Chapitre 3
La programmation orientée objet

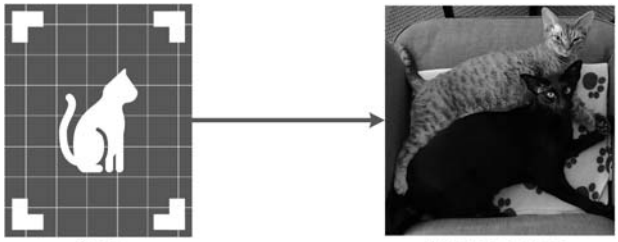
1. Introduction

La programmation orientée objet (aussi connu sous l'acronyme : POO) est un des paradigmes de développement les plus utilisés pour réaliser des applications. Dans les débuts de la programmation, les logiciels étaient réalisés à partir de séquences d'instructions s'exécutant les unes après les autres, c'est ce que l'on appelle la programmation impérative. Au fil du temps, les programmes informatiques sont devenus plus complexes, ce qui augmenta considérablement la difficulté à les maintenir et les faire évoluer. Dans les années 70, *Alan Kay* posa les bases de la programmation orientée objet. Ce paradigme est devenu rapidement populaire et est encore utilisé aujourd'hui comme fondation dans bon nombre de logiciels que nous utilisons au quotidien. Ce style de programmation permet d'organiser un logiciel en utilisant des sous-ensembles plus petits, communément appelés objets. Chacun de ces objets contient des données et de la logique qui lui sont propres. Les objets sont donc des parties autonomes d'un programme et ont des interactions les uns avec les autres pour le faire fonctionner.

TypeScript est un langage dit *multiparadigme*. Il permet donc de développer des applications en utilisant la programmation impérative, orientée objet ou encore fonctionnelle (cf. chapitre TypeScript et la programmation fonctionnelle). Les premières versions de TypeScript ont apporté de nombreuses capacités syntaxiques autour de la programmation orientée objet. C'est d'ailleurs l'une des raisons pour lesquelles le langage est devenu populaire auprès de certaines communautés de développeurs, notamment C# et Java. Cependant, il est important de garder en tête que même si TypeScript est syntaxiquement proche de Java et C#, il n'en est pas un équivalent. En réalité, les capacités objets de TypeScript suivent la norme ECMAScript, mais le langage dispose aussi de particularités qui lui sont propres. Ces différentes capacités seront décrites en détail dans ce chapitre.

2. Les classes

La première notion importante à apprendre lorsque l'on commence l'apprentissage de la programmation orientée objet est le concept de classe. Chaque objet doit être créé par une classe. Celle-ci peut être comparée à une notice de fabrication qui contient l'ensemble des informations nécessaires à la création d'un objet. Une fois définie, la classe est utilisée dans le programme pour créer un objet, on parle alors d'instance de classe. Il est possible de créer autant d'objets que l'on souhaite pour faire fonctionner un programme.



Les objets vont ensuite interagir les uns avec les autres pour faire fonctionner le programme.

Pour déclarer une classe en TypeScript, il faut utiliser le mot-clé `class`, lui donner un nom et ouvrir les accolades pour définir ses caractéristiques.

Remarque

Le concept de classe n'est pas nouveau en JavaScript. Le mot-clé `class` est un sucre syntaxique basé sur la notion de prototype dans le langage (cf. chapitre Types et instructions basiques). En ECMAScript 5, il est possible de définir une classe via une fonction constructrice. Celle-ci a la particularité de définir la structure des objets et de les créer. Depuis ECMAScript 2015, les classes sont privilégiées par rapport aux fonctions constructrices, car ces dernières ont une syntaxe peu intuitive.

Syntaxe :

```
class ClassName {
  // ...
}
```

Exemple :

```
class Employee {
  // ...
}
```

Remarque

Le nommage des classes en TypeScript suit la convention dite "PascalCase". Ce type de nommage précise que chaque mot composant le nom de la classe doit avoir sa première lettre en majuscule (exemple : `PaySplit`). Les classes et les interfaces sont les seuls éléments en TypeScript utilisant cette convention, cela permet de facilement les repérer lorsqu'on lit du code. Pour tous les autres éléments, c'est la convention de nommage "camelCase" qui est utilisée. Elle est similaire à la convention "PascalCase" à ceci près que la première lettre est en minuscule (exemple : `firstName`).

Une fois déclarée, la classe peut être utilisée pour créer de nouvelles instances. Chacune de ces instances sera alors considérée par TypeScript comme étant du type défini par la classe. Pour instancier une classe, il faut utiliser le mot-clé `new` et assigner l'instance dans une variable.

Syntaxe :

```
let/const variable: ClassName = new ClassName();
```

Exemple :

```
const employee = new Employee();
```

Plusieurs instances d'une même classe peuvent être créées si besoin. Chaque instance est alors complètement indépendante des autres.

Exemple :

```
const employee1 = new Employee();
const employee2 = new Employee();

// Log: false
console.log(employee1 === employee2);
```

Avec TypeScript, les classes sont dites "*first class citizen*" (Objet de premier ordre), il est donc possible de les assigner dans des variables.

Syntaxe :

```
let/const ClassName = class {
 // ...
};
```

Lorsqu'une classe est contenue dans une variable, il faut alors utiliser le mot-clé new sur celle-ci afin de créer une instance de la classe.

Exemple :

```
const Employee = class {
  // ...
};

const employee = new Employee();
```

Remarque

Cette syntaxe est plus particulière et parfois utilisée dans certaines implémentations plus complexes. Dans la suite de ce chapitre, l'ensemble des exemples de code ne s'appuieront pas sur cette syntaxe.

3. Les propriétés

Étant donné qu'un objet doit fonctionner de manière autonome, il est nécessaire qu'il conserve un état au cours de son utilisation. Cet état est contenu dans l'objet sous la forme de donnée. L'état d'un programme en programmation orientée objet correspondra donc à l'ensemble des états de chaque instance utilisée pour le faire fonctionner.

Pour assigner une donnée dans un objet en TypeScript, il faut utiliser une propriété. Celles-ci sont définies au niveau de la classe et doivent être typées. Pour déclarer une propriété, il suffit de l'ajouter entre les accolades de la classe en lui donnant un nom et en précisant ensuite son type.

Syntaxe :

```
class ClassName {
  propertyName: type;
}
```

Exemple :

```
class Employee {
  firstName: string;
  lastName: string;
}
```

Une fois l'instance d'une classe créée, il est alors possible d'assigner des valeurs aux différentes propriétés, mais aussi de les récupérer. Les propriétés sont disponibles en utilisant un "**.**" après le nom de la variable contenant l'objet.

Exemple :

```
let employee = new Employee();
employee.firstName = "Evelyn";
employee.lastName = "Miller";

// Log: Evelyn
console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);

employee = new Employee();
```

```
// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Remarque

Par défaut, les propriétés seront initialisées avec la valeur `undefined`.

Ce premier exemple pose un problème de compilation, lorsque l'on démarre un projet en TypeScript avec la configuration de base du compilateur (obtenue en lançant la commande `tsc --init`). Par défaut, le compilateur initialise le projet avec l'option `--strict` activée dans le fichier `tsconfig.json`. Elle active des sous-options dont l'option `--strictPropertyInitialization`. Cette dernière déclenche une erreur lorsque les valeurs des propriétés ne sont pas initialisées à l'instanciation d'une classe. Dans le précédent exemple, les propriétés ne sont pas initialisées lors de l'utilisation du mot-clé `new`, elles contiennent donc la valeur `undefined`. Cette situation peut être due à une erreur d'inattention, c'est pourquoi TypeScript ne l'autorise pas par défaut.

Il est possible de forcer le compilateur à ne pas remonter cette erreur en utilisant le caractère "!" à la fin du nom d'une propriété (aussi appelé *definite assignment assertion operator*).

Exemple :

```
class Employee {
  firstName!: string;
  lastName!: string;
}

const employee = new Employee();

// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Remarque

Attention à ne pas forcer le compilateur de manière systématique. La plupart des erreurs remontées par le mode strict de TypeScript sont importantes. Forcer l'initialisation des propriétés permet, par exemple, de ne pas obtenir d'erreur si elles sont utilisées sans valeur. C'est une erreur courante dans le développement web, surtout lorsque la propriété est censée contenir une fonction (ce qui déclenche l'erreur : undefined is not a function).

Assigner une valeur aux propriétés permet aussi de contourner l'erreur de compilation liée au mode strict.

Exemple :

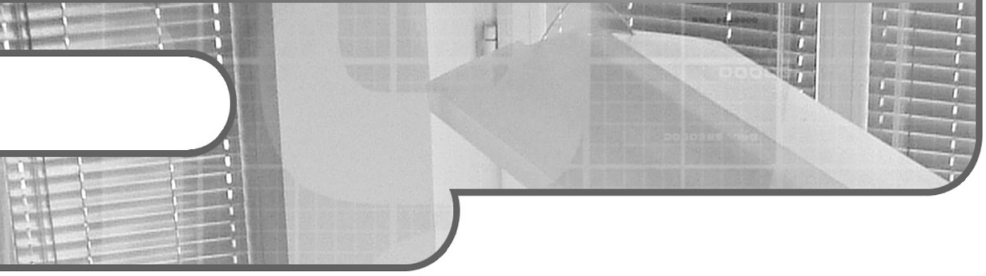
```
class Employee {
  firstName: string = "Evelyn";
  lastName: string = "Miller";
}

const employee = new Employee();

// Log: Evelyn
console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);
```

Remarque

Dans la suite de ce chapitre, l'option `--strict` sera toujours considérée comme active dans les exemples de code.

Une fois définies, les propriétés peuvent être retrouvées dans l'environnement de développement via l'autocomplétion.

Exemple (Visual Studio Code) :



Chapitre 3
Découvrir le JSX

1. Introduction au JSX

JSX (*JavaScript XML*) est une extension de syntaxe utilisée dans React pour décrire l'interface utilisateur sous forme de code JavaScript. Elle permet de mélanger du code JavaScript avec des balises HTML, ce qui rend la création d'interfaces utilisateur en React plus lisible. Il est important d'être à l'aise avec cette syntaxe le plus tôt possible.

Lorsque vous utilisez JSX, vous pouvez écrire des éléments React comme si vous écriviez du HTML, mais en réalité, le JSX est transpilé en code JavaScript pur avant d'être interprété par le navigateur.

Voici à quoi ressemble un exemple simple de JSX :

```
import React from 'react';

const MonComposant = () => {
  return <h1>Bienvenue dans mon application React !</h1>;
};
```

Dans cet exemple, nous avons utilisé JSX pour créer un élément `h1` contenant le message de bienvenue. Cela permet d'écrire du code d'interface utilisateur de manière déclarative et sans appeler à chaque fois `React.createElement`.

Avantages du JSX

Clarté et lisibilité du code

En utilisant le JSX, on se rapproche du langage naturel utilisé pour structurer une page web. Le JSX facilite la communication entre les développeurs et améliore la maintenabilité du code, car il est plus facile de comprendre la structure de l'interface utilisateur.

Composition des composants facilitée

Avec le JSX, vous pouvez facilement imbriquer des composants les uns dans les autres. Nous pouvons les stocker dans des variables, les passer à des fonctions ou même ajouter du code JavaScript à l'intérieur. Cela permet de créer des composants complexes en les formant de composants plus petits et réutilisables. La composition des composants est l'un des principes fondamentaux de React, et le JSX facilite cette approche.

Intégration de JavaScript dynamique

Le JSX permet d'incorporer des expressions JavaScript directement dans le code HTML-like. Cela permet d'interagir dynamiquement avec les données et de générer du contenu d'interface utilisateur en fonction de l'état de l'application. Vous pouvez utiliser des expressions JavaScript entre accolades `{ }` pour évaluer des variables, effectuer des opérations, et bien plus encore.

Validation statique

L'un des avantages du JSX est qu'il est vérifié statiquement lors de la compilation. Cela signifie que les erreurs de syntaxe ou de structure dans le JSX sont détectées à l'avance, avant que l'application ne s'exécute. Cela permet d'éviter de nombreux bogues qui pourraient survenir à l'exécution, ce qui rend le processus de développement plus sûr et plus fluide.

Intégration aisée de bibliothèques tierces

Étant donné que le JSX ressemble au HTML, il est plus facile d'intégrer des bibliothèques de composants tierces. Par exemple, les développeurs React peuvent utiliser des bibliothèques de composants UI, des outils de gestion de formulaires, et d'autres bibliothèques populaires en les utilisant simplement comme ils le feraient dans un projet HTML standard.

Transpilation vers JavaScript standard

Le JSX doit être transpilé en JavaScript standard avant d'être exécuté par les navigateurs. Cela permet aux développeurs d'utiliser les fonctionnalités les plus récentes de React tout en conservant une compatibilité avec les anciens navigateurs. Des outils tels que Babel ont longtemps été utilisés pour transpiler le JSX en JavaScript standard, bien que des alternatives aient vu le jour. Il n'est pas nécessaire de comprendre la manière dont les outils de transpilation fonctionnent, mais il faut retenir que le JSX n'est pas un standard du Web et qu'il doit passer par une étape avant d'être compris par les navigateurs.

Voici comment le JSX est transpilé en JavaScript :

```
const message = <h1>Bienvenue dans mon application React !</h1>;
```

Code JavaScript transpilé :

```
const message = React.createElement("h1", null, "Bienvenue dans
mon application React !");
```

Le JSX est transformé en appels de fonctions `React.createElement`, qui créent les éléments React en utilisant les propriétés passées en tant qu'arguments.

Avec un outil comme Vite qu'on a vu précédemment, il n'est pas nécessaire de configurer quoi que ce soit. Le JSX sera automatiquement transpilé. De la même manière, il est possible d'utiliser des fonctionnalités modernes de JavaScript sans se soucier de la compatibilité navigateur. Les fonctions fléchées par exemple peuvent être converties en fonctions standard si votre navigateur cible ne les prend pas en charge. Référez-vous à la documentation de Vite si vous devez assurer un support de vieux navigateurs comme Internet Explorer, par exemple.

2. Syntaxe et éléments JSX

Dans le JSX, vous pouvez utiliser une syntaxe similaire au HTML pour décrire l'interface utilisateur de votre application React. Cependant, il y a quelques différences-clés à noter.

2.1 Éléments JSX

Dans le JSX, vous pouvez utiliser des balises pour créer des éléments React. Les balises doivent correspondre à des composants React définis ou à des balises HTML natives. Les éléments JSX peuvent être imbriqués les uns dans les autres, tout comme dans le HTML.

Exemple de balises JSX

```
const monElement = <div>
  <h1>Titre</h1>
  <p>Contenu du paragraphe</p>
</div>;
```

Les éléments JSX sont les blocs de construction de base pour créer l'interface utilisateur. Ils ressemblent à des balises HTML, mais en réalité, ce sont des objets JavaScript qui représentent des composants React ou des éléments du DOM.

La syntaxe est similaire au HTML, ce qui les rend plus familiers aux développeurs web.

Exemples d'éléments JSX

```
const element1 = <div>Contenu du div</div>;
const element2 = <h1>Titre</h1>;
const element3 = <MonComposant />;
```

Dans cet exemple, la variable `element1` contient un élément JSX représentant une balise `div`, `element2` contient une balise `h1`, et `element3`, un composant React appelé `MonComposant`.

Vous pouvez également utiliser des attributs dans les balises JSX, tout comme dans le HTML.

2.2 Attributs JSX

Dans le JSX, vous pouvez utiliser des attributs pour configurer les éléments d'interface utilisateur de manière similaire à HTML. Les attributs permettent de personnaliser le comportement et l'apparence des éléments React. Voici comment vous pouvez utiliser des attributs JSX dans vos composants :

Exemple d'attributs JSX

```
const monElement = <input type="text" placeholder="Entrez votre nom" />;
const nom = "John";
const monElement2 = <h1>Bienvenue, {nom} !</h1>;
```

Dans cet exemple, `monElement` est un élément JSX représentant une balise `input` avec les attributs `type` et `placeholder`. La valeur de l'attribut `type` est une chaîne de caractères `"text"` tandis que la valeur de l'attribut `placeholder` est une autre chaîne de caractères `"Entrez votre nom"`.

Enfin, `monElement2` est un élément JSX représentant une balise `h1` qui utilise une expression JavaScript (`{nom}`) pour afficher le nom dynamique de l'utilisateur en fonction de la variable `nom`.

2.2.1 Attributs booléens

Les attributs booléens en JSX sont légèrement différents de ceux en HTML. Dans JSX, vous pouvez spécifier des attributs booléens sans valeur, ce qui les rend true par défaut.

Exemple d'attribut booléen JSX

```
const monElement = <input type="checkbox" checked />;
```

Dans cet exemple, l'attribut `checked` est spécifié sans valeur, ce qui signifie qu'il est évalué comme `checked={true}`. Cela permet de cocher la case par défaut.

2.2.2 Attributs personnalisés

Les attributs personnalisés (*data attributes*) sont une fonctionnalité de HTML permettant de stocker des informations supplémentaires directement dans une balise, utilisant une syntaxe qui commence généralement par `data-`. Ces attributs sont accessibles via JavaScript.

Par exemple, considérez un élément de liste HTML qui représente un produit dans une boutique en ligne :

```
<li data-product-id="123" data-product-category="électronique">
Tablette graphique</li>
```

Dans votre script JavaScript, vous pourriez ensuite accéder à ces informations comme suit :

```
let produit = document.querySelector('li');
console.log(produit.dataset.productId); // Affiche : 123
console.log(produit.dataset.productCategory); // Affiche : électronique
```

Dans le contexte de React, l'utilisation des data attributes est moins courante, principalement parce que React favorise la gestion des données via l'état (*state*) et les propriétés (*props*) des composants, que nous découvrirons bientôt. Ces derniers offrent un mécanisme plus structuré et intégré pour manipuler les données au sein de vos composants.

Cependant, les data attributes peuvent toujours trouver leur place dans une application React, spécialement lorsqu'il s'agit d'intégrer des scripts tiers ou de manipuler des éléments du DOM directement. Ces scénarios sont généralement évités, mais peuvent être nécessaires dans certains cas.

Par exemple, vous pourriez vouloir tracer des informations supplémentaires pour des outils d'analyse ou de suivi, où un data attribute pourrait être utile. Si nous reprenons notre exemple de produit dans une liste :

```
function Produit({ id, categorie, nom }) {
  return (
    <li data-product-id={id} data-product-category={categorie}>
      {nom}
    </li>
  );
}

// Utilisation du composant Produit avec des attributs data
<Produit id="123" categorie="électronique" nom="Tablette graphique" />
```

Dans ce fragment de code, nous définissons un composant `Produit` qui reçoit quelques props et utilise des data attributes pour ajouter des informations supplémentaires à l'élément `li`. Bien que ce ne soit pas une pratique courante dans les applications React, elle peut servir dans des situations spécifiques, où l'interaction directe avec le DOM est nécessaire ou préférable. Le concept de props sera introduit en détail dans le chapitre Les fondamentaux de React.

2.3 Éléments React et fragments

Un élément React est un objet JavaScript qui représente un composant ou un élément JSX. Il possède trois propriétés : le type (nom de la balise ou composant), les attributs et les enfants (les éléments ou le texte à l'intérieur).

JSX encourage le retour d'un seul élément par composant. Cela signifie que si vous voulez retourner plusieurs éléments, ils doivent être enveloppés dans un élément parent unique.

Les fragments permettent d'englober plusieurs éléments sans ajouter de nœud DOM supplémentaire. Ils sont utiles lorsque vous avez besoin de retourner plusieurs éléments sans les encapsuler dans une balise parente.

Exemple d'utilisation de fragments

```
import React from 'react';

const MonComposant = () => {
  return (
    <>
      <h1>Titre</h1>
      <p>Contenu du paragraphe</p>
    </>
  );
};
```

Nous reviendrons en détail sur les fragments plus loin dans ce chapitre, à la section Les fragments JSX.

2.4 Commentaires JSX

Les commentaires ont un rôle essentiel dans le développement : ils permettent aux développeurs d'ajouter des notes, des explications et des rappels dans le code. Dans le JSX, vous pouvez également inclure des commentaires pour documenter votre code et rendre le développement plus facile à comprendre et à maintenir.

Pour ajouter des commentaires dans le JSX, vous pouvez les entourer de `{/* */}`.

Exemple de commentaire JSX

```
const monElement = (
  <div>
    {/* Ceci est un commentaire */}
    <h1>Titre</h1>
  </div>
);
```

Syntaxe des commentaires JSX

JSX, étant une extension syntaxique pour JavaScript, a une manière distincte d'intégrer des commentaires dans le code.

En fait, pour insérer des commentaires dans du JSX, on utilise une syntaxe qui s'apparente à celle utilisée pour les blocs de commentaires en JavaScript. Il s'agit de l'encapsulation des commentaires entre `{/*` et `*/}`. Cette méthode permet d'intégrer des commentaires directement au sein de votre code JSX, garantissant ainsi que vos notes et clarifications restent visibles sans affecter le fonctionnement de votre code.

Exemple de commentaire en ligne dans le JSX

```
function App() {
  return (
    <div>
      {/* Ceci est un commentaire intégré dans du JSX, il ne sera
pas visible dans le DOM */}
      Bienvenue dans mon application React !
    </div>
  );
}
```