2 – Rappels sur l’Orienté Objet

1MODE - Modélisation d'applications

SUPINFO

Sommaire

1. Contextualisation

1. Contextualisation

Paradigme Orienté Objet

- Lors de modules de cours précédents, nous avons étudié un nouveau **paradigme** de programmation : l'**orienté objet**

- Dans ce cours, il est désormais question de voir **quels** objets créer par opposition à **comment** les créer

1. Contextualisation

Une approche naturelle pour modéliser le monde réel

- L'orienté objet permet de représenter les entités du monde réel sous forme d'objets dans le code
- On peut synthétiser les **propriétés** et **comportements** d’objets réels avec du code
- Cette pratique réduit le décalage entre le monde réel et le logiciel

1. Contextualisation

Rappel des concepts clés de l’orienté objet

- **Classe** : définit la structure et le comportement d'un type d'objet
- **Objet** : instance d'une classe avec ses propres valeurs d'attributs

- **Abstraction** : représenter les fonctionnalités d’un objet sans en exposer les détails
- **Encapsulation** : coupler un comportement avec un état et masquer le fonctionnement interne
- **Héritage** : permet de créer de nouvelles classes basées sur des classes existantes
- **Polymorphisme** : capacité d’un objet à prendre plusieurs formes

1. Contextualisation

Avantage de l’OOP pour las modélisation

- Facilite la compréhension et la maintenance du code grâce à la **modularité**
- Favorise la **réutilisabilité** du code via l'héritage et le polymorphisme
- Permet de modéliser fidèlement les concepts du **domaine** métier
- Supporte l'**évolution** incrémentale des modèles

1. Contextualisation

2. Installer et configurer TypeScript

2. Installer et configurer TypeScript

TypeScript

- TypeScript est un sur-ensemble de JavaScript développé par Microsoft
- Il ajoute un typage statique optionnel à JavaScript
- Le code TypeScript est *transpilé* en JavaScript pur
- Fonctionne sur n'importe quel navigateur, hôte ou système d'exploitation
- Open source

2. Installer et configurer TypeScript

Installer TypeScript

- Installer Node.js (https://nodejs.org)
- Installer TypeScript globalement via npm (*Node Package Manager*) :

```
$ npm install -g typescript
```

- Avoir un éditeur de code (Visual Studio Code, IDEs JetBrains, Sublime Text, etc.)

2. Installer et configurer TypeScript

Créer un fichier TypeScript

- Créez un nouveau fichier avec l'extension ".ts", par exemple "monscript.ts"
- Écrivez votre code TypeScript dans ce fichier, par exemple :

```
function direBonjour(nom: string) {
  console.log(`Bonjour, ${nom} !`)
}

direBonjour("Alice")
```

2. Installer et configurer TypeScript

***Transpiler* le code TypeScript en JavaScript**

- Ouvrez un terminal et naviguez jusqu'au dossier contenant votre fichier ".ts"
- Exécutez la commande suivante pour compiler votre code TypeScript en JavaScript :

```
$ tsc monscript.ts
```

- Un fichier "monscript.js" sera généré dans le même dossier

2. Installer et configurer TypeScript

Exécuter le code JavaScript généré

- Dans le terminal, exécutez le code JavaScript généré avec Node.js :

```
$ node monscript.js
```

- Le résultat de votre script s'affichera dans le terminal, par exemple :

```
Bonjour, Alice !
```

2. Installer et configurer TypeScript

Automatiser le processus de transpilation

- Initialisez un nouveau fichier package.json dans votre projet avec la commande :

```
$ npm init -y
```

- Installez TypeScript en tant que dépendance de développement dans votre projet :

```
$ npm install --save-dev typescript
```

2. Installer et configurer TypeScript

Automatiser le processus de transpilation

- Modifiez le fichier package.json pour ajouter une commande "run" qui compile et exécute le code TypeScript :

```
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "build": "tsc monscript.ts",
    "start": "npm run build && node monscript.js"
  },
  "devDependencies": {
    "typescript": "^5.4.5"
  }
}
```

2. Installer et configurer TypeScript

Automatiser le processus de transpilation

- Exécuter le code TypeScript en une seule commande :

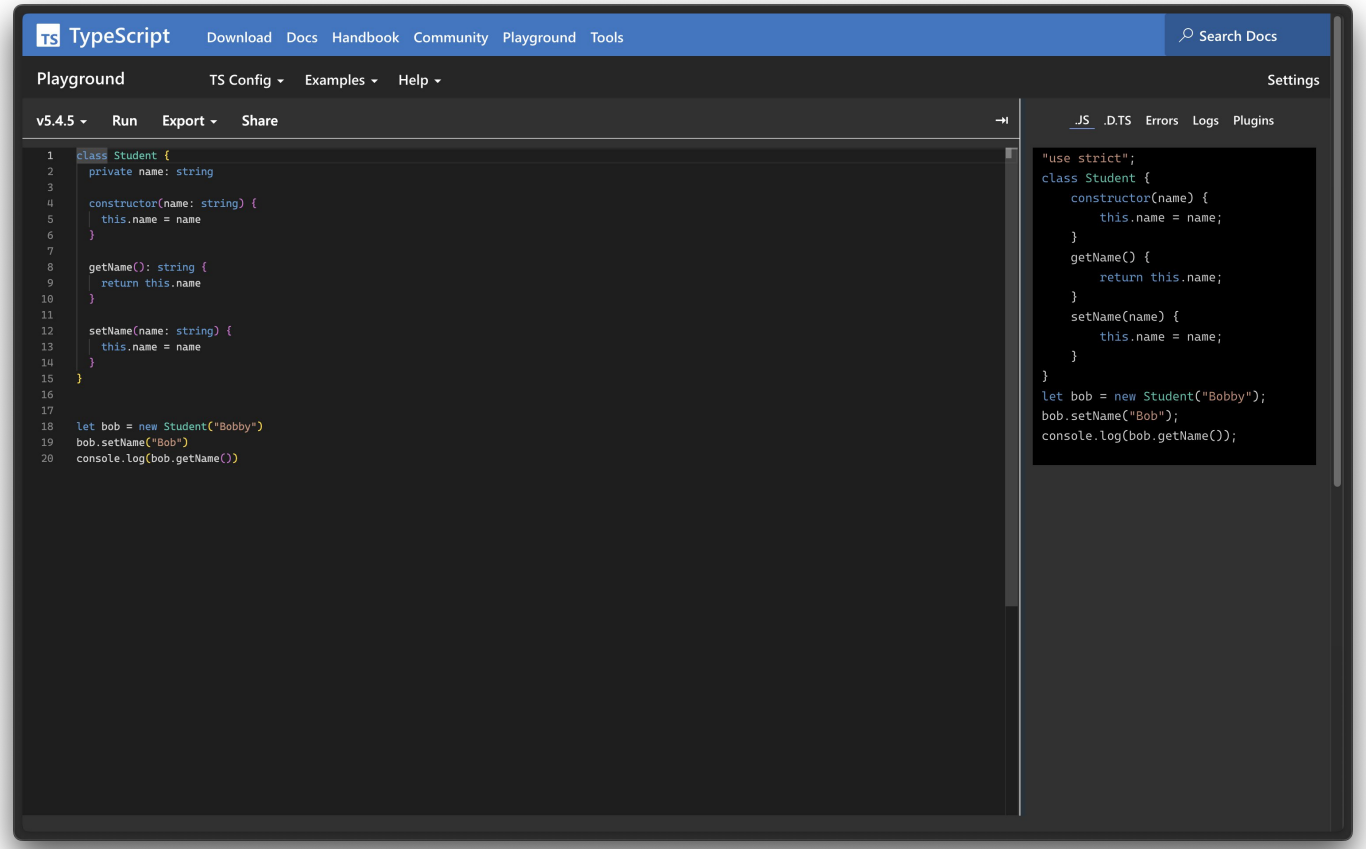
```
$ npm start
```

- Cette commande compilera votre code TypeScript et exécutera le JavaScript résultant, affichant le résultat dans le terminal.

2. Installer et configurer TypeScript

TypeScript Playground

https://www.typescriptlang.org/play



2. Installer et configurer TypeScript

3. Premiers pas avec TypeScript

3. Premiers pas avec TypeScript

Les types de bases

- `boolean` : true ou false
- `number` : entiers et nombres à virgule flottante
- `string` : chaînes de caractères
- `array` : liste d'éléments de même type, ex: number[]
- `tuple` : groupe avec un nombre fixe d'éléments de types pouvant être différents
- `any` : type permettant n'importe quelle valeur (à éviter)
- `void` : absence de valeur de retour pour les fonctions
- `null` et `undefined`

3. Premiers pas avec TypeScript

Déclaration de variables

- Utiliser let ou const pour déclarer des variables
- Typer les variables avec : type après le nom
- Exemples :

```
let age: number = 25
const nom: string = "Jean"
let notes: number[] = [12, 14, 9]
```

3. Premiers pas avec TypeScript

Fonctions

- Typer les paramètres et la valeur de retour
- ? pour les paramètres optionnels
- Valeurs par défaut des paramètres possibles

```
function saluer(nom: string, titre?: string): string {
  return `Bonjour ${titre || ''} ${nom}`
}
```

3. Premiers pas avec TypeScript

Classes

- Attributs et méthodes typés
- Modificateurs d'accès : **public**, **private**, **protected**
- Héritage avec **extends** et **implements**

3. Premiers pas avec TypeScript

Classes

```
class Student {
  private name: string

  constructor(name: string) {
    this.name = name
  }

  getName(): string {
    return this.name
  }

  setName(name: string) {
    this.name = name
  }
}
```

```
let bob = new Student("Bobby")

bob.setName("Bob")

console.log(bob.getName())
```

3. Premiers pas avec TypeScript

Interfaces

- Définir la structure d'un objet
- Propriétés obligatoires ou optionnelles avec ?
- Permet de typer des paramètres ou variables

```
interface Personne {
  nom: string
  age: number
  adresse?: string
}

let john: Personne = {
  nom: "John Doe",
  age: 30
}
```

3. Premiers pas avec TypeScript

Interfaces

```
interface Person {
  getName(): string
}

class Student extends Person {
  private name: string

  constructor(name: string) {
    super()
    this.name = name
  }

  getName(): string {
    return this.name
  }
}
```

3. Premiers pas avec TypeScript

Syntaxe alternative

```
interface Person {
  name: string
  age: number
}

class User implements Person {
  constructor(public name: string, public age: number) {}
}
```

3. Premiers pas avec TypeScript

Type

```
type Person = {
  name: string
  age: number
}

class User implements Person {
  constructor(public name: string, public age: number) {}
}
```

3. Premiers pas avec TypeScript

type VS interface

- Les interfaces sont toujours extensibles, les types ne le sont pas
- Les interfaces créent un nouveau nom utilisable partout, les types non
- Les interfaces ne peuvent décrire que la structure d'un objet
- Les types sont plus flexibles et peuvent représenter plus de structures

3. Premiers pas avec TypeScript

Classes abstraites

```
abstract class Person {
  abstract getName(): string
}

class Student extends Person {
  private name: string

  constructor(name: string) {
    super()
    this.name = name
  }

  getName(): string {
    return this.name
  }
}
```

3. Premiers pas avec TypeScript

Attributs de classe

```
class Point {
  static instances = 0
  private x: number
  private y: number

  constructor(x: number, y: number) {
    this.x = x
    this.y = y
    Point.instances++
  }
}
```

3. Premiers pas avec TypeScript

Héritage

```
class Point {...}

class Point3D extends Point {...}

interface Colored {...}

class Pixel extends Point implements Colored {...}
```

3. Premiers pas avec TypeScript

Polymorphisme

```
class Animal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log("L'animal émet un son")
  }
}
```

3. Premiers pas avec TypeScript

Polymorphisme

```
class Dog extends Animal {
  makeSound(): void {
    console.log("Le chien aboie")
  }
}

class Cat extends Animal {
  makeSound(): void {
    console.log("Le chat miaule")
  }
}
```

3. Premiers pas avec TypeScript

Polymorphisme

```
const animals: Animal[] = [
  new Dog("Rex"),
  new Cat("Félix")
]

animals.forEach(animal => {
  console.log(animal.name)
  animal.makeSound()
})
```

3. Premiers pas avec TypeScript

Chaînage de valeurs optionnelles

```
type Data = {
  message?: string
}

type SearchResult = {
  name: string
  data?: Data
}

if (result?.data?.message) {
  console.log(`${result.name} -> ${result.data.message}`)
}
```

3. Premiers pas avec TypeScript

Type Unions

```
type Color = "red" | "green" | "blue"
const r: Color = "red"
const r: Color = "yellow"  // Erreur: "yellow" n'est pas du type Color

function useColor(c: Color) {
  switch (c) {
    case "red":
      break
    case "green":
      break
    case "blue":
      break
  }
}
```

3. Premiers pas avec TypeScript

Type Prédicats

```
interface Square {
  kind: "square"
  size: number
}

interface Circle {
  kind: "circle"
  radius: number
}

type Shape = Square | Circle
```

3. Premiers pas avec TypeScript

Type Prédicats

```
function isSquare(shape: Shape): shape is Square {
  return shape.kind === "square"
}

function isCircle(shape: Shape): shape is Circle {
  return "radius" in shape
}
```

3. Premiers pas avec TypeScript

Type Prédicats

```
function calculateArea(shape: Shape): number {
  if (isSquare(shape)) {
    return shape.size ** 2
  }
  if (isCircle(shape)) {
    return Math.PI * shape.radius ** 2
  }
  throw "unknown shape"
}
```

3. Premiers pas avec TypeScript