miam | Web

Janvier–avril 2023

JUNIA
Grande école d'ingénieurs
HEI·ISEN·ISA

I. POO

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

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

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

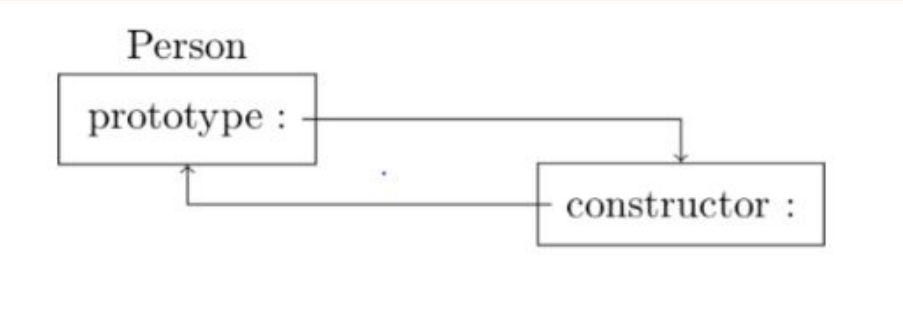
Chaque fonction constructeur possède un prototype, partagé entre toutes les instances

```
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;
};
```

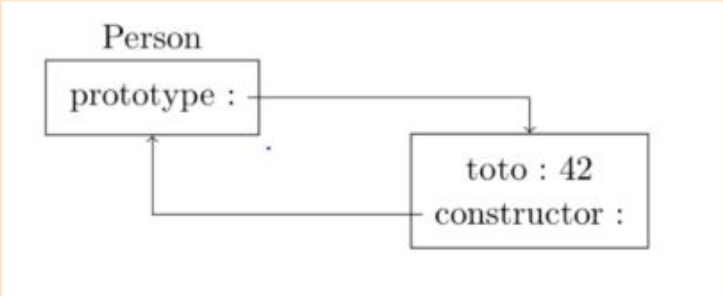
Un prototype est donc un objet référencé par une fonction constructeur et possédant une référence vers cette dernière



Utilisation des prototypes

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

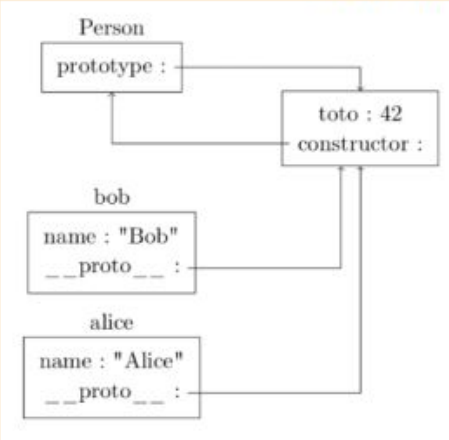
Stocker une information dans le prototype d'une fonction constructeur, c'est la stocker dans un objet à part partagé



Utilisation des prototypes

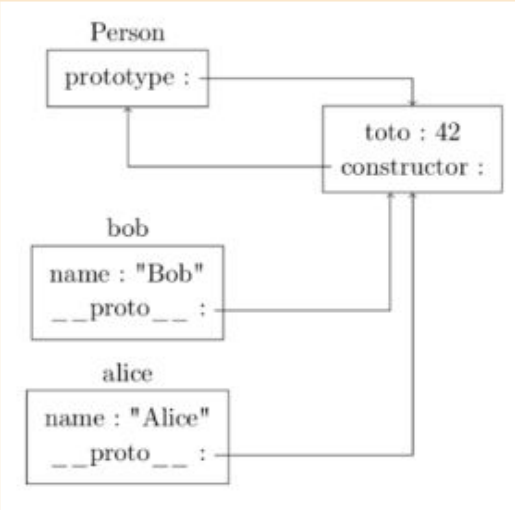
```
let Person = function(name) {
    this.name = name;
};
Person.prototype.toto = 42;

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

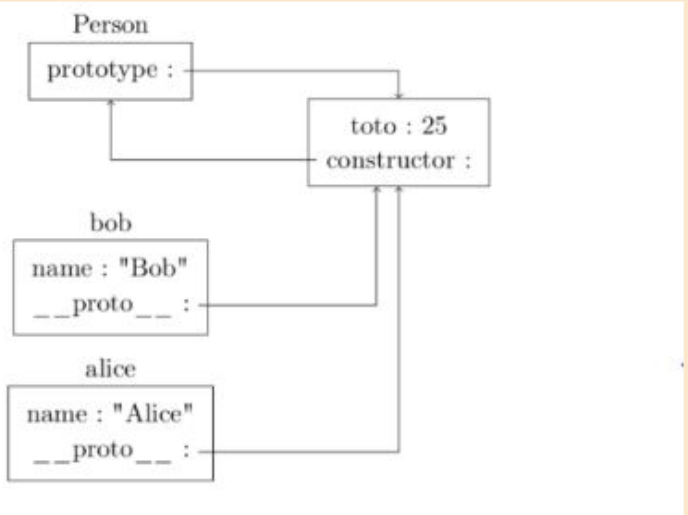
Lors de l'utilisation de new, un nouvel objet est créé avec une propriété cachée proto vers le prototype



Utilisation des prototypes

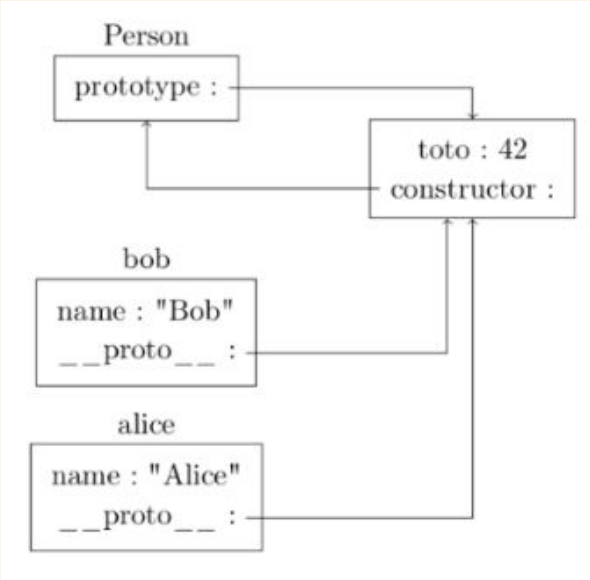
```
Person.prototype.toto = 25;
console.log(bob.toto);      // 25
console.log(alice.toto);    // 25
```

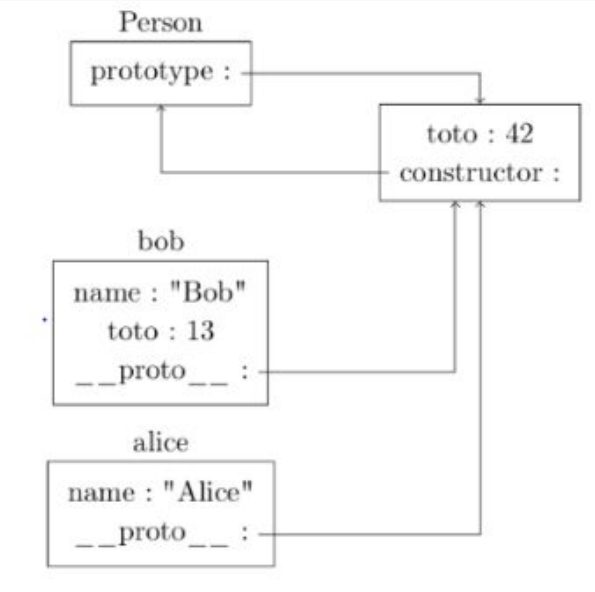




Utilisation des prototypes

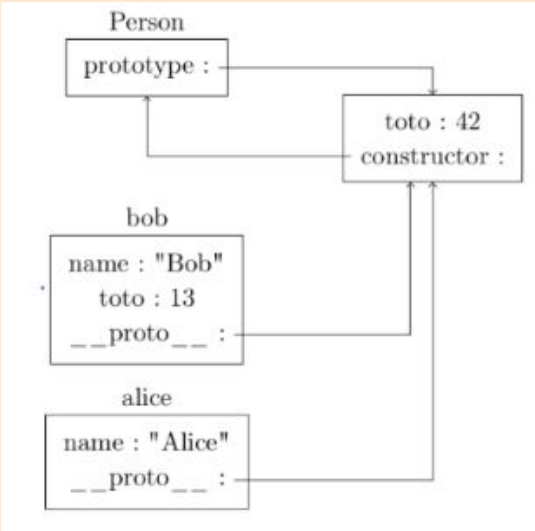
```
bob.toto = 13;
console.log(bob.toto);      // 13
console.log(alice.toto);    // 42
```

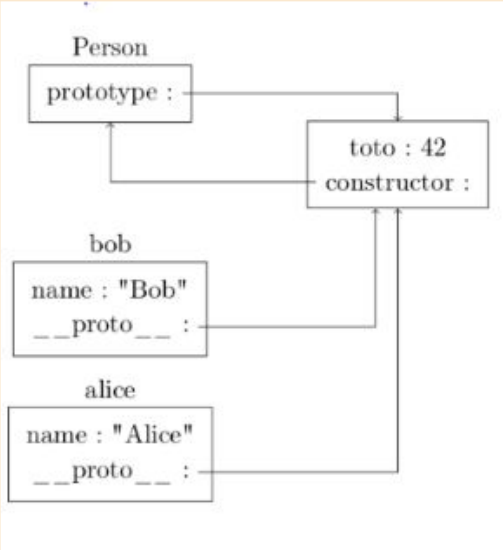




Utilisation des prototypes

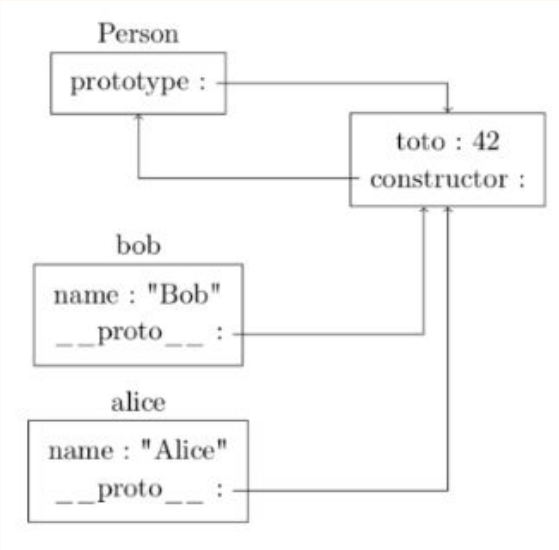
```
delete bob.toto;
console.log(bob.toto);      // 42
console.log(alice.toto);    // 42
```

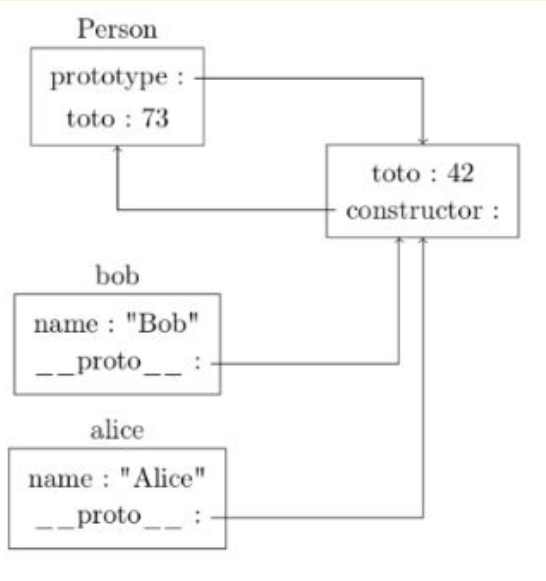




Variables de classes (statiques)

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





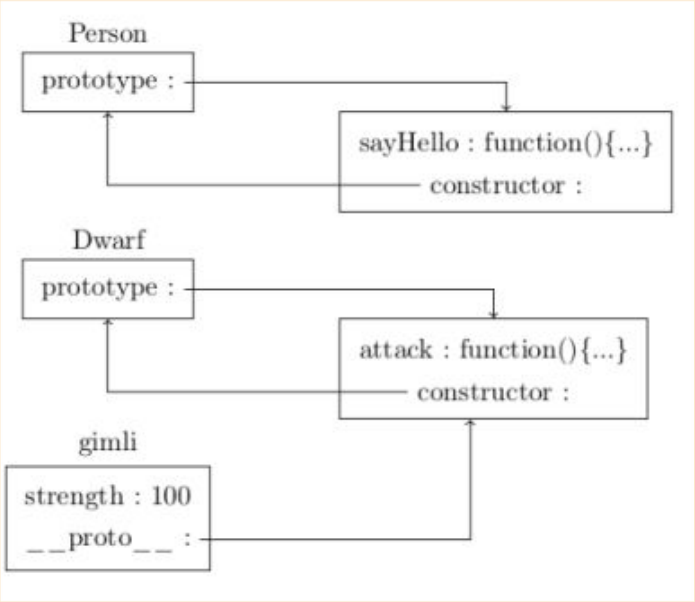
Vocabulaire

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

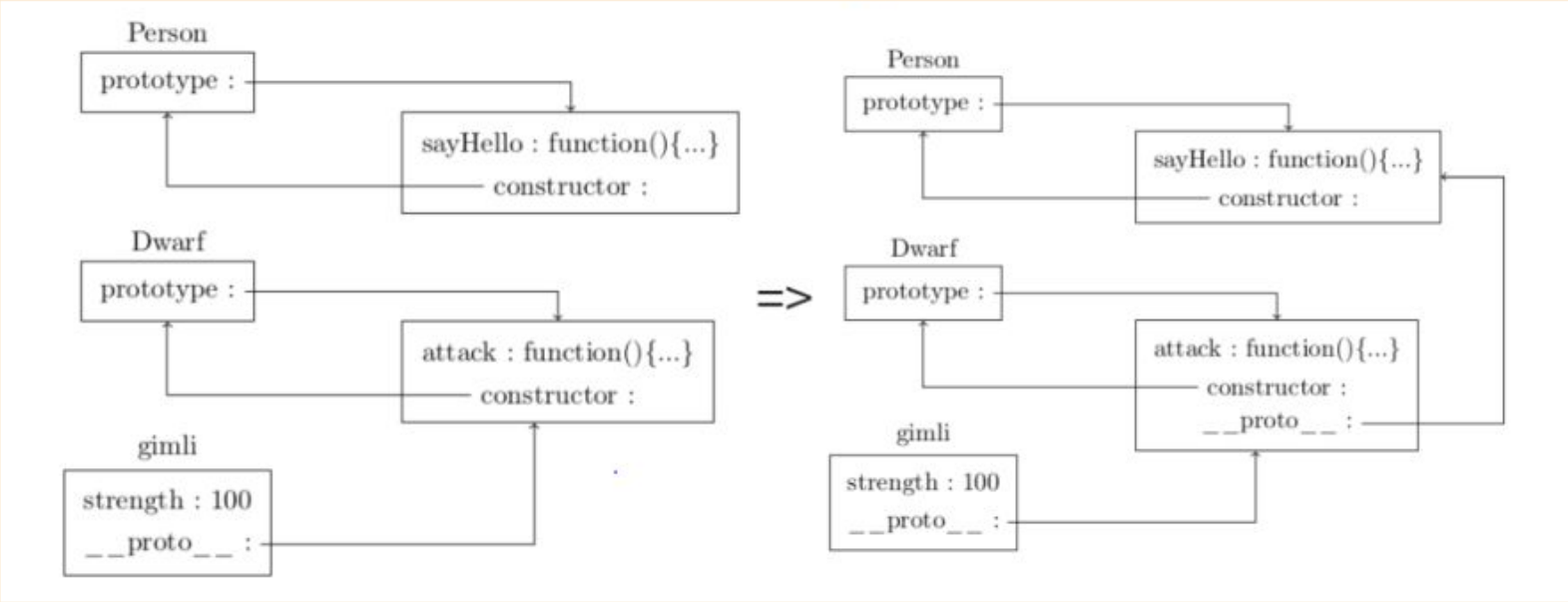
```
let MaClasse = function() {
    this.variablePropre = 42;
}
MaClasse.variableClasse = 13;
MaClasse.prototype.variableMembre = 73;
```

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
```

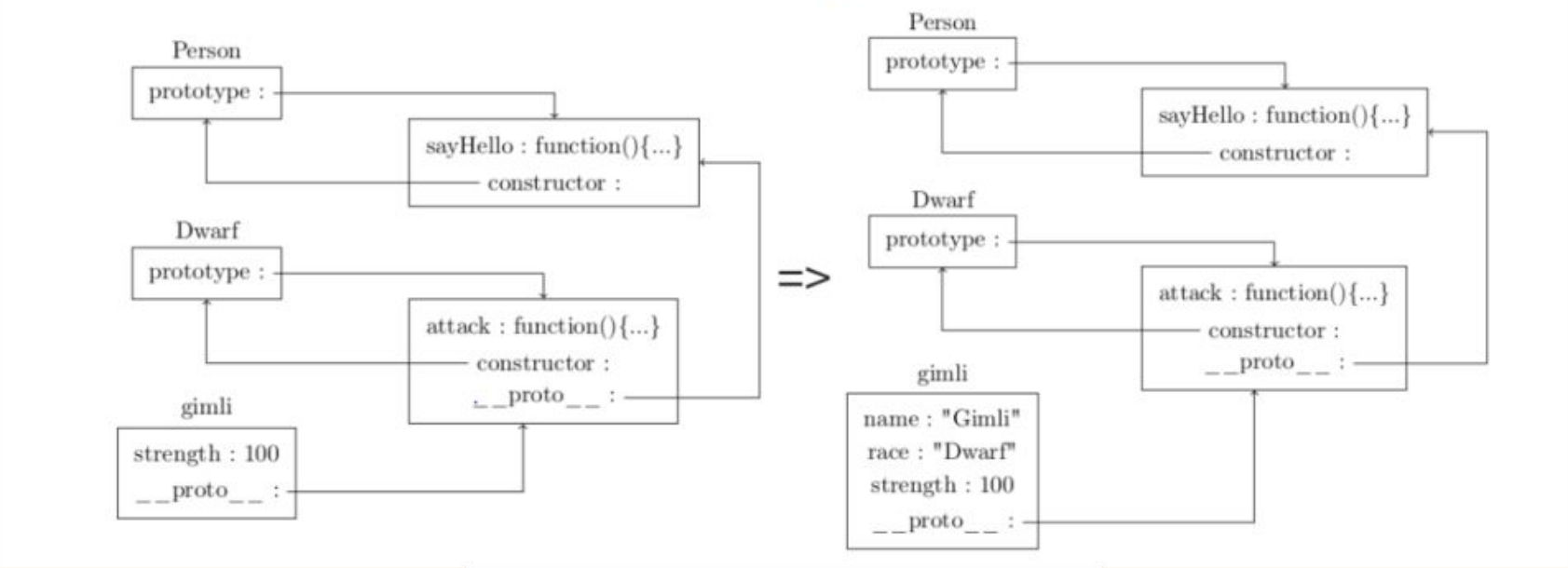


Héritage




```
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();
```

API = application programming interface

Le principe :
On expose des routes que le front vient utiliser.
Une route a une method et une url

ex :
POST https://api.leboncoin.fr/finder/search
GET https://api.leboncoin.fr/api/parrot/v3/complete

peut avoir un contenue ou non

Le back traite les demande et renvoie une réponse avec un code
2xx -> ok
3xx
4xx -> error (bad request etc)
5xx -> error (internal etc)

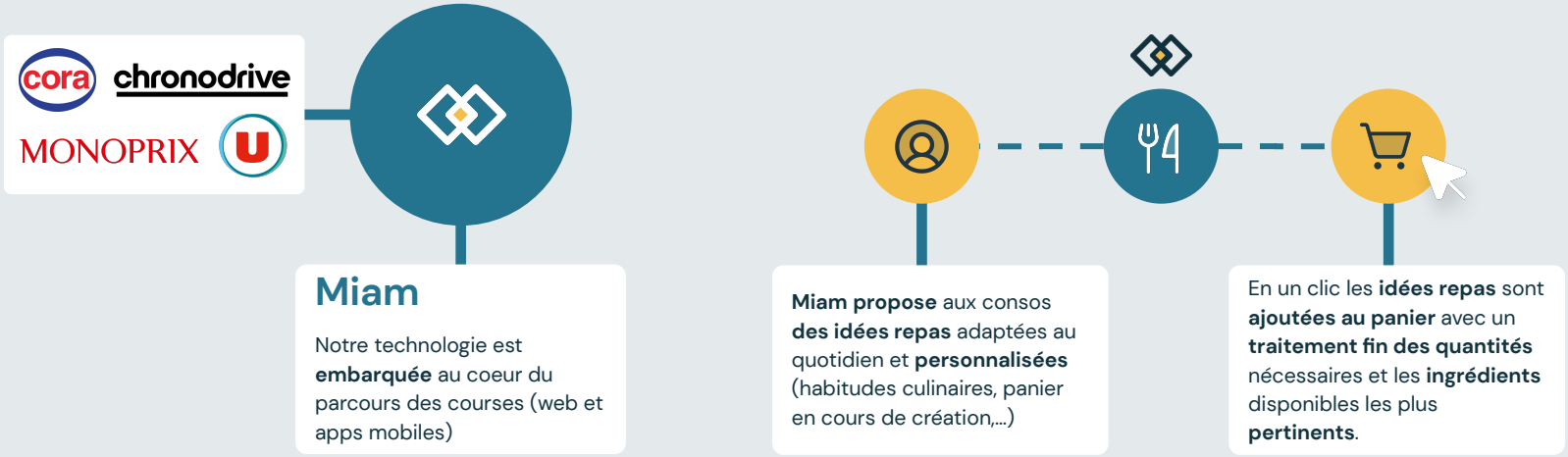
miam

Contact

alexis.tonnoir@miam.tech
06 33 74 64 10

A propos de Miam

Miam réinvente et simplifie les courses alimentaires.



Miam

Notre technologie est **embarquée** au coeur du parcours des courses (web et apps mobiles)

Miam propose aux consos **des idées repas** adaptées au quotidien et **personnalisées** (habitudes culinaires, panier en cours de création,...)

En un clic les **idées repas** sont **ajoutées au panier** avec un **traitement fin des quantités** nécessaires et les **ingrédients** disponibles les plus **pertinents**.

Qui sommes-nous ? Une startup tech lilloise, qui développe depuis 2019 des solutions d'intelligence artificielle à destination de la grande distribution.

Plus de détails sur https://miam.tech