RAPPEL SUR LES OBJETS LITTÉRAUX

```
1   const voiture = {
2     marque: "Toyota",
3     modele: "Corolla",
4     demarrer: function() {
5         console.log("La voiture démarre");
6     }
7   };
8
9   voiture.demarrer(); // Affiche : La voiture démarre
10
11  console.log(voiture.marque); // Toyota
12  voiture.modele = "Yaris";
13  console.log(voiture.modele); // Yaris
```

INTRODUCTION AUX CLASSES EN JAVASCRIPT

En JavaScript, une classe est une sorte de "plan" pour créer des objets. Elle permet de regrouper des données (propriétés) et des fonctions (méthodes) au sein d'une seule structure.

Les classes facilitent l'organisation du code et encouragent une approche modulaire, ce qui est particulièrement utile pour les grandes applications.

Syntaxe de base d'une classe

Voici la structure d'une classe en JavaScript :

```
class NomDeClasse {
    // Le constructeur initialise les propriétés
    constructor(param1, param2) {
      this.propriete1 = param1;
      this.propriete2 = param2;
    }

    // Méthode d'instance
    methode() {
      console.log("Méthode de l'instance");
    }
  }
```

Exemple : Créer une classe Personne avec un constructeur qui initialise le nom et l'âge :

```
1   class Personne {
2       constructor(nom, age) {
3         this.nom = nom;
4         this.age = age;
5       }
6
7       saluer() {
8         console.log(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
9       }
10    }
11
12    const personne1 = new Personne("Alice", 30);
13    personne1.saluer(); // "Bonjour, je m'appelle Alice et j'ai 30 ans."
```

ENCAPSULATION : PROPRIÉTÉS ET MÉTHODES PRIVÉES

```
class CompteBancaire {
    #solde;

    constructor(montantInitial) {
      this.#solde = montantInitial;
    }

    deposer(montant) {
      this.#solde += montant;
    }

    obtenirSolde() {
      return this.#solde;
    }
  }

  const compte = new CompteBancaire(100);
  compte.deposer(50);
  console.log(compte.obtenirSolde()); // 150
```

Depuis les dernières versions de JavaScript, il est possible de créer des propriétés privées (qui ne sont accessibles qu'à l'intérieur de la classe) en utilisant le symbole #

Dans cet exemple, #solde est privé et ne peut être accédé directement en dehors de la classe.

HÉRITAGE

```
class Utilisateur {
    constructor(nom, email) {
      this.nom = nom;
      this.email = email;
      this.dateInscription = new Date();
    }

    afficherProfil() {
      console.log(`Nom: ${this.nom} | Email: ${this.email}`);
    }
  }

  class Admin extends Utilisateur {
    constructor(nom, email) {
      super(nom, email);
      this.role = "admin";
    }

    supprimerUtilisateur(utilisateur) {
      console.log(`${utilisateur.nom} a été supprimé par l'admin ${this.nom}`);
    }
  }

  const utilisateur = new Utilisateur("Alice", "alice@example.com");
  utilisateur.afficherProfil();

  const admin = new Admin("Bob", "bob@admin.com");
  admin.afficherProfil();
  admin.supprimerUtilisateur(utilisateur);
```

L'héritage permet de créer de nouvelles classes basées sur des classes existantes. Une classe qui hérite d'une autre est appelée **sous-classe** et utilise le mot-clé extends. Le constructeur de la sous-classe appelle le constructeur de la classe parente avec super.

LES GETTERS ET SETTERS

Les **getters** et **setters** sont des méthodes spéciales qui permettent d'accéder ou de modifier les propriétés d'un objet de manière contrôlée. Ils jouent un rôle crucial dans l'**encapsulation**, un concept clé de la programmation orientée objet.

Les **getters** permettent de lire une valeur comme si c'était une propriété ordinaire, mais en coulisses, ils peuvent inclure une logique, comme des calculs ou des transformations.

Les **setters**, quant à eux, permettent de modifier une propriété tout en ajoutant une logique de validation ou de contrôle. Cela garantit que les données d'un objet restent cohérentes et valides. Par exemple, un setter peut empêcher l'attribution d'une valeur négative à une propriété censée représenter une quantité positive.

Les getters et setters rendent le code plus propre, intuitif et sécurisé, en exposant une interface claire tout en masquant les détails d'implémentation

LES GETTERS ET SETTERS

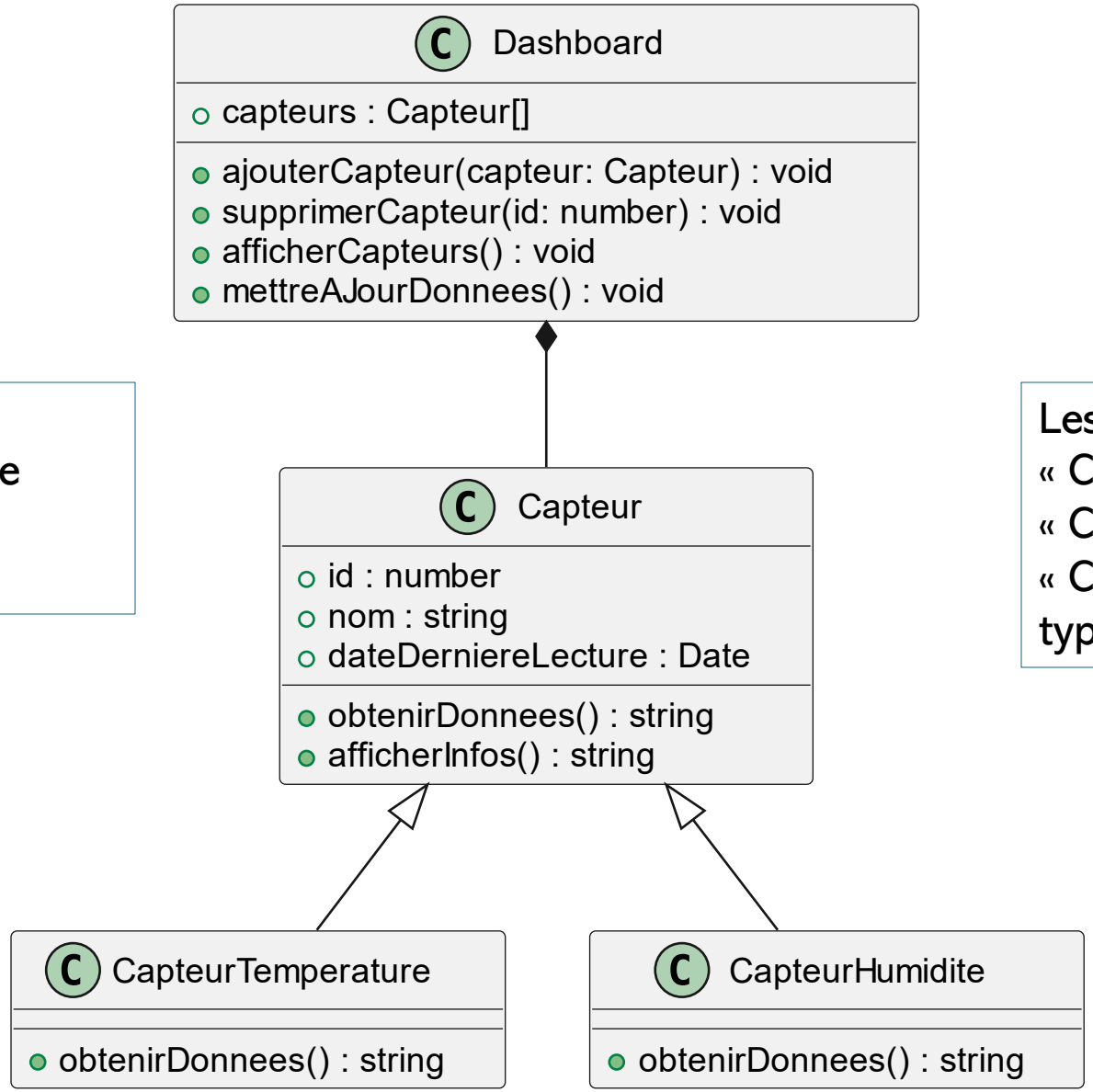
```
class Personne {
  #nom; // Champ privé

  constructor(nom) {
      this.#nom = nom;
  }

  // Getter pour accéder au nom
  get nom() {
      return this.#nom.toUpperCase(); // Retourne le nom en majuscules
  }

  // Setter pour modifier le nom avec validation
  set nom(nouveauNom) {
      if (nouveauNom && nouveauNom.trim().length > 0) {
          this.#nom = nouveauNom;
      } else {
          console.log("Le nom ne peut pas être vide !");
      }
  }
}

const personne = new Personne("Alice");
console.log(personne.nom); // ALICE
personne.nom = "Bob"; // Met à jour le nom
console.log(personne.nom); // BOB
personne.nom = ""; // Le nom ne peut pas être vide !
```

POLYMORPHISME

Le **polymorphisme** est un concept clé en programmation orientée objet, qui signifie "plusieurs formes". En JavaScript, il permet à une méthode ou une fonction d'agir différemment en fonction du type de données ou de l'objet avec lequel elle est utilisée. Ce mécanisme est généralement implémenté via l'**héritage**. Par exemple, une méthode définie dans une classe parent peut être **redéfinie** dans une classe enfant pour adapter son comportement. Cela permet d'écrire du code générique et réutilisable, tout en offrant la flexibilité de personnaliser le comportement lorsque cela est nécessaire.

```
1  class Animal {
2    parler() {
3        console.log("Cet animal fait un bruit.");
4    }
5  }
6
7  class Chien extends Animal {
8    parler() {
9        console.log("Le chien aboie.");
10   }
11 }
12
13 class Chat extends Animal {
14   parler() {
15       console.log("Le chat miaule.");
16   }
17 }
18
19 const animaux = [new Chien(), new Chat(), new Animal()];
20 animaux.forEach(animal => animal.parler());
21 // Résultat :
22 // Le chien aboie.
23 // Le chat miaule.
24 // Cet animal fait un bruit.
```

DASHBOARD DE CAPTEURS



La liaison entre la classe « Dashboard » et la classe « Capteur» est de type **composition**.

Les liaisons entre la classe « Capteur » et les classes « CapteurTemperature » et « CapteurHumidite » sont de type **héritage**

CLASSE DE BASE : CAPTEUR

Pour un exemple basé sur des capteurs, on peut imaginer une application qui gère un dashboard affichant des données issues de différents capteurs connectés (comme des capteurs de température, d'humidité, de pression, etc.). Nous allons créer une classe Capteur de base, avec des sous-classes pour chaque type de capteur. Cela permet d'ajouter ou de retirer facilement des capteurs du dashboard.

```
1   class Capteur {
2       constructor(id, nom) {
3         this.id = id;
4         this.nom = nom;
5         this.dateDerniereLecture = null;
6       }
7
8       obtenirDonnees() {
9         this.dateDerniereLecture = new Date();
10        return "Données du capteur";
11      }
12
13      afficherInfos() {
14        console.log(`[${this.id}] ${this.nom}`);
15      }
16    }
```

SOUS-CLASSE DE CAPTEUR :

Ensuite, on peut créer des sous-classes pour différents types de capteurs. Par exemple, CapteurTemperature pour mesurer la température, et CapteurHumidite pour l'humidité.

```
class CapteurTemperature extends Capteur {
    obtenirDonnees() {
      super.obtenirDonnees();
      const temperature = (Math.random() * 30).toFixed(2); // Simule une température
      return `Température : ${temperature}°C`;
    }
  }

  class CapteurHumidite extends Capteur {
    obtenirDonnees() {
      super.obtenirDonnees();
      const humidite = (Math.random() * 100).toFixed(2); // Simule une humidité
      return `Humidité : ${humidite}%`;
    }
  }
```

Gestionnaire de Dashboard

Pour gérer les capteurs sur le dashboard, créons une classe Dashboard. Cette classe permet d'ajouter, de supprimer, et d'afficher les capteurs.

```
class Dashboard {
    constructor() {
      this.capteurs = [];
    }

    ajouterCapteur(capteur) {
      this.capteurs.push(capteur);
      this.afficherCapteurs();
    }

    supprimerCapteur(id) {
      this.capteurs = this.capteurs.filter(capteur => capteur.id !== id);
      this.afficherCapteurs();
    }

```

```
afficherCapteurs() {
      const dashboardDiv = document.getElementById("dashboard");
      dashboardDiv.innerHTML = ""; // Réinitialiser l'affichage

      this.capteurs.forEach(capteur => {
        const capteurDiv = document.createElement("div");
        capteurDiv.className = "capteur";
        capteurDiv.id = `capteur-${capteur.id}`;

        capteurDiv.innerHTML = `
          <h3>${capteur.nom}</h3>
          <p>${capteur.afficherInfos()}</p>
          <p id="data-${capteur.id}">${capteur.obtenirDonnees()}</p>
          <button onclick="supprimerCapteur(${capteur.id})">Supprimer</button>
        `;

        dashboardDiv.appendChild(capteurDiv);
      });
    }

    mettreAJourDonnees() {
      this.capteurs.forEach(capteur => {
        const dataElement = document.getElementById(`data-${capteur.id}`);
        if (dataElement) {
          dataElement.innerText = capteur.obtenirDonnees();
        }
      });
    }
  }
```

Utilisation :

Maintenant, on peut créer des instances de capteurs et les ajouter au dashboard.

```
// Création du dashboard
const dashboard = new Dashboard();

// Fonctions pour gérer les capteurs dans le front
let compteurId = 1;

function ajouterCapteurTemperature() {
  const capteurTemp = new CapteurTemperature(compteurId++, "Capteur Température");
  dashboard.ajouterCapteur(capteurTemp);
}

function ajouterCapteurHumidite() {
  const capteurHumidite = new CapteurHumidite(compteurId++, "Capteur Humidité");
  dashboard.ajouterCapteur(capteurHumidite);
}

function supprimerCapteur(id) {
  dashboard.supprimerCapteur(id);
}

function mettreAJourDonnees() {
  dashboard.mettreAJourDonnees();
}
```

Dashboard en front

```html
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Dashboard Capteurs</title>
  <style>
    #dashboard {
      display: flex;
      flex-wrap: wrap; /* Permet de passer à la ligne si l'espace manque */
      gap: 10px; /* Espacement entre les capteurs */
    }

    .capteur {
      border: 1px solid #ccc;
      padding: 10px;
      margin: 5px;
      width: 300px;
      box-sizing: border-box; /* S'assure que padding et border sont inclus dans la largeur */
    }
  </style>
</head>
<body>

  <h1>Dashboard des Capteurs</h1>

  <div id="dashboard">
    <!-- Les capteurs seront affichés ici -->
  </div>

  <button onclick="ajouterCapteurTemperature()">Ajouter Capteur Température</button>
  <button onclick="ajouterCapteurHumidite()">Ajouter Capteur Humidité</button>
  <button onclick="mettreAJourDonnees()">Mettre à jour les données</button>

  <script src="dashboard.js"></script>

</body>
</html>
```