5XUI-1

PROJET WEB UI/UX

Thibault Six

Version 2 — V3

Généré le 05/09/2026

Sommaire

Introduction

Introduction — Avec le nouveau GPT, c'est fini le design ?

> *Chapitre d'ouverture du cours 5XUI · Projet Web UI/UX · 2026-2027*

Slides du chapitre

- deck2introdesignia.pdf

1. Exercice : le prompt générique

Commençons par ce que vous avez tous en tête. On ouvre un générateur, on lui demande un site, et trente secondes plus tard on a une page. Propre, responsive, en ligne en deux clics. Alors, à quoi sert un webdesigner en 2026 ?

Plutôt que d'en parler, faisons-le.

Consigne. Choisissez un sujet de petit site vitrine : un food truck, un salon de coiffure, un club de tennis de table, un photographe, un gîte. Écrivez un prompt **générique**, comme le ferait quelqu'un qui n'a jamais réfléchi au design : « Fais-moi un site moderne et épuré pour un food truck. » Lancez-le dans l'outil de votre choix. Quinze minutes.

Puis regardez le résultat, et posez-vous une seule question : **qui a décidé quoi ?**

Personne n'a décidé que ce food truck parlait aux étudiants du campus plutôt qu'aux familles du dimanche. Personne n'a décidé que la couleur devait ouvrir l'appétit plutôt que rassurer. Personne n'a décidé de ce que le visiteur doit faire en arrivant, ni pourquoi il reviendrait. La page ressemble à toutes les autres parce que **personne n'a rien demandé de précis**. Comparez vos résultats entre vous : ils se ressemblent tous.

2. Le mot clé, c'est « demande »

L'IA fait tout ce qu'on lui demande. C'est vrai, et c'est précisément le point : elle fait ce qu'on lui *demande*. Elle ne fonctionne pas en autonomie complète. Il lui faut un objectif, un contexte, un public, une intention. Sans ça, elle produit la moyenne statistique de tout ce qu'elle a vu, ce qu'on appelle aujourd'hui le **slop** : lisse, générique, interchangeable, oubliable.

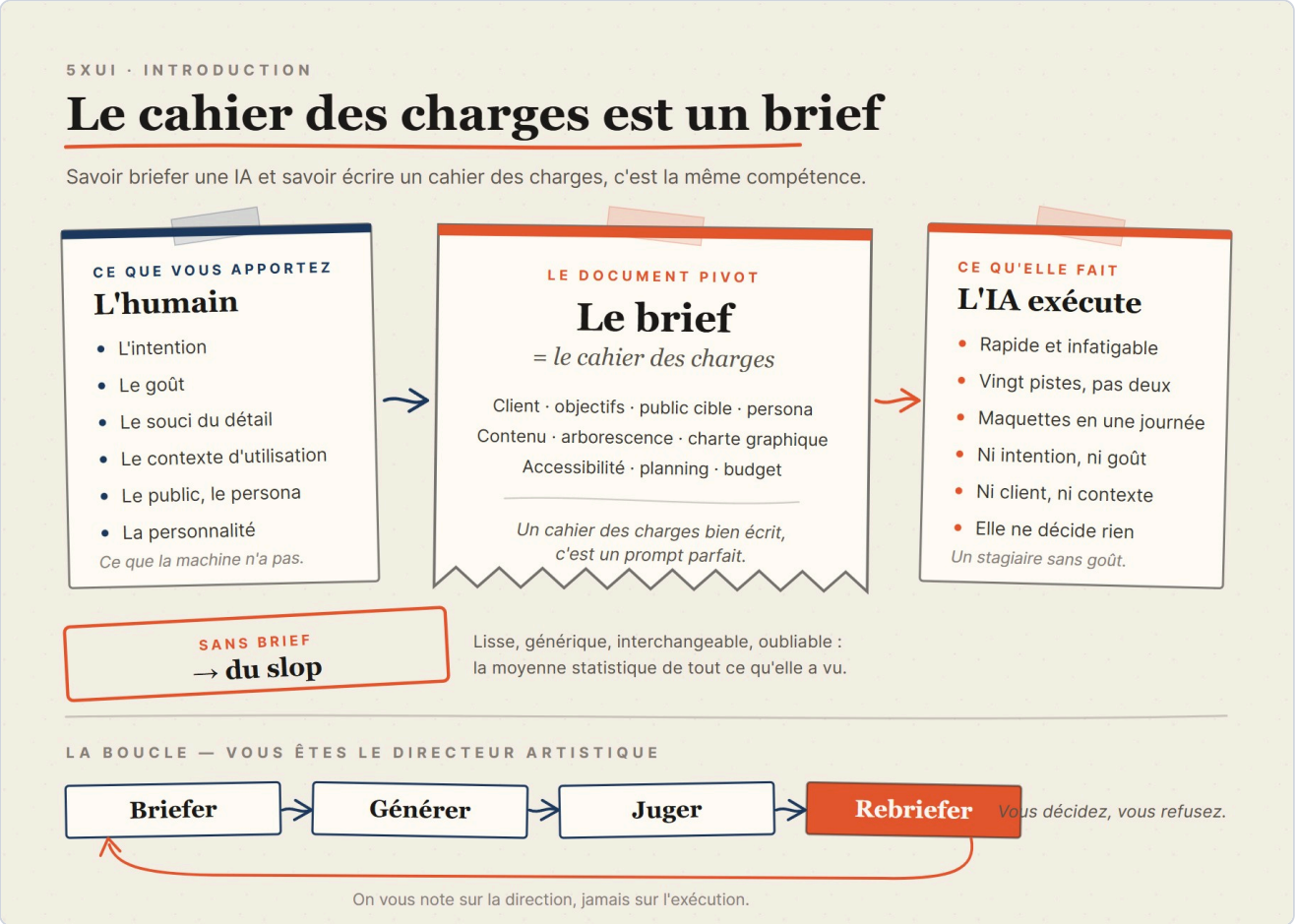
Ce qui manque à la machine, c'est exactement ce que ce cours vous apprend.

Le cahier des charges est un brief

Voici le pont entre les deux volets de ce cours. Pendant des années, on a enseigné le cahier des charges comme un document administratif : le client, les objectifs, le public cible, le contenu, la charte, le planning, le budget. C'est toujours vrai. Mais en 2026, ce document est devenu autre chose : c'est **le brief que vous donnez à vos outils**.

Un cahier des charges bien écrit, c'est un prompt parfait. Un objectif clair, un persona précis, une tonalité définie, une arborescence pensée, une charte graphique argumentée : donnez ça à une IA et elle produit quelque chose de juste. Donnez-lui « fais-moi un site sympa » et elle produit du slop.

Savoir briefer une IA et savoir écrire un cahier des charges, c'est **la même compétence**. Celui qui sait formuler une intention dirige la machine. Celui qui ne sait pas la subit.



3. Exercice : le prompt révisé

Reprenez votre sujet. Cette fois, **décidez** avant d'écrire. Répondez à ces questions sur papier, en cinq minutes :

- **Pour qui ?** Une personne précise : son âge, ce qu'elle cherche, ce qu'elle craint, où elle est quand elle ouvre le site.
- **Pour quoi faire ?** Une seule action attendue du visiteur : appeler, réserver, venir, commander.
- **Quelle ambiance ?** Trois adjectifs, et trois que vous refusez.
- **Quelle référence ?** Une image de votre téléphone, un lieu, une affiche, une pochette de disque que le site devrait évoquer.
- **Quel détail ?** Une idée qui n'appartient qu'à ce projet : le camion est orange, le coiffeur collectionne les vinyles, le club a 60 ans.

Réécrivez le prompt avec ces décisions. Relancez. Comparez avec la première version, et avec les résultats des autres.

Ce que vous devriez constater : les pages ne se ressemblent plus. Certaines sont ratées, mais elles sont ratées **de façon personnelle**. Et c'est vous qui avez fait la différence, pas l'outil.

4. L'IA, partenaire d'idéation et d'expérimentation

Diriger, ce n'est pas seulement filtrer. L'IA est aussi le meilleur partenaire d'**exploration** que vous aurez jamais, précisément parce qu'elle est rapide et qu'elle ne se vexe pas :

- Demandez **dix logos** différents, pas un. Gardez les deux qui vous parlent, demandez dix variantes de chacun.
- Demandez **vingt palettes** de couleurs à partir d'une photo de votre carnet d'inspiration.
- Demandez **cinq maquettes** de la même page, dans cinq directions opposées.
- Puis **croisez** : la typographie de la troisième avec la grille de la première, le ton de la cinquième. Itérez sur ce que vous préférez. Jetez le reste sans regret.

Explorer vingt directions au lieu de deux, produire des maquettes en une journée et passer le reste de la semaine à les rendre bonnes : voilà ce que ces outils changent pour vous. Mais à chaque étape, c'est **vous qui choisissez, qui croisez, qui tranchez**. L'outil propose, l'étudiant dirige.

5. Diriger, c'est aussi équiper

Il n'y a pas que diriger qu'il faut savoir faire. Un bon directeur artistique donne à son équipe les moyens de travailler. Avec un agent, c'est pareil : il faut **se mettre à sa place**.

- **Connaître ses capacités et ses limites.** Qu'est-ce qu'il fait bien, qu'est-ce qu'il rate, qu'est-ce qu'il ne peut pas voir ?
- **Se demander comment il va résoudre le problème.** Quelles infos lui manquent ? Quelles décisions avez-vous prises dans votre tête sans les lui dire ?

- **Lui fournir un vrai environnement.** Un ordinateur avec ce qu'il faut pour travailler, un navigateur pour tester ce qu'il produit, un projet bien structuré.
- **L'orienter vers les bons outils.** Un bon framework, une bonne bibliothèque, un bon package. Un agent qui part sur un mauvais outil produit du mauvais travail, vite.

C'est pour ça qu'on apprend aussi les **bases de Vue.js à la main** dans ce cours. Pas pour concurrencer la machine, mais pour savoir de quoi on parle : pour diriger, pour l'aider quand elle est coincée (ça arrive de moins en moins, mais ça arrive), et surtout pour l'orienter vers les bons outils pour le travail qu'on lui demande. Celui qui n'a jamais écrit un composant ne peut pas juger un composant.

6. À la main, avec la machine, ou les deux ?

Il y a trois façons de traverser ce moment, et elles sont toutes légitimes.

Tout à la main. Certains designers refusent les générateurs et mettent le cœur à l'ouvrage : dessin, collage, typographie, photo, code écrit ligne par ligne. Ce n'est pas un repli nostalgique. Regardez les tendances 2026 : l'imperfection assumée, le collage, la photocopie, le fait-main deviennent des codes de marque recherchés, **précisément parce que** le monde se noie dans le contenu généré. Quand tout est lisse, la trace de la main vaut cher. Cette voie exige d'être excellent, patient, et d'assumer d'être plus lent.

Tout avec la machine. D'autres embrassent les outils à fond : agents, générateurs, itération massive. C'est la voie de la vitesse et de l'exploration. Son risque est évident : sans œil, sans intention, elle mène droit au slop. Elle ne vaut que si la direction est forte.

Les deux. La plupart d'entre vous, et la plupart des studios, seront ici : la machine pour explorer, itérer, produire vite ; la main pour décider, finir, donner la texture, faire ce que le prompt ne donne pas. L'IA génère, vous finissez à la main.

Choisissez votre voie, mais choisissez-la **en connaissance**. C'est pour ça que ce cours vous fait travailler les deux mains : p5.js, le dessin, Figma et le code d'un côté ; les agents et les générateurs de l'autre. Et regardez votre position : vous êtes jeunes, vous n'avez pas encore de nom, pas de réputation à protéger. Vous n'avez rien à défendre et tout à essayer. C'est le meilleur moment pour tester les trois.

Ce qui a de la valeur sur le marché en 2026

Quelle que soit la voie, ce qui se paie est le même. Ce que la machine n'a pas :

- **L'intention.** Savoir ce qu'on veut dire, à qui, et pourquoi.
- **Le goût.** Reconnaître ce qui est juste et ce qui ne l'est pas, avant de pouvoir l'expliquer.
- **Le souci du détail.** Le pixel qui dépasse, la ligne trop longue, le contraste insuffisant. Le slop est fait de détails que personne n'a regardés.
- **Creuser une idée.** Ne pas s'arrêter au premier résultat. Le premier résultat, c'est la moyenne.

- **La personnalité.** Une direction artistique qui vous appartient, qu'on reconnaît.
- **Se démarquer.** Quand tout le monde a accès aux mêmes outils, la différence, c'est vous.

À ça s'ajoutent deux choses qu'on oublie souvent. D'abord la **communication** : parler à un client, écrire clairement, présenter et défendre un choix. Ensuite le **craft** : Photoshop, Figma, le dessin restent pertinents dès qu'on veut plus de contrôle que ce que donne un prompt. Continuez à apprendre à dessiner : la main pense.

7. Le contrat de l'année

Voici comment on va travailler. L'IA est votre stagiaire ultra-rapide, infatigable et sans goût. Vous êtes son **directeur artistique** : vous décidez, vous briefez, vous équipez, vous jugez, vous corrigez, vous refusez. On vous note sur la direction, jamais sur l'exécution.

Pour diriger, il faut un œil. Et un œil, ça se cultive. C'est l'objet des deux chapitres suivants : le carnet d'inspiration, pour remplir votre base de données visuelle, et le test du slop, pour apprendre à mettre des mots sur ce que vous voyez.

Soyez curieux. Allez à des expos, lisez, regardez des œuvres, consommez de la culture sous toutes ses formes. Travaillez une direction artistique qui soit la vôtre. Vous avez une année, un portfolio à faire et un TFE à préparer. Creusez.

Le carnet d'inspiration

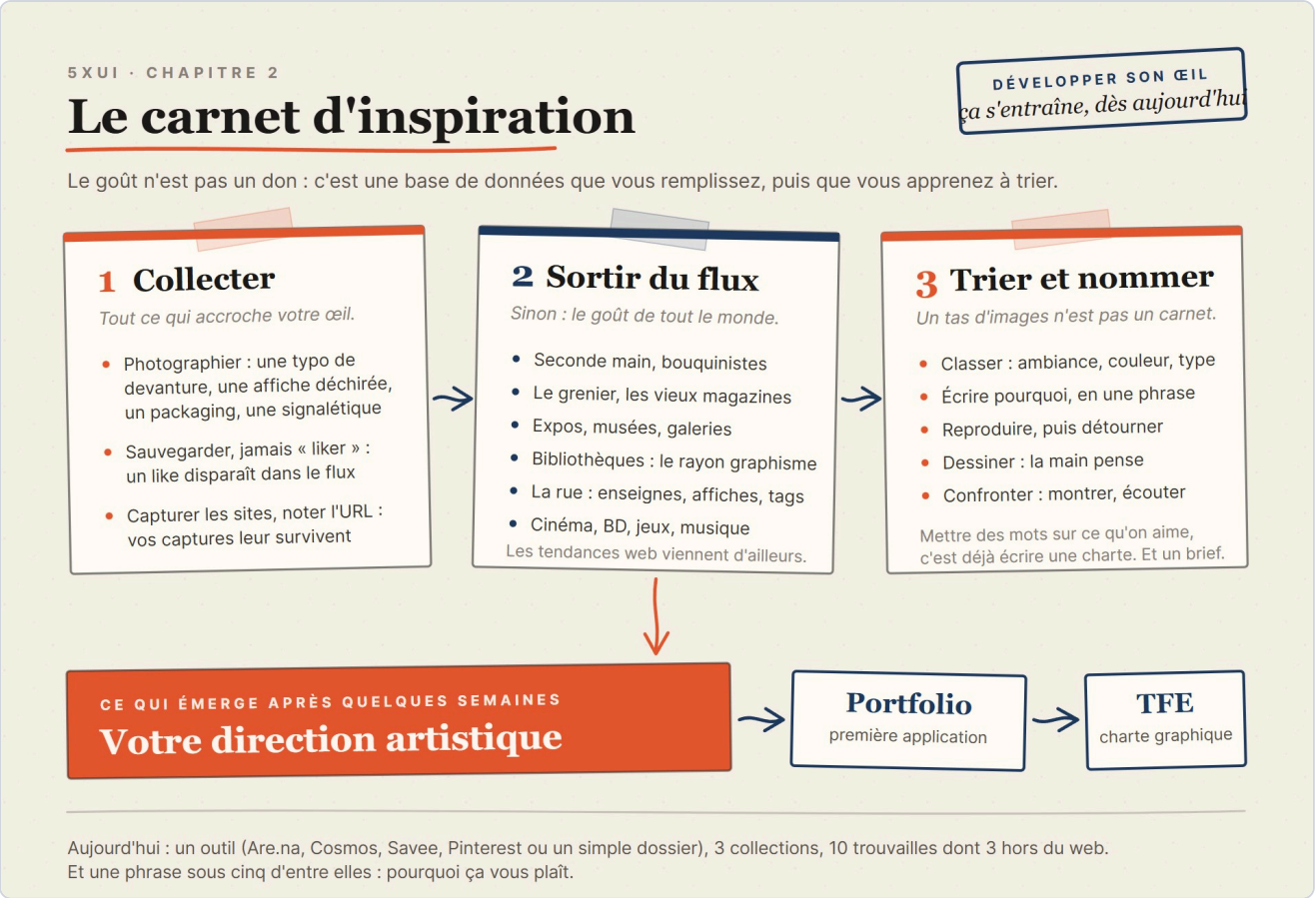
Le carnet d'inspiration — développer son œil

> *Chapitre 2 du cours 5XUI · Projet Web UI/UX · 2026-2027 · Activité en classe*

Le goût n'est pas un don

On croit souvent que le goût est inné : certains l'ont, d'autres pas. C'est faux. Le goût est une **base de données** que vous remplissez toute votre vie, et que vous apprenez à trier. Plus elle est riche et variée, plus vos choix sont justes, et plus vite vous reconnaissez le slop quand vous le voyez.

Un designer avec un œil, c'est quelqu'un qui a **regardé beaucoup de choses avec attention**. Rien de plus, rien de moins. La bonne nouvelle : ça s'entraîne, dès ce matin.



1. Collecter

Prenez l'habitude de **capturer tout ce qui accroche votre œil**, sans vous demander si c'est utile :

- **Photographiez** : une typo de devanture, une affiche déchirée, un packaging, un ticket de caisse, un carrelage, une signalétique, un vieux néon. Votre téléphone est votre premier outil de designer.
- **Sauvegardez** : une image, une œuvre, un site, une animation que vous croisez sur les réseaux. Ne likez pas, **enregistrez**. Un like disparaît dans le flux, une sauvegarde reste.
- **Capturez des sites** : quand une page web vous plaît, faites une capture d'écran complète et notez l'URL. Les sites disparaissent, vos captures restent.

2. Sortir du flux

Les réseaux sociaux vous montrent ce que tout le monde voit. Si vous ne vous nourrissez que là, vous aurez le goût de tout le monde, c'est-à-dire le goût moyen, celui que l'IA reproduit déjà parfaitement. Pour avoir un œil à vous, il faut aller chercher ailleurs :

- **Les boutiques de seconde main et les bouquinistes** : vieux livres de graphisme, magazines des années 70 à 2000, pochettes de disques, cartes postales. Pour quelques euros, des décennies de mise en page.

- **Le grenier** : les vieux magazines de vos parents ou grands-parents. Regardez la typographie, les couleurs, les publicités. C'est une mine.
- **Les expos, les musées, les galeries** : peinture, photo, architecture, design d'objet, mode. Le webdesign emprunte à tout ça, en retard de quelques années.
- **Les bibliothèques** : le rayon graphisme, typographie, photographie, architecture. Gratuit.
- **La rue** : la typographie urbaine, les enseignes, les affiches de concert, les tags, la signalétique.
- **Le reste de la culture** : cinéma, BD, jeux vidéo, musique, littérature. Un générique de film, une interface de jeu, une couverture de roman : tout est design.

Le web n'est pas la seule source de webdesign. Les tendances web viennent presque toujours d'ailleurs, avec quelques années de retard. Celui qui regarde ailleurs voit les tendances arriver.

3. Trier et nommer

Un tas d'images n'est pas un carnet d'inspiration. Ce qui construit le goût, c'est le **tri** et la **verbalisation** :

- **Classez** : par ambiance, par couleur, par type (typo, layout, photo, animation, logo), par projet.
- **Écrivez pourquoi** : sous chaque image, une phrase. « J'aime le contraste entre la serif énorme et le petit texte en monospace. » « Cette palette est chaude mais pas kitsch grâce au gris. » Mettre des mots sur ce qu'on aime, c'est exactement ce qu'on vous demandera dans une charte graphique, et dans un brief à une IA.
- **Reproduisez puis détournez** : copiez à la main un logo, une mise en page, une palette que vous admirez. Comprenez-la de l'intérieur. Puis faites-en quelque chose à vous.
- **Dessinez** : croquis, wireframes papier, thumbnails. Même mal. La main pense différemment de la souris.
- **Confrontez** : montrez vos trouvailles, dites pourquoi, écoutez ce qu'en pensent les autres. De temps en temps, je vous demanderai de présenter une trouvaille à la classe.

4. Construire votre direction artistique

Après quelques semaines, votre carnet va parler de vous. Des couleurs reviennent, des typos reviennent, une ambiance se dégage. Ce n'est pas un hasard : **c'est votre direction artistique qui apparaît**. Le portfolio que vous allez concevoir cette année en sera la première application, et la charte graphique de votre TFE la deuxième.

Activité : ouvrez votre carnet ce matin

Choisissez **un** outil et créez votre carnet pendant la séance :

Outil	Pour qui	Points forts
Are.na	Les curieux, les chercheurs	Canaux connectés, communauté de designers et d'artistes, zéro algorithme, très respecté dans le milieu
Cosmos	Les visuels	Interface magnifique, extension navigateur, collections, découverte de qualité
Pinterest	Les pragmatiques	Volume immense, tableaux, mais algorithme et beaucoup de slop : triez fort
Savee	Les minimalistes	Grille propre, extension, apprécié des studios
Un dossier + un fichier notes	Les indépendants	Aucun compte, aucune plateforme, vous gardez tout

Ensuite, en 30 minutes :

1. **Créez 3 collections** minimum : par exemple « Typo », « Couleurs », « Sites », ou par ambiance (« Chaud », « Brut », « Éditorial »).
2. **Ajoutez 10 trouvailles** dont **au moins 3 qui ne viennent pas du web**. Fouillez la galerie de votre téléphone : les photos prises ces derniers mois, votre dernière visite d'expo, vos vacances, une devanture, un livre, une affiche. Vous avez déjà un carnet sans le savoir.
3. **Écrivez une phrase** sous 5 d'entre elles : pourquoi ça vous plaît.
4. **Partagez le lien** de votre carnet dans le canal Teams du cours.

À partir de maintenant, ce carnet vit. On y reviendra pour le logo, la charte graphique et les maquettes de votre portfolio.

Galeries et sources pour démarrer

- **Sites** : Siteinspire, Godly, Awwwards, Minimal Gallery, Dark Mode Design, Brutalist Websites
- **Typographie** : Fonts In Use, Typewolf, Google Fonts
- **Graphisme et culture** : It's Nice That, Creative Boom, Eye on Design, Behance
- **Livres** : *Steal Like an Artist* (Austin Kleon), *Ways of Seeing* (John Berger), *Design is a Job* (Mike Monteiro), *Grid Systems* (Josef Müller-Brockmann)

Le test du slop

Le test du slop — mettre des mots sur ce qu'on voit

> *Chapitre 3 du cours 5XUI · Projet Web UI/UX · 2026-2027 · Classe inversée*

C'est quoi, le slop ?

Le mot vient de l'anglais, il désigne la pâtée, la bouillie. Depuis 2024, il désigne **la production générique et sans intention** des outils d'IA : les images lisses aux reflets partout, les textes qui disent tout et rien, les sites qui ont un hero, trois cartes, des témoignages et un footer, et qui pourraient vendre n'importe quoi à n'importe qui.

Le slop n'est pas « fait par une IA ». Le slop, c'est **l'absence de décision**. Un humain pressé qui copie un template produit du slop. Une IA bien briefée par un designer avec un œil n'en produit pas.

Apprendre à reconnaître le slop, c'est apprendre à reconnaître **où il manque une intention**. Et c'est la compétence la plus utile de votre carrière : celle qui vous permet de refuser un résultat et de demander mieux.

Les symptômes

Un design qui sent le slop présente souvent plusieurs de ces signes :

- **Interchangeable** : on pourrait remplacer le logo par celui d'un concurrent, rien ne changerait.
- **Sur-fini, sous-pensé** : tout brille, dégradés, reflets, ombres, mais rien n'a de raison d'être là.
- **La palette de tout le monde** : violet-bleu sur fond sombre, dégradé néon, glassmorphism. Ce n'était pas du slop en 2021. Ça l'est devenu.
- **Le texte remplissage** : « Solutions innovantes pour un monde en mouvement ». Qui parle ? À qui ?
- **Le détail abandonné** : alignements approximatifs, hiérarchie plate, contrastes insuffisants, une typo pour tout, images stock qui ne disent rien.
- **Sans contexte** : rien n'indique pour qui c'est fait, ni ce que le visiteur doit faire.
- **Le trop** : quand on ne sait pas ce qu'on veut dire, on met tout. Le slop est rarement sobre.

À l'inverse, un design avec une intention a **une idée**, on la voit tout de suite, et chaque détail la sert.

Activité : à vous de chasser

C'est vous qui constituez le corpus. Pour la prochaine séance :

1. **Trouvez 6 images** : des pages web, des affiches, des logos, des visuels de réseaux sociaux, des publicités. Un mélange de **slop** et de **pas slop**, dans les proportions que vous voulez. Ne dites pas lesquelles sont lesquelles.
2. Cherchez partout : les galeries de sites, vos réseaux, les pubs qu'on vous impose, les sites des commerces de votre quartier, les templates gratuits, les générateurs d'images, votre carnet d'inspiration.
3. **Déposez-les dans le dossier Teams** prévu pour ça, nommées avec votre prénom et un numéro : `prenom-01.png` à `prenom-06.png`. Pas de légende, pas d'indice.

Pendant la séance, on projette les images une par une, et **on juge ensemble** :

1. **Slop ou pas slop ?** Vote à main levée.
2. **Pourquoi ?** Un ou deux symptômes précis, avec les mots de la liste ci-dessus ou les vôtres.
3. **Pour qui c'est fait ?** Si personne ne sait répondre, c'est déjà un indice.
4. Celui qui a apporté l'image révèle ce qu'il en pensait, et d'où elle vient.

Je glisserai quelques images à moi dans le lot, dont au moins une générée par une IA **bien briefée**. Essayez de les repérer.

On ne cherche pas la bonne réponse, on cherche **les mots**. Le but est que vous sortiez de la séance avec un vocabulaire pour dire ce qui va et ce qui ne va pas : hiérarchie, contraste, rythme, tonalité, cohérence, intention.

Débrief

Trois choses à retenir :

- **L'outil ne fait pas le slop, l'absence de brief le fait.** Si une image générée par une IA bien briefée vous a échappé, c'est le point.
- **Vous savez déjà voir.** Vous avez trouvé la plupart des réponses. Ce qui vous manquait, c'étaient les mots. Ils viendront avec le carnet d'inspiration.
- **Le slop est une tendance qui vieillit vite.** Ce qui était frais en 2021 est du slop en 2026. D'où l'importance de regarder ailleurs que dans le flux.

Pour la suite

Chaque fois que vous générez quelque chose cette année, logo, image, maquette, composant, passez-le au test : slop ou pas slop, pourquoi, pour qui. Si vous ne pouvez pas répondre, ne le gardez pas. Rebriefez. Creusez.

P5.js

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")`

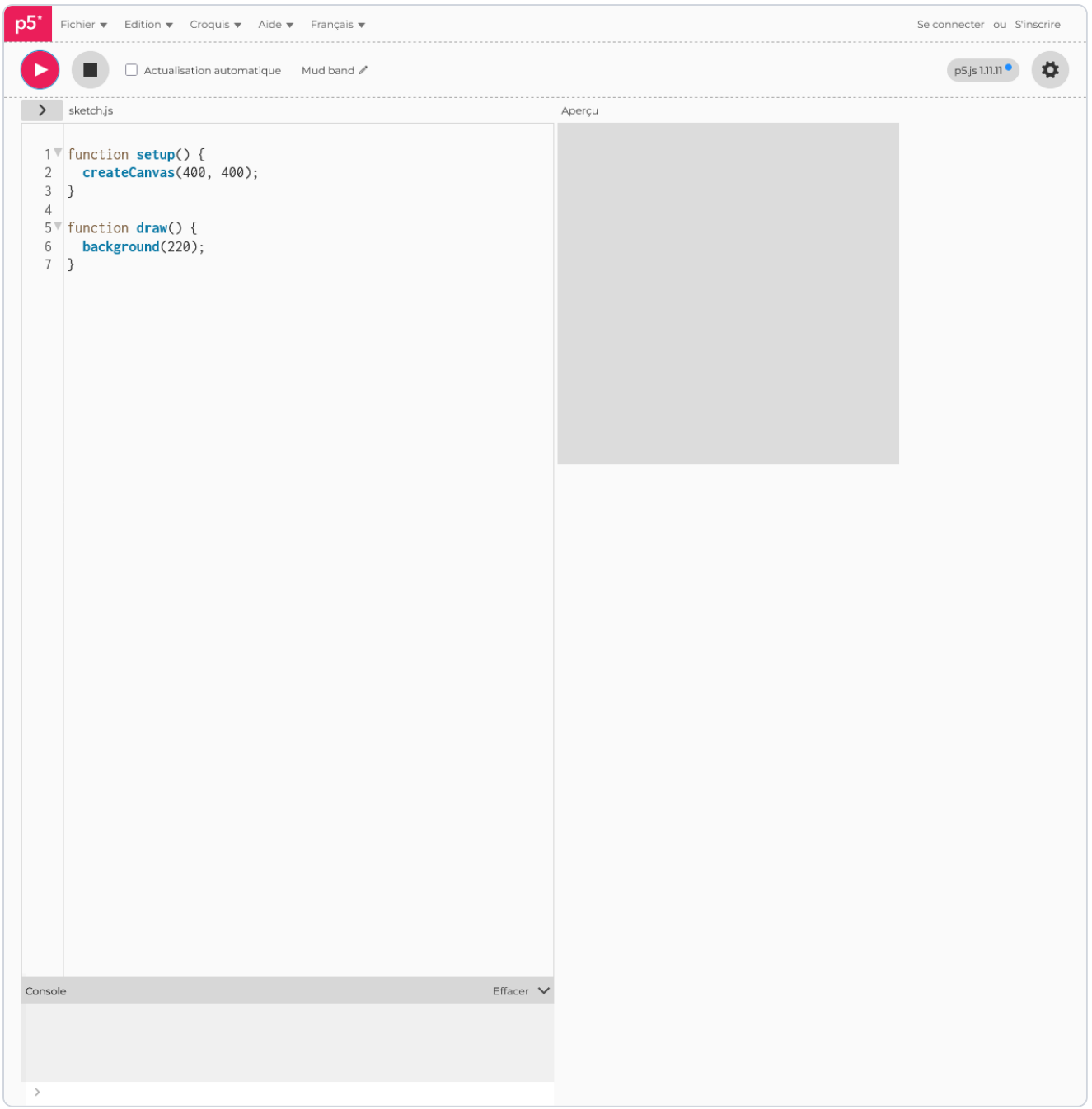
Tu vois ? C'est **exactement le même langage**. p5.js ajoute simplement des fonctions pour dessiner (`rect()` , `ellipse()` , `fill()` ...) et une structure de base (`setup()` et `draw()`) pour organiser ton programme.

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 :

```
<!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 :

```
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()` :

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

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

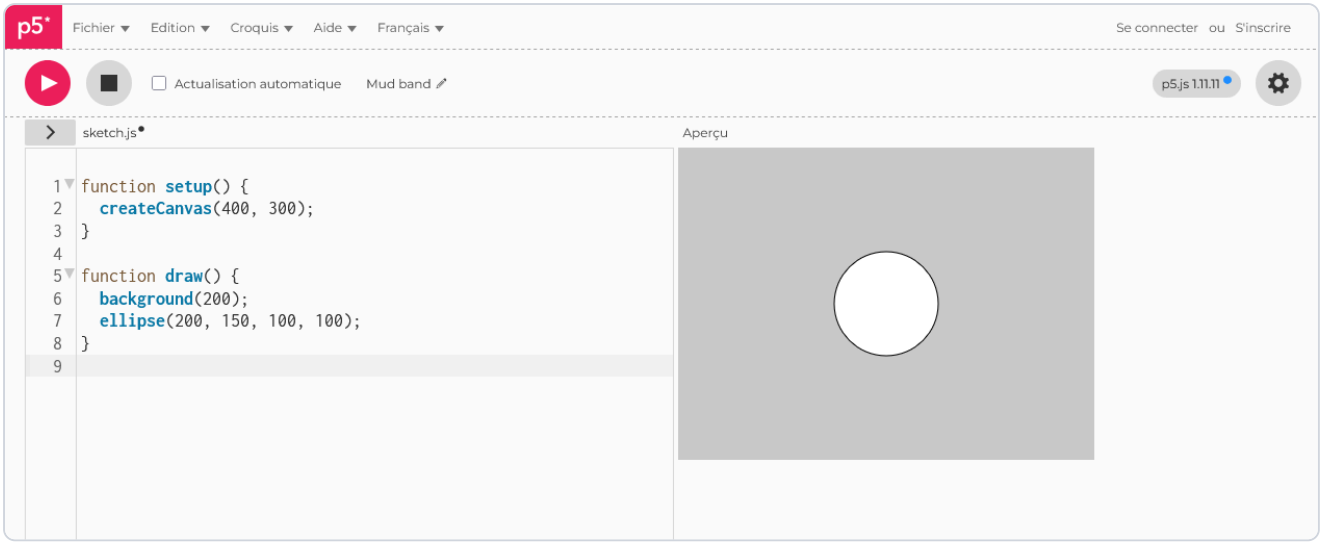
⚠ **Attention** : `createCanvas()` doit toujours être dans `setup()`, idéalement comme **première instruction**.

Ton premier programme

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

```
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 :

```
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 :

```
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

Outils de dessin

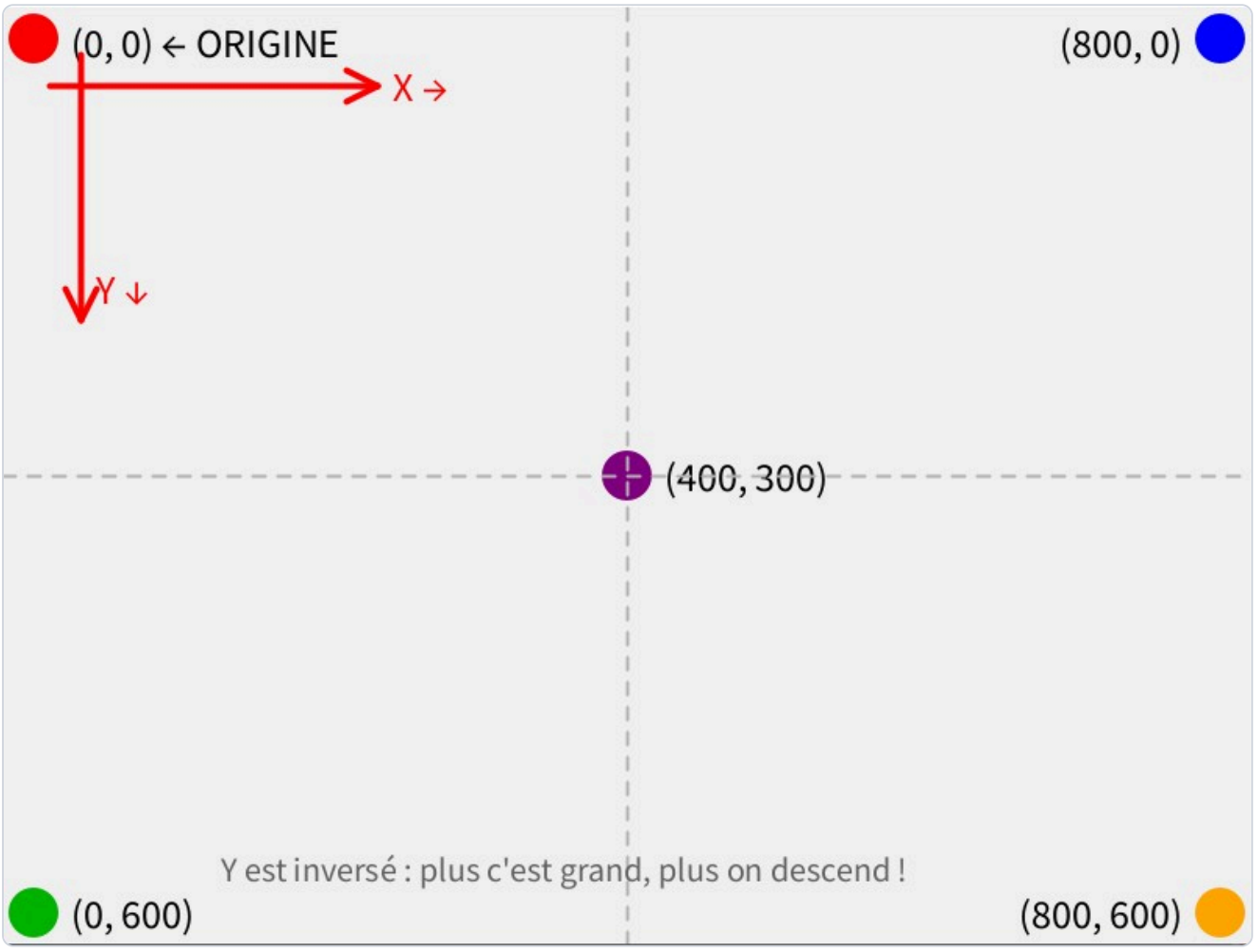
Les Outils de Dessin

> ***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

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

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

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

La ligne

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

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

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

Le rectangle

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

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

```
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 :

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

L'ellipse (et le cercle)

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

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

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

Le triangle

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

Dessine un triangle en reliant trois points.

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

L'arc (portion de cercle)

```
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`

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

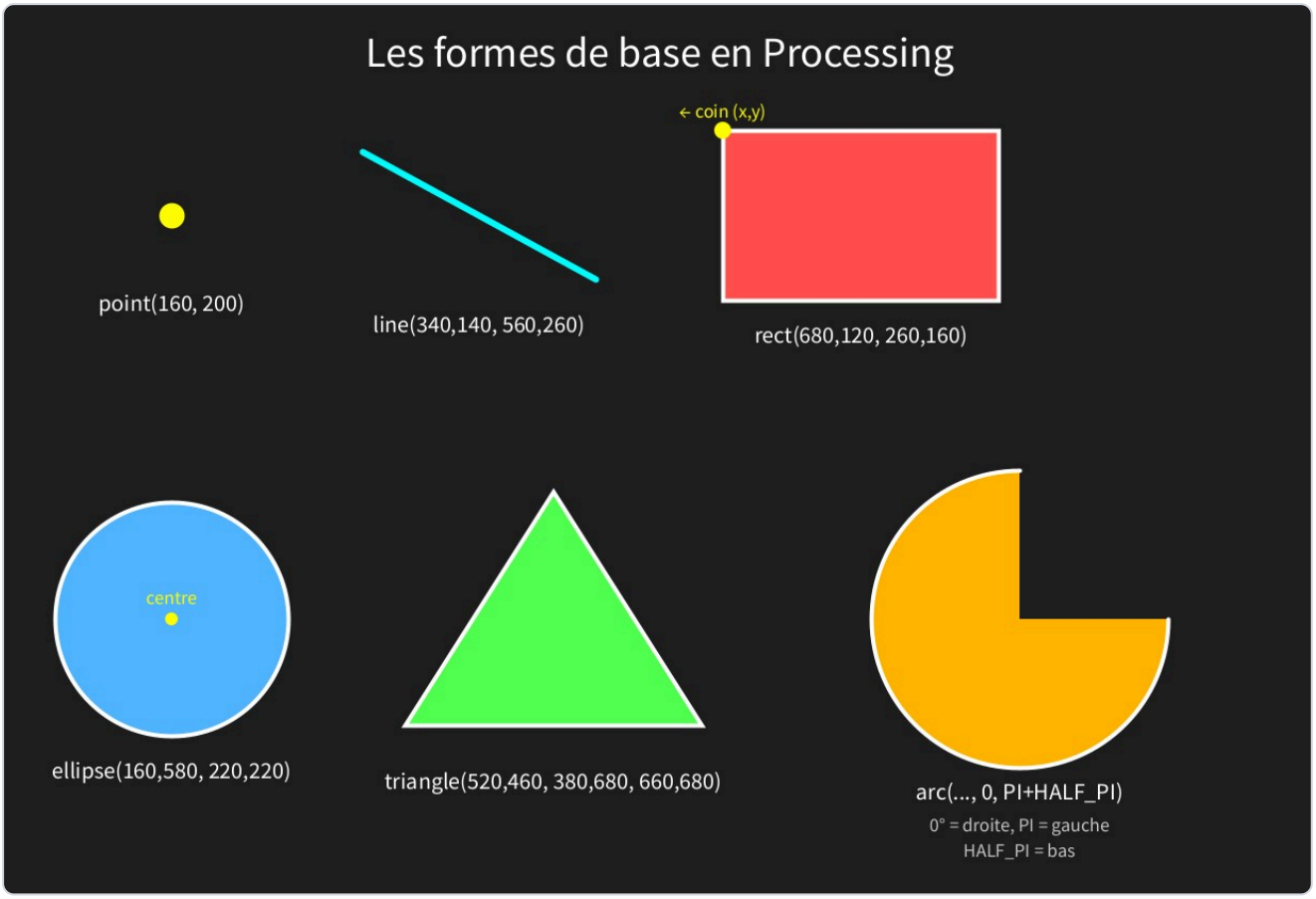


Tableau récapitulatif

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
`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.

```
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) :

```
noFill();
```

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

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

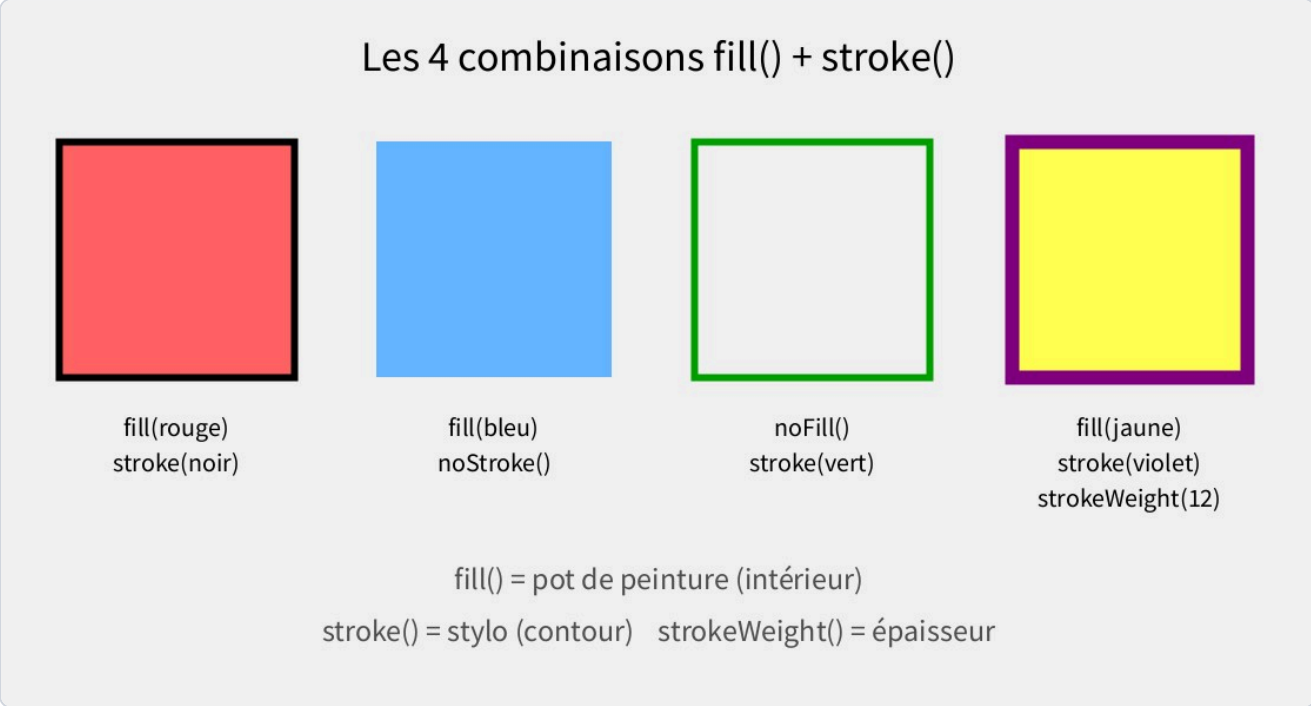
`strokeWeight()` définit l'**épaisseur** du trait en pixels.

```
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 :

```
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"

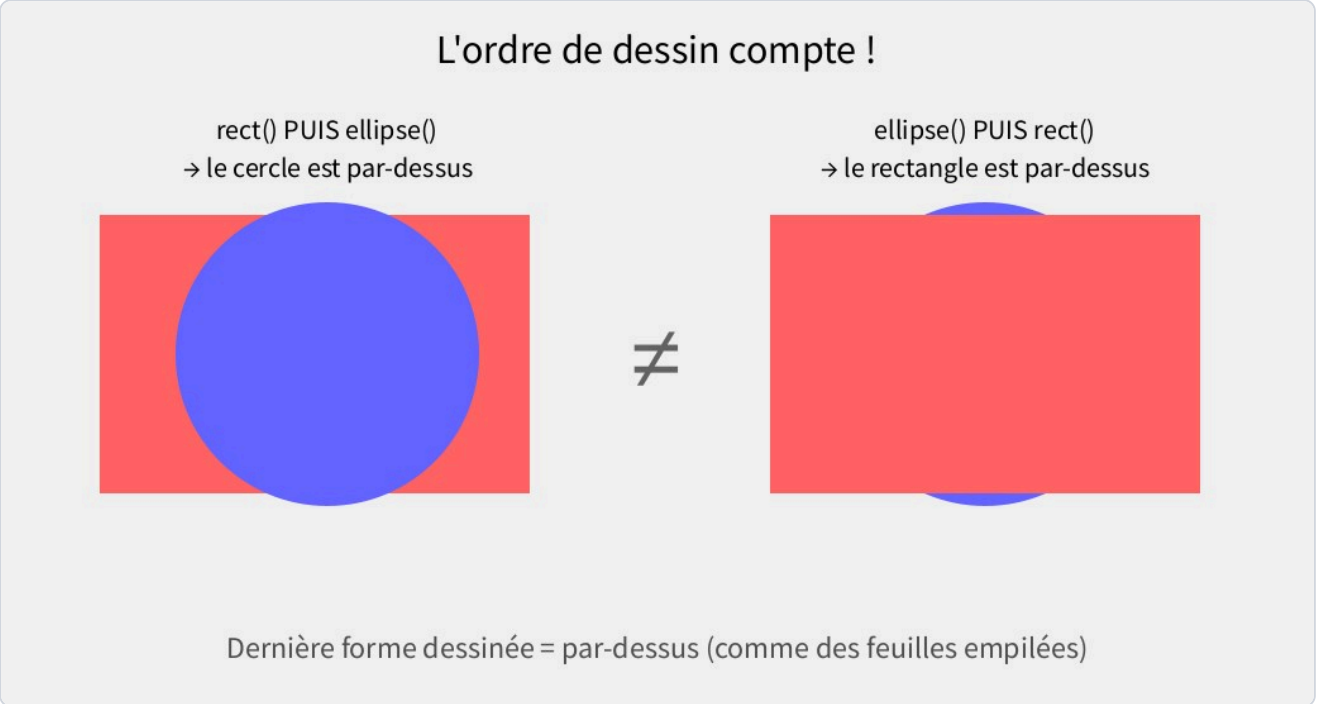
`fill()` et `stroke()` s'appliquent à **tout ce qui est dessiné après**, jusqu'au prochain appel. C'est comme si tu changeais de feutre : une fois que tu prends le rouge, tout ce que tu dessines est rouge jusqu'à ce que tu changes.

```
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.




```
// 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

Outil	Fonction	Ce que ça fait
✏ 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)

Couleurs

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

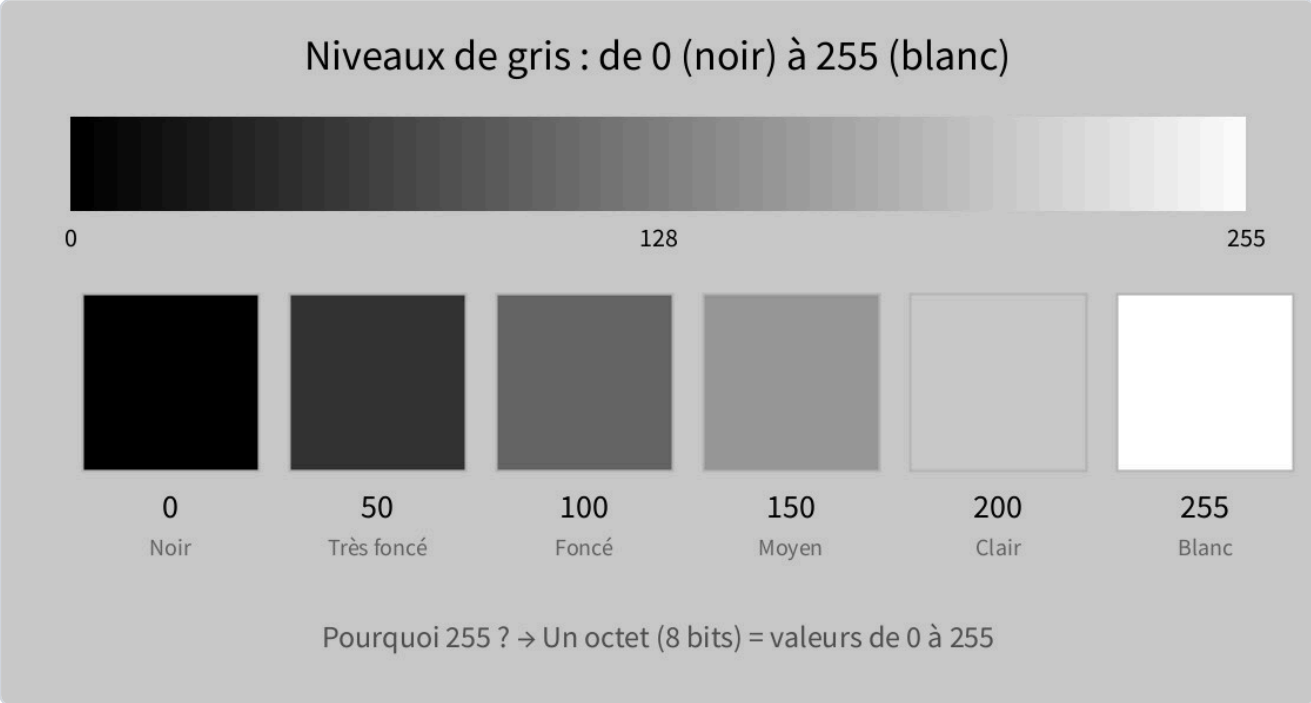
```
background(240);        // Fond gris très clair

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.

```
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).

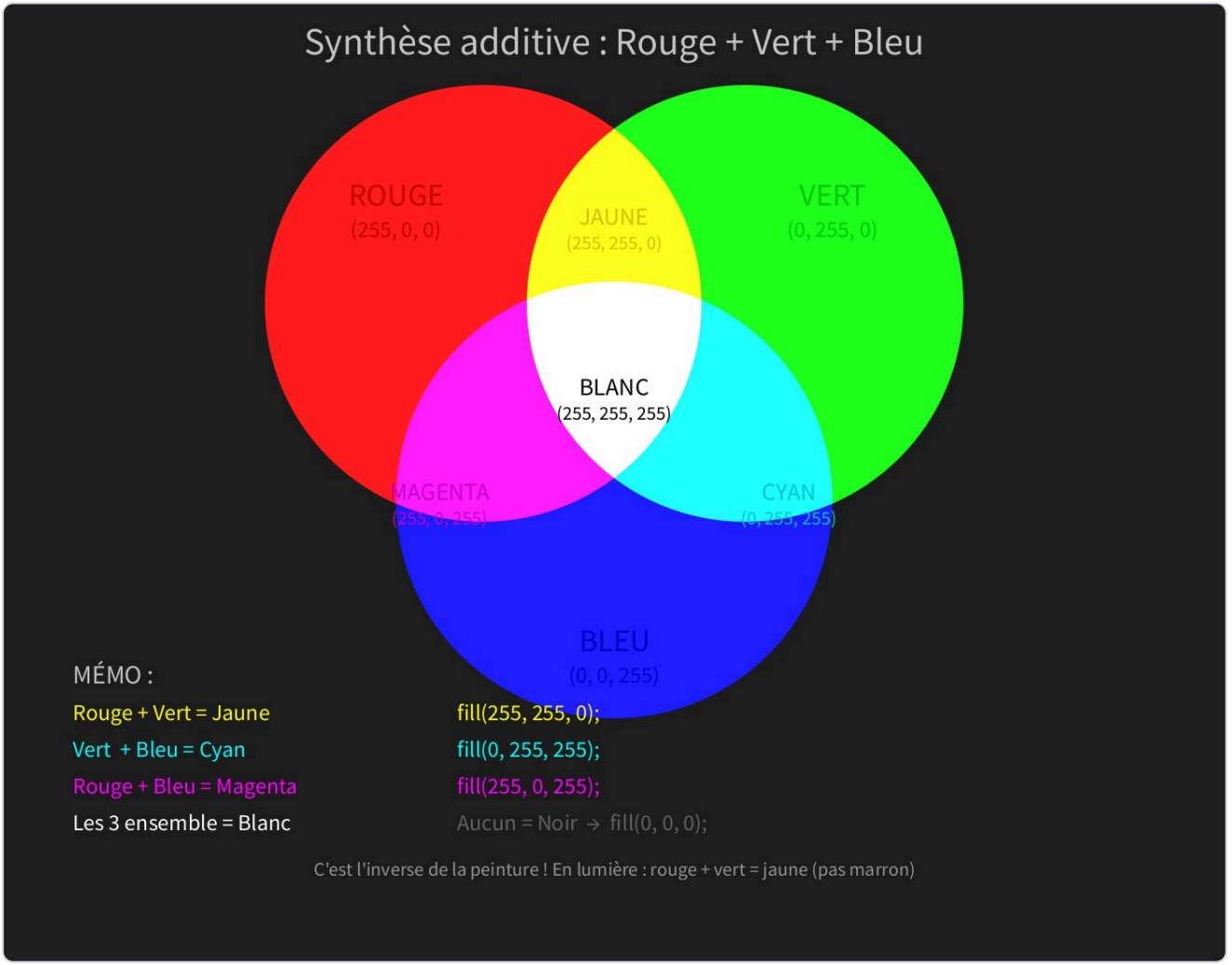
Les couleurs primaires

```
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

```
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 ?

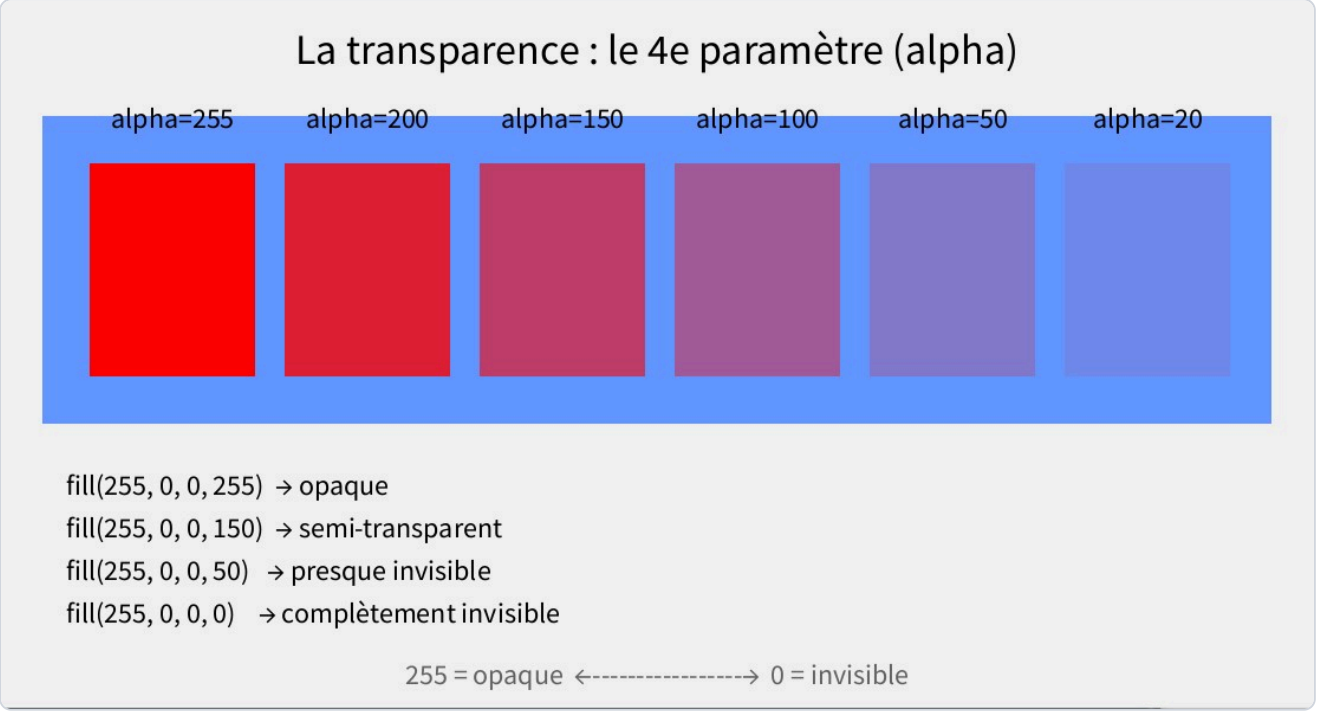
Les couleurs s'appliquent à trois endroits :

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 :

```
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) :

```
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 :

```
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()` :

```
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 :

```
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 :

```
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

Concept	Syntaxe	Exemple
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
Les 3 égales	Niveau de gris (du noir au blanc)
Les 3 à 255	Blanc
Les 3 à 0	Noir

Premières formes

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()`

Famille	Ce que tu sais faire	Fonctions
✏ **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**

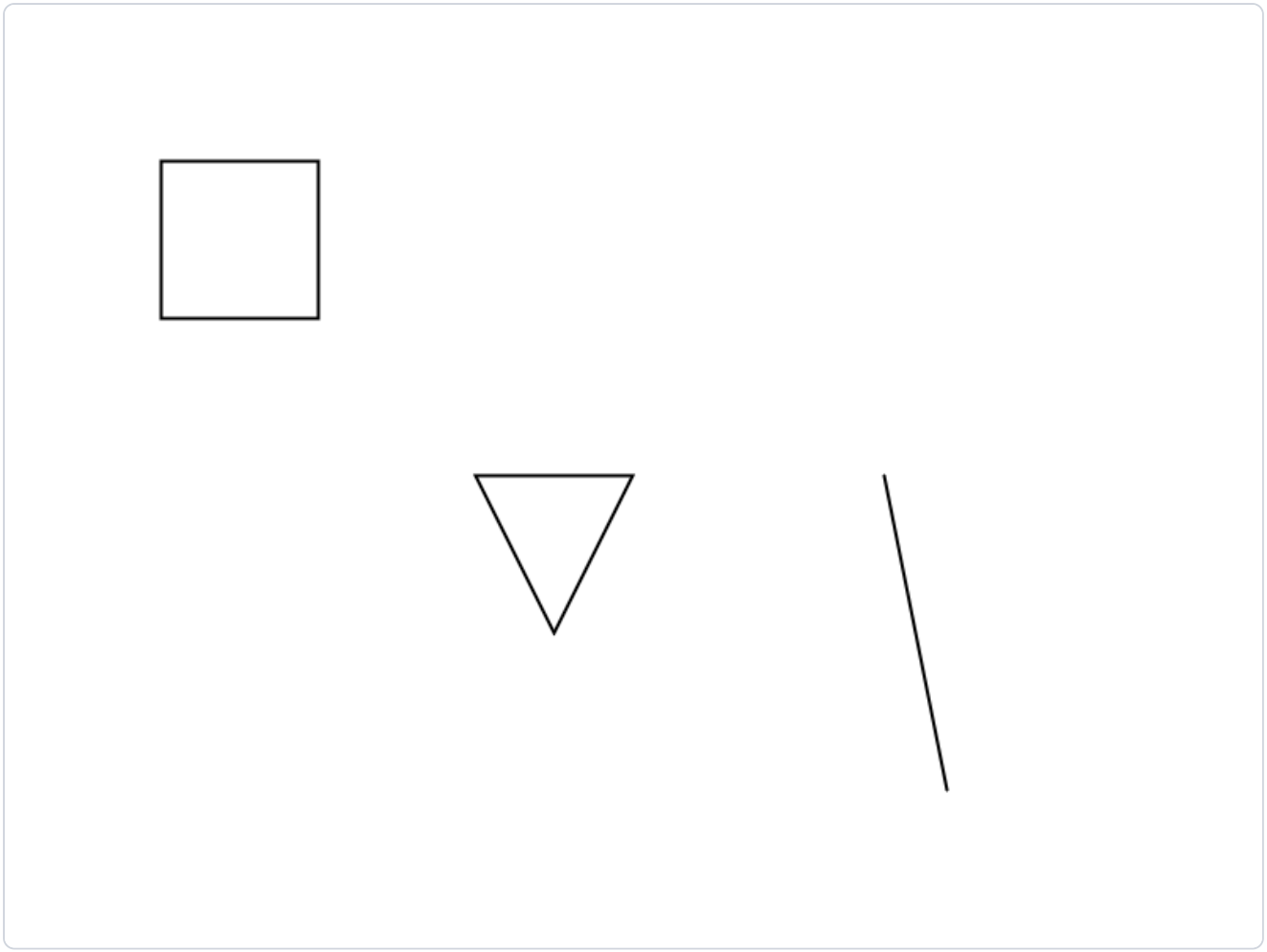
Les trois formes **ne doivent pas se chevaucher**.

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 :

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

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

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

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

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

Exercice 2 — Changement de couleurs

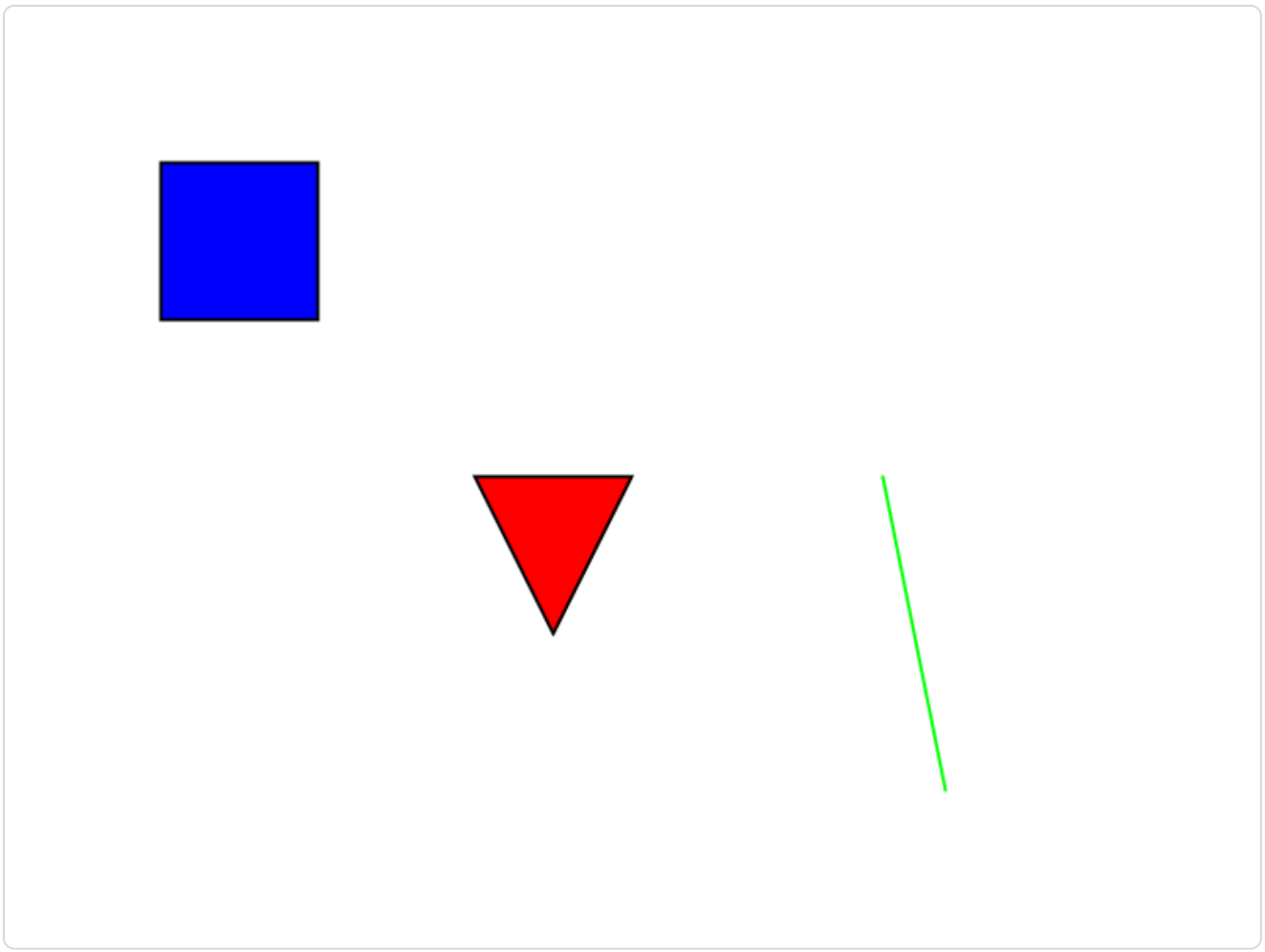
Consigne

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 :

```
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 !

Variables

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 :

```
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;`

```
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";`

Type	Ce que ça stocke	Exemple
`boolean`	Vrai ou faux	`let estVisible = true;`

```
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` :

```
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 :

```
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 :

```
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

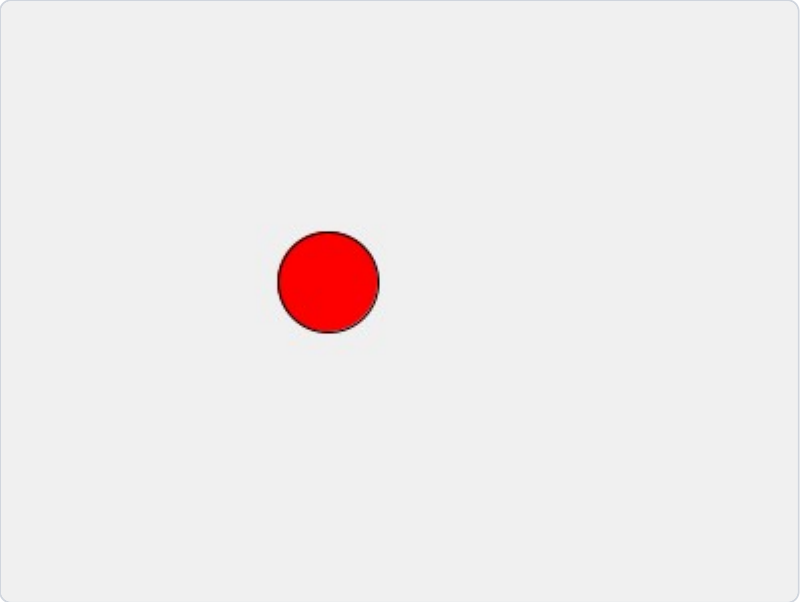
Variable	Ce qu'elle contient
`mouseX`	La position **horizontale** de la souris (en pixels depuis la gauche)
`mouseY`	La position **verticale** de la souris (en pixels depuis le haut)

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

```
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`

Règle	Bon	Mauvais
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

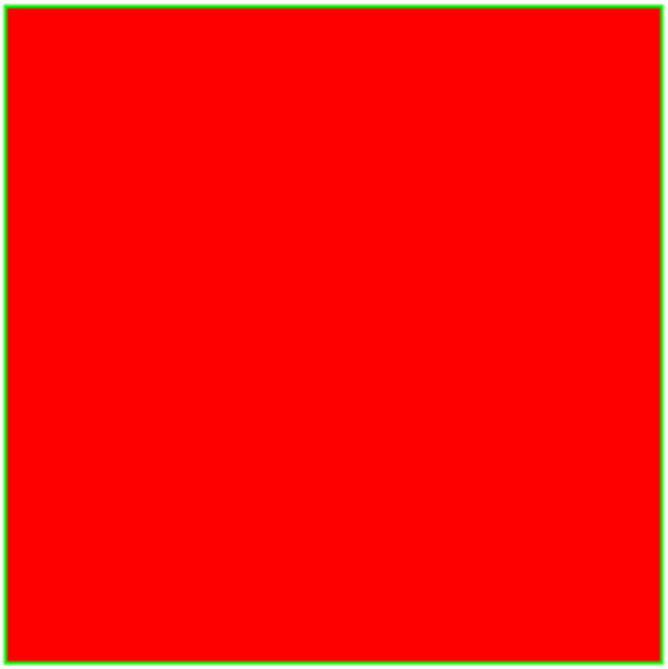
Dessine un **carré rouge** avec un **contour vert** parfaitement **au centre** de l'écran, en utilisant `width` et `height`.

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 :

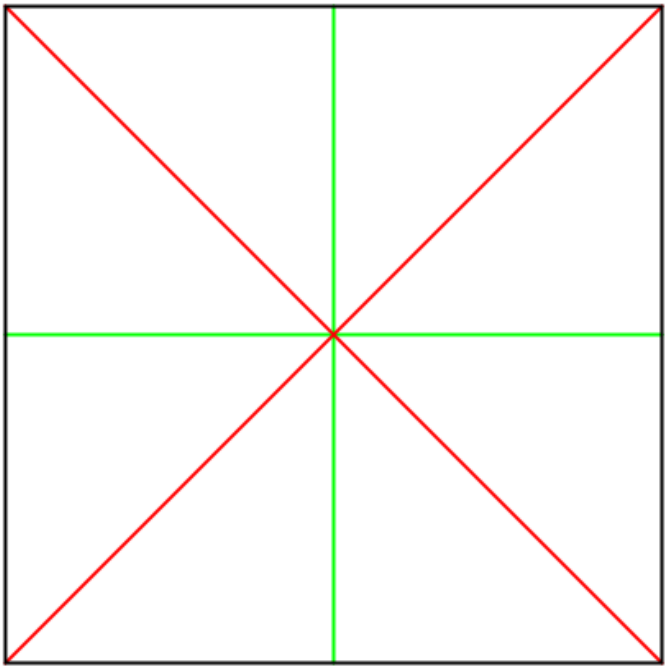
1. Crée des variables `x`, `y` pour la position du coin supérieur gauche du carré, `cote` pour la taille du côté, et `demiCote` pour la moitié
2. Dessine le carré en utilisant ces variables
3. Trace la **médiane verticale** (ligne du milieu haut au milieu bas du carré)
4. Trace la **médiane horizontale**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

Conditions

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"

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

Exemple concret :

```
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"

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

```
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 :

```
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 !

```
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`

```
// 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

```
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

```
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 `&&` :

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

Exemple : dessiner au clic gauche

```
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

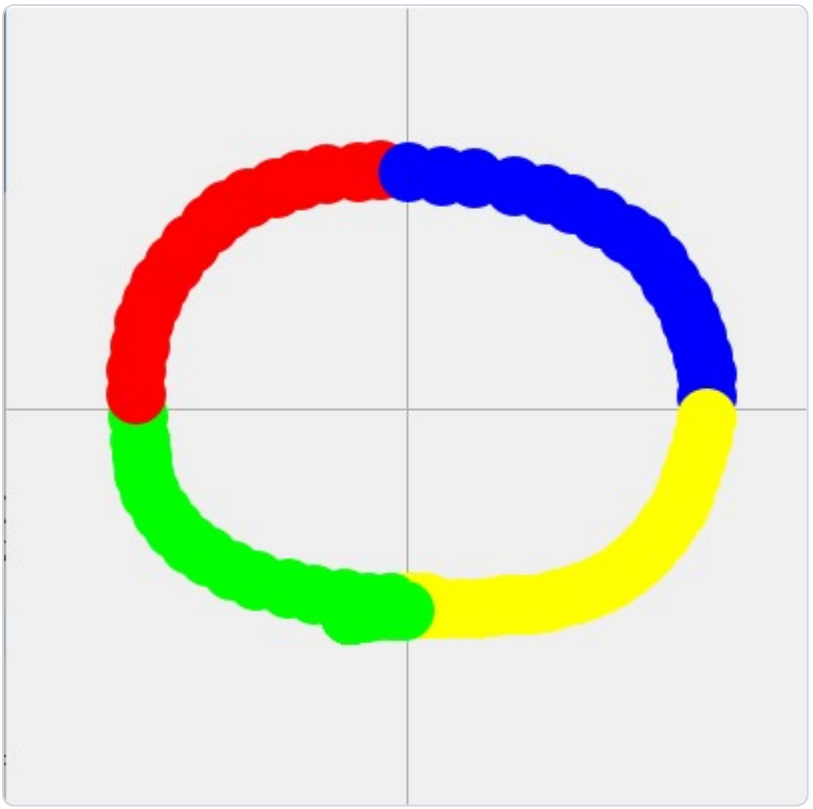
Contraintes

- 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 `&&` :

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

Indice 3 — Structure globale du programme :

```
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 :

```
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 !

Boucles

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 :

```
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`

```
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** :

```
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.

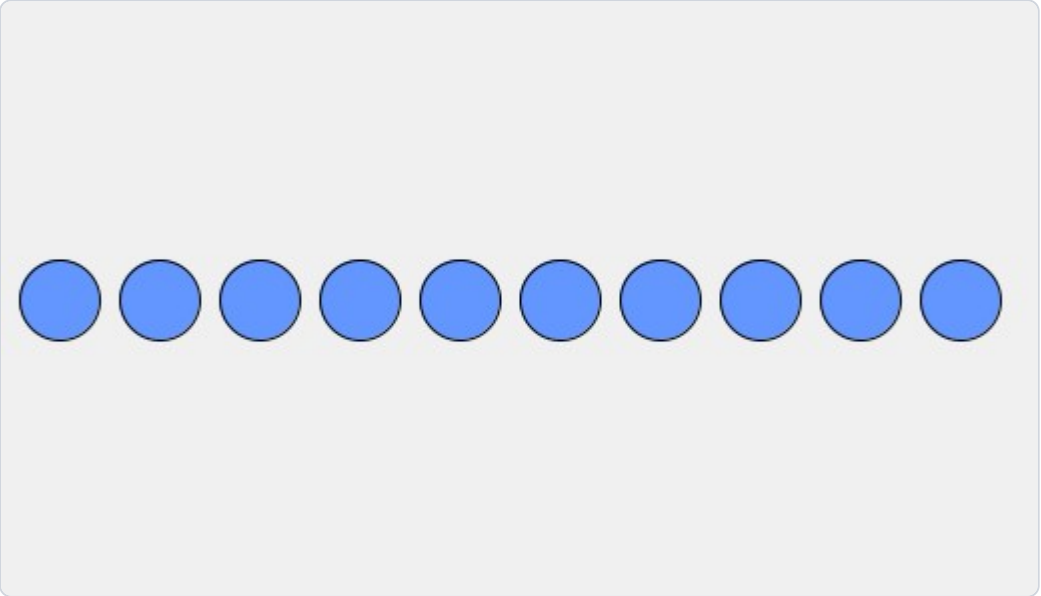
Reprenons nos 10 cercles du début :

```
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**
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` 😎

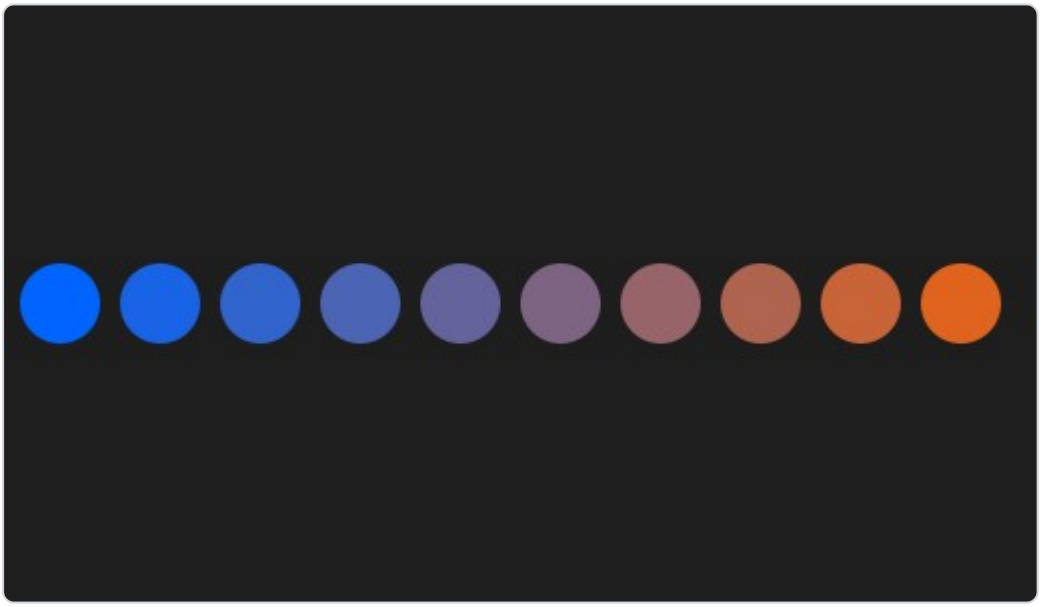
Utiliser `i` pour varier les couleurs

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

```
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** :

```
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**.

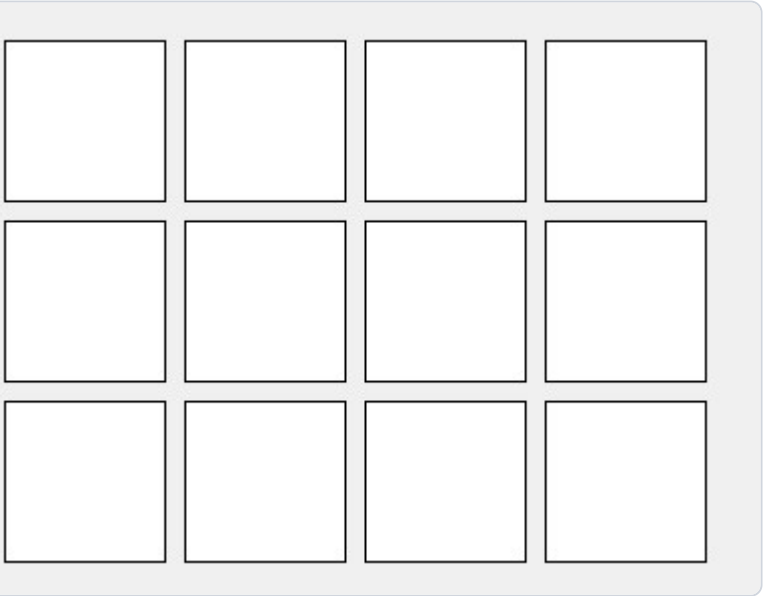
Les boucles imbriquées (grilles 2D)

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).

```
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

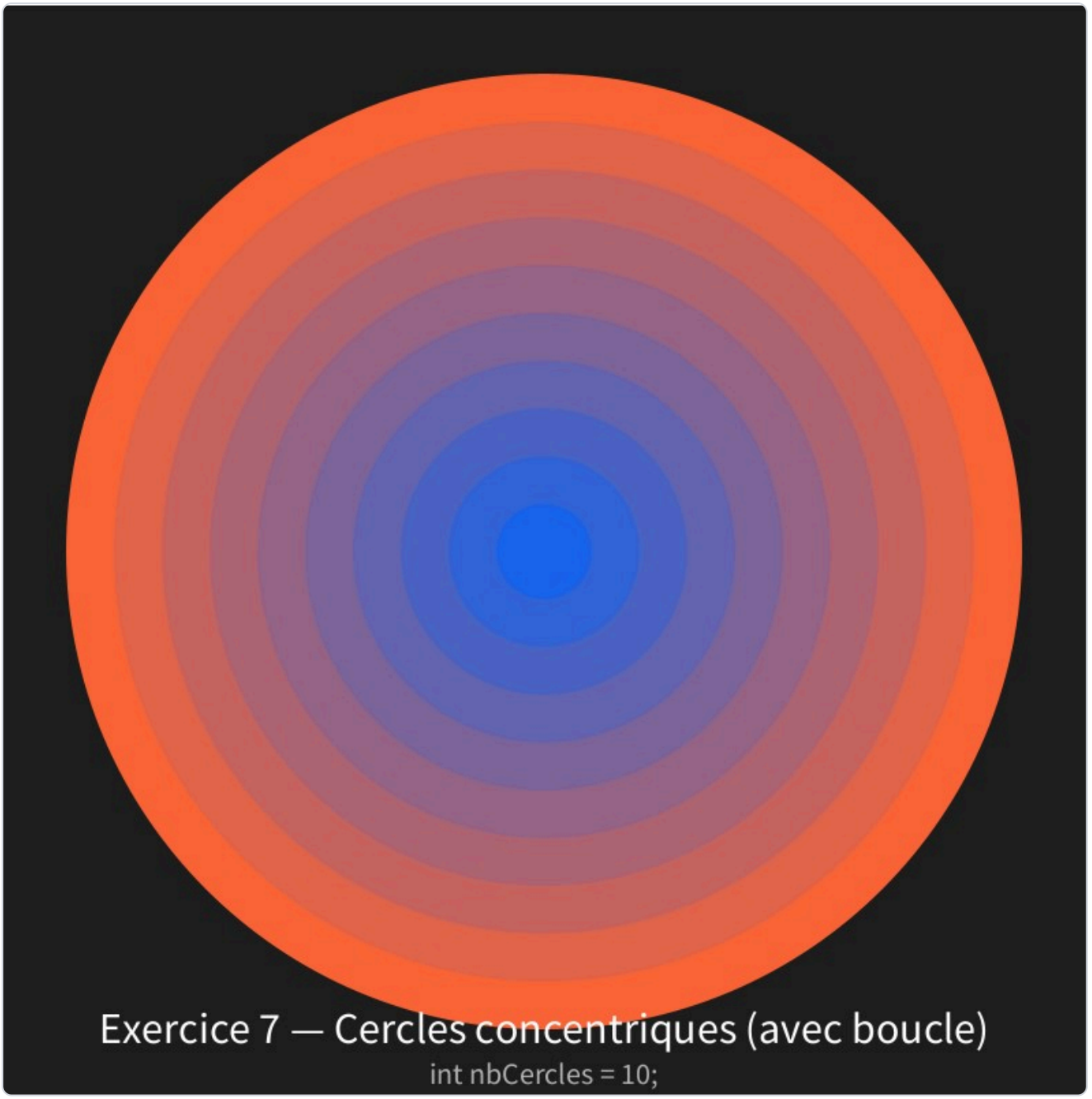
Reprends le principe des cercles concentriques, mais cette fois avec une **boucle** `for`. Le nombre de cercles est stocké dans une variable `nbCercles`. Chaque cercle a une **couleur différente** calculée automatiquement à partir du compteur `i`.

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 !

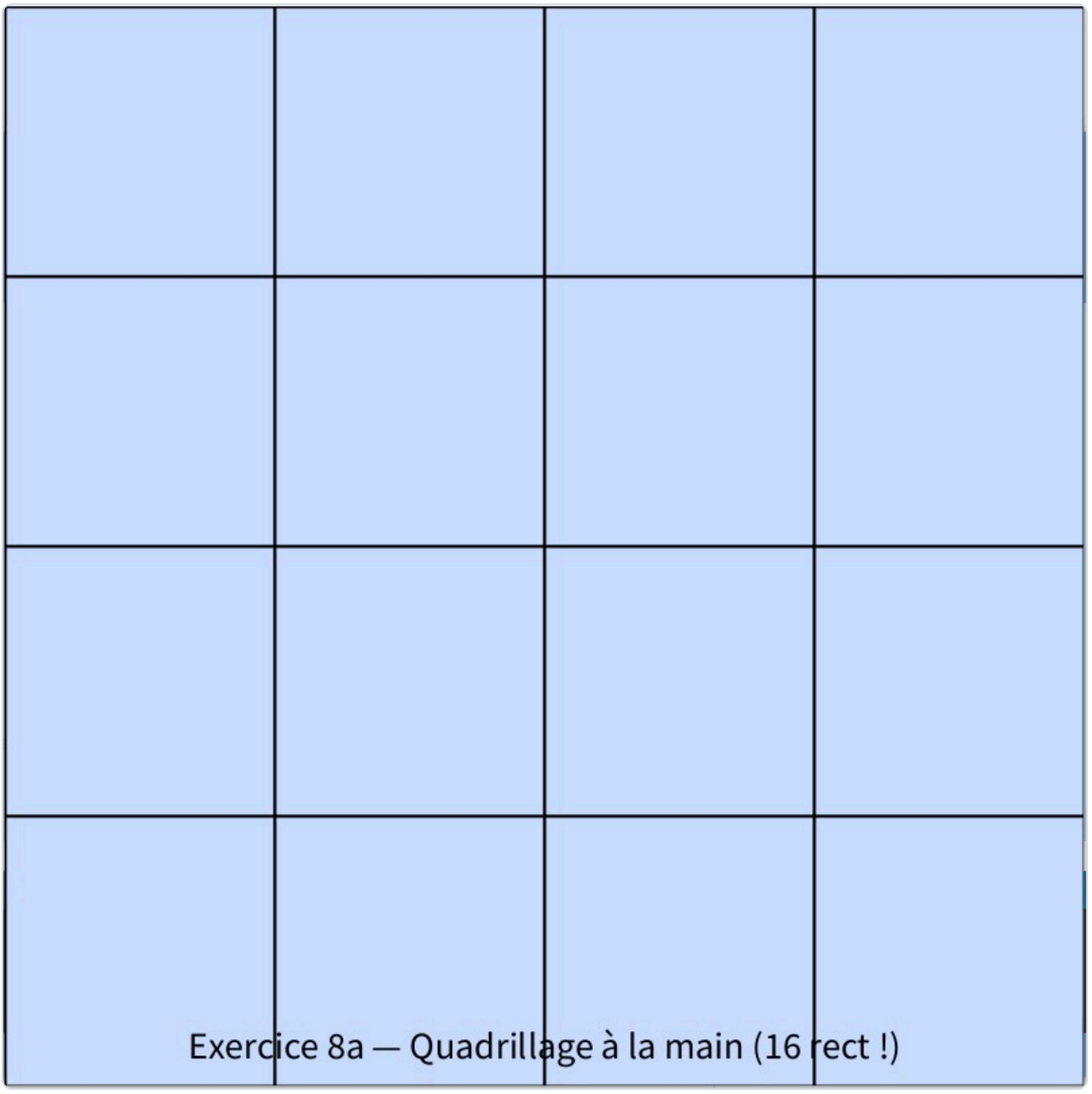
Exercice 8 — Le damier (boucles imbriquées)

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

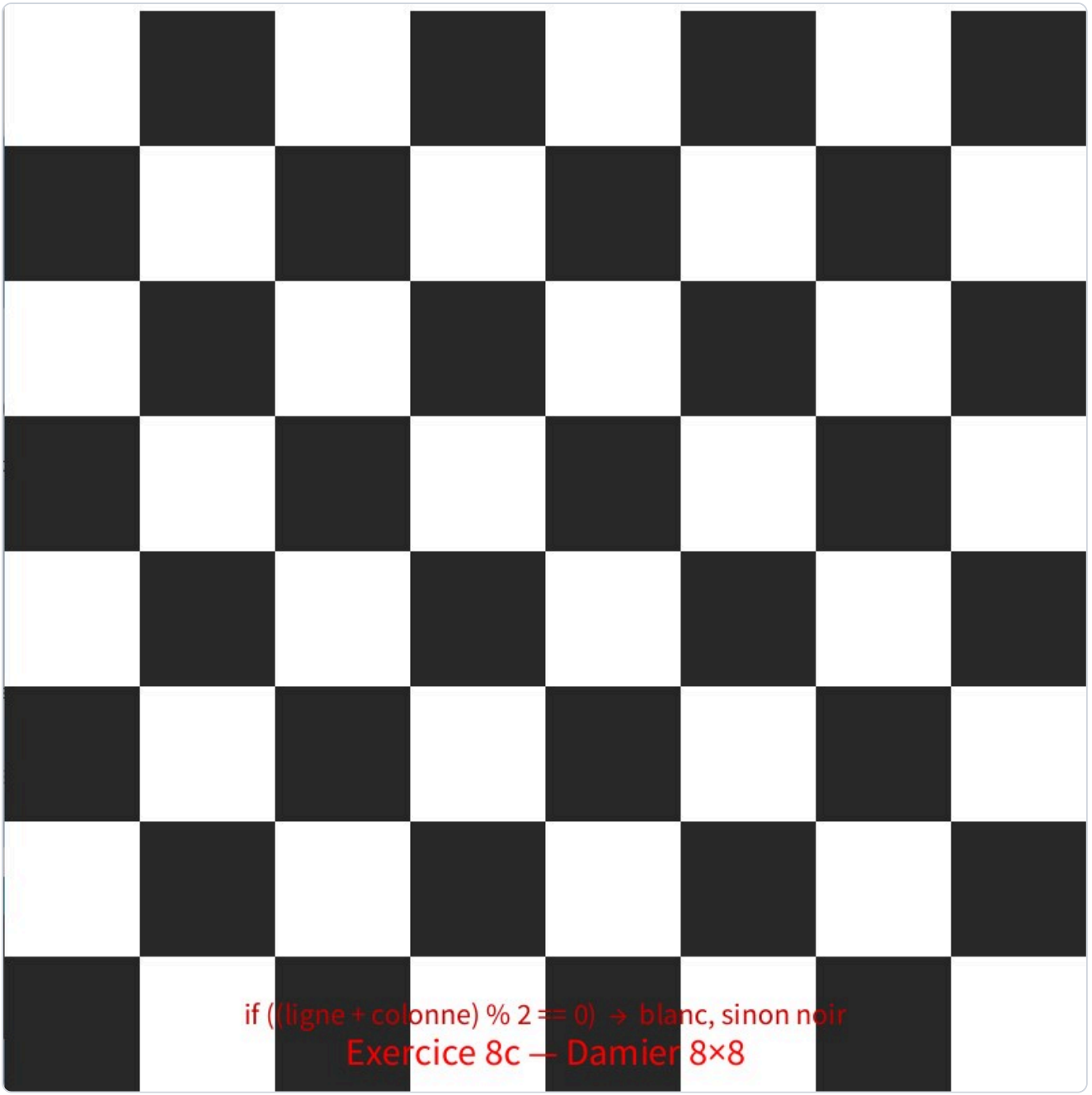
Alterne les couleurs **noir et blanc** comme un échiquier.

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 :

```
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 !

Art Génératif

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.

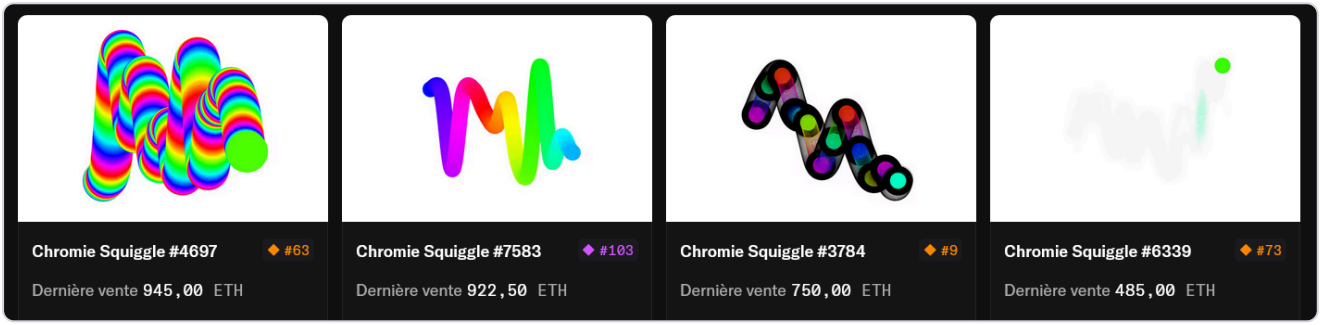
Le tournant blockchain

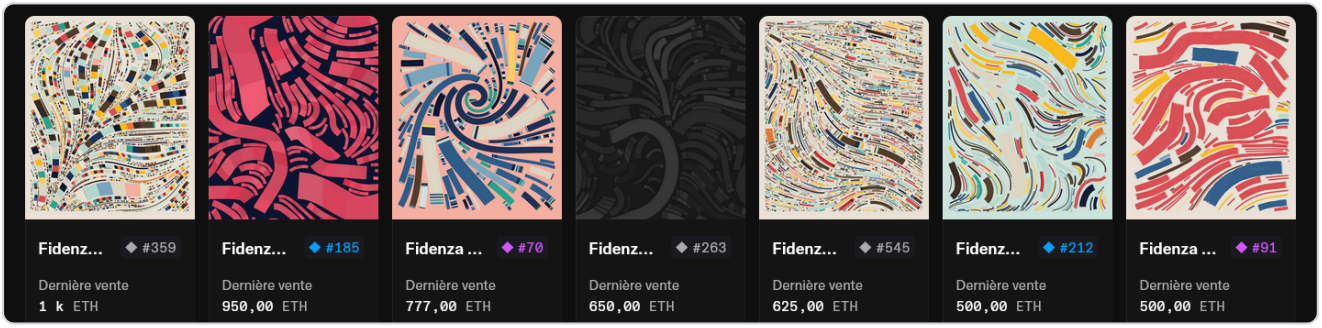
La blockchain apporte la **preuve d’unicité** et l’**exécution publique** d’algorithmes :

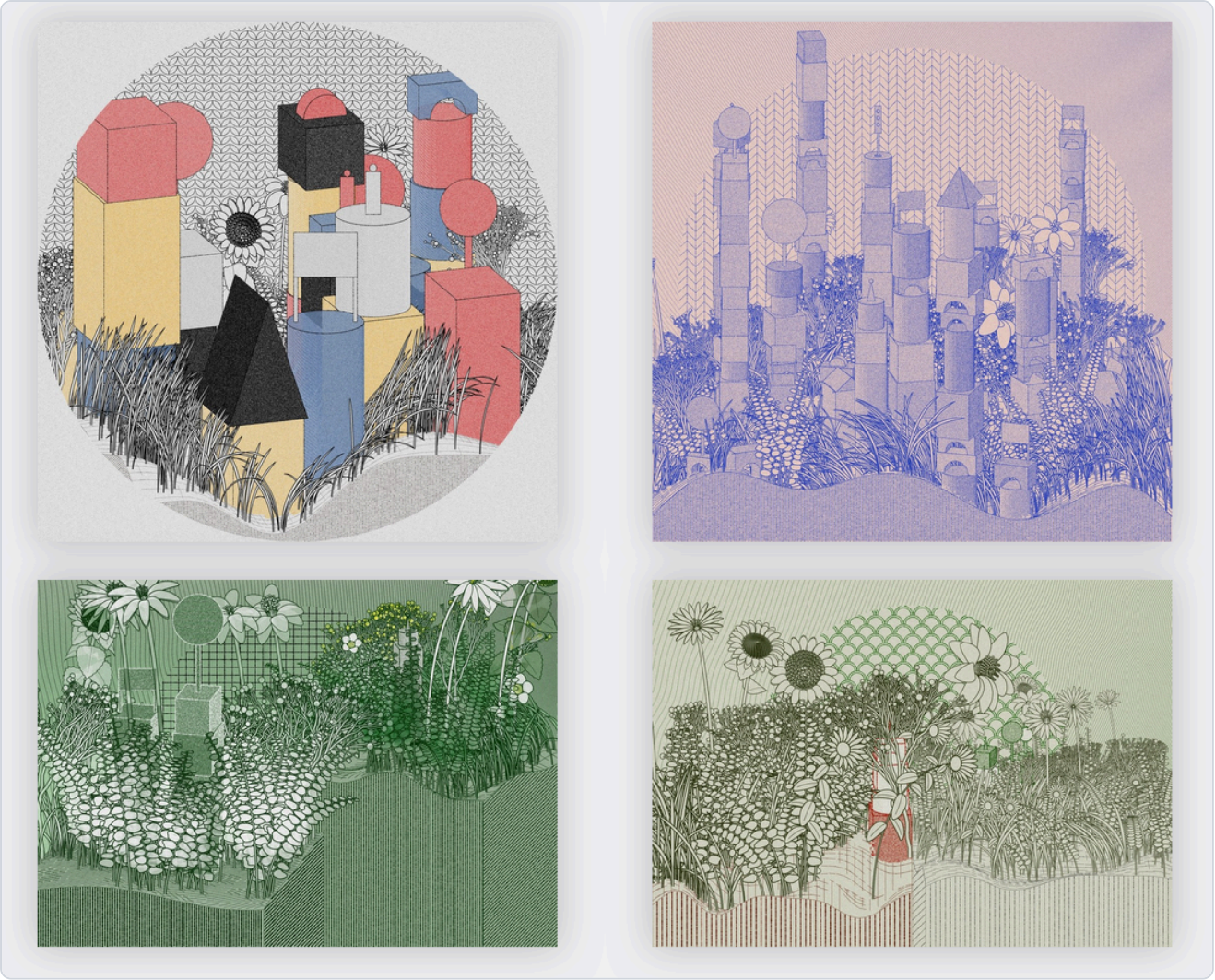
- **CryptoPunks (2017)** : 10k portraits générés par combinaison de traits, icône de la culture NFT.
- **Long-form** : une **seule** œuvre-algorithme génère **toute une série** d’outputs au mint, sans sélection manuelle (exécution autonome, paramètres variés, édition limitée).

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.