TypeScript

Les outils

Les outils

Compiler :

- La compilation est le fait de transformer du code dans un langage A vers un langage B plus bas niveau
  - Exemple : Transformer du code en C++ vers de l'assembleur

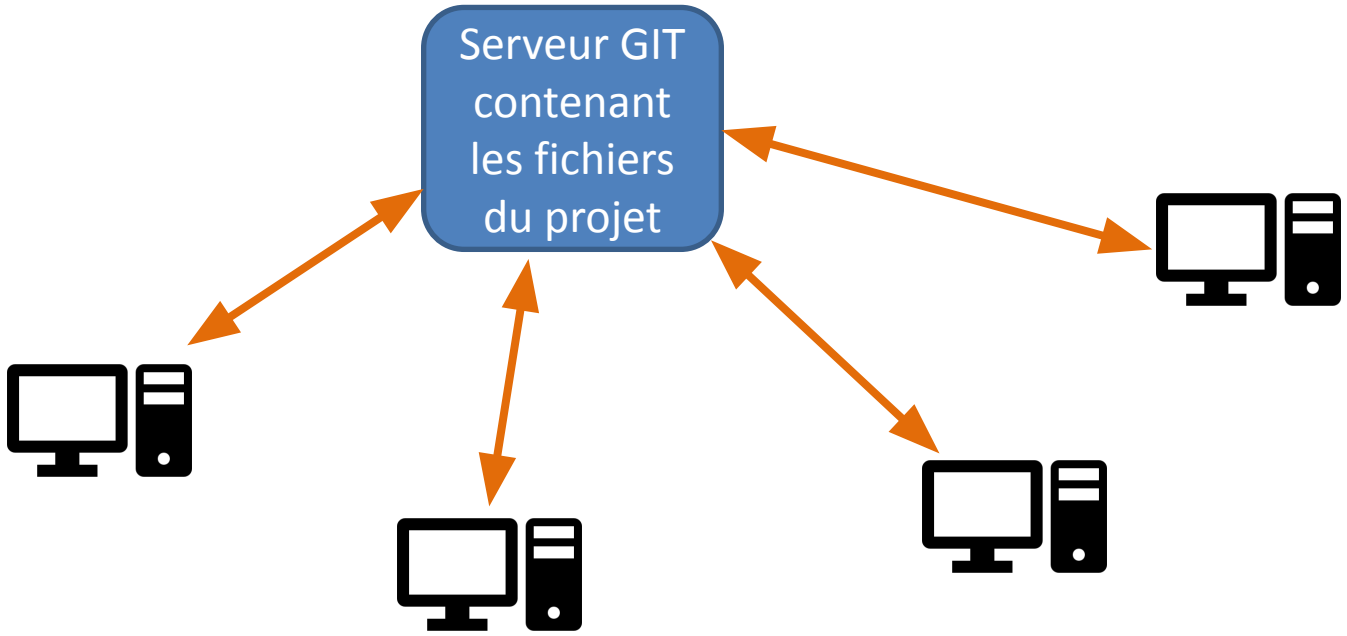
Transpiler :

- Transpiler, tout comme la compilation, transforme du code dans un langage vers un autre de même niveau.
  - Exemple transformer la version de Javascript ES6 vers la version ES5

Les outils : Git

Git :

- Gestionnaire de versions
- Permet de garder un historique des modifications
- Permet de mutualiser le développement d'un projet entre plusieurs développeurs



Chaque développeur peut modifier les fichiers du projet et ensuite le partager avec son équipe en l'envoyant sur un serveur Git.

Chaque développeur doit avoir un "client Git" pour pouvoir se connecter au serveur.

Les outils : Node.js

Node.js :

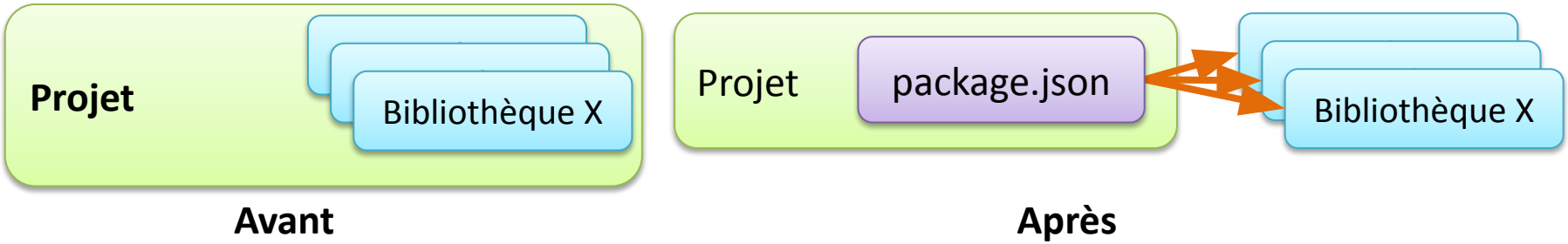
- Est une plateforme applicative pour exécuter des applications en JavaScript
- Très utilisé dans le web pour faire des serveurs ou des outils pour aider les développeurs.
- https://nodejs.org/en/

Les outils : npm

npm :

- Gestionnaire de paquet de Node.js
- Il permet de télécharger facilement des paquets (bibliothèques ou programmes)

La commande `npm init` dans le répertoire du projet va créer un fichier contenant les dépendances du projet.



Avant **Après**

Installer TypeScript

Installation :

Installer Git :

- https://git-scm.com/
- Cliquer sur le lien à droite

Installer Node.js :

- https://nodejs.org/en/
- Installer la version stable (LTS)

Créer un répertoire

Ouvrir une console dans ce répertoire

Taper :

```
npm install -g typescript
```

```
npm init
```

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```

Installer TypeScript

Nous avons installé le package Typescript **npm install -g typescript**

-g indique que l'installation est global à tous les projets, nous n'aurons plus à le refaire

tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata

Permet de mettre un fichier de configuration pour TypeScript.

Nous pouvons maintenant créer un fichier TypeScript : test.ts

Dans la console : **tsc --watch**

Indique au transpileur de transformer les fichiers **.ts** en **.js** à chaque modification de ceux-ci.

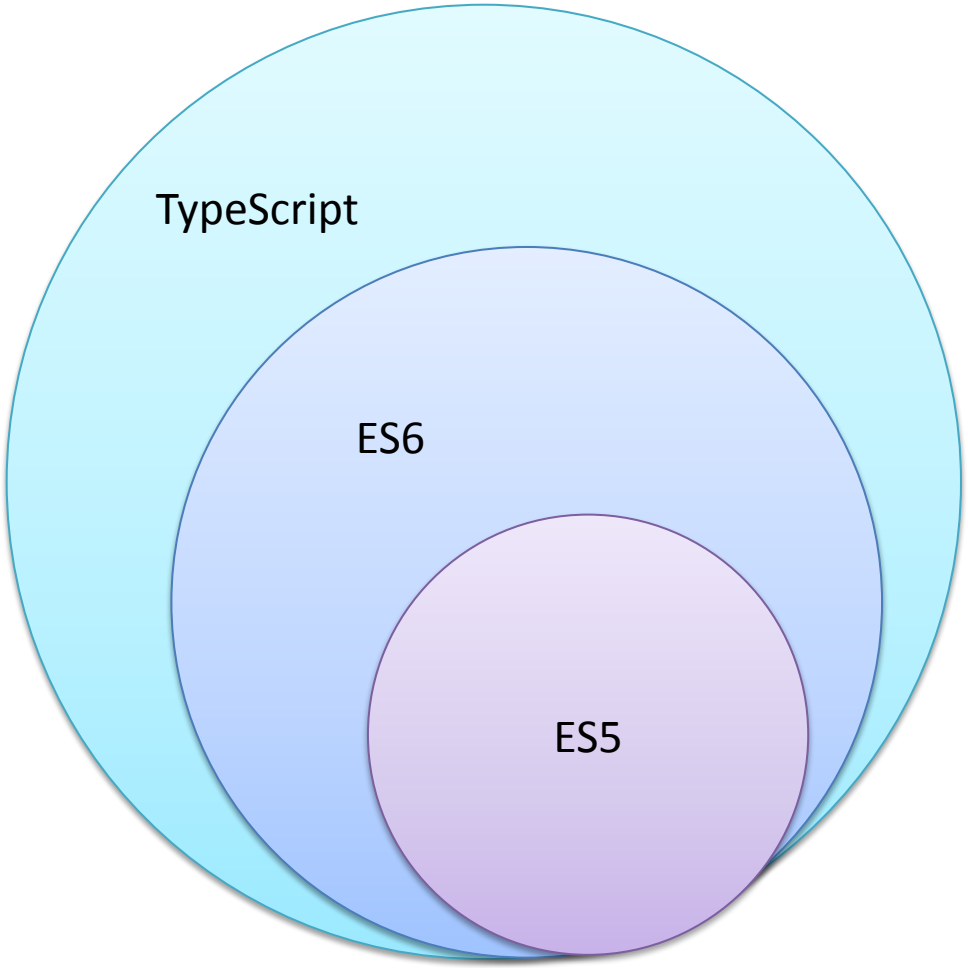
TypeScript

Présentation

Présentation

TypeScript :

- Langage "surcouche" de ES6
- Existe depuis 2012
- Ajoute du typage à ES6
- Porté par Microsoft
- Utilisé pour Angular



Présentation

TypeScript n'est pas compris par les navigateur, c'est pour cela qu'il est transpilé en JavaScript (généralement ES5).

TypeScript n'apporte que peu de nouvelles fonctionnalités par rapport à ES6.

Typage

Le typage

Typage :

- Le fait de différencier le type d'une variable

Dans un langage dit typé, on ne peut enregistrer qu'un type de valeur dans une variable.

Typage en TS :

```
let variable: type;
```

```
const unNombre: number = 12;
```

La typage

Typage :

- Le fait de différencier le type d'une variable

Dans un langage dit typé, on ne peut enregistrer qu'un type de valeur dans une variable.

Typage en TS :

```
let variable: type;
```

```
const unNombre: number = 12;
```

Les types : boolean et number

boolean :

- Valeur true, false

```
let isDone: boolean = false;
```

number :

- Comme en JS, tous les nombres sont des flottants.
  - 1.7976931348623157e+308 à 5e-324
- Accepte aussi les valeurs : Binaires, Octales, Hexadecimale

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Les types : string

string :

- Chaîne de caractères

```
let prenom: string = 'John';
let nom: string = "Doe";
```

Nous pouvons taper une chaîne sur plusieurs lignes avec le templating (guillemet ` avec altgr + 7 sur windows):

```
let texte: string = `mon prénom est ${ prenom }
et mon nom est ${ nom }`;
```

Équivalent à :

```
let texte2: string = "mon prénom est " + prenom + "\n"
+ "et mon nom est " + nom;
```

Les types : Array

Array :

- Tableau typé

Le tableau suivant ne peut contenir que des nombres :

```
let tab0: number[] = [ -5, 2, 8 ];

tab0.push(12.56); // OK

tab0.push("Salut !"); // Erreur
// Argument of type 'string' is not assignable to parameter of type 'number'.
```

Il existe une seconde écriture utilisant des tableaux génériques :

```
let tab1: Array<number> = [ -8, 15.9, 0 ];
```

Les types : Tuple

Tuple :

- Tableau avec différents types

```
// déclaration
let user: [string, number];
// initialisation correcte
user = ["John", 35]; // ok
// initialisation incorrecte
user = [35, "John"]; // Erreur
```

Utilisation :

```
console.log( user[0] );
// affiche : John

console.log( user[1] );
// affiche : 35
```

Les types : Enum

Enum :

- Ensemble de nombres constants ayant un nom.

```
enum Fruit {Banane, Kiwi, Orange};
let k: Fruit = Fruit.Kiwi;

console.log(k);
// Affiche : 1
```

Par défaut, la première valeur vaut 0.

Il est possible de définir des valeurs manuellement :

```
enum Couleur {Rouge, Vert=-0.2, Jaune=58};
let n: Couleur = Couleur.Jaune;

console.log(n);
// Affiche : 58
```

Il est possible de récupérer la chaîne de caractère ainsi :

```
let s: string = Couleur[-0.2];

console.log(s);
// Affiche : Vert
```

Les types : any et void

any :

- Accepte n'importe quel type de variable.

```
let v: any = "Salut";
v = 1;
v = true;

let liste: any[] = ["John", false, 123];
liste[0] = 35;
```

void :

- Accepte que les valeurs **null** et **undefined**

```
let n: void = null;
```

- Généralement utilisé pour indiquer qu'une fonction ne renvoie pas de valeurs.

```
function direCoucou(): void {
    console.log("Coucou !");
}
```

Les types : null et undefined

null : Indique qu'une valeur est définie comme n'ayant aucune valeur

```
let n: null = null;
```

undefined : Valeur par défaut lorsqu'une variable n'est pas initialisée

```
let n: undefined = undefined;
// équivaut à
let n: undefined;
```

Les types : null et undefined

null, undefined et **void** sont peu utilisés directement car très limités.

null et **undefined** sont des types enfants de tous les autres types et peuvent donc être utilisé comme valeur pour n'importe quel type.
Sauf lorsque l'option **: --strictNullChecks** est utilisé

```
let s: string;
console.log(s); // affiche : undefined

let n: number = null;
```

Les types : never

never :

- Indique qu'une fonction ne renvoie jamais de valeur et n'atteigne pas la fin de leur bloc.
  - Fonctions retournant que des exceptions
  - Fonctions ayant une boucle infinie

```
function erreur(message: string): never {
    throw new Error(message);
}

// retourne un type never
function fail() {
    return error("Affiche une erreur");
}

// boucle infinie
function nonStop(): never {
    while (true) {
    }
}
```

Cast

Caster :

- Changer une valeur dans un **type A** vers une valeur de **type B**

```
let uneValeur: any = "Peut-être une chaîne ?";

let longueur: number = (<string>uneValeur).length;
```

Le type any pouvant être de n'importe quel type, nous indiquons que **uneValeur** est de type **String**.

Dans ce cas, TypeScript nous fait confiance et autorise l'utilisation des méthodes propre aux objets String.

Seconde syntaxe :

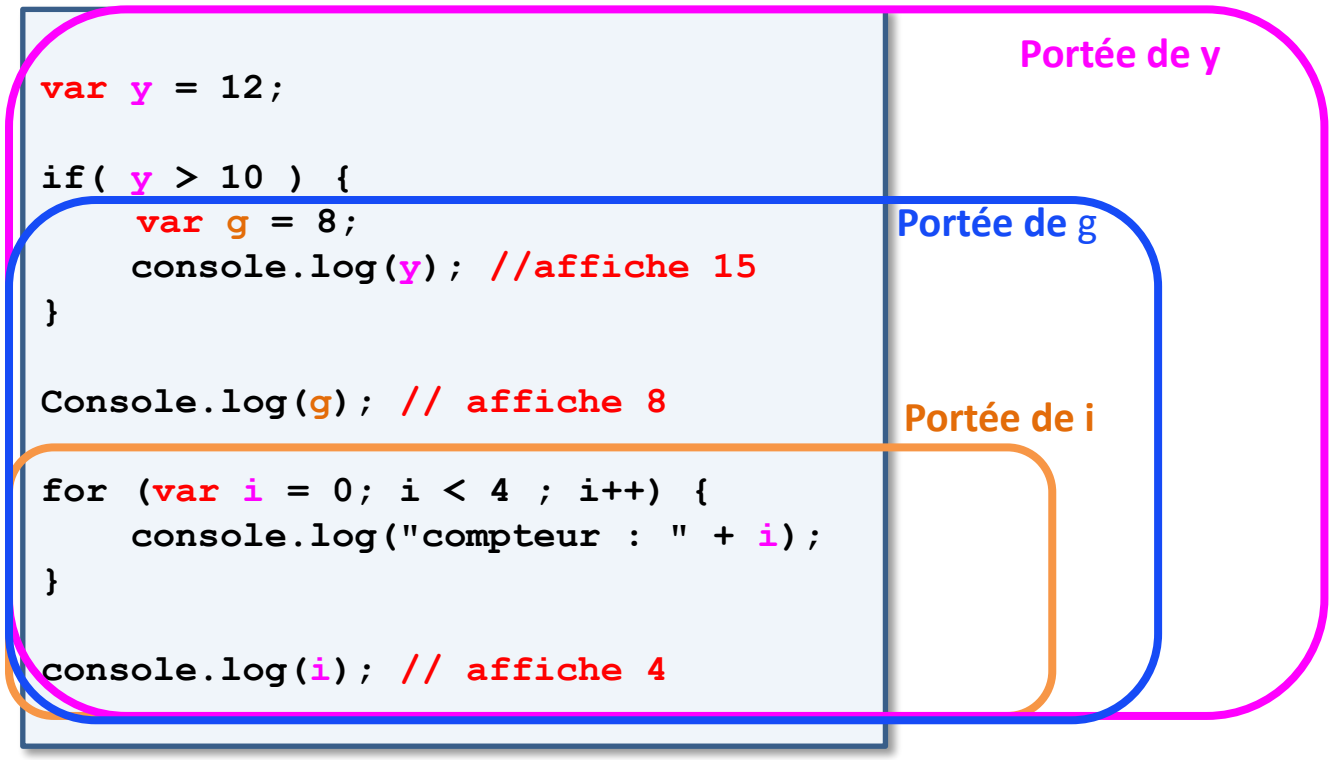
```
let uneValeur: any = "Peut-être une chaîne ?";

let longueur: number = (uneValeur as string).length;
```

Portée var et let

Déclarations de variables : var

Une variable définie avec le mot clé "**var**" est :

- accessible partout **si elle est définie dans l'espace global**
- accessible dans toute la **fonction** où elle est définie.



Déclarations de variables : var

Portée de n

```
var n = "bonjour";

function portee() {
    console.log(n); //affiche bonjour
    var t = "au revoir";
}

portee();

console.log(t);
// ReferenceError: t is not defined
```

Portée de t

Déclarations de variables : let

Une variable définie avec le mot clé "**let**" est :

- accessible partout **si elle est définie dans l'espace global**
- accessible que dans le **bloc** où elle est définie

```
let y = 15;                                          Portée de y

if( y > 10 ) {
    let g = 8;                                       Portée de g
    console.log(y); //affiche 15
}


console.log(g); // ReferenceError: g is not defined

for (let i = 0; i < 4 ; i++) {                       Portée de i
    console.log("i compteur de boucle" + i);
}

console.log(i); // ReferenceError: i is not defined
```

Déclarations de variables : let

Une variable définie par **let** ne peut pas être redéfinie dans le même espace de nom.

Une variable définie par **var** peut être redéfinie dans le même espace de nom.

Les constantes : const

const :

- Créé une référence accessible qu'en lecture.
- L'identifiant ne peut plus être réassigné.
- Sur un objet, la constante est sur la référence de celui-ci et non ses attributs.

```
const PATH = 2; // La valeur de la constante ne peut plus être changée

PATH = 18; //Uncaught TypeError: Assignment to constant variable
```

```
const OBJ = { a: 5 };

OBJ = 18; //Uncaught TypeError: Assignment to constant variable

// il est possible de changer la valeur d'un attribut de l'objet
OBJ.a = 8
```

Affecter par décomposition

Échanger des valeurs facilement :

```
let a = 1;
let b = 3;

[a, b] = [b, a];

console.log(a, b);
```

Affiche :

```
3 1
```

Décomposer depuis un objet :

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
}
let { a, b, k } = o;
console.log(a, b, k)
```

Affiche :

```
foo 12 undefined
```

Les noms des variables doivent correspondre aux noms des attributs contenus par l'objet, dans le cas contraire, les valeurs seront "undefined"

Affecter par décomposition

Récupérer les valeurs d'un tableau par décomposition :

```
let tab = [1,2,3];
let [a, b, c] = tab;

console.log(a, b, c);
```

Affiche :

```
1 2 3
```

Décomposer depuis un objet :

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
}
let { a, b, k } = o;
console.log(a, b, k)
```

Affiche :

```
foo 12 undefined
```

Les noms des variables doivent correspondre aux noms des attributs contenus par l'objet, dans le cas contraire, les valeurs seront "undefined"

Fonctions

Les fonctions

Fonction :

- Séquence d'instructions réalisant un calcul ou une tâche.
- Permet de découper un problème global en plusieurs éléments plus simples et réutilisables.

En JavaScript :

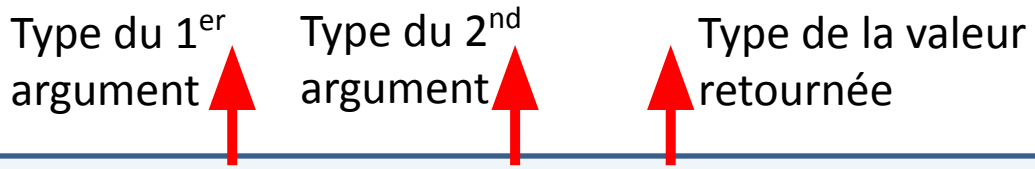
```
function addition(x, y) {
    return x + y;
}

let addition2 = function(x, y) {
    return x + y;
};
```

Utilisation :

```
let resultat = addition(5, 2);
let resultat2 = addition2(55, 20);
```

Les fonctions : typage

Type des paramètres et des valeurs de retour.




```
function addition(x: number, y: number): number {
    return x + y;
}

let addition2 = function(x: number, y: number): number {
    return x + y;
};
```

Utilisation :

```
let resultat0 = addition("b", 2);  // erreur de type
let resultat1 = addition(2);       // erreur, il manque un paramètre
let resultat2 = addition(2, 3, 8); // erreur, paramètre en trop

let resultat2 = addition(55, 20);  // ok
let resultat2 = addition2(10, 40); // ok
```

Les fonctions : paramètres optionnel

Paramètre optionnel :

- Paramètre pouvant être omis lors de l'appel de la fonction

Paramètre optionnel

```
function addition(x: number, y?: number): number {
    if(y) {        // nous testons si y est défini
        return x + y;
    }
     return x;
}
```

Utilisation :

```
let resultat1 = addition(2);          // ok
let resultat2 = addition(55, 20);     // ok
```

Dans l'exemple ci-dessus, si nous voulions que **x** soit optionnel, **y** doit l'être aussi.

Les fonctions : paramètres par défaut

Paramètre par défaut :

- Si le paramètre est omis, il recevra une valeur par défaut.
- La type d'un paramètre ayant une valeur par défaut peut être omis
- La valeur par défaut définit le type du paramètre

```
function addition(x: number, y = 12): number {
    return x + y;
}

function addition2(x = 8, y: number): number {
    return x + y;
}
```

Utilisation :

```
let resultat1 = addition(2);        // ok
let resultat2 = addition(55, 20);   // ok

console.log( resultat1, resultat2);

addition2(5, 2);          // ok
addition2(3);             // erreur, paramètre manquant
addition2(undefined, 5); // ok
```

Affiche :

```
14 75
```

Les fonctions : paramètres restant

Paramètres restant:

- Permet de récupérer les paramètres en trop et les placer dans un tableau

```
function addition(x: number, y: number): number {
    return x + y;
}

function addition2(x: number, ...leReste: number[]): number {
    for (let nbr of leReste) {
        x += nbr;
    }
    return x;
}

let resultat = addition2(55, 5,8);
console.log(resultat);
```

Affiche :

```
68
```

Utilisation :

```
let resultat1 = addition(2, 8, 9, 2);   // erreur, paramètres en trop
let resultat2 = addition2(2, 8, 9, 2);  // ok
```

Les fonctions : Surcharge

Surcharge :

- Fonctions portant **le même nom** avec un **nombre ou des types de paramètres différents**.
- En TypeScript ne permet qu'une surcharge de la définition pour un corps de fonction unique.

```
function foo(truc: string): number;

function foo(bidule: number, machin: string): string;

function foo(param1: string|number, param2?: string): any{
    if (typeof param1 === 'string') {
        // Première surcharge
        return param1.length;
    } else {
        // seconde surcharge
        return param2 + param1;
    }
}
```

Utilisation :

```
foo("coucou");              // ok
foo(12);                    // erreur, paramètre manquant
foo(12, "salut");           // ok
```

POO

Présentation des objets

- **Un objet est une entité logicielle modélisant une « chose » quelconque**
  - Exemples d'objets:
    - Une voiture
    - Un téléphone
    - Un compte en banque

- **Un objet est composé d'attributs (synonyme : champs, propriétés) :**
  - Une voiture a une couleur et un numéro d'immatriculation
  - Un téléphone a un numéro et une marque
  - Un compte en banque a un solde et un numéro qui l'identifie

Un attribut est donc une caractéristique propre à l'objet.

- **Un objet est composé de méthodes :**
  - Une voiture roule, accélère et freine
  - Un téléphone reçoit des appels, envoie des message
  - Un compte on peut lui ajouter des écritures, on peut le déplacer dans une autre banque

Une méthode est une action appliquée à l'objet.

Présentation des classes

Classe :

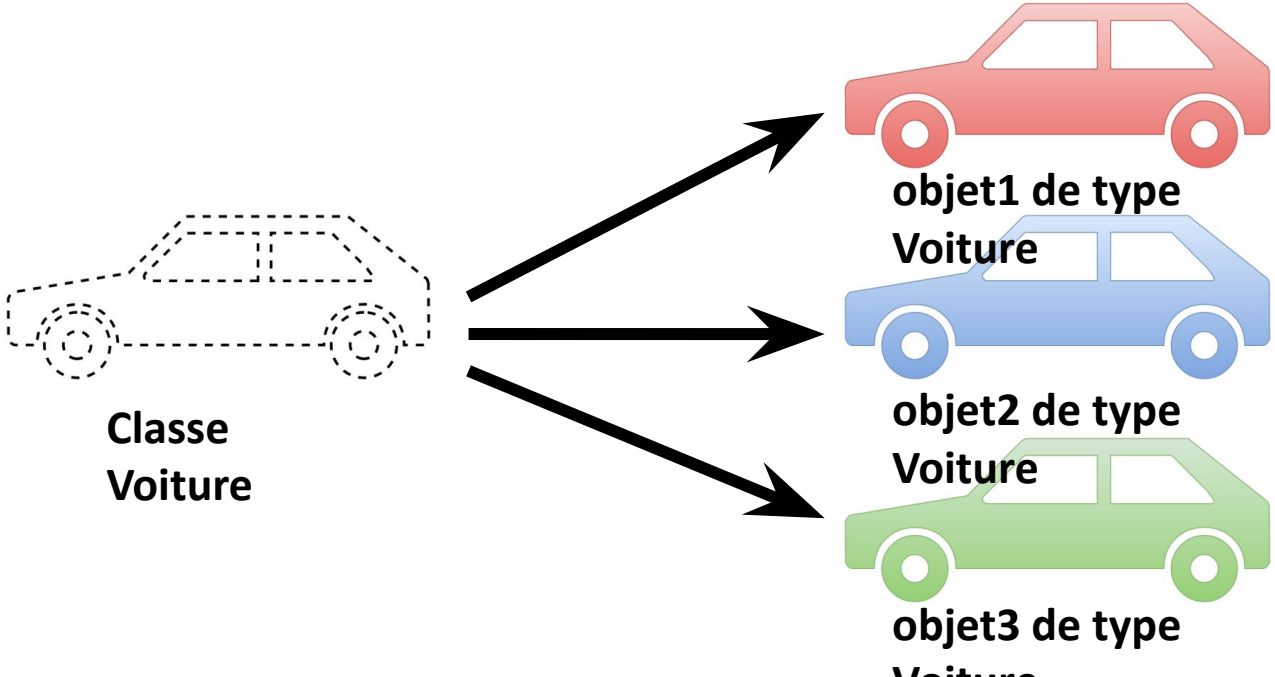
- Une classe est un model de données.
- On peut l'associer à un schéma que doit suivre une catégorie d'objet.

Exemple :

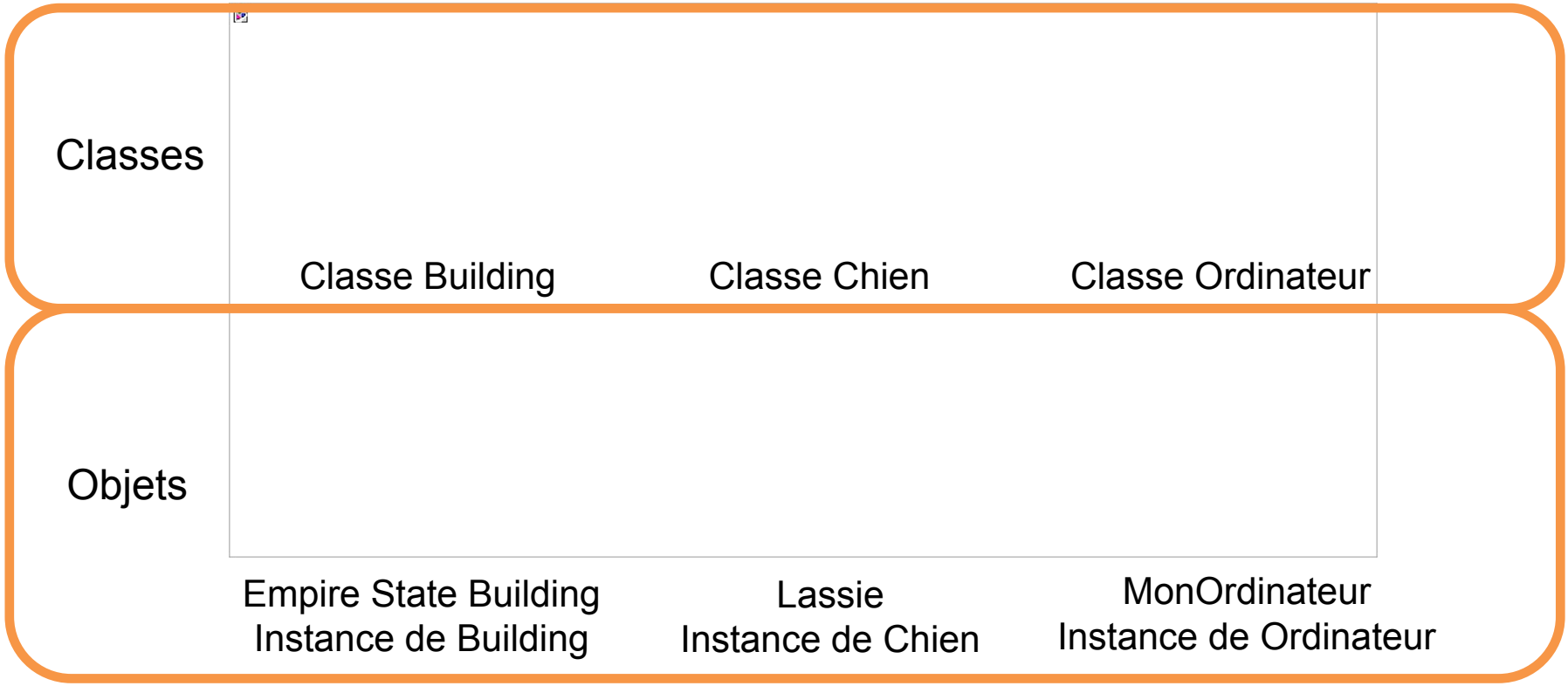
Une voiture a :

- Des attributs : un nombre de roues, une couleur, un volant, une immatriculation
- Des action : avancer, reculer, tourner, s'arrêter.

Les objets voitures devront suivre ce schéma pour être considérées comme une voiture.



Présentation des instances



- **Une instance est un objet**

Dire que Lassie est **instances de la classe Chien** revient à dire qu'elle est de type Chien.

Il est possible que deux objets aient les mêmes attributs, tout comme deux téléphones de la même marque soient identiques mais cela reste deux objets séparés.

Présentation : Pourquoi la POO ?

La programmation orientée objet (**POO**) permet de :

- Factoriser le code.
- Regrouper et manipuler un ensemble de variables et de méthodes associées à une entité (l'objet).
- Faciliter l'organisation et la réutilisation du code et donc sa correction.
- Rendre les projets plus évolutifs.

L'intégration de la POO est facile car elle représente un model cohérent de données.

Les classes

Classe :

- Schéma à suivre pour la création d'une classe

```
class Bonjour {
    message: string;

    constructor(message: string) {
        this.message = message;
    }
    affiche() {
        console.log( "Bonjour " + this.message );
    }
}

let bjr = new Bonjour("John");
```

Les classes : méthodes

Classe :

- Schéma à suivre pour la création d'une classe

```
class Bonjour {
    message: string;

    constructor(message: string) {
        this.message = message;
    }
    affiche() {
        console.log( "Bonjour " + this.message );
    }
}

let bjr = new Bonjour("John");

bjr.affiche()
```

Les classes : héritage

Héritage :

- Permet de créer une classe "parent" qui donnera toutes ses caractéristiques à ses enfants

Classe mère : `Animale`

```
class Animale {
    nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }
    deplacer(distance: number = 0) {
        console.log(`${this.nom} bouge de ${distance}m.`);
    }
}
```

Classe enfant : `Chien`

```
class Chien extends Animale {
    constructor(name: string) {
        super(name);
    }
}
```

Chien de par son parent :

- A un nom
- Peut se déplacer

Utilisation :

```
let monChien = new
Chien("Milou");
monChien.deplacer(5);
```

Les classes : héritage

C'est comme reprendre l' "ADN" de la classe Animale (les fonctions et les attributs) et l'injecter dans la classe Chien.

class Animale

class Chien extends Animal

class Chien

classe Animale

L'héritage multiple n'existe pas en TypeScript comme beaucoup de langages récents.
Ceci est dû à l'héritage en diamant et des conflits qu'il pouvait engendrer.

Les classes : héritage

Si nous considérons que l'héritage est reprendre l'ADN du parent, nous pouvons considérer que l'enfant est une évolution.

Nous pouvons donc lui ajouter de nouveaux attributs et fonctions que le parent n'a pas.

Classe enfant : Chien

```
class Chien extends Animale {
    couleurPoil: string;

    constructor(name: string) {
        super(name);
    }

    manger() {
        console.log("Je mange");
    }
}
```

Utilisation :

```
let monChien = new Chien("milou");
monChien.deplacer(5);
monChien.manger();
```

Les classes : super

super :

- Indique que nous utilisons une fonction crée dans le parent

Classe enfant : Chien

```
class Chien extends Animale {
    constructor(name: string) {
        super(name);
    }
}
```

Dans ce cas, super(), indique que nous appelons la fonction **constructor** du parent.

La visibilité

La visibilité

La visibilité

- parfois appelé contrôle d'accès
- permet de limiter l'accès **aux attributs** et **aux méthodes** depuis l'extérieur d'une classe ou d'un objet.

Il existe 3 modificateurs de visibilité :

public

- Accessible par tout le monde.

protected

- Limite l'accès aux objets de la classe courante **et** ceux des classes enfants.

private

- Accessible que dans les objets de la classe où l'élément est défini.

La visibilité : public (attributs)

public : permet la modification d'un attribut depuis n'importe où.

```
class Animale {
    public age: number;
    nom: string;      // sans indication : visibilité public

    constructor(leNom: string) {
        this.nom = leNom;
    }
    information() {
        console.log( this.nom, this.age);
    }
}
```

Utilisation :

```
let foo: Animale = new Animale("raspou");
foo.information(); // affiche : raspou undefined

foo.age = 12;
foo.information(); // affiche : raspou 12

foo.nom = "fedya";
foo.information(); // affiche : fedya 12
```

La visibilité : public (méthodes)

public s'applique aussi aux méthodes

```
class Coucou {
    constructor() {
    }

    public direBonjour() {
        console.log("Bonjour");
    }

    direCoucou() {              // sans indication : visibilité public
        console.log("Coucou");
    }
}
```

Utilisation :

```
let foo: Coucou = new Coucou();

foo.direCoucou();           // Affiche: Coucou
foo.direBonjour();          // Affiche: Bonjour
```

La visibilité : protected (attributs)

protected : empêche l'utilisation d'un attribut ou une méthode en dehors d'une instance de la classe courante ou des classes héritières.

```
class Bar {
    protected nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }

    public information() {
        console.log(this.nom);
    }
}
```

Utilisation :

```
let foo: Bar = new Bar("raspou");
foo.information();
console.log(foo.nom); // Erreur, nom est 'protected'
foo.nom = "test";     // Erreur, nom est 'protected'
```

Nous avons tenté d'atteindre l'attribut "**nom**" en dehors de de l'objet foo ce qui **n'est pas possible car** celui-ci est **protected**.

La visibilité : protected (attributs)

protected : Les objets des classes enfants ont accès aux attributs.

Classe mère : Bar

```
class Bar {
    protected nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }
}
```

Classe enfant : ChildBar

```
class ChildBar extends Bar {
    constructor(leNom: string) {
        super(leNom)
    }

    afficheInfo() {
        console.log(this.nom)
    }
}
```

ChildBar, enfant de Bar, a une méthode qui accède à l'attribut de son parent '**nom**'

Utilisation :

```
let cb: ChildBar = new ChildBar("raspou");

cb.afficheInfo();
```

La visibilité : protected (méthodes)

protected : s'applique aussi aux méthodes.

```
class Bar {
    protected nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }

    protected info() {
        console.log(this.nom);
    }

    public affiche() {
        this.info();
    }

}
```

La méthode 'affiche' accède à la méthode 'info' qui est protected.

Utilisation :

```
let foo: Bar = new Bar("raspou");
foo.info();              // Erreur, la méthode est 'protected'

foo.affiche();           // affiche: raspou
```

La visibilité : protected (méthodes)

protected : Les objets de la classe enfant ont accès aux méthodes protected de leur parent.

Classe mère : Bar

```
class Bar {
    protected nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }

    protected info() {
        console.log(this.nom);
    }
}
```

Classe enfant : ChildBar

```
class ChildBar extends Bar {
    constructor(leNom: string) {
        super(leNom)
    }

    public afficheInfo() {
        this.info();
    }
}
```

Utilisation :

```
let cb: ChildBar = new ChildBar("raspou");
foo.info();                    // Erreur, la méthode est 'protected'

foo.afficheInfo();             // affiche: raspou
```

La visibilité : private (attributs)

private : Seuls les objets de la classe courante ont le droit d'utiliser l'attribut.

```
class Bar {
    private nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }

    public setNom(unNom: string) {
        this.nom = unNom;
    }

    public info() {
        console.log(this.nom);
    }
}
```

Utilisation :

```
let foo: Bar = new Bar("raspou");

foo.info(); // affiche: raspou

foo.nom();  // Erreur, interdit

foo.setNom("fedya");

foo.info(); // affiche: fedya
```

La visibilité : private (attributs)

private : Seuls les objets de la classe courante ont le droit d'utiliser l'attribut.

Classe mère : Bar

```
class Bar {
    private nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }
}
```

Classe enfant : ChildBar

```
class ChildBar extends Bar {
    constructor(leNom: string) {
        super(leNom)
    }

    public changeNom(unNom: string) {
        this.nom = unNom; // Erreur, nom est private !
    }
}
```

La visibilité : private (méthodes)

private : Peut s'appliquer aux méthodes

```
class Bar {
    private nom: string;
    constructor(leNom: string) {
        this.nom = leNom;
    }

    private setNom(unNom: string) {
        this.nom = unNom;
    }

    public info() {
        console.log(this.nom);
    }
}
```

Utilisation :

```
let foo: Bar = new Bar("raspou");

foo.info();           // affiche: raspou
foo.setNom("fedya"); // Erreur, setNom est private
```

Encapsulation

Principe d'encapsulation

Le principe d'encapsulation :

- permet d'empêcher une mauvaise manipulation par un utilisateur d'une classe ou d'un objet qui pourrait conduire à un disfonctionnement.

Exemple de la vie courante :

L'utilisateur ne peut pas atteindre directement l'argent contenu dans le distributeur et doit passer par une interface.

Principe d'encapsulation

La base du principe d'encapsulation :

- Ne mettre aucun attribut d'une classe en '**public**'
- Mettre les attributs en '**private**' sinon '**protected**'.

Sans le príncipe d'encapsulation :

```
class Humain {
    public age: number;
    constructor() {
    }
}
```

Utilisation :

```
let georges: Humain = new Humain();

Georges.age = -12;
```

Avec un attribut 'public', nous n'avons aucun moyen de vérifier les données enregistrées

Principe d'encapsulation

Exemple de mise en œuvre du principe d'encapsulation :

Avec le príncipe d'encapsulation :

```
class Humain {
    private age: number = 0;
    constructor() {
    }
    public setAge(age: number) {
        if( age>0 && age<120 ) {
            this.age = age;
        }
    }

    public afficheAge() {
        console.log(this.age);
    }
}
```

Utilisation :

```
let georges: Humain = new Humain();

georges.age = -12; // interdit

georges.setAge(-12);
georges.afficheAge(); // affiche: 0

georges.setAge(12);
georges.afficheAge(); // affiche: 12
```

Désormais, nous pouvons filtrer les données avant de les enregistrer.

Accesseur / Mutateur (getter/setter)

Lors de la mise en place du principe d'encapsulation nous avons malgré tout besoin d'accéder à certains attributs :

- Accesseur ou getter est une méthode pour accéder à la valeur d'un attribut
- Mutateur ou setter est une méthode pour modifier la valeur d'un attribut.

```
class Point {
    private _x: number = 0;
    private _y: number = 0;

    constructor() {
    }
    get x(): number {
        return this._x;
    }
    set x(newX: number) {
        if( newX > 0 ) {
            this._x = newX;
        }
    }
    get y(): number {
        return this._y;
    }
    set y(newY: number) {
        if( newY > 0 ) {
            this._y = newY;
        }
    }
}
```

Utilisation :

```
let pt0: Point = new Point();

pt0.x = 12;
pt0.y = 5;
console.log(pt0.x, pt0.y);
// affiche: 12 5



pt0.x = -8;
pt0.y = -2;
console.log(pt0.x, pt0.y);
// affiche toujours: 12 5
```

Interfaces

Interfaces

TypeScript permet l'utilisation du **duck typing** qui se rapproche du polymorphisme.

Duck typing :

- Ca cancane comme un canard, ça marche comme un canard, c'est donc un canard.
- Permet d'utiliser dans une fonction des objets de types différents mais qui ont un comportement commun.

Exemple ci-dessous:

- La forme cylindrique rentre dans le trou ainsi que la sphère
  - ✔ Nous considérons donc que les 2 objets ont le comportement attendu

Interfaces

Le duck typing:

- Se base sur une série d'attributs et de méthodes attendus.
- L'objet est considéré valide quel que soit sa classe s'il respecte les attributs et les méthodes attendus.

Nous utilisons ce que l'on appelle une **interface** pour mettre en œuvre le duck typing.

Sans interface :

```
function test(obj) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}

let o1 = {};
let o2 = { x: "coucou" };
let o3 = { x: 10 };

test(o1);  // affiche: NaN car undefined + 3
test(o2);  // affiche: coucou3
test(o3);  // affiche: 13
```

Nous n'avons pas filtré les objets .
Le résultat est donc aléatoire et peut dans certains cas conduire à des erreurs.

Interfaces

Avec interface: 1ère écriture

```
function test( obj: {x: number} ) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}
```

Avec interface: 2ème écriture

```
interface xNumber {
    x: number;
}

function test( obj: xNumber) {
    console.log( obj.x + 3 ); // additionne l'attribute x avec 3
}
```

Utilisation

```
let o1 = {};
let o2 = { x: "coucou" };
let o3 = { x: 10 };

test(o1);  // erreur car pas d'attribute x
test(o2);  // erreur car x n'est pas de type number
test(o3);  // affiche: 13
```

Nous avons bien filtré les objets, seuls ceux ayant un comportement attendu sont acceptés

Interfaces

Il est préférable d'utiliser l'écriture complète de l'interface car plus lisible et facilement réutilisable

Interfaces

Il est possible d'utiliser les interfaces pour ne garder que les objets ayant certaines méthodes.

Sans interface :

```
function testeAffiche(obj) {
    obj.affiche("John");
}

let o1 = {
    affiche: function(prenom: string){
        console.log("coucou "+prenom);
    }
};
testeAffiche(o1);  // affiche: coucou John

let o2 = {};
testeAffiche(o2);  // erreur: testeAffiche appel une function qui
n'existe pas
```

Interfaces

Avec interface :

```
interface AfficheFunction {
    affiche(prenom: string): void;
}

function testeAffiche(obj: AfficheFunction) {
    obj.affiche("John");
}

let o1 = {
    affiche: function(prenom: string){
        console.log("coucou "+prenom);
    }
};
testeAffiche(o1);  // affiche: coucou John

let o2 = {};
testeAffiche(o2);  // erreur: o2 n'a pas de méthode affiche
```

Les promesses

Promesses

Promesse (*promise*):

- Simplifie la programmation asynchrone
- Évite l'abus des callbacks d'ES5
- C'est une valeur disponible immédiatement, dans le futur ou jamais. Une promesse on peut la tenir ou pas.

Avec callback

```
niveau1(value, function(info){
        niveau2(info, function(info2){
                niveau3(info2, function(info3){
                        // instructions
                });
        });
});
```

Avec promesse

```
niveau1
   .then(niveau2)
   .then(niveau3)
```

Promesses

Promesse :

- Est un objet
- 4 états :
  - pending (en attente) : état initial, la promesse n'est ni remplie, ni rompue ;
  - fulfilled (tenue) : l'opération a réussi ;
  - rejected (rompue) : l'opération a échoué ;
  - settled (acquittée) : la promesse est tenue ou rompue
- On peut fournir un callback en cas de succès (obligatoire) et un callback d'echec (optionnel)

Création promesse

```
let promesse =new Promise( function (fnReussite, fnEchec) {
        let response Requête_Ajax("John");
        if (response.status === 200) {
                fn_reussite(response.data);
        } else {
                fnEchec('No user');
        }
});
```

Utilisation promesse

```
promesse
        .then( function(resultat) {
                console.log(resultat);
                },
                function(erreur) {
                        console.log(erreur);
                })
```

Les modules

Modules

Module :

- Un module est une façon de découper du code pour le réutiliser facilement
- S'apparente à un **import de package** en Java ou un **include en PHP**

module1.js

```
export function test(message) {
	console.log(message);
}
export function truc() {
	console.log("Je ne fais rien");
}
```

module2.js

```
import {test, truc } from './module1.js';

test("coucou");
truc();
```

module3.js

```
import {*} from './module1.js';

test("coucou");
truc();
```

Les décorateurs

Décorateur

Décorateur :

- Ressemble aux annotations (Java, C#)
- Meta-programmation (modification des informations sur un objet/classe)

exemple.js

```
import {log} from './log.js';

class Exemple{
        @log
        maMethode(){
        }
        @log
        maMethode(param){
        }
}
```

log.js

```
export function log() {
        return (target: any, name: string, descriptor: any) => {
                console.log("appel de :"+name)
                return descriptor;
        };
}
```

- target : la methode ciblee
- name : le nom de la méthode
- descriptor : le descripteur de la méthode (le descripteur est un objet décrivant les propriétés d'un objet)