

Javascript

#03

Matthieu Nicolas
Licence Pro CIASIE
Slides par Christophe Bouthier

Plan

- Héritage

Héritage

Javascript

#03

Constructeur

- Fonction
 - appelée avec **new**
 - *this* = nouvel objet vide
 - définit le contenu de *this*
 - retourne *this*

Example

```
1  const Character = function () {  
2    ...this.hp = 100  
3    ...this.getHP = function () {  
4      ...return this.hp  
5    }  
6  }  
7  
8  const charac = new Character()  
9  charac.getHP() // 100  
10
```

En mémoire

```
1  const Character = function () {  
2    this.hp = 100  
3    this.getHP = function () {  
4      return this.hp  
5    }  
6  }  
7  
8  const charac = new Character()  
9  charac.getHP() // 100  
10
```

En mémoire

Character → fonction

charac → objet

```
1  const Character = function () {  
2    ... this.hp = 100  
3    ... this.getHP = function () {  
4      ... return this.hp  
5    ... }  
6  }  
7  
8  const charac = new Character()  
9  charac.getHP() // 100  
10
```

En mémoire

Character → fonction

charac → objet

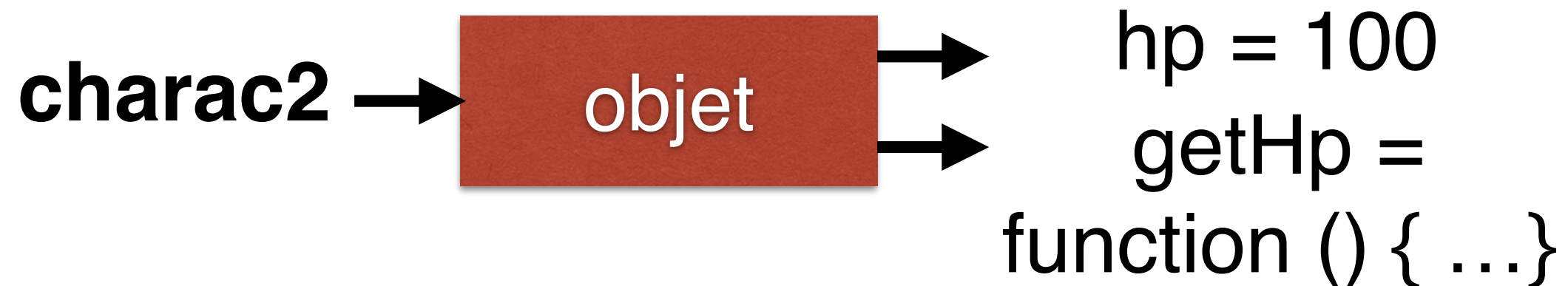
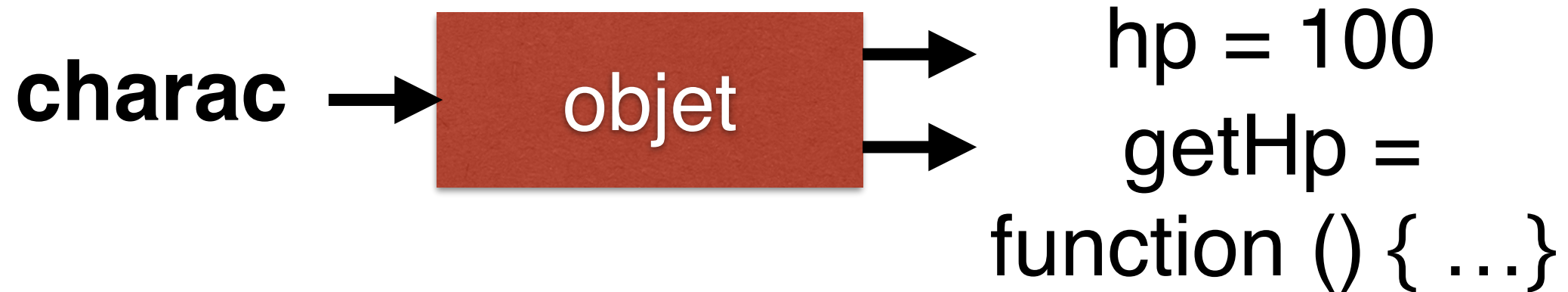
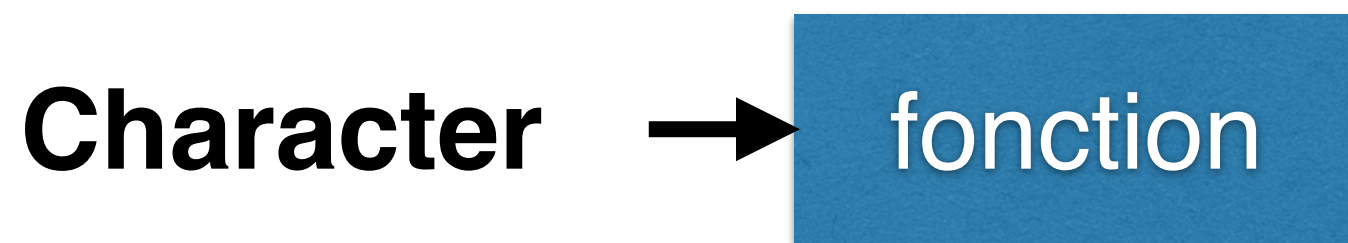
hp = 100
getHp =
function () { ... }

```
1  const Character = function () {  
2    ... this.hp = 100  
3    ... this.getHP = function () {  
4      ... return this.hp  
5    ... }  
6  }  
7  
8  const charac = new Character()  
9  charac.getHP() // 100  
10
```


Nouvelle instance

```
1  const Character = function () {  
2    this.hp = 100  
3    this.getHP = function () {  
4      return this.hp  
5    }  
6  }  
7  
8  const charac = new Character()  
9  charac.getHP() // 100  
10  
11 const charac2 = new Character()  
12
```

En mémoire



Duplication des propriétés

- Toutes les propriétés de **Character** sont dupliquées pour chaque instance
- C'est cool pour les attributs...
 - Permet de faire évoluer les instances différemment
- ... ça l'est moins pour les méthodes
 - Gaspillage de ressources

Héritage de prototype

- Nous permet de définir une *structure* et un *comportement* commun à un ensemble d'objet
- Permet à chacun de ces objets de partager le même comportement...
- ... tout en évoluant de façon différente

Héritage en JS

- 2 propriétés
 - *constructor*
 - *prototype*
- Complémentaires
- Automatiques

Propriété *constructor*

- Tout **objet** a une propriété *constructor*
 - type : **fonction**
 - celle utilisée avec **new** pour créer l'objet
- `obj.constructor`
 - créée automatiquement

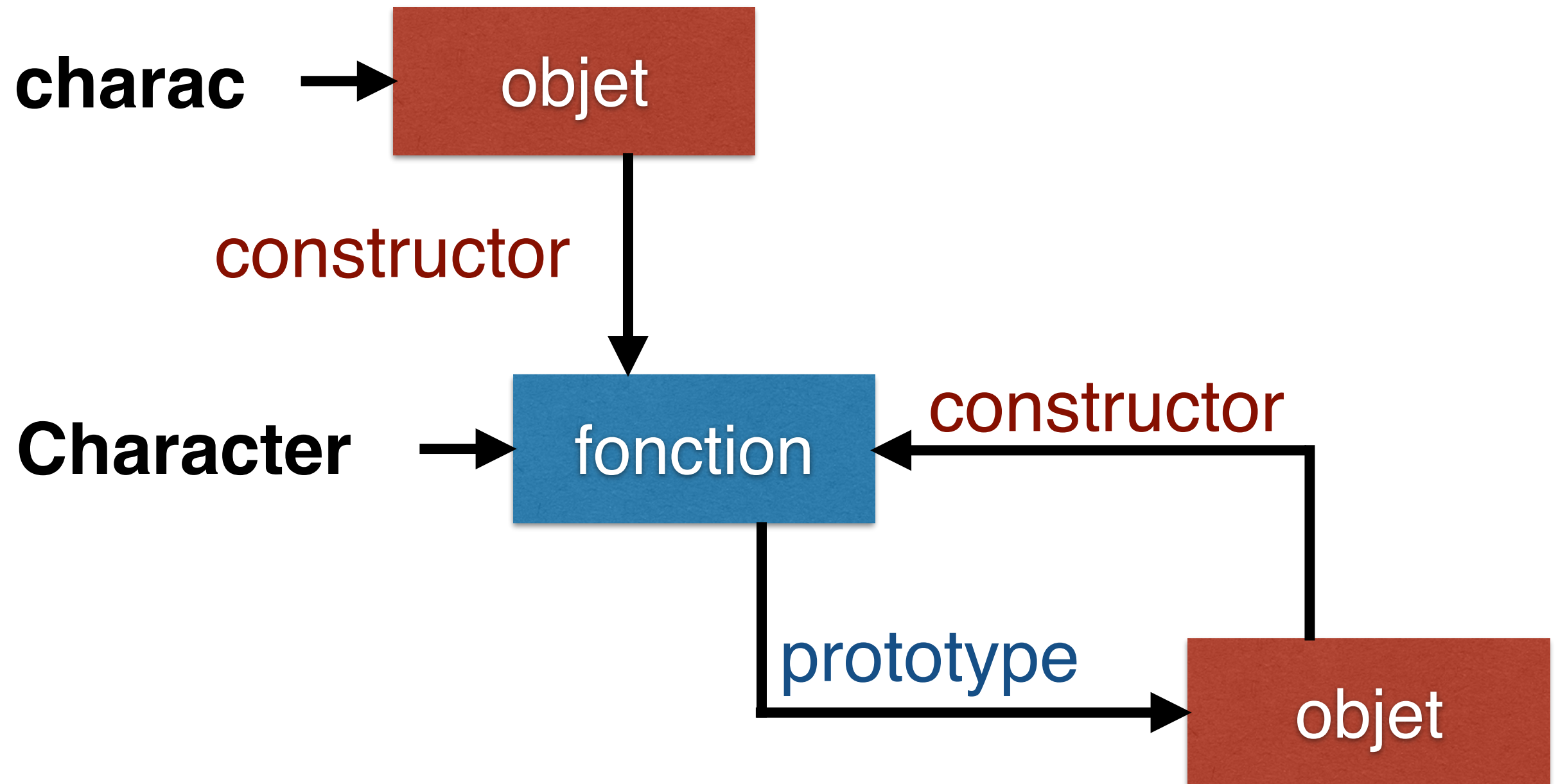
```
1  const charac = new Character()  
2  charac.constructor // Character  
3
```

Propriété *prototype*

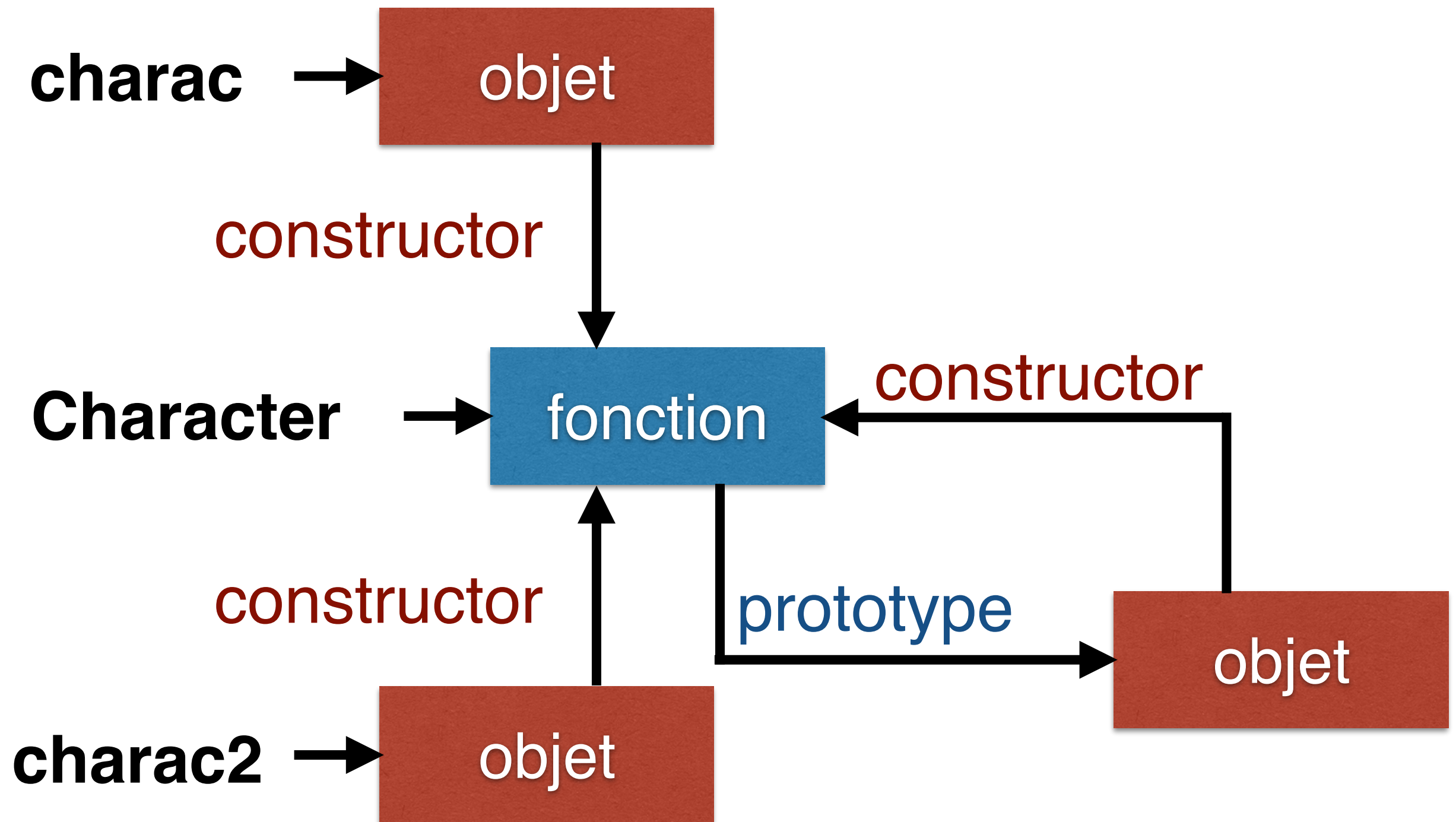
- Toute **fonction** a une propriété *prototype*
 - type : **objet**
 - valeur : *{ constructor: this }*
- **MyObj.prototype**
 - créée automatiquement

```
1  const charac = new Character()  
2  charac.prototype // { constructor: Character }  
3  |
```

```
1  const Character = function () { ... }  
2  const charac = new Character()  
3
```




```
1  const Character = function() { ... }  
2  const charac = new Character()  
3  const charac2 = new Character()  
4
```

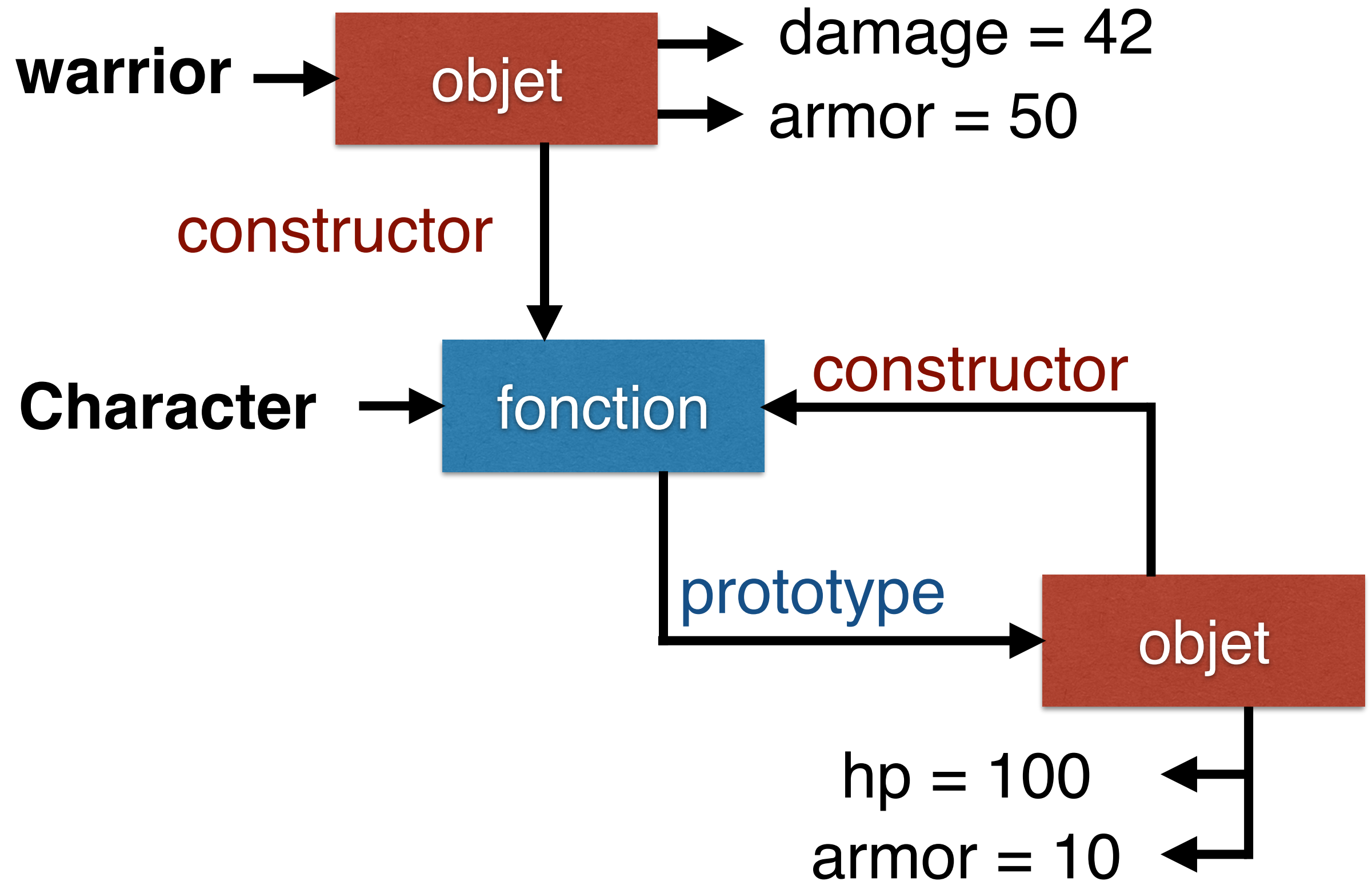


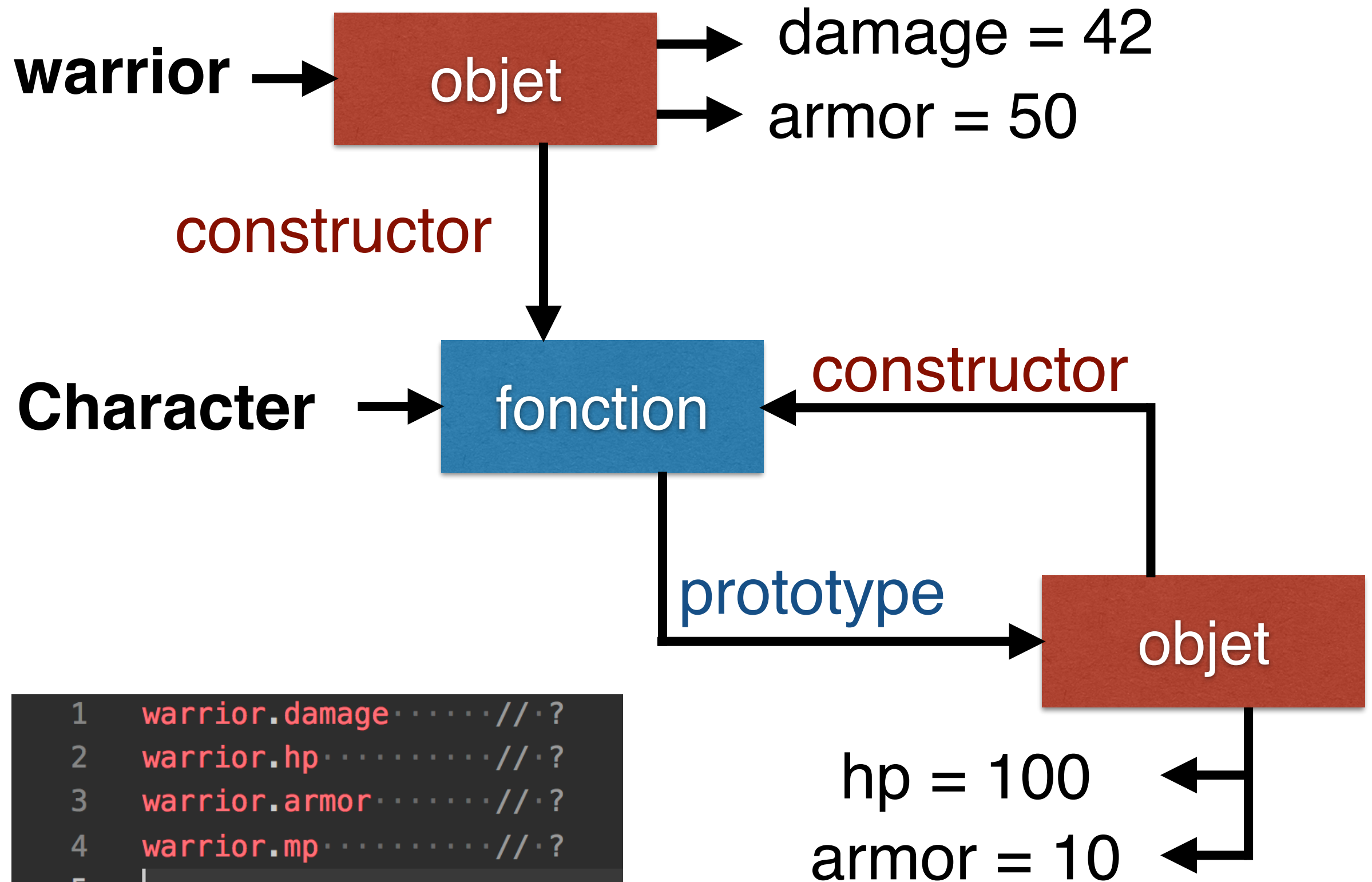
A quoi ça sert ?

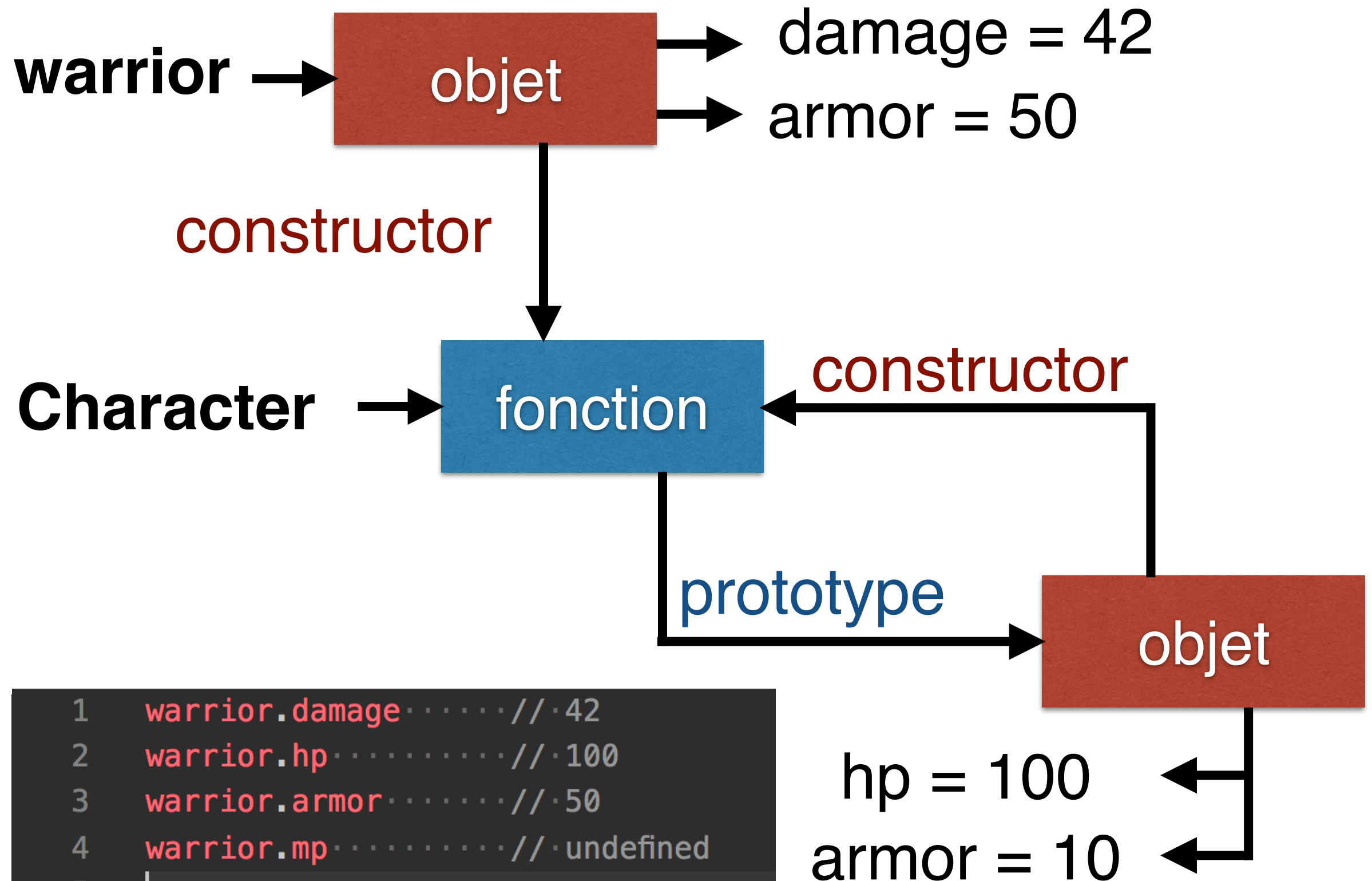
- À définir un ensemble de propriétés partagées par un type d'objet
- Sorte de typage

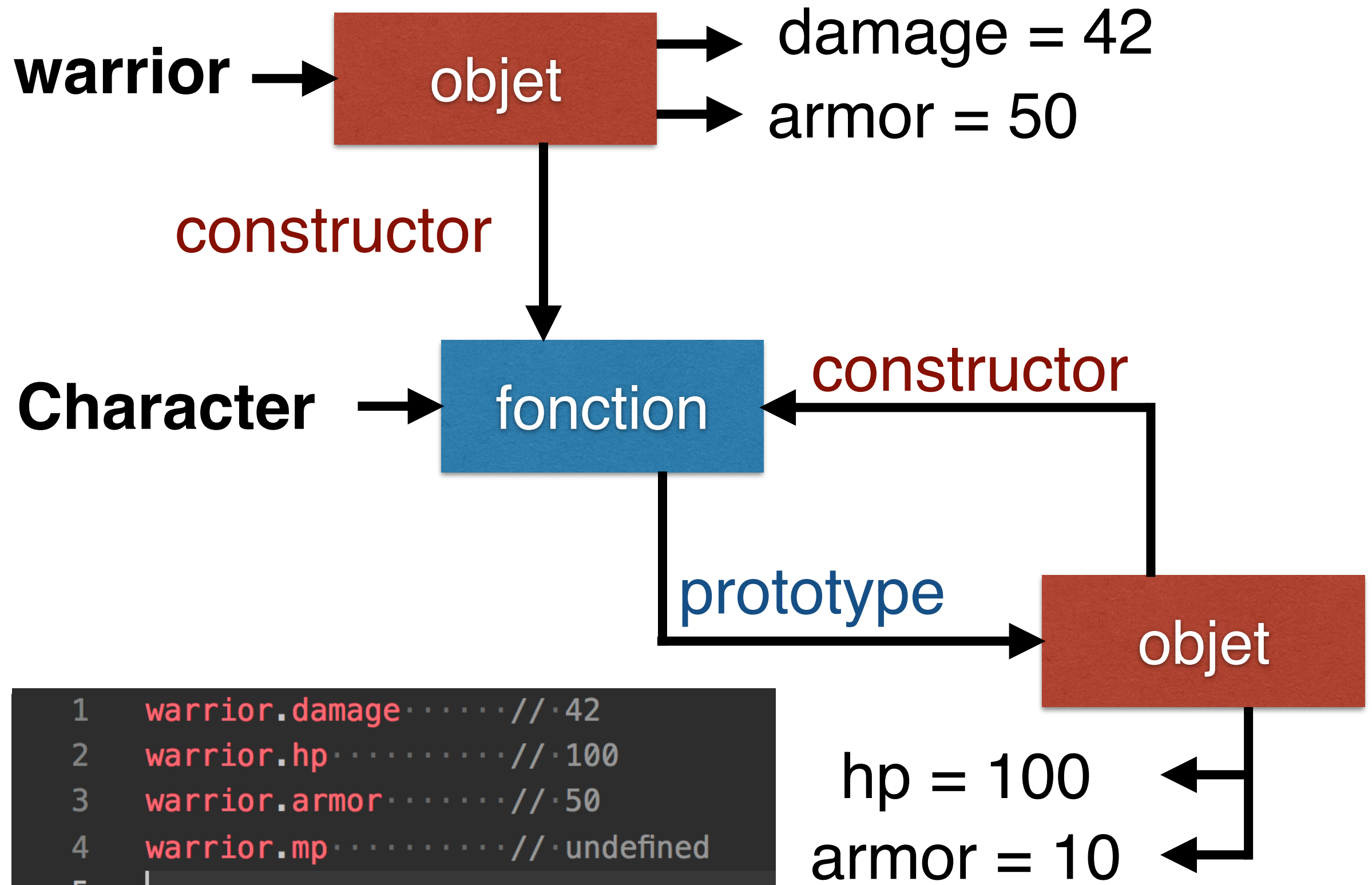
Example

```
1  const Character = function () {}  
2  Character.prototype.hp = 100  
3  Character.prototype.armor = 10  
4  
5  const warrior = new Character()  
6  warrior.damage = 42  
7  warrior.armor = 50  
8
```









Accès propriétés

- *warrior.damage*
 - accès à une propriété de l'objet
- *warrior.hp*
 - accès à une propriété du prototype
- *warrior.armor*
 - la propriété de l'objet masque le prototype
- *warrior.mp*
 - propriété non trouvée (ni objet ni prototype)

Tout est dynamique

- Possible de rajouter une propriété à un prototype d'un objet existant

```
1  const Character = function() {}
2  Character.prototype.hp = 100
3  Character.prototype.armor = 10
4
5  const warrior = new Character()
6  warrior.damage = 42
7  warrior.armor = 50
8
9  Character.prototype.mp = 50
10 warrior.mp ..... // 50
11
```

Tout est dynamique

- Possible de révéler une propriété du prototype en supprimant une de l'objet

```
1  const Character = function() {}
2  Character.prototype.hp = 100
3  Character.prototype.armor = 10
4
5  const warrior = new Character()
6  warrior.damage = 42
7  warrior.armor = 50
8
9  delete warrior.armor
10 warrior.armor ..... // 10
11
```

Jusque là...

- Tout **objet** a une propriété **constructor**
 - type : **function**
- Toute **fonction** a une propriété **prototype**
 - type : **object**
- Recherche propriété :
 - objet
 - puis *constructor.prototype*

Types existants

```
1  ({}).constructor ..... // function·Object()·{...}
2  [].constructor ..... // function·Array()·{...}
3  "".constructor ..... // function·String()·{...}
4  (1).constructor ..... // function·Number()·{...}
5  true.constructor ..... // function·Boolean() {...}
6  /aaa/.constructor ..... // function·RegExp()·{...}
7  f.constructor ..... // function·Function()·{...}
8  |
```

typeof

- Opérateur permettant de déterminer le type d'une valeur...

```
1  typeof 1 ..... // "number"
2  typeof "" ..... // "string"
3  typeof true ..... // "boolean"
4
5  typeof Infinity ..... // "number"
6  typeof NaN ..... // "number"
7  typeof undefined ..... // "undefined"
8  typeof null ..... // "object"
9  |
```

typeof

- ... mais peu utile avec les **Objects**

```
1  typeof warrior ..... // "object"
2  typeof [1, 2, 3, 4] .. // "object"
3  typeof {} ..... // "object"
4  typeof null ..... // "object"
5  |
```

instanceof

- Opérateur permettant de déterminer plus précisément le type d'un objet

```
1 warrior instanceof Character ..... // true
2 [1, 2, 3, 4] instanceof Character ... // false
3 ({}). instanceof Character ..... // false
4 null instanceof Character ..... // false
5
```

Temps pour un peu d'exercice

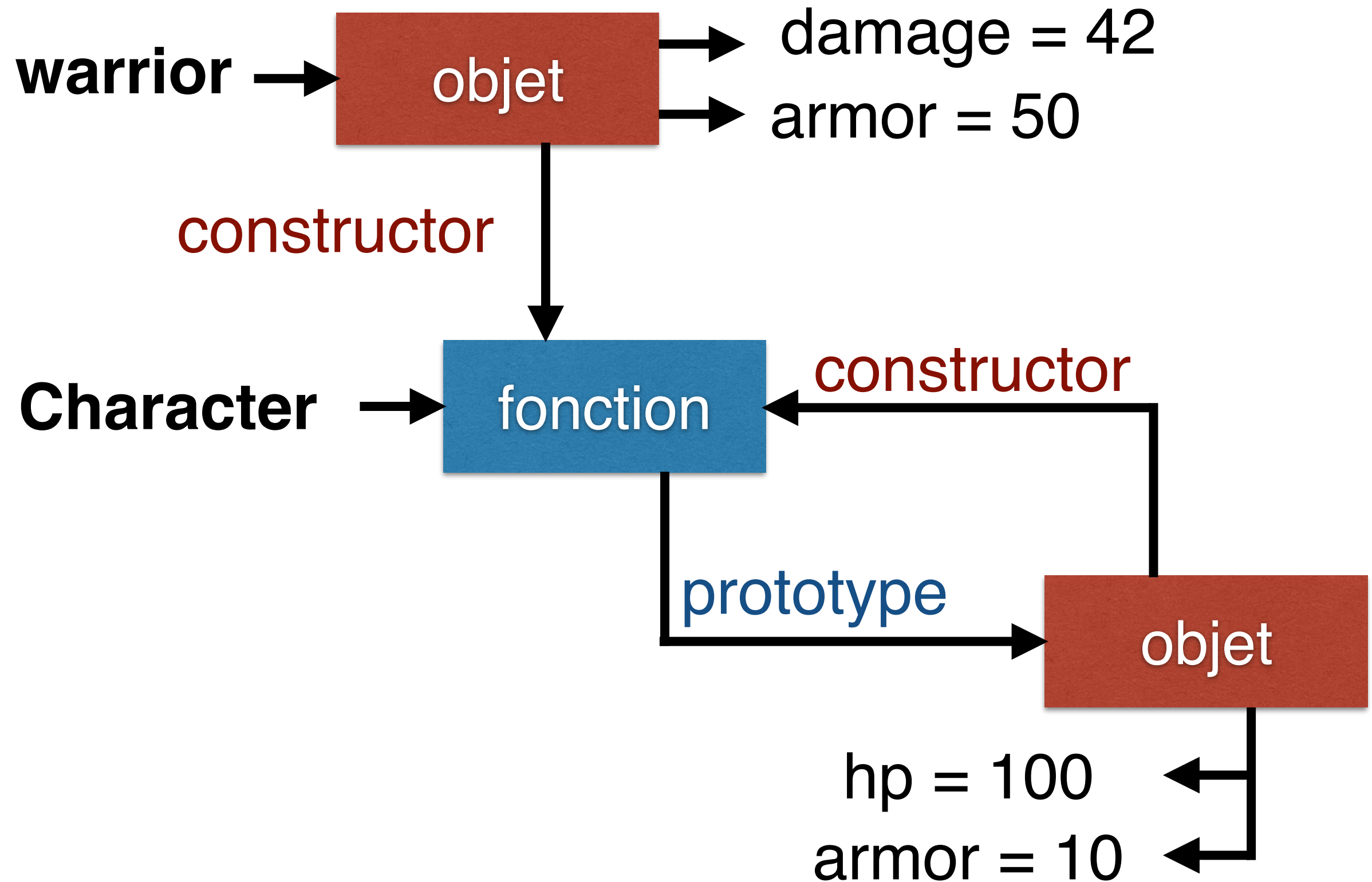
<https://classroom.github.com/a/ZoKpjWsO>
(lien dispo sur Arche)

Ça se complique

- Par défaut
 - *prototype* est créé automatiquement
 - *{ constructor: this }*
- Et si ce n'est pas le cas ?
 - Peut spécifier le prototype à la main
 - Création d'une chaine de prototypes

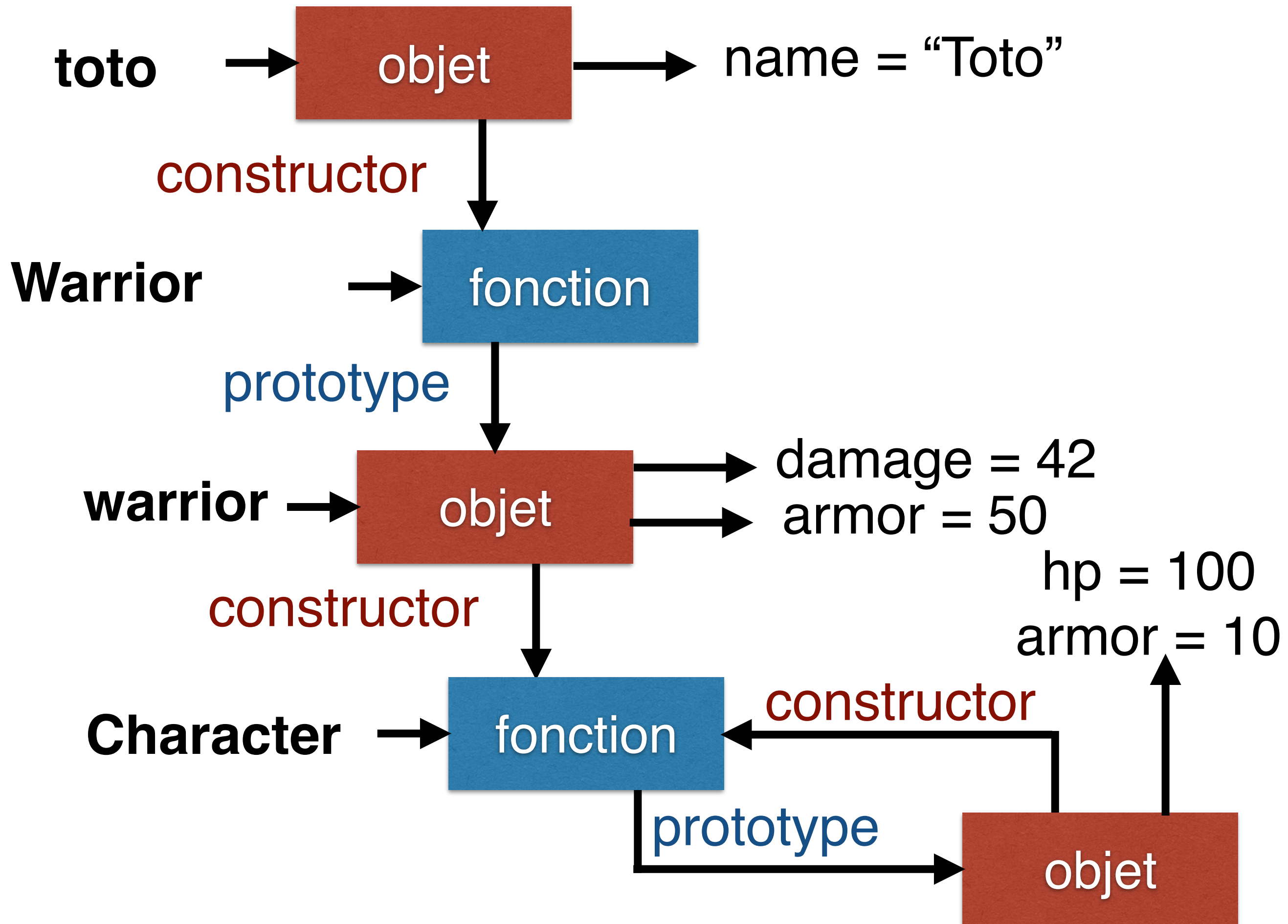
Example

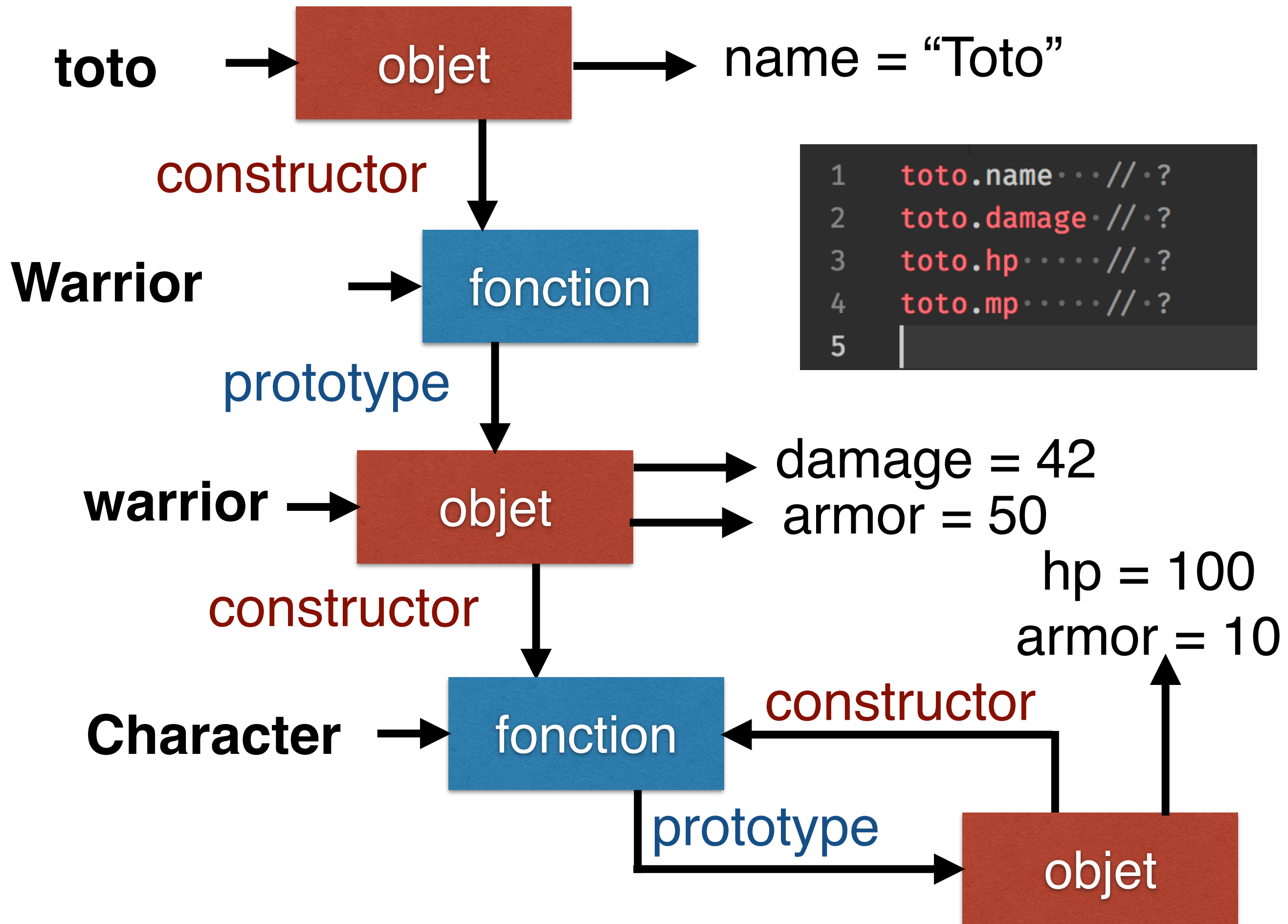
```
1  const Character = function () {}  
2  Character.prototype.hp = 100  
3  Character.prototype.armor = 10  
4  
5  const warrior = new Character()  
6  warrior.damage = 42  
7  warrior.armor = 50  
8
```



Example

```
1  const Warrior = function () {}  
2  Warrior.prototype = warrior  
3  
4  const toto = new Warrior()  
5  toto.name = "Toto"  
6
```





Accès propriétés

- *toto.name = "Toto"*
 - accès à une propriété de l'objet
- *toto.damage = 42*
 - accès à une propriété du prototype
- *toto.hp = 100*
 - accès à une propriété du prototype du constructeur du prototype
- On remonte la chaine...

instanceof

- Fonctionne aussi avec les chaînes de prototypes

```
1  toto instanceof Warrior ..... // true
2  toto instanceof Character ... // true
3
```


Object

- Tout est objet
 - « Tout les types dérivent du type Object »
 - Tous les types ont pour prototype le type **Object**
- *toto instanceof Object*
- toutes les propriétés d'**Object** se retrouvent dans tous les objets
 - *toString()*

Tout est Object ?

- Pas les Immuables

```
1  (1) · isinstance · Object · · · // · false
2  "" · isinstance · Object · · · // · false
3  true · isinstance · Object · · // · false
4  |
```

Tout est Object ?

- MAIS

```
1  new Number().instanceof Object .....// true
2  new String().instanceof Object .....// true
3  new Boolean().instanceof Object .....// true
4  |
```

Specials

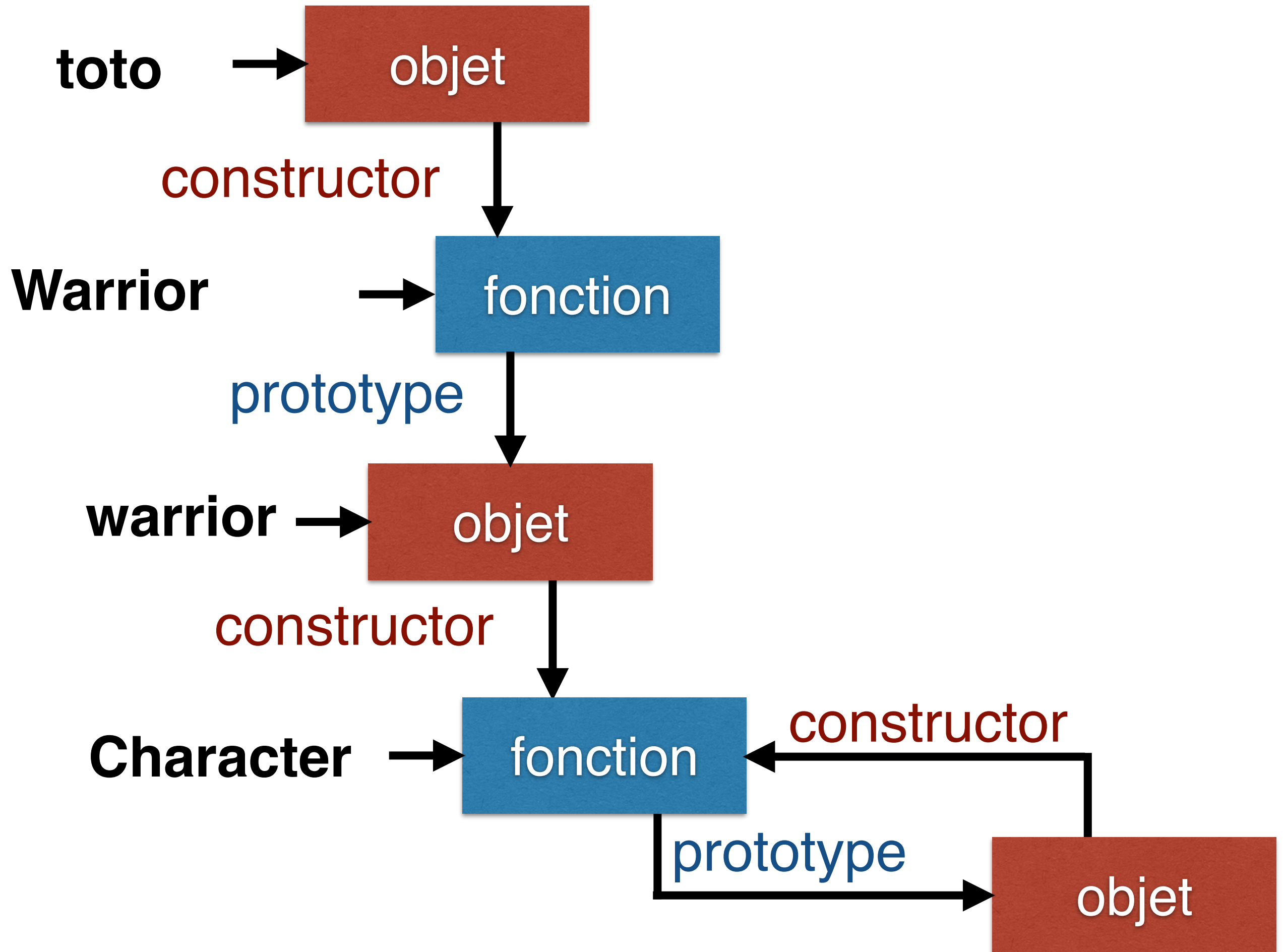
```
1  Infinity · instanceof · Object · · · // · false
2  NaN · instanceof · Object · · · · · // · false
3  undefined · instanceof · Object · · // · false
4  null · instanceof · Object · · · · · // · false
5  |
```

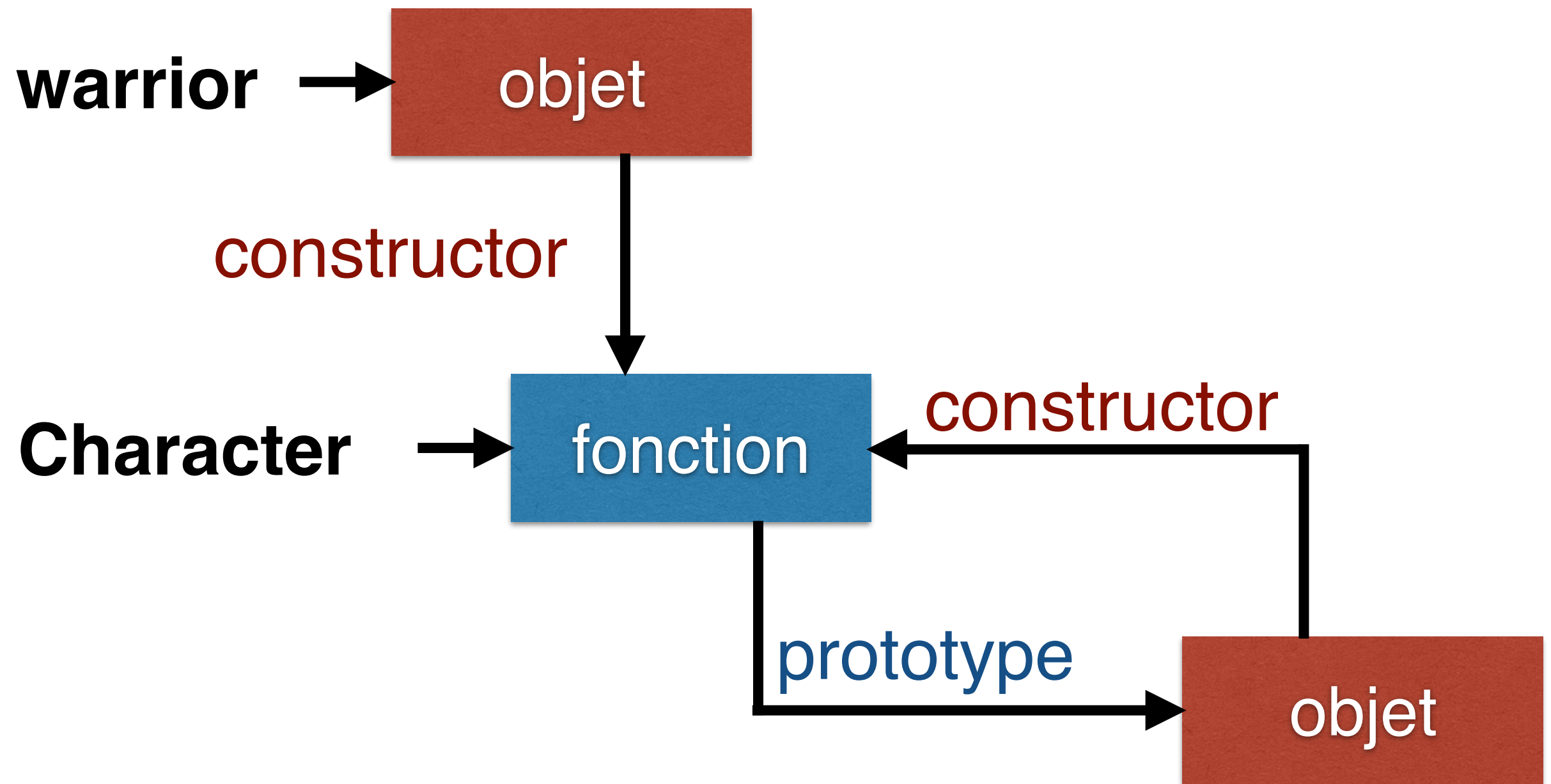
**Vous reprendrez bien
un peu de tests
unitaires ?**

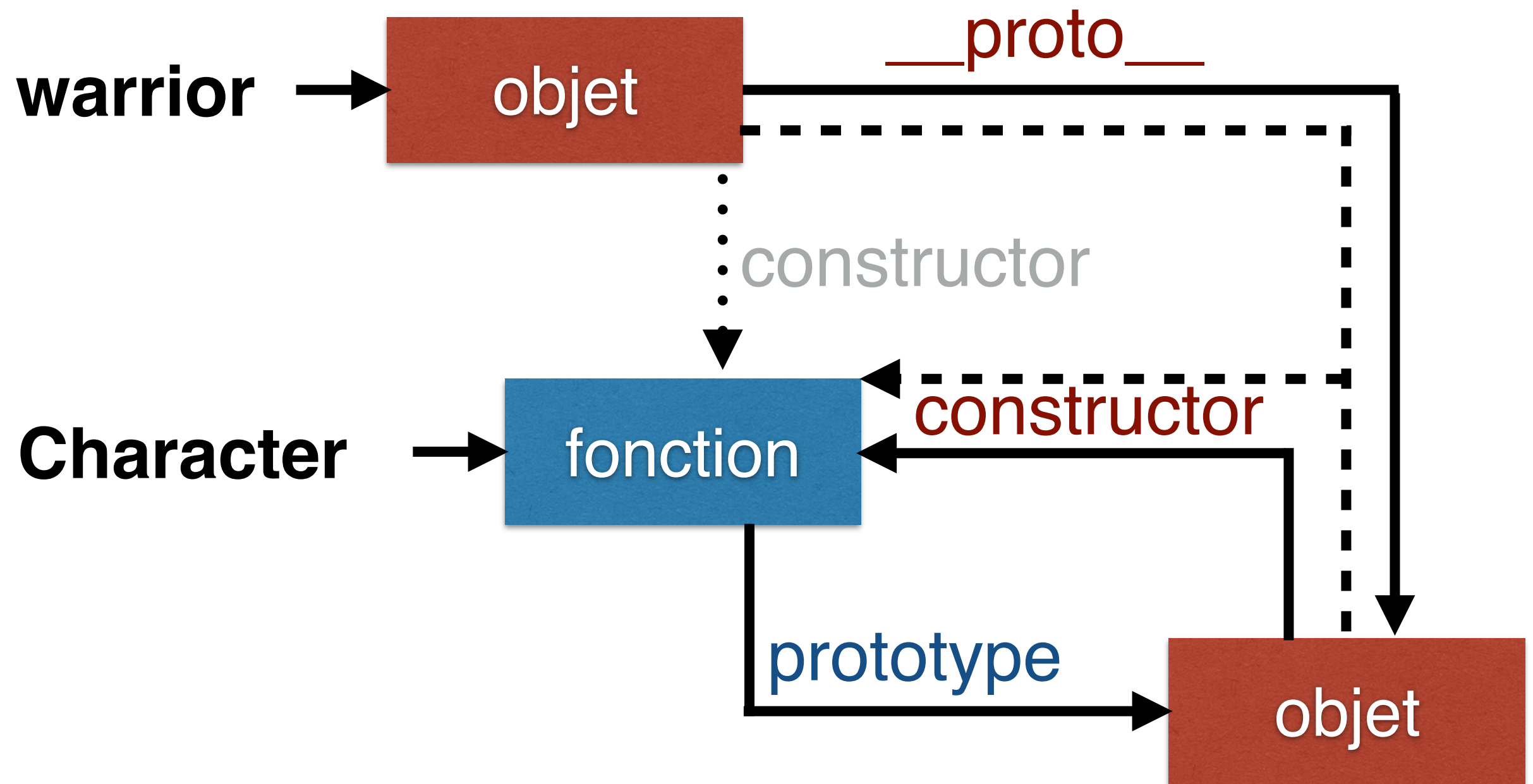
Récupérez la partie 2 à partir du repo distant
classroom

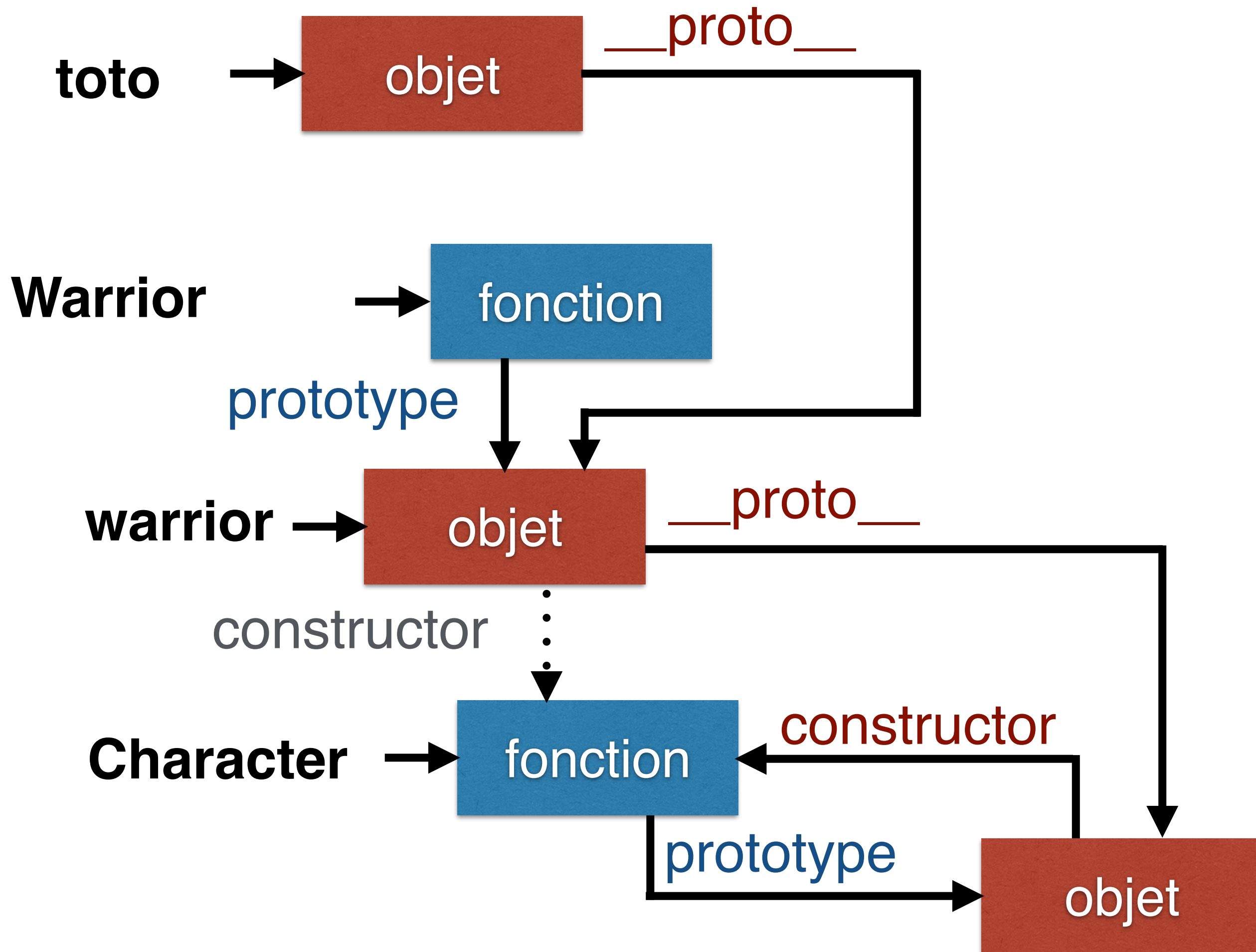
Sauf qu'en fait...

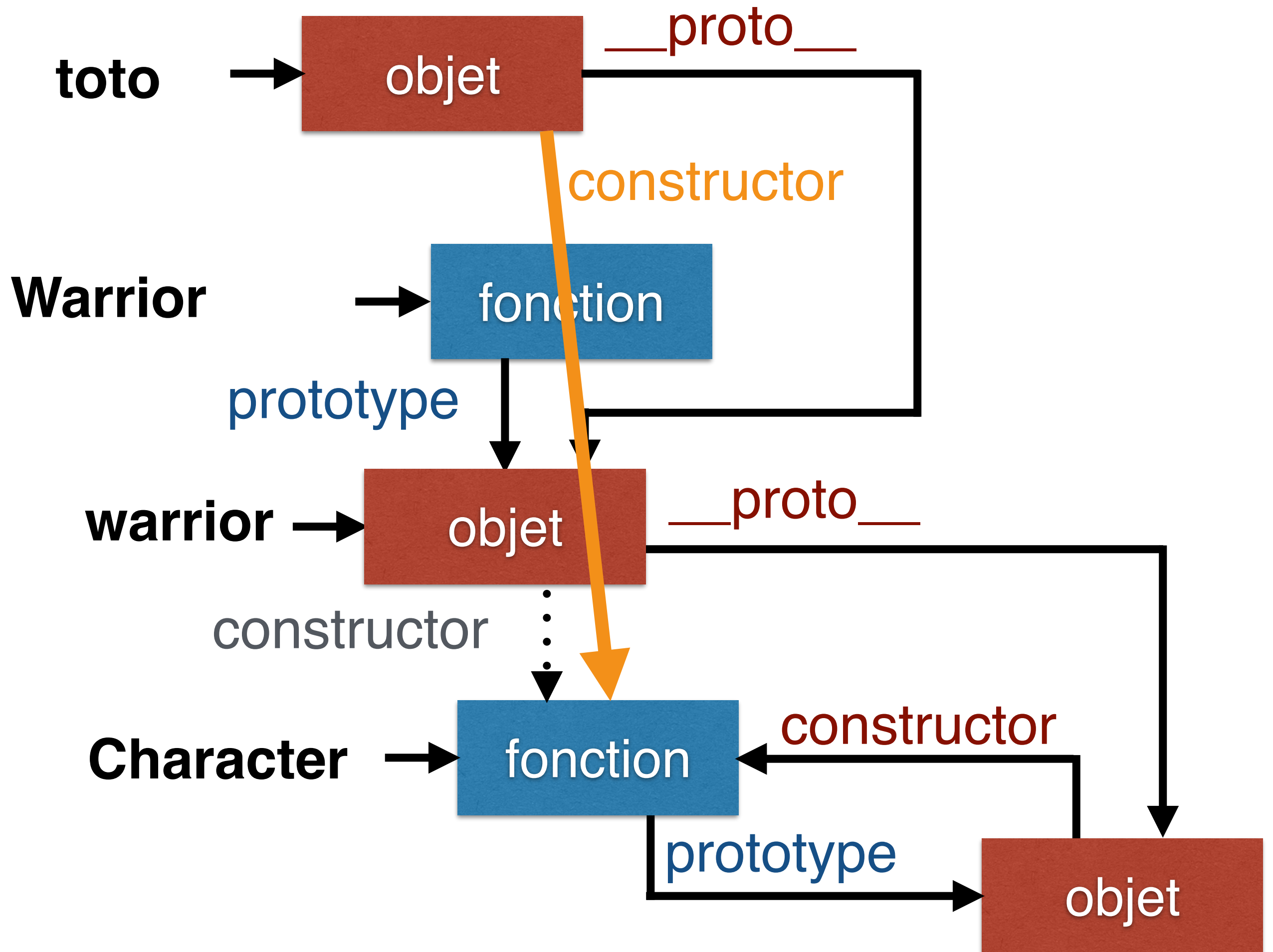
```
1  Character = function() {};  
2  const warrior = new Character();  
3  
4  Warrior = function() {};  
5  Warrior.prototype = warrior;  
6  
7  toto = new Warrior();  
8  
9  toto.constructor ... // ?  
10
```











instanceof

- Ne se sert pas de “constructor”
- Parcourt la chaine de prototypes
- À chaque étape, teste :
 - *obj.__proto__ === Class.prototype*

```
1  toto instanceof Warrior ..... // true
2  toto instanceof Character ... // true
3  toto instanceof Object ..... // true
4  |
```

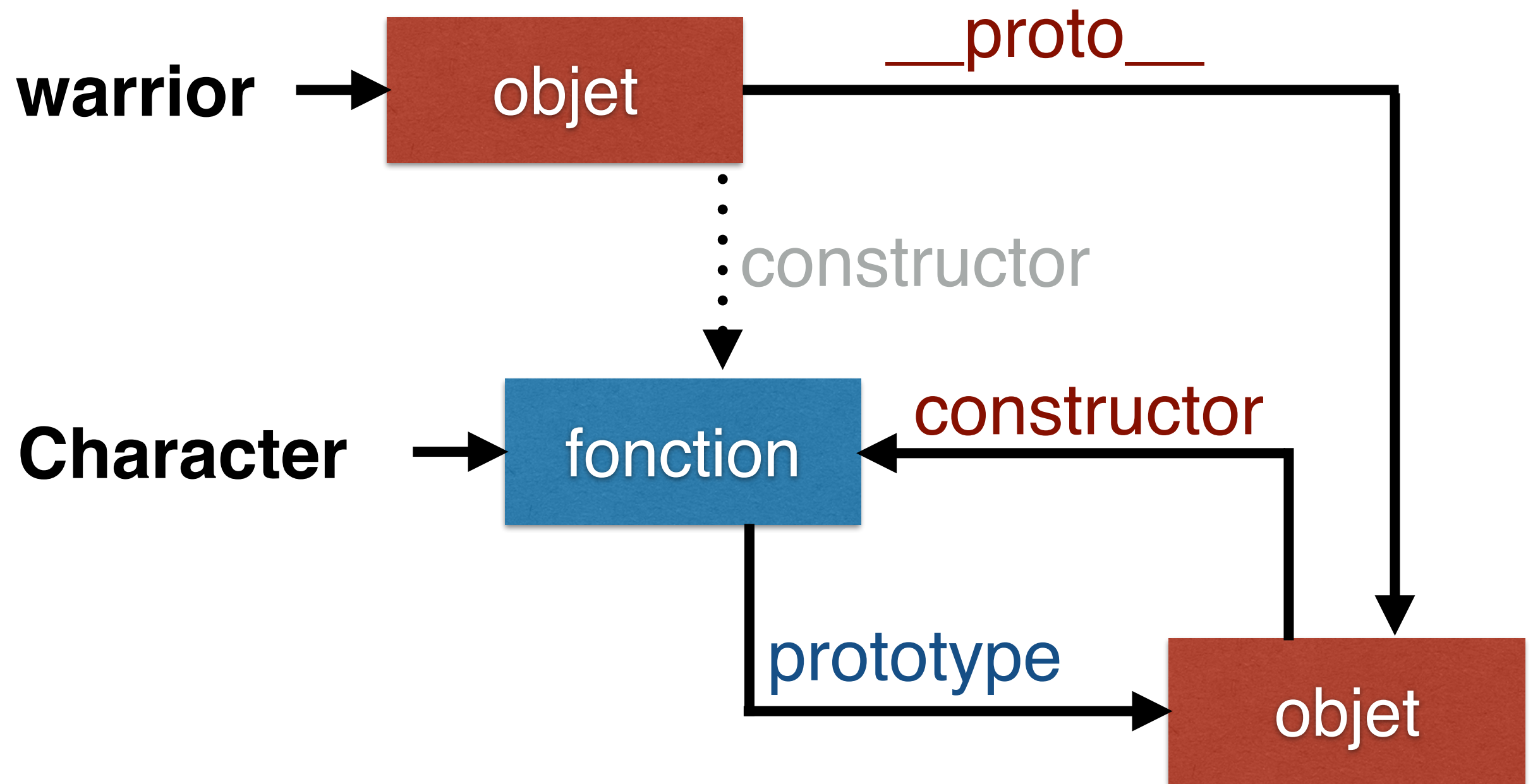
__proto__

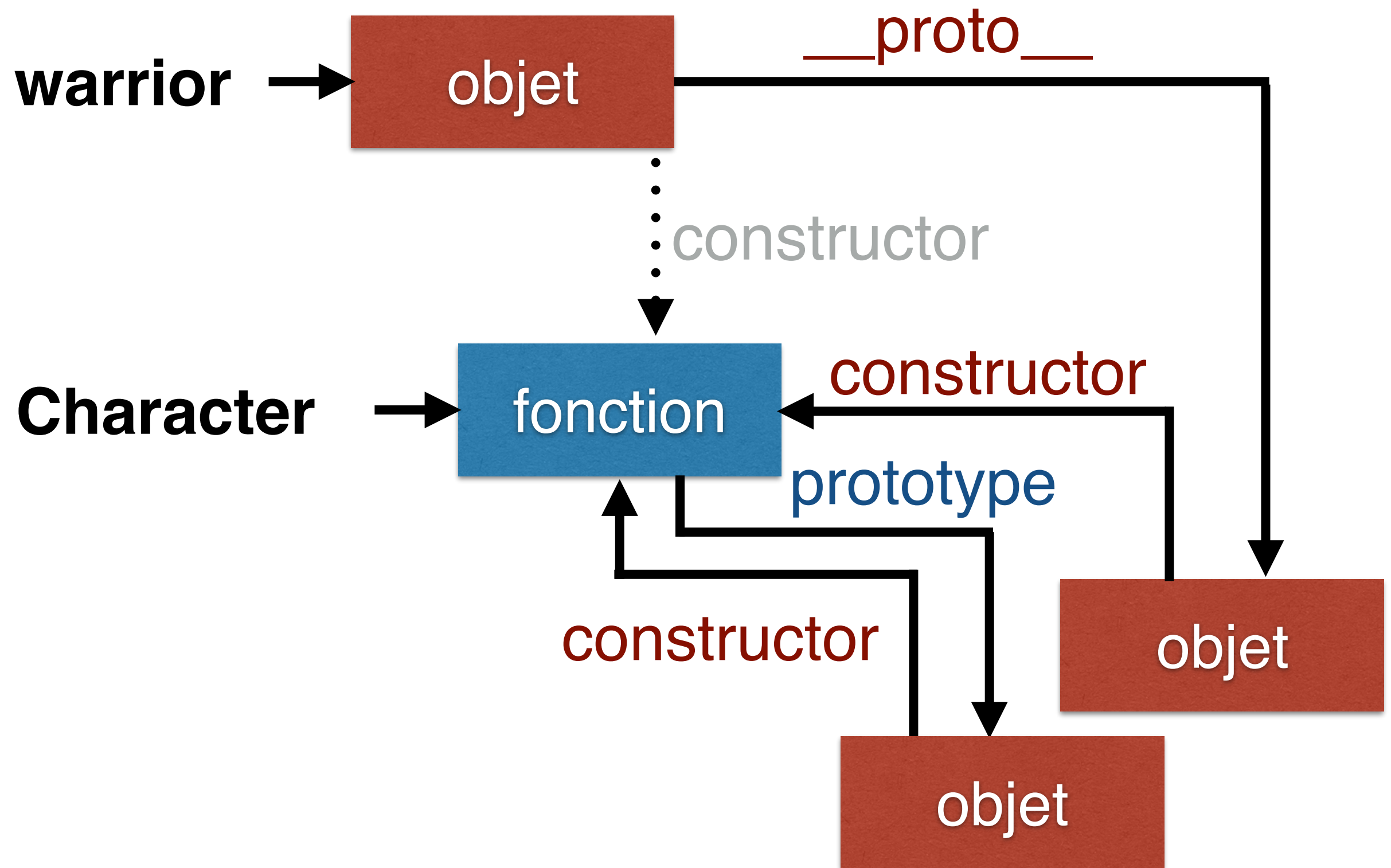
- Pourquoi ne pas apprendre directement le modèle avec __proto__ ?
 - __proto__ n'est PAS STANDARD
 - Pas implémenté dans IE
 - Implémentations différentes Mozilla/Webkit
- Fonctions spécifiées dans ECMAScript 6
 - Sous le nom **[[Prototype]]**
- MAIS : pas obligatoire !

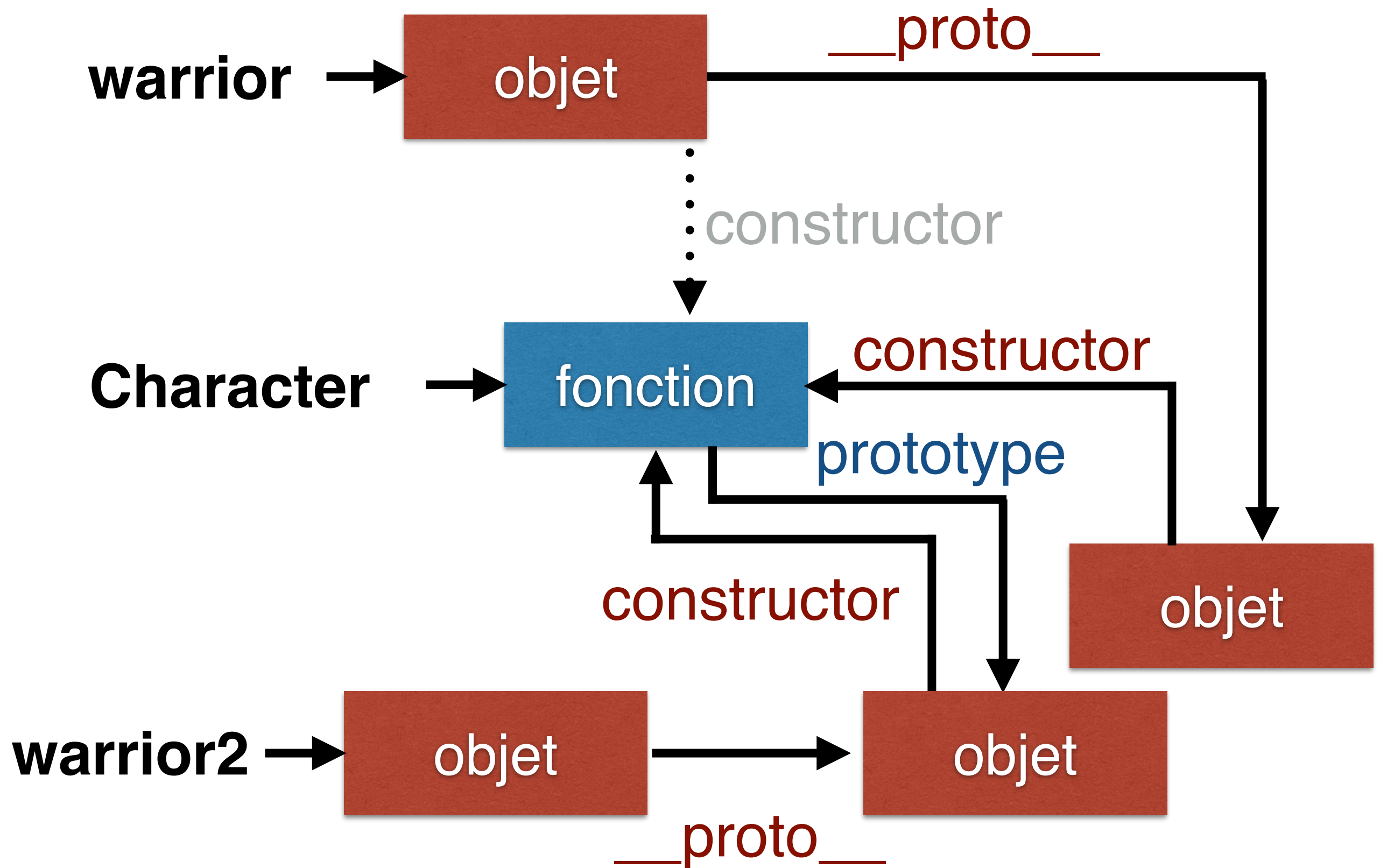
Tout est dynamique

- Possible de redéfinir le prototype d'une fonction à tout moment

```
1  const Character = function() {}
2  Character.prototype.hp = 100
3  Character.prototype.armor = 10
4
5  warrior = new Character()
6
7  Character.prototype = {
8    hp: 150,
9    shout: function() {
10     return "FUS RO DAH !"
11   }
12 }
13
14 warrior2 = new Character()
15
```







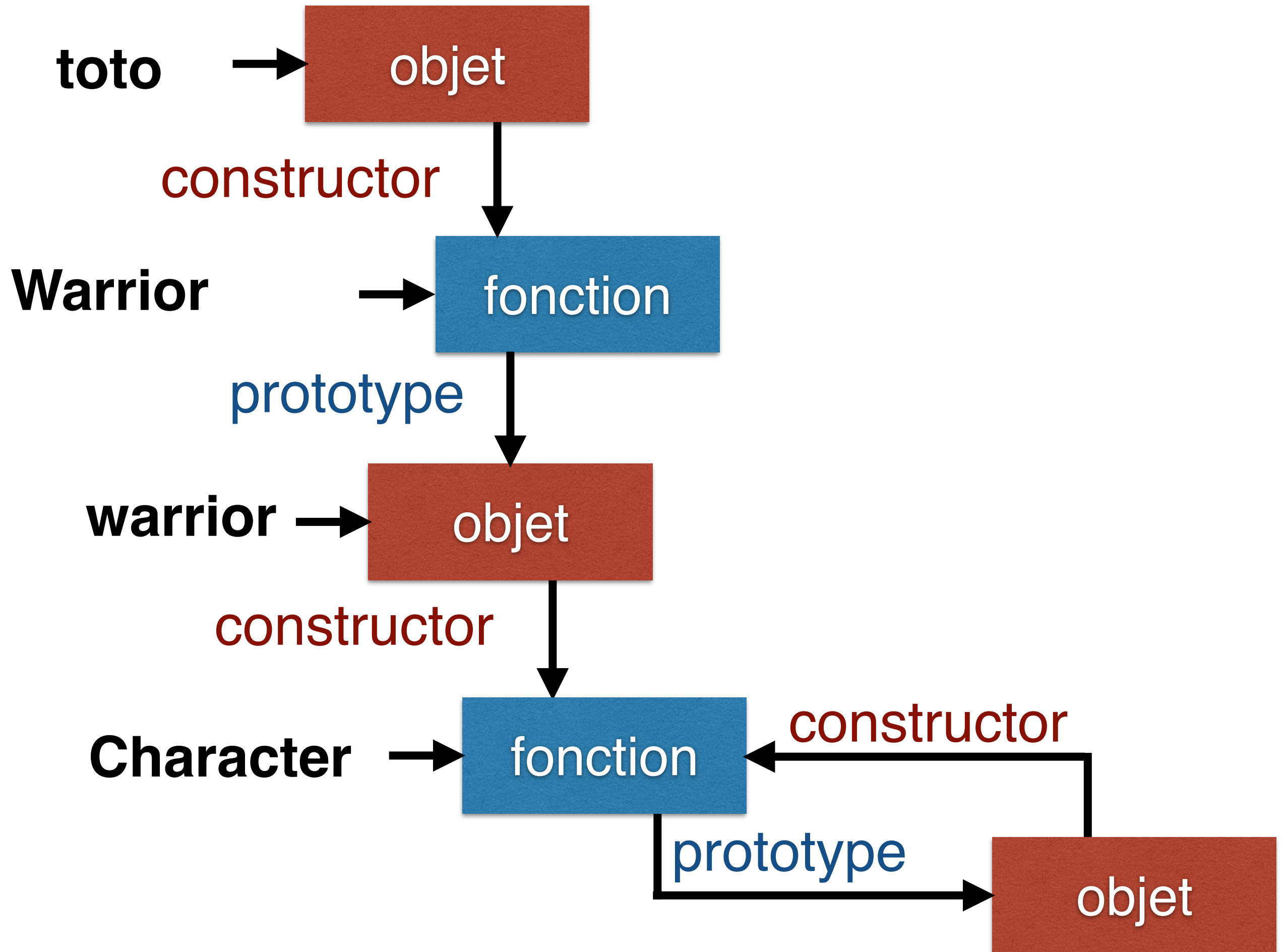
```
1  const Character = function() {}
2  Character.prototype.hp = 100
3  Character.prototype.armor = 10
4
5  warrior = new Character()
6
7  Character.prototype = {
8    hp: 150,
9    shout: function() {
10      return "FUS RO DAH !"
11    }
12  }
13
14  warrior2 = new Character()
15
16  warrior.hp ..... // ?
17  warrior.armor ..... // ?
18  warrior.shout() ..... // ?
19
20  warrior2.hp ..... // ?
21  warrior2.armor ..... // ?
22  warrior2.shout() ..... // ?
23
```

Jamais 2 sans 3

Récupérez la partie 3 à partir du repo distant
classroom

A retenir

- Modèle « simple »
 - constructor -> function
 - constructor.prototype -> object
- NE PAS UTILISER “constructor”
 - Savoir qu’il pointe le plus souvent un cran trop haut dans la hiérarchie
 - Garder une référence vers le prototype



Modification manuelle du prototype

- Objets conservent une référence vers prototype
 - modif du prototype = modif des objets
- Utilisé pour ajouter des fonctionnalités à tous les objets : **Polyfill**
 - Code implémentant une fonction d'une version supérieure

The `bind` function is a recent addition to ECMA-262, 5th edition; as such it may not be present in all browsers. You can partially work around this by inserting the following code at the beginning of your scripts, allowing use of much of the functionality of `bind()` in implementations that do not natively support it.

```
1  if (!Function.prototype.bind) {
2    Function.prototype.bind = function (oThis) {
3      if (typeof this !== "function") {
4        // closest thing possible to the ECMAScript 5 internal IsCallable function
5        throw new TypeError("Function.prototype.bind - what is trying to be bound is not callable");
6      }
7
8      var aArgs = Array.prototype.slice.call(arguments, 1),
9          fToBind = this,
10         fNOP = function () {},
11         fBound = function () {
12           return fToBind.apply(this instanceof fNOP && oThis
13                                ? this
14                                : oThis,
15                                aArgs.concat(Array.prototype.slice.call(arguments)));
16         };
17
18      fNOP.prototype = this.prototype;
19      fBound.prototype = new fNOP();
20
21      return fBound;
22    };
23  }
```

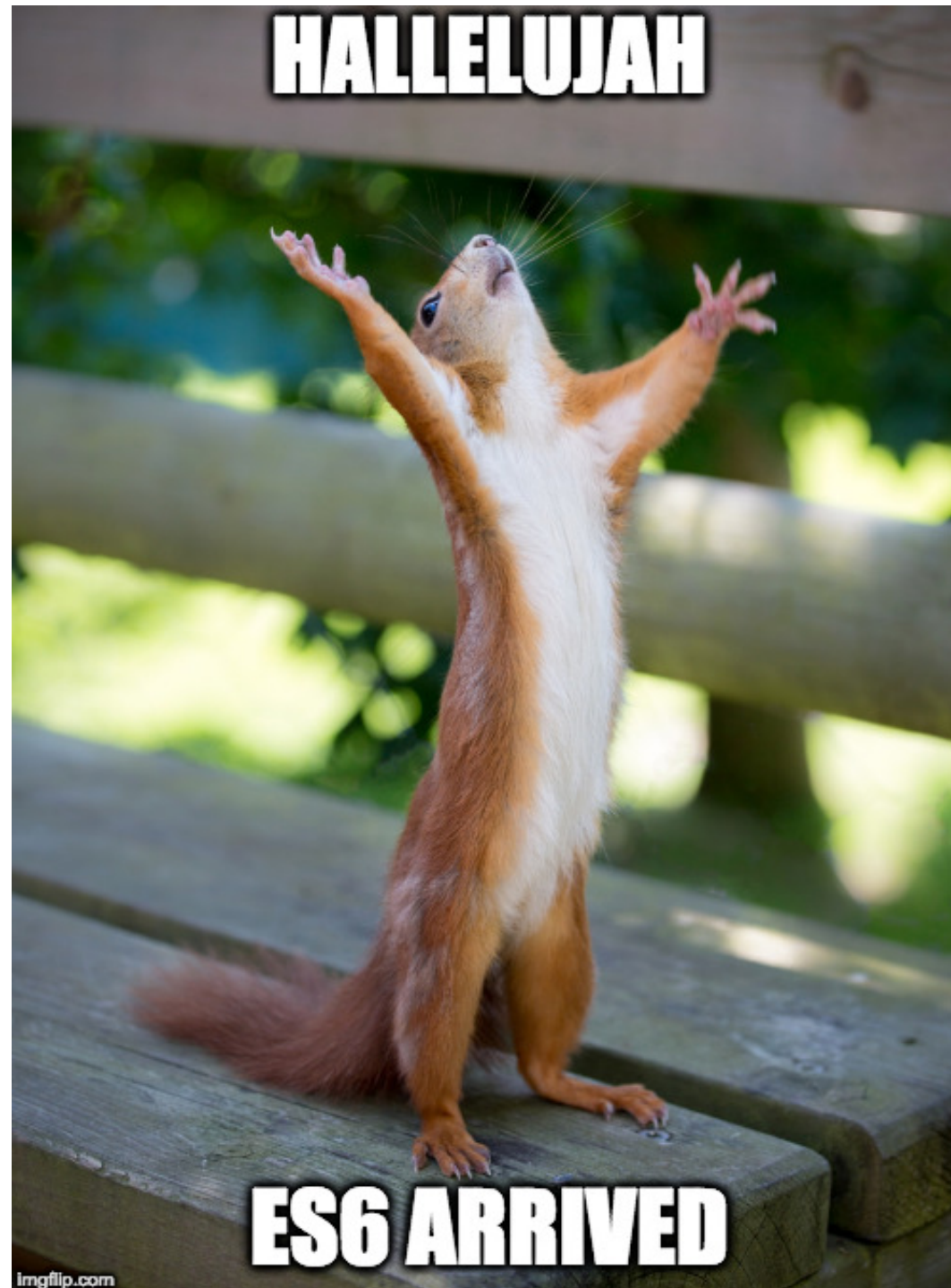
Attention !!!

- Modifier **Object**
 - Mauvaise idée
 - trop d'effet de bord
- Modifier **Array**
 - effet de bord sur `length`
 - effet de bord sur *`for ... in`*

Résumé

- Tout **objet** a une propriété '**constructor**'
 - type : **function**
 - Ne pas utiliser 'constructor'
- Toute **fonction** a une propriété '**prototype**'
 - type : **object**
- Recherche de propriété : chaine des prototypes

Mais ça c'était avant



Introduction du paradigme à classe

- Ajout de fonctions particulières: **class**

Exemple

```
1  class Character {
2      ... constructor() {
3          ... this.hp = 150
4          ... }
5      ...
6      ... shout() {
7          ... return "FUS RO DAH !"
8          ... }
9  }
10
11  warrior = new Character()
12  warrior.damage = 42
13  warrior.armor = 50
14
```

class : mots clefs

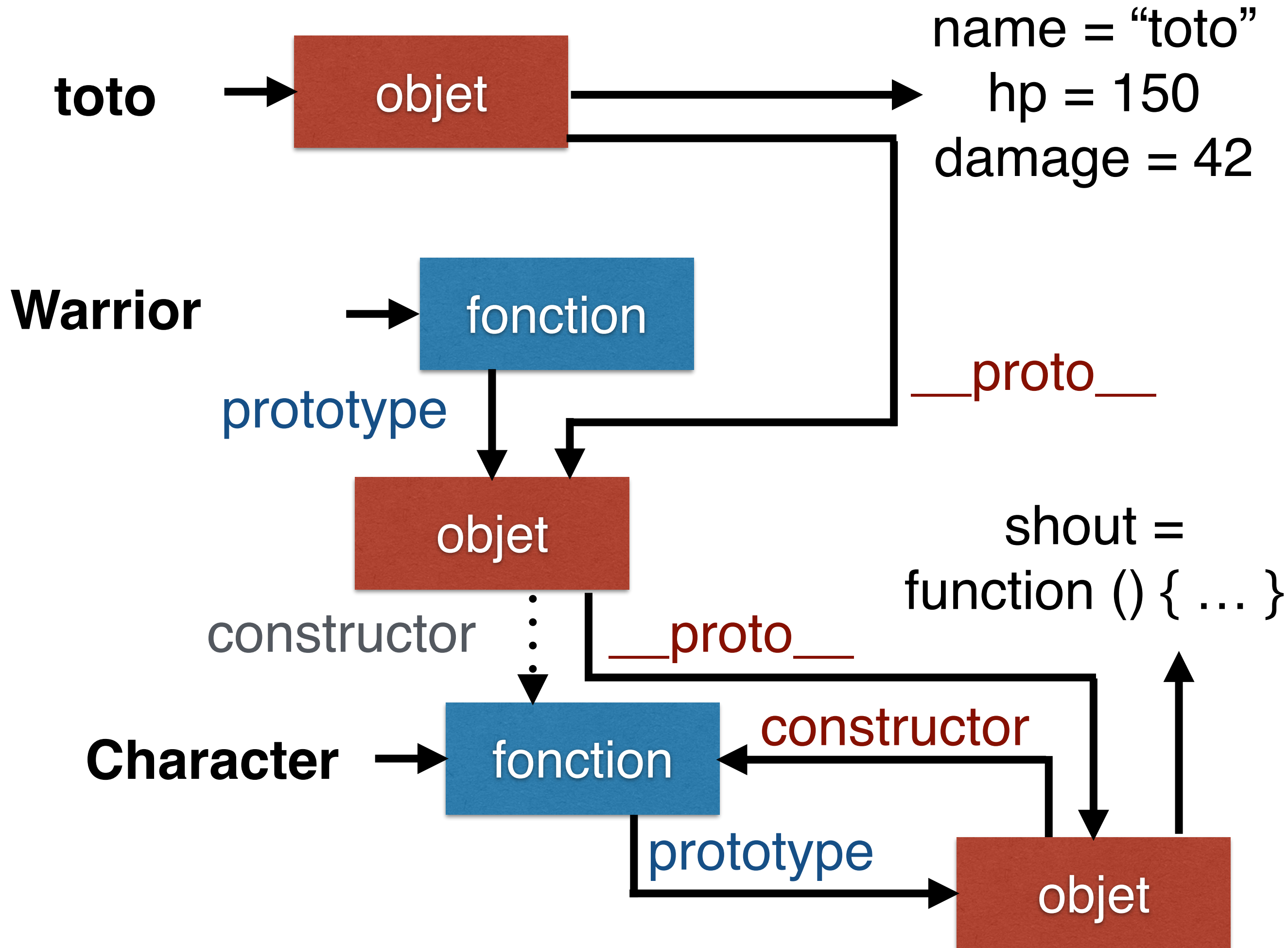
- **class**
 - Permet de définir une classe
- **constructor()**
 - Définit la fonction servant de constructor
- **extend**
 - Héritage de classes
- **super()**
 - Appel au constructor de la classe mère

Exemple

```
1  class Character {
2      ... constructor (hp) {
3          ... this.hp = hp
4      ... }
5      ...
6      ... shout () {
7          ... return "FUS RO DAH !"
8      ... }
9  }
10
11 class Warrior extends Character {
12     ... constructor (name, damage) {
13         ... super(150)
14         ... this.name = name
15         ... this.damage = damage
16     ... }
17 }
18
19 toto = new Warrior()
20
21 toto.hp ..... // ?
22 toto.name ..... // ?
23 toto.damage ..... // ?
24 toto.shout() ..... // ?
25
```

Poudre de perlimpinpin

- En fait, juste “sucre syntaxique”
 - Plus facile
 - Plus lisible
- Dissimule le paradigme à prototype
 - Génère tout le code qui va bien automatiquement



Warning

- Les **class** ne sont *hoisted* (hissées)

```
1  const warrior = new Character(100) // Reference error
2
3  class Character {
4      constructor (hp) {
5          this.hp = hp
6      }
7
8      shout () {
9          return "FUS RO DAH !"
10     }
11 }
12
```

Conclusion

- Intéressant de comprendre le fonctionnement du paradigme à prototype...
- ... mais dorénavant, utiliser le “sucre syntaxique” du paradigme à classe
 - Plus propre, plus lisible, moins error prone

Une dernière série d'exos pour la route ?

Récupérez la partie 4 à partir du repo distant
classroom