COURS 4

Conception orientée objets : Prototypes

BEQUART VALENTIN

Typage Dynamique
Faible

Interprété

Orienté Objet
(Prototype)

Fonctionnel et
événementiel

Inclusion dans une page Web

```
<html>
  <head>
    <!-- Les scripts sont chargés avant le body -->
    <script src="./myScript1.js"></script>
    <script src="./myScript2.js"></script>
    <script src="./myScript3.js"></script>
  </head>
  <body>
    <!-- Mon code HTML -->

    <!-- Les scripts sont chargés après le body -->
    <script src="./myScript4.js"></script>
    <script src="./myScript5.js"></script>
  </body>
</html>
```

Petits rappels

```
let monModule = (function() {

    let count = 0;

    return {
        inc: () => ++count,
        dec: () => --count,
        add: number => count += number,
        sub: number => count -= number
    }

})();
```

```
let monModule = (function () {
    function getModuleInterne(){
        console.log("fonction interne")
    }

    return {
        getModuleExterne(a){
            getModuleInterne();
            return "Fonction externe : "+a;
        }
    }
}) ();
```

Programmation Orientée Objet

- Plusieurs intérêts :
  - Lisibilité du code
  - Structure du code
  - Découpage du code
  - Duplication de code
  - Gestion des données

-> Sert à représenter nos données

Programmation Orientée Prototypes

- Forme de programmation orientée objets différentes des classes « classiques »
  - Les hiérarchies d'héritage peuvent être modifiées dynamiquement
  - Un objet ne contient que des « slots »
  - Les slots peuvent être ajoutés, retirés ou remplacés dynamiquement

- Un prototype est un objet de référence à partir duquel on crée d'autres objets
  - Par clonage du prototype
  - Par référence vers le prototype (En JS)

Programmation Orientée Prototypes

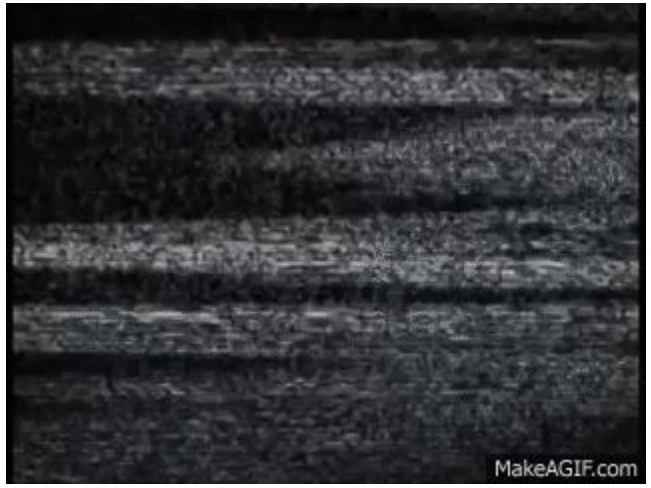
- PO à base de classe :
  - Une classe définie par son code source est statique
  - Elle représente une définition de l'objet
  - Tout objet est l'instance d'une classe
  - L'héritage se fait au niveau des classes

- PO à base de prototype
  - Un prototype défini par son code source est modifiable
  - Il est lui-même un objet et à une existence physique en mémoire
  - Il peut être appelé et modifié
  - Obligatoirement nommé
  - Un objet hérite des slots de son prototype
  - Un prototype peut être vu comme le modèle de la famille

Création d’objet

Manière bourrine
(Donc mauvaise)

Code dupliqué, mauvaise performances.




```
let frodo = {
    name: 'Frodon',
    race: 'Hobbit',
    sayHello: function() {
        console.log('Hi, my name is', this.name,
                    ', I am a', this.race, '!');
    }
};
frodo.sayHello();   // "Hi, my name is Frodon , I am a Hobbit !"
let gimli = {
    name: 'Gimli',
    race: 'Dwarf',
    sayHello: function() {
        console.log('Hi, my name is', this.name,
                    ', I am a', this.race, '!');
    }
};
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
```

Création d'objet

Possible avec des closures, MAIS nous les utilisons déjà pour nos « modules de codes ».

```
let Person = (function() {
    let display = function() {
        console.log('Hi, my name is', this.name,
                    ', I am a', this.race, '!');
    };

    return function(name,race) {
        return {
            name: name,
            race: race,
            sayHello: display
        };
    };
})();

let frodo = Person('Frodo', 'Hobbit');
let gimli = Person('Gimli', 'Dwarf');
frodo.sayHello();   // "Hi, my name is Frodo , I am a Hobbit !"
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
```

Création d'objet

Avec une fonction constructeur et l'opérateur new

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
    this.sayHello = function() {
        console.log('Hi, my name is', this.name,
                    ', I am a', this.race, '!');
    };
};
let frodo = new Person('Frodo', 'Hobbit');

console.log(frodo.name);    // "Frodo"
frodo.sayHello();   //"Hi, my name is Frodo , I am a Hobbit !"
```

Création d'objet

Toujours utiliser new avec une fonction constructeur

```
let frodo = new Person('Frodo', 'Hobbit');
let gimli = new Person('Gimli', 'Dwarf');
frodo.sayHello();   // "Hi, my name is Frodo , I am a Hobbit !"
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
```

```
let frodo = {};
Person.call(frodo, 'Frodo', 'Hobbit');
let gimli = {};
Person.call(gimli, 'Gimli', 'Dwarf');
```

Création d'objet

Problème de duplication de code:
A chaque instance de Person, sayHello est dupliquée en mémoire

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
    this.sayHello = function() {
        console.log('Hi, my name is', this.name,
                    ', I am a', this.race, '!');
    };
};

let frodo = new Person('Frodo', 'Hobbit');
let gimli = new Person('Gimli', 'Dwarf');
console.log(frodo.sayHello === gimli.sayHello); // false
```

Utilisation des prototypes

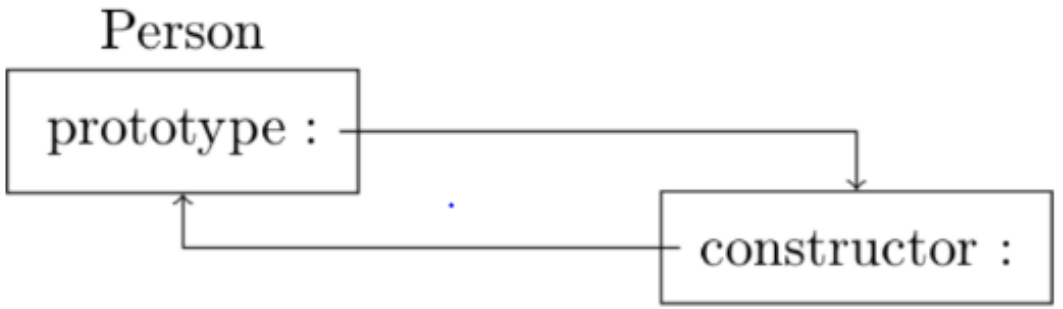
Chaque fonction constructeur possède un prototype, partagé entre toutes les instances

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
};
Person.prototype.sayHello = function() {
    console.log('Hi, my name is', this.name,
                ', I am a', this.race, '!');
};

let frodo = new Person('Frodo', 'Hobbit');
let gimli = new Person('Gimli', 'Dwarf');
frodo.sayHello();   // "Hi, my name is Frodo , I am a Hobbit !"
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
console.log(frodo.sayHello === gimli.sayHello); // true
```

Utilisation des prototypes

```
let Person = function(name) {
    this.name = name;
};
```

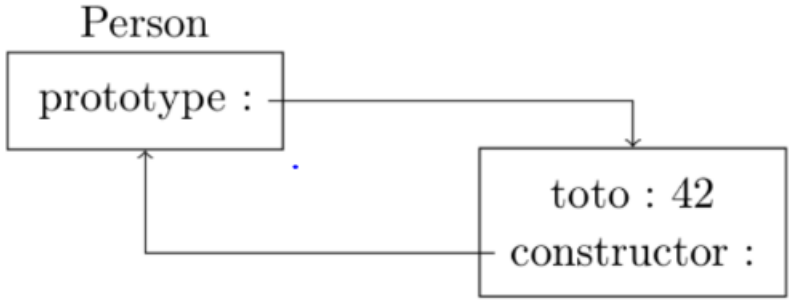
Un prototype est donc un objet référencé par une fonction constructeur et possédant une référence vers cette dernière



Utilisation des prototypes

```
let Person = function(name) {
    this.name = name;
};
Person.prototype.toto = 42;
```

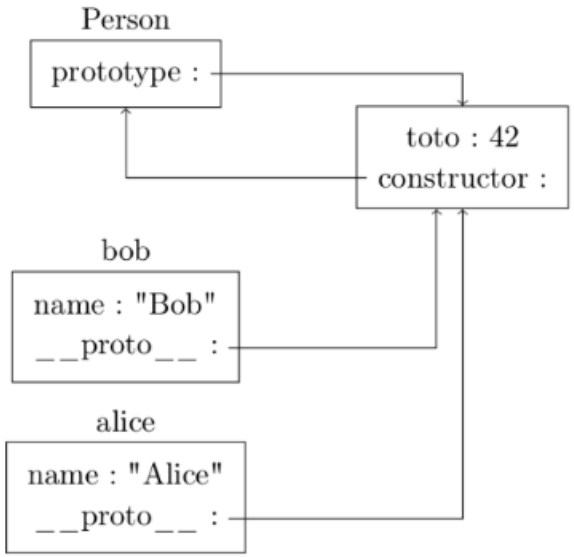
Stocker une information dans le prototype d'une fonction constructeur, c'est la stocker dans un objet à part partagé



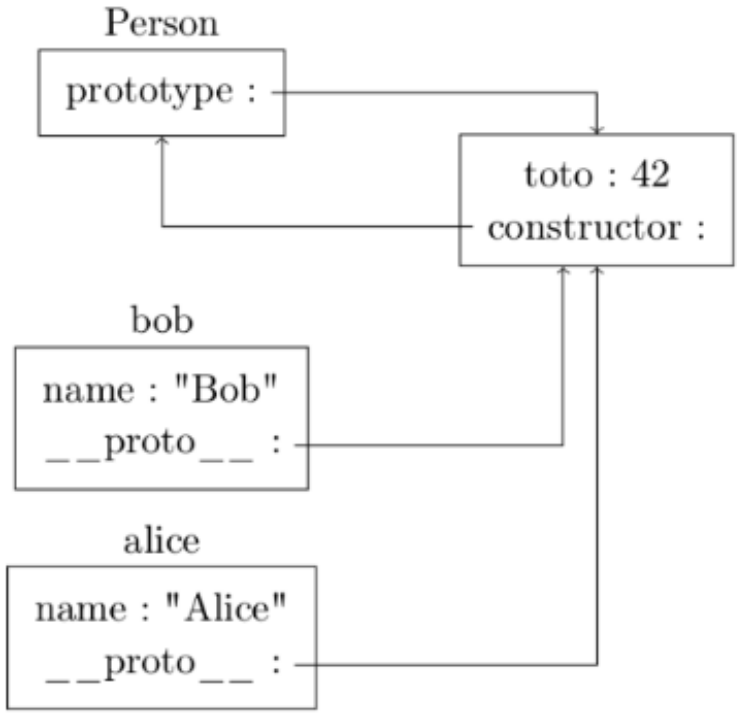
Utilisation des prototypes

```
let Person = function(name) {
    this.name = name;
};
Person.prototype.toto = 42;

let bob = new Person('Bob');
let alice = new Person('Alice');
```

Lors de l'utilisation de new, un nouvel objet est créé avec une propriété cachée proto vers le prototype



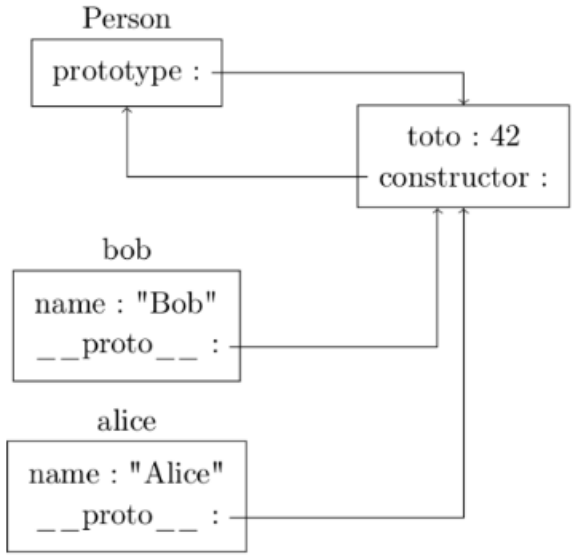
Utilisation des prototypes

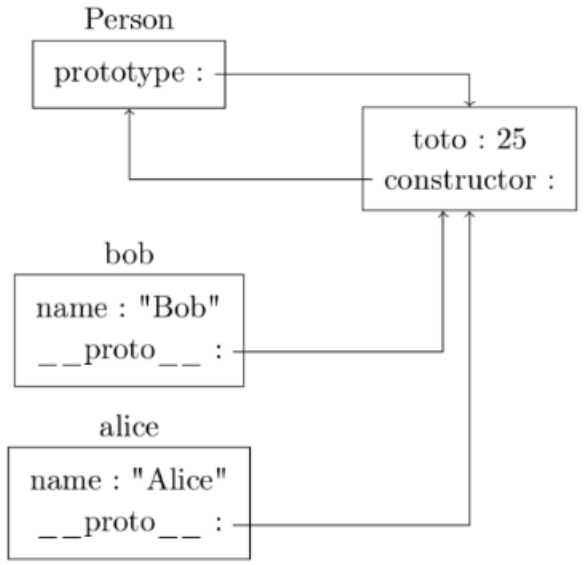



```
console.log(bob.name);  // "Bob"
console.log(bob.toto);  // 42
```

Utilisation des prototypes

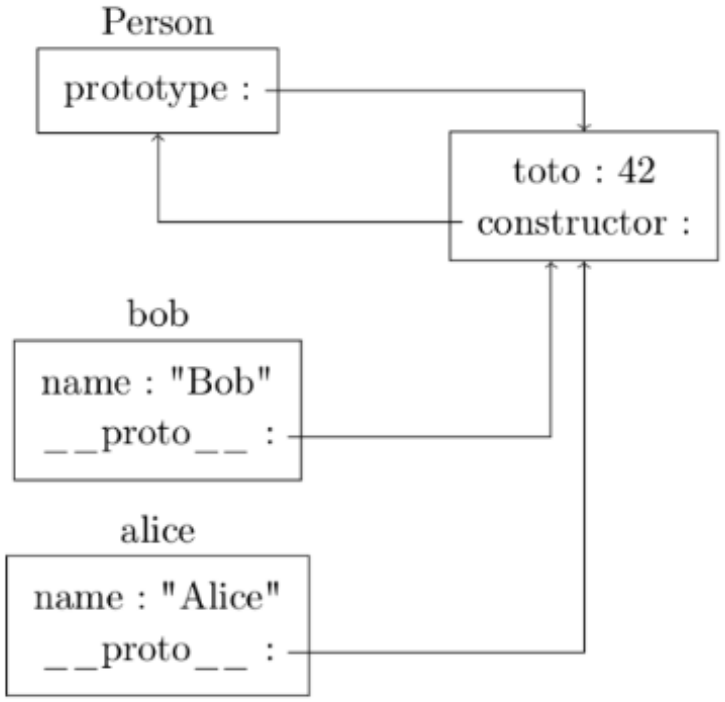
```
Person.prototype.toto = 25;
console.log(bob.toto);      // 25
console.log(alice.toto);    // 25
```

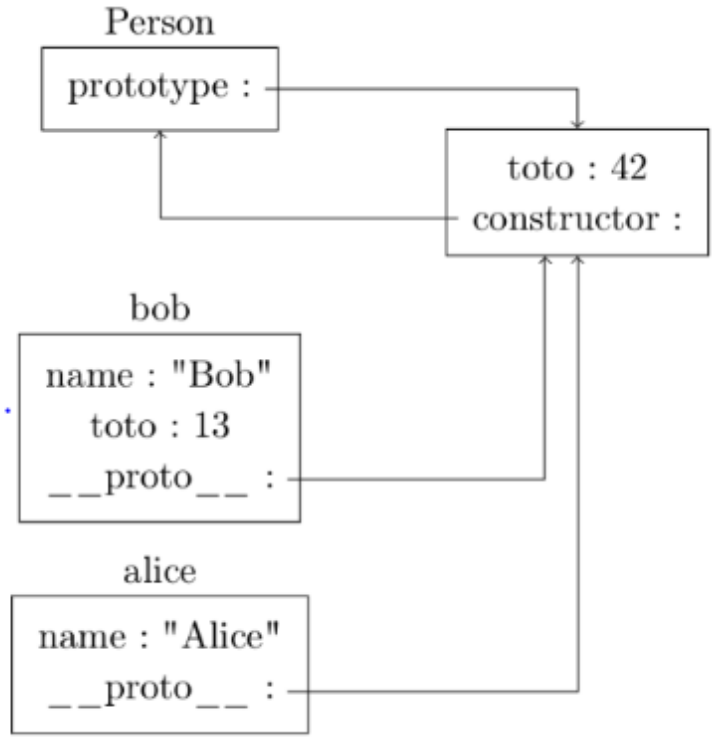




Utilisation des prototypes

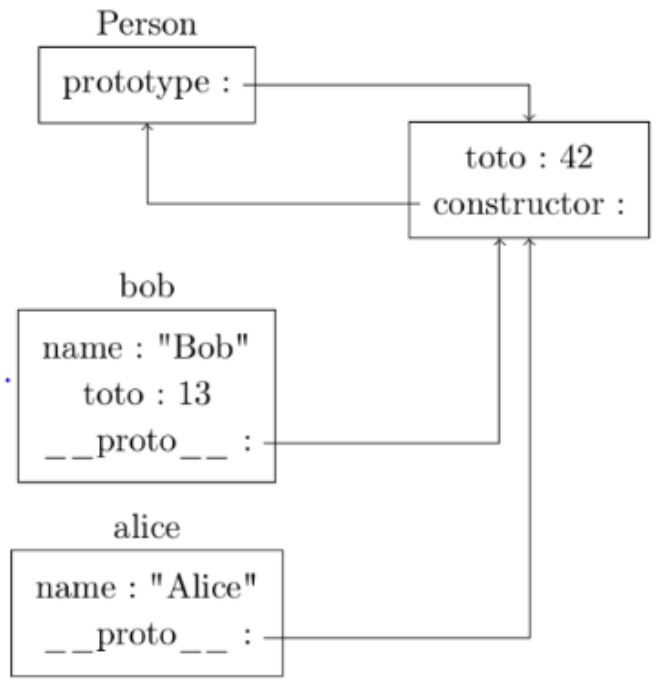
```
bob.toto = 13;
console.log(bob.toto);      // 13
console.log(alice.toto);    // 42
```

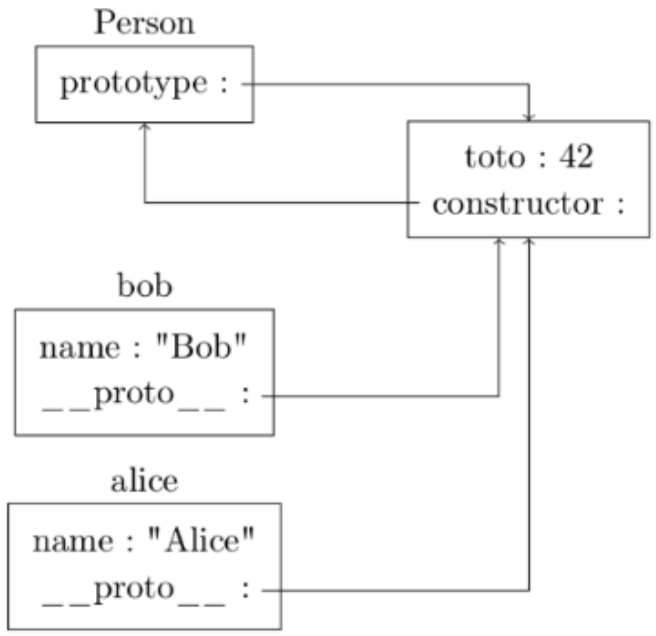




Utilisation des prototypes

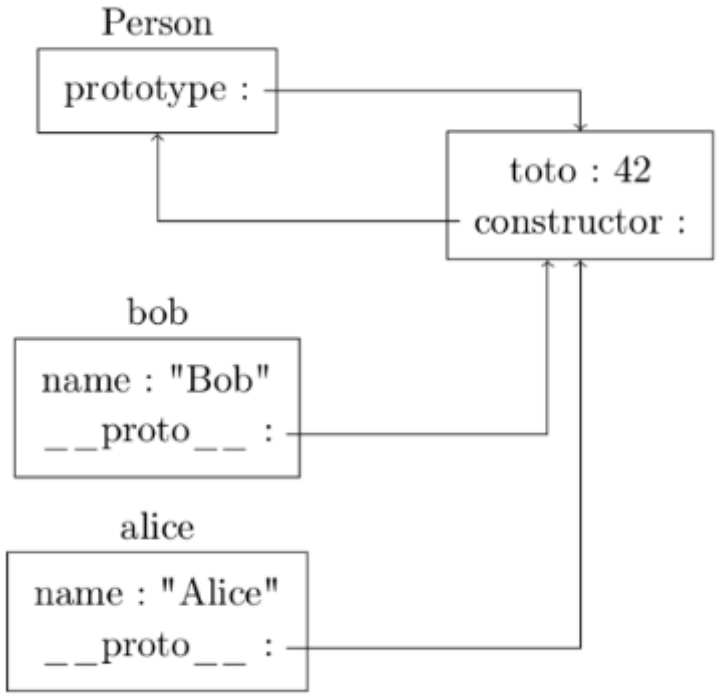
```
delete bob.toto;
console.log(bob.toto);      // 42
console.log(alice.toto);    // 42
```

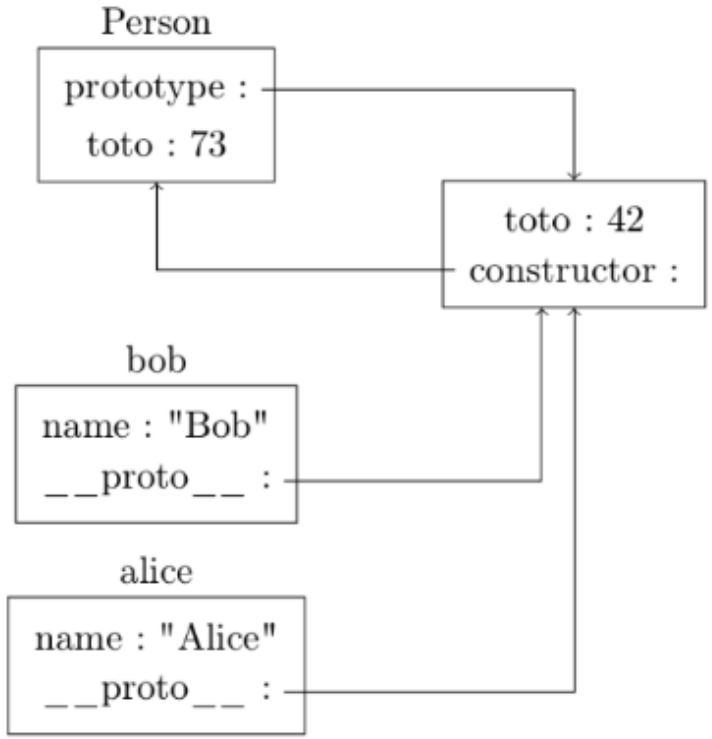




Variables de classes (statiques)

```
console.log(Person.toto);       // undefined
Person.toto = 73;
console.log(Person.toto);   // 73
console.log(bob.toto);          // 42
```





Vocabulaire

Classe : correspond à une fonction constructeur

Variable de classe : attachée à la fonction constructeur

Variable propre : attachée à l'instance

Variable membre : attachée au prototype

```
let MaClasse = function() {
    this.variablePropre = 42;
}
MaClasse.variableClasse = 13;
MaClasse.prototype.variableMembre = 73;
```

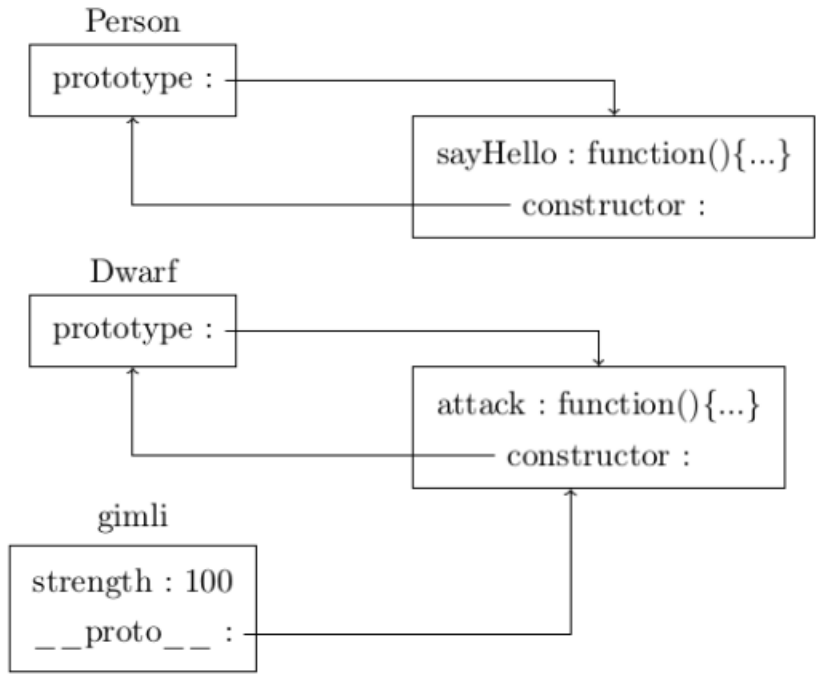
Conventions

Les variables contenant des fonctions constructeurs commencent par une majuscule (comme les classes)
Les attributs sont définis dans la fonction constructeur
Les méthodes le sont dans le prototype
Un fichier par fonction constructeur, et tous les fichiers dans un dossier class ou models

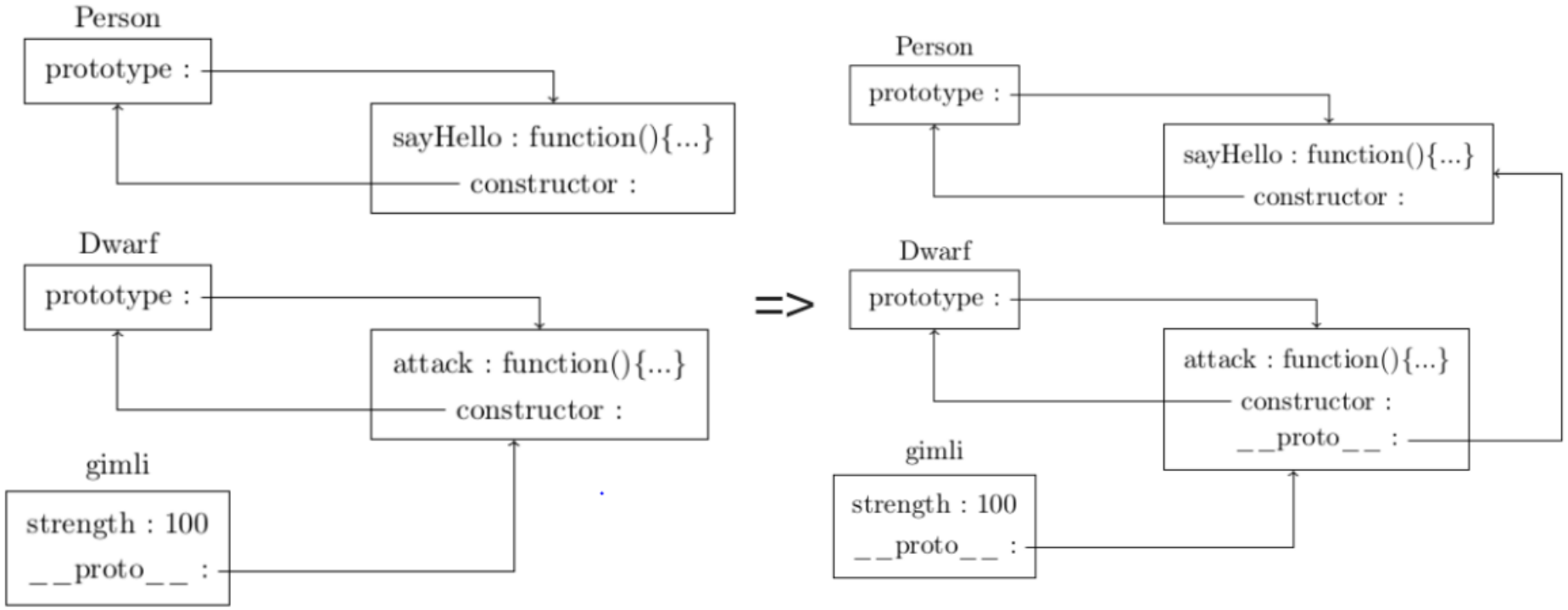
```
let MaClasse = function() {
    this.monAttribut = 42;
}
MaClasse.prototype.maMethode = function() { /* mon code */ };
```

Héritage

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
};
Person.prototype.sayHello = function() {
    console.log('Hi, my name is', this.name, ', I am a', this.race, '!');
};

let Dwarf = function(name) {
    this.strength = 100;
};
Dwarf.prototype.attack = function() {
    console.log('And my axe !');
};

let gimli = new Dwarf('Gimli');
gimli.attack();     // "And my axe !"
gimli.sayHello();   // ReferenceError
```

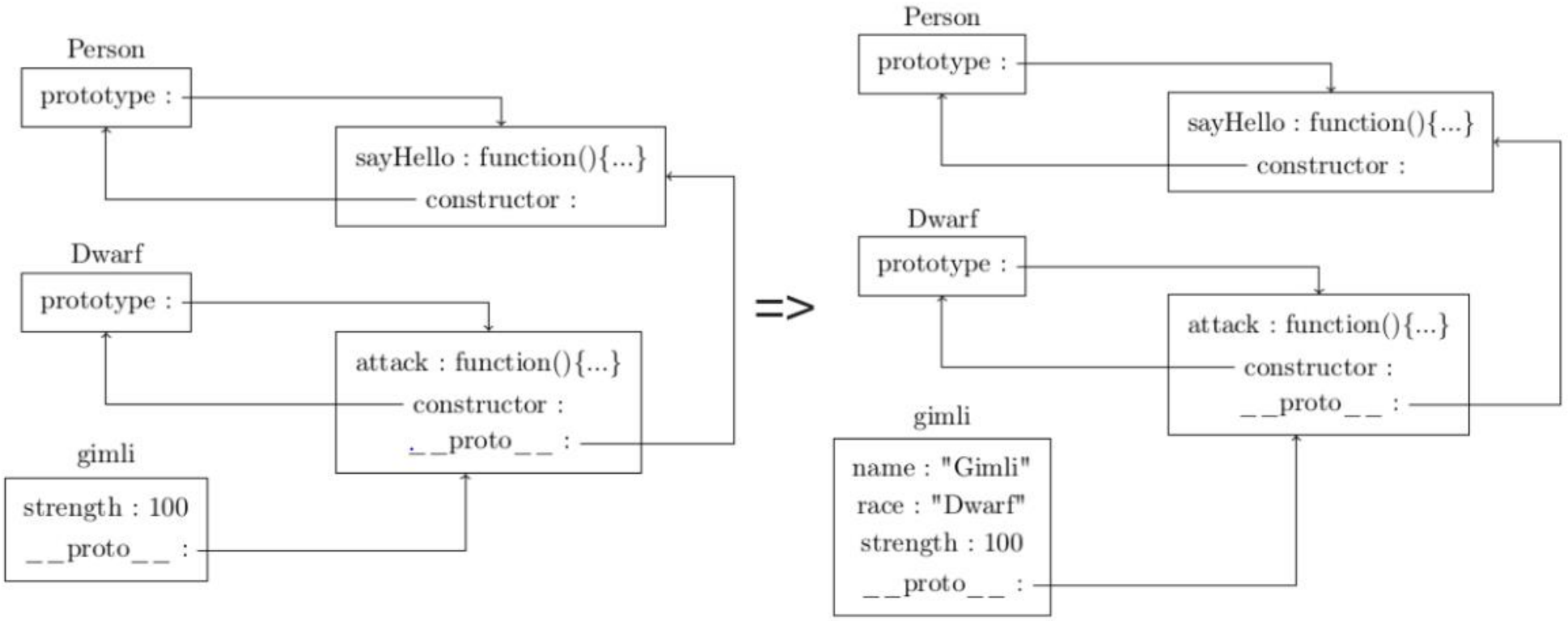


Héritage




```
Dwarf.prototype = Object.create(Person.prototype);
Dwarf.prototype.constructor = Dwarf;
```

Héritage



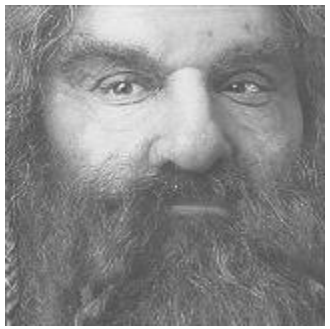

```
let Dwarf = function(name) {
    Person.call(this, name, 'Dwarf');
    this.strength = 100;
};
```

Héritage

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
};
Person.prototype.sayHello = function() {
    console.log('Hi, my name is', this.name,
                ', I am a', this.race, '!');
};

let Dwarf = function(name) {
    Person.call(this, name, 'Dwarf');
    this.strength = 100;
};
Dwarf.prototype = Object.create(Person.prototype);
Dwarf.prototype.constructor = Dwarf;
Dwarf.prototype.attack = function() {
    console.log('And my axe !');
};

let gimli = new Dwarf('Gimli');
gimli.attack();     // "And my axe !"
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
```

Surcharge de méthodes

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
};
Person.prototype.sayHello = function() {
    console.log('Hi, my name is', this.name,
                ', I am a', this.race, '!');
};

let Dwarf = function(name) {
    Person.call(this, name, 'Dwarf');
    this.strength = 100;
};
Dwarf.prototype = Object.create(Person.prototype);
Dwarf.prototype.constructor = Dwarf;
Dwarf.prototype.sayHello = function() {
    Person.prototype.sayHello.call(this);
    console.log('And my axe !');
};

let gimli = new Dwarf('Gimli');
gimli.sayHello();   // "Hi, my name is Gimli , I am a Dwarf !"
                    // "And my axe !"
```



Méthodes abstraites et exceptions

```
let Person = function(name, race) {
    this.name = name;
    this.race = race;
};
Person.prototype.sayHello = function() {
    throw new Error('Must be overrided');
};

let Dwarf = function(name) {
    Person.call(this, name, 'Dwarf');
};
Dwarf.prototype = Object.create(Person.prototype);
Dwarf.prototype.constructor = Dwarf;
Dwarf.prototype.sayHello = function() {
    console.log('I am', this.name, 'the Dwarf !');
};

let gimli = new Dwarf('Gimli');
gimli.sayHello();   // "I am Gimli the Dwarf !"

let gandalf = new Person('Gandalf', 'Wizard');
gandalf.sayHello(); // Error: Must be overrided
```

Instance of

```
let Being = function() {};

let Person = function() {};
Person.prototype = Object.create(Being.prototype);
Person.prototype.constructor = Person;

let Dwarf = function() {};
Dwarf.prototype = Object.create(Person.prototype);
Dwarf.prototype.constructor = Dwarf;

let Wizard = function() {};
Wizard.prototype = Object.create(Person.prototype);
Wizard.prototype.constructor = Wizard;

let gandalf = new Wizard();
let treebear = new Being();
```

```
console.log(gandalf instanceof Wizard);     // true
console.log(gandalf instanceof Dwarf);      // false
console.log(gandalf instanceof Person);     // true
console.log(gandalf instanceof Being);      // true
console.log(treebear instanceof Being);     // true
console.log(treebear instanceof Person);    // false
```

Utilisation de class

« Les classes JavaScript ont été introduites avec ECMAScript 2015. Elles sont un « sucre syntaxique » par rapport à l'héritage prototypal. En effet, cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript ! Elle fournit uniquement une syntaxe plus simple pour créer des objets et manipuler l'héritage. »

Utilisation de class

```
class Person {

    constructor(name, race) {
        this.name = name;
        this.race = race;
    }

    sayHello() {
        console.log('Hi, my name is', this.name, ', I am a', this.race, '!');
    }

}

let frodon = new Person('Frodon', 'Hobbit');
console.log(frodon.name);
```

Utilisation de class - Héritage

```
class Dwarf extends Person {

    constructor(name, race, strength) {
        super(name, race); // appelle le constructeur parent avec le paramètre
        this.strength = strength;
    }

    attack() {
        console.log('And my axe !');
    }

}
```

Utilisation de class – exemple complet

```
class Person {

    constructor(name, race) {
        this.name = name;
        this.race = race;
    }

    sayHello() {
        console.log('Hi, my name is', this.name, ', I am a', this.race, '!');
    }

}

class Dwarf extends Person {

    constructor(name, race, strength) {
        super(name, race); // appelle le constructeur parent avec le paramètre
        this.strength = strength;
    }

    attack() {
        console.log('And my axe !');
    }

}

let frodon = new Person('Frodon', 'Hobbit');
console.log(frodon.name);

let gimli = new Dwarf('Gimli', 'Dwarf', 150);
console.log(gimli.strength);
gimli.sayHello();
```

Utilisation de class - Override

```
class Person {

    constructor(name, race) {
        this.name = name;
        this.race = race;
    }

    sayHello() {
        console.log('Hi, my name is', this.name, ', I am a', this.race, '!');
    }

}

class Dwarf extends Person {

    constructor(name, race, strength) {
        super(name, race); // appelle le constructeur parent avec le paramètre
        this.strength = strength;
    }

    speak() {
        super.sayHello();
        console.log('And my axe !');
    }

}

let frodon = new Person('Frodon', 'Hobbit');
console.log(frodon.name);

let gimli = new Dwarf('Gimli', 'Dwarf', 150);
console.log(gimli.strength);
gimli.speak();
```

On corrige le TP?

C’est l’heure du TP!

Exercice 1 (par Binôme)

L’objectif du premier exercice : Développer la classe Observable qui servira de gestionnaire d’évènement.

3 Fonctions sont à implémenter :

- on(eventName, callback) : enregistre une callback en rapport avec un événement particulier donc le nom (eventName) est une chaine de caractères
- off(eventName, callback) : supprime une callback en rapport avec un événement particulier
- trigger(eventName, parameter1, ...) : exécute le callback correspondant au nom et au nombre de paramètre donné

Regardez le main.js, il contient des tests préparés qui vous permettront de voir si votre classe réagit correctement. Pas de variables globales hormis Observable, si vous choisissez les prototypes.
Il serait judicieux de développer une deuxième classe répertoriant un nom et un callback pour leur stockage…
Pensez aux méthodes javascript find, findIndex…

Exercice 2 (par Binôme)

L’objectif du deuxième exercice : Développer la classe TicTacToe qui permettra de manipuler un jeu en mémoire

- Aucun affichage demandé pour cet exercice
- Cette classe hérite de votre classe Observable
- Un évènement est déclenché lorsqu’un coup valide est joué, ou que la partie est terminée
- La classe se trouve dans le fichier TicTacToe.js
- Pas de variables globales (hormis celles réellement nécessaires)
- Les tests unitaires qui vous ont été fournis vous permettront de savoir si votre classe est terminée
- Ils vous fournissent également des informations sur les fonctions à réaliser et les attributs à donner
- Choisissez entre prototype et class, pas de préférence, si ce n’est que de la logique selon votre implémentation!

Exercice 3 (par Binôme)

- En reprenant les classes précédemment développées :
  - Implémentez la classe TicTacToeView
    - Celle-ci écoute les événements d'une instance de TicTacToe et modifie la vue en conséquence.
    - Il faut ajouter un listener lors d'un clic sur une case
    - Traiter les informations du clic, mettre à jour l'état du jeu.
    - Réagissez aux événements de votre instance de TicTacToe, afin de rafraichir l'interface graphique et logger en console
    - Réfléchissez à comment utiliser la class observable? ;)