255, 0, 0);      // 3 valeurs → couleur RGB
fill(255, 0, 0, 128); // 4 valeurs → couleur RGB + transparence
```

La transparence



On peut ajouter un **4e nombre** pour la transparence (appelée **alpha**), de 0 (invisible) à 255 (opaque) :

```javascript
fill(255, 0, 0, 255);  // Rouge complètement opaque (par défaut)
fill(255, 0, 0, 180);  // Rouge un peu transparent
fill(255, 0, 0, 100);  // Rouge assez transparent
fill(255, 0, 0, 30);   // Rouge presque invisible
```

C'est très utile pour superposer des formes et créer des effets de profondeur ou de verre :

```javascript
background(240);

// Cercle bleu
fill(0, 0, 255);
ellipse(180, 150, 150, 150);

// Cercle rouge semi-transparent par-dessus
fill(255, 0, 0, 120);
ellipse(250, 150, 150, 150);
```

💡 **Astuce** : la transparence fonctionne aussi avec `stroke()` :

```javascript
stroke(0, 0, 0, 100);  // Contour noir semi-transparent
```

Stocker une couleur dans une variable

p5.js possède une fonction `color()` pour créer des couleurs et les stocker dans des variables. C'est pratique quand tu utilises souvent la même couleur :

```javascript
let rouge = color(255, 0, 0);
let bleuCiel = color(135, 206, 235);

background(bleuCiel);
fill(rouge);
rect(50, 50, 100, 80);
```

On verra les variables en détail dans un prochain chapitre, mais sache que ça existe !

Mini-exemple : une scène colorée



Voici un exemple qui combine tout ce qu'on a vu dans les trois chapitres d'introduction — formes, couleurs, remplissage, contour et ordre de dessin :

```javascript
function setup() {
  createCanvas(500, 400);
}

function draw() {
  // Ciel
  background(135, 206, 235);

  // Soleil (jaune, sans contour)
  noStroke();
  fill(255, 220, 50);
  ellipse(420, 60, 80, 80);

  // Sol (vert)
  fill(80, 180, 80);
  rect(0, 300, 500, 100);

  // Maison : mur (marron avec contour)
  stroke(100, 60, 30);
  strokeWeight(2);
  fill(200, 120, 80);
  rect(150, 200, 200, 120);

  // Maison : toit (triangle rouge)
  fill(180, 40, 40);
  triangle(130, 200, 250, 120, 370, 200);

  // Maison : porte (marron foncé)
  fill(100, 60, 30);
  rect(220, 250, 50, 70);

  // Fenêtres (bleu clair)
  fill(180, 220, 255);
  rect(170, 230, 35, 35);
  rect(295, 230, 35, 35);
}
```

Remarque l'**ordre** : le ciel en premier (tout derrière), puis le sol, puis la maison, puis les détails par-dessus. Chaque élément recouvre les précédents.

Résumé

Concept	Syntaxe	Exemple
Niveau de gris	`fill(valeur)`	`fill(128)` → gris moyen
Couleur RGB	`fill(r, g, b)`	`fill(255, 0, 0)` → rouge
Avec transparence	`fill(r, g, b, a)`	`fill(255, 0, 0, 128)` → rouge transparent
Fond du canevas	`background(r, g, b)`	`background(135, 206, 235)` → ciel
Stocker une couleur	`let nom = color(r, g, b)`	`let c = color(255, 0, 0);`
Sélecteur de couleurs	Color picker Google / DevTools	Trouve n'importe quel code RGB

Valeur RGB	Règle
`0`	Rien de cette composante
`128`	Moitié
`255`	Maximum de cette composante

Valeur RGB	Règle
Les 3 égales	Niveau de gris (du noir au blanc)
Les 3 à 255	Blanc
Les 3 à 0	Noir

1.3 Premières Formes

> 🎯 ***Objectif*** *: Combiner formes, couleurs, remplissage et contour pour créer des compositions visuelles. À la fin de ce chapitre, tu seras capable de dessiner une scène complète en contrôlant précisément l'apparence de chaque élément.*

Rappel express

Dans les chapitres précédents, tu as découvert trois familles d'outils :

Famille	Ce que tu sais faire	Fonctions
📐 **Formes**	Dessiner des formes géométriques	`rect()`, `ellipse()`, `triangle()`, `line()`, `point()`, `arc()`
🎨 **Remplissage**	Colorier l'intérieur des formes	`fill()`, `noFill()`
✏️ **Contour**	Contrôler le trait autour des formes	`stroke()`, `noStroke()`, `strokeWeight()`

Ce chapitre, c'est la **mise en pratique**. On va combiner tout ça pour créer de vraies compositions, comme un artiste qui sort enfin ses pinceaux après avoir appris à les tenir.

Exercice 1 — Placement de formes

Consigne

Crée un canevas de **400×300** pixels avec un fond de la couleur de ton choix. Place dessus :

- Un **carré** (rappel : un carré, c'est un rectangle avec la même largeur et la même hauteur)
- Un **triangle**
- Une **ligne**

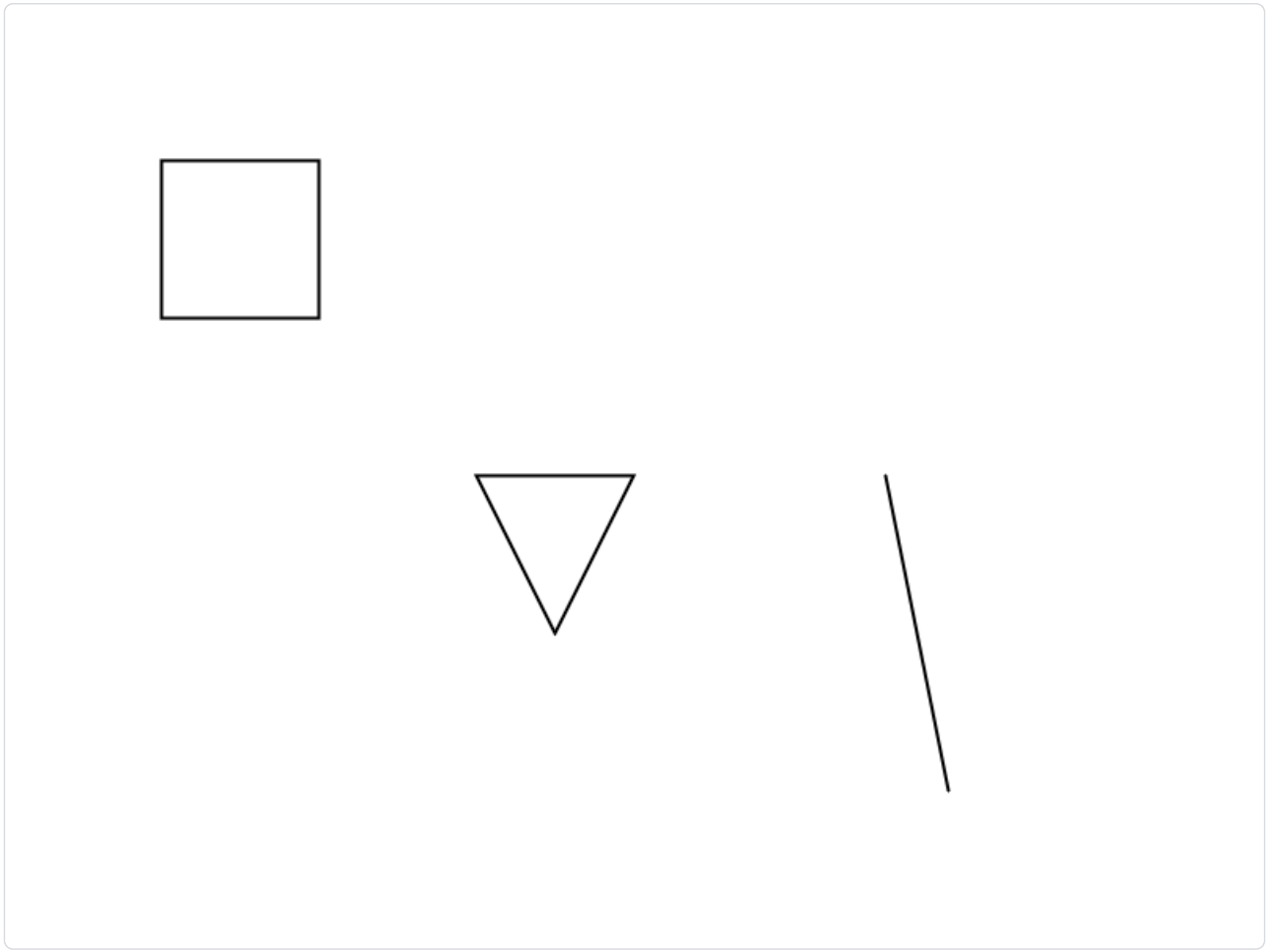
Les trois formes **ne doivent pas se chevaucher**.

Contraintes

- Utilise `createCanvas(400, 300)` dans le `setup()`
- Choisis un `background()` qui te plaît
- Laisse les couleurs par défaut (on les changera dans l'exercice 2)

Ce que tu devrais obtenir

Un canevas avec trois formes bien séparées. Par exemple :



Indices

💡 **Indice 1** — Un carré est un `rect()` dont la largeur et la hauteur sont **identiques** : `rect(50, 50, 80, 80)` fait un carré de 80×80 pixels.

💡 **Indice 2** — Ton canevas fait 400 pixels de large. Divise mentalement l'espace en trois zones : gauche (0 à ~130) pour le carré, centre (~130 à ~270) pour le triangle, droite (~270 à 400) pour la ligne.

💡 **Indice 3** — Structure du programme :

JAVASCRIPT

```
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(???);

  // Carré
  rect(???, ???, ???, ???);

  // Triangle
  triangle(???, ???, ???, ???, ???, ???);

  // Ligne
  line(???, ???, ???, ???);
}
```

Exercice 2 — Changement de couleurs

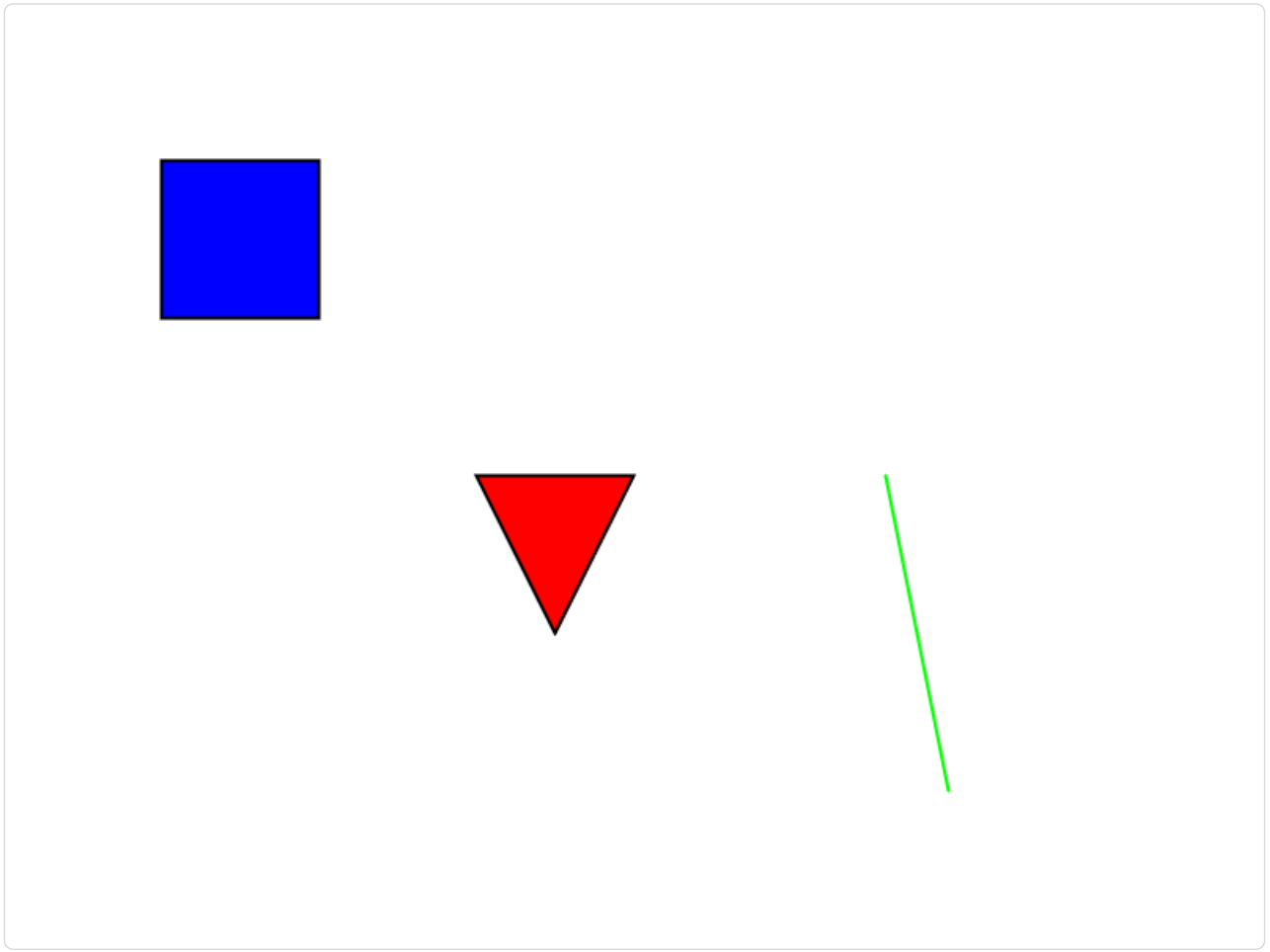
Consigne

Reprends ton code de l'exercice 1 et ajoute des couleurs :

- Le carré doit être rempli en **bleu** 🟦
- Le triangle doit être rempli en **rouge** 🟥
- La ligne doit être **verte** 🟩 et plus épaisse (au moins 3 pixels)

Contraintes

- Le carré doit avoir un **contour noir** visible
- Le triangle doit avoir un **contour noir** visible
- La ligne doit être **épaisse** (utilise `strokeWeight()`)



Indices

💡 **Indice 1** — Rappel des couleurs de base : `fill(0, 0, 255)` pour le bleu, `fill(255, 0, 0)` pour le rouge, `stroke(0, 255, 0)` pour un contour/ligne vert.

💡 **Indice 2** — Une ligne n'a **pas de remplissage** (`fill()` n'a aucun effet sur elle). Pour changer la couleur d'une ligne, c'est `stroke()` qu'il faut utiliser. Et pour l'épaissir : `strokeWeight()`.

💡 **Indice 3** — Si tu veux que le carré et le triangle aient un contour noir mais que la ligne soit verte, tu dois changer le `stroke()` entre les formes :

```javascript
stroke(0);            // Contour noir pour le carré et le triangle
fill(0, 0, 255);
rect(...);

fill(255, 0, 0);
triangle(...);

stroke(0, 255, 0);    // On passe au vert pour la ligne
strokeWeight(4);
line(...);
```

🎯 Défi bonus — Le drapeau

Tu as terminé les exercices 1 et 2 ? Dessine le drapeau d'un pays de ton choix ! Par exemple :

- 🇧🇪 **Belgique** : trois rectangles verticaux (noir, jaune, rouge)
- 🇫🇷 **France** : trois rectangles verticaux (bleu, blanc, rouge)
- 🇯🇵 **Japon** : fond blanc + cercle rouge au centre
- 🇩🇪 **Allemagne** : trois rectangles horizontaux (noir, rouge, jaune)
- 🇮🇹 **Italie** : trois rectangles verticaux (vert, blanc, rouge)

💡 **Astuce** : pour un drapeau, utilise `noStroke()` pour éviter les traits entre les bandes.

💡 **Astuce** : pour un drapeau à 3 bandes verticales qui remplit tout le canevas, chaque bande fait un tiers de la largeur. Par exemple avec `createCanvas(300, 200)`, chaque bande fait 100 pixels.

Résumé

Ce chapitre t'a permis de mettre en pratique les trois piliers du dessin en p5.js :

Pilier	Ce qu'il faut retenir
📐 **Formes**	Chaque fonction de dessin a ses propres paramètres (position, taille, points...)
🎨 **Couleurs**	`fill()` pour l'intérieur, `stroke()` pour le contour — ils restent actifs jusqu'au prochain appel
📋 **Ordre**	Ce qui est dessiné en dernier apparaît par-dessus — comme des couches de peinture

Et les bonnes habitudes à garder :

- Définis le style (`fill`, `stroke`, `strokeWeight`) **juste avant** chaque forme
- Pense en **couches** : fond → éléments lointains → éléments proches

Dans le prochain chapitre, on va découvrir les **variables** — et ça va tout changer. Au lieu d'écrire des nombres fixes partout, on pourra utiliser des noms pour stocker et réutiliser des valeurs. C'est le premier pas vers des programmes vraiment dynamiques !

1.4 Les Variables

> 🎯 ***Objectif*** *: Comprendre ce qu'est une variable, savoir en créer, et les utiliser pour positionner et dimensionner des formes de manière intelligente. À la fin de ce chapitre, tu seras capable de dessiner des formes qui s'adaptent à la taille du canevas.*

C'est quoi une variable ?

Imagine une **boîte** avec :

- une **étiquette** (le nom de la variable)
- une **valeur** à l'intérieur (le contenu)

```
┌──────────────┐
│     150      │  ← la valeur
└──────────────┘
  let posX       ← le mot-clé + le nom (l'étiquette)
```

En JavaScript (et donc en p5.js), ça s'écrit :

```javascript
let posX = 150;
```

Cette ligne fait trois choses d'un coup :

1. Elle crée une boîte avec `let`
2. Elle colle l'étiquette `posX` dessus
3. Elle met la valeur `150` à l'intérieur

Tu connais déjà ça !

Tu as déjà utilisé des variables en JavaScript. En p5.js, c'est **exactement la même syntaxe** :

JavaScript classique	p5.js
`let posX;`	`let posX;`
`posX = 150;`	`posX = 150;`
`let posX = 150;`	`let posX = 150;`
`posX = posX + 10;`	`posX = posX + 10;`

Zéro surprise : c'est du JavaScript pur et simple.

`let` vs `const`

En JavaScript, tu as deux façons principales de déclarer une variable :

Mot-clé	Quand l'utiliser	Exemple
`let`	La valeur peut **changer** au fil du temps	`let posX = 150;`
`const`	La valeur ne changera **jamais**	`const TAILLE_MAX = 500;`

```javascript
let score = 0;          // ✅ score va évoluer
score = 10;             // ✅ OK, on peut modifier un let

const GRAVITE = 9.81;   // ✅ la gravité ne change pas
GRAVITE = 10;           // ❌ Erreur ! On ne peut pas modifier un const
```

💡 **Bonne pratique** : utilise `const` quand la valeur ne doit pas changer, `let` pour le reste. Ça rend ton code plus clair et évite des bugs.

⚠️ **Et** `var` ? Tu as peut-être vu `var` dans du code JavaScript. C'est l'ancienne façon de déclarer des variables. On ne l'utilise plus — préfère toujours `let` ou `const`.

Les types en JavaScript

Contrairement à des langages comme Java ou C++, JavaScript ne demande **pas** de préciser le type. Il le devine tout seul. Mais il est important de connaître les types qui existent :

Type	Ce que ça stocke	Exemple
`number`	Un nombre (entier **ou** à virgule)	`let age = 25;` / `let prix = 9.99;`
`string`	Du texte	`let nom = "Alice";`
`boolean`	Vrai ou faux	`let estVisible = true;`

JAVASCRIPT

```
let largeur = 200;       // ✅ number (entier)
let vitesse = 3.5;       // ✅ number (à virgule)
let nom = "Alice";       // ✅ string
let actif = true;        // ✅ boolean
```

💡 **Avantage JavaScript** : pas besoin de choisir entre `int` et `float` comme dans d'autres langages. Un `number` gère les deux !

⚠️ **Attention** : en programmation, la virgule s'écrit avec un **point** ! On écrit `3.5` et pas `3,5`.

Modifier une variable

Une fois créée avec `let`, tu peux **changer** la valeur d'une variable autant de fois que tu veux. Par contre, tu ne remets **pas** le `let` :

JAVASCRIPT

```
let score = 0;       // Création : on met let
score = 10;          // Modification : PAS de let
score = score + 5;   // score vaut maintenant 15
```

⚠️ **Piège classique** : écrire `let score = 10;` puis plus loin `let score = 20;`. JavaScript va te dire que la variable existe déjà ! Le `let` ne se met qu'**une seule fois**, à la création.

Les opérations mathématiques

Les variables numériques supportent les opérations de base :

Opération	Symbole	Exemple	Résultat
Addition	`+`	`10 + 3`	`13`
Soustraction	`-`	`10 - 3`	`7`
Multiplication	`*`	`10 * 3`	`30`
Division	`/`	`10 / 3`	`3.3333...`

💡 **Bonne nouvelle** : contrairement à d'autres langages (Java, C++...), JavaScript ne fait **pas** de division entière. `10 / 3` donne bien `3.3333...` et pas `3`. Un souci en moins !

Tu peux bien sûr combiner variables et opérations :

```javascript
let cote = 100;
let demiCote = cote / 2;    // 50
let doubleCote = cote * 2;  // 200
```

Les variables spéciales de p5.js

p5.js te fournit des variables déjà toutes faites, prêtes à l'emploi. Pas besoin de les créer, elles existent automatiquement :

`width` et `height` — les dimensions du canevas

Variable	Ce qu'elle contient
`width`	La **largeur** du canevas (en pixels)
`height`	La **hauteur** du canevas (en pixels)

Si tu as écrit `createCanvas(400, 300)`, alors `width` vaut `400` et `height` vaut `300`.

Pourquoi c'est utile ? Parce que ça te permet de **positionner des formes par rapport au canevas**, sans connaître sa taille à l'avance :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  // Centre du canevas, quelle que soit la taille
  ellipse(width / 2, height / 2, 50, 50);
}
```

Essaie de changer `createCanvas(400, 300)` en `createCanvas(600, 400)` — le cercle reste **toujours au centre** ! Sans `width` et `height`, il faudrait recalculer à la main à chaque fois.

`mouseX` et `mouseY` — la position de la souris

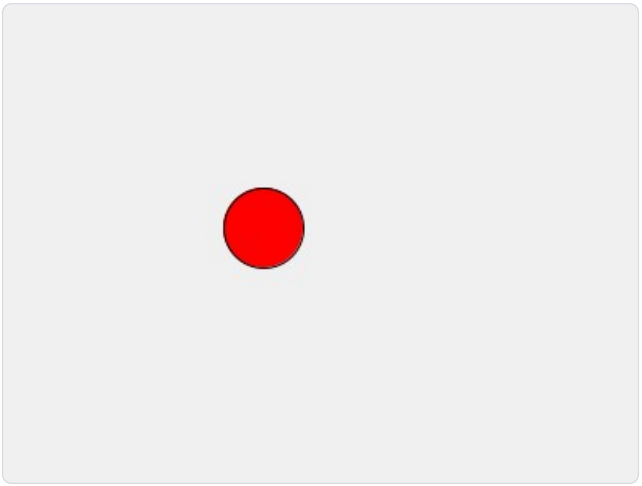
Variable	Ce qu'elle contient
`mouseX`	La position **horizontale** de la souris (en pixels depuis la gauche)
`mouseY`	La position **verticale** de la souris (en pixels depuis le haut)

Ces variables changent **60 fois par seconde** (à chaque passage dans `draw()`). C'est ce qui permet de créer de l'interactivité :

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  // Un cercle qui suit la souris
  fill(255, 0, 0);
  ellipse(mouseX, mouseY, 50, 50);
}
```



Essaie ce code ! Tu devrais voir un cercle rouge qui suit ta souris partout sur le canevas.

💡 **Expérience** : que se passe-t-il si tu **enlèves** la ligne `background(240)` ? Essaie, le résultat est surprenant ! (On reviendra sur cette astuce dans le chapitre sur l'animation.)

Bien nommer ses variables

Le nom d'une variable, c'est comme le nom d'un fichier sur ton ordinateur : il doit être **clair et descriptif**. Quelques règles :

Règle	✅ Bon	❌ Mauvais
Nom descriptif	`posX`, `largeurCarre`	`a`, `x1`, `truc`
Commence par une minuscule	`tailleCercle`	`TailleCercle`
Pas d'espaces ni d'accents	`couleurFond`	`couleur fond`, `côté`
Plusieurs mots → camelCase	`vitesseMax`	`vitesse_max`

La convention **camelCase** (première lettre en minuscule, puis majuscule à chaque nouveau mot) est le standard en JavaScript. Tu la retrouveras partout : dans les frameworks, les API, la doc…

Exercice 3 — Carré centré

Consigne

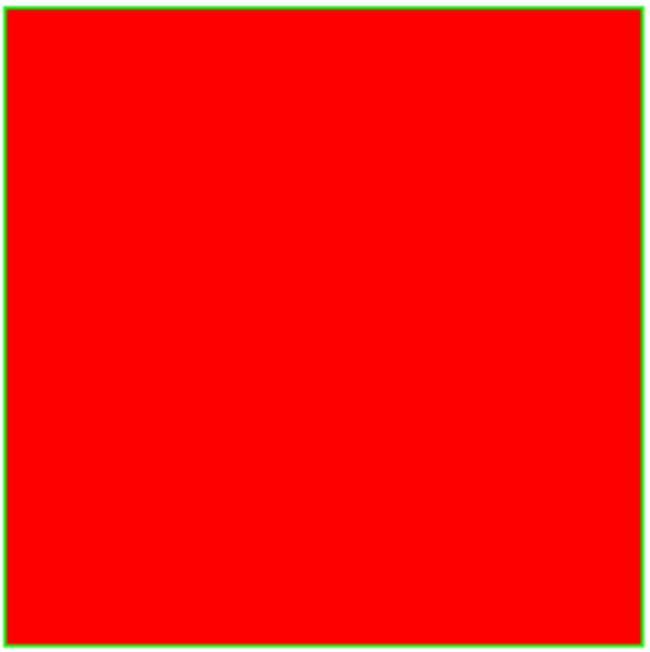
Dessine un **carré rouge** avec un **contour vert** parfaitement **au centre** de l'écran, en utilisant `width` et `height`.

Contraintes

- Le canevas fait `createCanvas(400, 300)`
- Le carré fait **100×100** pixels
- Le carré doit être **centré** sur le canevas
- Remplissage **rouge**, contour **vert**, épaisseur de contour de **3** pixels

Ce que tu devrais obtenir

Un carré rouge avec un contour vert, pile au milieu du canevas.



Indices

💡 **Indice 1** — Le centre du canevas est à la position `(width/2, height/2)`. Mais attention : par défaut, `rect()` dessine à partir du **coin supérieur gauche**, pas du centre !

💡 **Indice 2** — Tu as deux façons de centrer un rectangle :

- **Méthode calcul** : soustraire la moitié de la taille au point central → `rect(width/2 - 50, height/2 - 50, 100, 100)`
- **Méthode** `rectMode` : cherche `rectMode` dans la documentation de p5.js (https://p5js.org/reference). Il y a un mode qui change le comportement de `rect()` pour dessiner à partir du centre...

💡 **Indice 3** — `rectMode(CENTER)` fait en sorte que le premier paramètre de `rect()` soit le **centre** du rectangle, pas le coin. C'est beaucoup plus pratique pour centrer !

Exercice 4 — Dessiner dans le carré

Consigne

Reprends ton carré centré de l'exercice 3, et trace des **lignes à l'intérieur** en utilisant des variables pour tous les calculs de position.

Étape par étape :

1. Crée des variables `x`, `y` pour la position du coin supérieur gauche du carré, `cote` pour la taille du côté, et `demiCote` pour la moitié
2. Dessine le carré en utilisant ces variables
3. Trace la **médiane verticale** (ligne du milieu haut au milieu bas du carré)

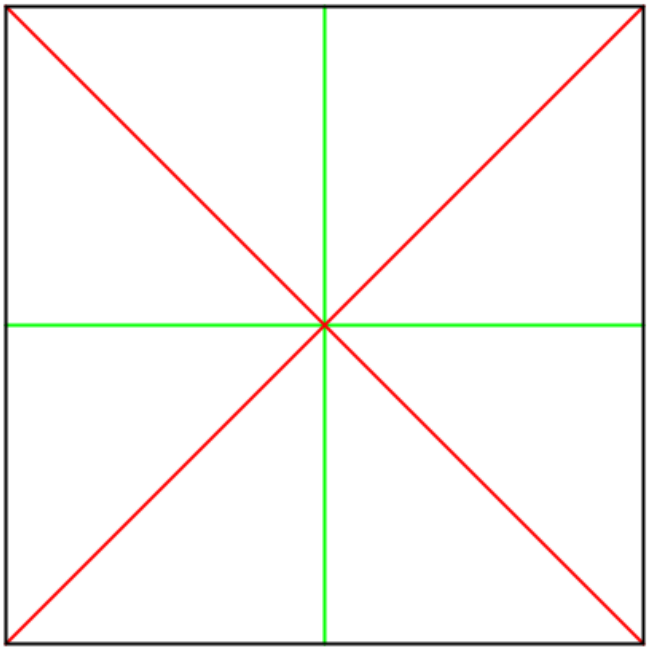
4. Trace la **médiane horizontale** (ligne du milieu gauche au milieu droit)
5. Trace les deux **diagonales** (coin à coin)

Contraintes

- Utilise **uniquement des variables** pour les positions (pas de nombres "en dur" sauf pour la taille initiale)
- Chaque ligne a une **couleur différente**
- Joue avec la **transparence** pour voir les lignes qui se croisent

Ce que tu devrais obtenir

Un carré avec 4 lignes colorées à l'intérieur : les deux médianes (en croix +) et les deux diagonales (en croix ×).



Indices

💡 **Indice 1** — Commence par créer tes variables. Si tu utilises `rectMode(CENTER)`, le coin supérieur gauche du carré est à `(width/2 - cote/2, height/2 - cote/2)`. Si tu préfères, tu peux aussi utiliser `rectMode(CORNER)` (le mode par défaut) et calculer `x` et `y` directement.

💡 **Indice 2** — La médiane verticale va du point `(x + demiCote, y)` au point `(x + demiCote, y + cote)`. La médiane horizontale va de `(x, y + demiCote)` à `(x + cote, y + demiCote)`.

💡 **Indice 3** — Les diagonales relient les coins opposés. La première va de `(x, y)` à `(x + cote, y + cote)`. La deuxième va de `(x + cote, y)` à `(x, y + cote)`.

🎯 Défi bonus — Le suiveur de souris

Reprends l'exercice 4, mais remplace les variables `x` et `y` par `mouseX` et `mouseY`. Ton carré (avec ses lignes intérieures) suivra la souris !

💡 **Astuce** : n'oublie pas le `background()` au début de `draw()`, sinon tu verras toutes les positions précédentes du carré.

Résumé

Concept	Syntaxe	Exemple
Créer une variable	`let nom = valeur;`	`let posX = 200;`
Créer une constante	`const nom = valeur;`	`const GRAVITE = 9.81;`
Modifier une variable	`nom = nouvelleValeur;`	`posX = posX + 10;`
Largeur du canevas	`width`	`ellipse(width/2, ...)`
Hauteur du canevas	`height`	`ellipse(..., height/2, ...)`
Position souris X	`mouseX`	`ellipse(mouseX, ...)`
Position souris Y	`mouseY`	`ellipse(..., mouseY, ...)`

Les points clés à retenir :

- Une variable se crée avec `let`, une constante avec `const`
- Le `let` / `const` ne se met qu'**une seule fois** (à la création)
- JavaScript devine le type automatiquement (`number`, `string`, `boolean`)
- `width`, `height`, `mouseX`, `mouseY` sont des variables **fournies par p5.js**
- Utiliser des variables rend ton code **flexible** : change une valeur, tout s'adapte

Dans le prochain chapitre, on va découvrir les **conditions** — le moyen de faire réagir ton programme différemment selon la situation. C'est le `if` qui ouvre la porte à l'interactivité !

1.5 Les Conditions

> 🎯 ***Objectif*** *: Faire réagir ton programme différemment selon la situation grâce à* `if`*,* `else if` *et* `else`*. À la fin de ce chapitre, tu seras capable de créer un dessin interactif qui change en fonction de la position de la souris et des clics.*

C'est quoi une condition ?

Dans la vie de tous les jours, tu prends des décisions en permanence :

- **SI** il pleut **ALORS** je prends un parapluie, **SINON** je mets mes lunettes de soleil
- **SI** le feu est rouge **ALORS** je m'arrête
- **SI** j'ai faim **ALORS** je mange

En programmation, c'est exactement pareil. Une **condition** permet à ton programme de faire des choix : exécuter certaines instructions **uniquement si** quelque chose est vrai.

Tu connais déjà ça !

Les conditions en p5.js sont **exactement les mêmes** qu'en JavaScript classique. Pas de nouvelle syntaxe à apprendre :

JavaScript classique	p5.js
`if (condition) { }`	`if (condition) { }`
`if (condition) { } else { }`	`if (condition) { } else { }`
`if ... else if ... else`	`if ... else if ... else`

La seule nouveauté, ce sont les **variables spéciales** de p5.js (`mouseX`, `mouseIsPressed`...) qu'on va utiliser dans nos conditions pour créer de l'interactivité.

La syntaxe du `if`

`if` seul — "si ... alors"

```javascript
if (condition) {
  // Ce code s'exécute UNIQUEMENT si la condition est vraie
}
```

Exemple concret :

```javascript
let temperature = 35;

if (temperature > 30) {
  fill(255, 0, 0);  // Rouge si c'est chaud
}
rect(50, 50, 100, 100);
```

Si `temperature` est supérieure à 30, le rectangle sera rouge. Sinon, il gardera la couleur par défaut (blanc).

`if ... else` — "si ... alors ... sinon"

```javascript
if (condition) {
  // Exécuté si la condition est VRAIE
} else {
  // Exécuté si la condition est FAUSSE
}
```

```javascript
let temperature = 15;

if (temperature > 30) {
  fill(255, 0, 0);  // Rouge si chaud
} else {
  fill(0, 0, 255);  // Bleu sinon
}
rect(50, 50, 100, 100);
```

C'est **l'un ou l'autre**, jamais les deux. Soit la condition est vraie et on passe dans le premier bloc, soit elle est fausse et on passe dans le `else`.

`if ... else if ... else` — plusieurs cas

Quand tu as plus de deux possibilités :

```javascript
let temperature = 22;

if (temperature > 30) {
  fill(255, 0, 0);    // Rouge : chaud
} else if (temperature > 20) {
  fill(255, 165, 0);  // Orange : tiède
} else if (temperature > 10) {
  fill(0, 255, 0);    // Vert : frais
} else {
  fill(0, 0, 255);    // Bleu : froid
}
rect(50, 50, 100, 100);
```

Les conditions sont testées **de haut en bas** et le programme s'arrête à la **première** qui est vraie. Si aucune n'est vraie, il passe dans le `else` (s'il y en a un).

⚠️ **Piège classique** : l'ordre des `else if` compte ! Si tu mets `temperature > 10` avant `temperature > 30`, la condition `> 10` sera vraie dès 11°C et le programme n'ira jamais tester `> 30`. Commence toujours par la condition la plus **restrictive**.

Les opérateurs de comparaison

Pour écrire des conditions, tu as besoin de **comparer** des valeurs :

Opérateur	Signification	Exemple	Vrai si...
`==`	Est égal à	`x == 10`	x vaut 10
`!=`	Est différent de	`x != 10`	x ne vaut pas 10
`<`	Est plus petit que	`x < 10`	x est strictement inférieur à 10
`>`	Est plus grand que	`x > 10`	x est strictement supérieur à 10
`<=`	Est plus petit ou égal	`x <= 10`	x vaut 10 ou moins
`>=`	Est plus grand ou égal	`x >= 10`	x vaut 10 ou plus

⚠️ **ATTENTION** : pour comparer, c'est **trois** signes égal `==` en JavaScript. Un seul `=`, c'est pour **assigner** une valeur. C'est l'erreur n°1 des débutants !

```javascript
let x = 10;       // ✅ Assigner : "x reçoit la valeur 10"
if (x == 10) {    // ✅ Comparer : "est-ce que x vaut 10 ?"
if (x = 10) {     // ❌ ERREUR ! Ça assigne au lieu de comparer
```

💡 `==` vs `=` en JavaScript : tu verras parfois `=` (deux signes) dans du code. La différence : `=` convertit les types avant de comparer (`"5" = 5` est vrai), tandis que `==` compare aussi le type (`"5" == 5` est faux). Préfère toujours `==`, c'est plus fiable et prévisible.

Combiner des conditions

Tu peux combiner plusieurs conditions avec les opérateurs logiques :

Opérateur	Signification	Exemple
`&&`	**ET** (les deux doivent être vraies)	`x > 0 && x < 100`
`		`
`!`	**NON** (inverse la condition)	`!mouseIsPressed`

```javascript
// Le clic est dans le quart supérieur gauche ?
if (mouseX < width / 2 && mouseY < height / 2) {
  fill(255, 0, 0);  // Rouge
}
```

C'est comme en français : "SI la souris est à gauche **ET** en haut **ALORS** c'est rouge."

Détecter la souris

p5.js fournit des variables et fonctions spéciales pour l'interactivité :

Variable/Fonction	Ce qu'elle fait
`mouseX`	Position horizontale de la souris
`mouseY`	Position verticale de la souris
`mouseIsPressed`	`true` si un bouton de la souris est enfoncé, `false` sinon
`mouseButton`	Quel bouton est enfoncé : `LEFT`, `RIGHT` ou `CENTER`

⚠️ **Différence importante** : en p5.js, c'est `mouseIsPressed` (avec "Is" au milieu). C'est une des rares différences de nom par rapport à Processing qui utilise `mousePressed`.

Exemple : changer de couleur au clic

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  if (mouseIsPressed) {
    fill(255, 0, 0);    // Rouge quand on clique
  } else {
    fill(200);          // Gris sinon
  }

  ellipse(width / 2, height / 2, 150, 150);
}
```

Exemple : détecter gauche vs droite

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  if (mouseX < width / 2) {
    background(100, 150, 255);  // Bleu à gauche
  } else {
    background(255, 150, 100);  // Orange à droite
  }

  // Ligne de séparation
  stroke(255);
  strokeWeight(2);
  line(width / 2, 0, width / 2, height);
}
```

Essaie ce code : le fond change de couleur quand ta souris passe d'un côté à l'autre !

Quel bouton de souris ?

`mouseIsPressed` détecte **tous** les boutons de la souris (gauche, droit, molette). Pour savoir **quel** bouton est enfoncé, on utilise la variable `mouseButton` :

Variable	Valeur	Signification
`mouseButton`	`LEFT`	Bouton gauche
`mouseButton`	`RIGHT`	Bouton droit
`mouseButton`	`CENTER`	Molette (clic du milieu)

On combine les deux avec `&&` :

```javascript
if (mouseIsPressed && mouseButton === LEFT) {
  // Le bouton GAUCHE est enfoncé
}
```

Exemple : dessiner au clic gauche

```javascript
function setup() {
  createCanvas(400, 300);
  background(240);
}

function draw() {
  if (mouseIsPressed && mouseButton === LEFT) {
    noStroke();
    fill(255, 0, 0);
    ellipse(mouseX, mouseY, 20, 20);
  }
}
```

Essaie ce code ! Tant que tu maintiens le clic gauche, tu dessines des cercles rouges comme un pinceau. `draw()` s'exécute 60 fois par seconde, donc le tracé est **fluide**.

💡 **Pourquoi pas de** `background()` dans `draw()` ? — Parce qu'on veut que les cercles **restent affichés**. Si on repeint le fond à chaque frame, les cercles précédents disparaissent. En mettant `background()` uniquement dans `setup()`, les dessins s'accumulent.

Exercice 5 — Dessin interactif par zones

Consigne

Le canevas est divisé en **4 zones** (quadrants). Quand on **maintient le clic gauche**, des cercles apparaissent à la position de la souris, mais leur **couleur dépend de la zone** :

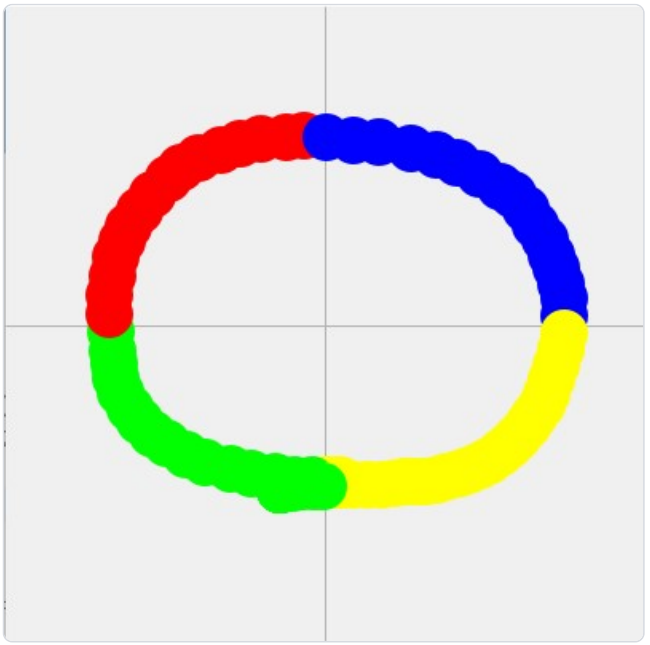
Zone	Position	Couleur
Haut-gauche	`mouseX < width/2` et `mouseY < height/2`	🟥 Rouge
Haut-droite	`mouseX ⩾ width/2` et `mouseY < height/2`	🟦 Bleu
Bas-gauche	`mouseX < width/2` et `mouseY ⩾ height/2`	🟩 Vert
Bas-droite	`mouseX ⩾ width/2` et `mouseY ⩾ height/2`	🟨 Jaune

Contraintes

- Canevas de `createCanvas(400, 400)`
- Le fond est dessiné **une seule fois** dans `setup()` (pas dans `draw()`) → les cercles restent visibles
- On dessine en continu avec `mouseIsPressed && mouseButton ⩵ LEFT` dans `draw()`
- Chaque cercle fait **30 pixels** de diamètre
- Dessine les lignes de séparation des 4 zones (croix au centre de l'écran)

Ce que tu devrais obtenir

Un canevas divisé en 4 zones visibles. En maintenant le clic gauche et en déplaçant la souris, des cercles colorés apparaissent comme un pinceau, avec la couleur correspondant à la zone.



Indices

💡 **Indice 1** — Pour que les cercles restent affichés et ne disparaissent pas, **ne mets pas** `background()` dans `draw()`. Mets-le dans `setup()` à la place. Comme `draw()` ne repeint pas le fond, les cercles s'accumulent.

💡 **Indice 2** — Pour détecter le quadrant, combine deux conditions avec `&&` :

```javascript
if (mouseX < width / 2 && mouseY < height / 2) {
  // Haut-gauche
}
```

💡 **Indice 3** — Structure globale du programme :

```javascript
function setup() {
  createCanvas(400, 400);
  background(240);

  // Lignes de séparation
  stroke(180);
  line(width / 2, 0, width / 2, height);
  line(0, height / 2, width, height / 2);
}

function draw() {
  if (mouseIsPressed && mouseButton === LEFT) {
    noStroke();

    // Détecte la zone et choisis la couleur
    if (mouseX < width / 2 && mouseY < height / 2) {
      fill(255, 0, 0);  // Rouge
    } else if (???) {
      // ...à toi de compléter !
    }

    ellipse(mouseX, mouseY, 30, 30);
  }
}
```

🎯 Défi bonus — La zone secrète

Ajoute une **5e zone circulaire** au centre de l'écran (un cercle de 100 pixels de diamètre). Si on dessine dans cette zone, les cercles sont **blancs**.

💡 **Astuce** : pour savoir si un point est dans un cercle, il faut calculer la **distance** entre le point (le clic) et le centre du cercle. p5.js a une fonction `dist()` toute prête :

```javascript
let d = dist(mouseX, mouseY, width / 2, height / 2);
if (d < 50) {
  // Le clic est dans la zone centrale (rayon = 50 pixels)
}
```

Pense à tester cette condition **en premier** (avant les quadrants), sinon les dessins au centre seront capturés par les quadrants avant d'arriver à la zone centrale !

Résumé

Concept	Syntaxe	Exemple
Si	`if (condition) { }`	`if (x > 10) { ... }`
Si ... sinon	`if (condition) { } else { }`	`if (x > 10) { ... } else { ... }`
Si ... sinon si ... sinon	`if ... else if ... else`	Tester plusieurs cas
Égal	`==`	`if (x == 10)`
Différent	`!=`	`if (x != 10)`
Plus petit / plus grand	`<`, `>`, `<=`, `>=`	`if (x < 100)`
ET logique	`&&`	`if (x > 0 && x < 100)`
OU logique	`	`	
Souris enfoncée	`mouseIsPressed`	`if (mouseIsPressed) { }`
Bouton de souris	`mouseButton`	`if (mouseButton == LEFT) { }`
Distance	`dist(x1, y1, x2, y2)`	`dist(mouseX, mouseY, 200, 200)`

JavaScript classique	p5.js
`if (condition) { }`	`if (condition) { }` (identique !)
`element.addEventListener('click', ...)`	`mouseIsPressed` / `mouseButton`
`event.clientX` / `event.clientY`	`mouseX` / `mouseY`

Les points clés à retenir :

- `==` pour comparer, `=` pour assigner — ne les confonds pas !
- Teste les conditions de la plus **restrictive** à la plus **générale**
- `&&` = les deux conditions doivent être vraies, `||` = au moins une suffit
- `mouseIsPressed` détecte le clic, `mouseButton` précise **quel** bouton (`LEFT`, `RIGHT`, `CENTER`)

Dans le prochain chapitre, on va découvrir les **boucles** — le moyen de répéter des instructions sans copier-coller, pour dessiner des motifs, des grilles et des dégradés en quelques lignes !

1.6 Les Boucles

> 🎯 ***Objectif*** *: Comprendre pourquoi et comment répéter des instructions automatiquement avec la boucle* `for`*. À la fin de ce chapitre, tu seras capable de dessiner des motifs répétitifs, des cercles concentriques et même un damier, en quelques lignes de code.*

Pourquoi des boucles ?

Imagine qu'on te demande de dessiner **10 cercles** les uns à côté des autres. Sans boucle, tu écrirais :

```javascript
ellipse(30, 150, 40, 40);
ellipse(80, 150, 40, 40);
ellipse(130, 150, 40, 40);
ellipse(180, 150, 40, 40);
ellipse(230, 150, 40, 40);
ellipse(280, 150, 40, 40);
ellipse(330, 150, 40, 40);
ellipse(380, 150, 40, 40);
ellipse(430, 150, 40, 40);
ellipse(480, 150, 40, 40);
```

Dix lignes quasi identiques 😩 Et si on t'en demande 100 ? 1000 ? Tu vas copier-coller pendant des heures.

En programmation, **copier-coller, c'est un signal d'alarme**. Chaque fois que tu te retrouves à écrire plusieurs fois la même chose en changeant juste un nombre, c'est qu'il te faut une **boucle**.

Tu connais déjà ça !

La boucle `for` en p5.js est **exactement la même** qu'en JavaScript classique. La seule différence avec d'autres langages : on déclare le compteur avec `let` au lieu de `int` :

Autre langage (Java, C++...)	JavaScript / p5.js
`for (int i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)`

Le résultat est le même : le code à l'intérieur s'exécute 10 fois. Et tu as un **compteur** (`i`) qui te dit à quelle répétition tu en es.

Anatomie d'une boucle `for`

```javascript
for (let i = 0; i < 10; i++) {
  // Ce code s'exécute 10 fois
  // i vaut 0, puis 1, puis 2, ... puis 9
}
```

Il y a trois parties séparées par des **points-virgules** :

```bash
for ( initialisation ; condition ; incrémentation )
       ↓                ↓             ↓
   let i = 0        i < 10         i++
   "On commence     "Tant que      "À chaque tour,
    à 0"             i < 10,        on ajoute 1
                     on continue"   à i"
```

Partie	Ce qu'elle fait	Dans notre exemple
Initialisation	Crée le compteur et lui donne une valeur de départ	`let i = 0` → on commence à 0
Condition	Vérifie si on continue la boucle	`i < 10` → tant que i est plus petit que 10
Incrémentation	Modifie le compteur à chaque tour	`i++` → on ajoute 1 à i

💡 `i++` est un raccourci pour `i = i + 1`. C'est la même chose, juste plus court.

Ce qui se passe à chaque tour

Voici le déroulé complet d'un `for (let i = 0; i < 5; i++)` :

Tour	Valeur de `i`	Condition `i < 5` ?	Action
1	0	✅ Oui	Exécute le code
2	1	✅ Oui	Exécute le code
3	2	✅ Oui	Exécute le code
4	3	✅ Oui	Exécute le code
5	4	✅ Oui	Exécute le code
6	5	❌ Non	STOP, on sort de la boucle

La boucle s'exécute **5 fois** (pour i = 0, 1, 2, 3, 4). Quand `i` atteint 5, la condition `i < 5` est fausse → on s'arrête.

⚠️ **Piège classique** : la boucle commence à **0**, pas à 1. Et elle s'arrête **avant** d'atteindre la limite. `i < 5` donne les valeurs 0, 1, 2, 3, 4 → c'est bien **5 tours**, mais le dernier tour est à `i = 4`, pas `i = 5`.

La puissance du compteur `i`

Le compteur `i` n'est pas juste un numéro de tour : tu peux l'**utiliser dans tes calculs**. C'est ça qui rend la boucle `for` si puissante.

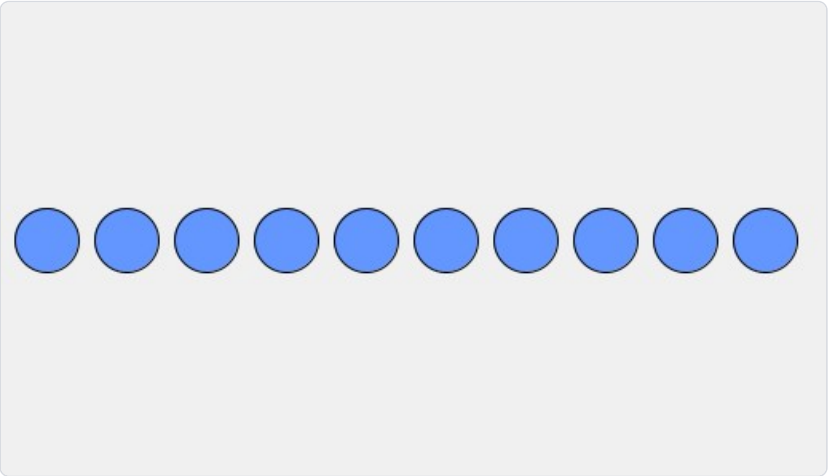
Reprenons nos 10 cercles du début :

```javascript
function setup() {
  createCanvas(520, 300);
}

function draw() {
  background(240);

  fill(100, 150, 255);
  stroke(0);

  for (let i = 0; i < 10; i++) {
    ellipse(30 + i * 50, 150, 40, 40);
  }
}
```



Décortiquons `30 + i * 50` :

Tour	`i`	`30 + i * 50`	Position X
1	0	30 + 0 × 50	**30**

Tour	`i`	`30 + i * 50`	Position X
2	1	30 + 1 × 50	**80**
3	2	30 + 2 × 50	**130**
…	…	…	…
10	9	30 + 9 × 50	**480**

Deux lignes de code au lieu de dix. Et si tu veux 100 cercles ? Change juste `i < 10` en `i < 100` 😎

Utiliser `i` pour varier les couleurs

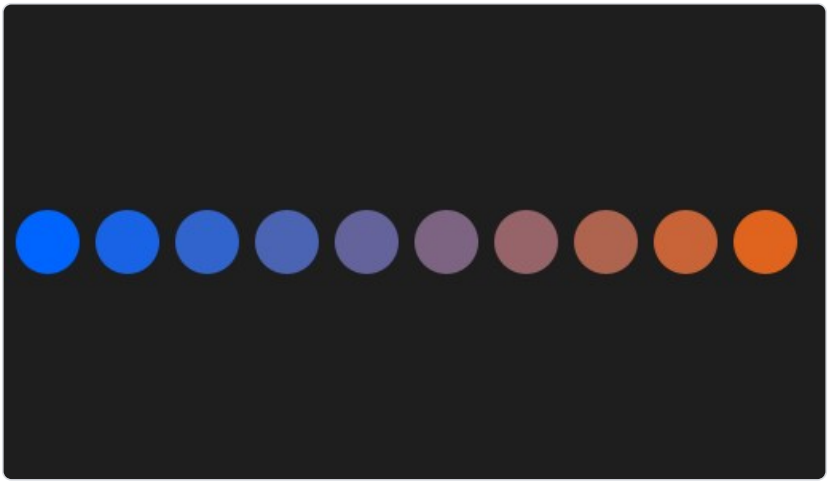
Le compteur `i` peut aussi servir à calculer des couleurs qui changent à chaque tour :

JAVASCRIPT

```
function setup() {
  createCanvas(520, 300);
}

function draw() {
  background(30);
  noStroke();

  for (let i = 0; i < 10; i++) {
    fill(i * 25, 100, 255 - i * 25);  // La couleur varie !
    ellipse(30 + i * 50, 150, 40, 40);
  }
}
```



Ici, `i * 25` fait varier la composante rouge de 0 à 225, et `255 - i * 25` fait varier le bleu en sens inverse. Résultat : un joli dégradé du bleu au rouge !

Utiliser une variable pour le nombre de répétitions

Plutôt que d'écrire un nombre "en dur" dans la boucle, utilise une **variable** :

JAVASCRIPT

```
let nbCercles = 10;

for (let i = 0; i < nbCercles; i++) {
  ellipse(30 + i * 50, 150, 40, 40);
}
```

L'avantage ? Tu changes **une seule valeur** et tout s'adapte. C'est exactement le même principe que les variables du chapitre précédent : rendre ton code **flexible**.

Les boucles imbriquées (grilles 2D)

Une boucle `for`, c'est une répétition en **une dimension** (une rangée). Deux boucles `for` imbriquées, c'est une répétition en **deux dimensions** (une grille).

```javascript
function setup() {
  createCanvas(400, 300);
}

function draw() {
  background(240);

  for (let ligne = 0; ligne < 3; ligne++) {
    for (let colonne = 0; colonne < 4; colonne++) {
      rect(20 + colonne * 90, 20 + ligne * 90, 80, 80);
    }
  }
}
```



Comment ça marche ? La boucle extérieure (`ligne`) parcourt les lignes, et pour **chaque ligne**, la boucle intérieure (`colonne`) dessine tous les éléments de cette ligne :

```
ligne=0 : colonne=0, colonne=1, colonne=2, colonne=3   → 1ère rangée
ligne=1 : colonne=0, colonne=1, colonne=2, colonne=3   → 2ème rangée
ligne=2 : colonne=0, colonne=1, colonne=2, colonne=3   → 3ème rangée
```

C'est comme lire un livre : on va de gauche à droite (colonnes), puis on passe à la ligne suivante.

💡 **Astuce** : nomme tes compteurs `ligne` et `colonne` (ou `row` et `col`) plutôt que `i` et `j`. C'est plus clair quand on relit le code.

Exercice 6 — L'arc-en-ciel (sans boucle)

Consigne

Dessine un **arc-en-ciel** : des demi-cercles concentriques de couleurs différentes. Le centre de l'arc-en-ciel est en **bas du canevas**. Ajoute un petit paysage autour (sol vert, ciel bleu, soleil...).

Les couleurs de l'arc-en-ciel, du plus grand au plus petit : rouge, orange, jaune, vert, bleu, indigo, violet.

Contraintes

- Canevas de `createCanvas(500, 400)`
- Utilise `arc()` avec les angles `0` et `PI` (demi-cercle supérieur) — ou bien des `ellipse()` que tu empiles
- **Pas de boucle** pour cet exercice (on fera la version boucle dans l'exercice suivant)
- `noStroke()` pour éviter les contours entre les arcs

Ce que tu devrais obtenir

Un arc-en-ciel avec 7 bandes de couleurs au-dessus d'un paysage simple.



Indices

💡 **Indice 1** — L'astuce est de dessiner des **ellipses** (ou arcs) du plus grand au plus petit, comme des poupées russes. La plus grande ellipse (rouge) est dessinée en premier, puis l'orange par-dessus (un peu plus petite), etc. La dernière ellipse (violette) est la plus petite.

💡 **Indice 2** — Centre tes demi-cercles en bas du canevas, par exemple à la position `(250, 400)`. Commence avec un diamètre de 400, puis diminue de 50 à chaque bande.

💡 **Indice 3** — Tu peux utiliser `arc(250, 400, 400, 400, PI, TWO_PI)` pour un demi-cercle vers le haut. Ou, plus simple : dessine des `ellipse()` complètes et cache le bas avec le rectangle du sol !

Exercice 7 — Cercles concentriques (avec boucle)

Consigne

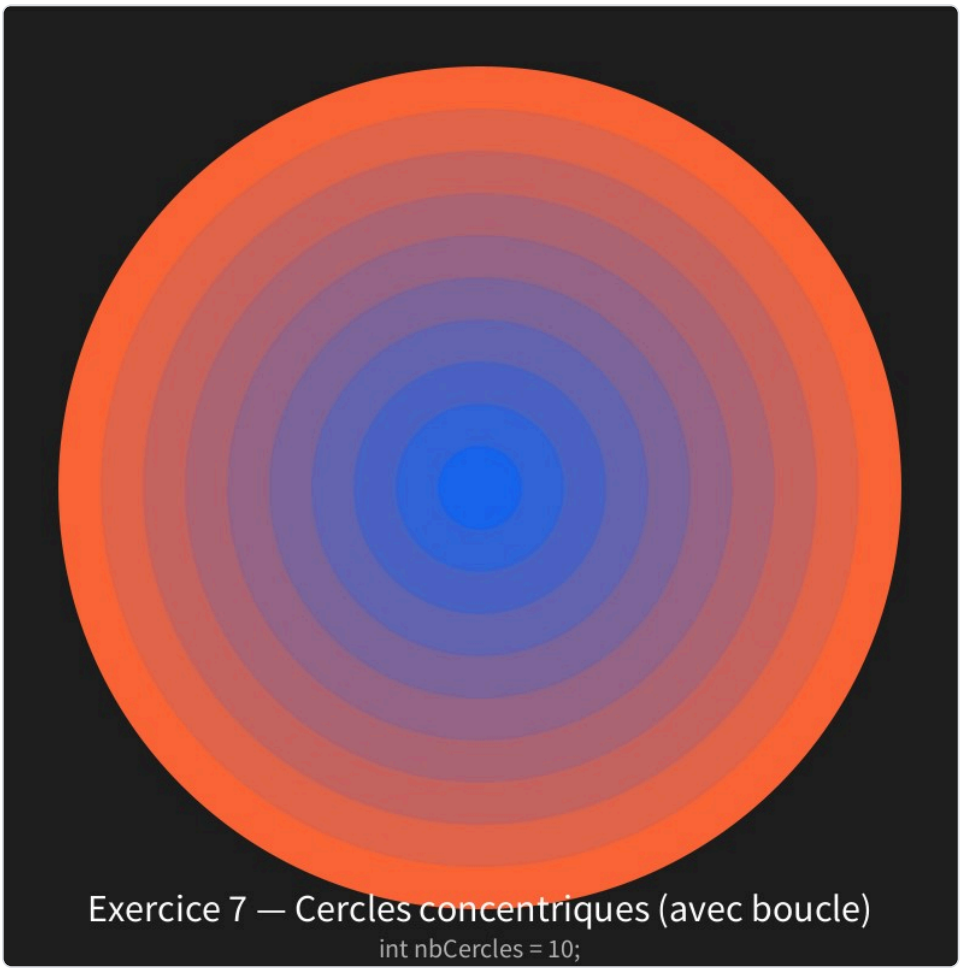
Reprends le principe des cercles concentriques, mais cette fois avec une **boucle** `for`. Le nombre de cercles est stocké dans une variable `nbCercles`. Chaque cercle a une **couleur différente** calculée automatiquement à partir du compteur `i`.

Contraintes

- Canevas de `createCanvas(400, 400)`
- Les cercles sont **centrés** au milieu du canevas
- Le nombre de cercles est dans une variable (commence avec `nbCercles = 8`)
- Chaque cercle a une couleur calculée avec `i` (pas de couleurs écrites à la main)
- `noStroke()` pour un rendu propre

Ce que tu devrais obtenir

Des cercles concentriques avec un dégradé de couleurs automatique, du plus grand (extérieur) au plus petit (intérieur).



Indices

💡 **Indice 1** — Attention à l'**ordre** ! Il faut dessiner du **plus grand au plus petit** (sinon les grands cercles cachent les petits). Astuce : fais ta boucle à l'envers avec `for (let i = nbCercles; i > 0; i--)`, ou bien calcule le rayon en partant du plus grand.

💡 **Indice 2** — Le diamètre du cercle `i` peut être `i * 40`. La couleur peut varier avec `fill(i * 30, 100, 255 - i * 30)`.

💡 **Indice 3** — Pour le bonus : remplace `nbCercles = 8` par `nbCercles = mouseX / 20`. Le nombre de cercles changera en fonction de la position horizontale de la souris !

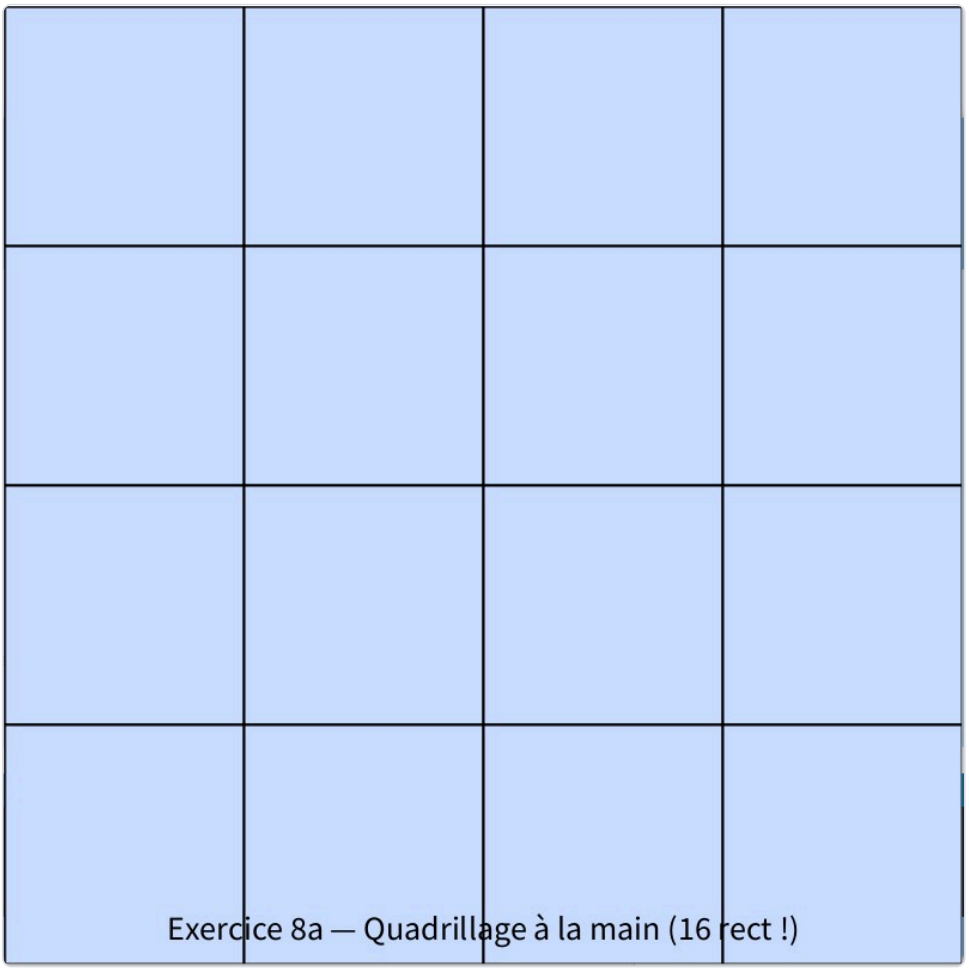
Exercice 8 — Le damier (boucles imbriquées)

Cet exercice se fait en **trois étapes progressives**.

Étape 1 — Quadrillage à la main (sans boucle)

Dessine un quadrillage de **4×4 carrés** de 100×100 pixels, tous de la même couleur. Oui, ça va être long et répétitif — c'est fait exprès ☺

Faites des copier-coller pour aller plus vite !



Étape 2 — Quadrillage avec boucles

Remplace tes 16 lignes de `rect()` par **2 boucles** `for` imbriquées. Le nombre de cases par côté est dans une variable `nbCases`.

Étape 3 — Damier

Alterne les couleurs **noir et blanc** comme un échiquier.

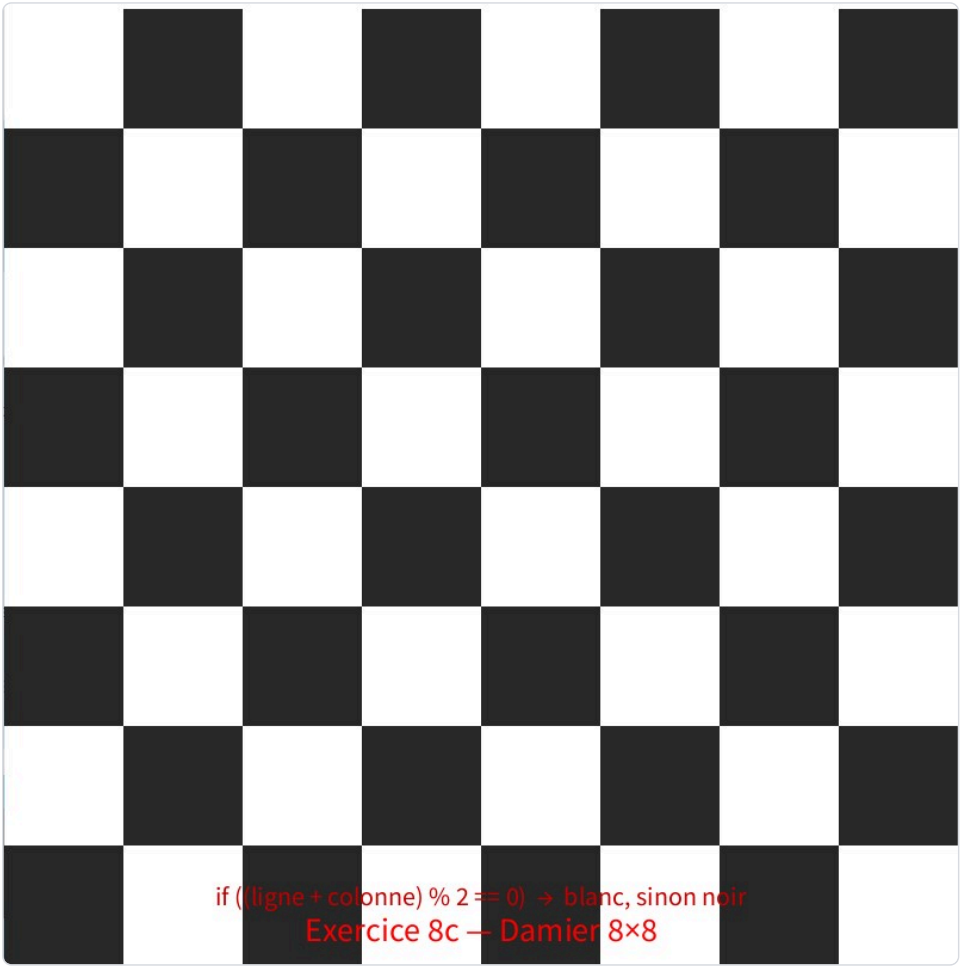
Contraintes

- Canevas de `createCanvas(400, 400)` (pour que les 4×4 carrés de 100px remplissent tout l'écran)
- Pour l'étape 2 : la taille des cases se calcule avec `width / nbCases`

- Pour l'étape 3 : utilise une condition `if` pour choisir la couleur

Ce que tu devrais obtenir

Étape 1 → une grille de carrés identiques. Étape 2 → la même grille, mais en 4 lignes de code. Étape 3 → un échiquier noir et blanc.



Indices

💡 **Indice 1 (étape 2)** — La position X d'une case est `colonne * taille` et la position Y est `ligne * taille`, avec `taille = width / nbCases`.

💡 **Indice 2 (étape 3)** — L'astuce du damier repose sur une observation mathématique : si la **somme** du numéro de ligne et du numéro de colonne est **paire**, la case est blanche ; si elle est **impaire**, la case est noire.

💡 **Indice 3 (étape 3)** — Pour savoir si un nombre est pair, on utilise l'opérateur **modulo** `%` (le reste de la division). `(ligne + colonne) % 2` donne `0` si la somme est paire, `1` si elle est impaire :

```javascript
if ((ligne + colonne) % 2 === 0) {
  fill(255);  // Blanc
} else {
  fill(0);    // Noir
}
```

Résumé

Concept	Syntaxe	Exemple
Boucle for	`for (let i = 0; i < n; i++)`	Répéter n fois
Compteur	`i`	Vaut 0, 1, 2, ... n-1
Incrémenter	`i++`	Raccourci pour `i = i + 1`
Position calculée	`debut + i * espacement`	`30 + i * 50`
Couleur calculée	`i * facteur`	`fill(i * 25, 0, 0)`
Boucles imbriquées	2 `for` = grille 2D	`for ligne { for colonne }`
Modulo	`a % b`	Reste de la division → pair/impair

JavaScript classique	p5.js
`for (let i = 0; i < 10; i++)`	`for (let i = 0; i < 10; i++)` (identique !)
`while (condition) { }`	`while (condition) { }` (identique !)

Les points clés à retenir :

- Une boucle **évite le copier-coller** et rend ton code adaptable
- Le compteur `i` est une **variable gratuite** que tu peux utiliser dans tes calculs
- Deux boucles imbriquées = une **grille** (lignes × colonnes)
- Mets le nombre de répétitions dans une **variable** pour pouvoir le changer facilement

Dans le prochain chapitre, on va découvrir les **fonctions** — le moyen de créer tes propres blocs réutilisables pour organiser et simplifier ton code !

1.7 Art génératif

Définition

L'art génératif, c'est concevoir un **système** (règles + algorithmes + hasard contrôlé) qui **produit** l'œuvre. L'artiste définit les paramètres; le programme explore l'«espace des possibles» et génère des variantes uniques (souvent à partir d'une *seed* aléatoire mais **reproductible**).

Origines

- **Années 1960–70** : premiers "algoristes" et **traceurs/plotters** (ex.: Vera Molnár).
- **Années 2000** : démocratisation via **Processing** (Reas/Fry) puis p5.js, openFrameworks, etc.
- **Aujourd'hui** : JavaScript dans le navigateur, shaders/WebGL, IA générative.

Le tournant blockchain

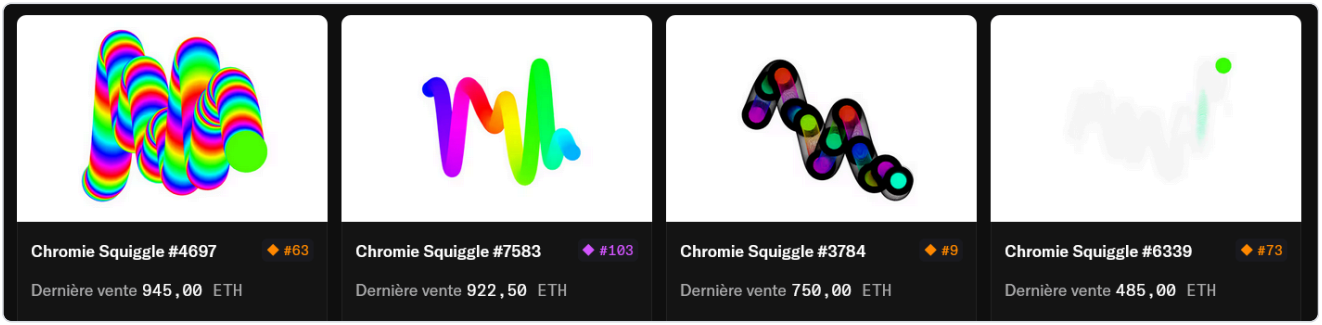
La blockchain apporte la **preuve d'unicité** et l'**exécution publique** d'algorithmes :

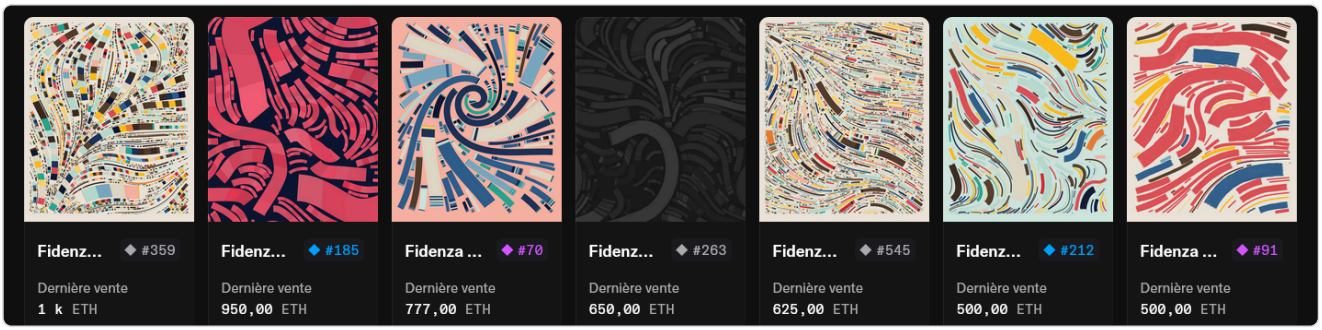
- **CryptoPunks (2017)** : 10k portraits générés par combinaison de traits, icône de la culture NFT.
- **Long-form** : une **seule** œuvre-algorithme génère **toute une série** d'outputs au mint, sans sélection manuelle (exécution autonome, paramètres variés, édition limitée).

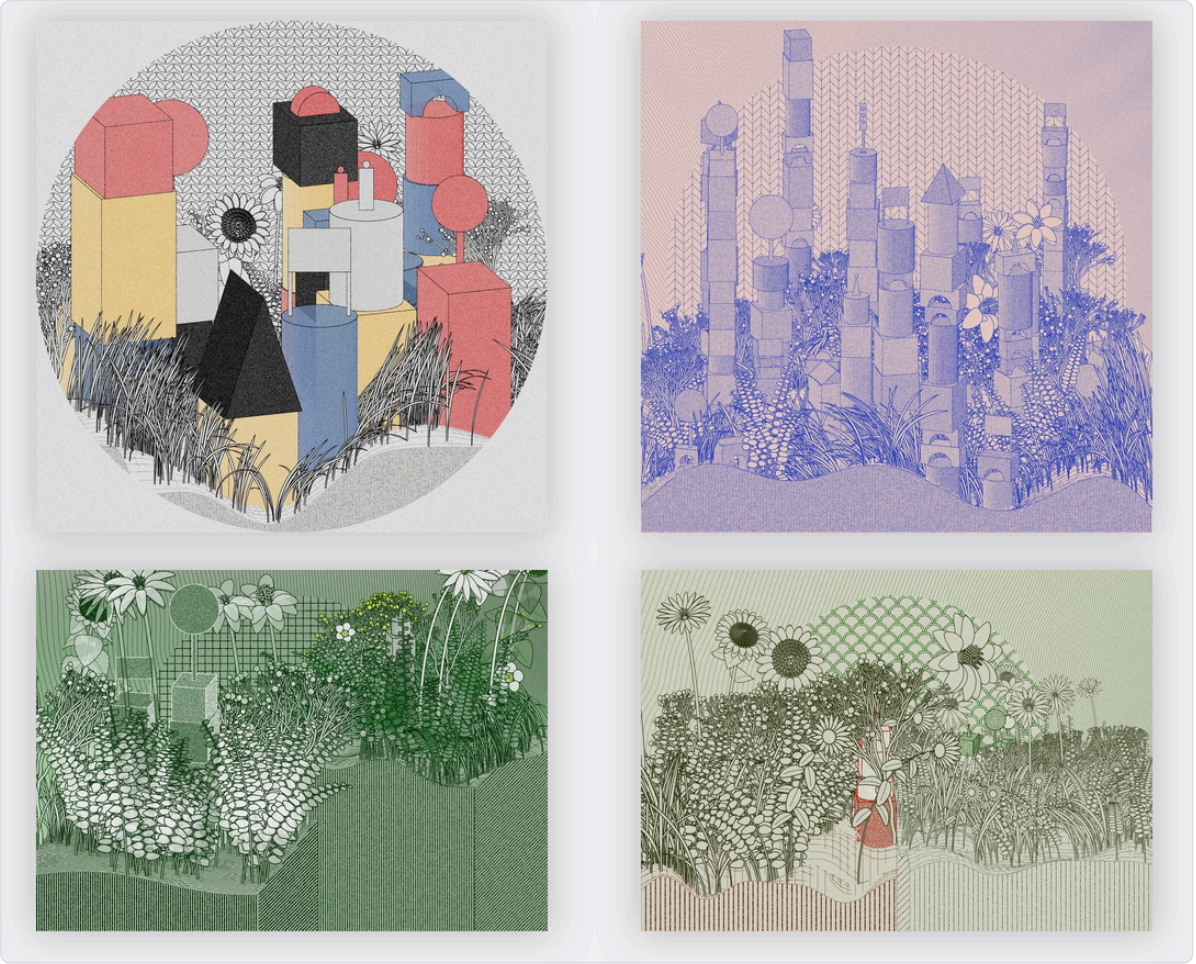


Références clés

- **Chromie Squiggles** (Snowfro, Art Blocks) : lignes ondulées, palette/épaisseur contrôlées par la seed, manifeste du long-form.
- **Fidenza** (Tyler Hobbs) : champs de formes “brossées”, variation riche via règles de composition/couleur/texture.
- **Zancan** : paysages génératifs inspirés du dessin technique (ex.: *Garden, Monoliths*), souvent sur Tezos/fxhash.







Pourquoi c'est utile pour vous

- **Penser en systèmes** (contraintes → styles cohérents).
- **Paramétrer la variation** (design, palettes, grilles).
- **Reproductibilité** (même seed → même visuel).
- **Perf web** : canvas/SVG, shaders, génération à la volée.

Projet — Œuvre générative personnelle (p5.js)

Objectif

Créer une **œuvre d'art génératif personnelle** avec p5.js. À **chaque lancement**, le rendu doit être **différent**. L'œuvre peut être **statique** ou **animée**.

Contraintes

- Réalisation **dans l'éditeur web p5.js** (editor.p5js.org).
- **GPT/IA explicitement encouragés** pour vous aider à atteindre un résultat **unique**.

Pistes de recherche

- **Rectangle packing** (remplissage non chevauchant, densité/échelle variables).

- **Palettes de couleurs** (HSL, harmonies, tirages pondérés).
- **Flow fields** (bruit/ `noise()` pour guider des particules ou des traits).
- **Grilles & perturbations** (symétries, jitter, subdivisions).
- **Textures procédurales** (hachures, pointillisme, “brosses” algorithmiques).
- **Masques & formes composées** (clips, superpositions, transparences).

Remise & présentation

- **À envoyer par e-mail** (le **code du projet**) à **thibault.six@ifosup.wavre.be**
- **Date limite : lundi 15 septembre 2025**
- **Présentation en classe** : démonstration en direct de votre œuvre.

1.8 Intégration de p5.js avec Vite

Ce chapitre détaille la procédure pour démarrer un projet en utilisant Vite et intégrer la bibliothèque p5.js, facilitant la création d'éléments graphiques interactifs dans une application web. Vite assure une expérience de développement optimisée, tandis que p5.js ouvre un champ de possibilités pour le dessin, l'animation et l'interaction visuelle.

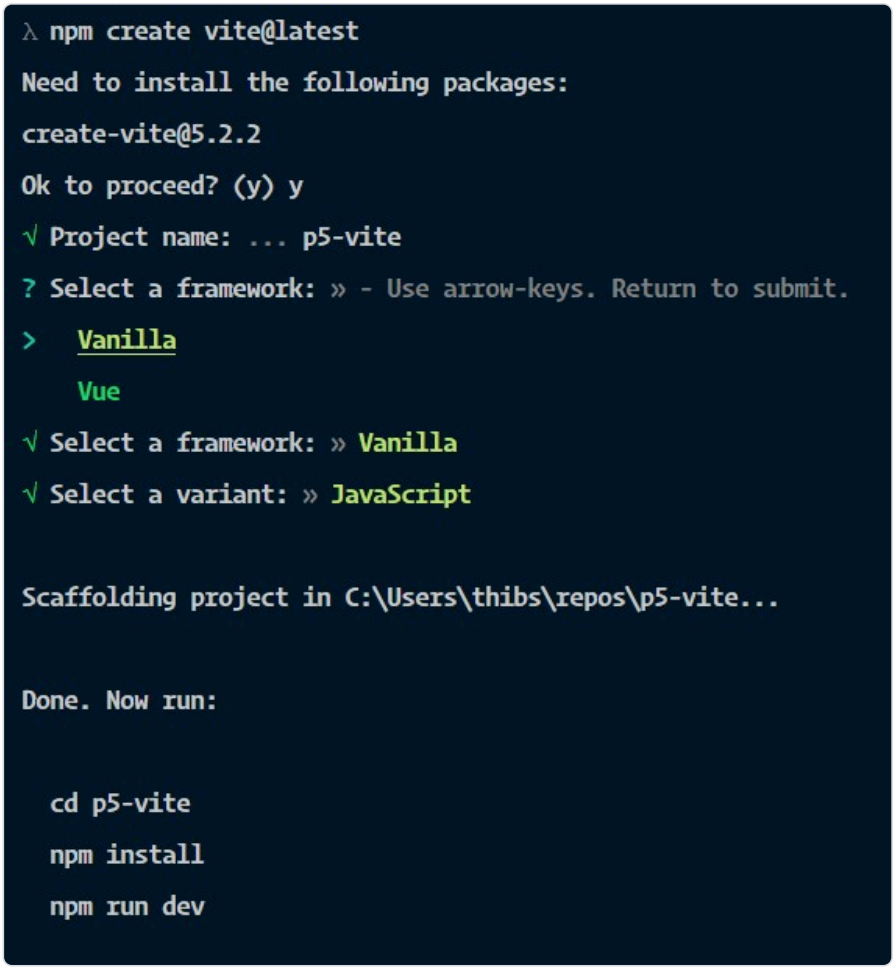
Initialisation avec Vite

Création du projet

Pour initier, lancez la commande suivante dans un terminal :

```bash
npm create vite@latest
```

Suivez les instructions pour sélectionner un modèle de projet et nommer votre projet. Optez pour un projet JavaScript simple pour ce tutoriel.



Ajout de p5.js

Dirigez-vous dans le dossier de votre projet et installez p5.js :

```bash
npm i p5
```

Cela intègre p5.js à votre projet, vous permettant d'exploiter ses fonctionnalités pour des créations graphiques.

Configuration du Projet

Nettoyage initial

Avant de coder, éliminez les fichiers superflus (`counter.js` et `javascript.svg`) pour clarifier l'espace de travail.

Mise à jour de `main.js`

Adaptez le contenu de `main.js` comme suit :

```javascript
import "./style.css";
import p5 from "p5";

new p5(p => {
  p.setup = () => {
    p.createCanvas(800, 800);
    p.background(0);
  };

  p.draw = () => {
    p.fill(255);
    p.ellipse(p.mouseX, p.mouseY, 50, 50);
  };
}, document.querySelector("#app"));
```

Ce code initialise un canevas via p5.js et y dessine un cercle blanc suivant le curseur de la souris.

Réinitialisation de `style.css`

Videz `style.css` pour prévenir tout conflit de styles. Ce fichier pourra être utilisé ultérieurement pour personnaliser les styles de votre projet.

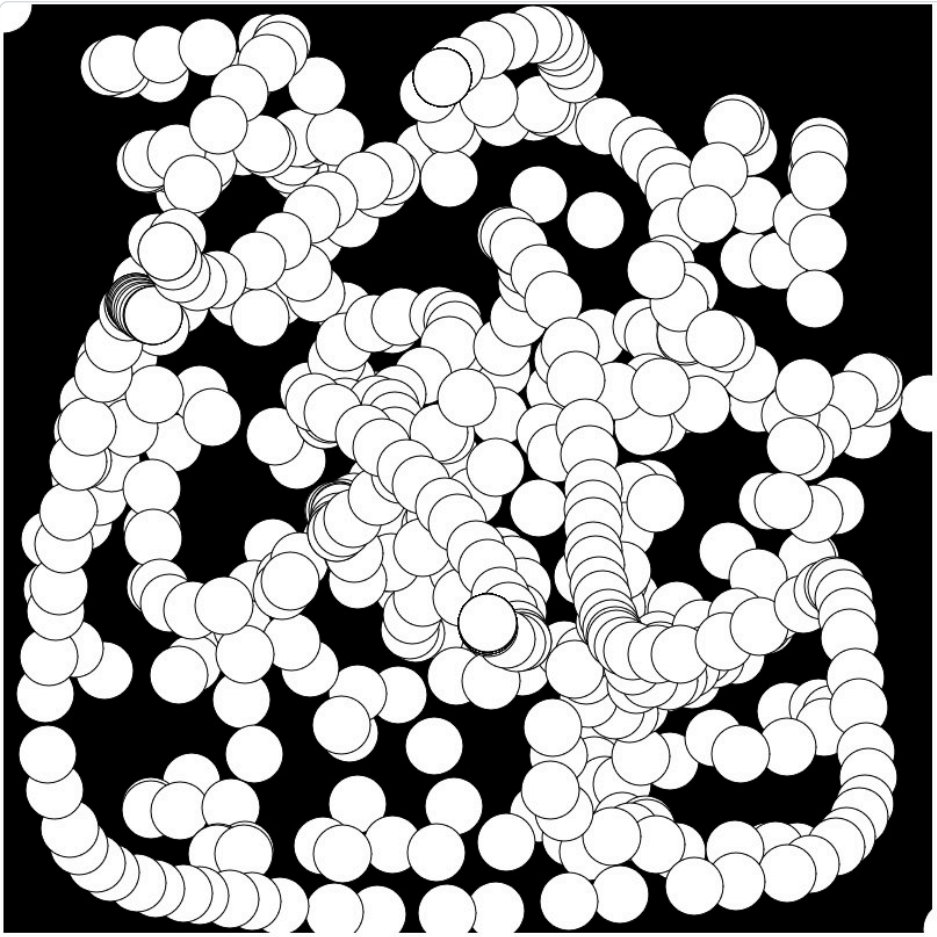
Visualisation du projet

Pour observer le résultat, activez le serveur de développement :

```bash
npm run dev
```

Rendez-vous à l'adresse indiquée dans votre navigateur pour visualiser le canevas p5.js en action, illustré par un cercle blanc suivant le mouvement de la souris sur un fond noir.

Vous avez maintenant intégré avec succès p5.js dans un projet Vite, ouvrant la voie à la création de visuels interactifs enrichissants pour vos applications web.



CHAPITRE 2

JS

2.1 Introduction au Terminal (CLI) de Laragon

> ***Objectif*** *: Apprendre à naviguer dans le système de fichiers Windows via le terminal, parce que cliquer partout comme un raton laveur hyperactif, c'est 2005.*

Pourquoi Utiliser le Terminal ? 🤔

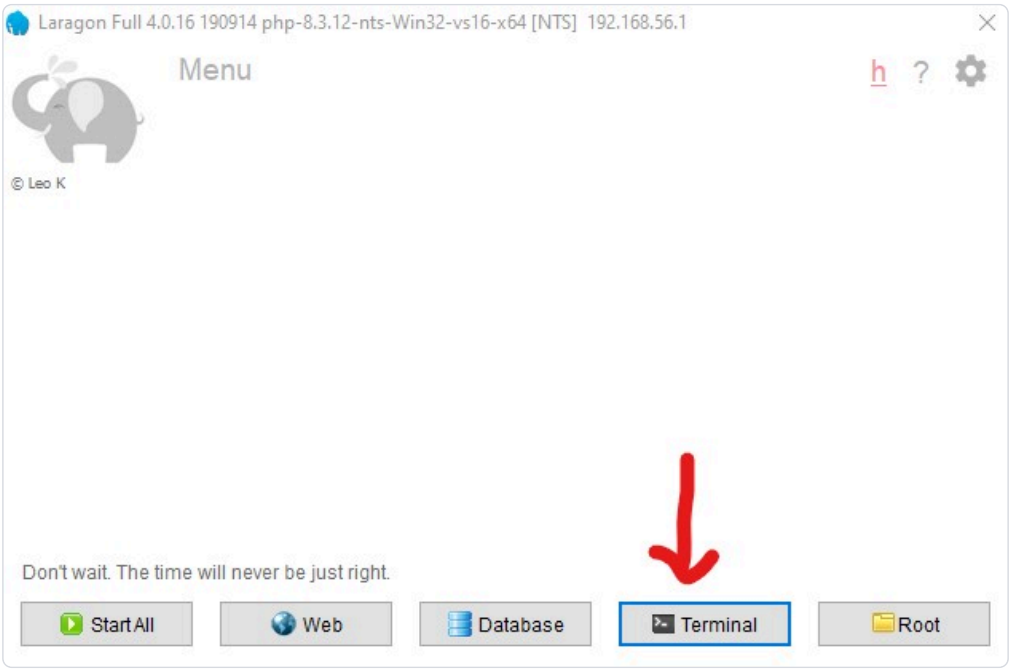
Le terminal (ou CLI - Command Line Interface) permet de :

- ✅ Naviguer **ultra rapidement** entre les dossiers
- ✅ Créer des fichiers/dossiers sans ouvrir 12 fenêtres
- ✅ Exécuter des commandes de dev (npm, git, composer...)
- ✅ Avoir l'air cool devant tes potes (argument principal tbh)

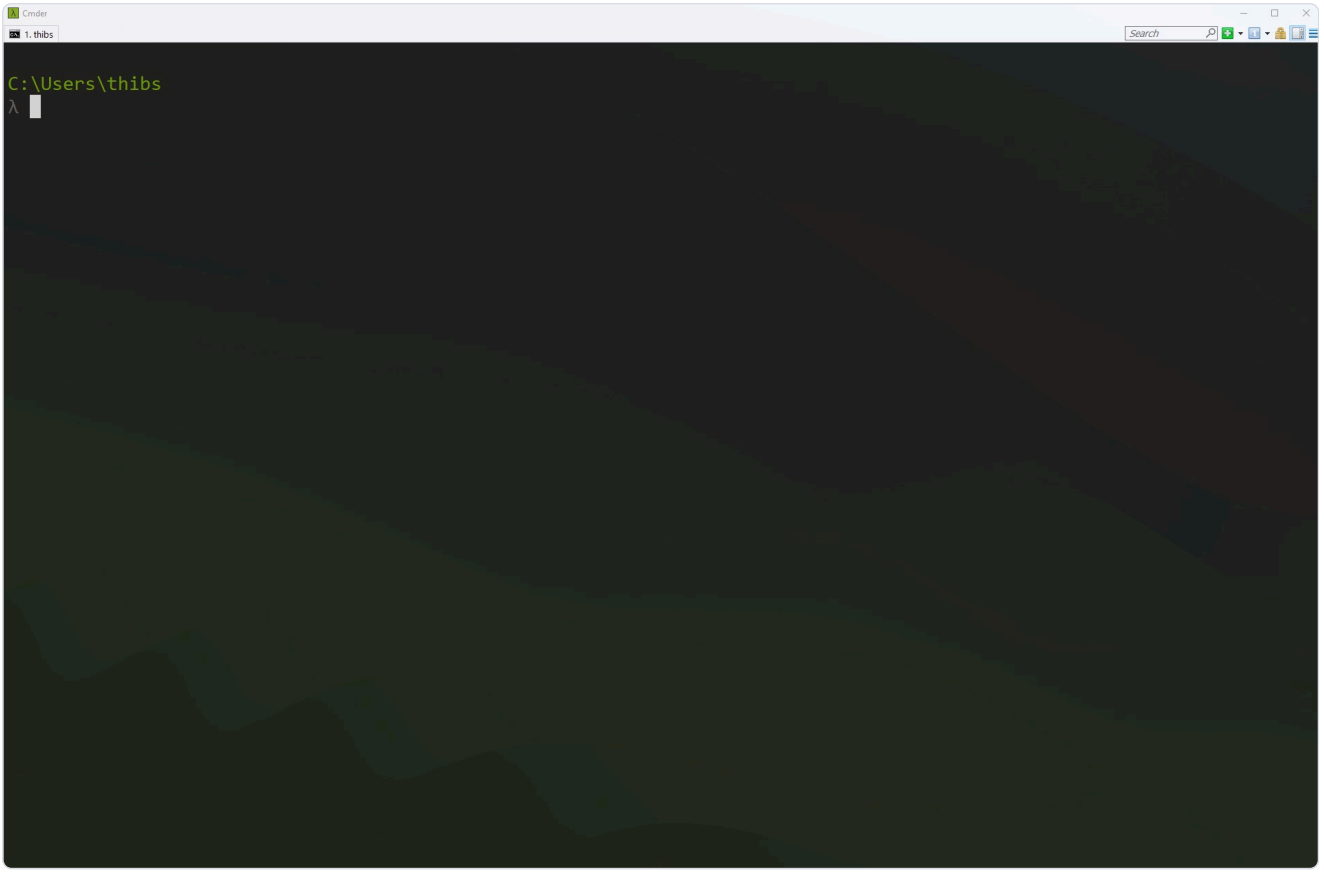
En gros : Si tu veux être efficace et pas perdre 3h par jour à chercher des dossiers, le terminal c'est **LA** compétence de base.

Ouvrir le Terminal de Laragon

Dans Laragon, clique sur le bouton **Terminal** en bas à droite de l'interface.



Une fenêtre noire (ou verte si t'as Cmder) s'ouvre. Bienvenue dans la matrice. 😎



Par défaut, tu es dans ton dossier utilisateur Windows :

```
C:\Users\thibs
λ
```

Le petit symbole `λ` (lambda) indique que le terminal est prêt à recevoir tes commandes.

⚠️ AVANT DE COMMENCER : Les Touches Spéciales

SUPER IMPORTANT : Dans ce cours, tu verras des notations comme `[ENTER]` et `[TAB]`.

- `[ENTER]` = Appuie sur la touche **Entrée** de ton clavier pour valider/exécuter la commande
- `[TAB]` = Appuie sur la touche **TAB** (↹, au-dessus de Caps Lock) pour l'autocomplétion

TU NE TAPES PAS LE MOT "ENTER" OU "TAB", tu appuies physiquement sur ces touches ! ⌨️

Exemple :

```bash
# Ce que tu vois dans le cours :
cd Documents[ENTER]

# Ce que tu fais VRAIMENT :
# 1. Tu tapes : cd Documents
# 2. Tu appuies sur la touche Entrée
```

Maintenant qu'on est clairs, let's go ! 🚀

Commandes de Base pour Naviguer 🧭

1. `cd` - Changer de Dossier (Change Directory)

Syntaxe :

```bash
cd nom_du_dossier
```

Exemples :

```bash
# Aller dans le dossier Documents
cd Documents[ENTER]

# Revenir au dossier parent
cd ..[ENTER]

# Aller directement à la racine C:\
cd \[ENTER]

# Aller dans le dossier Laragon
cd \laragon\www\[ENTER]
```

Rappel : `[ENTER]` = tu appuies sur Entrée pour exécuter la commande !

⚠️ **ATTENTION** : Sur Windows, les noms de dossiers sont **sensibles à la casse** dans certains terminaux.

- `cd documents` ≠ `cd Documents`
- Utilise **TAB** pour l'autocomplétion (voir plus bas)

---

2. `ls` - Lister les Fichiers/Dossiers (List)

Syntaxe :

```bash
ls
```

Exemple :

```bash
C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/  Pictures/  repos/
```

Affiche tous les fichiers et dossiers du répertoire actuel.

Variantes utiles :

```bash
# Afficher les détails (taille, date)
ls -l

# Afficher les fichiers cachés aussi
ls -a
```

---

3. `mkdir` - Créer un Dossier (Make Directory)

Syntaxe :

```bash
mkdir nom_du_dossier
```

Exemple :

```bash
# Créer un dossier "repos"
mkdir repos

# Créer plusieurs dossiers d'un coup
mkdir projets tests brouillons
```

---

4. `touch` - Créer un Fichier Vide

Syntaxe :

```bash
touch nom_fichier.extension
```

Exemple :

```bash
# Créer un fichier index.html
touch index.html

# Créer plusieurs fichiers
touch style.css script.js
```

Note : `touch` n'existe pas nativement sur Windows, mais fonctionne dans Cmder (le terminal de Laragon). Sinon, utilise :

```bash
echo. > fichier.txt
```

---

5. **ENTER** - Valider une Commande 🎯

IMPORTANT : Après avoir tapé ta commande, appuie sur **ENTER (Entrée)** pour l'exécuter !

Sans ça, ton terminal va juste te regarder comme un poisson rouge qui attend. 🐟

Exemple :

```bash
C:\Users\thibs
λ cd Documents[ENTER]
# La commande s'exécute
C:\Users\thibs\Documents
λ
```

Note : Dans les exemples de ce cours, on écrit `[ENTER]` pour te rappeler d'appuyer sur Entrée, mais dans la vraie vie tu l'écris pas, tu appuies juste sur la touche. Logique, non ? 😉

---

6. TAB - Autocomplétion Magique ✨

LE TRUC LE PLUS IMPORTANT DE TA VIE : Appuie sur la touche **TAB** (au-dessus de Caps Lock) pour autocompléter les noms de fichiers/dossiers.

C'est quoi [TAB] ?

- `[TAB]` dans les exemples = la touche TAB de ton clavier (↹)
- Tu NE TAPES PAS le mot "TAB", tu APPUIES sur la touche
- C'est comme `[ENTER]`, c'est une TOUCHE, pas du texte

Exemple concret :

```bash
C:\Users\thibs
λ cd Do        ← Tu tapes juste "Do"
λ cd Do↹       ← Tu appuies sur TAB
# Devient automatiquement :
C:\Users\thibs
λ cd Documents\
```

Si plusieurs options existent, appuie sur **TAB** 2 fois pour voir la liste :

```bash
C:\Users\thibs
λ cd \Users\↹↹
Default\  Public\   thibs\
```

Règle d'or : TOUJOURS utiliser TAB. Taper en entier c'est pour les masochistes. 🥲

Pourquoi c'est VITAL ?

- ✅ Gain de temps monstrueux
- ✅ Évite les fautes de frappe
- ✅ Gère automatiquement les majuscules/minuscules
- ✅ Te montre ce qui existe vraiment

Le Système de Fichiers Windows (Rappel Rapide)

Structure Typique

```
C:\                         (Racine du disque C:\)
|-- Users\                  (Dossiers utilisateurs)
|   |-- thibs\              (Ton dossier perso)
|   |   |-- Desktop\
|   |   |-- Documents\
|   |   |-- Downloads\
|   |   +-- repos\          (<-- On va créer ça)
|   +-- Public\
|-- Program Files\
+-- laragon\                (<-- Tes projets web ici)
    +-- www\
        |-- projet-1\
        +-- projet-2\
```

Chemins Absolus vs Relatifs

Chemin Absolu (depuis la racine) :

```bash
cd C:\laragon\www\mon-projet
```

Chemin Relatif (depuis où tu es) :

```bash
# Si tu es dans C:\laragon\
cd www\mon-projet
```

Raccourcis :

- `.` = Dossier actuel
- `..` = Dossier parent
- `~` = Dossier utilisateur (C:\Users\thibs)

Exercices Pratiques 💪

RAPPEL : N'oublie JAMAIS d'appuyer sur **ENTER** après chaque commande ! Sinon rien ne se passe. 😄

Exercice 1 : Créer le Dossier `repos`

Objectif : Créer un dossier `repos` dans ton dossier utilisateur.

Étapes :

1. Ouvre le terminal Laragon
2. Vérifie que tu es dans `C:\Users\[ton-nom]\`
3. Liste les dossiers existants avec `ls`
4. Crée le dossier `repos` avec `mkdir repos`
5. Vérifie qu'il existe avec `ls`

Solution :

```bash
C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/

C:\Users\thibs
λ mkdir repos

C:\Users\thibs
λ ls
Desktop/  Documents/  Downloads/  repos/
```

Exercice 2 : Naviguer jusqu'à Laragon

Objectif : Aller dans le dossier `C:\laragon\www\` en utilisant **TAB** pour l'autocomplétion.

Étapes :

1. Depuis n'importe où, tape `cd \la` puis appuie sur **TAB** → Doit compléter en `cd \laragon\`
2. Continue avec `www\` en tapant `www` puis **TAB**
3. Appuie sur **ENTER**
4. Vérifie avec `ls` que tu vois tes projets web

Solution :

```bash
C:\Users\thibs
λ cd \la[TAB]
# Autocomplétion vers :
λ cd \laragon\

λ cd \laragon\www[TAB]
# Autocomplétion vers :
λ cd \laragon\www\
# Puis tu appuies sur ENTER

C:\laragon\www
λ ls
projet-1/  projet-2/
```

Note : `[TAB]` = appuyer sur la touche TAB, pas taper le mot "TAB". C'est une touche physique de ton clavier ! ⌨️

Exercice 3 : Créer une Structure de Projet

Objectif : Créer un nouveau projet web avec ses fichiers de base.

Étapes :

1. Va dans `C:\laragon\www\`
2. Crée un dossier `mon-premier-site`
3. Entre dans ce dossier
4. Crée les fichiers `index.html`, `style.css`, `script.js`
5. Liste les fichiers pour vérifier

Exercice 4 : Remonter dans l'Arborescence

Objectif : Naviguer entre les dossiers avec `..` (parent).

Situation : Tu es dans `C:\laragon\www\mon-projet\assets\images\`

Questions :

1. Comment revenir dans `assets\` ?
2. Comment revenir directement dans `www\` ?
3. Comment aller à la racine `C:\` ?

Exercice 5 : Challenge Final 🏆

Objectif : Créer cette structure de dossiers dans `C:\Users\[ton-nom]\repos\` :

```
repos/
|-- html-css/
|   |-- tp1/
|   +-- tp2/
|-- javascript/
|   +-- exercices/
+-- projets/
```

Contraintes :

- Utilise **uniquement** le terminal (pas l'explorateur Windows)
- Utilise **TAB** pour l'autocomplétion
- Vérifie chaque création avec `ls`

Récap' des Commandes Essentielles 📋

Commande	Action	Exemple
`cd dossier`	Aller dans un dossier	`cd Documents[ENTER]`
`cd ..`	Remonter d'un niveau	`cd ..[ENTER]`
`cd \`	Aller à la racine	`cd \[ENTER]`
`ls`	Lister fichiers/dossiers	`ls[ENTER]`
`mkdir nom`	Créer un dossier	`mkdir repos[ENTER]`
`touch fichier`	Créer un fichier	`touch index.html[ENTER]`
TAB	Autocomplétion	*Touche physique*
ENTER	Valider/Exécuter	*Touche physique*

Erreurs Fréquentes à Éviter ⚠️

1. Oublier les Majuscules

```bash
# ❌ FAUX (si le dossier s'appelle "Documents")
cd documents

# ✅ CORRECT
cd Documents
```

Solution : Utilise TAB pour éviter ce problème.

---

2. Confondre `/` et `\`

- **Linux/Mac** : `/` (slash)
- **Windows** : `\` (backslash)

```bash
# ❌ FAUX sur Windows
cd C:/laragon/www/

# ✅ CORRECT
cd C:\laragon\www\
```

Note : Sur Cmder (Laragon), les deux fonctionnent souvent, mais respecte la convention Windows.

---

3. Taper Tout à la Main

Non. Juste non. 🤦

Utilise **TAB** ou tu vas perdre 10 ans de ta vie à corriger des typos.

Prochaines Étapes 🚀

Maintenant que tu maîtrises la navigation de base, tu es prêt pour :

- Installer Node.js et npm
- Créer des projets Vue/Nuxt
- Lancer des serveurs de dev

Spoiler : Toutes ces commandes se font... dans le terminal. ðŸ˜

Ressources Complémentaires 📚

- Cheat Sheet Bash - Antisèche complète
- Cmder Documentation - Le terminal de Laragon
- Explainshell - Explique les commandes complexes

2.2 Node.js et npm : Les Outils du Web Designer Moderne

> ***En une phrase** : Node.js te permet d'utiliser des outils de dev web (Vite, Tailwind, etc.) et npm gère l'installation de ces outils. Point.* 🎯

Pourquoi Tu Dois Connaître Ça ? 🤔

Situation classique de designer en 2025 :

❌ **Avant (2010)** : Tu codes en HTML/CSS direct, tu refresh le navigateur, c'est bon.

✅ **Maintenant** : Tu veux utiliser :

- **Tailwind CSS** → Besoin de npm
- **Vue.js / Nuxt** → Besoin de npm
- **Vite** (serveur de dev ultra rapide) → Besoin de npm
- **Autoprefixer** (compatibilité navigateurs) → Besoin de npm

Bref : Si tu veux bosser avec des outils modernes (et pas coder comme en 2005), tu DOIS avoir Node.js et npm. 💀

C'est Quoi Node.js ? 🟢

La version simple :

- Node.js = JavaScript qui tourne **en dehors du navigateur** (sur ton PC directement)
- Ça permet d'exécuter des outils de développement web
- C'est comme le moteur qui fait tourner toutes les apps de dev

Analogie :

- Photoshop = outil pour designer
- Node.js = outil pour développer/builder des sites web

Tu ne vas PAS coder en Node.js (t'es pas dev backend), mais tous tes outils de design/intégration web **ont besoin de Node.js pour fonctionner**.

C'est Quoi npm ? 📦

npm = **N**ode **P**ackage **M**anager

En gros : Le "store d'applications" pour les outils de développement web.

Exemples concrets :

```bash
# Installer Tailwind CSS
npm install tailwindcss

# Installer Vue.js
npm install vue

# Créer un projet Nuxt
npm create nuxt@latest
```

Analogie :

- npm = App Store / Google Play
- Les "packages" = les apps que tu télécharges

Note : npm est installé automatiquement avec Node.js. T'as pas besoin de l'installer séparément. 👍

Vérifier Si Node.js est Installé 🔍

Ouvre le terminal Laragon et tape :

```bash
node --version
```

Si t'as un numéro qui s'affiche (genre `v20.11.0`), c'est bon, Node.js est installé ! ✅

Vérifie npm aussi :

```bash
npm --version
```

Exemple de résultat attendu :

```bash
C:\Users\thibs
λ node --version
v20.11.0

C:\Users\thibs
λ npm --version
10.2.4
```

Si tu vois des numéros → Everything is fine. 😎

Si tu vois "command not found" ou une erreur → Faut installer/mettre à jour Node.js (voir section suivante).

Mettre à Jour Node.js dans Laragon 🔄

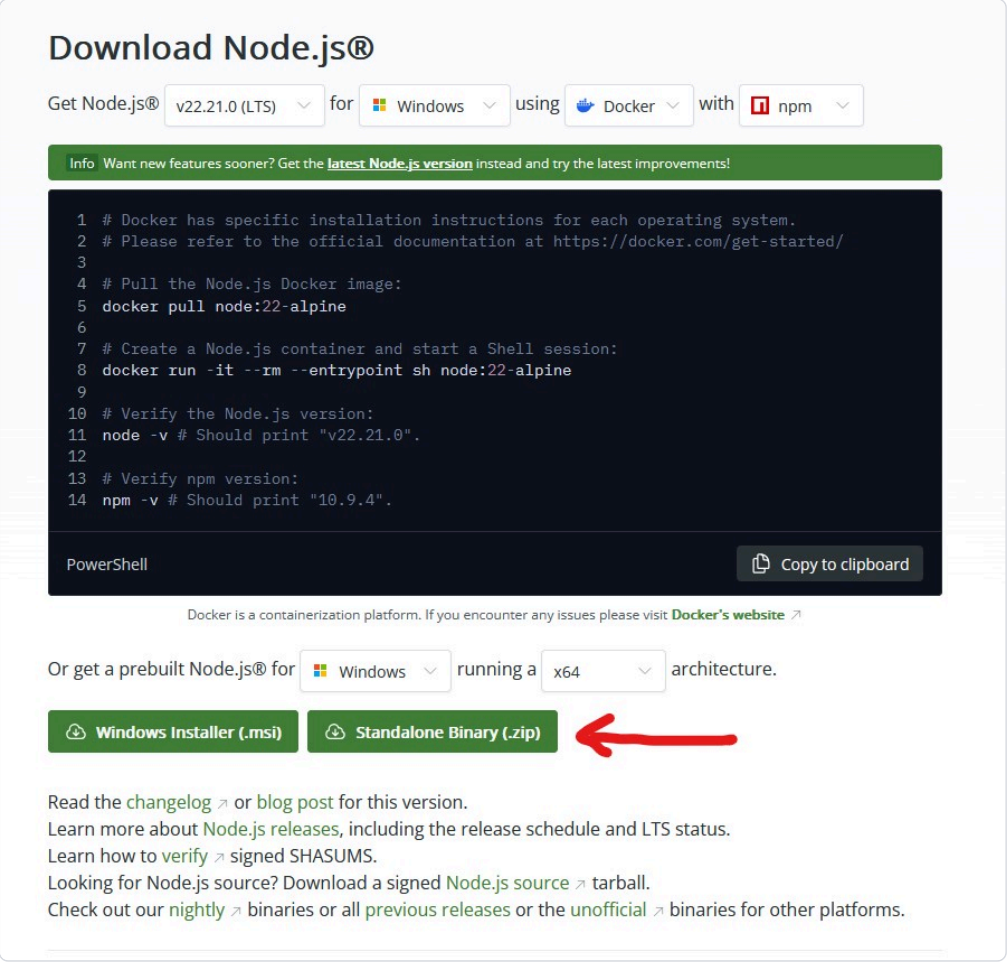
Laragon vient avec une version de Node.js, mais elle peut être ancienne. Voici comment installer la dernière version.

Étape 1 : Télécharger Node.js

1. Va sur le site officiel : https://nodejs.org
2. **Télécharge la version LTS** (Long Term Support = stable et recommandée)
3. **ATTENTION** : Choisis **"Standalone Binary (.zip)"**, PAS l'installeur Windows !

Où trouver le Standalone Binary ?

- Sur la page de téléchargement
- Télécharge la version **Windows x64** (.zip)



Étape 2 : Décompresser l'Archive

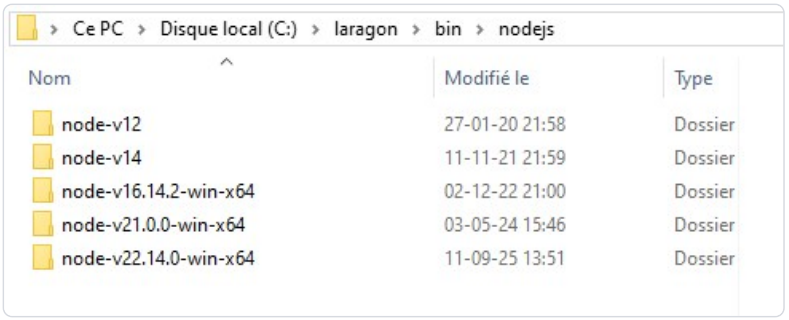
1. Une fois le fichier `.zip` téléchargé, **extrais-le**
2. Tu devrais avoir un dossier avec dedans : `node.exe`, `npm`, etc.

Étape 3 : Placer dans Laragon

1. Ouvre le dossier d'installation de Laragon (par défaut : `C:\laragon`)
2. Va dans le dossier `bin\nodejs\`
3. **Colle ton dossier Node.js décompressé ici**

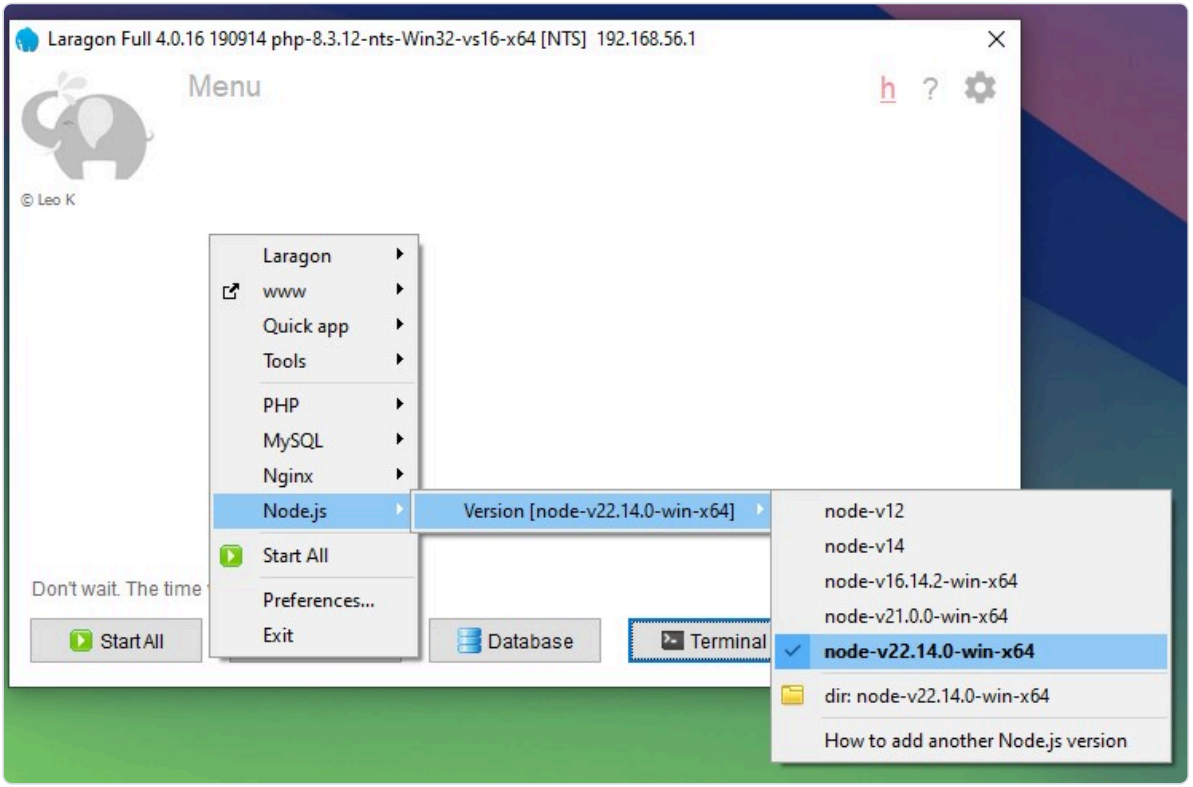
Structure finale :

```
C:\laragon\
+-- bin\
    +-- nodejs\
        |-- nodejs-20.11.0\    (ancienne version)
        +-- nodejs-22.0.0\     (nouvelle version)
            |-- node.exe
            |-- npm
            +-- ...
```



Étape 4 : Sélectionner la Version dans Laragon

1. Ouvre Laragon
2. Clique sur **Menu** (en haut à droite) ou clic droit sur l'icône Laragon
3. Va dans **Node.js** → **Version**
4. Sélectionne ta nouvelle version (genre `nodejs-22.21.0`)
5. Laragon va redémarrer automatiquement



Étape 5 : Vérifier que Ça Marche

Ouvre le terminal Laragon et vérifie :

```bash
node --version
```

Tu devrais voir ta nouvelle version ! 🎉

```bash
C:\Users\thibs
λ node --version
v22.21.0
```

Si ça affiche toujours l'ancienne version :

- Rouvre un nouveau terminal

Les Commandes npm de Base (Que Tu Vas Utiliser Souvent) ⚡

Tu vas pas devenir expert npm, mais voici les 5 commandes que tu DOIS connaître :

1. `npm install` - Installer les Dépendances

Quand ? Quand tu télécharges un projet existant (GitHub, GitLab, etc.)

```bash
# Installe tous les packages listés dans package.json
npm install
```

Ce que ça fait : Télécharge toutes les dépendances du projet dans le dossier `node_modules/`.

2. `npm install <package>` - Installer UN Package

Quand ? Quand tu veux ajouter un outil à ton projet

```bash
# Installer Tailwind CSS
npm install tailwindcss

# Installer plusieurs packages d'un coup
npm install tailwindcss autoprefixer postcss
```

3. `npm run dev` - Lancer le Serveur de Développement

Quand ? Quand tu veux bosser sur ton projet (hot reload, etc.)

```bash
npm run dev
```

Ce que ça fait : Lance ton projet en mode développement (généralement sur `http://localhost:5173` ou similaire)

4. `npm run build` - Builder pour la Production

Quand ? Quand tu veux déployer ton site en ligne

```bash
npm run build
```

Ce que ça fait : Crée une version optimisée et compressée de ton site dans un dossier `dist/` ou `build/`.

5. `npm create <template>` - Créer un Nouveau Projet

Quand ? Quand tu démarres un nouveau projet from scratch

```bash
# Créer un projet Vue
npm create vue@latest
```

Ce que ça fait : Lance un assistant interactif pour générer la structure de base du projet.

Le Fichier `package.json` (Juste pour Comprendre) 📄

Quand tu crées un projet avec npm, tu vas trouver un fichier `package.json` à la racine.

C'est quoi ? Le "carnet de bord" de ton projet qui liste :

- Les dépendances (packages installés)
- Les scripts disponibles (`dev` , `build` , etc.)
- Les infos du projet (nom, version, etc.)

Exemple simplifié :

```json
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "dependencies": {
    "vue": "^3.4.0",
    "tailwindcss": "^3.4.0"
  }
}
```

Ce que tu dois retenir :

- `scripts` = les commandes que tu peux lancer avec `npm run`
- `dependencies` = la liste des packages installés

Tu ne vas PAS éditer ce fichier manuellement (sauf cas rares). npm le gère pour toi. 👌

Le Dossier `node_modules/` (À Ignorer) 📁

Quand tu fais `npm install` , un gros dossier `node_modules/` apparaît.

C'est quoi ? Tous les packages téléchargés (peut peser 200-500 MB facile).

Ce que tu dois faire avec : **RIEN.** 🚫

- ❌ Ne le commite PAS sur Git
- ❌ Ne l'édite PAS manuellement
- ❌ Ne le partage PAS (trop gros)

Pourquoi ? Parce que n'importe qui peut recréer ce dossier en faisant `npm install` avec le fichier `package.json` .

Règle d'or : Si tu partages ton projet, tu partages `package.json` , PAS `node_modules/` . 📦

Récap' : Ce Que Tu Dois Retenir 🎯

Concept	C'est Quoi ?	Pourquoi Tu T'en Fous (ou Pas)
Node.js	Moteur JavaScript	Tu le lances jamais directement, mais t'en as besoin pour npm
npm	Gestionnaire de packages	Tu l'utilises TOUT LE TEMPS (install, run, etc.)
package.json	Fichier de config	npm le gère, tu le lis juste de temps en temps
node_modules/	Dossier des packages	Tu l'ignores complètement
npm install	Installe les dépendances	Commande #1 quand tu récupères un projet
npm run dev	Lance le serveur de dev	Commande #2 pour bosser sur ton projet

Prochaines Étapes 🚀

Maintenant que Node.js et npm sont opérationnels, tu es prêt pour :

- ✅ Créer ton premier projet Vue.js
- ✅ Installer Tailwind CSS
- ✅ Lancer un serveur de développement avec Vite
- ✅ Utiliser des outils modernes comme un pro

Let's gooo ! 🔥

Ressources Utiles 📚

- Site officiel Node.js - Télécharger les nouvelles versions
- Documentation npm - Si tu veux creuser
- npm Package Search - Trouver des packages

Note : Tu vas pas lire la doc npm de A à Z (personne le fait). Tu cherches juste ce dont t'as besoin quand t'en as besoin. 😎

2.3 Exercices complémentaires

1. Changement de couleur du fond

Dans cet exercice, vous devez créer un bouton qui permet de changer la couleur de fond d'une page HTML. Lorsqu'on clique sur le bouton, la couleur du fond doit passer du bleu au rouge. Si on re-clique sur le bouton, la couleur du fond doit revenir au rouge.

2. Bouton "Like/Dislike"

Dans cet exercice, vous allez créer un bouton "Like/Dislike" en utilisant HTML, CSS et JavaScript. Ce bouton permettra aux utilisateurs d'indiquer s'ils aiment ou n'aiment pas un contenu spécifique.



3. Compteur de clics

Dans cet exercice, vous allez créer un compteur de clics en utilisant HTML, CSS et JavaScript. Ce compteur permettra de suivre le nombre de fois où l'utilisateur clique sur un bouton spécifique.

4. Changement de couleur de fond avec un sélecteur de couleur

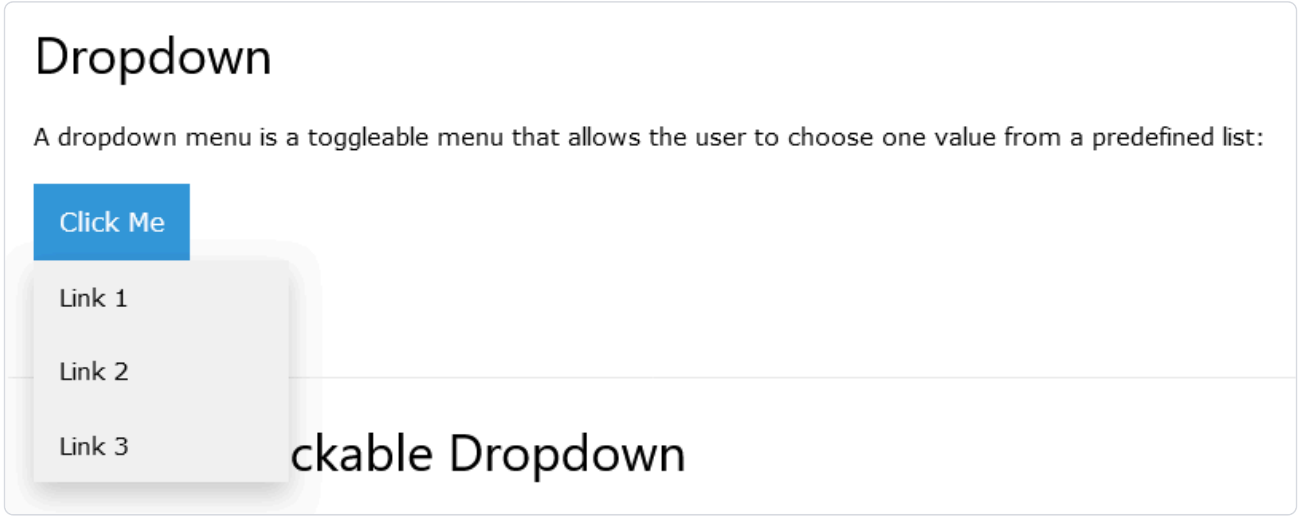
Dans cet exercice, vous allez créer un formulaire avec un sélecteur de couleur et un bouton. L'objectif est de permettre à l'utilisateur de choisir une couleur à l'aide du sélecteur de couleur, puis de changer la couleur de fond de la page en cliquant sur le bouton.

5. Minuterie

Dans cet exercice, vous allez créer une minuterie en utilisant HTML, CSS et JavaScript. L'utilisateur pourra spécifier une durée en secondes à l'aide d'un champ de saisie numérique, puis démarrer et arrêter la minuterie à l'aide de boutons dédiés.

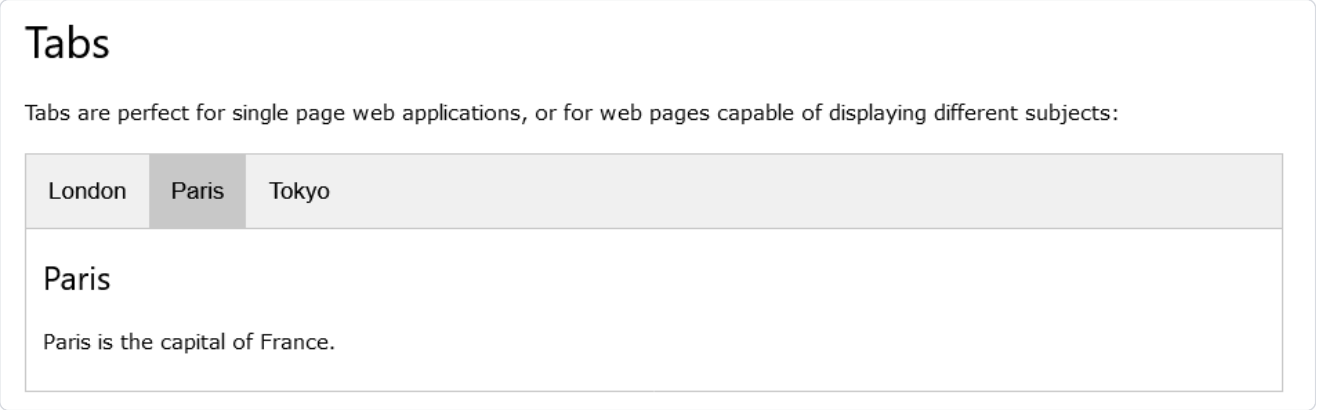
6. Menu dropdown

Dans cet exercice, vous allez créer un menu dropdown en utilisant HTML, CSS et JavaScript. Un menu dropdown est un élément de navigation qui affiche une liste d'options lorsque l'utilisateur clique dessus.



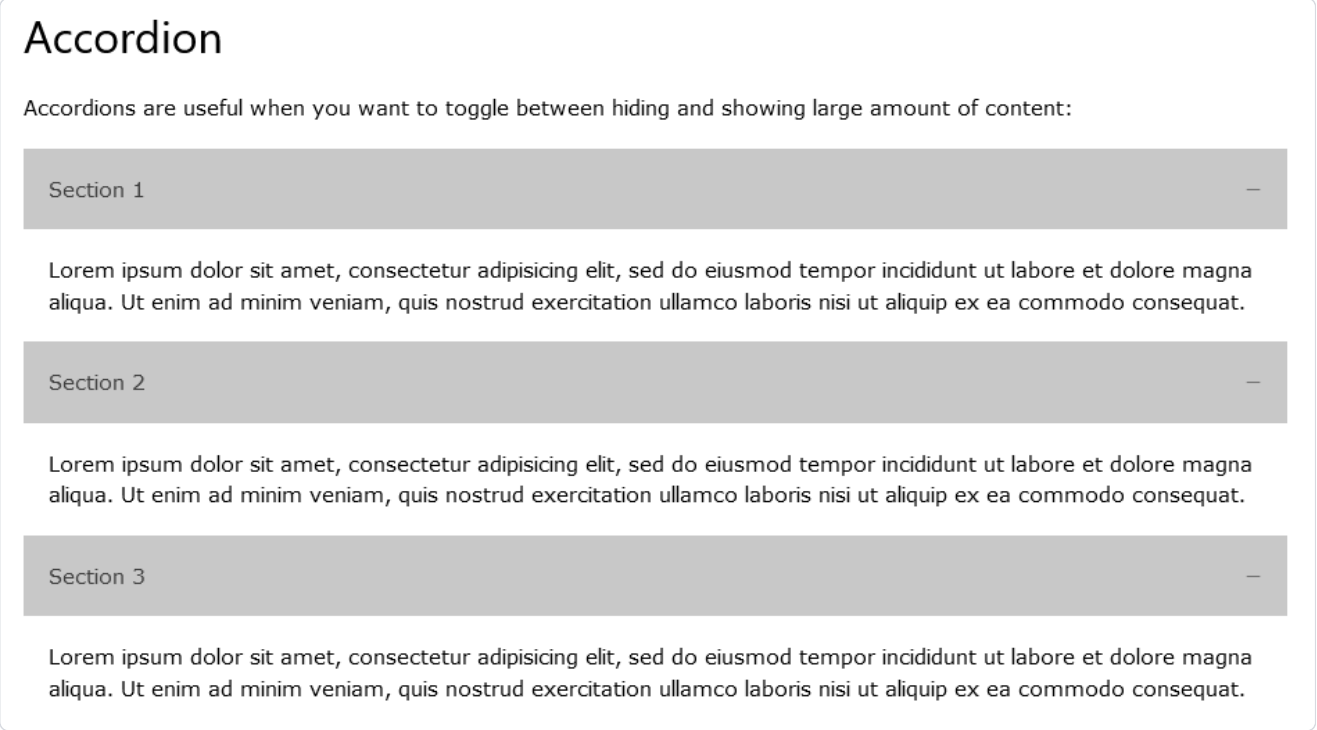
7. Un élément avec plusieurs "tabs"

Dans cet exercice, vous allez créer un élément avec plusieurs "tabs" ou onglets en utilisant HTML, CSS et JavaScript. Chaque onglet affichera un contenu différent lorsque l'utilisateur clique dessus.



8. Accordéon

Dans cet exercice, vous allez créer un accordéon interactif en utilisant HTML et JavaScript. Un accordéon est un élément qui comporte un titre et une description, et lorsque l'utilisateur clique sur le titre, la description s'affiche ou se cache.



9. Navigation responsive avec menu hamburger

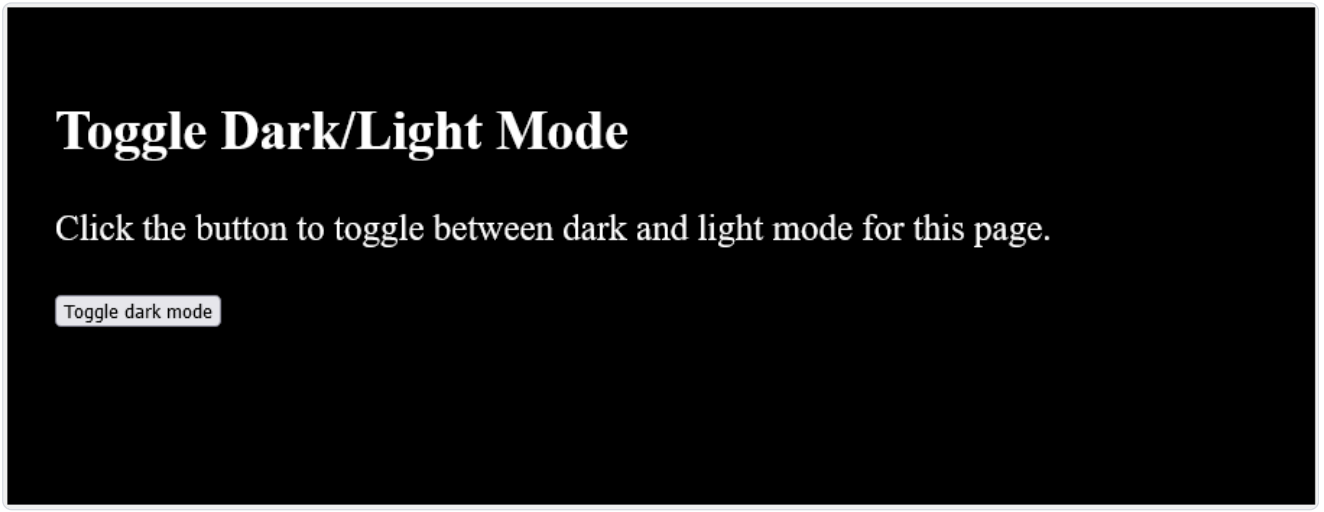
Dans cet exercice, vous allez créer une navigation responsive avec un menu hamburger en utilisant HTML, CSS et JavaScript. Cette navigation s'adaptera à différents appareils et affichera un menu hamburger lorsqu'elle est consultée sur des écrans plus petits.





10. Création d'un mode sombre (Dark Mode)

Dans cet exercice, vous allez créer un mode sombre (dark mode) pour votre page web en utilisant HTML, CSS et JavaScript. Le mode sombre permettra de modifier l'apparence de la page en utilisant des couleurs sombres pour offrir une expérience visuelle différente.Pour sauvegarder une valeur en session, vous pouvez utiliser l'objet `sessionStorage` en JavaScript. L'objet `sessionStorage` permet de stocker des données de manière temporaire pendant la durée de vie de la session de navigation.



Voici comment sauvegarder une valeur en session à l'aide de `sessionStorage` :

1. Pour stocker une valeur en session, utilisez la méthode `setItem()` de l'objet `sessionStorage`. Cette méthode prend deux paramètres : la clé (ou l'identifiant) de la valeur et sa valeur elle-même.

   ```javascript
   sessionStorage.setItem('maValeur', 'valeur');
   ```

   Dans cet exemple, la valeur `'valeur'` est stockée en utilisant la clé `'maValeur'`.

2. Pour récupérer la valeur stockée en session, utilisez la méthode `getItem()` de l'objet `sessionStorage`. Cette méthode prend la clé de la valeur en tant que paramètre et renvoie la valeur correspondante.

```javascript
const valeur = sessionStorage.getItem('maValeur');
```

Dans cet exemple, la valeur stockée avec la clé `'maValeur'` est récupérée et assignée à la variable `valeur`.

3. Pour supprimer une valeur stockée en session, utilisez la méthode `removeItem()` de l'objet `sessionStorage`. Cette méthode prend la clé de la valeur en tant que paramètre et supprime la valeur correspondante.

```javascript
sessionStorage.removeItem('maValeur');
```

Dans cet exemple, la valeur stockée avec la clé `'maValeur'` est supprimée de la session.

Il est important de noter que les données stockées en session sont spécifiques à chaque onglet ou fenêtre de navigation. Elles sont également supprimées lorsque la session de navigation se termine (par exemple, lorsque l'onglet est fermé).

Utilisez ces méthodes pour sauvegarder et récupérer des valeurs en session à mesure que l'utilisateur interagit avec votre application. Pour récupérer la préférence de thème sombre du navigateur, vous pouvez utiliser l'API `matchMedia` en JavaScript. L'API `matchMedia` permet de vérifier si une requête de media correspond à l'environnement d'affichage actuel du navigateur, y compris les préférences de thème sombre ou clair.

Voici comment récupérer la préférence de thème sombre du navigateur :

```javascript
if (window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches) {
    // Le navigateur a la préférence pour le thème sombre
    // Faites quelque chose en conséquence
} else {
    // Le navigateur n'a pas la préférence pour le thème sombre
    // Faites quelque chose en conséquence
}
```

Dans cet exemple, nous utilisons `window.matchMedia` pour vérifier si la préférence de thème sombre est activée. La requête `'(prefers-color-scheme: dark)'` spécifie que nous recherchons la préférence de thème sombre. Si la préférence de thème sombre est détectée, la condition `matches` renvoie `true` et vous pouvez effectuer des actions spécifiques en conséquence. Si la préférence de thème sombre n'est pas détectée, la condition `matches` renvoie `false` et vous pouvez effectuer d'autres actions en conséquence.

Il est important de noter que tous les navigateurs ne prennent pas en charge l'API `matchMedia` ou la requête `prefers-color-scheme`. Assurez-vous de vérifier la compatibilité du navigateur avant de l'utiliser dans votre application.

Source

- https://www.w3schools.com/

2.4 Vite

Introduction à Vite

Qu'est-ce que Vite ?

Vite est un outil de build moderne conçu pour faciliter et accélérer le développement de projets web. Il se sert de la puissance de JavaScript moderne et des modules ES pour offrir une expérience de développement plus rapide.

Pourquoi utiliser Vite dans les projets web ?

- **Rapidité :** Démarrage instantané du serveur de développement et rechargement des modules à la volée.
- **Facilité :** Configuration simplifiée par rapport aux outils de build traditionnels.
- **Modulable :** Extensible avec des plugins et compatible avec de nombreux frameworks.

Fonctionnalités clés de Vite

- **Serveur de développement ultra-rapide :** Profitez d'un temps de chargement réduit lors du développement.
- **Hot Module Replacement (HMR) :** Appliquez les changements en temps réel sans recharger toute la page.
- **Build optimisé pour la production :** Générez votre application avec les meilleures pratiques pour une performance optimale.

L'écosystème Vite

Plugins populaires	Description
`@vitejs/plugin-vue`	Intègre le support de Vue.js
`@vitejs/plugin-react`	Intègre le support de React
`vite-plugin-svelte`	Intègre le support de Svelte

Intégration avec des frameworks

Vite peut être facilement intégré avec des frameworks tels que Vue.js, React et bien d'autres, grâce à des templates de projet ou des plugins spécifiques.

Conclusion

Vite offre une expérience de développement moderne et efficace, avec des fonctionnalités qui accélèrent le processus de développement et de build. Pour en apprendre davantage, consultez la documentation officielle de Vite.

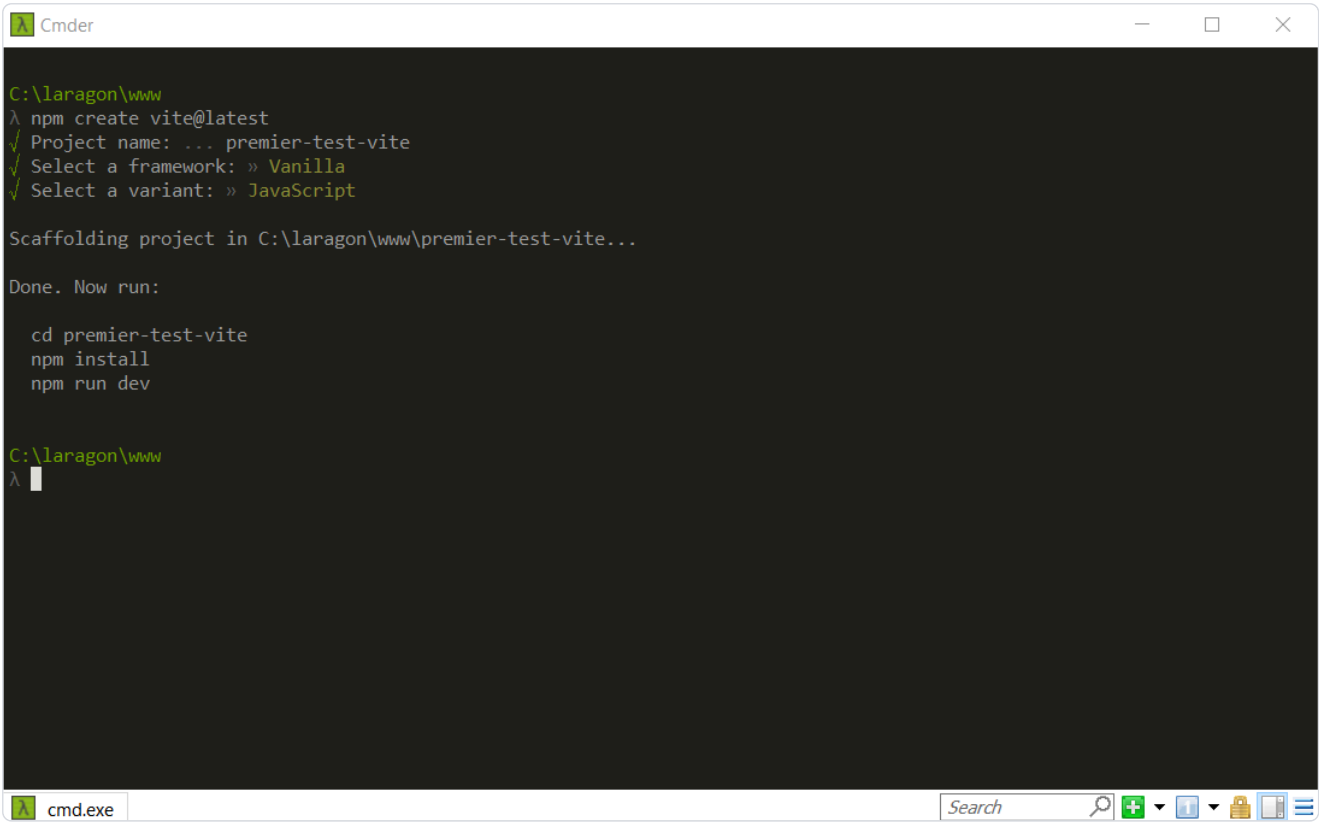
Installation et configuration de base

Comment installer Vite

Pour installer Vite, il suffit d'utiliser votre gestionnaire de paquets préféré, comme NPM ou Yarn. Cette commande va créer un projet Vite vide dans un dossier. L'installeur vous demandera de choisir un nom pour votre projet :

```bash
npm create vite@latest
```

Choisissez les mêmes options que dans l'exemple suivant :



Ensuite, rendez-vous dans le dossier du projet :

```bash
cd mon-projet
```

La commande `cd` permet de Changer de Dossier (Change Directory en anglais).

```bash
npm install

## ou l'alias

npm i
```

```
C:\laragon\www\premier-test-vite
λ npm i

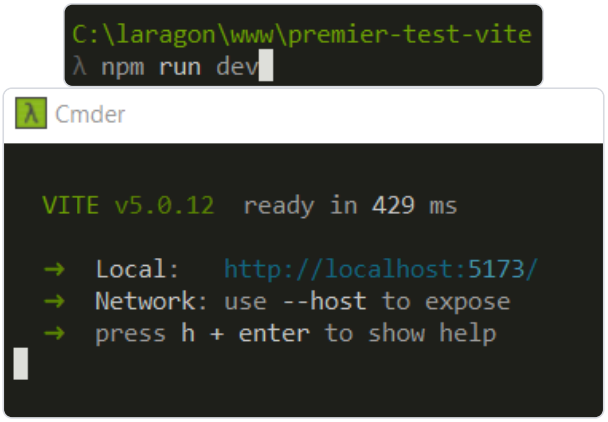
added 10 packages, and audited 11 packages in 4s

3 packages are looking for funding
  run `npm fund` for details

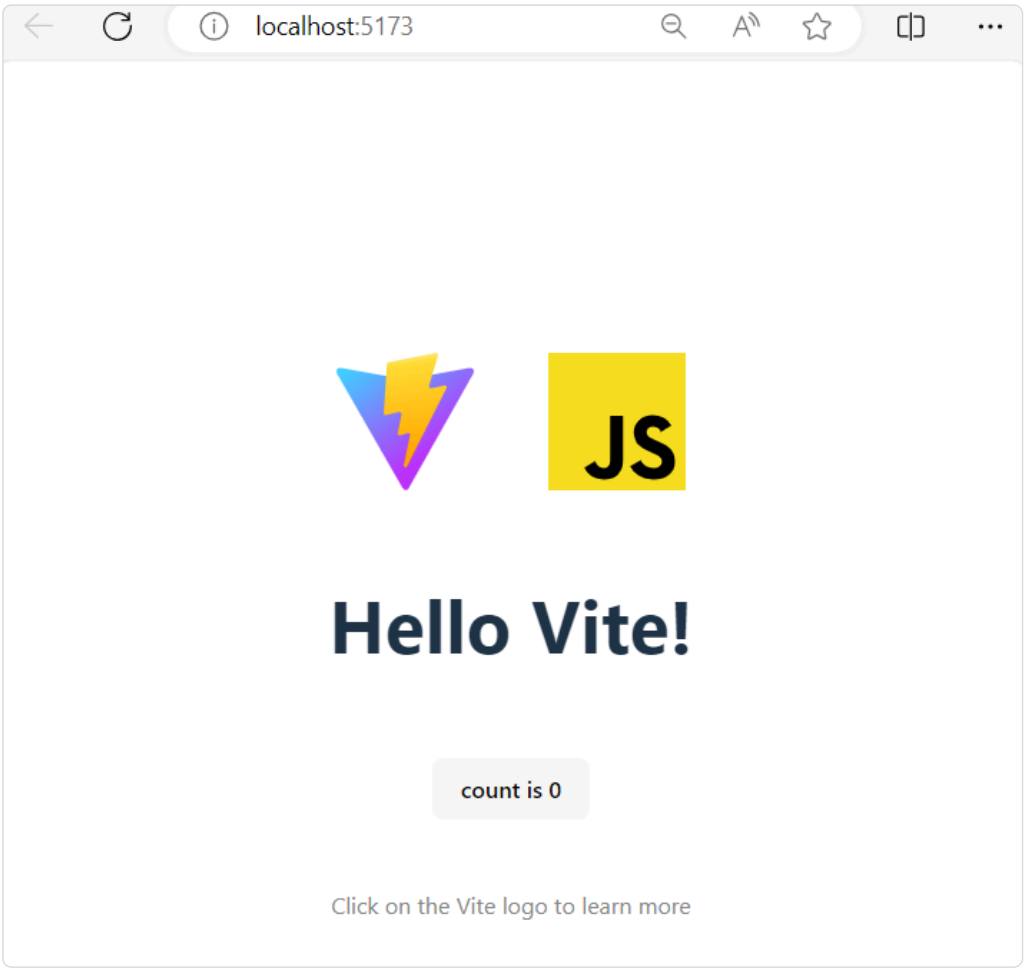
found 0 vulnerabilities
```

Cette commande va installer les “dépendances” de notre projet. C’est à dire qu’il va télécharger le code du logiciel Vite dans notre projet. Ces dépendances seront installées dans le dossier `node_modules`. Il ne faut surtout pas modifier les fichiers contenus dans ce dossier.

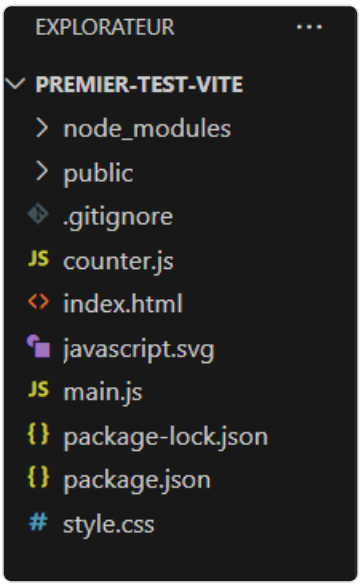
```
npm run dev
```



Cette commande permet de lancer le serveur de développement de Vite. Vous pouvez cliquer sur le lien (CMD+clic) pour ouvrir la page dans votre navigateur.



Structure d’un nouveau projet Vite



Package.json

C’est le fichier dans lequel sont définies les dépendances du projet ainsi que les commandes disponibles (scripts).

```json
{
  "name": "premier-test-vite",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "vite": "^5.0.8"
  }
}
```

- `"dev"` : lance le serveur de développement local (`npm run dev`)
- `"build"` : construit votre application pour la production (`npm run build`)
- `"preview"` : lance un serveur local avec la version buildée de votre app (`npm run preview`)

index.html

Contient le code html de la page d’accueil du site.

```xml
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/main.js"></script>
  </body>
</html>
```

main.js

```javascript
import './style.css'
import javascriptLogo from './javascript.svg'
import viteLogo from '/vite.svg'
import { setupCounter } from './counter.js'

document.querySelector('#app').innerHTML = `
  <div>
    <a href="https://vitejs.dev" target="_blank">
      <img src="viteLogo"class="logo"alt="Vitelogo"/></a>
<ahref="https://developer.mozilla.org/en-US/docs/Web/JavaScript"target="blank"><imgsrc="{viteLogo}"
class="logo" alt="Vite logo" />
    </a>
    <a href="https://developer.mozilla.org/en-US/docs/Web/JavaScript" target="_blank">
      <img src="viteLogo"class="logo"alt="Vitelogo"/></a>
<ahref="https://developer.mozilla.org/en-US/docs/Web/JavaScript"target="blank"><imgsrc="{javascriptLogo}"
class="logo vanilla" alt="JavaScript logo" />
    </a>
    <h1>Hello Vite!</h1>
    <div class="card">
      <button id="counter" type="button"></button>
    </div>
    <p class="read-the-docs">
      Click on the Vite logo to learn more
    </p>
  </div>
`

setupCounter(document.querySelector('#counter'))
```

counter.js

```javascript
export function setupCounter(element) {
  let counter = 0
  const setCounter = (count) => {
    counter = count
    element.innerHTML = `count is ${counter}`
  }
  element.addEventListener('click', () => setCounter(counter + 1))
  setCounter(0)
}
```

style.css

```css
:root {
  font-family: Inter, system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

a {
  font-weight: 500;
  color: #646cff;
  text-decoration: inherit;
}
a:hover {
  color: #535bf2;
}

body {
  margin: 0;
  display: flex;
  place-items: center;
  min-width: 320px;
  min-height: 100vh;
}

h1 {
  font-size: 3.2em;
  line-height: 1.1;
}

#app {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}

.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.vanilla:hover {
  filter: drop-shadow(0 0 2em #f7df1eaa);
}

.card {
  padding: 2em;
}

.read-the-docs {
  color: #888;
```

```
}

button {
  border-radius: 8px;
  border: 1px solid transparent;
  padding: 0.6em 1.2em;
  font-size: 1em;
  font-weight: 500;
  font-family: inherit;
  background-color: #1a1a1a;
  cursor: pointer;
  transition: border-color 0.25s;
}
button:hover {
  border-color: #646cff;
}
button:focus,
button:focus-visible {
  outline: 4px auto -webkit-focus-ring-color;
}

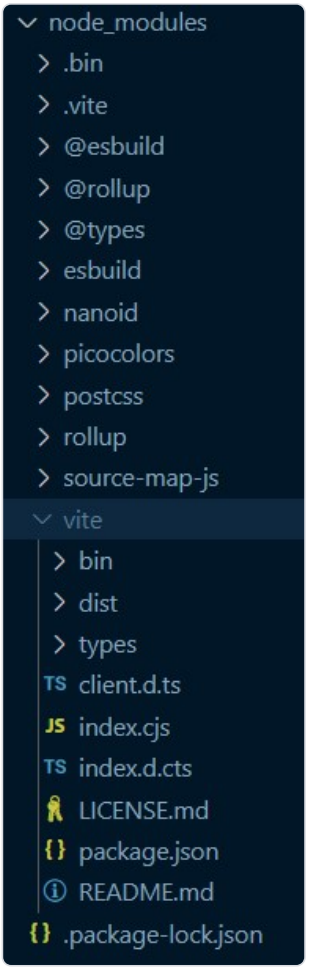
@media (prefers-color-scheme: light) {
  :root {
    color: #213547;
    background-color: #ffffff;
  }
  a:hover {
    color: #747bff;
  }
  button {
    background-color: #f9f9f9;
  }
}
```

Dossier public

Le dossier public contient les fichiers statiques comme les images.

Dossier node_modules

Contient les fichiers des dépendances de notre application.



!!! Il ne faut jamais modifier les fichiers contenus dans ce dossier !!!

Gestion des images

Répertoire `public`

Utilisez le répertoire `public` pour :

- Fichiers non référencés dans le code (ex : `robots.txt`).
- Fichiers nécessitant un nom constant sans hash.
- Accès direct sans import.

Exemple : Pour utiliser une image du répertoire `public`, insérez-la simplement dans votre HTML :

```xml
<img src="/logo.png" alt="Logo">
```

Importation de Fichiers Statiques

L'importation d'images retourne l'URL publique lorsqu'elle est servie :

Exemple :

```javascript
import imgUrl from './img.png';
document.getElementById('hero-img').src = imgUrl;
```

Cette méthode est idéale pour les images référencées dans votre code, permettant l'optimisation automatique par Vite.

Résumé : Vite.js offre deux approches pour les fichiers : le répertoire `public` pour un accès simple et direct, ou l'importation pour l'optimisation et le contrôle des URLs.

JSON

Pour récupérer les données d'un fichier JSON, il suffit de l'importer comme ceci :

```javascript
import json from './example.json'
```

MPA (Multi Pages App)

Pour créer une application avec plusieurs pages (MPA) dans Vite, vous devez organiser vos fichiers par page et configurer Vite pour reconnaître chaque page comme un point d'entrée. Voici un guide simplifié :

Pourquoi Faire Ça ?

- **Organisation** : Séparez clairement le contenu de chaque page de votre application.
- **Flexibilité** : Gérez des ressources et scripts spécifiques à chaque page.

Comment Procéder

Structure de Base

Imaginez une structure avec une page d'accueil (`index.html`) et une page secondaire (`nested/index.html`).

```bash
.
├── index.html       # Page d'accueil
├── main.js          # Script de la page d'accueil
└── nested
    ├── index.html   # Page secondaire
    └── nested.js    # Script de la page secondaire
```

En Développement

En mode développement, accédez simplement à `/nested/` pour voir la page secondaire. Vite sert ces pages comme un serveur de fichiers statiques.

Pour la Production (Build)

1. **Configurer les Points d'Entrée :** Dans `vite.config.js` , spécifiez chaque fichier `.html` comme un point d'entrée.

```javascript
// vite.config.js
import { defineConfig } from 'vite'
import { resolve } from 'path'

export default defineConfig({
  build: {
    rollupOptions: {
      input: {
        main: resolve(__dirname, 'index.html'),         // Page d'accueil
        nested: resolve(__dirname, 'nested/index.html'), // Page secondaire
      },
    },
  },
})
```

Cette configuration indique à Vite de préparer chaque page pour la production, en conservant leur structure et accessibilité.

Point Clé : Utilisez `resolve(__dirname, 'chemin/vers/fichier.html')` pour s'assurer que Vite trouve correctement les fichiers sur votre ordinateur. Pensez à importer le module `resolve`.

Pour gérer plusieurs pages dans Vite, structurez vos fichiers par page et ajustez `vite.config.js` pour inclure chaque page comme un point d'entrée. Cela permet à Vite de traiter et servir chaque page de manière appropriée, en mode développement et pour la version finale.

2.5 Objets et JSON

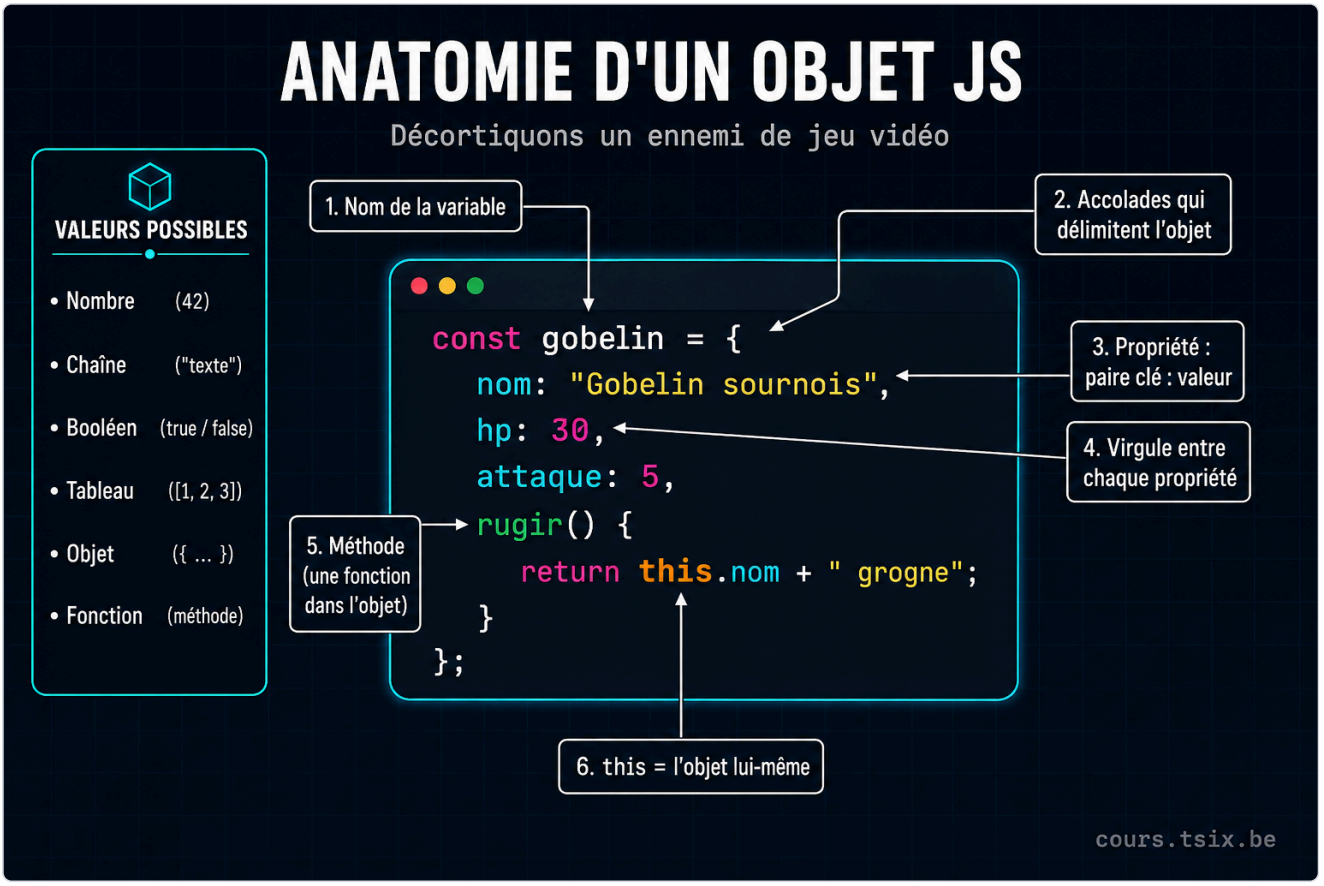
Pourquoi les objets ?

Imagine que tu codes un jeu vidéo. Tu veux représenter un héros. Avec ce que tu connais déjà, tu pourrais faire :

```javascript
let pseudo = "Lex";
let niveau = 12;
let hp = 85;
let vivant = true;
```

Quatre variables pour **un seul** personnage. Maintenant ajoute trois ennemis dans la pièce, et tu te retrouves avec `pseudoEnnemi1`, `niveauEnnemi1`, `hpEnnemi1`, `pseudoEnnemi2`... cauchemar 😵.

Un **objet** te permet de regrouper toutes ces informations dans une seule structure cohérente. Une variable, plein de propriétés.



Créer un objet

Un objet, c'est un sac à variables. On le crée avec des accolades `{}`, et chaque variable à l'intérieur s'appelle une **propriété**.

```javascript
const heros = {
  pseudo: "Lex",
  niveau: 12,
  hp: 85,
  vivant: true
};
```

Chaque propriété est une paire **clé : valeur**, séparées par une virgule. Les valeurs peuvent être de **n'importe quel type** : nombre, chaîne, booléen, tableau, et même un autre objet.

Accéder aux propriétés

Deux notations possibles : la **notation point** et la **notation crochets**.

```javascript
console.log(heros.pseudo);    // "Lex"
console.log(heros["niveau"]); // 12
```

Quelle utiliser ? La notation point est plus lisible, c'est donc la **valeur par défaut**. Les crochets servent quand le nom de la propriété est dans une variable :

```javascript
const champ = "hp";
console.log(heros[champ]); // 85 — équivaut à heros.hp
```

💡 Les crochets sont aussi obligatoires si la clé contient un espace ou un caractère bizarre — mais en pratique on évite ces cas.

Modifier les propriétés

Même syntaxe, mais avec un `=` :

```javascript
heros.hp = 50;       // Le héros a pris des dégâts
heros.niveau = 13;   // Level up!
```

Tu peux aussi **ajouter une propriété qui n'existait pas** :

```javascript
heros.classe = "Mage";
console.log(heros.classe); // "Mage"
```

Ou la **supprimer** avec `delete` :

```javascript
delete heros.classe;
```

⚠️ Note : même si on a déclaré `heros` avec `const`, on peut modifier ses propriétés. `const` empêche de **réassigner** la variable elle-même (`heros = autreChose`), mais pas de **modifier le contenu** de l'objet.

Méthodes et `this`

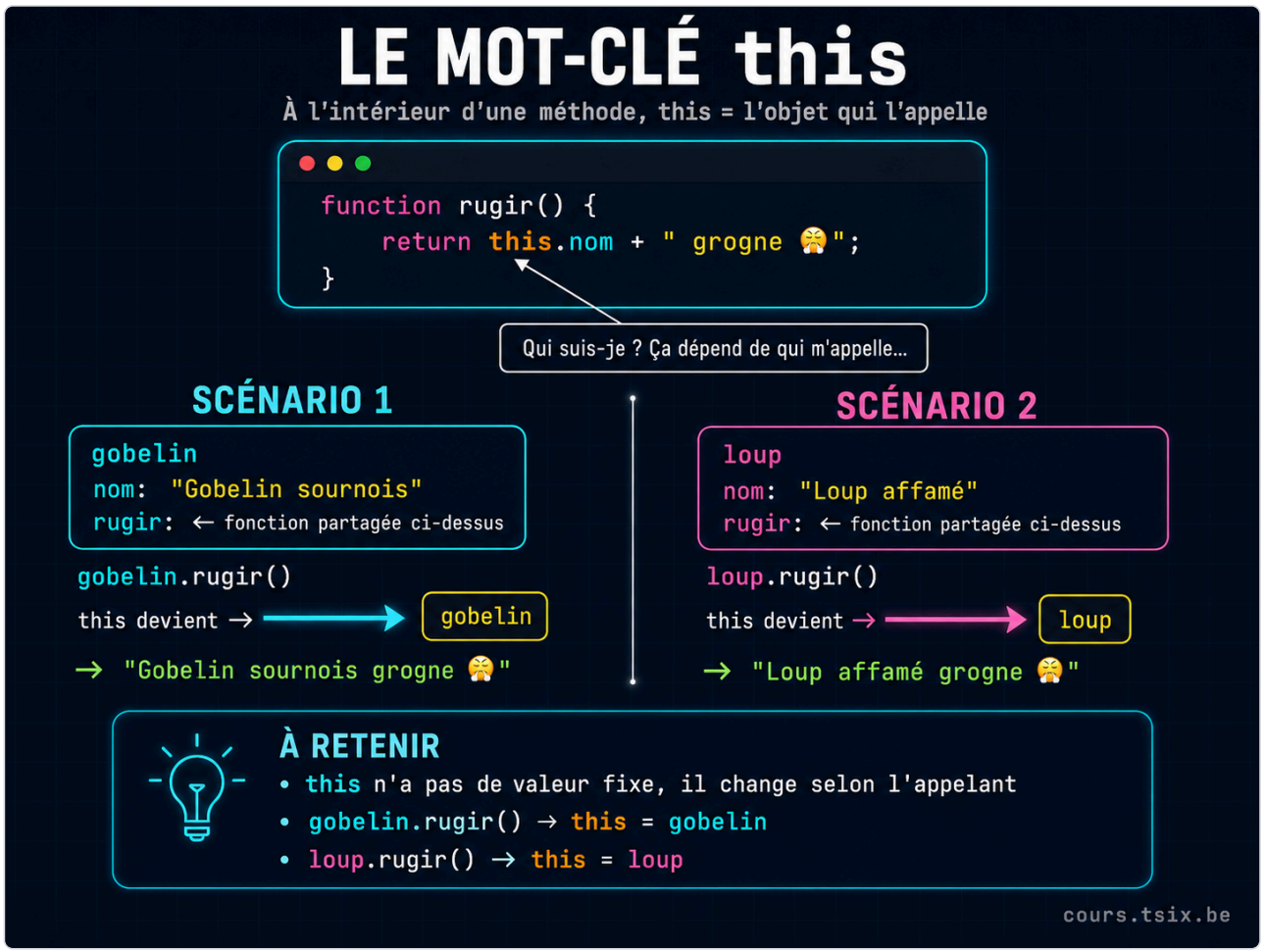
Une **méthode** est juste une fonction stockée dans un objet. Exemple avec un ennemi qui a un comportement :

```javascript
const gobelin = {
  nom: "Gobelin sournois",
  hp: 30,
  attaque: 5,
  rugir: function() {
    return this.nom + " grogne 😤";
  }
};

console.log(gobelin.rugir()); // "Gobelin sournois grogne 😤"
```

Le mot-clé `this` désigne **l'objet propriétaire de la méthode**. Ici, à l'intérieur de `rugir`, `this` représente `gobelin`. Donc `this.nom` équivaut à `gobelin.nom`.

💡 **Pourquoi ne pas écrire** `gobelin.nom` directement ? Parce qu'avec `this`, la méthode reste valide même si tu copies l'objet ou si tu en crées d'autres similaires. C'est plus robuste et plus réutilisable.



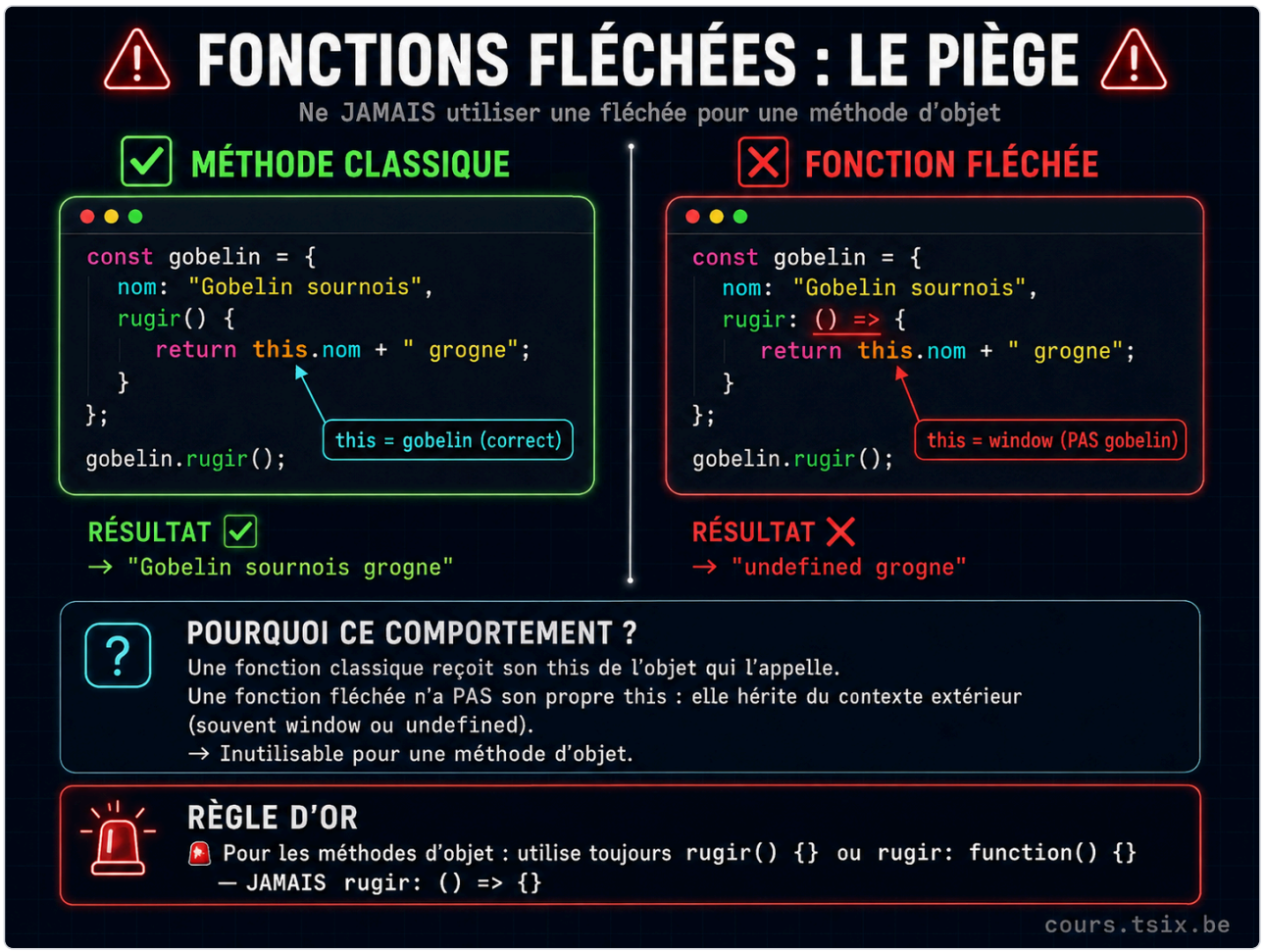
Syntaxe raccourcie pour les méthodes

JavaScript propose un raccourci très pratique pour écrire les méthodes :

```javascript
const gobelin = {
  nom: "Gobelin sournois",
  hp: 30,
  rugir() {  // au lieu de "rugir: function()"
    return this.nom + " grogne 😤";
  },
  prendreDegats(montant) {
    this.hp = this.hp - montant;
    return `this.nomamaintenant{this.nom} a maintenant this.nomamaintenant{this.hp} HP`;
  }
};

console.log(gobelin.prendreDegats(10)); // "Gobelin sournois a maintenant 20 HP"
```

C'est cette syntaxe qu'on préfère aujourd'hui : plus lisible, moins de bruit visuel.



Objets imbriqués et tableaux

Une propriété peut contenir **un autre objet** ou **un tableau d'objets**. C'est là que ça devient puissant — tu peux modéliser des structures aussi complexes que tu veux.

```javascript
const carte = {
  nom: "Forêt Maudite",
  taille: { largeur: 100, hauteur: 100 },
  spawn: { x: 5, y: 5 },
  ennemis: [
    { type: "gobelin", x: 10, y: 12, hp: 30 },
    { type: "loup",    x: 25, y: 30, hp: 45 },
    { type: "sorcier", x: 60, y: 80, hp: 70 }
  ]
};
```

Pour accéder aux infos imbriquées, on **enchaîne les accès** :

```javascript
carte.taille.largeur;       // 100
carte.spawn.x;              // 5
carte.ennemis.length;       // 3
carte.ennemis[0].type;      // "gobelin"
carte.ennemis[2].hp;        // 70
```

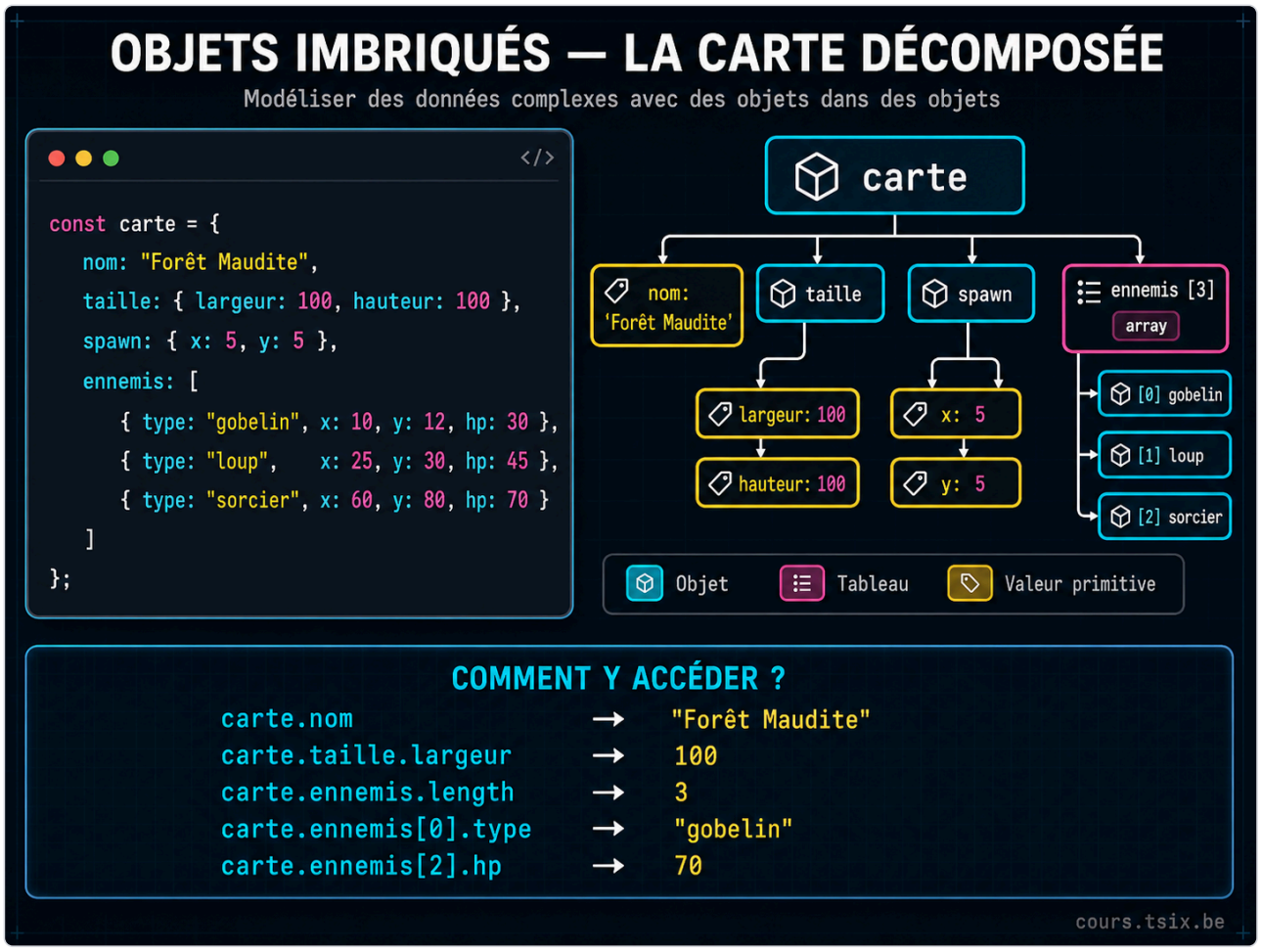
Tu peux aussi **boucler** sur les ennemis comme tu le ferais sur n'importe quel tableau :

```javascript
for (const ennemi of carte.ennemis) {
  console.log(`Un ennemi.typeen({ennemi.type} en (ennemi.typeen({ennemi.x}, ${ennemi.y})`);
}
```

⚠️ **Piège classique** : si tu accèdes à une propriété qui n'existe pas, tu obtiens `undefined`. Et si tu essaies d'aller plus profond, ça plante :

```javascript
carte.boss;          // undefined (pas d'erreur)
carte.boss.nom;      // ❌ ERREUR : Cannot read properties of undefined
```

Pense toujours à vérifier qu'une propriété existe avant d'aller dedans.



Petit tour des objets natifs JS

JavaScript te fournit déjà toute une collection d'objets prêts à l'emploi. En voici trois utiles pour vos jeux.

`Math` — pour les calculs

```javascript
Math.random();          // Nombre aléatoire entre 0 (inclus) et 1 (exclus)
Math.floor(3.7);        // 3 (arrondi inférieur)
Math.max(10, 20, 5);    // 20
Math.PI;                // 3.14159...
```

Recette utile : un **nombre entier aléatoire entre 0 et 9** :

```javascript
Math.floor(Math.random() * 10);
```

`Date` — pour les dates

```javascript
const maintenant = new Date();
console.log(maintenant);                // date actuelle
console.log(maintenant.getFullYear());  // 2026
```

`Array` — tu connais déjà

Les tableaux sont des objets en JavaScript. Les méthodes que tu utilises (`push`, `length`, `map`, `filter`...) sont en fait des **méthodes de l'objet** `Array`.

💡 Le navigateur expose plein d'autres objets puissants (`window`, `document`, `localStorage`...) qu'on découvrira dans un prochain chapitre.

JSON : du texte qui ressemble à un objet

JSON signifie **JavaScript Object Notation**. C'est un **format texte** qui permet de représenter des données structurées.

À quoi ça ressemble ?

```json
{
  "pseudo": "Lex",
  "niveau": 12,
  "hp": 85,
  "vivant": true
}
```

Ça a l'air d'un objet JavaScript, mais attention aux **différences importantes** :

Objet JS	JSON
Clés sans guillemets : `pseudo:`	Clés **toujours entre guillemets doubles** : `"pseudo":`
Guillemets simples ou doubles autorisés	**Guillemets doubles uniquement**
Peut contenir des fonctions	**Pas de fonctions**
Commentaires possibles	**Pas de commentaires**

À quoi ça sert ?

JSON est le format **universel** pour échanger des données. Quand ton appli web parle à un serveur, à une API, à un fichier de config — c'est presque toujours du JSON qui transite.

Convertir entre objet et JSON

JavaScript te donne deux outils intégrés :

`JSON.stringify(objet)` transforme un objet en texte JSON :

```javascript
const heros = { pseudo: "Lex", niveau: 12 };
const texte = JSON.stringify(heros);
console.log(texte); // '{"pseudo":"Lex","niveau":12}'
```

`JSON.parse(texte)` fait l'inverse :

```javascript
const texte = '{"pseudo":"Lex","niveau":12}';
const objet = JSON.parse(texte);
console.log(objet.pseudo); // "Lex"
```

JSON peut contenir tout ce que tu veux

Tableaux, sous-objets, mélanges — tout est permis :

```json
{
  "pseudo": "Lex",
  "stats": { "hp": 85, "mana": 120 },
  "inventaire": [
    { "nom": "Potion", "quantite": 3 },
    { "nom": "Épée",   "quantite": 1 }
  ],
  "actif": true
}
```

C'est exactement la même structure que les objets imbriqués qu'on a vus plus haut, juste écrite en texte.

Importer un JSON dans Vite

Dans un projet Vite, on charge rarement le JSON via `fetch`. Il existe une syntaxe beaucoup plus simple : **l'import**.

Crée un fichier `data/joueurs.json` à côté de ton script :

```json
[
  { "id": 1, "pseudo": "Lex", "niveau": 12 },
  { "id": 2, "pseudo": "Nyx", "niveau": 24 }
]
```

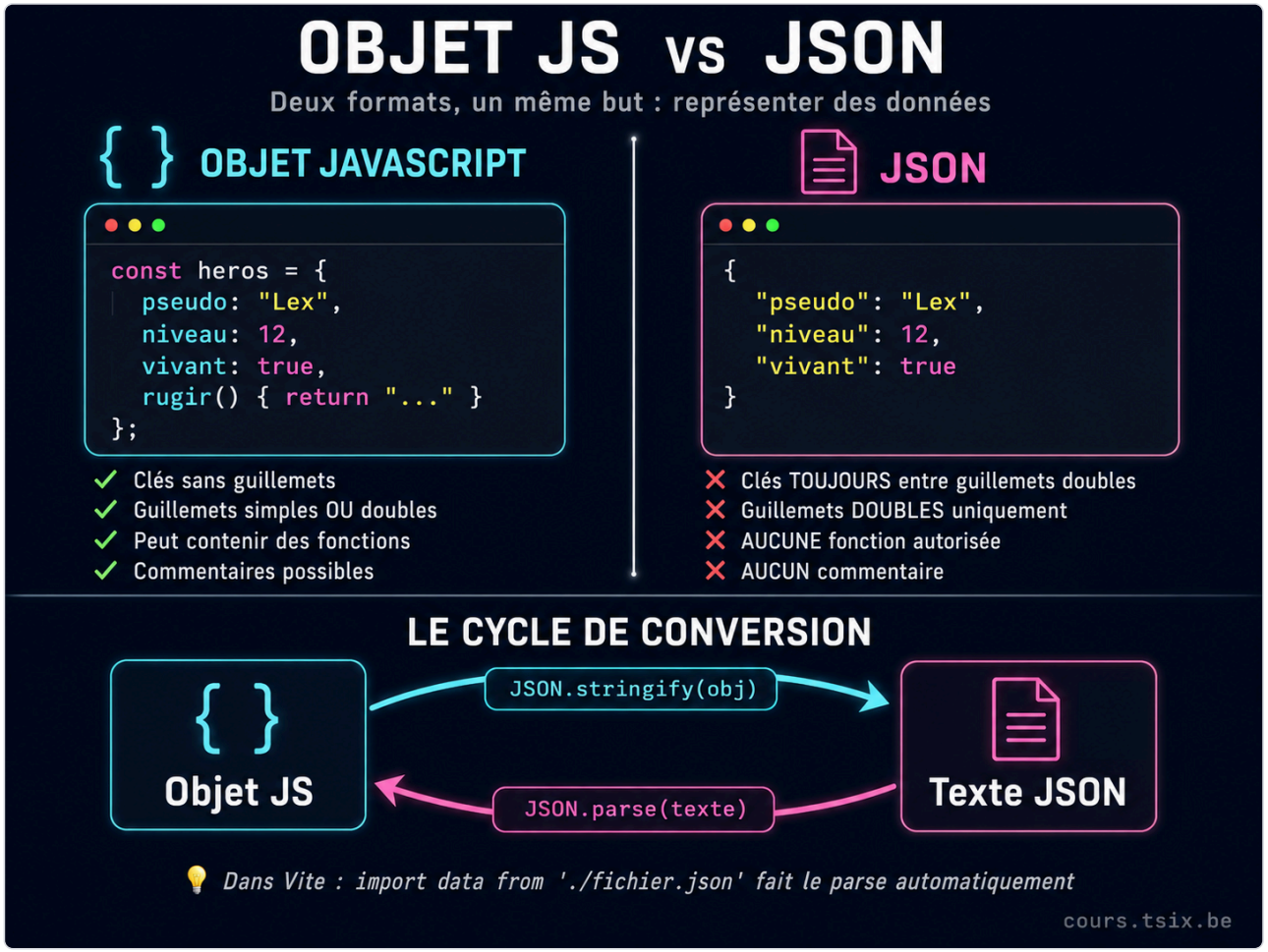
Puis dans ton `main.js` :

```javascript
import joueurs from './data/joueurs.json';

console.log(joueurs);            // tableau d'objets
console.log(joueurs[0].pseudo);  // "Lex"
```

✅ Vite parse automatiquement le JSON pour toi. Tu manipules un objet JavaScript directement, sans aucune conversion.

⚠️ Cette syntaxe ne fonctionne **pas en JavaScript pur** dans le navigateur (sans bundler). Elle est spécifique aux outils comme Vite, Webpack, Parcel.



Exercices de lecture

Exercice 1

Considère cet objet :

```javascript
const personnage = {
  pseudo: "Mira",
  niveau: 18,
  stats: { hp: 150, mana: 80, force: 40 },
  inventaire: ["potion", "torche", "carte"]
};
```

1. Comment accéder au pseudo ?
2. Comment accéder à la valeur de mana ?
3. Comment accéder au premier objet de l'inventaire ?
4. Comment afficher combien d'objets sont dans l'inventaire ?

Exercice 2

Considère cette chaîne JSON :

```javascript
const texte = '{"pseudo":"Mira","niveau":18,"hp":150}';
```

1. Comment la convertir en objet JavaScript ?
2. Une fois convertie, comment accéder au niveau ?

Exercice 3

Considère cet objet :

```javascript
const epee = {
  nom: "Lame du crépuscule",
  degats: 25,
  decrire() {
    return `this.nominflige{this.nom} inflige this.nominflige{this.degats} dégâts`;
  }
};
```

1. Comment appeler la méthode `decrire` ?
2. Que retourne-t-elle ?

Exercice 4

Considère cet objet :

```javascript
const equipe = {
  nom: "Les Corbeaux",
  membres: [
    { pseudo: "Lex",  rang: "Capitaine" },
    { pseudo: "Nyx",  rang: "Mage"      },
    { pseudo: "Tank", rang: "Tank"      }
  ]
};
```

1. Comment accéder au pseudo du second membre ?
2. Comment accéder au rang du dernier membre **sans connaître la longueur d'avance** ?

Exercices pratiques

📚 Exercice 1 — Bibliothèque chaotique

Objectif : manipuler un tableau d'objets, gérer un formulaire, afficher dynamiquement.

Énoncé

Tu vas créer une mini-app de bibliothèque avec **des livres complètement inventés**. Règles obligatoires :

- Titres **drôles et fictifs** (ex. *"Comment perdre à StarCraft en 4 leçons"*, *"Le grimoire du goûter"*)
- Auteurs **fictifs** (ex. *Bartholomé Croquette*, *Jeanne-Élise du Tartiflette*)
- Couvertures = images dans le dossier `public/covers/`. Récupère-les sur Unsplash, génère-les en IA, ou dessine-les toi-même

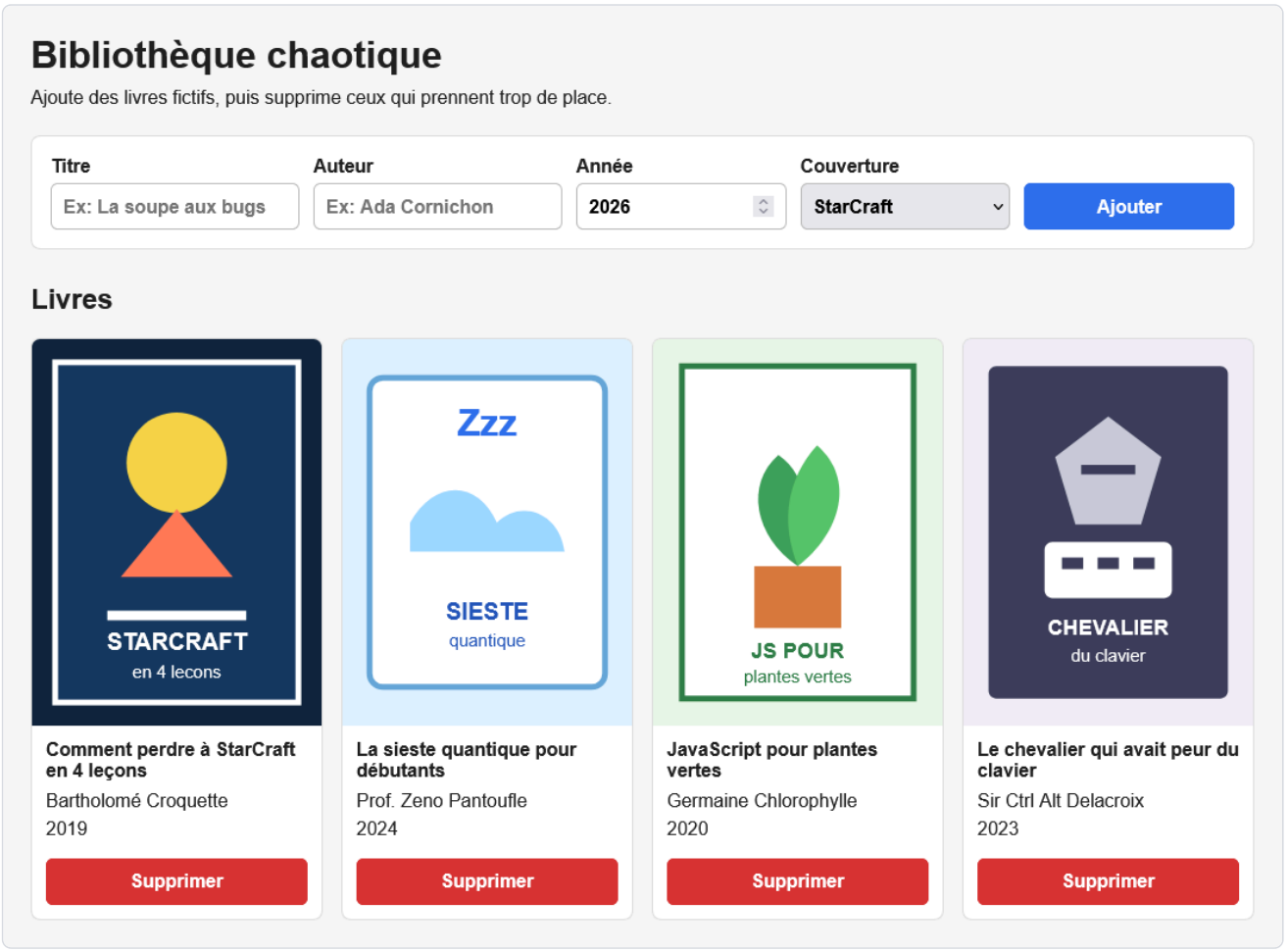
Étapes :

1. Crée un tableau `livres` qui contient au moins **5 livres** au démarrage. Chaque livre est un objet avec : `titre`, `auteur`, `annee`, `couverture` (chemin vers l'image dans `public/`).
2. Crée un **formulaire HTML** avec les champs nécessaires + un bouton "Ajouter".
3. Au clic, ajoute le nouveau livre au tableau et **réaffiche** la liste.

4. Affiche les livres en **grille de cards** : image de couverture, titre, auteur, année.
5. Ajoute un bouton "Supprimer" sur chaque card.

Bonus 💡 : sauvegarde la liste dans le navigateur pour qu'elle persiste après un refresh.

- Pistes : `localStorage.setItem`, `JSON.stringify`, `JSON.parse`
- MDN — localStorage



🎮 Exercice 2 — Roster de joueurs

Objectif : importer un JSON dans Vite, afficher des données imbriquées, manipuler des sous-tableaux.

Énoncé

Tu reçois un fichier `players.json` **incomplet** avec seulement 2 joueurs :

```json
[
  {
    "id": 1,
    "pseudo": "NyxBlade",
    "niveau": 24,
    "classe": "Mage",
    "avatar": "/avatars/nyx.png",
    "stats": { "hp": 180, "mana": 220, "force": 45 },
    "inventaire": [
      { "nom": "Bâton de glace", "type": "arme",        "rarete": "rare" },
      { "nom": "Potion de vie",  "type": "consommable", "quantite": 3    }
    ],
    "guilde": { "nom": "Les Corbeaux d'Argent", "rang": "Lieutenant" }
  },
  {
    "id": 2,
    "pseudo": "TankFromage",
    "niveau": 18,
    "classe": "Guerrier",
    "avatar": "/avatars/tank.png",
    "stats": { "hp": 300, "mana": 40, "force": 80 },
    "inventaire": [
      { "nom": "Marteau du brunch", "type": "arme", "rarete": "épique" }
    ],
    "guilde": { "nom": "Les Corbeaux d'Argent", "rang": "Membre" }
  }
]
```

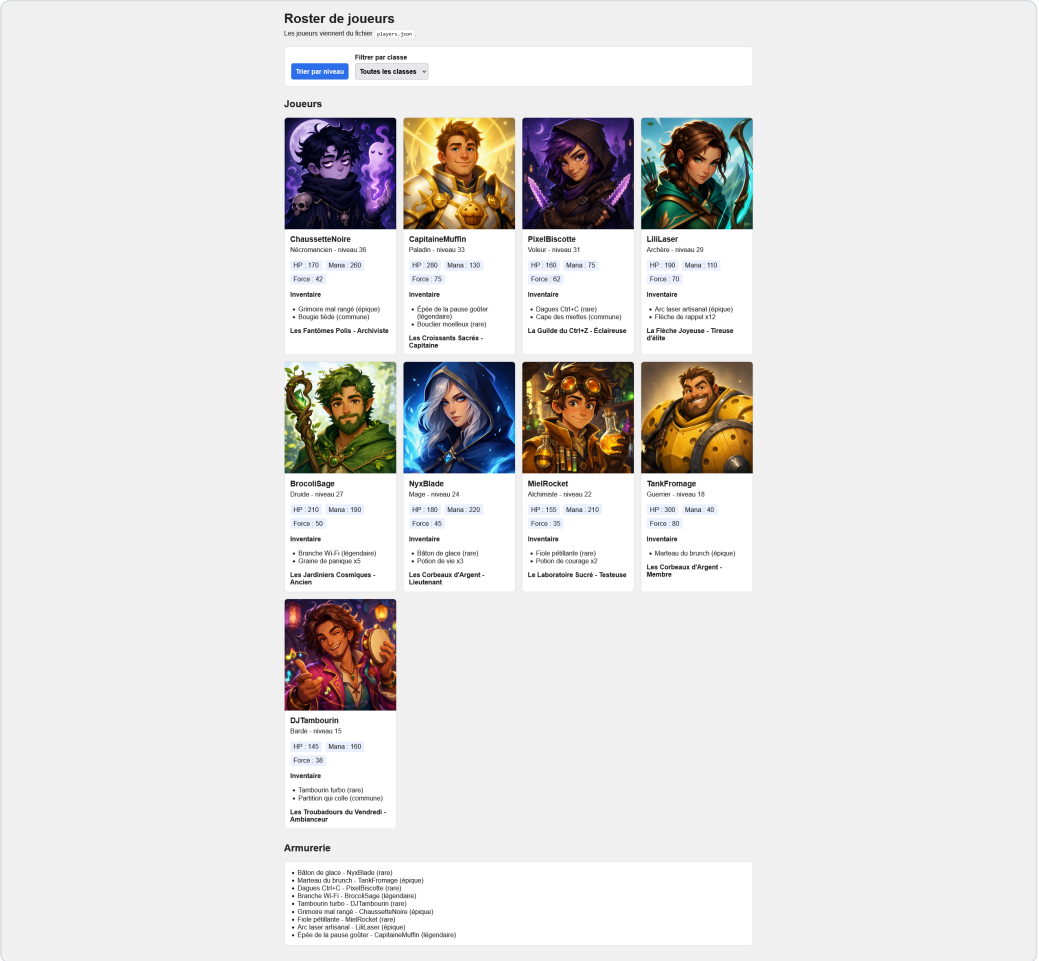
Étapes :

1. **Complète le fichier** pour atteindre **8 joueurs minimum**, en respectant le schéma. Sois créatif sur les pseudos, les classes, les noms d'objets et de guildes.
2. **Importe** ce JSON dans ton script Vite avec `import joueurs from './data/players.json'`.
3. Affiche chaque joueur en **card** : avatar, pseudo, classe, niveau, ses 3 stats (HP / Mana / Force), la **liste des objets de son inventaire** (avec leur rareté quand elle existe), et sa guilde avec son rang.
4. Ajoute un bouton **"Trier par niveau"** (du plus haut au plus bas).

Bonus 1 💡 : ajoute un `<select>` pour filtrer l'affichage par classe.

- Pistes : `Array.filter`, événement `change` sur le select
- MDN — Array.filter

Bonus 2 💡 : affiche une section **"Armurerie"** qui rassemble UNIQUEMENT les armes de tous les joueurs réunis dans une seule liste.



🎨 Exercice 3 — Petit dessinateur SVG

Objectif : manipuler un tableau d'objets hétérogènes (de types différents), générer du SVG dynamiquement.

Énoncé

Tu vas créer un mini-éditeur de formes. Démarre avec ce tableau :

```javascript
const formes = [
  { type: "rect",   x: 10,  y: 10,  largeur: 80, hauteur: 60, couleur: "#ff0066" },
  { type: "cercle", cx: 200, cy: 100, r: 40, couleur: "#00ccff" },
  { type: "ligne",  x1: 0, y1: 0, x2: 300, y2: 200, couleur: "#000000" }
];
```

Étapes :

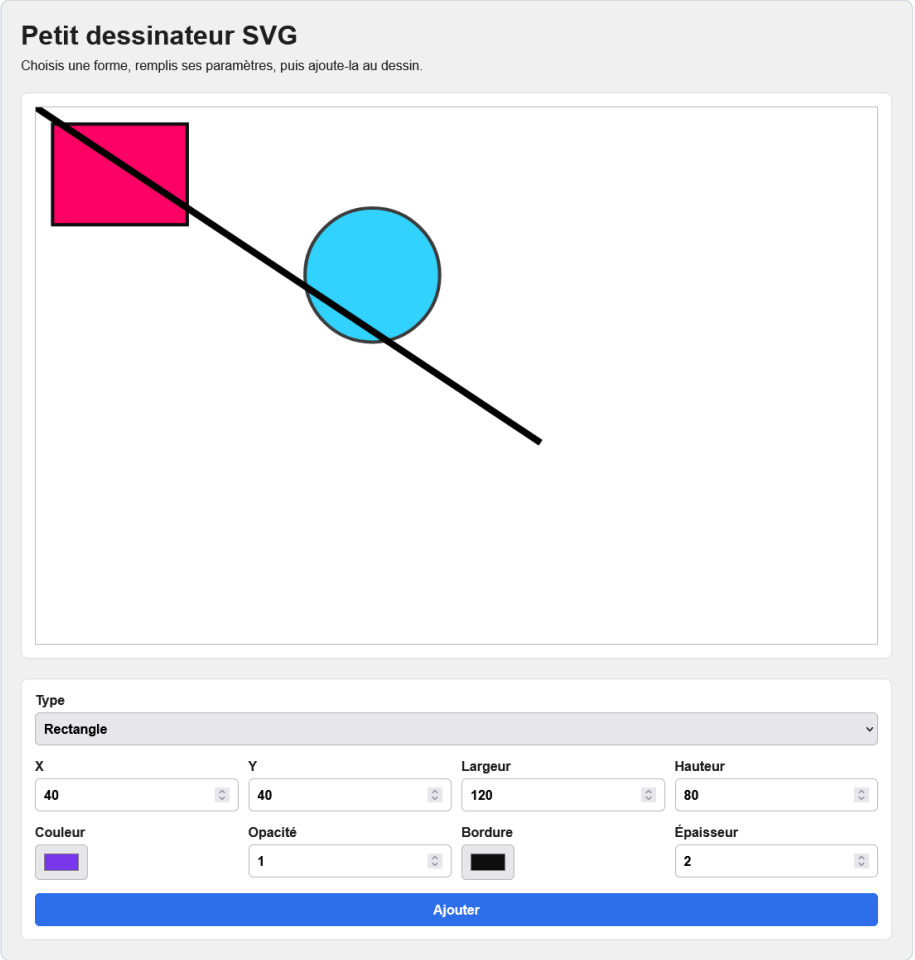
1. Affiche les 3 formes initiales dans un `<svg>` en bouclant sur le tableau (utilise un `if` ou un `switch` selon le `type` pour générer la bonne balise SVG).
2. Crée un **formulaire** où l'utilisateur choisit :
   - le **type** de forme (rectangle, cercle, ligne) via un `<select>`
   - les **paramètres correspondants** (position, taille, couleur)
3. Au clic sur "Ajouter", push la nouvelle forme dans `formes` et **redessine tout le SVG**.

💡 **Astuce UX** : affiche/cache les inputs selon le type sélectionné (un cercle n'a pas besoin de `largeur` / `hauteur`).

Bonus 💡 : ajoute une **opacité** et une **bordure** (couleur + épaisseur) à toutes les formes.

- Pistes : attributs SVG `opacity`, `stroke`, `stroke-width`

- MDN — SVG attribute reference



Récap

À la fin de ce chapitre, tu sais :

- ✅ Créer un objet et lui donner des propriétés
- ✅ Accéder à ses propriétés (point ou crochets) et les modifier
- ✅ Définir des méthodes et utiliser `this`
- ✅ Imbriquer des objets et des tableaux pour modéliser des données complexes
- ✅ Distinguer un objet JavaScript d'un texte JSON
- ✅ Convertir entre objet et JSON avec `JSON.stringify` et `JSON.parse`
- ✅ Importer un fichier JSON dans un projet Vite

2.6 🥭 Méthodes sur les tableaux en JavaScript

Qu'est-ce qu'une méthode sur un tableau ?

Une méthode sur un tableau est une fonction intégrée en JavaScript qui simplifie la manipulation des éléments. Ces méthodes remplacent avantageusement les boucles traditionnelles en rendant le code plus court, lisible et élégant.

Fonctions anonymes et fonctions fléchées

Fonction anonyme

Une fonction anonyme est une fonction sans nom utilisée généralement une seule fois, par exemple en tant qu'argument dans une autre fonction.

```javascript
// Fonction anonyme classique
arr.forEach(function(el) {
  console.log(el);
});
```

On peut également stocker une fonction anonyme dans une variable pour la réutiliser :

```javascript
const afficherElement = function(el) {
  console.log(el);
};

arr.forEach(afficherElement);
```

Fonction fléchée (Arrow Function)

C'est une version raccourcie, très utilisée en JavaScript moderne.

- Si la fonction fléchée tient sur une seule ligne, le `return` est implicite :

```javascript
const double = num => num * 2;
console.log(double(4));
// Output: 8
```

- Si la fonction fléchée contient plusieurs lignes, elle nécessite des accolades `{}` et un `return` explicite :

```javascript
const doubleEtLog = num => {
  const resultat = num * 2;
  console.log(resultat);
  return resultat;
};

doubleEtLog(4);
// Output: 8
```

Comme les fonctions anonymes classiques, on peut stocker les fonctions fléchées dans des variables pour les réutiliser :

```javascript
const afficherElement = el => console.log(el);

arr.forEach(afficherElement);
```

Principales méthodes sur les tableaux illustrées avec des emojis

1. forEach() — 🏃💨

Parcourt chaque élément sans rien retourner.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
fruits.forEach(fruit => console.log(fruit));
// Output:
// 🍎
// 🍌
// 🍇
```

2. map() — 🪞📦

Transforme chaque élément et retourne un nouveau tableau.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
const smoothies = fruits.map(fruit => fruit + "🥤");
console.log(smoothies);
// Output: ["🍎🥤", "🍌🥤", "🍇🥤"]
```

3. filter() — 🧹🗑️

Garde uniquement les éléments qui respectent une condition.

```javascript
const animals = ["🐶", "🐱", "🐶", "🐭"];
const dogsOnly = animals.filter(animal => animal === "🐶");
console.log(dogsOnly);
// Output: ["🐶", "🐶"]
```

4. find() — 🔍🔑

Renvoie le premier élément correspondant à une condition.

```javascript
const animals = ["🐶", "🐱", "🐭"];
const cat = animals.find(animal => animal === "🐱");
console.log(cat);
// Output: 🐱
```

5. some() — 🤔❓

Vérifie si au moins un élément respecte une condition.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
const hasBanana = fruits.some(fruit => fruit === "🍌");
console.log(hasBanana);
// Output: true
```

6. every() — ✅📏

Vérifie si tous les éléments respectent une condition.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
const allFruits = fruits.every(fruit => fruit.includes("🍌"));
console.log(allFruits);
// Output: false
```

7. reduce() — 🍲➕

Combine tous les éléments en un seul résultat.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
const fruitSalad = fruits.reduce((acc, fruit) => acc + fruit, "🥗:");
console.log(fruitSalad);
// Output: 🥗:🍎🍌🍇
```

8. includes() — 📦❓

Vérifie si un élément est présent dans le tableau.

```javascript
const animals = ["🐶", "🐱", "🐭"];
console.log(animals.includes("🐱"));
// Output: true
```

Utiliser ces méthodes vous permet d'écrire un code plus concis et facile à maintenir, tout en gardant votre style moderne et efficace.

🧩 Exercices pratiques sur les méthodes d'arrays en JS

Exercice 1 : Adieu les boucles, bonjour les méthodes ! 👋

Convertissez chacune des boucles suivantes en utilisant la méthode appropriée (`map` , `filter` , `find` , etc.) :

1. Créez un nouveau tableau contenant chaque fruit avec un "🍰" à côté :

```javascript
const fruits = ["🍎", "🍌", "🍓"];
const newFruits = [];
for(let i = 0; i < fruits.length; i++){
  newFruits.push(fruits[i] + "🍰");
}
```

2. Gardez uniquement les chats "😸" du tableau suivant :

```javascript
const animals = ["🐱", "🐶", "🐱", "🐭", "🐶"];
const cats = [];
for(let i = 0; i < animals.length; i++){
  if(animals[i] === "🐱"){
    cats.push(animals[i]);
  }
}
```

Exercice 2 : Debugging express 🚨

Expliquez pourquoi le code suivant ne fonctionne pas correctement, puis corrigez-le :

```javascript
const fruits = ["🍎", "🍌", "🍓"];
const juices = fruits.map(fruit => {
  fruit + "🥤";
});
console.log(juices);
```

Exercice 3 : Comparaison de performances 🏎️💨

Générez un tableau contenant plusieurs milliers d'éléments aléatoires (par exemple des nombres). Comparez ensuite les performances (temps d'exécution) des différentes approches suivantes pour traiter ces éléments :

- Utilisation des méthodes modernes vues précédemment :
  - utilisez `.map()` et multipliez les nombres par 2.
  - utilisez la méthode `sort`.
- Utilisation d'une boucle classique (`for`) :
  - Pour multiplier les nombres par 2.
  - Pour trier le tableau.
- Utilisation d'une méthode avec une fonction anonyme stockée dans une variable.

Mesurez précisément le temps pris par chaque méthode et expliquez vos observations.

🧑‍💻 Instructions pour mesurer le temps en JS :

Utilisez `console.time("label")` et `console.timeEnd("label")` pour mesurer le temps d'exécution de vos opérations en JavaScript. Exemple :

```javascript
console.time("exemple");
const result = [1, 2, 3].map(num => num * 3);
console.timeEnd("exemple");
```

🔗 Chaînage des méthodes sur les tableaux

Le chaînage (ou *chaining*) est une technique puissante en JavaScript qui consiste à appeler plusieurs méthodes successivement sur un même tableau, en une seule ligne, pour le traiter efficacement et de manière concise.

Cela fonctionne car chaque méthode retourne elle-même un tableau, sur lequel on peut immédiatement appeler une autre méthode.

Exemple concret :

Prenons un exemple simple avec un tableau d'emojis d'animaux :

```javascript
const animaux = ["🐱", "🐶", "🐭", "🐰", "🐻", "🐼"];

// Chaînage de méthodes : filter puis map
const resultat = animaux
  .filter(animal => animal !== "🐭")   // On enlève la souris 🐭
  .map(animal => animal + "❤️");       // On ajoute un cœur ❤️

console.log(resultat);
// Output: ["🐱❤️", "🐶❤️", "🐰❤️", "🐻❤️", "🐼❤️"]
```

Grâce au chaînage, le traitement du tableau est clair, compact et rapide à comprendre.

🚀 Exercices de chaînage

Voici quelques exercices progressifs pour tester votre compréhension du chaînage des méthodes.

🟢 Exercice 1 : Smoothie express 🍹

À partir du tableau de fruits suivant :

```javascript
["🍎", "🍌", "🍒", "🍓", "🍑"]
```

- Retirez les bananes (🍌) du tableau.
- Transformez les fruits restants en smoothies (en ajoutant 🥤).
- Affichez le résultat final dans la console.

🟠 Exercice 2 : Panier bio 🌿

À partir d'un tableau contenant des fruits frais et pourris :

```javascript
["🍎", "🍌", "🍎💩", "🍌💩", "🍓", "🍓💩"]
```

- Filtrez uniquement les fruits frais (ceux qui ne contiennent pas l'emoji 💩).
- Vérifiez ensuite si tous les fruits restants sont des fraises (🍓).
- Affichez dans la console le résultat final (true ou false).

🔴 Exercice 3 : Super salade 🥗

À partir d'un tableau contenant divers aliments :

```javascript
["🥕", "🍔", "🥦", "🍟", "🍅", "🍩", "🥑", "🍕"]
```

- Filtrez uniquement les aliments considérés comme « sains » : 🥕, 🥦, 🍅, 🥑.
- Transformez chaque aliment sain en ajoutant un emoji salade 🥗 à côté.
- Fusionnez-les ensuite en une seule chaîne avec `reduce()` pour obtenir une « super salade ».
- Affichez le résultat final dans la console.

🌶️ Exercice bonus (très spicy) : Animal Crossing 🐾

À partir du tableau suivant d'animaux :

```javascript
["🐱", "🐶", "🐭", "🐱", "🐰", "🐶", "🐻", "🐭"]
```

- Filtrez uniquement les animaux uniques (qui apparaissent une seule fois dans le tableau).

- Convertissez chaque animal unique en ajoutant à côté son nom écrit en toutes lettres (ex : `"🐱 Chat"`).
- Vérifiez ensuite si parmi ces animaux uniques il y a au moins un lapin (🐰).
- Affichez enfin l'ensemble des résultats obtenus clairement dans la console.

🧰 Autres méthodes utiles sur les tableaux en JavaScript

1. `concat()` — Fusionne des tableaux

Description : Combine plusieurs tableaux en un seul tableau, sans modifier les originaux.

```javascript
const fruits = ["🍎", "🍌"];
const veggies = ["🥦", "🥕"];
const mix = fruits.concat(veggies);
console.log(mix); // ["🍎", "🍌", "🥦", "🥕"]
```

2. `slice()` — Extrait une portion du tableau

Description : Crée une copie superficielle d'une portion du tableau, sans modifier l'original.

```javascript
const fruits = ["🍎", "🍌", "🍇", "🍓"];
const someFruits = fruits.slice(1, 3);
console.log(someFruits); // ["🍌", "🍇"]
```

3. `splice()` — Modifie le tableau original

Description : Permet d'ajouter, de supprimer ou de remplacer des éléments directement dans le tableau.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
fruits.splice(1, 1, "🍓");
console.log(fruits); // ["🍎", "🍓", "🍇"]
```

4. `sort()` — Trie les éléments du tableau

Description : Trie les éléments du tableau selon leur ordre alphabétique ou numérique.

```javascript
const numbers = [3, 1, 4, 2];
numbers.sort();
console.log(numbers); // [1, 2, 3, 4]
```

5. `reverse()` — Inverse l'ordre des éléments

Description : Inverse l'ordre des éléments directement dans le tableau original.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
fruits.reverse();
console.log(fruits); // ["🍇", "🍌", "🍎"]
```

6. `fill()` — Remplit le tableau avec une valeur spécifique

Description : Remplit tout ou partie du tableau avec une même valeur.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
fruits.fill("🍓", 1, 2);
console.log(fruits); // ["🍎", "🍓", "🍇"]
```

---

7. `copyWithin()` — Copie une portion du tableau à une autre position

Description : Copie une partie du tableau dans une autre position du même tableau, sans en modifier la taille.

```javascript
const fruits = ["🍎", "🍌", "🍇", "🍓"];
fruits.copyWithin(1, 2);
console.log(fruits); // ["🍎", "🍇", "🍓", "🍓"]
```

---

8. `flat()` — Aplatit les tableaux imbriqués

Description : Aplatit les tableaux imbriqués jusqu'à la profondeur spécifiée.

```javascript
const nested = ["🍎", ["🍌", ["🍇"]]];
console.log(nested.flat(2)); // ["🍎", "🍌", "🍇"]
```

---

9. `flatMap()` — Combine map() et flat()

Description : Applique une fonction à chaque élément puis aplatit le résultat d'un niveau.

```javascript
const numbers = [1, 2, 3];
const doubled = numbers.flatMap(n => [n, n * 2]);
console.log(doubled); // [1, 2, 2, 4, 3, 6]
```

---

10. `join()` — Crée une chaîne à partir des éléments du tableau

Description : Assemble tous les éléments d'un tableau en une chaîne de caractères, séparés par un délimiteur.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
console.log(fruits.join(" + ")); // "🍎 + 🍌 + 🍇"
```

---

11. `indexOf()` — Retourne l'index du premier élément trouvé

Description : Trouve l'index de la première occurrence d'un élément donné ou renvoie `-1` s'il n'existe pas.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
console.log(fruits.indexOf("🍌")); // 1
```

---

12. `lastIndexOf()` — Retourne le dernier index trouvé d'un élément

Description : Trouve l'index de la dernière occurrence d'un élément donné ou renvoie `-1` s'il n'existe pas.

```javascript
const fruits = ["🍎", "🍌", "🍇", "🍌"];
console.log(fruits.lastIndexOf("🍌")); // 3
```

---

13. `findIndex()` — Retourne l'index selon une condition

Description : Trouve l'index du premier élément respectant une condition définie.

```javascript
const numbers = [1, 3, 5, 8, 9];
console.log(numbers.findIndex(num => num > 5)); // 3
```

---

14. `toString()` — Convertit le tableau en chaîne de caractères

Description : Convertit automatiquement le tableau en une chaîne séparée par des virgules.

```javascript
const fruits = ["🍎", "🍌", "🍇"];
console.log(fruits.toString()); // "🍎,🍌,🍇"
```

2.7 Templating

Introduction

Le templating est une pratique essentielle dans le développement web, permettant de séparer la structure du contenu, de réutiliser des morceaux de code, et d'améliorer la maintenance de votre projet. En utilisant Vanilla JS et Vite, nous allons apprendre à créer des templates pour des éléments communs tels que la navigation et le footer, ainsi qu'à dynamiser des composants réutilisables comme une "card" pour afficher une liste d'utilisateurs.

Pour structurer votre application Vite avec un seul fichier HTML (`index.html`) et plusieurs composants JavaScript dynamiques, nous allons organiser les composants dans un dossier `components` et les intégrer via `main.js`. Voici comment procéder :

Structure de Projet

- **index.html** : Le point d'entrée de votre application.
- **/components** : Dossier contenant les fichiers JS pour chaque composant (`nav.js`, `footer.js`, `card.js`). Vous devez le créer dans votre projet.

`index.html`

Votre fichier `index.html` reste comme défini, servant de conteneur pour les composants injectés dynamiquement :

```xml
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <link rel="icon" type="image/svg+xml" href="/vite.svg" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Vite App</title>
</head>
<body>
  <div id="app"></div>
  <script type="module" src="/main.js"></script>
</body>
</html>
```

Composants JavaScript

Navigation (`components/nav.js`)

```javascript
export function renderNav() {
  return `
    <nav>
      <ul>
        <li><a href="/">Accueil</a></li>
        <li><a href="/about">À Propos</a></li>
      </ul>
    </nav>
  `;
}
```

Footer (`components/footer.js`)

```javascript
export function renderFooter() {
  return `
    <footer>
      <p>&copy; 2024 Mon Site Web</p>
    </footer>
  `;
}
```

Card (`components/card.js`)

```javascript
export function renderCard(user) {
  return `
    <div class="card">
      <img src="user.image"alt="{user.image}" alt="user.image"alt="{user.name}" class="card-img">
      <h3 class="card-title">user.name</h3><pclass="card-info">{user.name}</h3>
      <p class="card-info">user.name</h3><pclass="card-info">{user.info}</p>
    </div>
  `;
}
```

Intégration dans `main.js`

Dans `main.js`, importez vos composants et utilisez-les pour construire le contenu de votre application :

```javascript
import { renderNav } from './components/nav.js';
import { renderFooter } from './components/footer.js';
import { renderCard } from './components/card.js';
import './style.css';
import users from './users.json';


const app = document.getElementById('app');

app.innerHTML = `
  renderNav()<divid="cards-container"></div>{renderNav()}
  <div id="cards-container"></div>
  renderNav()<divid="cards-container"></div>{renderFooter()}
`;

document.getElementById('cards-container').innerHTML = users.map(user => renderCard(user)).join('');
```

Style CSS (`style.css`)

Ajoutez vos styles CSS dans `style.css` pour styliser vos composants de navigation, de footer et de card comme montré précédemment.

```css
/* Style global */
body,
html {
  margin: 0;
  padding: 0;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
  background-color: #f5f5f5;
  color: #333;
}

a {
  color: #007bff;
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}

#app {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

/* Navigation */
nav {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

nav ul {
  list-style: none;
  padding: 0;
  display: flex;
  justify-content: center;
  margin: 0;
}

nav ul li {
  padding: 1rem;
}

/* Cards */
#cards-container {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  gap: 20px;
  padding: 20px;
}

.card {
  background-color: #fff;
  border: 1px solid #ddd;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  width: 300px;
  overflow: hidden;
}

.card-img {
  width: 100%;
  height: 200px;
  object-fit: cover;
```

```
}

.card-title {
  margin: 10px;
  font-size: 20px;
}

.card-info {
  margin: 0 10px 10px 10px;
  font-size: 16px;
  color: #666;
}

/* Footer */
footer {
  margin-top: auto;
  background-color: #fff;
  box-shadow: 0 -2px 4px rgba(0, 0, 0, 0.1);
  padding: 1rem;
  text-align: center;
}
```

En suivant cette structure, vous créez une application modulaire où chaque composant est géré séparément, rendant votre code plus propre, réutilisable et facile à maintenir.

Nous devons encore créer le fichier json avec les données des utilisateurs et ajouter leurs avatars dans le dossier public. Dans le json, `user.image` contiendra le chemin vers l'avatar.

2.8 Hébergement

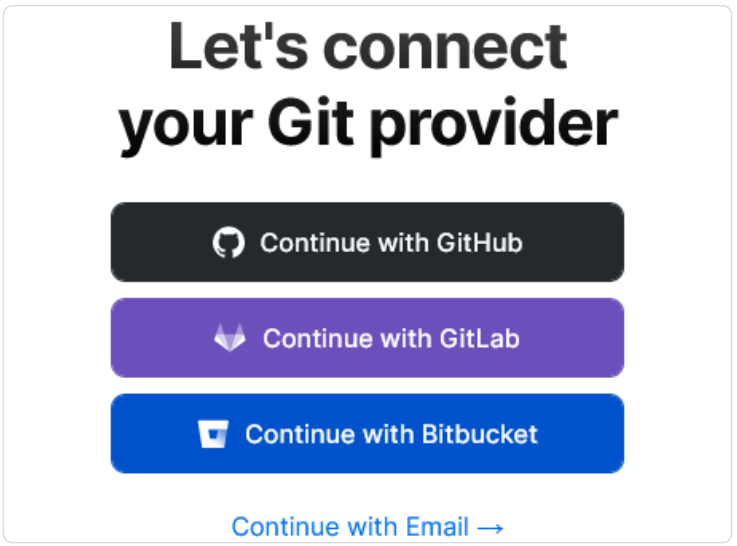
Hébergement avec Vercel

Dans cette section, nous allons apprendre comment héberger notre projet en utilisant Vercel. Vercel est une plateforme d'hébergement qui permet de déployer facilement des applications web.

Étape 1: Créer un compte Vercel

Tout d'abord, nous devons créer un compte Vercel. Pour cela, suivez les étapes suivantes :

1. Rendez-vous sur le site web de Vercel à l'adresse vercel.com.
2. Cliquez sur le bouton "Sign Up" pour créer un nouveau compte.
3. Choisissez l'option de connexion avec GitHub pour lier votre compte GitHub à Vercel.
4. Suivez les instructions pour autoriser Vercel à accéder à votre compte GitHub.



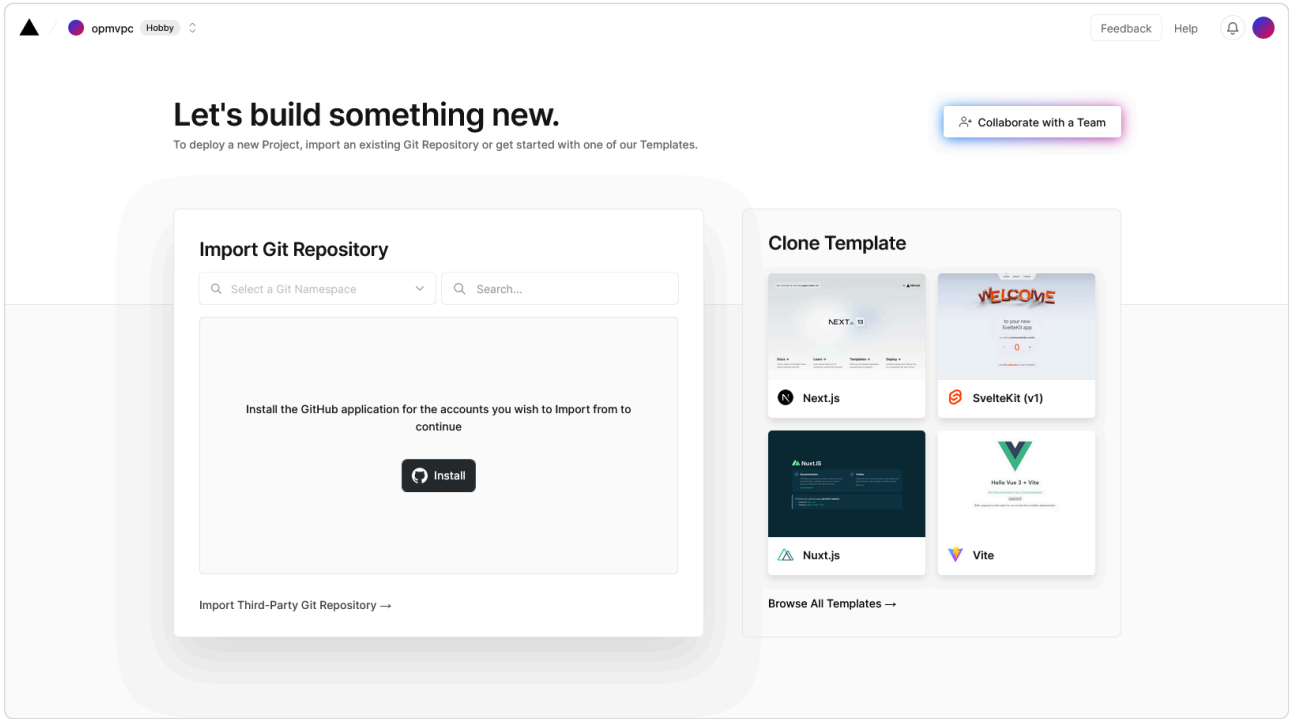
Étape 2: Sauvegarder le projet sur GitHub

Maintenant que nous avons créé notre compte Vercel, nous allons sauvegarder notre projet sur GitHub.

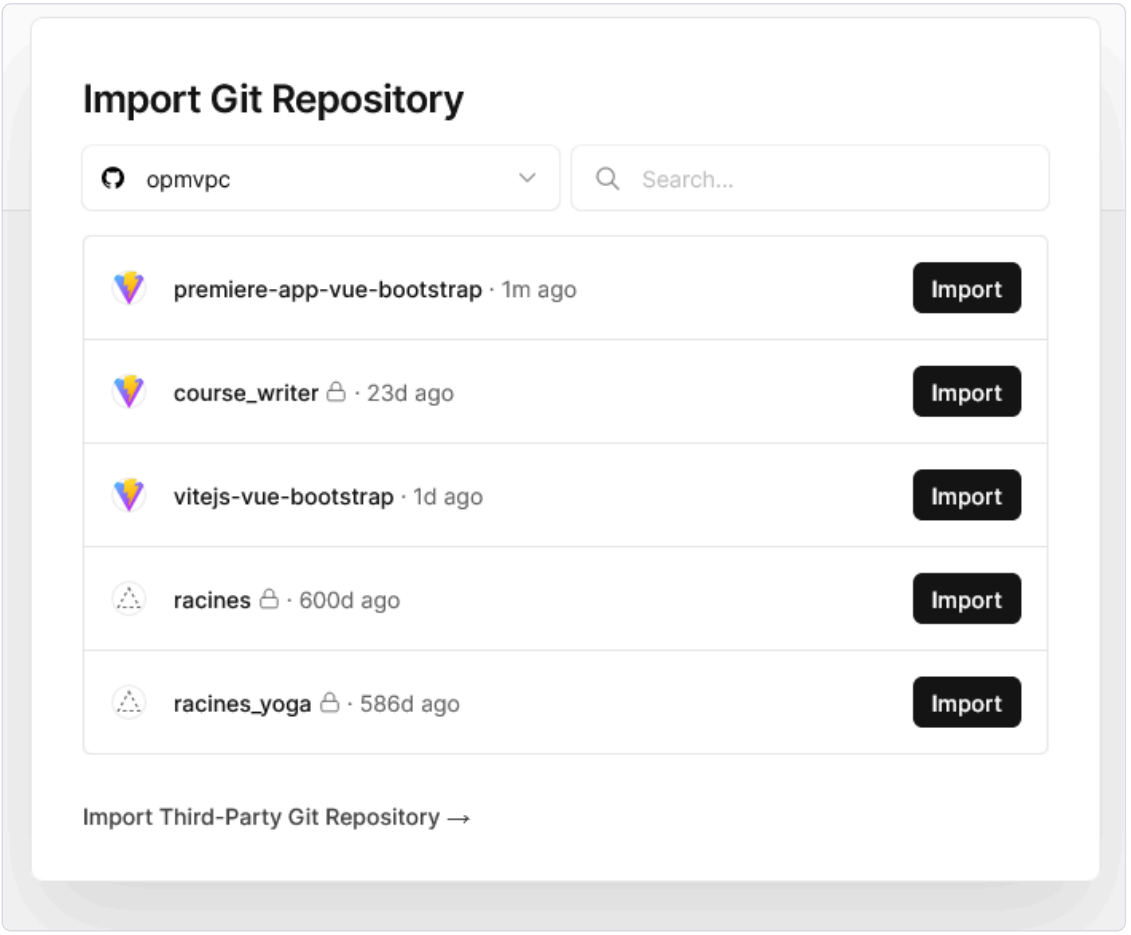
Étape 3: Lier votre compte GitHub à Vercel

Maintenant, nous allons lier notre compte GitHub à Vercel. Voici les étapes à suivre :

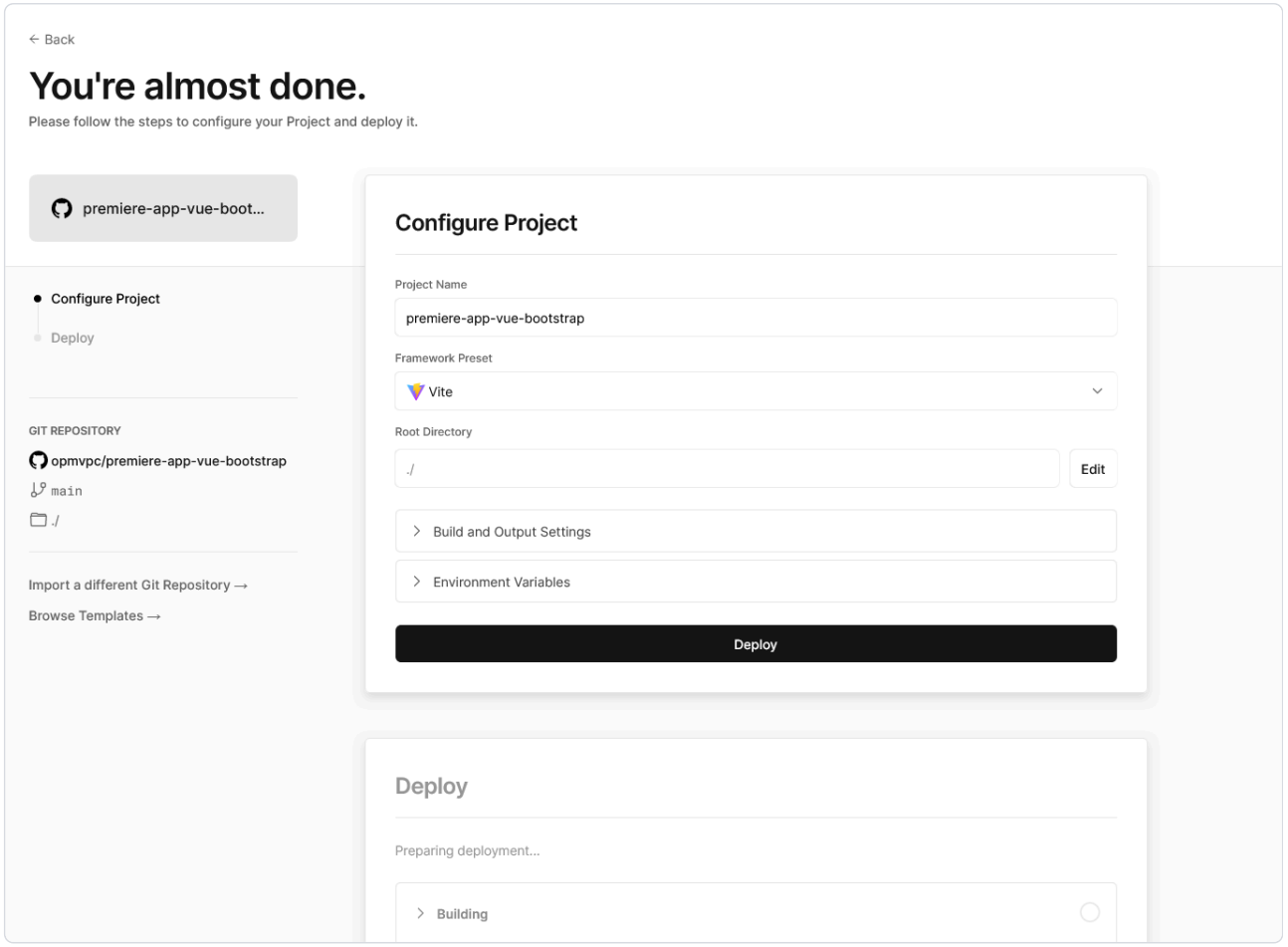
1. Retournez sur le site web de Vercel.
2. Cliquez sur le bouton "Install" pour autoriser Vercel à accéder à votre compte GitHub.



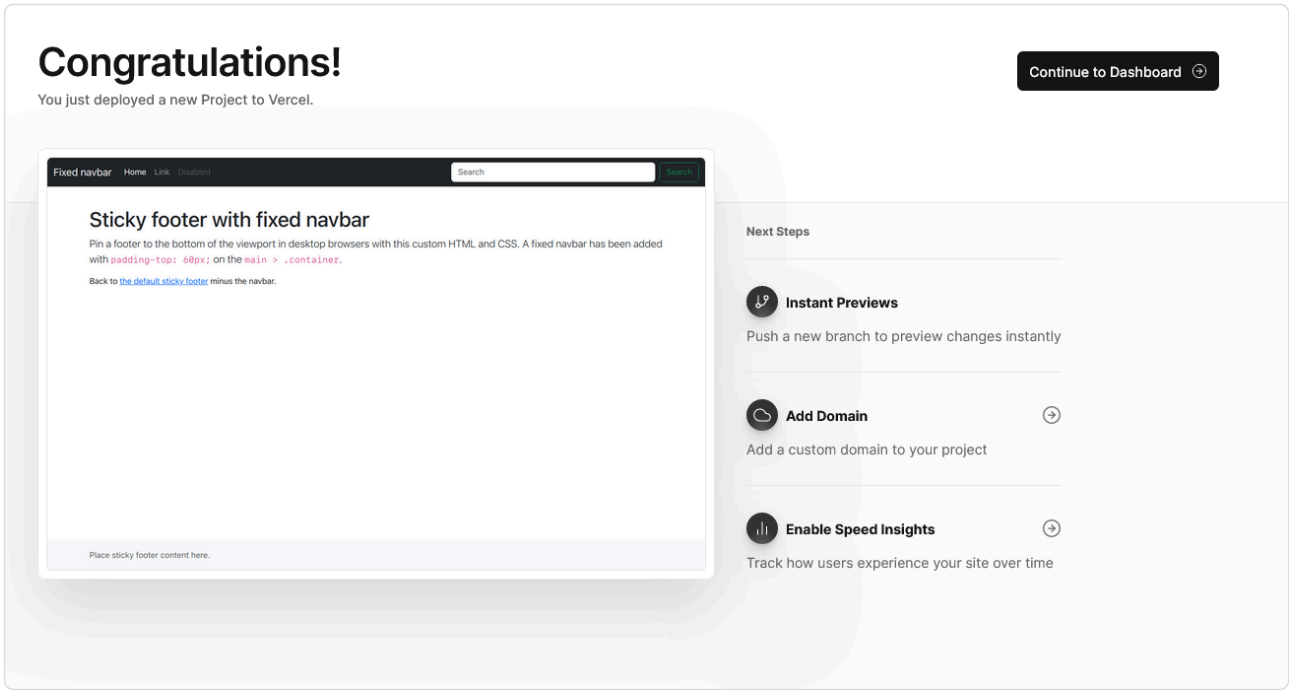
3. Une fois installé, vous verrez une liste de vos projets GitHub.



4. Cliquez sur le bouton "Import" du projet que vous voulez héberger.
5. Vous serez redirigé vers la page de configuration de votre projet.



6. Vercel détectera automatiquement que vous utilisez Vite.
7. Cliquez sur le bouton "Deploy" pour déployer votre projet.
8. Attendez quelques secondes et normalement, si tout s'est bien passé, vous pourrez accéder à votre site en cliquant sur le lien de prévisualisation fourni par Vercel.



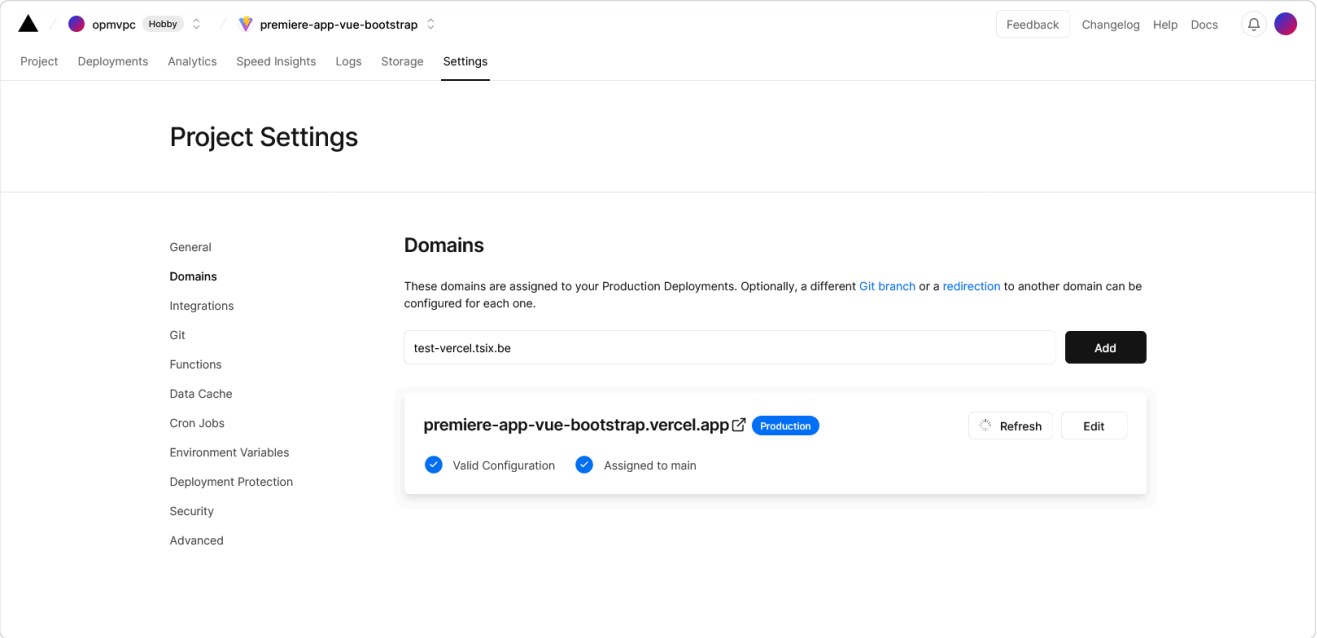
Et voilà ! Vous avez maintenant hébergé votre projet sur Vercel. Vous pouvez partager le lien de prévisualisation avec d'autres personnes pour qu'elles puissent voir votre site en ligne.

Chaque fois que vous ferez des modifications sur cette application, la nouvelle version sera automatiquement mise en ligne.

Nom de domaine

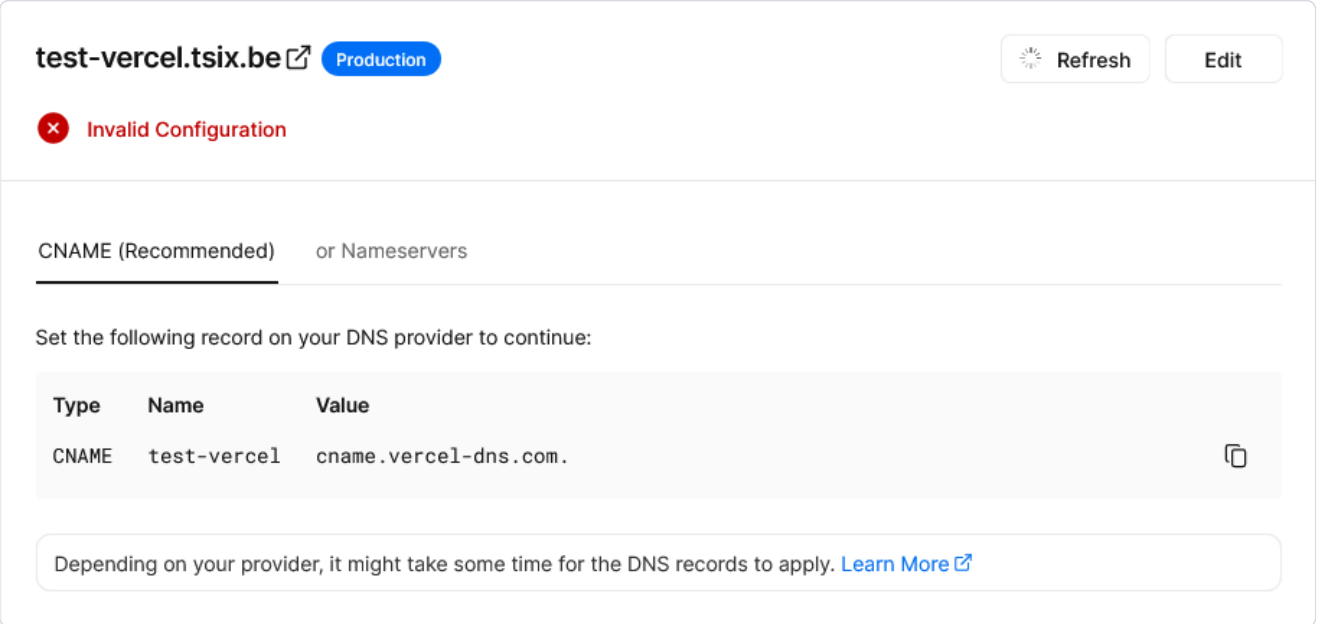
Vous pouvez lier votre nom de domaine à un site hébergé sur Vercel.

Pour cela, allez dans les paramètres de votre projet, puis dans l'onglet "Domains".



Ajoutez votre nom de domaine avec un sous domaine, puis cliquez sur "Add".

Nous devons ensuite suivre les instructions.



Allez dans l'interface de gestion de votre nom de domaine et ajoutez un champs CNAME avec les paramètres fournis.



Nous pouvons désormais accéder à notre application avec notre nom de domaine. Vercel émet automatiquement un certificat SSL pour notre site.

Url de base :
https://premiere-app-vue-bootstrap.vercel.app/
Url utilisant le domaine :
https://test-vercel.tsix.be/

2.9 Projet Morpion OXO — ton premier jeu en JavaScript

Tu vas programmer le morpion (tic-tac-toe ou encore OXO) : deux joueurs posent tour à tour leurs pions sur une grille 3 × 3, le premier qui aligne trois symboles gagne. Puis tu iras plus loin : ton programme jouera **contre** toi.

C'est le projet idéal pour apprendre à construire un jeu, parce que toute la difficulté est dans la **logique**, pas dans la technique. En chemin, tu vas pratiquer :

- la manipulation du DOM (créer du HTML depuis JavaScript, réagir aux clics)
- les tableaux 2D (la grille de jeu)
- la séparation entre l'**état** du jeu et son **affichage** — LE concept de ce chapitre
- la logique de jeu : victoire, match nul, tours de jeu
- tes premiers **algorithmes d'IA**

Temps estimé : 4 à 6 heures, selon jusqu'où tu pousses l'IA.

Comment travailler

L'énoncé est découpé en étapes. Chaque étape te donne un **objectif**, des **explications sur les concepts nécessaires**, et parfois un **coup de pouce**. On ne te donne pas le code : à toi de le trouver. Mais si tu suis les étapes dans l'ordre, chacune est un petit pas faisable.

Règle d'or : **ne passe pas à l'étape suivante tant que l'actuelle ne fonctionne pas**. Teste après chaque modification, et utilise `console.log` dès que tu ne comprends pas ce qui se passe.

Le point de départ

Crée un projet Vite (vanilla JS). Remplace le contenu de `main.js` par ceci :

JAVASCRIPT

```javascript
import "./style.css";

const app = document.querySelector("#app");

app.innerHTML = `
  <h1>Morpion</h1>
  <div id="message"></div>
  <div id="jeu"></div>
  <div>
    <button id="recommencer">Recommencer</button>
  </div>
`;

let board = [];
let joueurActuel = "";

function initialiserBoard() {
  return [
    ["", "", ""],
    ["", "", ""],
    ["", "", ""],
  ];
}

function choisirPremierJoueur() {
  return Math.random() < 0.5 ? "X" : "O";
}

function afficherBoard(board) {
  const jeu = document.querySelector("#jeu");
  jeu.innerHTML = `
    <div class="board-grid">
      <div class="case" data-row="0" data-col="0">board[0][0]</div><divclass="case"data-row="0"data-col="1">
{board[0][0]}</div>
      <div class="case" data-row="0" data-col="1">board[0][0]</div><divclass="case"data-row="0"data-col="1">
{board[0][1]}</div>
      <div class="case" data-row="0" data-col="2">board[0][2]</div><divclass="case"data-row="1"data-col="0">
{board[0][2]}</div>
      <div class="case" data-row="1" data-col="0">board[0][2]</div><divclass="case"data-row="1"data-col="0">
{board[1][0]}</div>
      <div class="case" data-row="1" data-col="1">board[1][1]</div><divclass="case"data-row="1"data-col="2">
{board[1][1]}</div>
      <div class="case" data-row="1" data-col="2">board[1][1]</div><divclass="case"data-row="1"data-col="2">
{board[1][2]}</div>
      <div class="case" data-row="2" data-col="0">board[2][0]</div><divclass="case"data-row="2"data-col="1">
{board[2][0]}</div>
      <div class="case" data-row="2" data-col="1">board[2][0]</div><divclass="case"data-row="2"data-col="1">
{board[2][1]}</div>
      <div class="case" data-row="2" data-col="2">${board[2][2]}</div>
    </div>
  `;
}

function demarrerJeu() {
  board = initialiserBoard();
  joueurActuel = choisirPremierJoueur();
  afficherBoard(board);
}

// lancement du programme
demarrerJeu();
```

Et le contenu de `style.css` par ceci :

```css
.board-grid {
  display: grid;
  grid-template-columns: auto auto auto;
  gap: 10px;
  width: fit-content;
}

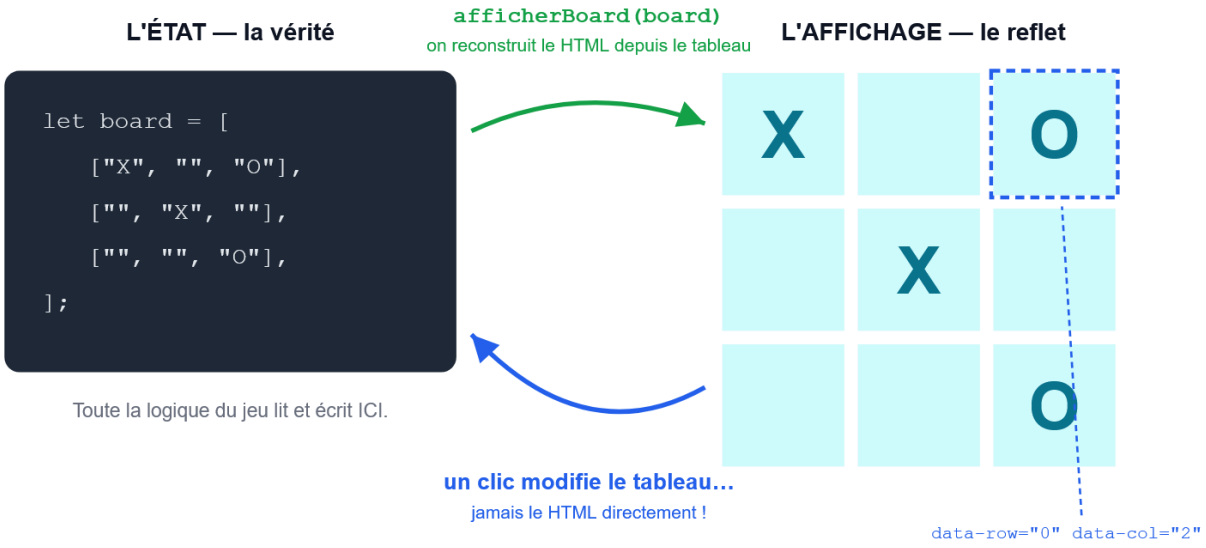
.board-grid > div {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100px;
  width: 100px;
  background-color: aqua;
  font-size: 3rem;
  cursor: pointer;
}
```

Lance le projet : une grille de 9 cases vides s'affiche. Elle ne réagit pas encore aux clics — c'est tout l'objet de l'étape 1.

Concept clé : l'état et l'affichage

Schéma 1 — L'état du jeu et son affichage

Le tableau `board` est la vérité. Le HTML n'est que son reflet.



Le cycle : clic → on modifie board → afficherBoard() → l'écran rattrape la réalité.

Regarde bien le code de départ : il y a **deux mondes**.

Le premier, c'est l'**état du jeu** : le tableau `board`, un tableau 2D de 3 lignes × 3 colonnes. C'est lui la **vérité** : qui a joué où, quelles cases sont libres. Toute la logique du jeu (peut-on jouer ici ? quelqu'un a-t-il gagné ?) interroge ce tableau, et lui seul.

Le second, c'est l'**affichage** : le HTML que voit le joueur. Il n'est qu'un **reflet** de l'état : la fonction `afficherBoard(board)` lit le tableau et reconstruit le HTML qui correspond.

Le cycle de vie d'un coup sera donc toujours le même : un clic **modifie le tableau** `board`, puis on appelle `afficherBoard(board)` pour que l'écran rattrape la réalité. On ne modifie **jamais** le HTML directement, et on ne lit **jamais** le contenu du HTML pour savoir où en est la partie.

C'est le concept le plus important du chapitre. Il te resservira dans tous tes projets, jeux ou pas : tout Vue.js, que tu verras plus tard, est construit sur cette idée.

Vérifie que tu as compris (réponds avant de continuer) :

1. Que contient `board[1][2]` ? Quelle case de la grille est-ce à l'écran ?
2. À quoi servent les attributs `data-row` et `data-col` dans le HTML, à ton avis ?
3. Dans la console du navigateur, que donne `document.querySelector(".case").textContent` après quelques coups ? Pourquoi ne s'en sert-on pas pour la logique ?

Étape 1 — Poser un pion au clic

Objectif : cliquer sur une case vide y place le pion du joueur actuel, et les joueurs alternent (X, O, X, O...).

Les concepts qu'il te faut

Sélectionner et écouter. `querySelectorAll(".case")` te donne les 9 cases ; une boucle dessus permet d'attacher un écouteur de clic (`addEventListener`) à chacune.

Savoir quelle case a été cliquée. Dans l'écouteur, `event.currentTarget` est l'élément cliqué. Ses attributs `data-row` et `data-col` (lisibles avec `getAttribute("data-row")` ou via `dataset`) te donnent les coordonnées de la case dans `board`. Voilà à quoi ils servaient.

Alterner les joueurs. Écris une petite fonction `joueurSuivant()` : si `joueurActuel` vaut `"X"`, il devient `"O"`, et inversement. Le code de départ contient déjà un opérateur ternaire dont tu peux t'inspirer.

> ⚠️ ***Le piège de l'*** *`innerHTML`. Quand* *`afficherBoard`* *réécrit* *`jeu.innerHTML`*, *les anciennes cases sont* ***détruites*** *et remplacées par des neuves — qui n'ont* ***aucun écouteur de clic****. Il faut donc ré-attacher les écouteurs après* ***chaque*** *affichage. Mets ta boucle d'écouteurs dans une fonction (par exemple* *`ajouterHandlerAuxCases()`*) *et appelle-la après chaque* *`afficherBoard`*. *Si ta grille ne répond plus après le premier coup, c'est ça.*

> 💡 ***Coup de pouce.*** *Dans l'écouteur de clic, dans l'ordre : lire les coordonnées de la case → vérifier que la case est* ***vide*** *dans* *`board`* *(sinon, ne rien faire) → écrire le pion dans* *`board`* *→ réafficher → ré-attacher les écouteurs → passer au joueur suivant.*

Validation : tu peux remplir toute la grille en alternant X et O, et rejouer sur une case occupée ne fait rien.

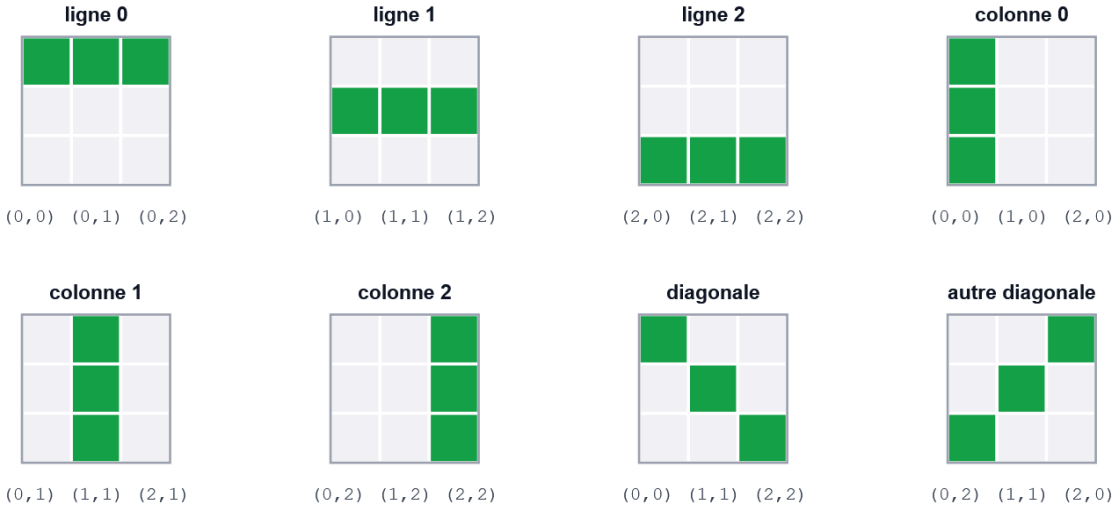
Étape 2 — Détecter la victoire

Objectif : dès qu'un joueur aligne trois pions, un message l'annonce et la partie s'arrête.

C'est l'étape star du projet. Au morpion, il n'existe que **8 façons de gagner** : 3 lignes, 3 colonnes, 2 diagonales. Pas besoin d'être malin — il suffit de vérifier les 8, à chaque coup.

Schéma 2 — Les 8 lignes gagnantes

Il n'existe que 8 façons de gagner : 3 lignes, 3 colonnes, 2 diagonales. On les vérifie toutes, à chaque coup.



Remarque : la case centrale (1,1) participe à 4 lignes, les coins à 3, les bords à 2 seulement.

a) La fonction de vérification

Écris une fonction `verifierVictoire(board)` qui renvoie `"X"` ou `"O"` s'il y a un gagnant, et `null` sinon. La renvoyer ainsi (plutôt qu'un simple `true` / `false`) te resservira pour l'IA, fais-moi confiance.

> 💡 ***Coup de pouce.*** *Liste les 8 lignes gagnantes dans un tableau, où chaque ligne est un trio de coordonnées :*
>
> ```javascript
> const lignesGagnantes = [
>   [[0, 0], [0, 1], [0, 2]], // ligne du haut
>   // ... à toi de compléter les 7 autres
> ];
> ```
>
> *Ensuite, une boucle sur ces 8 lignes : si les trois cases d'une ligne contiennent la **même valeur** et que cette valeur n'est **pas** `""`, c'est gagné. Vérifie ton tableau avec le schéma 2.*

b) Arrêter la partie

Quand quelqu'un gagne : affiche un message dans le `<div id="message">` (avec `textContent`), et fais en sorte qu'on ne puisse plus jouer.

> 💡 ***Coup de pouce.*** *Une variable d'état `partieTerminee` (un booléen) suffit : le clic ne fait plus rien quand elle est à `true`. C'est la même idée que les états d'un feu rouge : le code se comporte différemment selon la valeur d'une variable.*

Validation : fais gagner X sur une ligne, une colonne et une diagonale. À chaque fois, le message s'affiche et plus aucun coup n'est possible.

Étape 3 — Le match nul

Objectif : quand les 9 cases sont pleines sans gagnant, annoncer le match nul.

À toi de trouver comment savoir si la grille est pleine. Attention à l'ordre : vérifie la victoire **avant** le match nul — une partie peut se gagner au 9e coup !

Validation : remplir la grille sans gagnant affiche « Match nul » ; gagner au dernier coup affiche bien la victoire, pas le nul.

Étape 4 — Recommencer

Objectif : le bouton « Recommencer » relance une partie propre, à tout moment.

Le bouton existe déjà dans le HTML du départ, et la fonction `demarrerJeu()` aussi : il ne reste qu'à les connecter, et à vérifier que `demarrerJeu` remet bien **tout** à zéro.

> ⚠️ *Piège classique : tu recommences après une victoire et… impossible de jouer ? Une certaine variable d'état a échappé à ta remise à zéro.*

Validation : recommencer en pleine partie, après une victoire et après un match nul fonctionne à chaque fois.

Étape 5 — Une IA qui joue au hasard

Objectif : un mode solo : tu joues X, et l'ordinateur répond en posant son O sur une case vide **au hasard**.

Ton premier algorithme d'IA ! Il est volontairement bête, mais le jeu devient vivant : tu joues **contre** ton programme.

L'algorithme

1. Construis la liste de toutes les cases encore vides (un tableau d'objets `{ row, col }`, rempli en parcourant `board`).
2. Tires-en une au hasard :

```javascript
const indexAleatoire = Math.floor(Math.random() * casesVides.length);
```

3. Joue le pion de l'IA sur cette case, comme si elle avait cliqué : même logique, mêmes vérifications de victoire et de nul.

> 💡 ***Coup de pouce.*** *Range cette logique dans une fonction `jouerCoupIA()`, appelée juste après le coup du joueur humain — seulement si la partie n'est pas terminée ! Pour le confort, tu peux retarder le coup de l'IA de quelques centaines de millisecondes avec `setTimeout` : l'ordinateur a l'air de « réfléchir ».*

Validation : tu joues seul contre l'IA, elle ne joue que sur des cases vides, et la victoire/le nul fonctionnent toujours.

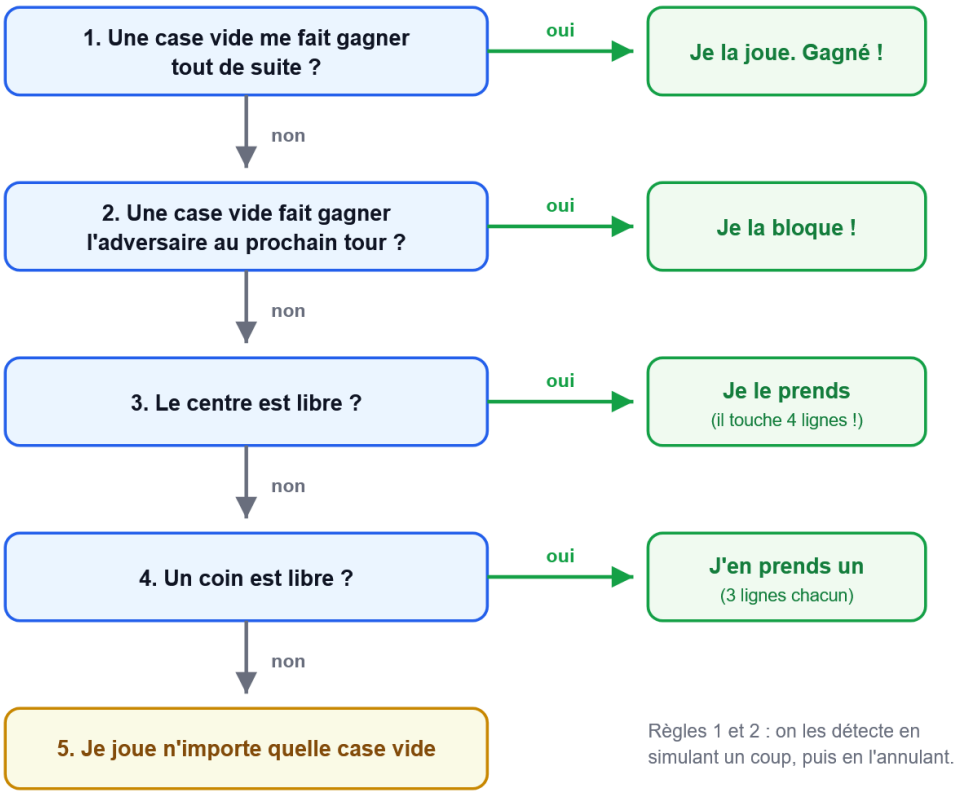
Étape 6 — Une IA qui réfléchit (avec des règles)

Objectif : remplacer le hasard par une stratégie. Ton IA devient difficile à battre.

Pas besoin d'algorithme savant : quelques règles de bon sens, appliquées **dans le bon ordre**, suffisent à faire une IA redoutable sur une grille 3 × 3.

Schéma 3 — Les priorités de l'IA à règles

À son tour, l'IA descend cette liste et s'arrête à la première règle qui s'applique.



Les priorités

1. **Gagner.** Si une case vide me donne la victoire immédiate, je la joue.
2. **Bloquer.** Si une case vide donne la victoire à l'adversaire au prochain tour, je la prends.
3. **Le centre.** S'il est libre, c'est la case la plus puissante (elle participe à 4 lignes gagnantes — recompte sur le schéma 2).
4. **Un coin.** Les coins participent à 3 lignes chacun, les bords à 2 seulement.
5. **Sinon**, n'importe quelle case vide (ton code de l'étape 5 sert encore !).

L'astuce qui change tout : la simulation

Comment savoir si une case « me donne la victoire » ? **Fais semblant d'y jouer.** Pour chaque case vide : pose ton pion dans `board`, appelle ta fonction `verifierVictoire(board)`, puis **remets la case à** `""`. Si la vérification a renvoyé ton symbole, cette case est gagnante. Tu vois pourquoi je t'ai fait renvoyer `"X"` / `"O"` / `null` plutôt qu'un booléen ? La même fonction sert à tester ta victoire **et** celle de l'adversaire.

> ⚠️ *N'oublie jamais d'**annuler** la simulation (remettre `""`), sinon ton IA pose des pions fantômes. Si ta grille se remplit toute seule, tu sais d'où ça vient.* 👻

> 💡 ***Coup de pouce.*** *Écris une fonction `trouverCaseGagnante(board, symbole)` qui renvoie la case gagnante pour ce symbole, ou `null`. Appelée avec le symbole de l'IA, elle sert pour la règle 1 ; appelée avec celui de l'adversaire, pour la règle 2. Une fonction, deux règles.*

Validation : l'IA gagne dès qu'elle le peut, bloque tes alignements, et préfère le centre puis les coins. Si tu joues parfaitement, la partie doit finir nulle.

Bonus

Bonus 1 — Choisir son mode de jeu

Un petit menu au lancement (ou des boutons) : « 2 joueurs » ou « contre l'ordinateur ». Une variable `modeDeJeu` et le tour de l'IA ne se déclenche que dans le bon mode.

Bonus 2 — Le score sur plusieurs manches

Compte les victoires de X, de O et les nuls, affichés en permanence. « Recommencer » garde les scores ; ajoute un bouton « Tout remettre à zéro ».

Bonus 3 — Mettre en valeur la ligne gagnante

À la victoire, colore les 3 cases gagnantes (une classe CSS ajoutée aux bonnes cases). Indice : ta boucle de `verifierVictoire` sait déjà **quelle** ligne a gagné — il suffit de la faire parler.

Pour aller plus loin : Minimax, l'IA invincible

Ton IA à règles est forte, mais on peut la coincer. Les IA parfaites des jeux à deux joueurs utilisent l'algorithme **Minimax** : explorer **tous** les coups possibles, puis tous les coups de réponse, et ainsi de suite jusqu'aux fins de partie, en supposant que chacun joue au mieux. Au morpion il n'y a que quelques milliers de parties possibles : un ordinateur les examine toutes en un clin d'œil, et devient littéralement imbattable. C'est un algorithme **récursif** (une fonction qui s'appelle elle-même) — un beau défi pour quand tu seras à l'aise. Cherche « minimax tic tac toe » ou commence par la page Wikipédia.

Ton plan de bataille

✓	Étape	Tu sauras que ça marche quand…
☐	Point de départ	la grille vide s'affiche
☐	Étape 1	X et O se posent en alternance au clic
☐	Étape 2	les 8 victoires sont détectées, la partie s'arrête
☐	Étape 3	grille pleine sans gagnant → match nul
☐	Étape 4	« Recommencer » marche dans toutes les situations
☐	Étape 5	tu joues contre une IA aléatoire
☐	Étape 6	l'IA gagne, bloque, et vise centre/coins
☐	Bonus	choix du mode, scores, ligne gagnante colorée

Derniers conseils

- **Une fonction = une responsabilité** : afficher, vérifier la victoire, jouer un coup, trouver une case gagnante… Si une fonction fait deux choses, coupe-la en deux.
- **Teste ta logique dans la console.** `verifierVictoire` se teste sans cliquer : construis un tableau gagnant à la main dans la console et appelle ta fonction dessus.
- `console.log(board)` est ton ami. L'affichage ment parfois (un oubli de `afficherBoard`), le tableau jamais.
- **Bloqué plus de 30 minutes sur le même problème ?** Cherche sur Google ou sur MDN, demande à une IA de t'**expliquer** ce qui coince (pas de coder à ta place — c'est toi qui apprends), pose ta question sur Teams, ou note-la pour le prochain cours.

Documentation utile

- `querySelector` / `querySelectorAll` (MDN) — sélectionner des éléments
- `addEventListener` (MDN) — réagir aux clics
- Les attributs `data-*` et `dataset` (MDN) — stocker les coordonnées dans le HTML
- Les template strings (MDN) — écrire le HTML de la grille
- L'opérateur ternaire (MDN) — alterner les joueurs en une ligne
- `Math.random()` (MDN) — l'aléatoire
- `setTimeout` (MDN) — faire « réfléchir » l'IA
- `textContent` (MDN) — afficher les messages

Bonne partie — et ne sois pas vexé le jour où ton propre programme te battra. C'est le but. 🤖

Énoncé rédigé en collaboration avec Claude Fable 5.

2.10 Projet Snake 🐍 — Crée ton jeu en JavaScript

Tu vas programmer le jeu du serpent : un serpent qui se déplace sur une grille, mange des fruits, grandit… et meurt s'il touche un mur ou sa propre queue.

C'est un vrai petit jeu vidéo, et tu vas le construire entièrement toi-même. En chemin, tu vas pratiquer :

- le dessin dans un `<canvas>`
- une boucle de jeu avec `setInterval`
- les événements clavier
- les tableaux d'objets (la structure de données du serpent)
- la logique de collision
- la manipulation du DOM pour l'interface (score, écrans)

Temps estimé : 4 à 8 heures, selon jusqu'où tu pousses les bonus.

Comment travailler

L'énoncé est découpé en étapes. Chaque étape te donne un **objectif**, des **explications sur les concepts nécessaires**, et parfois un **coup de pouce**. On ne te donne pas le code : à toi de le trouver. Mais si tu suis les étapes dans l'ordre, chacune est un petit pas faisable.

Règle d'or : **ne passe pas à l'étape suivante tant que l'actuelle ne fonctionne pas**. Teste après chaque modification, et utilise `console.log` dès que tu ne comprends pas ce qui se passe.

Le point de départ

Crée un projet Vite (vanilla JS) et remplace le contenu de `main.js` par ceci :

```javascript
import "./style.css";

const app = document.getElementById("app");

app.innerHTML = `
  <h1>Le jeu du serpent</h1>
  <canvas id="snake" width="400" height="400"></canvas>
`;

const canvas = document.getElementById("snake");
const ctx = canvas.getContext("2d");

const tailleCase = 20;                        // 1 case = 20 px
const nbColonnes = canvas.width / tailleCase; // 20 colonnes
const nbLignes = canvas.height / tailleCase;  // 20 lignes

const directions = {
  ArrowLeft:  { dx: -1, dy: 0 },
  ArrowUp:    { dx: 0, dy: -1 },
  ArrowRight: { dx: 1, dy: 0 },
  ArrowDown:  { dx: 0, dy: 1 },
};

// Pour l'instant, pas encore de serpent :
// juste une case verte qui se déplace.
let position = { x: 5, y: 5 };
let direction = { dx: 1, dy: 0 };

setInterval(() => {
  // 1. on déplace : la position avance d'une case
  position.x += direction.dx;
  position.y += direction.dy;

  // 2. on efface tout le canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // 3. on dessine la case à sa nouvelle position
  ctx.fillStyle = "green";
  ctx.fillRect(
    position.x * tailleCase,
    position.y * tailleCase,
    tailleCase,
    tailleCase
  );
}, 200);

document.addEventListener("keydown", (event) => {
  if (directions[event.code]) {
    direction = directions[event.code];
  }
});
```

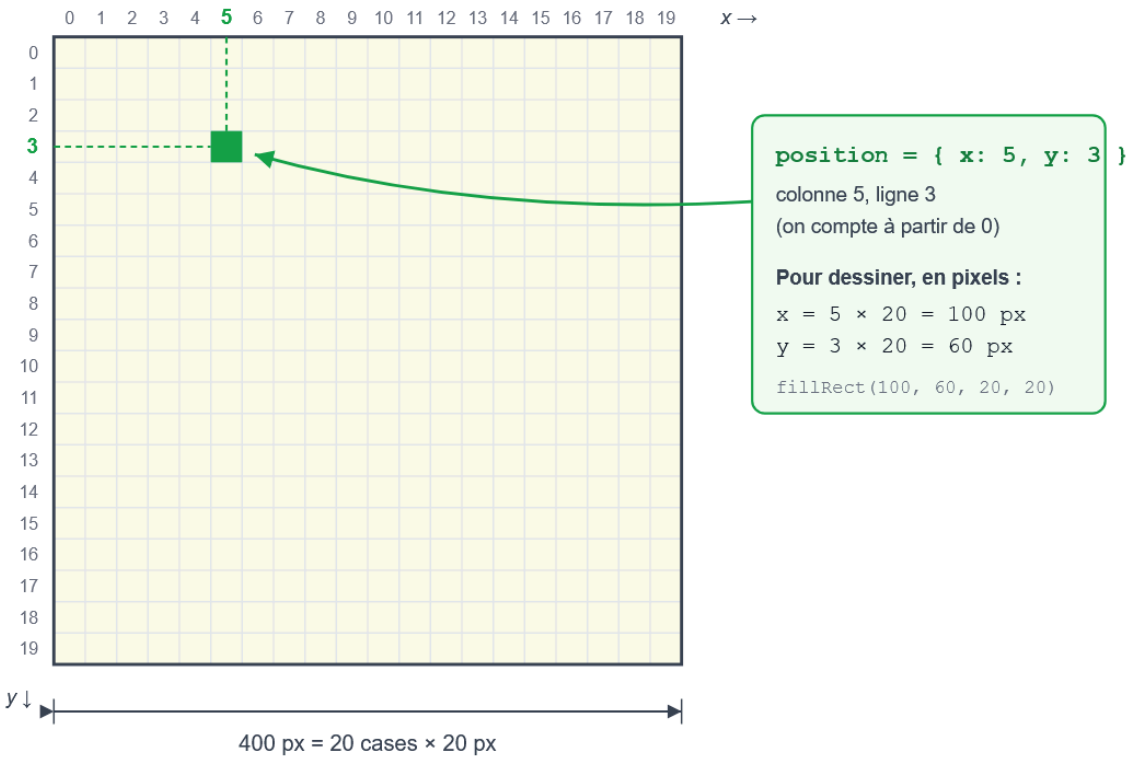
Lance le projet : une case verte glisse vers la droite, et tu peux changer sa direction avec les flèches. Elle sort du canvas et disparaît ? **C'est normal**, on s'en occupera à l'étape des collisions.

⚠️ Remarque l'ordre dans la boucle : on **déplace d'abord, on dessine ensuite**. C'est la règle d'or de toute boucle de jeu : d'abord mettre à jour l'état du jeu (la simulation), puis l'afficher (le rendu). L'écran montre ainsi toujours l'état réel du jeu — et plus tard, tu vérifieras les collisions juste après le déplacement, **avant** de dessiner.

Concept clé n° 1 : la grille

Schéma 1 — La grille : on raisonne en cases, on dessine en pixels

Le canvas de 400 × 400 px est découpé en 20 × 20 cases de 20 px.



Le canvas fait 400 × 400 **pixels**, mais on ne raisonne jamais en pixels. On découpe le canvas en une grille de 20 × 20 **cases** de 20 px chacune. Toute la logique du jeu (positions, déplacements, collisions) se fait en coordonnées de cases : `{ x: 5, y: 3 }` veut dire 6e colonne, 4e ligne (on compte à partir de 0).

La conversion ne se fait **qu'au moment de dessiner** : la case `{ x: 5, y: 3 }` se dessine au pixel `(5 × 20, 3 × 20)`, donc `(100, 60)`.

Pourquoi ce choix ? Parce que comparer des cases est trivial : le serpent est sur le fruit si `tete.x === fruit.x && tete.y === fruit.y`. Essaie de faire ça en pixels et tu vas pleurer.

Vérifie que tu as compris (réponds avant de continuer) :

1. Pourquoi `clearRect` à chaque tour de boucle ? Commente la ligne et observe.
2. À quel pixel se dessine la case `{ x: 0, y: 19 }` ? Où est-elle à l'écran ?
3. Que représente exactement `{ dx: 0, dy: -1 }` ? Pourquoi `-1` pour monter ?

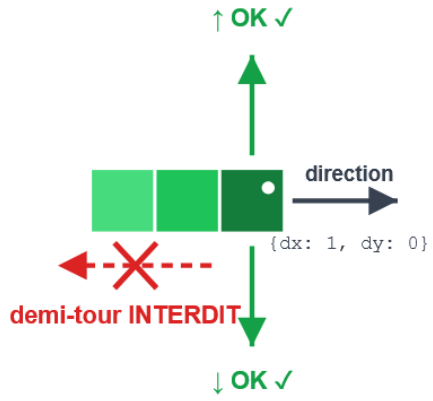
Étape 1 — Interdire le demi-tour

Objectif : quand on va vers la droite, appuyer sur ← ne doit rien faire.

Pour l'instant, toutes les flèches sont acceptées. Avec une simple case, peu importe. Mais quand le serpent aura un corps, faire demi-tour signifiera rentrer dans sa propre nuque : mort instantanée et injuste. Tous les vrais Snake interdisent le demi-tour.

Schéma 2 — Le demi-tour est interdit

Le serpent va vers la droite : quelles directions accepter ?



Comment détecter le demi-tour ?

Deux directions opposées s'annulent :

```
droite {dx: 1} + gauche {dx: -1}
→ somme des dx = 0 et des dy = 0 ✗
```

Alors que pour un virage :

```
droite {dx: 1} + haut {dx: 0}
→ somme des dx = 1 ≠ 0, virage OK ✓
```

> 💡 ***Coup de pouce.*** *Deux directions sont opposées quand leurs vecteurs s'annulent. Compare la direction demandée à la direction actuelle : si la somme des `dx` vaut 0* ***et*** *la somme des `dy` vaut 0, c'est un demi-tour → on ignore la touche. Vérifie sur papier que ça marche pour ↑/↓ et que ça n'interdit pas ↑ quand on va à droite.*

Étape 2 — Une case devient un serpent

Objectif : remplacer la case unique par un serpent de plusieurs cases qui avance.

Concept clé n° 2 : le serpent est un tableau

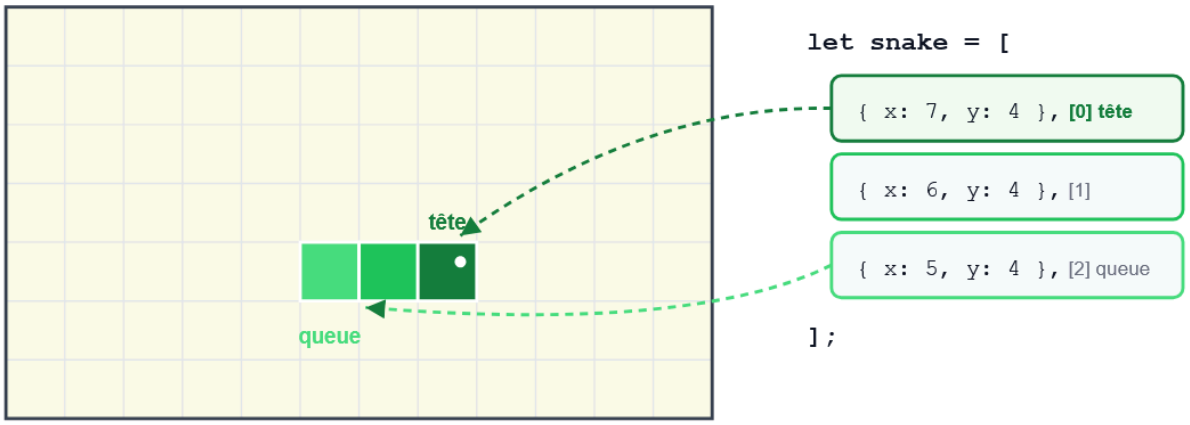
Le serpent, c'est un **tableau d'objets** `{x, y}`, un par case du corps :

JAVASCRIPT

```
let snake = [
  { x: 7, y: 4 },  // snake[0] = la tête
  { x: 6, y: 4 },
  { x: 5, y: 4 },  // le dernier élément = la queue
];
```

Schéma 3 — Le serpent est un tableau d'objets {x, y}

Chaque case du corps est un élément du tableau. La tête est toujours `snake[0]`.



Un segment = un objet. Pour dessiner le serpent : une boucle sur le tableau, un fillRect par segment.

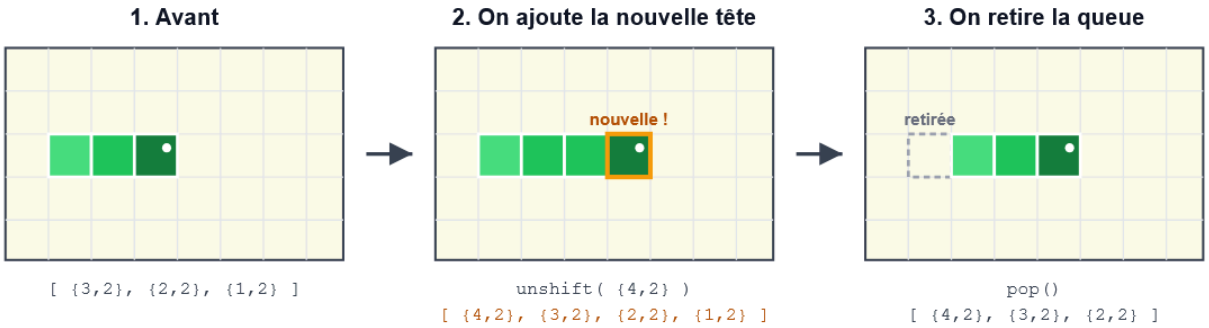
Et voilà l'idée géniale qui rend Snake si simple à programmer : pour faire avancer le serpent, **on ne déplace aucune case**. On en ajoute une et on en retire une :

1. on calcule la position de la **nouvelle tête** (l'ancienne tête + la direction) ;
2. on l'ajoute **au début** du tableau avec `unshift()` ;
3. on retire le **dernier** élément — la queue — avec `pop()`.

Le corps n'a pas bougé, mais visuellement le serpent a avancé d'une case. Prends 2 minutes pour le vérifier sur le schéma ci-dessous : c'est LE déclic de cet exercice.

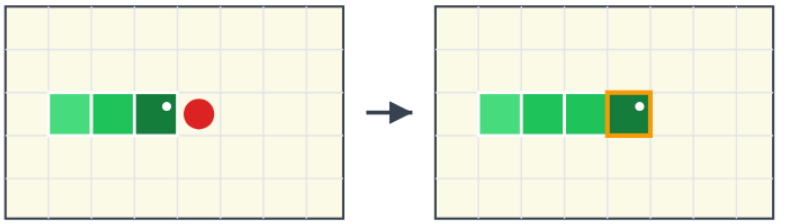
Schéma 4 — Avancer = ajouter une tête, retirer la queue

On ne déplace aucune case ! Le serpent va vers la droite, un tick de la boucle de jeu :



Résultat : même longueur, mais tout le serpent semble avoir avancé d'une case.

Et si la nouvelle tête est sur un fruit ? On saute l'étape 3 : pas de pop() !



Le serpent a grandi d'une case, sans aucun code en plus.

score + 1, nouveau fruit !

> 💡 ***Coup de pouce.*** *La nouvelle tête doit être un* ***nouvel objet*** `{ x: ..., y: ... }`*, construit à partir de* `snake[0]` *— ne modifie surtout pas l'objet de l'ancienne tête. Pour le dessin : une boucle* `for` *sur le tableau, et chaque segment se dessine exactement comme la case verte du départ.*

Validation : un serpent de 3 cases avance et tourne avec les flèches (sans demi-tour possible). Il sort toujours du canvas, c'est toujours normal.

Étape 3 — Le fruit et le score

Objectif : un fruit rouge apparaît à une position aléatoire. Quand la tête l'atteint, le serpent grandit d'une case, le score augmente, un nouveau fruit apparaît.

a) Générer un fruit

Le fruit est un simple objet `{x, y}`. Pour une coordonnée aléatoire valide :

```javascript
const x = Math.floor(Math.random() * nbColonnes); // un entier entre 0 et 19
```

`Math.random()` renvoie un décimal entre 0 (inclus) et 1 (exclu). Multiplié par 20, ça donne un décimal entre 0 et 19,999… et `Math.floor()` arrondit vers le bas : un entier entre 0 et 19, jamais 20. Exactement ce qu'il nous faut.

⚠️ Un fruit ne doit **jamais** apparaître sur le serpent. Il te faut donc une fonction qui vérifie si une case `(x, y)` est occupée par un segment du serpent — tire une nouvelle position tant que c'est le cas (regarde la boucle `do...while`). Cette fonction te resservira plus tard, ce n'est pas du travail perdu.

> 🚀 ***Au passage, les performances.*** *Pour parcourir le serpent, écris une boucle* `for` *classique qui fait* `return true` *dès qu'un segment correspond. C'est plus rapide que les méthodes de tableau comme* `forEach` *ou* `filter`*, qui appellent une fonction pour* ***chaque*** *élément et parcourent toujours le tableau en entier. Cette vérification tournera très souvent : autant prendre le bon réflexe.*

b) Manger le fruit

Après avoir calculé la nouvelle tête, compare-la à la position du fruit.

> 💡 ***Coup de pouce.*** *Grandir ne demande aucun code en plus, seulement du code* ***en moins*** *: si le serpent vient de manger, on ajoute la nouvelle tête mais on* ***ne retire pas la queue****. Le tableau gagne un élément, le serpent a grandi. Relis le bas du schéma 4 si ce n'est pas limpide.*

c) Afficher le score

Le canvas dessine **le jeu**, le HTML affiche **l'interface**. N'écris pas le score dans le canvas : ajoute un élément dans le HTML (par exemple `<div>Score : <span id="score"></span></div>`) et mets-le à jour avec `textContent` à chaque fruit mangé.

Validation : tu peux manger plusieurs fruits d'affilée, le serpent grandit, le score s'affiche, et aucun fruit n'apparaît jamais sur le serpent (mange beaucoup pour tester).

Étape 4 — Game over

Objectif : la partie se termine quand la tête touche un mur ou le corps, et un écran de fin affiche le score avec un bouton « Rejouer ».

a) Détecter la mort

Deux cas, à tester après chaque déplacement, sur **la tête uniquement** :

- **les murs :** la tête est sortie de la grille (réfléchis aux 4 bords — quelles valeurs de `x` et `y` sont encore valables ?) ;
- **la queue :** la tête a les mêmes coordonnées qu'un autre segment du corps.

> 💡 ***Coup de pouce.*** *Pour la queue : parcours le tableau* ***à partir de l'indice 1****. Pourquoi pas l'indice 0 ?*

b) Arrêter la boucle

Détail crucial que personne ne connaît la première fois : `setInterval` renvoie un identifiant. Garde-le dans une variable, car c'est lui qu'on passe à `clearInterval(id)` pour arrêter la boucle. Sans ça, ton jeu continue de tourner en arrière-plan pour l'éternité. 👻

c) L'écran de fin

À la mort : `clearInterval`, puis remplace le contenu de `app` (avec `innerHTML`, comme au démarrage) par un écran de fin : message, score, bouton « Rejouer ».

> 💡 ***Coup de pouce.*** *Pour rejouer, regroupe* ***toute*** *l'initialisation d'une partie dans une fonction* `newGame()` *: remise à zéro du serpent, du score, de la direction, du HTML, et démarrage de l'interval. Appelle-la au chargement et au clic sur « Rejouer ».*
>
> ⚠️ *Piège classique : tu rejoues et le serpent a gardé sa taille ou sa direction d'avant ? C'est qu'une variable a échappé à ta remise à zéro.*

⚠️ En remplaçant `app.innerHTML`, tu détruis l'ancien canvas et ses éléments. Les éléments recréés sont **neufs** : il faut re-sélectionner le canvas et ré-attacher les écouteurs de clic après chaque changement d'écran. (L'écouteur `keydown` sur `document`, lui, survit — pourquoi ?)

Validation : mourir contre un mur et contre sa queue affiche l'écran de fin ; « Rejouer » relance une partie propre, plusieurs fois de suite.

Étape 5 — L'écran d'accueil

Objectif : le jeu démarre sur un écran d'accueil avec un bouton « Nouvelle partie », au lieu de démarrer directement.

Tu as déjà tout ce qu'il faut : c'est le même principe que l'écran de fin. Pour garder un code lisible, écris une fonction par écran qui renvoie son HTML (un template string), par exemple `home()`, `game()`, `end(score)`.

À ce stade, ton jeu est **complet** : accueil → partie → game over → rejouer. Bravo. 🎉 Soigne un peu le CSS, puis attaque les bonus.

Bonus (dans l'ordre de difficulté)

Bonus 1 — La pause

La touche `Escape` (ou un bouton) met le jeu en pause et le reprend.

> 💡 *Pas besoin d'arrêter l'interval : un booléen `isPaused`, et au début de la boucle de jeu, si on est en pause… on ne fait rien (`return`). Affiche par-dessus le jeu un panneau HTML semi-transparent (`position: absolute` + un fond `rgba`) avec un bouton « Reprendre ».*

Bonus 2 — Les fruits rares

Tous les fruits ne se valent pas :

Type	Probabilité	Points	Couleur
normal	70 %	+1	rouge
or	25 %	+3	or
diamant	5 %	+7	cyan

Ajoute une propriété `type` au fruit, qui détermine sa couleur et ses points.

> 💡 *Un seul `Math.random()` suffit : si le tirage est sous 0.70 → normal, sous 0.95 → or, sinon → diamant.*

Bonus 3 — La vitesse augmente

Tous les 5 fruits, le jeu accélère. Attention : on ne peut pas modifier le délai d'un interval existant… mais on peut l'arrêter et en relancer un autre.

Bonus 4 (costaud) — Le bug des doigts rapides

Ton anti-demi-tour a une faille. Tu vas à droite, et **entre deux ticks** tu appuies très vite sur ↑ puis ←. Chaque touche, prise isolément, est valide… mais le serpent n'a avancé qu'une fois. Résultat : il part à gauche, plein demi-tour, mort. 💀

La solution pro : au lieu d'écraser `direction` à chaque touche, mets les touches dans une **file d'attente**, et la boucle de jeu en consomme une seule par tick. Attention à valider chaque nouvelle direction par rapport à la **dernière de la file**, pas la direction courante !

Pour aller plus loin : `requestAnimationFrame`

Notre version utilise `setInterval` : simple et parfait pour apprendre, mais le serpent saute de case en case. Les vrais jeux utilisent `requestAnimationFrame`, qui se synchronise avec l'écran (60+ images/seconde) et permet des animations fluides : un serpent qui **glisse** entre les cases. C'est un beau défi pour plus tard — cherche « game loop requestAnimationFrame » si tu veux transformer ton Snake en jeu pro.

Ton plan de bataille

✓	Étape	Tu sauras que ça marche quand…
☐	Point de départ	la case verte glisse et obéit aux flèches
☐	Étape 1	le demi-tour est ignoré
☐	Étape 2	un serpent de 3 cases avance et tourne
☐	Étape 3	il mange, grandit, le score monte
☐	Étape 4	mur ou queue → écran de fin, « Rejouer » fonctionne
☐	Étape 5	accueil → jeu → fin → rejouer
☐	Bonus	pause, fruits rares, vitesse, file de touches

Derniers conseils

- **Une fonction = une responsabilité** : déplacer, dessiner, générer un fruit, détecter les collisions… Si une fonction fait deux choses, coupe-la en deux.
- `console.log` est ton ami. Le serpent fait n'importe quoi ? Affiche le tableau `snake` à chaque tick et regarde ce qui s'y passe vraiment.
- **Commence lent.** 200 ms par tick, c'est bien pour développer. Tu accéléreras quand tout sera stable.
- **Bloqué plus de 30 minutes sur le même problème ?** Cherche sur Google ou sur MDN, demande à une IA de t'**expliquer** ce qui coince (pas de coder à ta place — c'est toi qui apprends), pose ta question sur Teams, ou note-la pour le prochain cours.

Documentation utile

- Tutoriel canvas (MDN) — dessiner des formes, `fillRect`, `clearRect`
- `setInterval` et `clearInterval` (MDN) — la boucle de jeu
- `KeyboardEvent.code` (MDN) — les codes des touches du clavier
- Les tableaux et leurs méthodes (MDN) — `unshift`, `pop`, et toutes les autres
- `Math.random()` (MDN) — l'aléatoire
- Les template strings (MDN) — pour écrire le HTML de tes écrans
- `textContent` (MDN) — mettre à jour le score

Bon développement, et amuse-toi : c'est un jeu, après tout. 🍎

Énoncé rédigé en collaboration avec Claude Fable 5.

CHAPITRE 3

Examen – Jeu « Mémo Kawaii »

Enoncé

Vous devez réaliser un jeu de memory en **vanilla JS** et **CSS** (projet Vite).
Toutes les ressources graphiques (recto, verso, logos, boutons, titres) sont disponibles ici :
https://ifosup-my.sharepoint.com/:f:/g/personal/thibault_six_ifosup_wavre_be/EqhNW7snsJBDirOobVwmZ2wBqKevUu4jprmSkhxck50AvQ?e=Ow1T1C

Règles de base

- L'écran d'accueil affiche le logo et un bouton **Jouer**.
- Une fois lancé, le plateau contient *deux exemplaires* de chaque image choisie dans le tableau `images` ci-dessous.

```javascript
const images = [
  "bunny","dog","fox","mouse","owl","whale",
  "alpaca","cat","chick","hedgehog","koala",
  "panda","parrot","penguin","raccoon",
  "shark","sloth","tiger",
];
```

- Les étudiants **déterminent eux-mêmes** la taille de la grille et la construisent avec **CSS Grid**.
- Le tableau des cartes est mélangé par :

```javascript
boardState.sort(() => Math.random() - 0.5);
```

- Au clic, une carte se retourne ; lorsqu'une deuxième est révélée :
  - si les deux images sont identiques : elles restent visibles ;
  - sinon : elles restent affichées **1 seconde** puis se re-cachent.
- L'image du dos s'appelle `back.jpeg`.
- Quand toutes les paires sont trouvées, l'écran de fin affiche le titre **Gagné !** et un bouton **Rejouer**.

Fonctionnalités supplémentaires (pour plus de points)

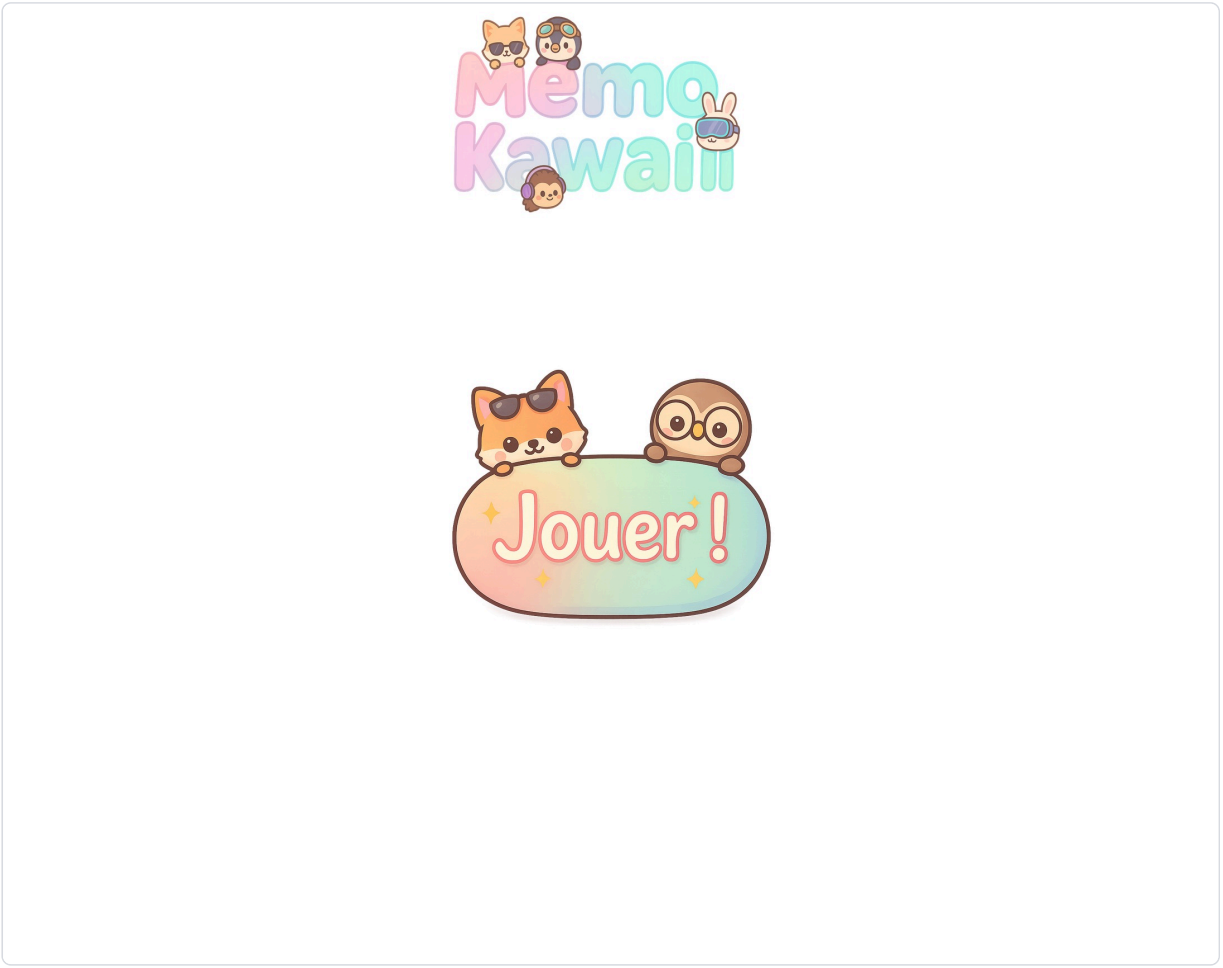
- **Design soigné** : animations de flip, hover, responsive.
- **Calcul de score** : coups, temps, étoiles… au choix.
- **Modes de difficulté** : plusieurs tailles de plateau.
- **Sauvegarde des scores** : `localStorage` ou requête PHP via `fetch`.

Livraison

Un **seul lien vers le dépôt Git** est à déposer dans le devoir Teams du cours.

Exemples

Écran de démarrage



Grille de jeu



Partie terminée

