Javascript
#02

Matthieu Nicolas
Licence Pro CIASIE

Plan

- Programmation orientée objet

Programmation orientée objet

Javascript
#02

Programmation orientée prototype

- Différent du modèle à classes qui
  - Repose sur des des classes
  - Définies de façon statique
  - Patron de conception d’un objet
  - Tout objet est instance d’une classe

Programmation orientée prototype

- Repose sur la notion de prototype
  - Un objet comme un autre
  - Possède des propriétés, des méthodes
  - Peut être modifié

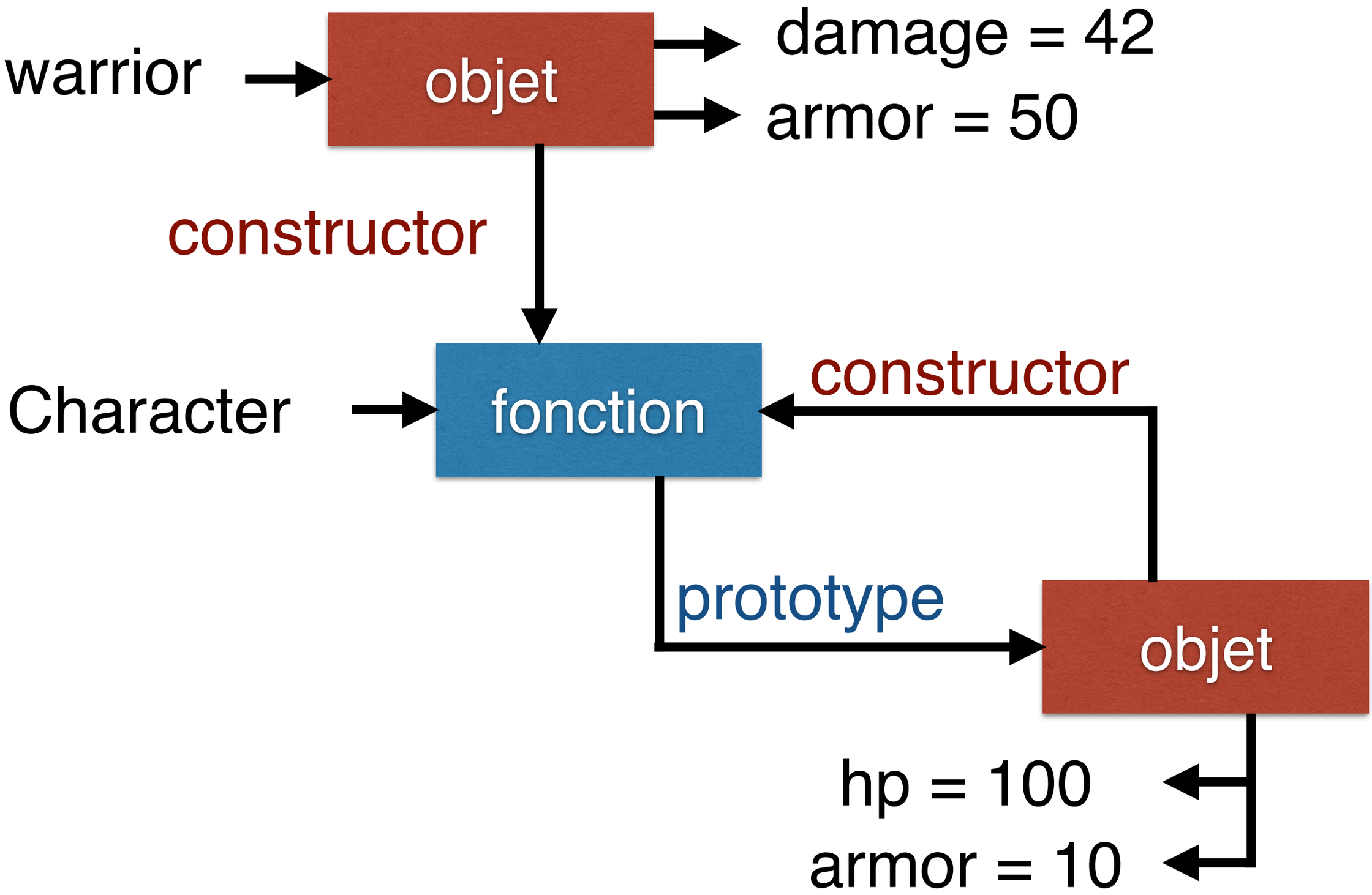
- Crée des objets à partir de ce prototype

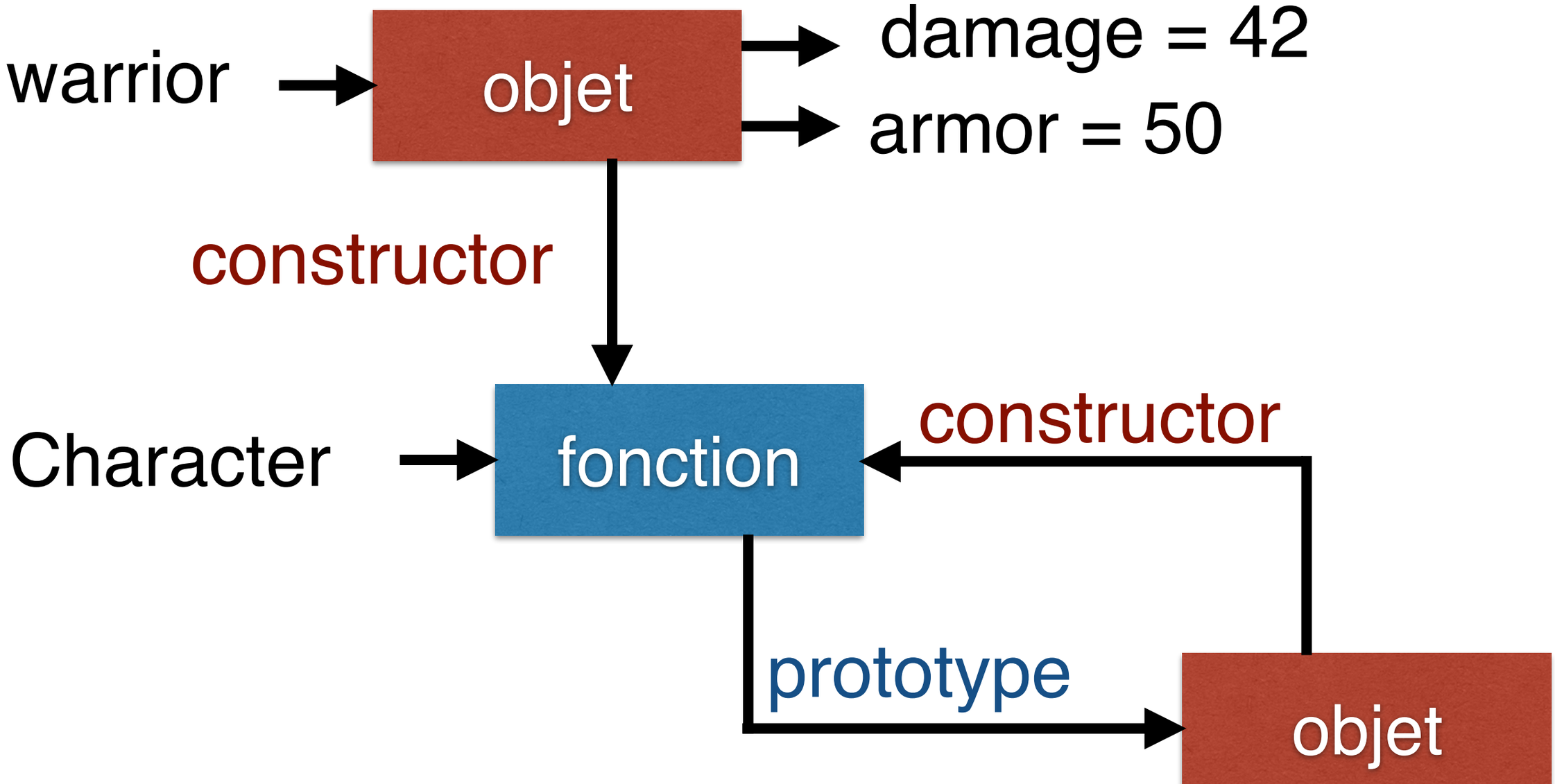
Orienté objet en JS

- Toute fonction peut être utilisée comme **constructeur**
  - À l'aide du mot clé **new**
- Toute fonction possède une propriété **prototype**
  - Référence un objet initialement vide
  - Peut y ajouter des propriétés et méthodes
  - Seront partagées par tous les objets construits à l'aide de la fonction

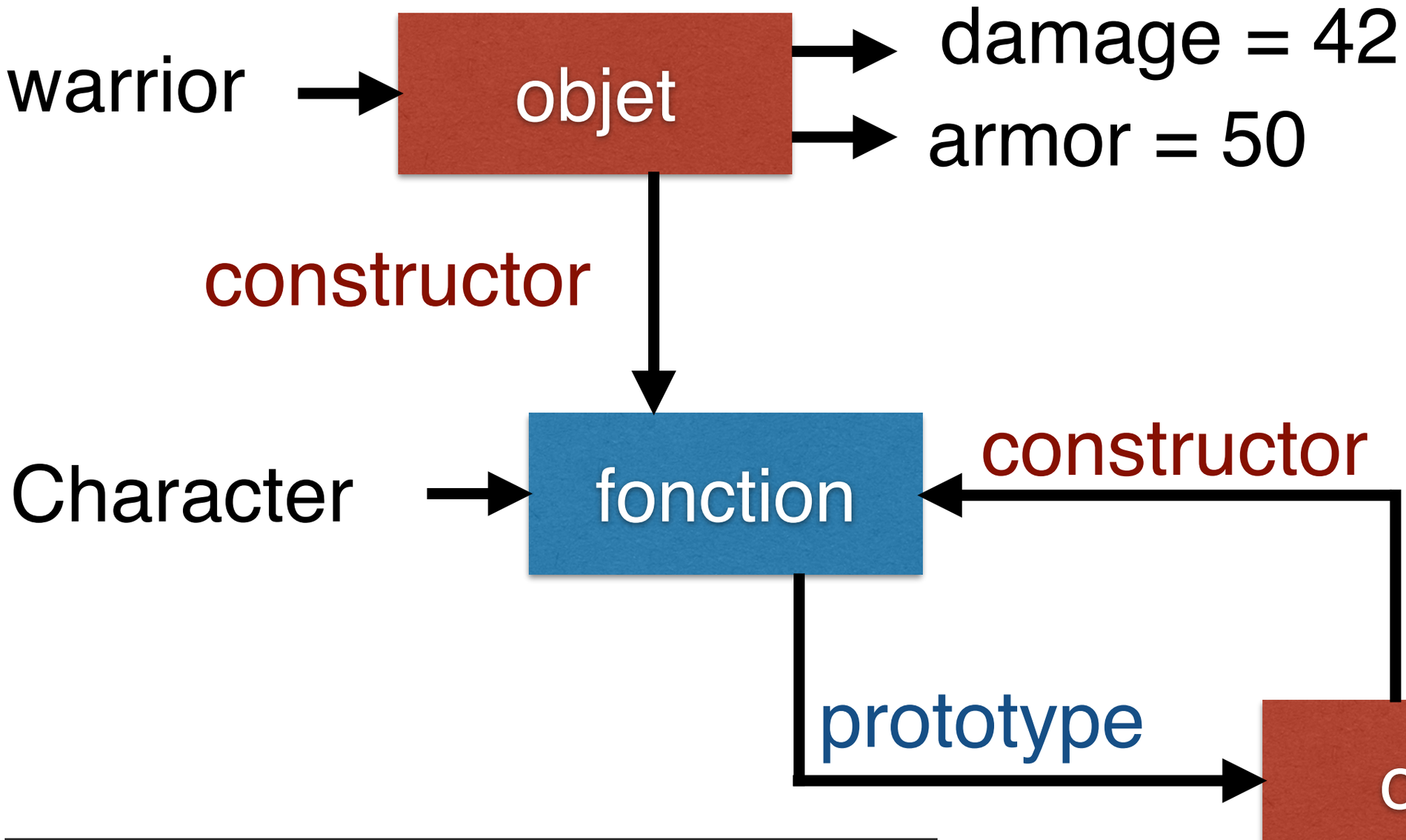
Exemple

```
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
```


warrior
objet
damage = 42
armor = 50
constructor
Character
fonction
constructor
prototype
objet
hp = 100
armor = 10




```
1   warrior.damage·····//·?
2   warrior.hp·········//·?
3   warrior.armor·····//·?
4   warrior.mp·········//·?
5
```




```
1   warrior.damage·····//·42
2   warrior.hp·········//·100
3   warrior.armor·····//·50
4   warrior.mp·········//·undefined
5
```

Prototype vs. Classe

- Plus puissant que le modèle à classes
  - Peut ré-implémenter le modèle à classes à l'aide de prototypes
  - Plus flexible
    - Possible de changer le prototype d'un objet à la volée
    - Possible de modifier un prototype directement
    - Possible de recomposer la chaîne de prototypes à l'exécution

Défauts de l’orienté objet en JS

- Complexe, verbeux, lourd…
- Paradigme à base de prototypes étranger à bon nombre de développeurs…

Introduction du paradigme à classes

- Introduit par ES6

- Ajout de fonctions particulières: class

class : mots clefs

- class
  - Une fonction particulière
  - Permet de définir une classe
- constructor()
  - Définit la fonction servant de constructeur
  - Attention: seulement 1 seule fonction constructor() dans le corps d'une classe
    - Déclenche une erreur sinon

Exemple

```
class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
    }

    shout () {
        return "FUS RO DAH!"
    }
}

const warrior = new Character("toto", 100)
warrior.damage = 42
warrior.armor = 50
```

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)

```
const warrior = new Character("toto", 100) // Reference error

class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
    }

    shout () {
        return "FUS RO DAH!"
    }
}
```

Propriétés de classe

- Possible d’ajouter des propriétés à une classe
- Correspond à des propriétés “static”

```
class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
        Character.NB_CHARACTERS++
    }

    shout () {
        return "FUS RO DAH!"
    }
}
Character.NB_CHARACTERS = 0

console.log(Character.NB_CHARACTERS)    // 0
const warrior = new Character("toto", 100)
console.log(Character.NB_CHARACTERS)    // ?

console.log(warrior.NB_CHARACTERS)    // ?
```

Méthodes de classe

- Possible de définir des méthodes static

```
class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
        Character.NB_CHARACTERS++
    }

    static fromJSON (json) {
        const plainCharacter = JSON.parse(json)
        return new Character(plainCharacter.name, plainCharacter.maxHP)
    }

    shout () {
        return "FUS RO DAH!"
    }
}
Character.NB_CHARACTERS = 0

const warrior = Character.fromJSON('{"name":"toto","hp":100,"maxHP":100}')
```

getters / setters

- Possible de définir des getters/setters
- Permet un accès uniforme entre propriétés et méthodes

```
class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
    }

    get name () {
        return this._name
    }

    set name (name) {
        this._name = name.charAt(0).toUpperCase() + name.slice(1).toLowerCase()
    }
}

const warrior = new Character("toTo", 100)
```

Héritage en JS

- extends
  - Héritage de classes
- super()
  - Appel au constructor de la classe mère
- super.xxx()
  - Appel à la méthode xxx() de la classe mère

Exemple

```
class Character {
    constructor (name, hp) {
        this.name = name
        this.hp = hp
        this.maxHP = hp
    }

    shout () {
        return "FUS RO DAH!"
    }
}

class Warrior extends Character {
    constructor (name, damage) {
        super(name, 150)
        this.damage = damage
    }
}

const toto = new Warrior("toTo", 15)

toto.name       // ?
toto.hp         // ?
toto.damage     // ?
toto.shout()    // ?
```

Limites de l’orienté objet en JS

- Pas de classes abstraites
- Pas d’interfaces
- Pas de traits

Niveaux de visibilité

- Pas de niveaux de visibilité pour les propriétés et méthodes d'une "classe"
  - TOUT EST PUBLIC
- Proposition d'ajout du niveau "private"
  - Mais encore à un stade expérimental
  - Pas supporté par les navigateurs
  - Besoin de transpiler le code pour le rendre compatible

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

TD

https://classroom.github.com/a/z47KUzWS
(lien dispo sur Arche)

- Cloner le projet
- Suivre l’énoncé se trouvant dans le répertoire sujets/