Le paradigme objet

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-02

Table des matières

1 Feuille de route : le paradigme avant le langage 1
2 Aux origines de l'approche objet 2
3 L'objet : identité, état, comportement 2
4 Messages et contrat : utiliser sans lire 3
5 Classes : un patron, des exemplaires 5
6 Spécialiser et polymorphiser 6
7 UML : modéliser très brièvement 7
8 Le même concept, quatre autres langages 7
9 Java : introduction globale 13
10 Avant-goût : un premier programme 14

1 Feuille de route : le paradigme avant le langage

Cette première séance pose le **vocabulaire du paradigme objet** — objet, message, classe, encapsulation, spécialisation, polymorphisme — et le matérialise dans quelques lignes de code que nous écrivons après avoir nommé les concepts. Le parti pris est délibéré : nous ne commençons pas par la syntaxe d'un langage, parce que les idées que nous allons manipuler existent bien avant Java et se retrouvent, avec d'autres choix d'écriture, dans la plupart des langages contemporains.

Quatre moments. D'abord **d'où vient l'approche objet** et quel problème elle a résolu. Ensuite **ce qu'est un objet** : une identité, un état, un comportement, et un contrat de messages. Puis **comment le même concept est noté et décliné** : le diagramme de classes comme langage de modélisation, et un panorama de quelques langages autour d'une seule et même classe. Enfin **Java lui-même**, en dernier volet et sous son aspect descriptif : ce qu'il est, avec quels outils on le pratique. L'histoire du langage, ses origines et la chronologie de ses versions ne sont pas traitées ici : elles font l'objet d'une séance entière.

2 Aux origines de l'approche objet

La programmation orientée objet trouve ses origines dans les problèmes de **simulation** des années 1960 : comment représenter un système composé de parties qui évoluent chacune de leur côté, tout en restant cohérentes avec les autres ? La réponse apportée — regrouper données et comportements dans une même entité qui communique avec ses voisines — s'est diffusée bien au-delà de ce domaine initial. On la retrouve en programmation (Smalltalk, C++, Java, etc.), en modélisation avec UML, et jusqu'aux bases de données avec les SGBDOO, aujourd'hui presque disparus, et les SGBDRO.

Le concept majeur de l'approche est **l'encapsulation**, et le corollaire en est la communication par **messages** : un objet envoie à un autre un message appartenant à un ensemble prédéfini, sans avoir à connaître la façon dont ce message sera exécuté. Les objets de même forme sont regroupés en **classes**, et trois autres concepts venus du même contexte — composition, délégation, héritage — permettent d'organiser les modèles de grande taille (nous n'en retiendrons que la spécialisation, §5 ; le reste vous sera présenté plus tard).

Deux lectures d'un objet coexistent et le cours les utilisera toutes les deux. La lecture **structurelle** le décrit comme l'instance d'un type, c'est-à-dire d'une classe, qui masque une structure derrière des opérations. La lecture **conceptuelle** le décrit comme une entité du monde réel, susceptible d'être généralisée ou spécialisée — c'est cette seconde lecture qui nous conduira, plus bas, à la notion de spécialisation de patron.

Notre objet ici est l'histoire du **paradigme**. Celle du langage Java est une autre séance.

3 L'objet : identité, état, comportement

Définition 1 (Objet). Un objet, en programmation orientée objet, représente une entité dotée d'une identité unique, d'un état défini par des attributs et d'un comportement décrit par des méthodes.

Les **attributs** sont des données nommées et valorisées qui décrivent l'état interne de l'objet. Les **méthodes** sont ce par quoi l'objet réagit aux messages qu'il reçoit ; ces réactions peuvent modifier son état. L'**encapsulation** est le principe qui masque cet état interne et ne livre que les méthodes d'accès et de modification choisies par l'auteur de la classe.

Ce masque a deux gains et un prix. Gain de **sûreté** : un code extérieur ne peut pas écrire une valeur invérifiée dans un état qu'il ne voit pas. Gain d'**évolutivité** : tant que les opérations fournies se comportent comme avant, leur implémentation peut être entièrement réécrite sans que le code appelant change. Prix : chaque

lecture d'attribut passe par un appel de méthode, ce qui peut introduire une légère surcharge. Nous verrons que les autres langages du panorama paient ce choix autrement : en lignes de code, pas en temps.

4 Messages et contrat : utiliser sans lire

Chaque objet possède un ensemble de méthodes, publiques ou privées. Les méthodes **publiques** constituent l'**interface** de l'objet : la liste des messages qu'il peut recevoir et traiter. Un appelant qui connaît cette liste peut utiliser l'objet sans avoir lu une seule ligne de son implémentation — c'est la définition opérationnelle de l'encapsulation.

Considérez un objet `unePersonne` muni de quatre méthodes publiques : `unePersonne.saluer();` lui demande de saluer ; `unePersonne.donnerPrenom("Pierre");` lui attribue le prénom « Pierre » ; `unePersonne.afficherPrenom();` affiche ce prénom ; `unePersonne.retournerPrenom();` le rend comme valeur. Un seul de ces messages modifie l'état ; les trois autres se contentent d'afficher ou de rendre une information : c'est déjà, en miniature, la distinction entre ce qui change l'objet et ce qui se contente de le lire.

Nous allons maintenant voir ce contrat tenu par un objet réel, dont le code s'affiche ici dans son intégralité.

Voici le type, puis les quatre messages — `saluer()`, `donnerPrenom(p)`, `afficherPrenom()`, `retournerPrenom()` — exécutés dans la seconde cellule.

```
class Personne {
    String prenom = "";
    void saluer() { System.out.println("Bonjour, je suis " + prenom); }
    void donnerPrenom(String p) { prenom = p; }
    String retournerPrenom() { return prenom; }
    void afficherPrenom() { System.out.println(prenom); }
}
```

```
Personne unePersonne = new Personne();
unePersonne.donnerPrenom("Pierre");
unePersonne.saluer();
unePersonne.afficherPrenom();
unePersonne.retournerPrenom();
```

```
Bonjour, je suis Pierre
Pierre
```

```
unePersonne.retournerPrenom();: Pierre
```

Le contrat de cette carte tient en quatre messages publics — `valider()`, `recharger(voyages)`, `voyagesRestants()`, `voyagesConsommes()` — et un constructeur qui reçoit un nombre de voyages initiaux. Nous n'avons pas lu son code, et nous n'allons pas le lire : les lignes ci-dessous envoient ces messages à un objet, rien de plus. Deux notations vous concernent peu ici — `new` et le typage de la boucle `for` ; vous les verrez en détail plus tard. Lisez seulement les messages envoyés et les réponses reçues. Le constructeur reçoit 10 voyages, puis une boucle tente 12 validations consécutives — les deux dernières itérations doivent donc buter sur le refus.

```
CarteDeTransport carte = new CarteDeTransport(10);
for (int i = 1; i ⩽ 12; i++) {
    boolean ok = carte.valider();
    if (ok) {
            System.out.println("Voyage " + i + " : validé — restants=" +
carte.voyagesRestants() + ", consommés=" + carte.voyagesConsommes());
    } else {
            System.out.println("Voyage " + i + " : REFUSÉ — restants=" +
carte.voyagesRestants() + ", consommés=" + carte.voyagesConsommes());
    }
}
```

```
Voyage 1 : validé — restants=9, consommés=1
Voyage 2 : validé — restants=8, consommés=2
Voyage 3 : validé — restants=7, consommés=3
Voyage 4 : validé — restants=6, consommés=4
Voyage 5 : validé — restants=5, consommés=5
Voyage 6 : validé — restants=4, consommés=6
Voyage 7 : validé — restants=3, consommés=7
Voyage 8 : validé — restants=2, consommés=8
Voyage 9 : validé — restants=1, consommés=9
Voyage 10 : validé — restants=0, consommés=10
Voyage 11 : REFUSÉ — restants=0, consommés=10
Voyage 12 : REFUSÉ — restants=0, consommés=10
```

Le comportement observé mérite d'être lu avec attention, car il illustre deux idées que le reste du cours va réutiliser. La première est l'utilité du contrat : nous avons écrit une boucle qui adresse trois messages — `valider()`, `voyagesRestants()`, `voyagesConsommes()` — sans avoir lu la moindre ligne de leur implémentation. C'est la définition même de l'encapsulation : l'état interne est masqué derrière des opérations.

La seconde idée est visible dans les deux derniers résultats. À l'itération 10, `restants` atteint 0 et `consommés` vaut 10. À l'itération 11, `valider()` renvoie `false` et, surtout, **les deux compteurs ne bougent plus** : le refus n'a rien consommé, et `restants` ne devient jamais négatif. La méthode `valider` protège ainsi la cohérence des compteurs — un voyage consommé correspond toujours à un voyage validé — sans que l'appelant ait à vérifier quoi que ce soit avant chaque validation. Notons enfin la forme du refus : `valider()` ne lève pas d'exception, elle répond par un booléen. Le choix entre « situation attendue » et « anomalie » est ici explicite, et nous le retrouverons lorsque nous parlerons des exceptions.

`recharger` mérite votre attention : contrairement au constructeur, elle ne vérifie rien — une recharge négative casserait l'invariant de `valider`. C'est délibéré : la séance consacrée aux erreurs et aux exceptions reprendra ce problème, sous un angle général.

5 Classes : un patron, des exemplaires

Certains objets possèdent des propriétés très proches, voire identiques : les voitures d'un programme partagent marque, modèle, année et couleur ; les cartes de transport partagent un solde et un compteur. Une **classe** est un ensemble d'objets présentant des similitudes en termes d'identification, de comportement ou de description de l'état dans une application ; un objet appartenant à cette classe est appelé une **instance** de la classe. La classe `Voiture`, par exemple, décrit quatre attributs et les opérations `demarrer()`, `arreter()`, `accelerer(vitesse)` ; `v1` et `v2` en sont deux instances aux valeurs distinctes.

Une classe remplit trois rôles que nous venons d'observer. Elle **décrit** les propriétés de ses instances, par ses variables d'état et ses méthodes. Elle **fabrique** de nouveaux objets par ses **constructeurs**, qui initialisent l'état de l'exemplaire créé avec des valeurs de départ. Elle **partage** des informations entre instances, en associant certaines propriétés à la classe elle-même plutôt qu'aux objets.

Repartons de la démonstration : le contrat de la carte est **unique**, le code est écrit une seule fois, et pourtant le programme peut créer autant d'objets qu'il veut à partir de cette seule description.

```
CarteDeTransport carteEtudiante = new CarteDeTransport(10);
CarteDeTransport carteEnseignant = new CarteDeTransport(30);
System.out.println("avant        :        etudiante        restants="        +
carteEtudiante.voyagesRestants()    +    "    /    enseignant    restants="    +
carteEnseignant.voyagesRestants());
carteEtudiante.valider();
System.out.println("après        :        etudiante        restants="        +
```

```
carteEtudiante.voyagesRestants()        +        ",        consommés="        +
carteEtudiante.voyagesConsommes()    +    "    /    enseignant    restants="
+      carteEnseignant.voyagesRestants()      +      ",      consommés="      +
carteEnseignant.voyagesConsommes());
```

```
avant : etudiante restants=10 / enseignant restants=30
après : etudiante restants=9, consommés=1 / enseignant restants=30, consommés=0
```

Ces deux lignes d'appel à `new` résument le couple classe/objet. La classe `CarteDeTransport` est un texte unique ; les deux variables `carteEtudiante` et `carteEnseignant` désignent deux objets distincts, chacun muni de **son propre** couple de compteurs. Valider un voyage sur la carte étudiante laisse intact le solde de la carte enseignant : l'état est par instance. C'est la raison pour laquelle le diagramme de classes ne décrit jamais des valeurs, mais des valeurs possibles.

Cette distinction prépare un contraste important, que le cours exploitera longuement. Tous les membres d'une classe ne sont pas par instance : un membre marqué `static` — noté `{static} nombreDeVoitures: int` dans un diagramme de classes — appartient à la classe elle-même et non aux objets, et se trouve donc **partagé** par toutes les instances. La classe `Voiture` que nous manipulons deux sections plus bas, dans « Le même concept, quatre autres langages » montre exactement ce duo : quatre attributs par instance et un compteur partagé qui compte les voitures créées. Confondre les deux est l'erreur classique — et la section finale de cette leçon la vérifie.

6 Spécialiser et polymorphiser

Décrire des objets isolés ne suffit pas à modéliser un système réel : les entités se ressemblent sans être identiques. Une carte de transport, un abonnement de vélo partagé et un pass annuel de piscine sont trois choses différentes, mais toutes trois consomment des droits et refusent quand le solde est épuisé. Le paradigme objet répond à cette situation par la **spécialisation** : une classe peut en généraliser d'autres, auquel cas elle lui doit son état et son comportement de base, et n'ajoute que la différence.

La conséquence la plus féconde est le **polymorphisme**. Si trois objets spécialisent la même généralité, alors ils acceptent tous les trois le même message — et chacun y répond **à sa manière**, avec son propre état et sa propre logique. L'appelant, lui, n'a pas à savoir laquelle des réponses il reçoit : il écrit son code contre le contrat commun. Ajouter une quatrième spécialisation ne demandera donc aucune modification de ce code.

Ces deux idées — spécialiser un patron, recevoir un même message sous plusieurs formes — sont des idées du paradigme, pas des idées de Java. Nous les retrouvons

dans les langages du panorama ci-dessous, et les mécanismes d'écriture propres au langage — déclarer une spécialisation, réutiliser le comportement de la généralité, redéfinir une méthode — font l'objet des séances consacrées à l'héritage.

7 UML : modéliser très brièvement

Le **diagramme de classes** est la notation par laquelle UML — langage de modélisation — décrit une classe. En voici un, tracé d'après le contrat que nous avons déjà utilisé sans le lire.

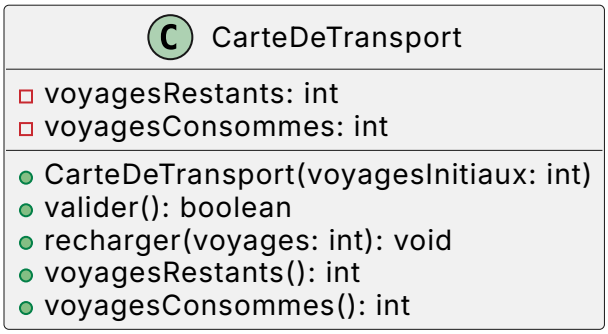


Figure 1. – Diagramme de classes de CarteDeTransport

Un diagramme de classes UML se lit en trois zones verticales : le nom de la classe en haut, la liste de ses **attributs** au milieu, la liste de ses **méthodes** en bas. Il décrit la structure complète de la classe, parties privées comprises ; mais seule sa partie **publique**, celle des membres préfixés `+`, constitue le contrat — exactement les messages que nous avons envoyés dans la démonstration, sans un mot de la façon dont l'objet s'y prend. Deux conventions demandent une attention immédiate parce qu'elles diffèrent de celles de Java : UML écrit `nom: Type` là où Java écrit `Type nom`, et un signe préfixe porte la visibilité, `-` pour privé, `+` pour public. Ce que le diagramme tait volontairement, enfin, c'est l'algorithme : le nombre de lignes, les conditions, les variables locales. Un lecteur du diagramme peut déjà utiliser la carte ; il ne peut pas encore la réécrire.

8 Le même concept, quatre autres langages

Une seule classe, `Voiture`, écrite dans cinq langages. Les concepts sont identiques ; les choix de rendu ne le sont pas, et chacun est raisonnable dans son contexte. Voici d'abord la version Java, qui nous sert de référence — exécutée, avec ses quatre attributs privés, ses accesseurs, ses trois comportements et son compteur partagé.

```
public class Voiture {
    private String marque;
    private String modele;
```

```
    private int annee;
    private String couleur;
    private static int nombreDeVoitures = 0;

    public Voiture(String marque, String modele, int annee, String couleur) {
        this.marque = marque;
        this.modele = modele;
        this.annee = annee;
        this.couleur = couleur;
        nombreDeVoitures++;
    }

    public void demarrer() {
        System.out.println(marque + " " + modele + " démarre.");
    }

    public void arreter() {
        System.out.println(marque + " " + modele + " s'arrête.");
    }

    public void accelerer(int vitesse) {
        System.out.println(marque + " " + modele + " accélère à " + vitesse + "
km/h.");
    }

    public String getMarque() {
        return marque;
    }

    public String getModele() {
        return modele;
    }

    public int getAnnee() {
        return annee;
    }

    public String getCouleur() {
        return couleur;
    }

    public static int getNombreDeVoitures() {
        return nombreDeVoitures;
```

```
    }
}
```

Grille de lecture. Chaque snippet ci-dessous montre le MÊME modèle `Voiture` — un extrait comparatif : dans ce carnet, seules les cellules Java sont exécutées par le noyau ; ceux des quatre autres langages se lisent sans s'exécuter. N'y lisez que deux choses : comment les attributs sont déclarés et comment le constructeur s'écrit — le reste est identique. La notation de la spécialisation, elle, figure dans le tableau comparatif en fin de section.

Python

```
class Voiture:
    nombre_de_voitures = 0

    def __init__(self, marque, modele, annee, couleur):
        self.marque = marque
        self.modele = modele
        self.annee = annee
        self.couleur = couleur
        Voiture.nombre_de_voitures += 1

    def demarrer(self):
        print(f"{self.marque} {self.modele} démarre.")

    def arreter(self):
        print(f"{self.marque} {self.modele} s'arrête.")

    def accelerer(self, vitesse):
        print(f"{self.marque} {self.modele} accélère à {vitesse} km/h.")

    @classmethod
    def get_nombre_de_voitures(cls):
        return cls.nombre_de_voitures

v = Voiture("Renault", "Clio", 2021, "bleue")
v.demarrer()
v.accelerer(90)
print(v.__dict__)
print(Voiture.get_nombre_de_voitures())
```

Chaque extrait ne construit que les objets qu'il déclare lui-même : le compteur qu'on y lit repart de zéro d'un langage à l'autre.

Kotlin

```kotlin
// le reste est identique — seule la notation change
class Voiture(
    private val marque: String,
    private val modele: String,
    private val annee: Int,
    private val couleur: String
) {
    companion object {
        var nombreDeVoitures: Int = 0
            private set
    }

    init {
        nombreDeVoitures++
    }

    fun demarrer() {
        println("$marque $modele démarre.")
    }

    fun arreter() {
        println("$marque $modele s'arrête.")
    }

    fun accelerer(vitesse: Int) {
        println("$marque $modele accélère à $vitesse km/h.")
    }
}
```

TypeScript

```typescript
// le reste est identique — seule la notation change
class Voiture {
    private static nombreDeVoitures: number = 0;

    constructor(
        private marque: string,
        private modele: string,
        private annee: number,
        private couleur: string
    ) {
        Voiture.nombreDeVoitures++;
```

```
    }

    public demarrer(): void {
        console.log(`${this.marque} ${this.modele} démarre.`);
    }

    public arreter(): void {
        console.log(`${this.marque} ${this.modele} s'arrête.`);
    }

    public accelerer(vitesse: number): void {
       console.log(`${this.marque} ${this.modele} accélère à ${vitesse} km/h.`);
    }

    public static getNombreDeVoitures(): number {
        return Voiture.nombreDeVoitures;
    }
}
```

JavaScript — rendu dans sa forme prototype, celle sur laquelle tout repose : la syntaxe `class` d'ES6 est du sucre sur les prototypes.

```
// le reste est identique — seule la notation change
// Constructeur Voiture
function Voiture(marque, modele, annee, couleur) {
    this.marque = marque;
    this.modele = modele;
    this.annee = annee;
    this.couleur = couleur;
    Voiture.nombreDeVoitures++;
}

// Propriété statique pour compter le nombre de voitures
Voiture.nombreDeVoitures = 0;

// Méthodes de la classe Voiture
Voiture.prototype.demarrer = function() {
    console.log(this.marque + " " + this.modele + " démarre.");
};

Voiture.prototype.arreter = function() {
    console.log(this.marque + " " + this.modele + " s'arrête.");
};
```

```
Voiture.prototype.accelerer = function(vitesse) {
    console.log(this.marque + " " + this.modele + " accélère à " + vitesse + "
km/h.");
};

// Méthode statique pour obtenir le nombre de voitures
Voiture.getNombreDeVoitures = function() {
    return Voiture.nombreDeVoitures;
};
```

Le tableau ci-dessous range les cinq versions selon six axes qui font vraiment différence. Il faut le lire dans l'esprit suivant : **mêmes concepts, choix raisonnables différents**. Aucune de ces colonnes n'est une faute ; chacune est le compromis que le langage a choisi pour rester cohérent avec ce qu'il est.

Langage	Constructeur	Visibilité	Propriétés et accesseurs	Mutabilité par défaut	Données de classe	Spécialisation
Java	`Voiture(...)` : le nom de la classe, aucun type de retour	`private` / `public` écrits sur chaque membre	champs privés + accesseurs écrits à la main	attributs modifiables librement dès qu'ils sont visibles	mot-clé `static`	`extends`
Python	`def __init__(self, ...)`	aucune visibilité imposée (convention `_x`, non bloquée)	attributs posés sur `self`, publics ; `@property` si besoin	ouverte, y compris l'ajout d'attributs à chaud	attribut de classe + `@classmethod`	`class Voiture(Véhicule)`
Kotlin	constructeur primaire dans les parenthèses de la classe	`private` / `public`, `public` par défaut	`val` / `var` : champ et accesseur en une seule ligne	`val` en lecture seule par défaut	`companion object`	`:`

Langage	Constructeur	Visibilité	Propriétés et accesseurs	Mutabilité par défaut	Données de classe	Spécialisation
TypeScript	`constructor(private marque: string)`	`private` / `public`, effacés à la compilation	le paramètre de constructeur devient la propriété	`readonly` en option, sinon ouverte	mot-clé `static`	`extends`
JavaScript	fonction constructeur appelée avec `new`	aucune	propriétés posées sur `this`	entièrement ouverte	propriété attachée à la fonction	`prototypes`

Ces cinq écritures ne sont pas traduites ligne à ligne ici : chaque langage a ses habitudes, que les séances dédiées reprendront.

Sur la première ligne, reconnaissez le prix de l'encapsulation évoqué plus haut : nos quatre accesseurs sont vingt lignes que Kotlin écrit en quatre mots, et que Python n'écrit pas du tout — parce que Python fait le choix inverse, celui d'une confiance explicite dans l'appelant. De même, la visibilité de TypeScript est vérifiée par le compilateur puis **effacée** : le JavaScript produit ne s'en souvient plus, ce qui est un choix d'interopérabilité, pas une distraction. Quant à la colonne « mutabilité », elle dit qui, par défaut, a le droit d'écrire : Python et JavaScript ouvrent tout et comptent sur la discipline ; Kotlin ferme par défaut et demande `var` pour ouvrir ; Java, dans le code que vous avez sous les yeux, ouvre ce qu'il veut bien exposer et masque le reste.

Et l'héritage dans tout cela ? Les cinq langages le notent chacun à leur façon — et le JavaScript nous le montre en creux, puisque la forme prototype que nous avons retenue laisse voir le mécanisme que la syntaxe `class` d'ES6 se contente d'habiller.

9 Java : introduction globale

Java est un langage de programmation orienté objet, ce qui signifie que sa façon naturelle de découper un problème est d'y faire collaborer des objets — nous venons de le pratiquer. Il se décrit aussi, de manière positive, par six caractéristiques. Sa **portabilité** : un même programme s'exécute sur des machines très différentes, et le responsable en est la machine virtuelle Java, la JVM — comment et pourquoi ce choix a été fait est précisément le sujet de la séance suivante. Sa **sûreté**, tenue par un typage fort et par des vérifications exercées avant et pendant l'exécution. Sa capacité **multi-tâche**, qui permet de mener plusieurs traitements de front. Son

caractère **dynamique**, qui autorise le chargement de classes pendant l'exécution plutôt qu'à la mise en route. Enfin la **complétude** de sa bibliothèque standard, qui couvre les interfaces graphiques, le réseau, les bases de données et bien d'autres domaines, et qui explique une grande part de sa présence durable dans les systèmes d'information.

Nous restons ici descriptive : le détail de la chaîne qui conduit d'un fichier source à un programme qui tourne appartient à la séance d'origines.

Pour travailler en Java, deux outils suffisent au départ : un éditeur de texte et l'outillage qui traduit le source en programme exécutable. Nous l'installons et nous l'utilisons à la main à la séance « Mise en œuvre de Java », qui est la suite mécanique de celle-ci. Dès que les projets deviennent plurifichiers, s'ajoutent les outils de gestion de dépendances et de construction — Maven et Gradle en tête — que nous aborderons aux séances consacrées à la construction de projet. Enfin, la majorité des développements professionnels se fait dans un environnement intégré, qui offre navigation dans les sources, saisie assistée et débogage : Eclipse, projet open source soutenu par IBM (https://www.eclipse.org) ; IntelliJ IDEA de JetBrains, réputé pour son éditeur, en versions communautaire et commerciale (https://www.jetbrains.com/idea) ; ou Visual Studio Code, plus léger, avec ses extensions Java (https://code.visualstudio.com/docs/languages/java).

Ce que Java doit à ses choix se lit en trois avantages et un coût. La **portabilité** d'abord : un programme écrit une fois circule sur des systèmes très différents. La **sûreté** ensuite : le langage et son environnement d'exécution vérifient ce que le programme fait, au lieu de supposer que l'auteur a raison. La **richesse des bibliothèques** enfin, qui fournit des fonctionnalités prêtes à l'emploi dans presque tous les domaines. Le coût est celui de la même médaille : une exécution qui passe par un niveau d'abstraction peut être plus lente que celle d'un langage compilé directement pour le processeur, comme C ou C++ ; les optimisations modernes réduisent fortement l'écart, mais ne le suppriment pas lorsque la vitesse est critique, et recourir à du code natif se paie alors en portabilité.

Séance suivante : remonter aux sources du langage — le projet Green, le pari de la machine virtuelle, les guerres de licences et le train des versions — dans « Java : d'où ça vient, pourquoi c'est comme ça » (`01_java_L_origines.qmd`). L'installation de l'outil et la manipulation à la main de la chaîne de compilation viendront juste après, dans « Mise en œuvre de Java » (`02_java_P_mecanique.qmd`).

10 Avant-goût : un premier programme

Vérifions dans le code, une dernière fois, le duo observé plus haut : deux objets, un compteur partagé.

```java
Voiture v1 = new Voiture("Clio", "Clio II", 2020, "bleue");
Voiture v2 = new Voiture("Boxer", "Dingo", 1975, "rouge");
System.out.println("Total de voitures créées : " +
Voiture.getNombreDeVoitures());
v1.demarrer();
```

```
Total de voitures créées : 2
Clio Clio II démarre.
```

Cet avant-goût se travaille comme un atelier, dans le carnet :

1. **Lisez** la cellule ci-dessus, et retrouvez dans la définition de `Voiture` donnée plus haut les quatre attributs, le compteur de classe et les messages.
2. **Prédisez** ligne à ligne la sortie affichée : quelle ligne de code produit quelle ligne de texte ?
3. **Comparez** avec la démo Python du panorama — un extrait donné à titre de comparaison, non exécuté ici : d'après son code, pourquoi le total y serait-il 1, alors qu'il s'affiche 2 dans la cellule ci-dessus ?
4. **Anticipez** : si l'on créait une troisième voiture, que vaudrait le compteur, et pourquoi ?
5. **Relisez** la définition de `Voiture` : chaque ligne doit désormais vous parler d'attributs, de constructeur ou de messages.

Le total affiché vaut 2 alors que deux objets distincts ont été construits : chacun porte ses quatre attributs, le compteur, lui, n'existe qu'une fois — c'est une donnée de classe. Et `v1.demarrer()` n'imprime que les informations de `v1`. Ce programme s'exécute ici dans le noyau du carnet, ce qui nous a dispensés de toute installation ; vous compilerez et exécuterez vous-mêmes votre premier programme Java à une séance ultérieure, celle de la mise en œuvre. Nous, nous passons à l'histoire : comprendre d'où vient ce langage explique pourquoi il est écrit ainsi.