Introduction aux objets en JavaScript

Philippe Genoud
Philippe.Genoud@univ-grenoble-alpes.fr

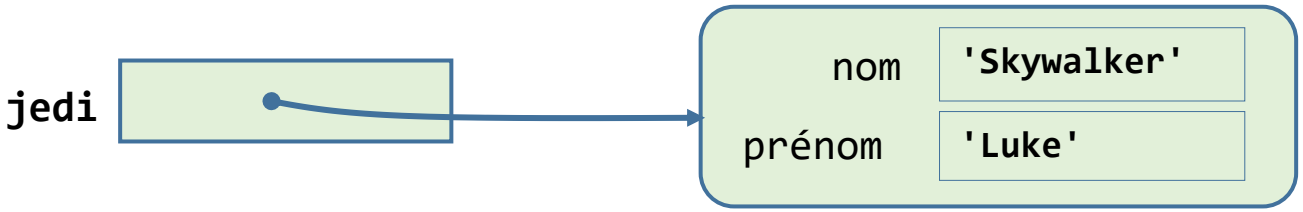
Types primitif / type objet

- Type primitif
  - Une variable de type primitif ne stocke qu'une simple valeur (chaîne de caractères, nombre, booléen…)
- Type objet
  - Un objet permet de définir des entités plus complexes en regroupant un ensemble de valeurs pouvant être soit des valeurs primitives soit d'autres objets.

```
let nom = 'Skywalker';
```

variable de type primitif
la variable stocke la valeur

```
let jedi = {
    nom : 'Skywalker',
    prenom: 'Luke';
}
```



variable de type objet
la variable stocke la référence (adresse) de l'objet

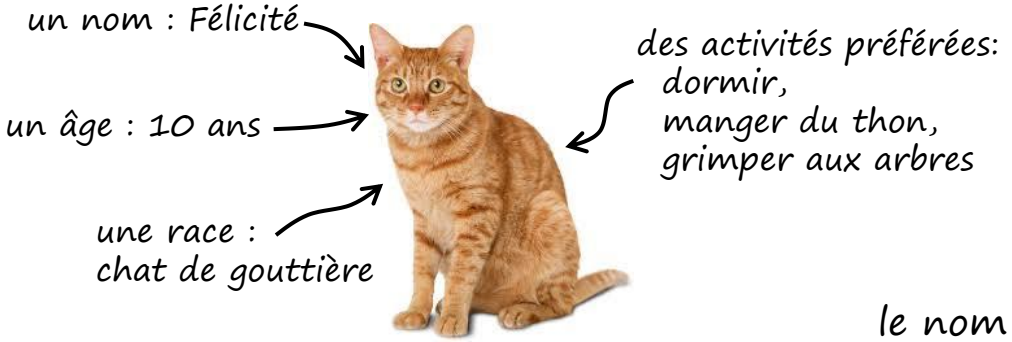
JavaScript et programmation orientée objet

- JavaScript permet l'utilisation d'objets et dispose d'objets natifs mais n'est pas à proprement parler un langage orienté objet au sens classique du terme
  - ne fournit pas d'éléments de langage pour supporter ce paradigme (par exemple pas de notion explicite de classe comme en Java, C++) mais émule certains de ses principes

- Deux types d'objets
  - Objets natifs
    - types prédéfinis : **`Array`**, **`String`**, **`Date`** …
    - objets liés à l'environnement d'exécution
      - **`window`** , **`document`** … dans le navigateur
      - **`process`** … dans NodeJs
  - Objets personnalisés
    - types définis par l'application

Syntaxe littérale

- objet JavaScript = une collection de valeurs nommées (propriétés ou attributs).
- sous sa forme littérale un objet est défini par :
  - un ensemble de couples *nom* : *valeur*, séparés par des , et délimité par { }



objet représentant ce chat

nom : "Félicité"
age : 10
race : "chat de goutière"
aime : [
"manger du thon",
"grimper aux arbres"
"dormir"]

code JavaScript : Littéral Objet

une variable référence

```
let felicite = {
    "nom": "Félicité",
    age: 10,
    race: "chat de gouttière",
    aime: ["manger du thon",
           "grimper aux arbres",
           "dormir"],
};
```

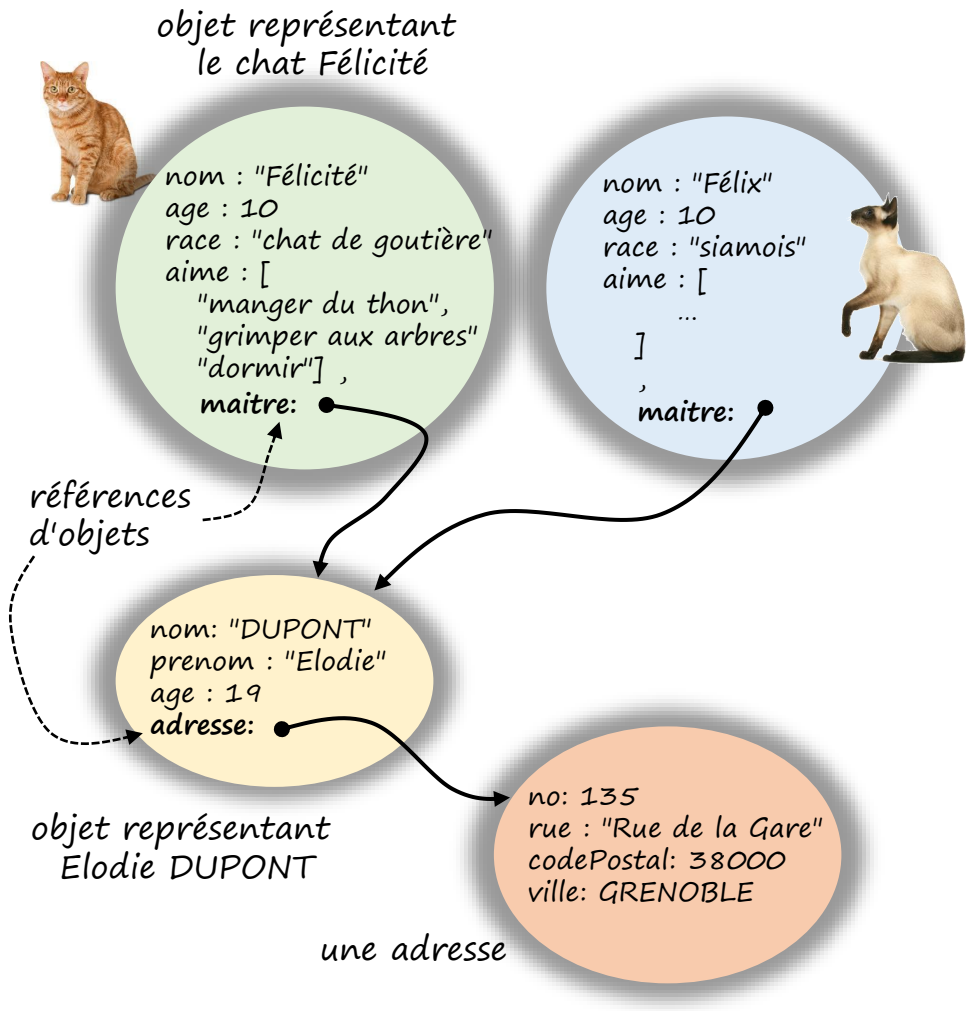
le nom d'une propriété peut être n'importe quelle chaîne de caractères (les guillemets sont facultatifs sauf si le nom est une chaîne qui ne correspond pas à un identificateur valide *)

trailing comma facultative
facilite ajout, suppression, déplacement de propriétés

*chaîne sans espaces ni caractères spéciaux (hormis $ et _) et ne commençant pas par un chiffre

Syntaxe littérale

- Objet javascript = un ensemble de propriétés (couples nom : valeur)
- Valeur d'une propriété peut être définie par n'importe quelle expression, et même depuis un autre objet littéral




```
let elodie = {
    nom: "DUPONT",
    prenom: "Elodie",
    age : 19,
    adresse:  {
        no: 135,
        rue: "Rue de la Gare",
        codePostal: 38000,
        ville: "GRENOBLE"
    } ,
};
```

objet littéral imbriqué

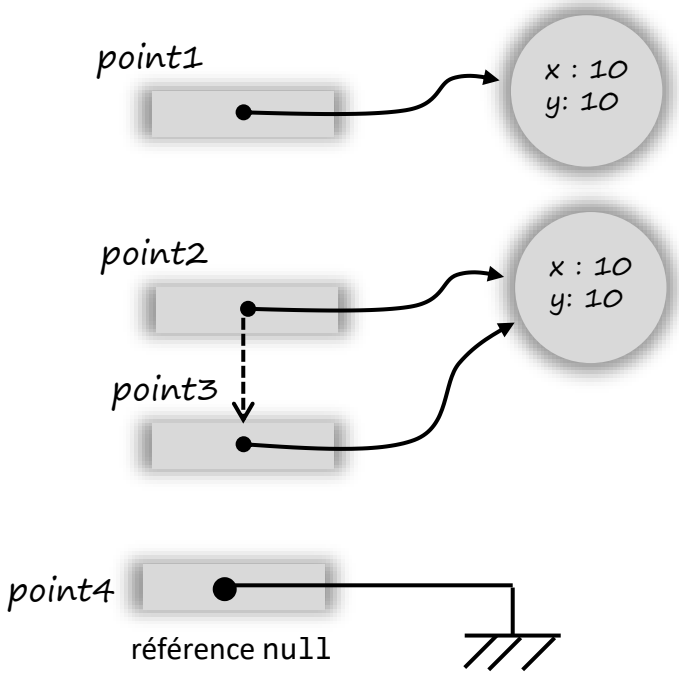
```
let felicite = {
   nom: "Félicité",
   age : 10,
   race : "chat de gouttière",
   aime : [
            "manger du thon",
            "grimper aux arbres",
            "dormir"
          ],
   maitre: elodie ,
};
```

référence

```
let felix = {
   nom: "Félix",
   age : 10,
   race : "Siamois",
   aime : [
            ...
          ],
   maitre: elodie ,
};
```

Références

- Pour désigner des objets on utilise des variables d'un type particulier : les **références**
- Une référence contient l'**adresse** d'un objet (elle pointe vers la structure de données correspondant aux propriétés (attributs) de l'objet)
- Affecter une référence à une autre référence consiste à recopier les pointeurs
- Une référence peut posséder la valeur `null` (aucun objet n'est accessible par cette référence)




```
let point1 = {
   x : 10,
   y : 10
};
```

```
let point2 = {
   x : 10,
   y : 10
};
```

```
let point3 = point2;
```

une référence n'est pas un objet, c'est un nom pour accéder à un objet

```
let point3 = null;
```

Accès à la valeur d'une propriété d'un objet

- notation pointée

```
ref.nomDePropriété
```

```
if (felicite.age > 2) {
   console.log("MAAOUU");
} else {
   console.log("miaou");
}
```

ne peut être utilisée que si le nom de la propriété est un identificateur valide (chaîne sans espaces ni caractères spéciaux (hormis `$` et `_`) et ne correspondant pas à un mot réservé du langage)

- notation crochets (*brackets*)

```
ref[expressionDeTypeString]
```

```
if (felicite["age"] > 2) {
 ...
```

une alternative à la notation pointée, indispensable si le nom de la propriété ne correspond pas à un identificateur valide

Accès à la valeur d'une propriété d'un objet

```
let myObject = {
    nom :  "attribut dont le nom est un identificateur",
    $nom : "autre attribut 'normal'",
    let : 'attribut dont le nom est un mot réservé',
    15 : 'attribut dont le nom est un entier',
    14 : 'attribut dont le nom est un entier',
    1.5 : "attribut dont le nom est un nombre flottant",
    "le prenom" : 'attribut dont le nom est un chaîne avec espaces',
    "5aaaaa" : "un attribut dont le nom n'est pas un identificateur valide"
};
```

nom de la propriété	notation pointée	notation crochets
nom	myObject.nom	myObject["nom"]
$nom	myObject.$nom	myObject["$nom"]
let	myObject.let	myObject["let"]
15	-	myObject["15"] mais aussi myObject[15]
14	-	myObject["14"] mais aussi myObject[14]
1.5	-	myObject["1.5"]
le prenom	-	myObject["le prenom"]
5aaaaa	-	myObject["5aaaaa"]

Obligatoire si le nom de la propriété n'est pas un identificateur valide

Accès à la valeur d'une propriété d'un objet

```
let elodie = {
   nom: "DUPONT",
   prenom: "Elodie",
   age : 19,
   adresse: {
      no: 135,
      rue: "Rue de la Gare",
      codePostal: 38000,
      ville: "Grenoble"
   } ,
};
```

```
let felicite = {
   nom: "Félicité",
   age : 10,
   race : "chat de gouttière",
   aime : [
            "manger du thon",
            "grimper aux arbres" ,
            "dormir"
           ],
   maitre: elodie
};
```

- accès à la valeur d'attributs d'objets imbriqués

  ex : accès au nom du maître de Félicité

  - notation pointée

    ```
    felicite.maitre.nom;
    ```

  - notation crochets (*brackets*)

    ```
    felicite["maitre"]["nom"]
    ```

  - possibilité de mélanger les différentes notations

    ```
    felicite["maitre"].nom
    ```

    ```
    felicite.maitre["nom"]
    ```

Accès à la valeur d'une propriété d'un objet

- avec la notation [ ] le nom de la propriété peut être calculé à l'exécution ou dépendre d'une valeur fournie par l'utilisateur

```
const readline = require('readline-sync');
const utils = require('./utils.js');

let felicite = {
   nom: "Félicité",
   age: 10,
   race: "chat de gouttière",
   aime: [
      "manger du thon",
      "grimper aux arbres",
      "dormir"
   ]
};

do {
   let nomProp = readline.question("nom de la propriété : ");
   console.log(`valeur de ${nomProp} : ${felicite[nomProp]}`);
} while (utils.encore('Voulez vous continuer ? '));
```

```
PS P:\M2CCI > node .\dynamicProp.js
nom de la propriété : nom
valeur de nom : Félicité
Voulez vous continuer ?  (O/N): o
nom de la propriété : race
valeur de race : chat de gouttière
Voulez vous continuer ?  (O/N): o
nom de la propriété : aime
valeur de aime : manger du thon,grimper aux arbres,dormir
Voulez vous continuer ?  (O/N): o
nom de la propriété : maitre
valeur de maitre : undefined
Voulez vous continuer ?  (O/N): n
PS P:\M2CCI >
```

renvoie la valeur **undefined** *lorsque l'on accède à une propriété qui n'est définie dans l'objet*

idem pour notation pointée

`felicite.maitre` → `undefined`

Accès à la valeur d'une propriété d'un objet

- Si on accède à un propriété qui n'existe pas la valeur retournée est `undefined`

```
let felicite = {
   "nom": "Félicité",
   age : 10,
   race : "chat de gouttière",
   aime : [
            "manger du thon",
            "grimper aux arbres",
            "dormir"
           ]
};
```

```
felicite.poids  →   undefined
```

- L'opérateur `in` permet de tester l'existence d'une propriété

```
"age" in felicite →   true
```

```
"poids" in felicite →   false
```

l'opérande gauche de `in` est une chaîne de caractères ou si c'est une expression d'un autre type elle sera convertie en chaîne de caractères

Accès à la valeur d'une propriété d'un objet

chainageOptionnel.js > ...

```
let chats = [
  {
    nom: "Félicité",
    age: 10,
    race: "chat de gouttière",
    aime: ["manger du thon", "grimper aux arbres", "dormir"],
    maitre: {
      nom: "DUPONT", prenom: "Elodie", age: 19,
      adresse: {
        no: 135,
        rue: "Rue de la Gare",
        codePostal: 38000,
        ville: "Grenoble",
      },
    },
  },
  {
    nom: "Felix",
    age: 12,
    race: "chartreux",
    aime: ["chasser les oiseaux", "dormir"],
  },
  {
    nom: "Eustache",
    age: 5,
    race: "chat siamois",
    aime: ["faire ses griffe sur le fauteuil", "dormir"],
    maitre: {
      nom: "DUPONT", prenom: "Mathieu", age: 19,
    },
  },
];
```

- chaîne de propriétés

```
for (let i = 0; i < chats.length; i++) {
  console.log(`${chats[i].nom}
      a pour maitre ${chats[i].maitre.prenom}
      habitant à ${chats[i].maitre.adresse.ville}`);
}
```

erreur pour le 2ème chat (Félix n'a pas de maître)

```
P:\M2CCI> node .\chainageOptionnel.js
Félicité
      a pour maitre Elodie
      habitant à Grenoble
chainageOptionnel.js:36
      a pour maitre ${chats[i].maitre.prenom}
                                           ^

TypeError: Cannot read properties of undefined (reading 'prenom')
    at Object.<anonymous> (P:\M2CCI\chainageOptionnel.js:36:39)
    ...

Node.js v17.7.2
P:\M2CCI>
```

Accès à la valeur d'une propriété d'un objet

chainageOptionnel.js > ...

```
let chats = [
  {
    nom: "Félicité",
    age: 10,
    race: "chat de gouttière",
    aime: ["manger du thon", "grimper aux arbres", "dormir"],
    maitre: {
      nom: "DUPONT", prenom: "Elodie", age: 19,
      adresse: {
        no: 135,
        rue: "Rue de la Gare",
        codePostal: 38000,
        ville: "Grenoble",
      },
    },
  },
  {
    nom: "Felix",
    age: 12,
    race: "chartreux",
    aime: ["chasser les oiseaux", "dormir"],
  },
  {
    nom: "Eustache",
    age: 5,
    race: "chat siamois",
    aime: ["faire ses griffe sur le fauteuil", "dormir"],
    maitre: {
      nom: "DUPONT", prenom: "Mathieu", age: 19,
    },
  },
];
```

- chaîne de propriétés
- opérateur **`.?`**

 + détails sur MDN

  - fonctionne de manière similaire à opérateur `.` mais si sur le chemin une référence est `undefined` ou `null` l'expression se court-circuite et renvoie la valeur `undefined`

```
for (let i = 0; i < chats.length; i++) {
  console.log(`${chats[i].nom}
      a pour maitre ${chats[i]?.maitre?.prenom}
      habitant à ${chats[i]?.maitre?.adresse?.ville}`);
}
```

```
P:\M2CCI> node .\chainageOptionnel.js
Félicité
      a pour maitre Elodie
      habitant à Grenoble
Felix
      a pour maitre undefined
      habitant à undefined
Eustache
      a pour maitre Mathieu
      habitant à undefined
P:\M2CCI>
```

Accès à la valeur d'une propriété d'un objet

- la boucle ***for … in*** permet d'énumérer toutes les propriétés d'un objet

```
for (let nomProp in unObjet) {
      // exécute le corps de la boucle pour chacun
      // des noms de propriété de l'objet unObjet
}
```

```
for (let prop in felicite) {
   console.log("ce chat a une propriété : " + prop);

   if (prop === "maitre") {
      console.log("Ce chat a pour maitre :");
      console.log(felicite[prop]);
   }

}
```

boucle for-in

la variable **prop** contient le nom de la propriété

c'est une chaîne de caractères

utilisation de la notation tableau pour accéder à la valeur de la propriété

```
λ node objectLiteraux2.js
ce chat a une propriété : nom
ce chat a une propriété : age
ce chat a une propriété : race
ce chat a une propriété : aime
ce chat a une propriété : maitre
Ce chat a pour maitre :
{
  nom: 'DUPONT',
  prenom: 'Elodie',
  age: 19,
  adresse: {
    no: 135,
    rue: 'Rue de la Gare',
    codePostal: 38000,
    ville: 'Grenoble'
  }
}
```

Accès à la valeur d'une propriété d'un objet

- dans quel ordre les propriétés de l'objet sont-elles accédées avec une boucle ***for … in*** ?

```
let myObject = {
    nom :  "attribut dont le nom est un identificateur",
    $nom : "autre attribut 'normal'",
    let : 'attribut dont le nom est un mot réservé',
    15 : 'attribut dont le nom est un entier',
    14 : 'attribut dont le nom est un entier',
    1.5 : "attribut dont le nom est un nombre flottant",
    "le prenom" : 'attribut dont le nom est un chaîne avec espaces',
    "5aaaaa" : "un attribut dont le nom n'est pas un identificateur valide"
};

for (let  prop in myObject) {
    console.log(prop + ' : ' + myObject[prop]);
}
```

```
14 : attribut dont le nom est un entier
15 : attribut dont le nom est un entier
nom : attribut dont le nom est un identificateur
$nom : autre attribut 'normal'
let : attribut dont le nom est un mot réservé
1.5 : attribut dont le nom est un nombre flottant
le prenom : attribut dont le nom est un chaîne avec espaces
5aaaaa : un attribut dont le nom n'est pas un identificateur valide
```

Les propriétés "entières" sont triées par ordre croissant et affichées en premier

Les autres propriétés apparaissent dans leur ordre de création

Modification des propriétés d'un objet

- changer la valeur d'une propriété
  - Par affectation directe

    ```
    felicite.race = "mélange Chartreux et " + felix.race;
    ```

  - Par opérateur

    ```
    felicite.age++;
    ```

  - Par appel d'une méthode (si la propriété est un objet ou un pseudo objet)

    ```
    felicite.aime.push("faire ses griffes sur le canapé");
    ```

- ajouter une propriété
  - si une affectation concerne une propriété non définie, une nouvelle propriété est crée

    ```
    felicite.poids = 3.5;  // à partir de ce point l'objet félicité a une propriété poids
    ```

- supprimer une propriété
  - opérateur `delete` permet de supprimer un propriété d'un objet

    ```
    delete felicite.poids; // à partir de ce point félicité.poids est undefined
    ```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);
```

```
{
  nom: 'Félicité',
  age: 10,
  race: 'chat de gouttière',
  aime: [ 'manger du thon', 'grimper aux arbres', 'dormir' ]
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);
```

```
{
  nom: 'Félicité',
  age: 10,
  race: 'chat de gouttière',
  aime: [ 'manger du thon', 'grimper aux arbres', 'dormir' ]
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);
```

```
{
  nom: 'Félicité',
  age: 11,
  race: 'mélange Chartreux + X',
  aime: [
    'manger du thon',
    'grimper aux arbres',
    'dormir',
    'faire ses griffes sur le canapé'
  ]
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);

// ajout d'une propriété
felicite.poids = 3.5;
console.log(felicite);
```

```
{
  nom: 'Félicité',
  age: 11,
  race: 'mélange Chartreux + X',
  aime: [
    'manger du thon',
    'grimper aux arbres',
    'dormir',
    'faire ses griffes sur le canapé'
  ]
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);

// ajout d'une propriété
felicite.poids = 3.5;
console.log(felicite);
```

à partir de ce point l'objet félicité a une propriété ***poids***

```
{
  nom: 'Félicité',
  age: 11,
  race: 'mélange Chartreux + X',
  aime: [
    'manger du thon',
    'grimper aux arbres',
    'dormir',
    'faire ses griffes sur le canapé'
  ],
  poids: 3.5
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
             ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);

// ajout d'une propriété
felicite.poids = 3.5;
console.log(felicite);

// supression d'une propriété
delete felicite.race;
console.log(felicite);
console.log(felicite.race);
```

```
{
  nom: 'Félicité',
  age: 11,
  race: 'mélange Chartreux + X',
  aime: [
    'manger du thon',
    'grimper aux arbres',
    'dormir',
    'faire ses griffes sur le canapé'
  ],
  poids: 3.5
}
```

Modification des propriétés d'un objet

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de gouttière",
    aime : [
              "manger du thon",
              "grimper aux arbres" ,
              "dormir"
            ]
};

console.log(felicite);

// modification d'une propriété
felicite.race = "mélange Chartreux + X";
felicite.age++;
felicite.aime.push("faire ses griffes sur le canapé");
console.log(felicite);

// ajout d'une propriété
felicite.poids = 3.5;
console.log(felicite);
```

*à partir de ce point l'objet félicité a une propriété **poids***

```
// supression d'une propriété
delete felicite.race;
console.log(felicite);
console.log(felicite.race);
```

*à partir de ce point **félicité.race** est **undefined***

```
{
  nom: 'Félicité',
  age: 11,
  aime: [
    'manger du thon',
    'grimper aux arbres',
    'dormir',
    'faire ses griffes sur le canapé'
  ],
  poids: 3.5
}
undefined
```

Objets en paramètres de fonction

- Un paramètre de fonction peut être une référence d'objet
  - déclaration

Le paramètre objet est défini comme n'importe quelle autre paramètre par un simple identifiant

accès aux propriétés de l'objet

```
function miauler(unChat) {
   if (unChat.age > 2) {
      console.log(unChat.nom + " dit MAAOUU");
   } else {
      console.log(console.log(unChat.nom + " dit miaou");
   }
}
```

  - appel

appel de la fonction avec la référence de l'objet sur lequel l'appliquer.

```
miauler(felicite);
```

```
Félicité dit MAAOUU
```

```
miauler( { nom: 'felix', age: 1.5} );
```

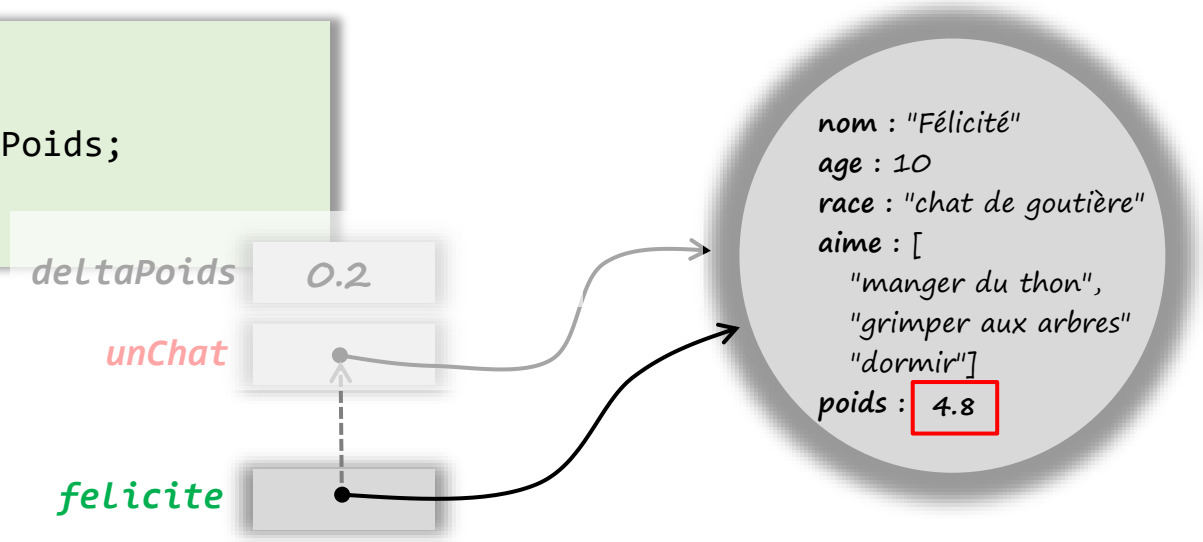
```
Félix dit miaou
```

appel de la fonction avec un littéral objet

Objets en paramètres de fonction

- comme en Java le passage de paramètres de fonctions est un passage par valeurs
  - la fonction dispose d'une copie de la variable, toute modification de cette copie ne sera pas visible en dehors de la fonction.
  - si la variable est une référence, les propriétés de l'objet référencé peuvent elles être modifiées.

```
function maigrir(unChat, deltaPoids) {

    unChat.poids = unChat.poids - deltaPoids;

}
```



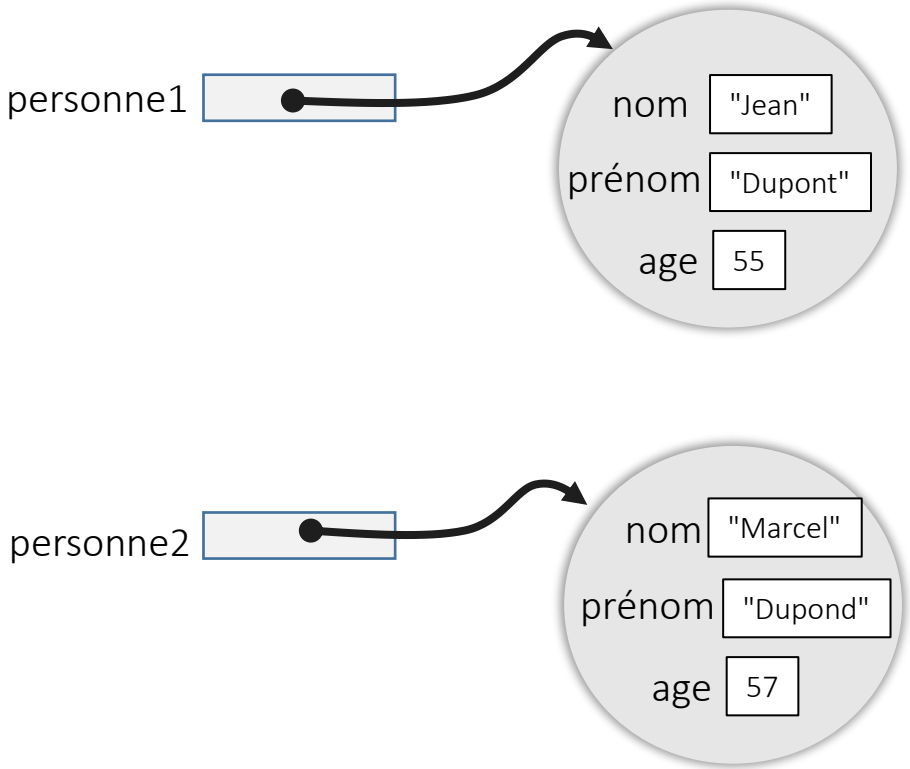

```
let felicite = { nom: "Félicité", age:10, poids: 5.0, …};
maigrir(felicite, 0.2);
console.log("Félicité pèse  " + felicite.poids + "  kg");
```

```
Félicité pèse  4.8  kg
```

Modification des objets `const`

- un objet déclaré avec `const` peut être modifié

```
1  const personne1 = {
2      prénom: "Jean",
3      nom: "Dupont",
4      age : 55,
5  };
6
7  let personne2 = {
8      prénom: "Marcel",
9      nom : "Dupond",
10     age : 57
11 };
12
13 personne1.age = 56;
14
15 console.log(personne1.age);
16
17 // personne1 = personne2;
18
19 personne2 = personne1;
```

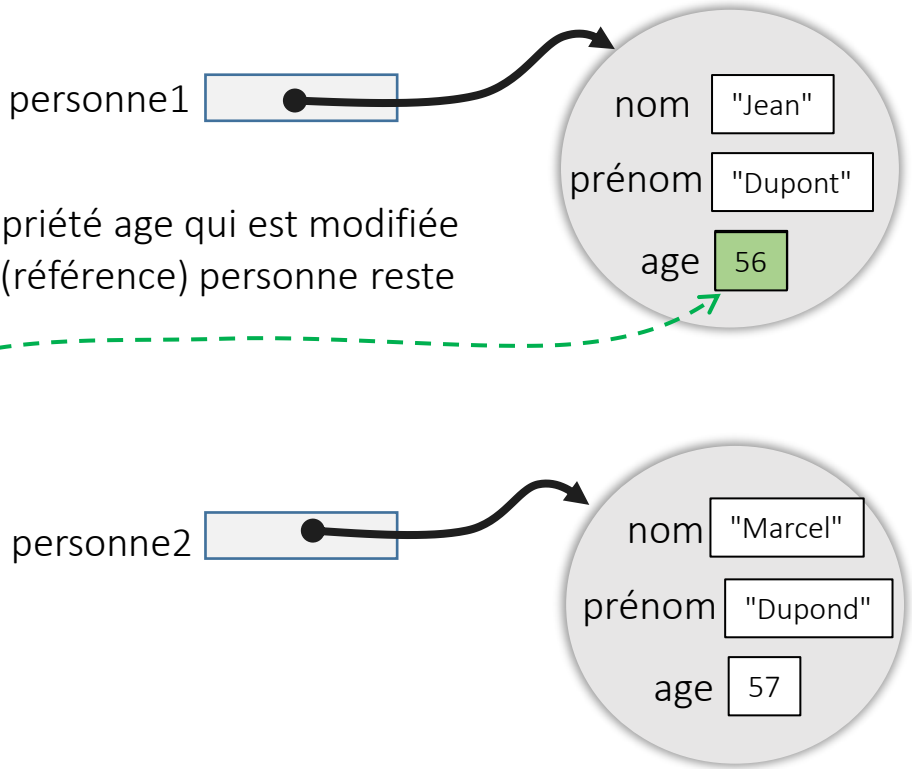


Modification des objets `const`

- un objet déclaré avec `const` peut être modifié

```
1  const personne1 = {
2      prénom: "Jean",
3      nom: "Dupont",
4      age : 55,
5  };
6
7  let personne2 = {
8      prénom: "Marcel",
9      nom : "Dupond",
10     age : 57
11 };
12
13 personne1.age = 56;
14
15 console.log(personne1.age);
16
17 // personne1 = personne2;
18
19 personne2 = personne1;
```

56



Modification des objets `const`

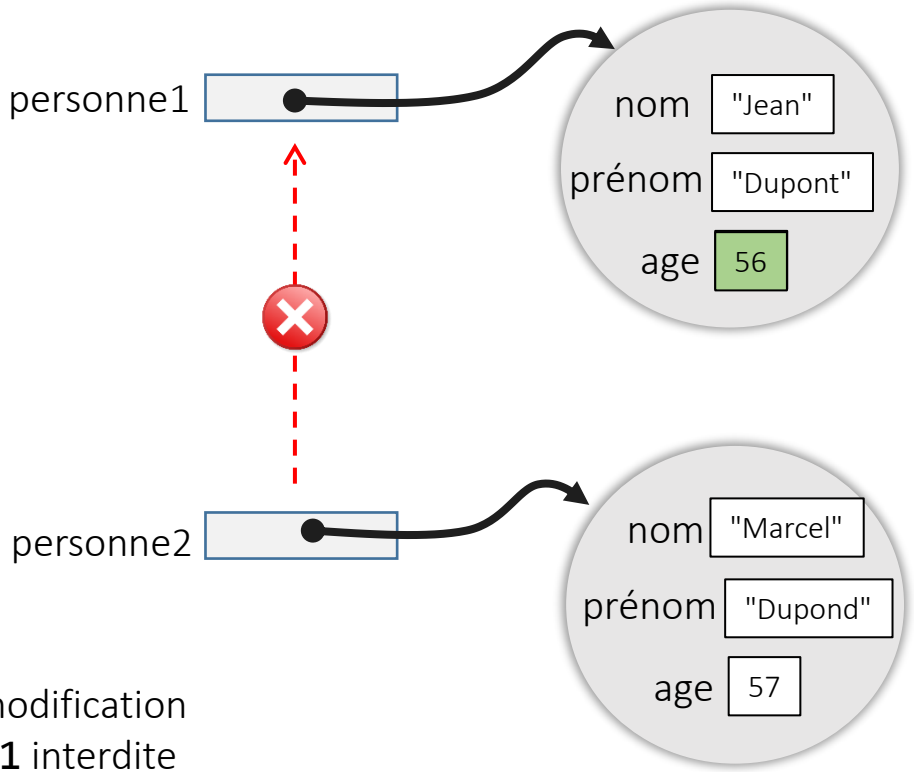
- un objet déclaré avec `const` peut être modifié

```
1  const personne1 = {
2      prénom: "Jean",
3      nom: "Dupont",
4      age : 55,
5  };
6
7  let personne2 = {
8      prénom: "Marcel",
9      nom : "Dupond",
10     age : 57
11 };
12
13 personne1.age = 56;
14
15 console.log(personne1.age);
16
17 // personne1 = personne2;
18
19 personne2 = personne1;
```

56

tentative de modification de **`personne1`** interdite

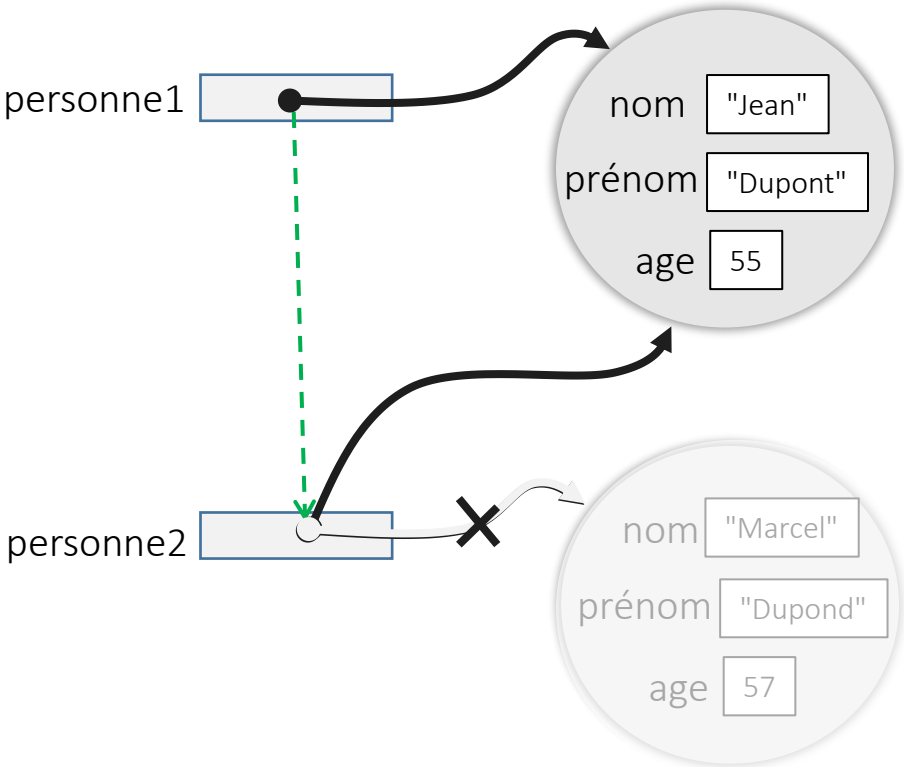
```
personne1 = personne2;
          ^

TypeError: Assignment to constant variable.
```



Modification des objets `const`

- un objet déclaré avec `const` peut être modifié

```
1  const personne1 = {
2      prénom: "Jean",
3      nom: "Dupont",
4      age : 55,
5  };
6
7  let personne2 = {
8      prénom: "Marcel",
9      nom : "Dupond",
10     age : 57
11 };
12
13 personne1.age = 56;
14
15 console.log(personne1.age);
16
17 // personne1 = personne2;
18
19 personne2 = personne1;
```



recopie de la valeur de la variable **personne1** dans la variable **personne2**

Si il n'y a pas d'autres références sur l'objet il ne pourra plus être accédé. La mémoire allouée sera libérée automatiquement par l'interpréteur JavaScript (*garbage collection*)

voir MDN

Méthodes

- Les objets ne sont pas qu'un regroupement de valeurs, les propriétés peuvent être aussi des fonctions (les objets peuvent aussi avoir un comportement).

```
let felicite = {
   nom: "Félicité",
   age : 10,
   race : "chat de gouttière",
   aime : [
            "manger du thon",
            "grimper aux arbres",
            "dormir"
           ],
   poids: 3.5 ,
   miauler: function () {
          console.log("Miaou ! Miaou !");
   }
};
```

une fonction anonyme est affectée à une propriété de l'objet

ce type de fonction est généralement appelé **méthode** de l'objet

- Invocation d'une méthode

```
felicite.miauler();
```

envoi du **message** `miauler` à l'objet référencé par `felicite`

Le mot clé `this`

- comment accéder aux attributs d'un objet dans une méthode ?

```
let felicite = {
    nom: "Félicité",
    age : 10,
    race : "chat de goutière",
    aime : [
              "manger du thon",
              "grimper aux arbres",
              "dormir"
            ],
    poids: 3.5,
    miauler: function () {
        console.log("Miaou ! Miaou !");
    },
     maigrir: function (deltaPoids) {

        this.poids = this.poids - deltaPoids;
     }

};
```

```
function maigrir(unChat, deltaPoids) {

   unChat.poids = unChat.poids - deltaPoids;

}
```

```
maigrir(felicite, 0.2);
```

transformer cette fonction en une méthode de l'objet.

pour désigner l'une des propriétés de l'objet, le mot clé **`this`** ***doit*** *être utilisé*

appel de la méthode

```
felicite.maigrir(0.2);
```

Dans une méthode **`this`** *désigne l'objet qui reçoit le message.*

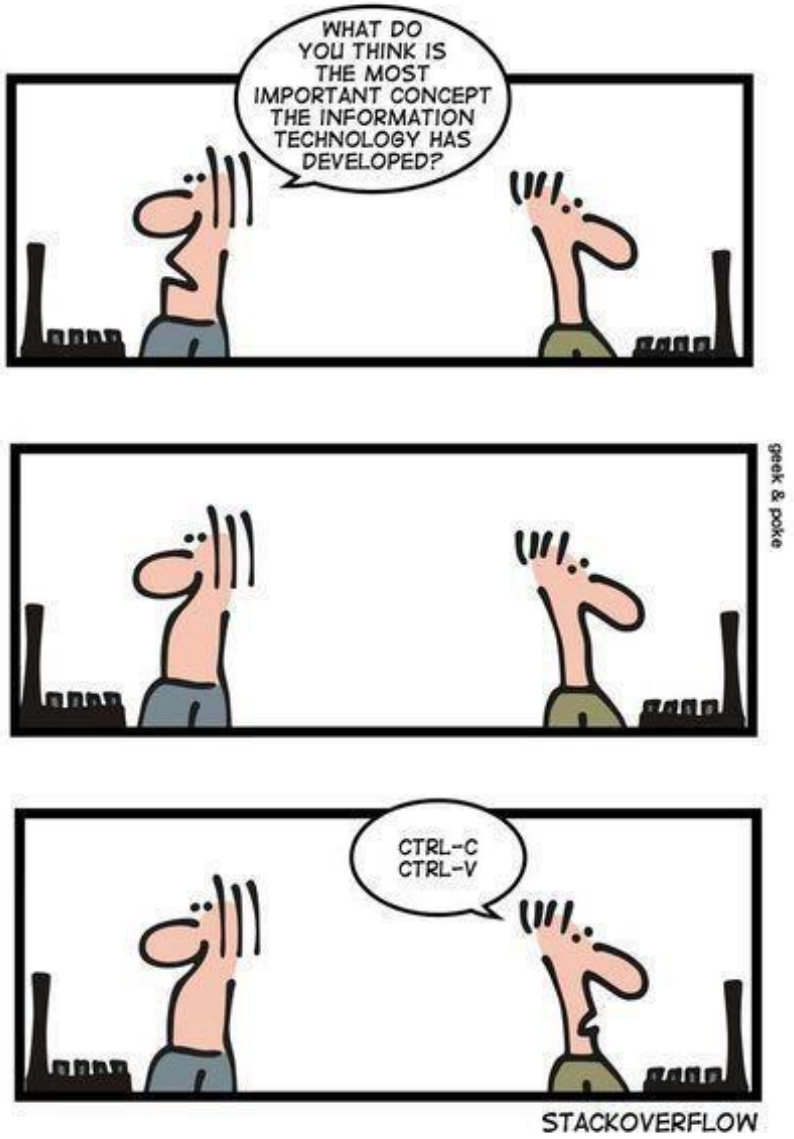
Créer plusieurs objets du même type ?

J'aimerais bien avoir un deuxième chat. Mais comment faire ?

```
let felicite = {
   nom: "Félicité",
   age : 10,
   race : "chat de gouttière"
   aime : [
            "manger du thon",
            "grimper aux arbres"
            "dormir"
          ],
   poids: 3.5,

   miauler: function () {
     console.log("Miaou ! Miaou !");
   },
   maigrir : function(deltaPoids) {
      this.poids -= deltaPoids;
   }
};
```

```
let felix = {
   nom: "Félix",
   age : 6,
   race : "siamois"
   aime : [
            "se lécher",
            "manger des croquettes",
            "dormir"
          ],
   poids: 3.,

    miauler: function () {
      console.log("Miaou ! Miaou !");
    },
    maigrir : function(deltaPoids) {
       this.poids -= deltaPoids;
    }
};
```

Créer plusieurs objets du même type ?



Dupliquer du code est souvent (toujours ?) une mauvaise idée

- Source potentielle d'erreurs
- Difficulté de mises à jour
- Taille du code
- Lisibilité

Constructeur

Utiliser une fonction de création: **constructeur**

une fonction comme une autre par convention débute par Majuscule

Souvent les paramètres définissent les valeurs des propriétés que l'on souhaite initialiser à la création de l'objet

```
var felicite = {
   nom: "Félicité",
   age : 10,
   race : "chat de gouttière"
   aime : [
            "manger du thon",
            "grimper aux arbres"
            "dormir"
           ],
   poids: 3.5,
   miauler: function () {
        console.log("Miaou ! Miaou !");
   },
   maigrir : function(deltaPoids) {
      this.poids -= deltaPoids;
   }
};
```

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
   this.miauler = function () {
      console.log("Miaou ! Miaou !");
   };
   this.maigrir = function(deltaPoids) {
      this.poids -= deltaPoids;
   };
}
```

initialisation des propriétés de l'objet avec les valeurs des paramètres

Définition des méthodes

un constructeur est invoqué par l'opérateur **new**

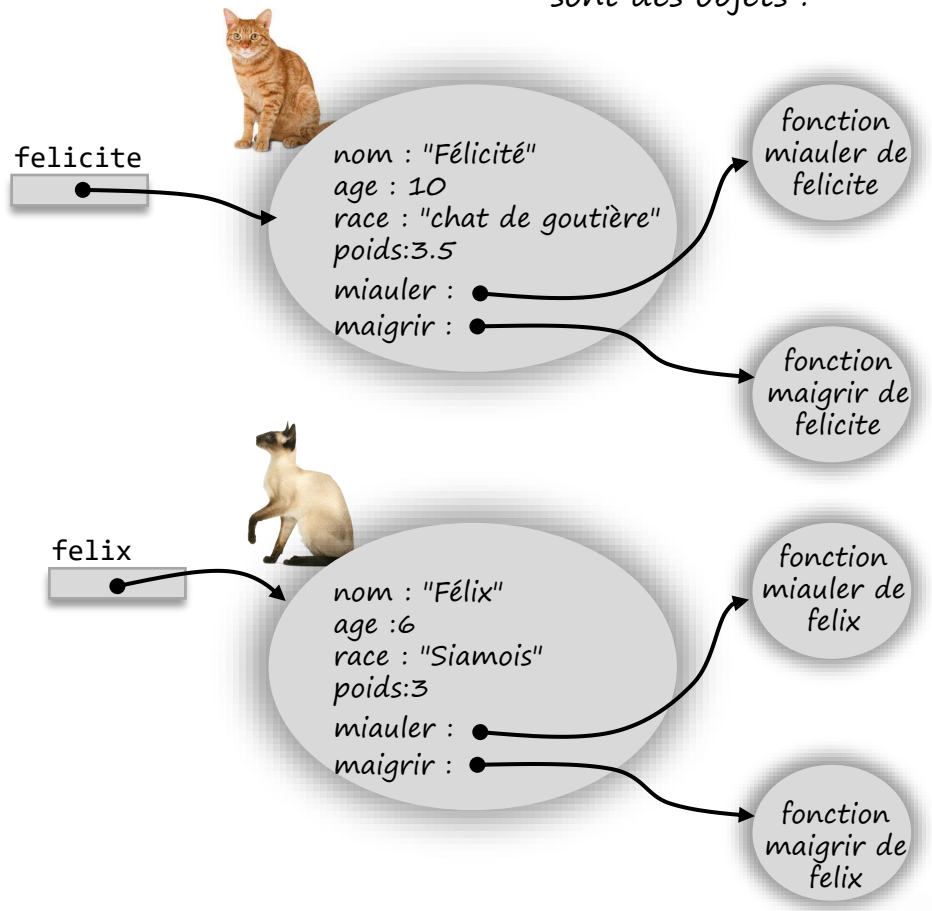
```
let felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
let felix    = new Chat("Felix",6,"siamois",3);
```

les objets référencés par `felicite` *et* `felix` *sont des '**instances**' de* **Chat**.

Constructeur et méthodes d'un objet

En JavaScript les fonctions sont des objets !

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
   this.miauler = function () {
     console.log(this.nom + "-> Miaou ! Miaou !");
   };
   this.maigrir = function(deltaPoids) {
      this.poids -= deltaPoids;
   };
}

let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
let felix    = new  Chat("Felix",6,"siamois",3);
felicite.miauler();   // Felicité -> Miaou ! Miaou !
felix.miauler();      // Felix -> Miaou ! Miaou !
console.log("felicite.miauler === felix.miauler --> " +
             (felicite.miauler === felix.miauler));   // false
```



`felicite.miauler` *et* `felix.miauler` *référencent deux objets fonctions différents (même si ils font la même chose)*

Constructeur et méthodes d'un objet : prototype

- Les fonctions sont des objets
- Elles ont une propriété `prototype` :
  - liste de propriétés attachée à un constructeur (initialement vide)
  - une propriété rajoutée sur le prototype du constructeur devient disponible sur tous les objets créés à l'aide de ce constructeur (*fallback*)

Utiliser `prototype` du constructeur pour partager les méthodes

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};
Chat.prototype.maigrir = function(deltaPoids) {
      this.poids -= deltaPoids;
};
let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
let felix    = new  Chat("Felix",6,"siamois",3);
felicite.miauler();   ➡  Felicité -> Miaou ! Miaou !
felix.miauler();      ➡  Felix -> Miaou ! Miaou !
console.log("felicite.miauler === felix.miauler --> " +
              (felicite.miauler === felix.miauler));   ➡  true
```

Prototype pour définir des propriétés

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};

Chat.prototype.maigrir = function(deltaPoids) {
     this.poids -= deltaPoids;
};

Chat.prototype.famille = "Felidés";
```

les propriétés rajoutées sur le prototype d'un constructeur ne se limitent pas à des fonctions méthodes elles peuvent être de n'importe quel type.

Une propriété définie sur le prototype d'un constructeur est accessible pour tous les objets (instances) créés via ce constructeur

```
let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
let felix    = new  Chat("Felix",6,"siamois",3);
console.log(felicite),
felicite.miauler();
console.log(felicite.famille);
console.log(felix.famille);
```

```
Chat {
  nom: 'Félicité',
  age: 10,
  race: 'chat de gouttière',
  poids: 3.5
}
```

Felicité -> Miaou ! Miaou !

Félidés

Félidés

Redéfinition d'une méthode

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};

Chat.prototype.maigrir = function(deltaPoids) {
      this.poids -= deltaPoids;
};

let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
let felix    = new  Chat("Felix",6,"siamois",3);

felicite.miauler();
felix.miauler();
```

Felicité -> Miaou ! Miaou !

Felix -> Miaou ! Miaou !

```
felix.miauler = function () {
    console.log(this.nom + "-> Meaow ! Meaow !");
};
```

redéfinition de la méthode miauler pour felix

```
felix.miauler();
felicite.miauler();
```

Felix -> Meaow ! Meaow !

Felicité -> Miaou ! Miaou !

Une propriété du prototype peut être redéfinie sur une instance.

Dans ce cas la redéfinition ne concerne que l'instance

Modification du prototype du constructeur

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};

Chat.prototype.maigrir = function(deltaPoids) {
     this.poids -= deltaPoids;
};

let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
felicite.miauler();
```

Félicité -> Miaou ! Miaou !

```
Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Meaow ! Meaow !");
};
```

modification de la méthode miauler du prototype du constructeur

```
felicite.miauler();
```

Félicité -> Meaow ! Meaow !

```
let felix  = new  Chat("Felix",6,"siamois",3);
felix.miauler();
```

Felix -> Meaow ! Meaow !

Une modification du prototype est immédiate pour les instances déjà existantes. Le *fallback* se fait à l'exécution (*runtime*) au moment de l'accès à la propriété.

Modification du prototype du constructeur

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids ;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};

Chat.prototype.maigrir = function(deltaPoids) {
     this.poids -= deltaPoids;
};

let felicite = new  Chat("Félicité",10,"chat de gouttière", 3.5);
felicite.miauler();
felicite.ronronner();
```

Félicité -> Miaou ! Miaou !

Ce message provoquerait une erreur la méthode n'est pas définie

```
Chat.prototype.ronronner = function () {
    console.log(this.nom + "-> Rrr... Rrr...");
};
```

ajout de la méthode ronronner au prototype du constructeur

```
felicite.ronronner();
```

Félicité -> Rrr... Rrr...

De la même manière qu'une modification, un ajout au prototype est immédiatement actif et s'applique à toutes les instances déjà créées

Objets et chaînes de <prototypes> en JavaScript

Philippe Genoud
Philippe.Genoud@univ-grenoble-alpes.fr

<prototype> d'un objet

protosObjetsConstructeur1.js

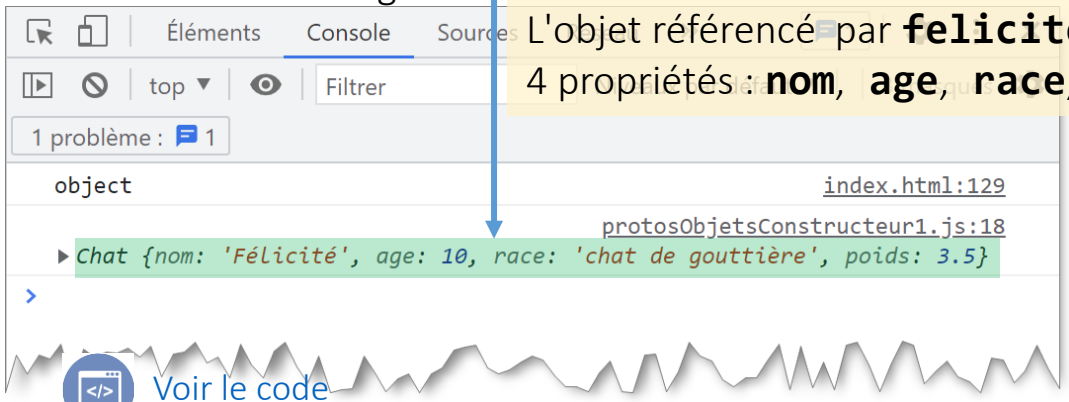
```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
17
18 console.log(felicite);
19
```

définition du constructeur d'objets **Chat**

méthodes associées aux objets **Chat**

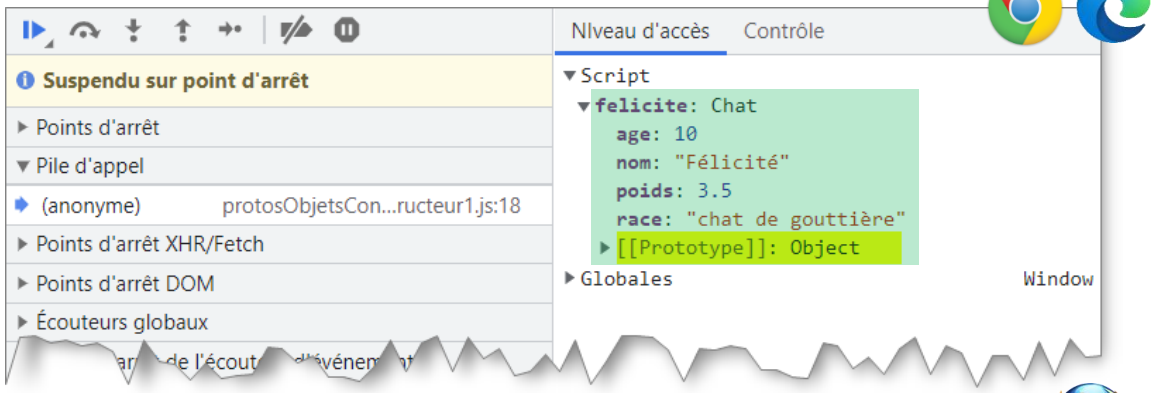
création d'un objet **Chat** (instance)

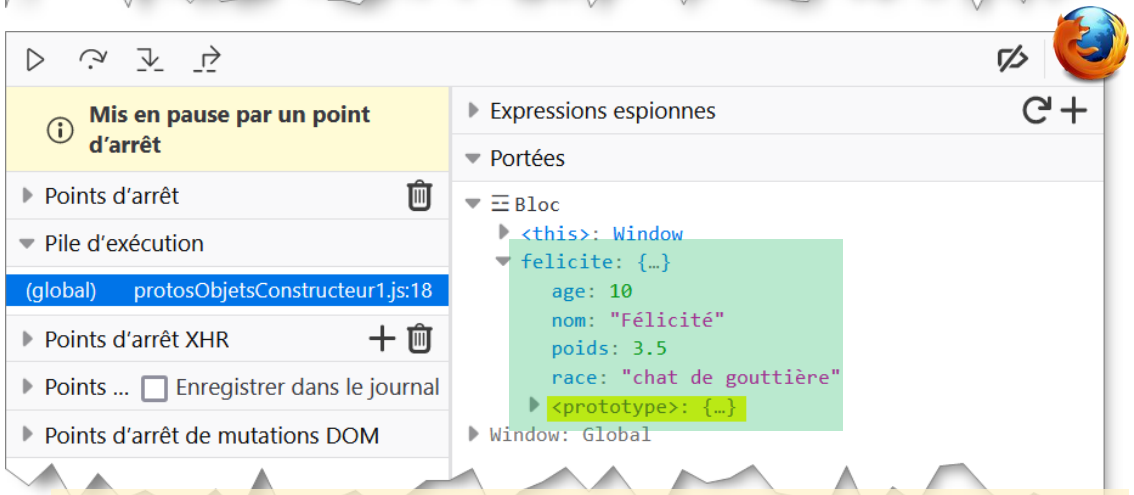
Sur la console du navigateur



L'objet référencé par **felicite** défini avec 4 propriétés : **nom**, **age**, **race**, **poids**

Dans le débogueur du navigateur





En interne l'objet référencé par **felicite** possède une propriété supplémentaire que l'on peut observer à l'aide du débogueur : **[[Prototype]]** (Chrome ou Edge) ou **<prototype>** (Firefox)

L'attribut interne **<prototype>** ≠ attribut **prototype** d'une fonction constructeur

Voir le code

Chaîne de <prototypes>

- Chaque objet possède un lien interne (masqué) nommé son `<prototype>` (Firefox) ou `[[Prototype]]` (Chrome, Edge, Node) qui le relie soit à un autre objet soit à `null`.

- Ce lien est unique : un objet possède un et un seul lien `<prototype>`
- Ces liens constituent ce que l'on nomme la **chaîne de <prototypes>** de l'objet

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {…}
```

`<prototype>` de l'objet **felicite**

```
▸ constructor:Chat()
▸ maigrir:Chat.prototype.maigrir()
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▾ Bloc
  ▸ <this>: Window
  ▸ felicite: {…}
▸ Window: Global
```

chaîne de **<prototypes>** de l'objet référencé par **felicite**

dernier objet de la chaîne son `<prototype>` est **null**

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

- Bloc
  - <this>: Window
  - felicite: {…}
- Window: Global

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▸ constructor:Chat()
▸ maigrir:Chat.prototype.maigrir()
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

lors de l'accès à une propriété (attribut) ou lors de l'exécution d'une méthode d'un objet, l'interpréteur JavaScript, s'il ne trouve pas la propriété ou la méthode dans l'objet lui-même, effectue une recherche de celle-ci en parcourant la chaîne de <prototypes>.

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {…}
```

Prototype de l'objet **felicite**

```
▸ constructor:Chat()
▸ maigrir:Chat.prototype.maigrir()
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

Prototype du prototype de l'objet **felicite**

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▾ Bloc
  ▸ <this>: Window
  ▸ felicite: {…}
▸ Window: Global
```

```
console.log(felecite.nom);
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
```

Voir le code

Chaîne de prototypes

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

- Bloc
  - <this>: Window
  - felicite: {…}
- Window: Global

L'objet **felicite**

- felicite: {…}
  - age: 10
  - nom: "Félicité"
  - poids: 3.5
  - race: "chat de gouttière"
  - <prototype>: {…}

<prototype> de l'objet **felicite**

- constructor:Chat()
- maigrir:Chat.prototype.maigrir()
- miauler:Chat.prototype.miauler()
- <prototype>: {…}

<prototype> du <prototype> de l'objet **felicite**

- __defineGetter__()
- __defineSetter__()
- __lookupGetter__()
- __lookupSetter__()
- __proto__: »
- constructor:Object()
- hasOwnProperty()
- isPrototypeOf()
- propertyIsEnumerable()
- toLocaleString()
- toSource()
- toString()
- valueOf()

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
```

Voir le code

Chaîne de <prototypes> : fallback

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
```

Voir le code

Chaîne de <prototypes> : fallback

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

- Bloc
  - <this>: Window
  - felicite: {…}
- Window: Global

L'objet **felicite**

- felicite: {…}
  - age: 10
  - nom: "Félicité"
  - poids: 3.5
  - race: "chat de gouttière"
  - <prototype>: {…}

<prototype> de l'objet **felicite**

- constructor:Chat()
- maigrir:Chat.prototype.maigrir()
- miauler:Chat.prototype.miauler()
- <prototype>: {…}

<prototype> du <prototype> de l'objet **felicite**

- __defineGetter__()
- __defineSetter__()
- __lookupGetter__()
- __lookupSetter__()
- __proto__: »
- constructor:Object()
- hasOwnProperty()
- isPrototypeOf()
- propertyIsEnumerable()
- toLocaleString()
- toSource()
- toString()
- valueOf()

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
felicite.grossir(1.0);
```

Voir le code

Chaîne de <prototypes> : fallback

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés et une forme d'héritage

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
felicite.grossir(1.0);
```

Voir le code

Chaîne de <prototypes> : *fallback*

- Les chaînes de <prototypes> mécanisme utilisé par JavaScript pour implémenter le partage de propriétés entre objet issus d'un même constructeur et une forme d'héritage (mécanisme de *fallback*)

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

<prototype> de l'objet **felicite**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

protosObjetsConstructeur1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"
17
18 console.log(felicite);
19
20
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
▶ Window: Global
```

```
console.log(felicite.nom);
felicite.miauler();
console.log(felicite.toString());
felicite.grossir(1.0);
```

```
▶ TypeError: felicite.grossir is not a function
```

Voir le code

<prototype> et constructeur

- Les objets créés à partir d'un même constructeur partagent le même <prototype>

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(deltaPoids) {
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Portées

- Bloc
  - <this>: Window
  - felicite: {…}
  - felix: {…}
- Window: Global

L'objet **felicite**

- felicite: {…}
  - age: 10
  - nom: "Félicité"
  - poids: 3.5
  - race: "chat de gouttière"
  - <prototype>: {…}

<prototype> de l'objet **felicite**

- constructor:Chat()
- maigrir:Chat.prototype.maigrir()
- miauler:Chat.prototype.miauler()
- <prototype>: {…}

<prototype> du <prototype> de l'objet **felicite**

- __defineGetter__()
- __defineSetter__()
- __lookupGetter__()
- __lookupSetter__()
- __proto__: »
- constructor:Object()
- hasOwnProperty()
- isPrototypeOf()
- propertyIsEnumerable()
- toLocaleString()
- toSource()
- toString()
- valueOf()

Voir le code

<prototype> et constructeur

- Les objets créés à partir d'un même constructeur partagent le même <prototype>

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

L'objet **felicite**

```
felicite: {_}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {_}
```

```
felix: {_}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▶ <prototype>: {_}
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {_}
  ▶ felix: {_}
▶ Window: Global
```

<prototype> de l'objet **felicite** et de l'objet **felix**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {_}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

Voir le code

<prototype> et constructeur

- Les objets créés à partir d'un même constructeur partagent le même <prototype>

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Portées

```
Bloc
  <this>: Window
  felicite: {…}
  felix: {…}
Window: Global
```

Voir le code

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  <prototype>: {…}
```

```
felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  <prototype>: {…}
```

```
constructor:Chat()
    arguments: null
    caller: null
    length: 4
    name: "Chat"
  prototype: {…}
  <prototype>()
```

La fonction constructeur **Chat** est aussi un objet

<prototype> de l'objet **felicite** et de l'objet **felix**

```
constructor:Chat()
maigrir:Chat.prototype.maigrir()
miauler:Chat.prototype.miauler()
<prototype>: {…}
```

Le **<prototype>** des objets créés avec le constructeur **Chat** est le **prototype** de l'objet constructeur **Chat**

<prototype> du constructeur est la propriété **prototype** de la fonction qui a servi à créer l'objet constructeur Chat()

<prototype> du <prototype> de l'objet **felicite**

```
__defineGetter__()
__defineSetter__()
__lookupGetter__()
__lookupSetter__()
__proto__: »
constructor:Object()
hasOwnProperty()
isPrototypeOf()
propertyIsEnumerable()
toLocaleString()
toSource()
toString()
valueOf()
```

<prototype> et constructeur

- Les objets créés à partir d'un même constructeur partagent le même <prototype>

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Portées

```
Bloc
  <this>: Window
  felicite: {…}
  felix: {…}
Window: Global
```

Voir le code

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  <prototype>: {…}
```

```
felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  <prototype>: {…}
```

<prototype> de l'objet **felicite** et de l'objet **felix**

```
constructor:Chat()
maigrir:Chat.prototype.maigrir()
miauler:Chat.prototype.miauler()
<prototype>: {…}
```

```
constructor:Chat()
    arguments: null
    caller: null
    length: 4
    name: "Chat"
  prototype: {…}
  <prototype>()
```

```
apply()
arguments: »
bind()
call()
caller: »
constructor:Function()
length: 0
name: ""
toSource()
toString()
Symbol(Symbol.hasInstance):[Symb
<get arguments()>: arguments()
<set arguments()>: arguments()
<get caller()>: caller()
<set caller()>: caller()
<prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
<prototype>: {…}
  __defineGetter__()
  __defineSetter__()
  __lookupGetter__()
  __lookupSetter__()
  __proto__: »
  constructor:Object()
  hasOwnProperty()
  isPrototypeOf()
  propertyIsEnumerable()
  toLocaleString()
  toSource()
  toString()
  valueOf()
```

Comme tout objet le constructeur **Chat**[1] possède une **chaîne de <prototypes>**

Toutes les chaînes de <prototypes> partagent le même objet terminal

[1] **Chat** est une fonction et en JavaScript les fonctions sont des objets

<prototype> vs. Constructeur.prototype

Ne pas confondre

1. la propriété (attribut) **prototype** d'un objet constructeur
2. le lien prototype (**<prototype>**) de cet objet constructeur

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Voir le code

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

```
felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▶ <prototype>: {…}
```

Portées

```
▼ Bloc
  ▶ <this>: Window
  ▶ felicite: {…}
  ▶ felix: {…}
▶ Window: Global
```

<prototype> de l'objet **felicite** et de l'objet **felix**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

```
▼ constructor:Chat()
    arguments: null
    caller: null
    length: 4
    name: "Chat"
  ▶ prototype: {…}
  ▶ <prototype>()
```

```
▶ apply()
  arguments: »
▶ bind()
▶ call()
  caller: »
▶ constructor:Function()
  length: 0
  name: ""
▶ toSource()
▶ toString()
▶ Symbol(Symbol.hasInstance):[Symb
▶ <get arguments()>: arguments()
▶ <set arguments()>: arguments()
▶ <get caller()>: caller()
▶ <set caller()>: caller()
▶ <prototype>: {…}
```

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

<prototype> vs. Constructeur.prototype

Ne pas confondre

1. la propriété (attribut) **prototype** d'un objet constructeur
2. le lien prototype (**<prototype>**) de cet objet constructeur

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Voir le code

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▶ <prototype>: {…}
```

```
felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▶ <prototype>: {…}
```

Portées

```
▼ Bloc
    ▶ <this>: Window
    ▶ felicite: {…}
    ▶ felix: {…}
▶ Window: Global
```

```
▼ constructor:Chat()
      arguments: null
      caller: null
      length: 4
      name: "Chat"
  ▶ prototype: {…}
  ▶ <prototype>()
```

1 2

<prototype> de l'objet **felicite** et de l'objet **felix**

```
▶ constructor:Chat()
▶ maigrir:Chat.prototype.maigrir()
▶ miauler:Chat.prototype.miauler()
▶ <prototype>: {…}
```

```
▶ apply()
  arguments: »
▶ bind()
▶ call()
  caller: »
▶ constructor:Function()
  length: 0
  name: ""
▶ toSource()
▶ toString()
▶ Symbol(Symbol.hasInstance):[Symb
▶ <get arguments()>: arguments()
▶ <set arguments()>: arguments()
▶ <get caller()>: caller()
▶ <set caller()>: caller()
▶ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▶ __defineGetter__()
▶ __defineSetter__()
▶ __lookupGetter__()
▶ __lookupSetter__()
  __proto__: »
▶ constructor:Object()
▶ hasOwnProperty()
▶ isPrototypeOf()
▶ propertyIsEnumerable()
▶ toLocaleString()
▶ toSource()
▶ toString()
▶ valueOf()
```

1 le **prototype** de l'objet constructeur **Chat** est le **<prototype>** des objets créés avec le constructeur **Chat**

<prototype> vs. Constructeur.prototype

Ne pas confondre

1. la propriété (attribut) **prototype** d'un objet constructeur
2. le lien prototype (**<prototype>**) de cet objet constructeur

protosObjetsConstructeur1_1.js

```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félic
17 let felix    = new Chat("Felix
18
19 console.log(felicite);
20
```

Portées

- Bloc
  - <this>: Window
  - felicite: {…}
  - felix: {…}
- Window: Global

Voir le code

L'objet **felicite**

```
felicite: {…}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {…}
```

```
felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▸ <prototype>: {…}
```

```
▾ constructor:Chat()
    arguments: null
    caller: null
    length: 4
    name: "Chat"
  ▸ prototype: {…}
  ▸ <prototype>()
```

<prototype> de l'objet **felicite** et de l'objet **felix**

```
▸ constructor:Chat()
▸ maigrir:Chat.prototype.maigrir()
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

```
▸ apply()
  arguments: »
▸ bind()
▸ call()
  caller: »
▸ constructor:Function()
  length: 0
  name: ""
▸ toSource()
▸ toString()
▸ Symbol(Symbol.hasInstance):[Symb
▸ <get arguments()>: arguments()
▸ <set arguments()>: arguments()
▸ <get caller()>: caller()
▸ <set caller()>: caller()
▸ <prototype>: {…}
```

<prototype> du <prototype> de l'objet **felicite**

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

1. le **prototype** de l'objet constructeur **Chat** est le **<prototype>** des objets créés avec le constructeur **Chat**
2. **<prototype>** du constructeur est la propriété **prototype** de la fonction qui a servi à créer l'objet constructeur **Chat()**

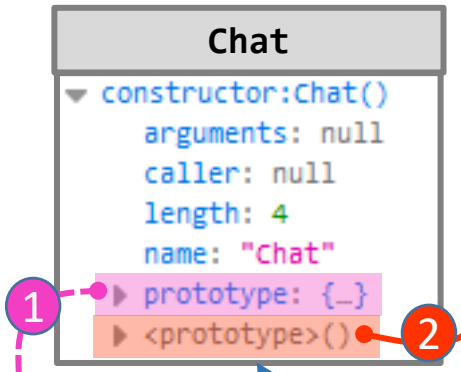
Built-in objects

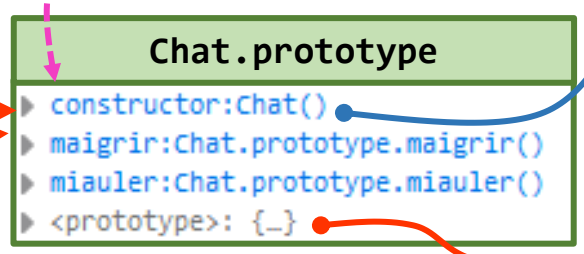
Object et **Function** sont des objets (fonctions constructeur) prédéfinis (*built-in*)

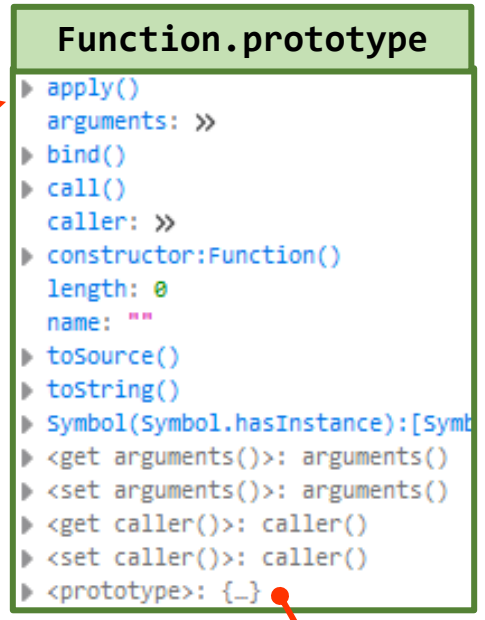
protosObjetsConstructeur1_1.js

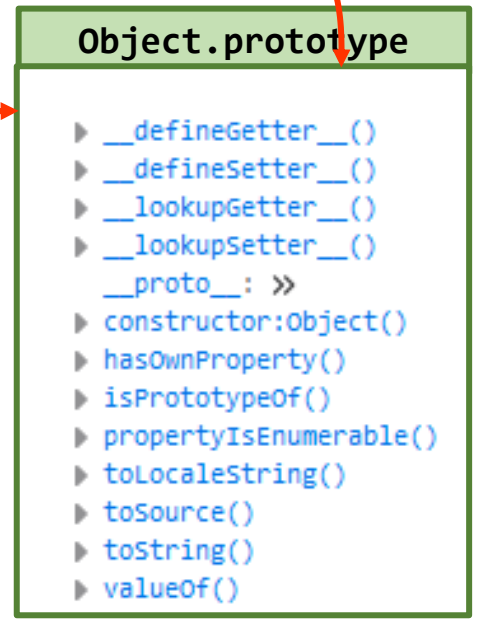
```
1  function Chat(nom,age,race,poids) {
2     this.nom = nom ;
3     this.age = age ;
4     this.race = race ;
5     this.poids = poids ;
6  }
7
8  Chat.prototype.miauler = function () {
9        console.log("Miaou ! Miaou !");
10    };
11
12 Chat.prototype.maigrir  = function(del
13       this.poids -= deltaPoids;
14    };
15
16 let felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
17 let felix    = new Chat("Felix",6,"siamois",3);
18
19 console.log(felicite);
20
```

Voir le code







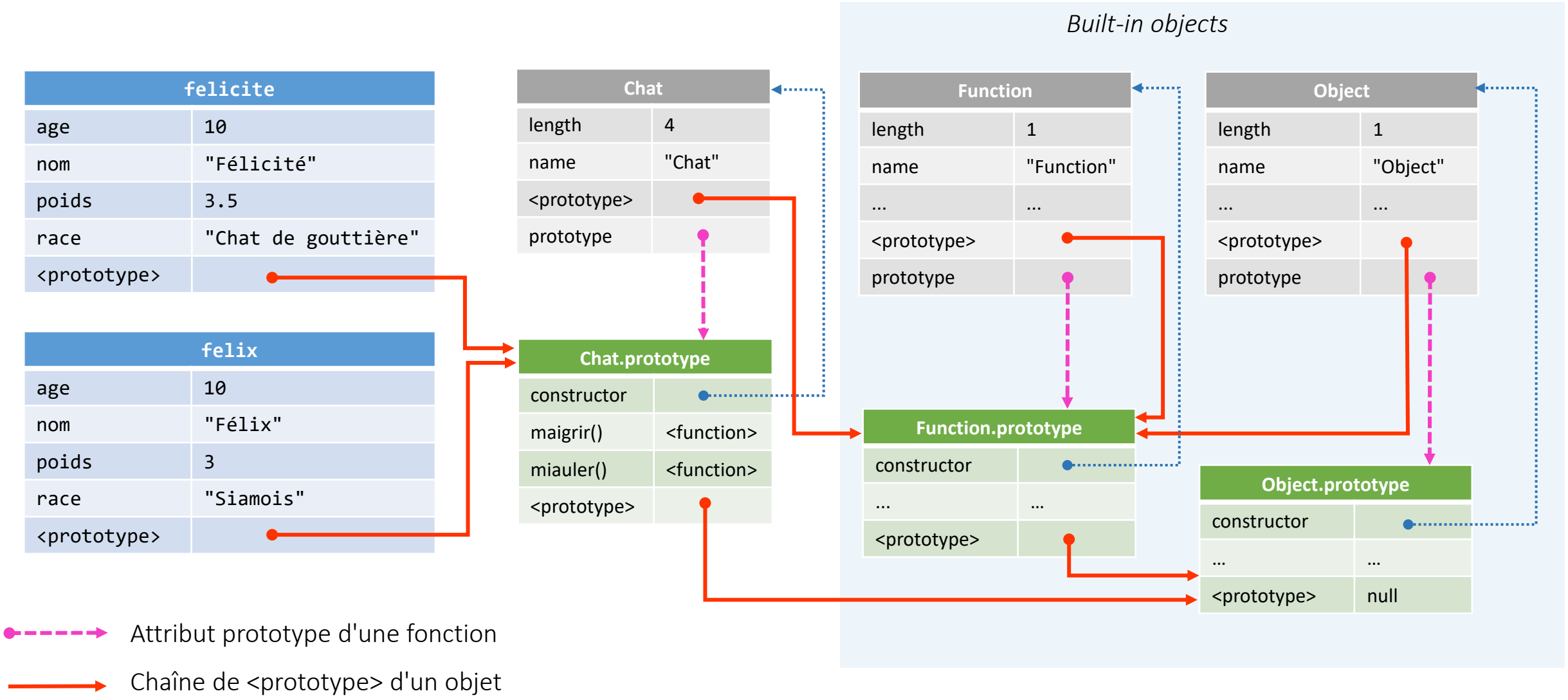


Function.prototype est le <prototype> commun à toutes les fonctions constructeur

Object.prototype est l'objet terminal de toutes les chaînes de <prototypes>

Built-in objects

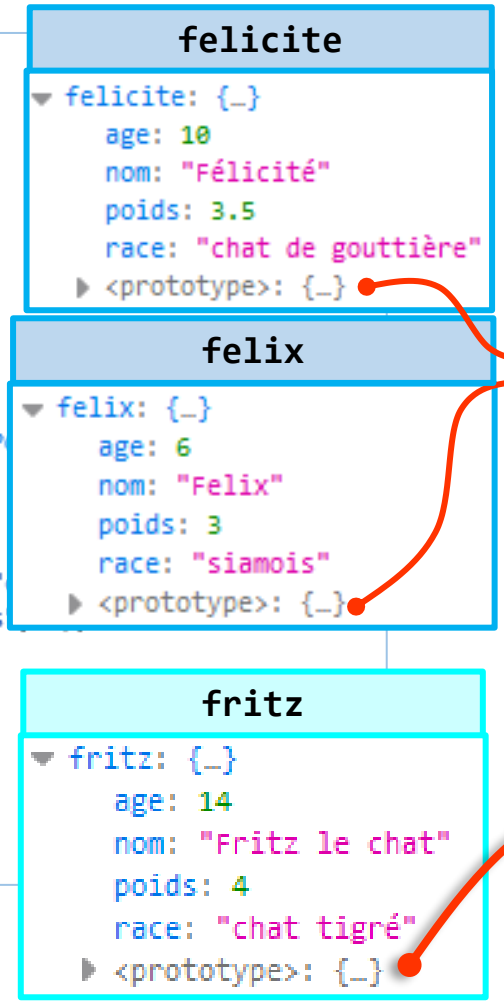
- Les fonctions `Function()` et `Object()` et leur chaîne de `<prototypes>`

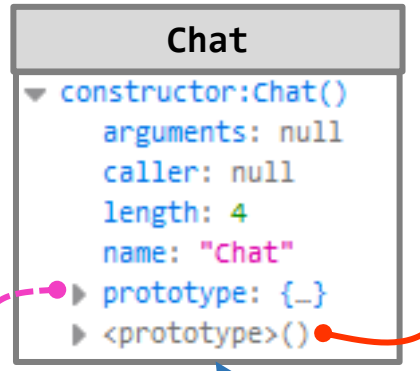


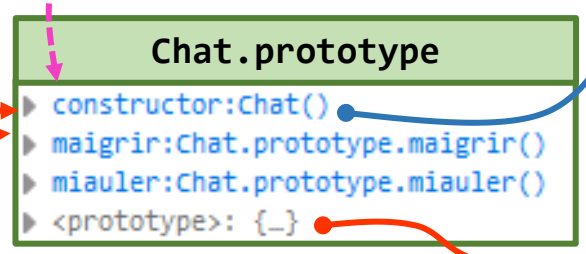
<prototype> d'un objet Littéral

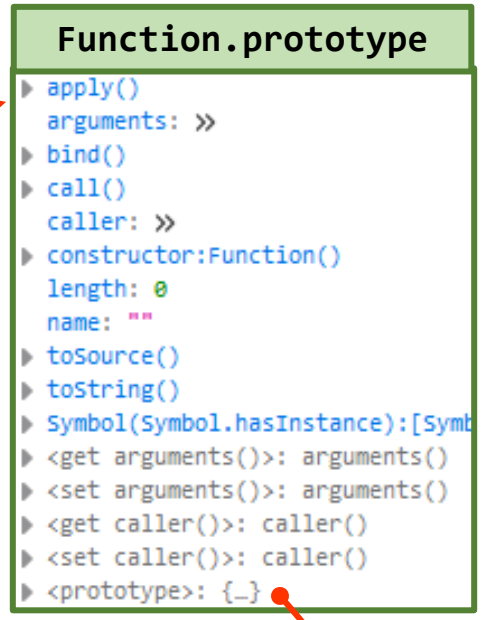
```
1  function Chat(nom, age, race, poids) {
2      this.nom = nom;
3      this.age = age;
4      this.race = race;
5      this.poids = poids;
6  }
7
8  Chat.prototype.miauler = function () {
9      console.log("Miaou ! Miaou !");
10 };
11
12 Chat.prototype.maigrir = function (deltaP
13     this.poids -= deltaPoids;
14 };
15
16 let felicite = new Chat("Félicité", 10, "
17 let felix = new Chat("Felix", 6, "siamois
18
19 let fritz = {
20     nom: "Fritz le chat",
21     age: 14,
22     race: "chat tigré",
23     poids: 4.0
24 };
```

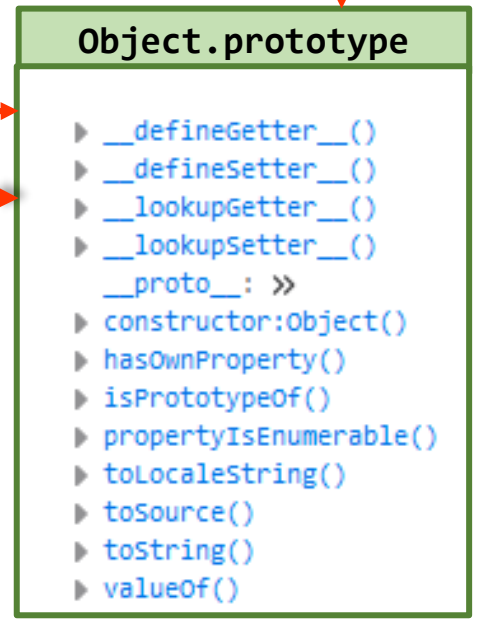
Voir le code











- Le <prototype> d'un objet littéral est `Object.prototype`

Fallback de propriétés

- Une propriété définie au niveau du **`prototype`** d'un constructeur est 'héritée' par tous les objets ayant ce **`<prototype>`**

```
1  function Chat(nom, age, race, poids) {
2      this.nom = nom;
3      this.age = age;
4      this.race = race;
5      this.poids = poids;
6  }
7
8  Chat.prototype.cri = "Miaou Miaou";
9
10 Chat.prototype.miauler = function () {
11     console.log(this.nom + " : " + this.cri);
12 };
13
14 Chat.prototype.maigrir = function (deltaPoids) {
15     this.poids -= deltaPoids;
16 };
17
18 function afficherCris() {
19     console.log("cri de félicité " + felicite.cri);
20     console.log("cri de félix " + felix.cri);
21     felicite.miauler();
22     felix.miauler();
23 }
24
25 let felicite = new Chat("Félicité", 10, "chat de gouttière", 3.5);
26 let felix = new Chat("Felix", 6, "siamois", 3);
27 afficherCris();
```

felicite

```
felicite: {_}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {_}
```

felix

```
felix: {_}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▸ <prototype>: {_}
```

Chat.prototype

```
▸ constructor:Chat(nom, age, race, poids)
  cri: "Miaou Miaou"
▸ maigrir:Chat.prototype.maigrir(deltaPoids)
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {_}
```

Object.prototype

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

```
cri de félicité Miaou Miaou
cri de félix Miaou Miaou
Félicité : Miaou Miaou
Felix : Miaou Miaou
```

Voir le code

Fallback de propriétés

- Modification d'une propriété du **`prototype`** d'un constructeur s'applique à toutes les instances

```
1  function Chat(nom, age, race, poids) {
2      this.nom = nom;
3      this.age = age;
4      this.race = race;
5      this.poids = poids;
6  }
7
8  Chat.prototype.cri = "Miaou Miaou";
9
10 Chat.prototype.miauler = function () {
11     console.log(this.nom + " : " + this.cri);
12 };
13
14 Chat.prototype.maigrir = function (deltaPoids) {
15     this.poids -= deltaPoids;
16 };
17
18 function afficherCris() {
19     console.log("cri de félicité " + felicite.cri);
20     console.log("cri de félix " + felix.cri);
21     felicite.miauler();
22     felix.miauler();
23 }
24
25 let felicite = new Chat("Félicité", 10, "chat de gouttière", 3.5);
26 let felix = new Chat("Felix", 6, "siamois", 3);
27 afficherCris();
```

felicite

```
felicite: {_}
    age: 10
    nom: "Félicité"
    poids: 3.5
    race: "chat de gouttière"
  ▸ <prototype>: {_}
```

felix

```
felix: {_}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▸ <prototype>: {_}
```

Chat.prototype

```
▸ constructor:Chat(nom, age, race, poids)
  cri: "Meow !  Meow !"
▸ maigrir:Chat.prototype.maigrir(deltaPoids)
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {_}
```

Object.prototype

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

```
cri de félicité Miaou Miaou
cri de félix Miaou Miaou
Félicité : Miaou Miaou
Felix : Miaou Miaou
```

```
28
29 Chat.prototype.cri = "Meow !  Meow !";
30 afficherCris();
31
32 felicite.cri = "Miaou !  Miaou !";
33 afficherCris();
34
35 delete felicite.cri;
36 afficherCris();
37
38 delete felix.cri;
39 afficherCris();
```

```
cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !
```

Voir le code

Fallback de propriétés

- L'affectation d'une propriété héritée en passant par une instance modifie uniquement l'objet instance (ajout de la propriété si elle n'existe pas, modification si elle existe déjà)

```
function Chat(nom, age, race, poids) {
    this.nom = nom;
    this.age = age;
    this.race = race;
    this.poids = poids;
}

Chat.prototype.cri = "Miaou Miaou";

Chat.prototype.miauler = function () {
    console.log(this.nom + " : " + this.cri);
};

Chat.prototype.maigrir = function (deltaPoids) {
    this.poids -= deltaPoids;
};

function afficherCris() {
    console.log("cri de félicité " + felicite.cri);
    console.log("cri de félix " + felix.cri);
    felicite.miauler();
    felix.miauler();
}

let felicite = new Chat("Félicité", 10, "chat de gouttière", 3.5);
let felix = new Chat("Felix", 6, "siamois", 3);
afficherCris();
```

```
cri de félicité Miaou Miaou
cri de félix Miaou Miaou
Félicité : Miaou Miaou
Felix : Miaou Miaou
```

felicite

```
felicite: {…}
  age: 10
  cri: "Miaou !  Miaou !"
  nom: "Félicité"
  poids: 3.5
  race: "chat de gouttière"
▸ <prototype>: {…}
```

felix

```
▾ felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▸ <prototype>: {…}
```

Chat.prototype

```
▸ constructor:Chat(nom, age, race, poids)
  cri: "Meow !  Meow !"
▸ maigrir:Chat.prototype.maigrir(deltaPoids)
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

Object.prototype

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

la propriété **`cri`** est redéfinie au niveau de l'instance **`felicite`**

```
28
29 Chat.prototype.cri = "Meow !  Meow !";
30 afficherCris();
31
32 felicite.cri = "Miaou !  Miaou !";
33 afficherCris();
34
35 delete felicite.cri;
36 afficherCris();
37
38 delete felix.cri;
39 afficherCris();
```

```
cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !
```

```
cri de félicité Miaou !  Miaou !
cri de félix Meow !  Meow !
Félicité : Miaou !  Miaou !
Felix : Meow !  Meow !
```

La valeur de la propriété **`cri`** est différente
`felicite` : propriété propre (*own property*)
`felix` : propriété héritée (*fallback*)

Voir le code

Fallback de propriétés

- La suppression d'une propriété en passant par une instance modifie uniquement l'objet instance (retrait de la propriété)

```
function Chat(nom, age, race, poids) {
    this.nom = nom;
    this.age = age;
    this.race = race;
    this.poids = poids;
}

Chat.prototype.cri = "Miaou Miaou";

Chat.prototype.miauler = function () {
    console.log(this.nom + " : " + this.cri);
};

Chat.prototype.maigrir = function (deltaPoids) {
    this.poids -= deltaPoids;
};

function afficherCris() {
    console.log("cri de félicité " + felicite.cri);
    console.log("cri de félix " + felix.cri);
    felicite.miauler();
    felix.miauler();
}

let felicite = new Chat("Félicité", 10, "chat de gouttière", 3.5);
let felix = new Chat("Felix", 6, "siamois", 3);
afficherCris();
```

```
cri de félicité Miaou Miaou
cri de félix Miaou Miaou
Félicité : Miaou Miaou
Felix : Miaou Miaou
```

felicite

```
felicite: {…}
  age: 10
  cri: "Miaou ! Miaou !"
  nom: "Félicité"
  poids: 3.5
  race: "chat de gouttière"
▸ <prototype>: {…}
```

felix

```
▾ felix: {…}
    age: 6
    nom: "Felix"
    poids: 3
    race: "siamois"
  ▸ <prototype>: {…}
```

Chat.prototype

```
▸ constructor:Chat(nom, age, race, poids)
  cri: "Miaou Miaou"
▸ maigrir:Chat.prototype.maigrir(deltaPoids)
▸ miauler:Chat.prototype.miauler()
▸ <prototype>: {…}
```

Object.prototype

```
▸ __defineGetter__()
▸ __defineSetter__()
▸ __lookupGetter__()
▸ __lookupSetter__()
  __proto__: »
▸ constructor:Object()
▸ hasOwnProperty()
▸ isPrototypeOf()
▸ propertyIsEnumerable()
▸ toLocaleString()
▸ toSource()
▸ toString()
▸ valueOf()
```

La suppression de la propriété en passant par une instance modifie uniquement l'objet instance (retrait de la propriété)

```
29 Chat.prototype.cri = "Meow !  Meow !";
30 afficherCris();
31
32 felicite.cri = "Miaou !  Miaou !";
33 afficherCris();
34
35 delete felicite.cri;
36 afficherCris();
37
38 delete felix.cri;
39 afficherCris();
```

```
cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !
```

```
cri de félicité Miaou !  Miaou !
cri de félix Meow !  Meow !
Félicité : Miaou !  Miaou !
Felix : Meow !  Meow !
```

```
cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !
```

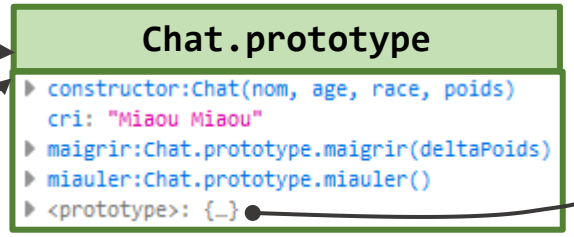
La valeur de la propriété **`cri`** pour **`felicite`** est à nouveau la propriété héritée (*fallback*)

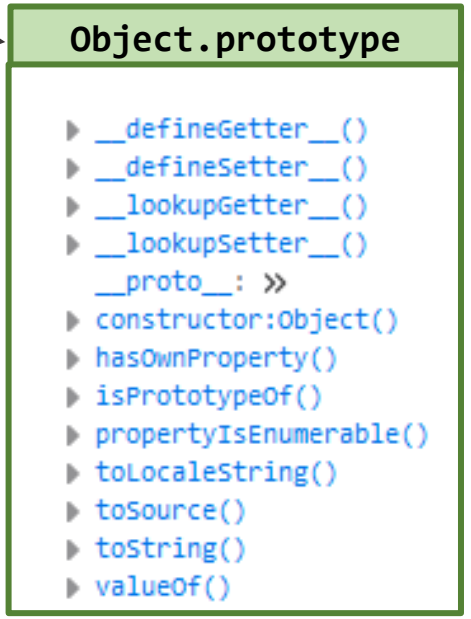
Voir le code

Chaîne de prototypes

- La suppression d'une propriété est sans effet si l'instance n'a pas directement cette propriété (*own property*)

```
1  function Chat(nom, age, race, poids) {
2      this.nom = nom;
3      this.age = age;
4      this.race = race;
5      this.poids = poids;
6  }
7
8  Chat.prototype.cri = "Miaou Miaou";
9
10 Chat.prototype.miauler = function () {
11     console.log(this.nom + " : " + this.cri);
12 };
13
14 Chat.prototype.maigrir = function (deltaPoids) {
15     this.poids -= deltaPoids;
16 };
17
18 function afficherCris() {
19     console.log("cri de félicité " + felicite.cri);
20     console.log("cri de félix " + felix.cri);
21     felicite.miauler();
22     felix.miauler();
23 }
24
25 let felicite = new Chat("Félicité", 10, "chat de gouttière", 3.5);
26 let felix = new Chat("Felix", 6, "siamois", 3);
27 afficherCris();
```






```
cri de félicité Miaou Miaou
cri de félix Miaou Miaou
Félicité : Miaou Miaou
Felix : Miaou Miaou
```

```
28
29 Chat.prototype.cri = "Meow !  Meow !";
30 afficherCris();
31
32 felicite.cri = "Miaou !  Miaou !";
33 afficherCris();
34
35 delete felicite.cri;
36 afficherCris();
37
38 delete felix.cri;
39 afficherCris();
```

```
cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !

cri de félicité Miaou !  Miaou !
cri de félix Meow !  Meow !
Félicité : Miaou !  Miaou !
Felix : Meow !  Meow !

cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !

cri de félicité Meow !  Meow !
cri de félix Meow !  Meow !
Félicité : Meow !  Meow !
Felix : Meow !  Meow !
```

Voir le code

Objets et ES5/ES6

- La création d'objets avec une fonction constructeur et l'opérateur `new` est historique, elle masque le mécanisme de prototypage souvent mal compris par les programmeurs.
- ES5 puis ES6 proposent de nouvelles manières de définir des objets (même si en interne le principe du prototypage est inchangé) afin de rendre l'utilisation des objets plus accessible aux développeurs:
  - `Object.create()` (ES5)
  - Classes (ES6)

Objets et ES5 : Object

- Depuis la version ES5 de JavaScript, l'objet prédéfini (*built-in*) `Object` propose un certain nombre de méthodes pour créer et manipuler les prototypes :

  - `Object.create(proto:Object) : Object`
    crée un nouvel objet ayant pour <prototype> l'objet `proto` passé en paramètre

  - `Object.getPrototypeOf(obj : Object) : Object`
    renvoie l'objet <prototype> de l'objet `obj` passé en paramètre

  - `Object.setPrototypeOf(obj: Object, proto: Object)`
    fixe le <prototype> de `obj` avec l'objet `proto`

Objets et ES6 : Classes

- Avec l'introduction de **`Class`**, ES6 offre une syntaxe compacte pour définir des chaînes de prototypes et qui se rapproche de l'approche plus couramment utilisée dans les langages orientés objets (langages de classes comme Java)

avant ES6

```
function Chat(nom,age,race,poids) {
   this.nom = nom ;
   this.age = age ;
   this.race = race ;
   this.poids = poids;
}

Chat.prototype.miauler = function () {
    console.log(this.nom + "-> Miaou ! Miaou !");
};
Chat.prototype.maigrir = function(deltaPoids) {
      this.poids -= deltaPoids;
};
```

constructeur

⇔

méthodes

ES6 +

```
class Chat {
   constructor(nom,age,race,poids) {
      this.nom = nom ;
      this.age = age ;
      this.race = race ;
      this.poids = poids;
   }
   miauler() {
        console.log(this.nom + "-> Miaou ! Miaou !");
   }
   maigrir(deltaPoids) {
       this.poids -= deltaPoids;
   }
}
```

mot réservé pour identifier la fonction constructeur

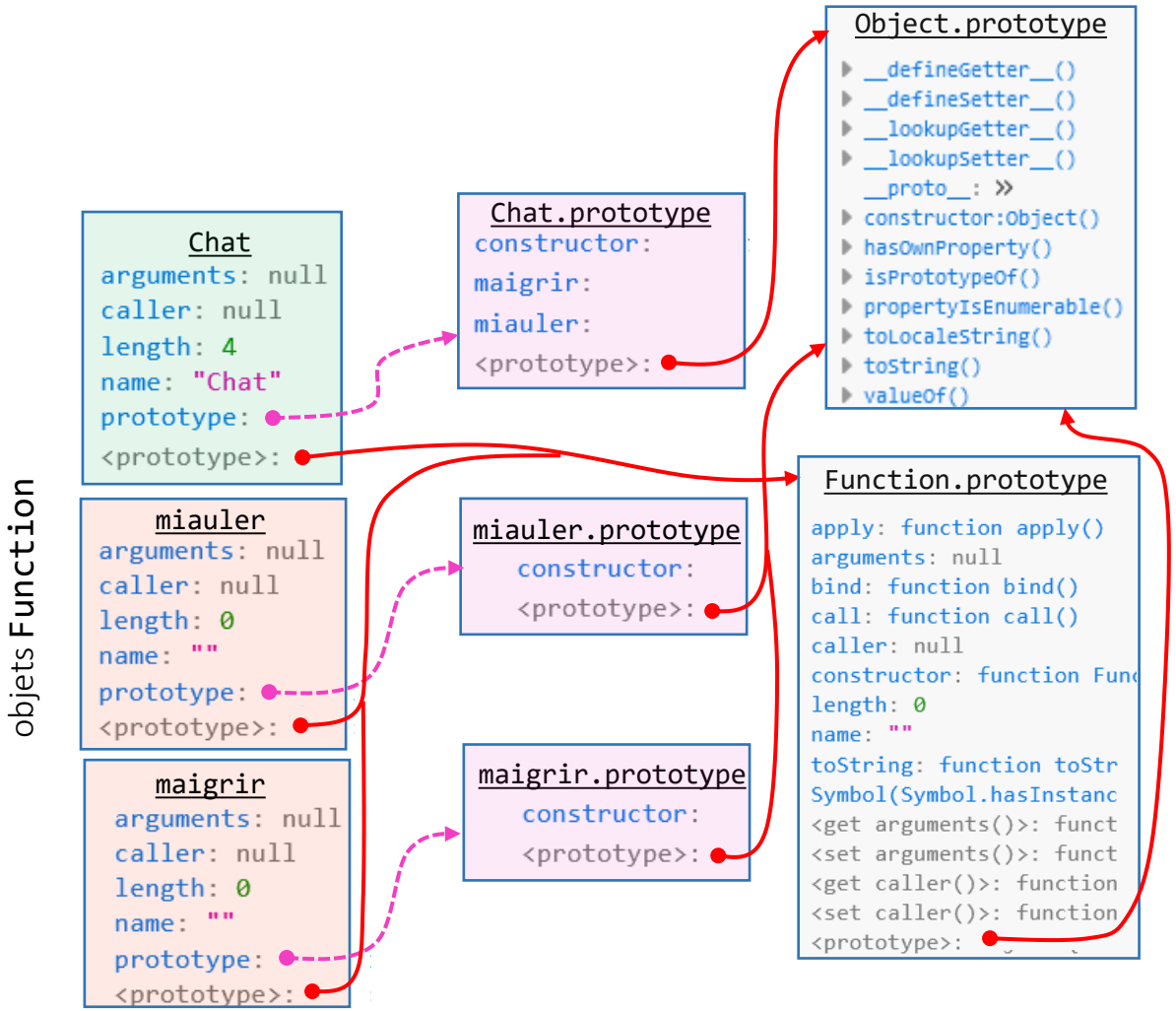
Une classe ne peut avoir qu'un seul constructeur

pas de **;** entre les déclarations de méthodes

Voir le code

Objets et ES6 : Classes

- Les objets créés par une déclaration de classe sont des objets fonction correspondant respectivement au constructeur et aux méthodes de la classe ainsi que les **`prototypes`** associés



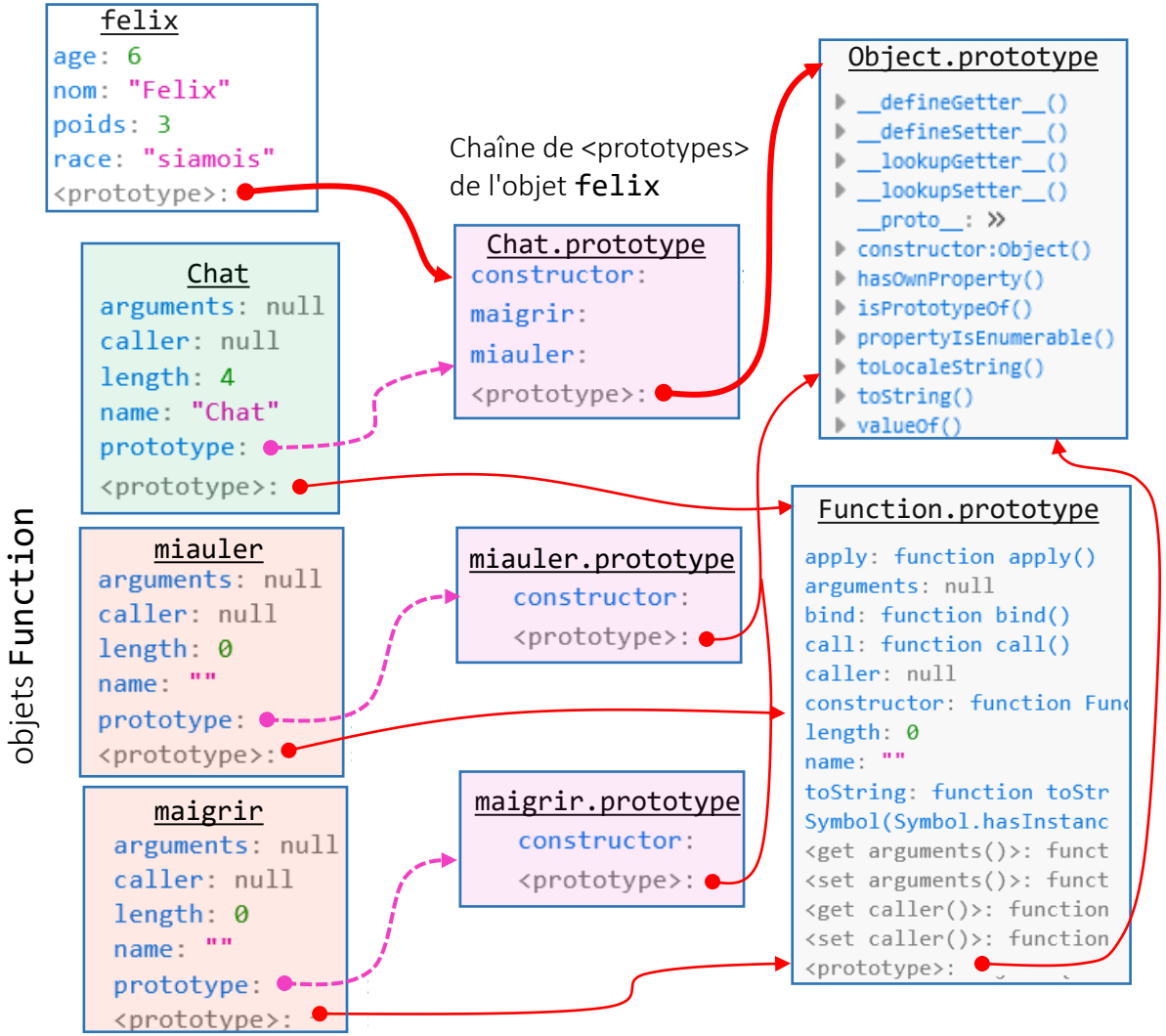

```
class Chat {
    constructor(nom,age,race,poids) {
        this.nom = nom ;
        this.age = age ;
        this.race = race ;
        this.poids = poids;
    }
    miauler() {
         console.log(this.nom + "-> Miaou ! Miaou !");
    }
    maigrir(deltaPoids) {
        this.poids -= deltaPoids;
    }
}
```

Voir le code

Objets et ES6 : Classes

- La création des objets se fait comme précédemment avec l'opérateur **new**

```
let felix = new Chat("Felix",6,"siamois",3);
```




```
class Chat {
    constructor(nom,age,race,poids) {
        this.nom = nom ;
        this.age = age ;
        this.race = race ;
        this.poids = poids;
    }
    miauler() {
        console.log(this.nom + "-> Miaou ! Miaou !");
    }
    maigrir(deltaPoids) {
        this.poids -= deltaPoids;
    }
}
```

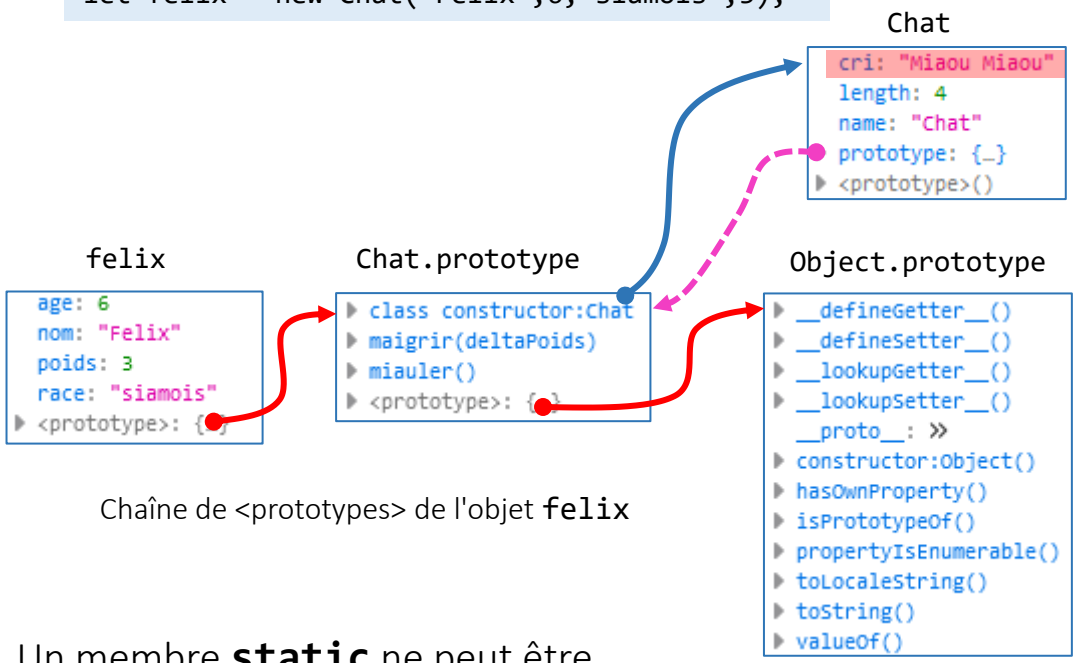
appelée immédiatement après la création d'une nouvelle instance pour l'initialiser. Cette fonction reçoit les arguments passés après le nom de la classe suivant l'opérateur **new**.

Voir le code

Objets et ES6 : Classes – membres statiques

- Propriétés **statiques**
  - Possibilité d'associer des **propriétés** à la classe directement
  - déclaration de variable préfixée par le mot clé **`static`**
  - Rattaché à la fonction Constructeur et non pas au `prototype`

```
let felix = new Chat("Felix",6,"siamois",3);
```



Chaîne de <prototypes> de l'objet `felix`

```
class Chat {
  constructor(nom,age,race,poids) {
     this.nom = nom ;
     this.age = age ;
     this.race = race ;
     this.poids = poids;
  }
  static cri = "Miaou ! Miaou !";
   miauler() {
       console.log(this.nom + "-> " + Chat.cri);
   }
   maigrir(deltaPoids) {
      this.poids -= deltaPoids;
   }
}
```

Voir le code

- Un membre **`static`** ne peut être accédé via une instance de la classe (pas dans la chaîne de <prototypes>)

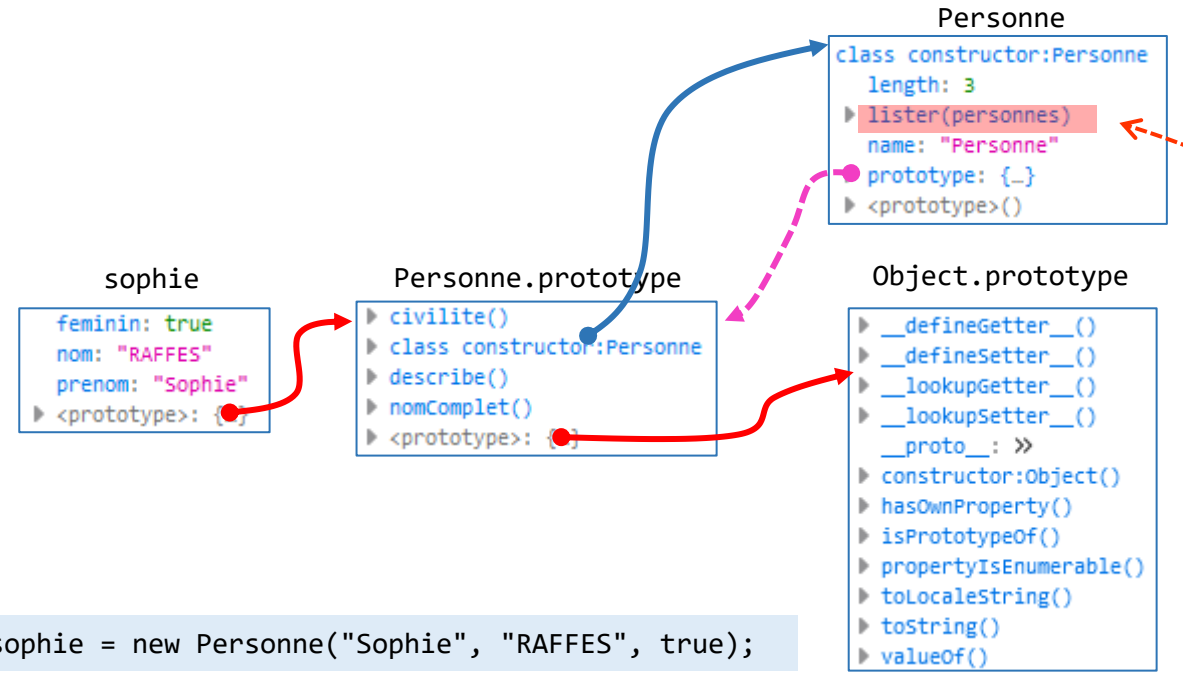
`felix.cri → undefined`

- Un membre **`static`** est accédé via l'identifiant de la classe

`Chat.cri → "Miaou ! Miaou !`

Objets et ES6 : Classes – membres statiques

- Méthodes **statiques**
  - déclaration de méthode variable préfixée par le mot clé **`static`**
  - fonction rattaché à la fonction Constructeur et non pas au `prototype`




```
let sophie = new Personne("Sophie", "RAFFES", true);
```

```
let marcel = new Personne("Marcel", "TROUSP", false);

Personne.lister([sophie, marcel]);
```

→

```
Personne
--------
Mme RAFFES Sophie
M. TROUSP Marcel
```

Pour invoquer la méthode statique il faut passer par l'objet classe (la fonction constructeur)

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin; // true si une femme, false sinon
    }

    nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite()} ${this.nomComplet()}`;
    }

    /**
     * affiche la description des personnes contenues dans un tableau
     * @param {Array[Personne]} personnes le tableau des personnes
     */
    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

Voir le code

Méthode statique se situe non pas au niveau du prototype pour les Personnes mais au niveau de la fonction constructeur `Personne`

Objets et ES6 : Classes - *getters* et *setters*

- Accesseurs : fonctions *getters*
  - **get** permet de lier une propriété d'un objet à une fonction qui sera appelée lorsqu'on accédera à la propriété.

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin; // true si une femme, false sinon
    }

    nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite()} ${this.nomComplet()}`;
    }

    /**
     * affiche la description des personnes contenues dans un tableau
     * @param {Array[Personne]} personnes le tableau des personnes
     */
    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

Méthodes transformées en getters

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    /**
     * affiche la description des personnes contenues dans un tableau
     * @param {Array[Personne]} personnes le tableau des personnes
     */
    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

```
let sophie = new Personne("Sophie", "RAFFES", true);
console.log(sophie.civilite()); → 'Mme'
console.log(sophie.nomComplet()); → 'RAFFES Sophie'
```

Voir le code

```
console.log(sophie.civilite);   → 'Mme'
console.log(sophie.nomComplet); → 'RAFFES Sophie'
```

Les getters peuvent être utilisés comme des propriétés

Objets et ES6 : Classes - *getters* et *setters*

- Modifieurs (fonctions *setters*)
  - **set** permet de lier une propriété d'un objet à une fonction qui sera appelée lorsqu'on affectera la propriété.

```
let p1 = new Personne("Sophie", "RAFFES", true);

console.log(p1.nomComplet);     → 'RAFFES Sophie'

p1.nomComplet = "  Maeva   MERLIN";

console.log(p1.prenom);         → 'Maeva'

console.log(p1.nom);            → 'MERLIN'
```

Appel du *setter*

On peut aussi définir getters et setters dans des objets littéraux

```
let date =  {
    jour : '17',
    mois : 'Novembre',
    annee : '2020',
    get dateComplete() {
        return `${this.jour} ${this.mois} ${this.annee}`;
    },
    set dateComplete(dateString) {
        const tokens = dateString.split(/\b\s+(?!$)/);
        this.jour = parseInt(tokens[0]);
        this.mois = tokens[1];
        this.annee = parseInt(tokens[2]);
    }
};
```

Voir le code

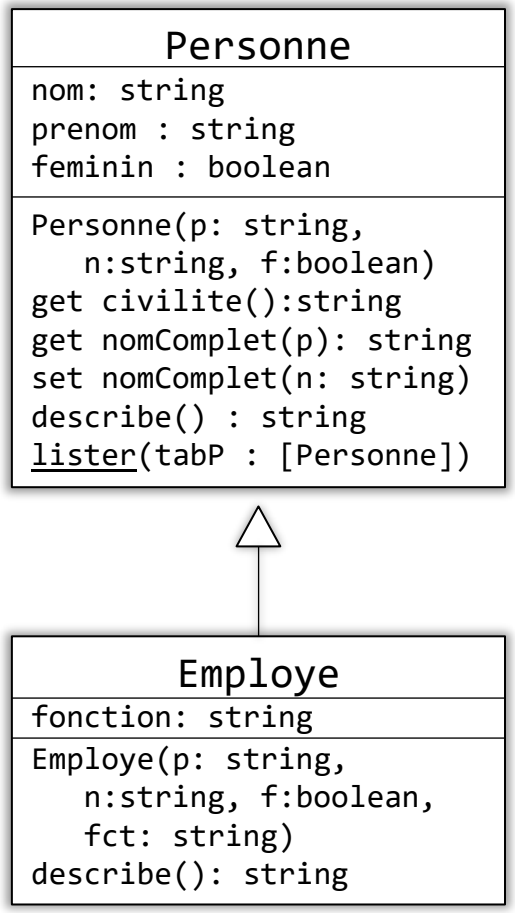
```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);  // expression régulière*
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    /**
     * affiche la description des personnes contenues dans un tableau
     * @param {Array[Personne]} personnes le tableau des personnes
     */
    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

Voir le code

*https://blog.abelotech.com/posts/split-string-into-tokens-javascript/

Objets et ES6 : Classes - Héritage

- Héritage – possibilité de définir une classe comme étendant une classe existante (sous classe)



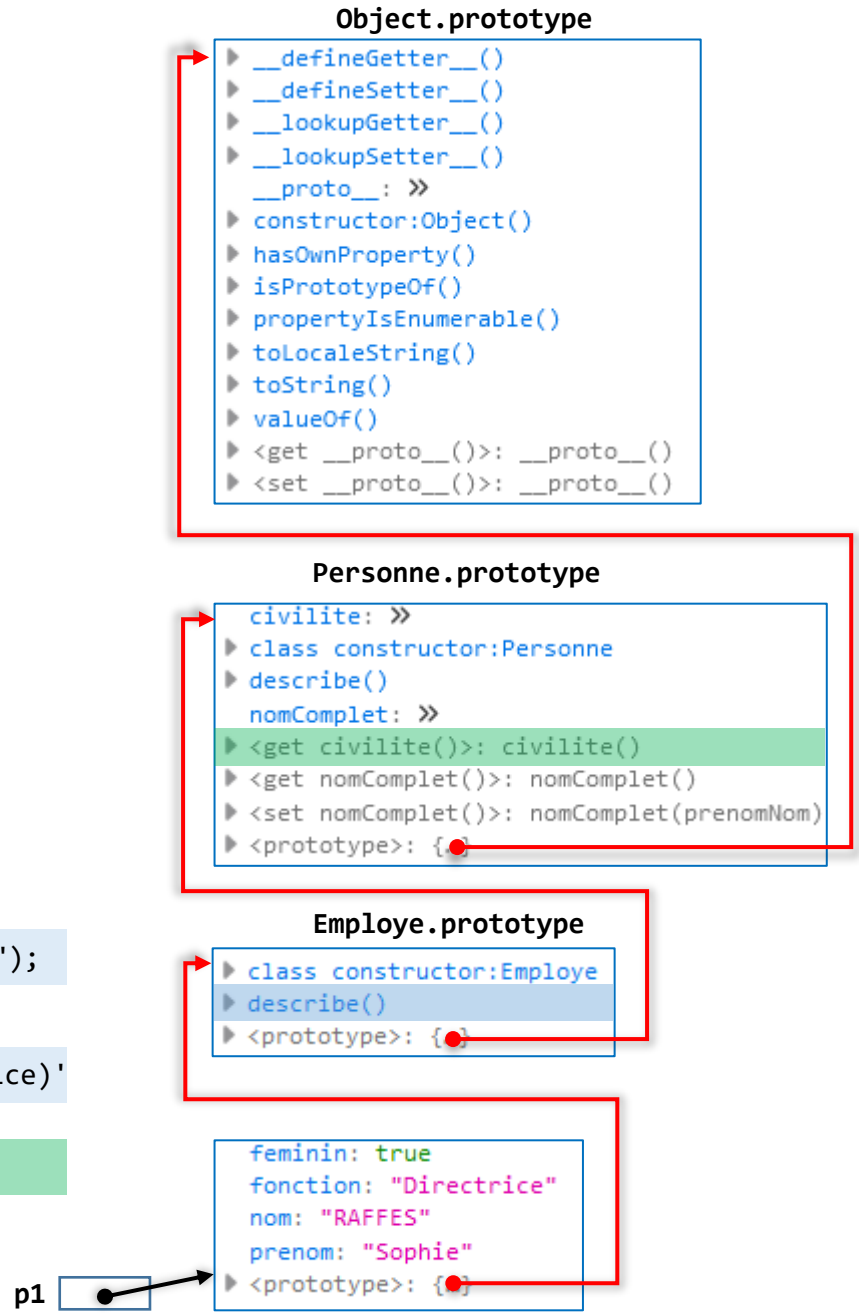
On veut exprimer le fait qu'un employé c'est une personne mais qui en plus a une fonction.

Possibilité d'obtenir ce comportement avec les chaînes de prototypes en JavaScript

```
let p1 = new Employe("Sophie", "RAFFES", true, "Directrice");
```

```
console.log(p1.describe()); → 'Mme RAFFES Sophie (Directrice)'
```

```
console.log(p1.civilite);   → 'Mme'
```



Objets et ES6 : Classes - Héritage

- Héritage – **`extends`** permet d'exprimer cette relation d'héritage (et de mettre en place la chaîne de prototypes)

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++)
            console.log(personnes[i].describe());
        }
    }
}
```

`super()` appelle le constructeur de la super classe.
Cet appel doit être fait avant d'accéder à `this`

```
class Employe extends Personne {

    constructor(prenom, nom, feminin, fonction) {
        super(prenom, nom, feminin);
        this.fonction = fonction;
    }

    describe() {
        return super.describe() + ` (${this.fonction})`;
    }
}
```

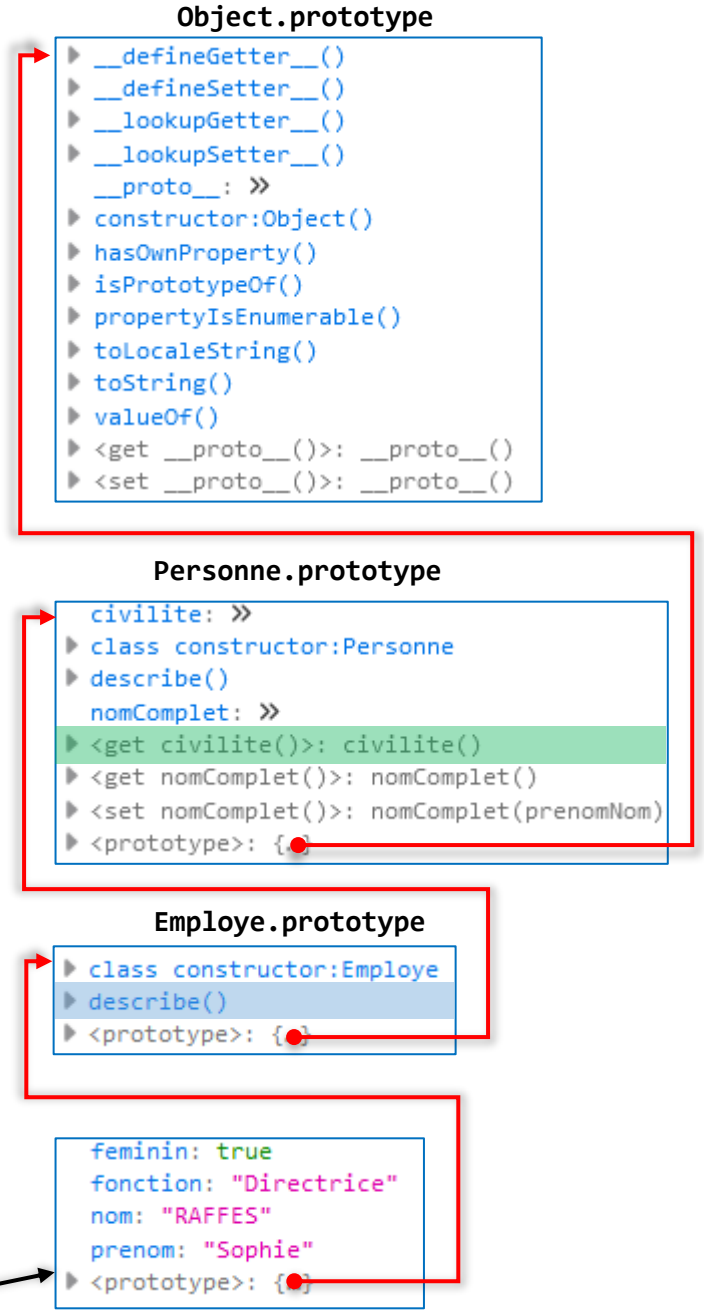
`super.nomMethode()` invoque une méthode héritée

```
let p1 = new Employe("Sophie", "RAFFES", true, "Directrice");
```

```
console.log(p1.describe()); → 'Mme RAFFES Sophie (Directrice)'
```

```
console.log(p1.civilite);   → 'Mme'
```

Voir le code



Objets et ES6 : Classes - Héritage

- Héritage – **`extends`** défini aussi une chaîne de <prototype> au niveau des classes (fonctions constructeurs) → héritage des membres statiques

```
class Personne {
    // ...
    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }

}

class Employe extends Personne {

    // ...
}
```

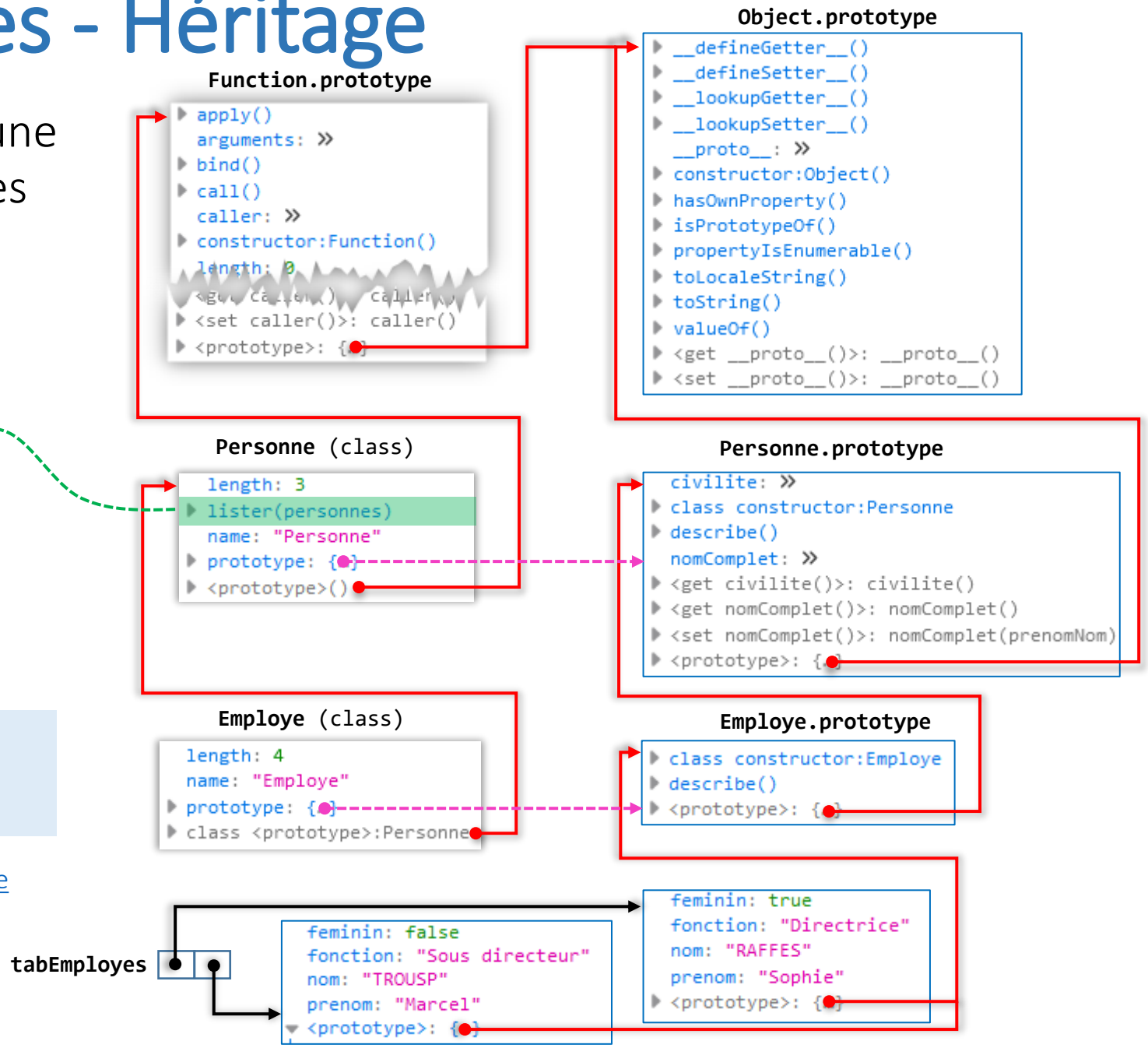
```
let tabEmployes = [
  new Employe("Sophie", "RAFFES", true, "Directrice") ,
  new Employe("Marcel", "TROUSP", false, "Sous directeur")
];
```

```
Employe.lister(tabEmployes);
```

→ Employe

Mme RAFFES Sophie (Directrice)
M. TROUSP Marcel (Sous directeur)

Voir le code



Objets et ES6 : opérateur `instaceof`

- `x instanceof C` renvoie `true` si l'objet référencé par `x` est instance de la classe `C` ou d'une sous classe de `C` (plus précisément si `C.prototype` se trouve dans la chaîne de `<prototypes>` de `x`), `false` sinon

```
let p1 = new Employe("Sophie", "RAFFES", true, "Directrice");
```

```
p1 instanceof Employe     → true
```

```
p1 instanceof Personne    → true
```

```
p1 instanceof Object      → true
```

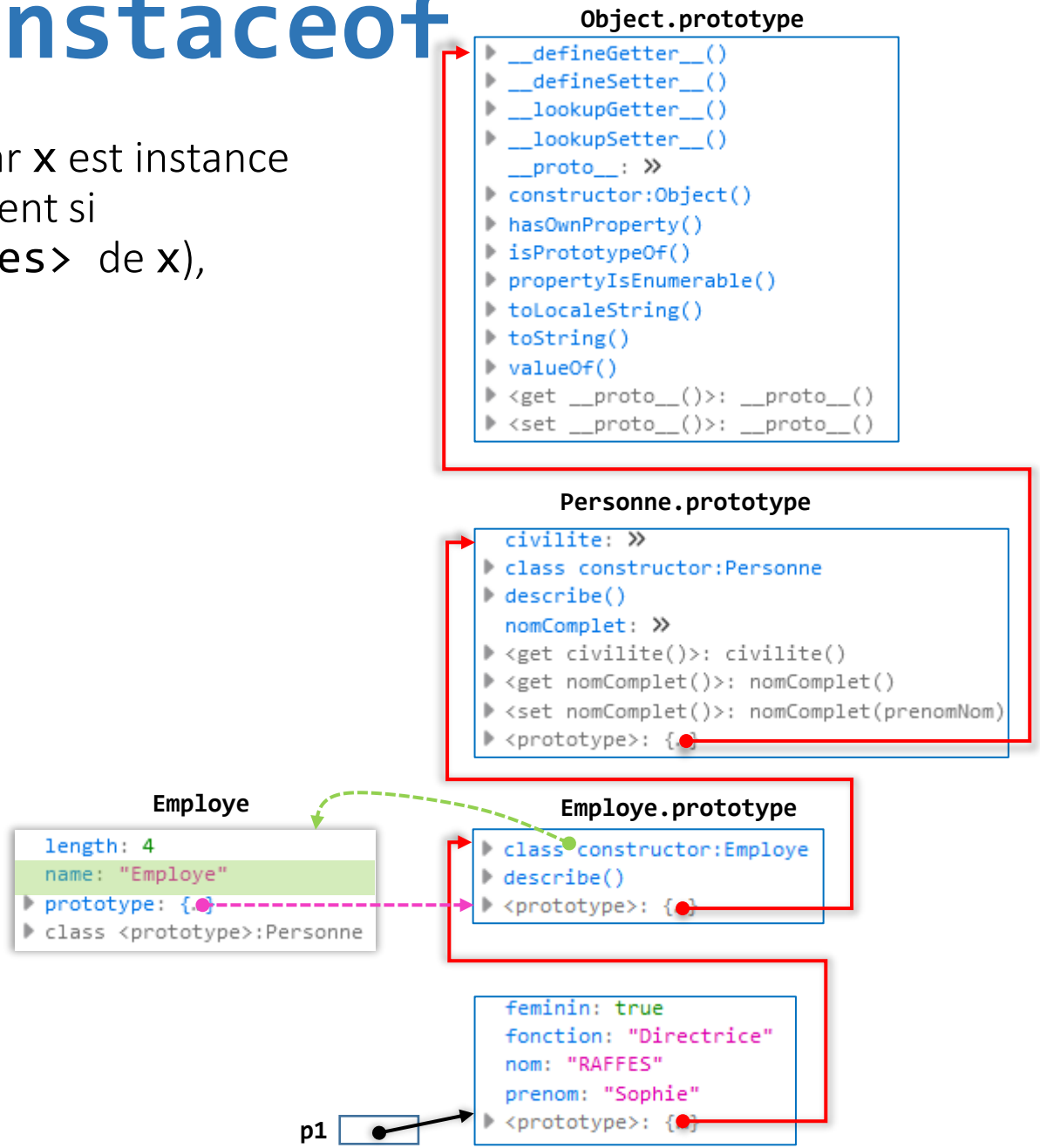
```
let p2 = new Personne("Maeva", "FASFER", true);
```

```
p2 instanceof Employe     → false
```

```
typeof p1 → object
```

Nom de la classe qui a permis de créer p1

```
p1.constructor.name       → Employe
```



Objets JavaScript

- Quelques liens

  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

  - https://www.w3schools.com/js/js_object_definition.asp
  - https://www.w3schools.com/js/js_class_intro.asp

  - http://blog.xebia.fr/2013/06/10/javascript-retour-aux-bases-constructeur-prototype-et-heritage/

Objets et ES6 : Classes - Héritage

- Avez-vous compris les objets et chaines de <prototype> ?
  - Exercices
    soit les classes **`Personne`** et **`Employe`** ci-contre
    quel affichage produisent les instructions suivantes ?
    Justifiez vos réponses

```
let tab1 = [
    new Personne("Marcel", "TROUSP", false),
    new Personne("Sophie", "RERHAX", true);
];

let tab2 = [
    new Employe("J.M.", "FAVRE", false, 'Sous Directeur'),
    new Employe("Maeva", "FARRES", true, 'Directrice');
];

let tab3 = [
    new Personne("J.M.", "FAVRE", false),
    new Employe("Maeva", "FARRES", true, 'Directrice');
];
```

```
Personne.lister(tab1);          → ???
```

```
Employe.lister(tab2);           → ???
```

```
Employe.lister(tab3);           → ???
```

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

```
class Employe extends Personne {

    constructor(prenom, nom, feminin, fonction) {
        super(prenom, nom, feminin);
        this.fonction = fonction;
    }

    describe() {
        return super.describe() + ` (${this.fonction})`;
    }
}
```

Objets et ES6 : Classes

- Avez-vous compris les objets et chaines de <prototype> ?
  - Exercices
    soit les classes **`Personne`** et **`Employe`** ci-contre
    quel affichage produisent les instructions suivantes ?
    Justifiez vos réponses

```
const p1 = new Personne("Marcel", "TROUSP", false);
const p2 = new Employe("Sophie", "RAFFES", true, "Directrice");
```

```
console.log(p1.describe());     → ???
```

```
console.log(p2.describe());     → ???
```

```
p1.fonction = 'Sous Directeur';
console.log(p1.describe());     → ???
```

```
delete p2.fonction;
console.log(p2.describe());     → ???
```

```
const p3 = new Personne("J.M.", "FAVRE", false, "Directeur Général");
console.log(p3.describe());     → ???
```

```
const p4 = new Employe("Maeva", "RAFFES", true);
console.log(p4.describe());     → ???
```

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

```
class Employe extends Personne {

    constructor(prenom, nom, feminin, fonction) {
        super(prenom, nom, feminin);
        this.fonction = fonction;
    }

    describe() {
        return super.describe() + ` (${this.fonction})`;
    }
}
```

Objets et ES6 : Classes

- Avez-vous compris les objets et chaines de <prototype> ?
  - Exercices
    soit les classes **Personne** et **Employe** ci-contre
    quel affichage produisent les instructions suivantes ?
    Justifiez vos réponses

```
const p1 = new Personne("Marcel", "TROUSP", false);
const p2 = new Employe("Sophie", "RAFFES", true, "Directrice");
```

```
console.log(p1.describe());     → M. TROUSP Marcel
```

```
console.log(p2.describe());     → Mme RAFFES Sophie (Directrice)
```

```
p1.fonction = 'Sous Directeur';
console.log(p1.describe());     → M. TROUSP Marcel
```

```
delete p2.fonction;
console.log(p2.describe());     → Mme RAFFES Sophie (undefined)
```

```
const p3 = new Personne("J.M.", "FAVRE", false, "Directeur Général");
console.log(p3.describe());     → M. FAVRE J.M.
```

```
const p4 = new Employe("Maeva", "RAFFES", true);
console.log(p4.describe());     → Mme RAFFES Maeva (undefined)
```

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

```
class Employe extends Personne {

    constructor(prenom, nom, feminin, fonction) {
        super(prenom, nom, feminin);
        this.fonction = fonction;
    }

    describe() {
        return super.describe() + ` (${this.fonction})`;
    }
}
```

Objets et ES6 : Classes

- Avez-vous compris les objets et chaines de <prototype> ?
  - Exercices
    soit les classes **Personne** et **Employe** ci-contre
    quel affichage produisent les instructions suivantes ?
    Justifiez vos réponses

```
let tab1 = [
    new Personne("Marcel", "TROUSP", false),
    new Personne("Sophie", "RERHAX", true)
];

let tab2 = [
    new Employe("J.M.", "FAVRE", false, 'Sous Directeur'),
    new Employe("Maeva", "FARRES", true, 'Directrice')
];

let tab3 = [
    new Personne("J.M.", "FAVRE", false),
    new Employe("Maeva", "FARRES", true, 'Directrice')
];
```

```
Personne.lister(tab1);
```

```
Personne
--------
M. TROUSP Marcel
Mme RERHAX Sophie
```

```
Employe.lister(tab2);          → ???
```

```
Employe
--------
M. FAVRE J.M. (Sous Directeur)
Mme FARRES Maeva (Directrice)
```

```
Employe.lister(tab3);
```

```
Employe
--------
M. FAVRE J.M.
Mme FARRES Maeva (Directrice)
```

```
class Personne {
    constructor(prenom, nom, feminin) {
        this.prenom = prenom;
        this.nom = nom;
        this.feminin = feminin;
    }

    get nomComplet() {
        return `${this.nom} ${this.prenom}`;
    }

    set nomComplet(prenomNom) {
        const tokens = prenomNom.split(/\b\s+(?!$)/);
        this.prenom = tokens[0];
        this.nom = tokens[1];
    }

    get civilite() {
        return (this.feminin) ? "Mme" : "M.";
    }

    describe() {
        return `${this.civilite} ${this.nomComplet}`;
    }

    static lister(personnes) {
        console.log(this.name);
        console.log("--------");
        for (let i = 0; i < personnes.length; i++) {
            console.log(personnes[i].describe());
        }
    }
}
```

```
class Employe extends Personne {

    constructor(prenom, nom, feminin, fonction) {
        super(prenom, nom, feminin);
        this.fonction = fonction;
    }

    describe() {
        return super.describe() + ` (${this.fonction})`;
    }
}
```