

INTRODUCTION AUX CLASSES



DÉFINITION ET UTILITÉ

Les classes sont un concept fondamental de la programmation orientée objet (POO). Elles permettent de créer des objets avec des propriétés et des méthodes partagées, facilitant la modularité et la réutilisabilité du code.

```
class Voiture {
  marque: string;
  modele: string;

constructor(marque: string, modele: string) {
    this.marque = marque;
    this.modele = modele;
  }
}
```





HÉRITAGE DU CONCEPT DE LA PROGRAMMATION ORIENTÉE OBJET

TypeScript, comme d'autres langages **POO**, utilise les **classes** pour gérer la complexité, promouvoir la réutilisabilité et créer une structure de code organisée.









MODULARITÉ ET RÉUTILISABILITÉ DU CODE

Les **classes** permettent de découper le code en blocs logiques, qui peuvent être réutilisés à travers le projet. Cela facilite la maintenance et la compréhension du code.











SYNTAXE DE BASE

TypeScript utilise des mots-clés spécifiques pour déclarer une classe et son constructeur.

```
class MaClasse {
  attribut: string;

constructor(attribut: string) {
    this.attribut = attribut;
  }

maMethode(): void {
    console.log("Ma méthode exécutée");
  }
}

const objet = new MaClasse("Exemple d'attribut");
objet.maMethode(); // Affiche "Ma méthode exécutée"
```











MOTS-CLÉS "CLASS", "CONSTRUCTOR"

La syntaxe de base pour déclarer une **classe** et son **constructeur** est la suivante :

```
class MyClass {
  constructor() {
    //...
  }
}
```









DÉCLARATION ET INSTANCIATION D'UNE CLASSE

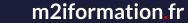
Pour instancier une classe, utilisez le mot-clé new :

```
const obj = new MyClass();
```











PROPRIÉTÉS DES CLASSES



PROPRIÉTÉS DES CLASSES

```
class Personne {
  nom: string;
  age: number;

  constructor(nom: string, age: number) {
    this.nom = nom;
    this.age = age;
  }
}

const personnel = new Personne("John", 25);
```

- La classe Personne possède deux propriétés: nom et age.
- Lors de la création d'une instance de la classe, les valeurs de ces propriétés sont passées en paramètres au constructeur.





DÉCLARATION DES PROPRIÉTÉS

Pour déclarer une propriété, il suffit de définir son **nom** et son **type**.

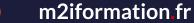
```
class Car {
   marque: string;
   modele: string;
   prix: number;
}
```













TYPAGE DES PROPRIÉTÉS

Le **typage des propriétés** est essentiel pour garantir la **sécurité** et la **lisibilité** du code.

```
class Car {
   marque: string;
   modele: string;
   prix: number;
}
```











ACCESSEURS ET MUTATEURS

Les **accesseurs** et **mutateurs** (getters et setters) permettent de contrôler l'accès et la modification des propriétés d'une classe.

```
class ExampleClass {
 private _property: number;
  get property(): number {
    return this._property;
 set property(value: number) {
   if (value >= 0) {
      this._property = value;
   } else {
      console.log("Valeur incorrecte");
```









GETTER ET SETTER

Les getters et setters sont des méthodes spéciales pour accéder et modifier les propriétés.

```
class Car {
    private _marque: string;

    get marque(): string {
        return this._marque;
    }

    set marque(value: string) {
        this._marque = value;
    }
}
```









m2iformation.fr



SYNTAXE ET UTILITÉ

Le mot-clé get est utilisé pour déclarer un getter et set pour déclarer un setter. Ils sont utiles pour valider les données et **contrôler** l'accès aux propriétés.











VISIBILITÉ DES PROPRIÉTÉS

La **visibilité** des propriétés permet de déterminer l'accès et la modification des propriétés d'une classe. Les niveaux de visibilité sont public, private et protected.

Niveau de visibilité	Accès	Modification
public	Toutes classes	Oui
private	Classe uniquement	Oui
protected	Classe et sous-classes	Oui











PUBLIC

Les propriétés **publiques** peuvent être accédées et modifiées depuis **n'importe où**.

```
class Car {
    public marque: string;
}
```











PRIVATE

Les propriétés **privées** ne peuvent être accédées et modifiées qu'à **l'intérieur** de la classe.

```
class Car {
   private marque: string;
```











PROTECTED

Les propriétés protégées peuvent être accédées et modifiées à l'intérieur de la classe et de ses sousclasses.

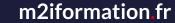
```
class Car {
   protected marque: string;
```













MÉTHODES DES CLASSES



DÉCLARATION DE MÉTHODES

Les **méthodes** sont des fonctions définies à l'intérieur d'une **classe**, permettant d'effectuer des actions sur les objets de cette classe.

```
class Personne {
    sePresenter() {
        console.log(`Bonjour, je m'appelle Christopher Loisel.`);
    }
}
let unePersonne = new Personne();
unePersonne.sePresenter(); // Affiche "Bonjour, je m'appelle Christopher Loisel."
```









m2iformation.fr



SYNTAXE

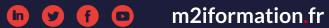
```
class MyClass {
   methodName(param1: Type1, param2: Type2): ReturnType {
```













UTILITÉ

Les méthodes permettent d'encapsuler les actions et les manipulations des propriétés de la classe, améliorant ainsi la **modularité** et la **lisibilité** du code.











TYPAGE DES PARAMÈTRES ET DU RETOUR

Les **paramètres** et le **type de retour** doivent être déclarés pour garantir la **sécurité du typage** lors de l'utilisation des méthodes.

```
function saluer(nom: string): string {
   return 'Bonjour, ' + nom;
}

const message: string = saluer('John');
console.log(message);
```











MÉTHODES STATIQUES









m2iformation.fr



SYNTAXE

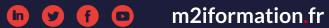
```
class MyClass {
   static staticMethodName(param1: Type1): ReturnType {
```













UTILITÉ

Les **méthodes statiques** appartiennent à la **classe** elle-même et peuvent être appelées sans instancier un objet de la classe.

```
class Exemple {
   static maMethode() {
      console.log("Cette méthode est statique");
   }
}

// Appel de la méthode statique sans instancier de nouvel objet
Exemple.maMethode();
```











ACCÈS AUX MÉTHODES

MyClass.staticMethodName(arg1); // Appel d'une méthode statique









m2iformation.fr



SURCHARGE DE MÉTHODES

La **surcharge de méthodes** permet de définir des méthodes avec le même nom, mais avec des **paramètres différents**.

```
class Example {
  method(param: string): void;
  method(param: number): void;
  method(param: any): void {
    if (typeof param === "string") {
        console.log(`Paramètre de type string: ${param}`);
    } else if (typeof param === "number") {
        console.log(`Paramètre de type number: ${param}`);
    }
}

const exampleInstance = new Example();
    exampleInstance.method("exemple texte");
    exampleInstance.method(123);
```









SYNTAXE

La **surcharge** de méthodes en TypeScript se fait en déclarant des **signatures** de méthode, puis en implémentant une méthode avec des paramètres plus **génériques**.











HÉRITAGE ET POLYMORPHISME



HÉRITAGE DE CLASSES

L'héritage permet à une classe d'hériter des propriétés et des méthodes d'une autre classe.

L'héritage en TypeScript permet de créer une hiérarchie de classes, où une classe dérivée peut hériter des propriétés et des méthodes d'une classe de base, tout en ayant la possibilité d'ajouter ou de surcharger des méthodes et des propriétés.











EXEMPLE:

```
class Base {
  constructor(public name: string) {}
  afficherNom() {
    console.log(`Nom: ${this.name}`);
class Derived extends Base {
  constructor(name: string, public age: number) {
   super(name);
  afficherInfo() {
    this.afficherNom();
```









SYNTAXE ET UTILITÉ

Pour hériter d'une classe, utilisez le mot-clé **extends**.

```
class Animal {
   // ...
}
class Chien extends Animal {
   // ...
}
```









HÉRITAGE DES PROPRIÉTÉS ET DES MÉTHODES

La classe dérivée (**Chien** dans l'exemple) hérite des propriétés et des méthodes de la classe de base (**Animal**).

```
class Chien extends Animal {
  race: string;
  constructor(nom: string, race: string) {
    super(nom);
    this.race = race;
  }
  decrire(): string {
    return `${super.decrire()} C'est un ${this.race}.`;
  }
}

const chien1 = new Chien("Max", "labrador");
  console.log(chien1.decrire());
```









POLYMORPHISME

Le **polymorphisme** permet d'utiliser une **interface commune** pour plusieurs types de classes dérivées, rendant le code plus **flexible**.

```
interface Animal {
    son(): void;
}

class Chien implements Animal {
    son(): void {
        console.log("Woof!");
    }
}

class Chat implements Animal {
    son(): void {
        console.log("Miaou!");
    }
}
```









SURCHARGE DE MÉTHODES DANS LES CLASSES DÉRIVÉES

La surcharge de méthodes permet à une classe dérivée de redéfinir une méthode héritée avec la même signature.

```
class Animal {
 parler() {
class Chien extends Animal {
 parler() {
```











MOT-CLÉ "SUPER"

Utilisez le mot-clé **super** pour appeler la méthode de la classe de base.

```
class Chien extends Animal {
  parler() {
    super.parler() // Appelle la méthode parler() de la classe Animal
    // Code spécifique au chien
  }
}
```













INTERFACES

Les **interfaces** définissent un **contrat** que les classes implémentant cette interface doivent respecter.

```
interface Person {
    firstName: string;
    lastName: string;
}

class Employee implements Person {
    firstName: string;
    lastName: string;
    position: string;
}

const employee1 = new Employee();
employee1.firstName = "John";
employee1.lastName = "Doe";
employee1.lastName = "Doe";
employee1.position = "Developer";
```











MOT-CLÉ "IMPLEMENTS"

Utilisez le mot-clé implements pour indiquer qu'une classe implémente une interface.

```
interface AnimalInterface {
   parler(): void
}

class Chien implements AnimalInterface {
   parler() {
        // Implémentation de la méthode parler()
   }
}
```









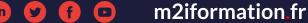
SYNTAXE ET UTILITÉ

Les **interfaces** permettent de garantir la conformité avec une **structure spécifiée**, offrant une plus grande **modularité** et une **réutilisabilité** du code.

```
interface MonInterface {
  propriete1: type1;
  propriete2: type2;
  methode1(): typeRetour;
}

class MaClasse implements MonInterface {
  propriete1: type1;
  propriete2: type2;

  methode1(): typeRetour {
    // Implementation de la méthode
  }
}
```





GÉNÉRICITÉ



UTILISATION DES GÉNÉRIQUES

Les **génériques** permettent de créer des classes et des méthodes qui peuvent fonctionner avec **différents types**.

```
function identity<T>(arg: T): T {
   return arg;
}

let output1 = identity<string>("Hello");
let output2 = identity<number>(42);
```









m2iformation.fr



UTILISATION DES GÉNÉRIQUES

Syntaxe	Explication
<t></t>	Définit un type générique
arg: T	Un argument du type générique
identity <t>(arg:T)</t>	Un appel de fonction avec un type générique

- Les génériques permettent de **réutiliser** du code
- Ils offrent une **flexibilité** pour travailler avec **divers types**
- La type spécifique est déterminé au moment de l'appel de la fonction/classe











SYNTAXE

On utilise les chevrons <> pour déclarer des **types génériques**.

```
class ClasseGenerique<T> {
 propriete: T;
```













UTILITÉ

Les **génériques** assurent la **réutilisabilité** et la **modularité** du code en évitant la duplication pour gérer différents types.

Avantages	Exemples
Meilleure réutilisabilité	Fonctions génériques
Réduction de la duplication	Classes génériques
Gestion d'une variété de types	Collections génériques











CLASSES ET MÉTHODES GÉNÉRIQUES



DÉCLARATION ET UTILISATION DE GÉNÉRIQUES

On utilise les génériques dans les classes et les méthodes pour déclarer des types variables.

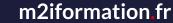
```
class ClasseGenerique<T> {
  propriete: T;

methodeGenerique(param: T): T {
   return param;
  }
}
```











UTILISATION DANS LES CLASSES ET LES MÉTHODES

On instancie des **classes génériques** en précisant le type entre **chevrons** <>.

```
const exemple = new ClasseGenerique<string>();
exemple.propriete = "test";
exemple.methodeGenerique("texte");
```











CONTRAINTES SUR LES GÉNÉRIQUES



SYNTAXE AVEC "EXTENDS"

On peut restreindre les types autorisés pour les génériques en utilisant le mot-clé **extends**.

```
class ClasseGenerique<T extends number> {
  propriete: T;
}
```









m2iformation.fr



RESTREINDRE LES TYPES AUTORISÉS

Cette syntaxe permet de n'autoriser que certains types, évitant ainsi les erreurs à l'utilisation.

```
const exempleCorrect = new ClasseGenerique<number>();
const exempleIncorrect = new ClasseGenerique<string>(); // Erreur de compilation
```









m2iformation.fr









