

# *Chapitre 3 :*

## *Héritage*

- 2 notions importantes :
  - ❶ Rationaliser le code
  - ❷ Produire un code robuste
- Plan :
  - ❶ Héritage
  - ❷ Chaîne de prototypes en Javascript

# *Production de code*

---

- Réutilisation de code
- Factorisation d'attributs et de méthodes

*→ Comment faire lors de l'écriture d'une classe...*

# Classes Voiture et Camion

```
class Voiture {  
    constructor(num, marque, type) {  
        this.num=num;  
        this.marque=marque;  
        this.type=type;  
        this.vitesse=0;  
    }  
    accélérer(plus) {  
        this.vitesse+=plus;  
    }  
    ralentir(moins) {  
        this.vitesse=(this.vitesse -  
moins>=0)?this.vitesse-moins:0;  
    }  
}
```

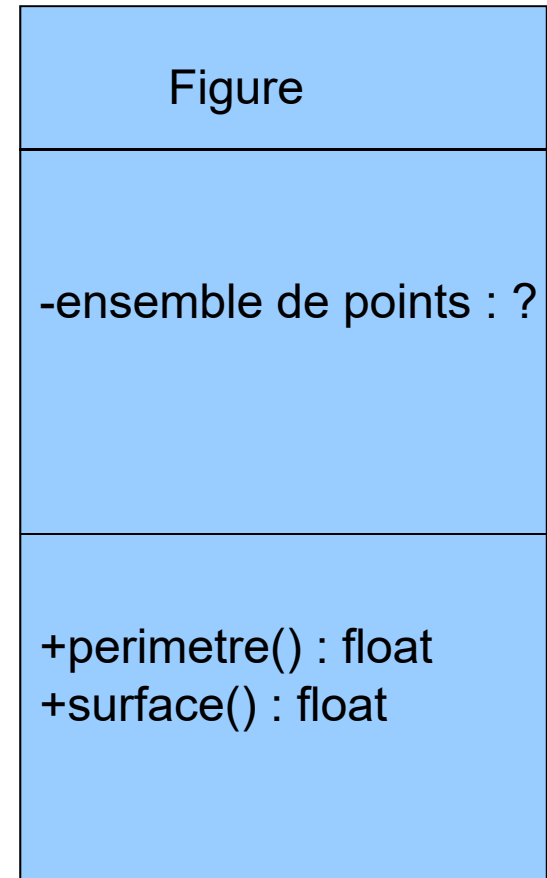
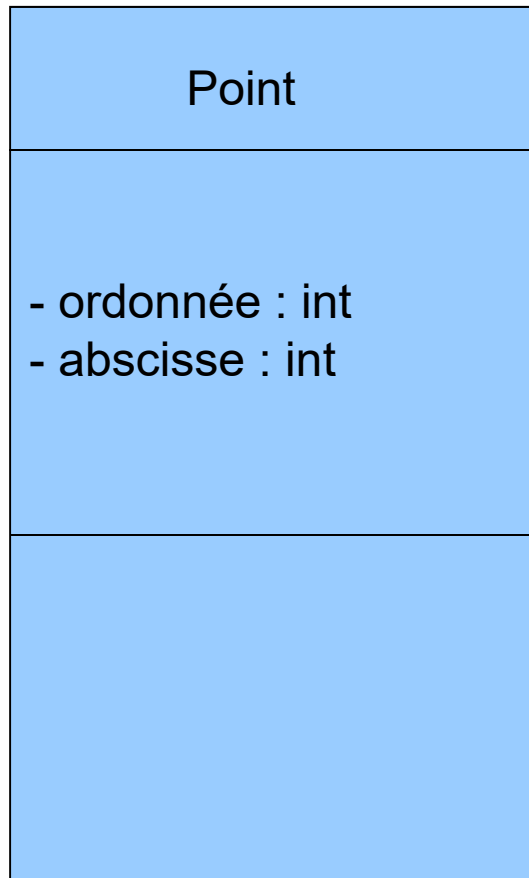
```
class Camion {  
    constructor(num, marque, chargeMax) {  
        this.num=num;  
        this.marque=marque;  
        this.chargeMax=chargeMax;  
        this.vitesse=0;  
        this.charge=0;  
    }  
    accélérer(plus) {  
        this.vitesse+=plus;  
    }  
    ralentir(moins) {  
        this.vitesse=(this.vitesse -  
moins >= 0)?this.vitesse-moins:0;  
    }  
    charger(poids) {  
        if (this.charge+poids<=this.chargeMax)  
            this.charge += poids;  
    }  
    decharger(poids) {  
        if (this.charge-poids>=0)  
            this.charge -=poids;  
    }  
}
```

# *Exemple les figures géométriques dans le plan*

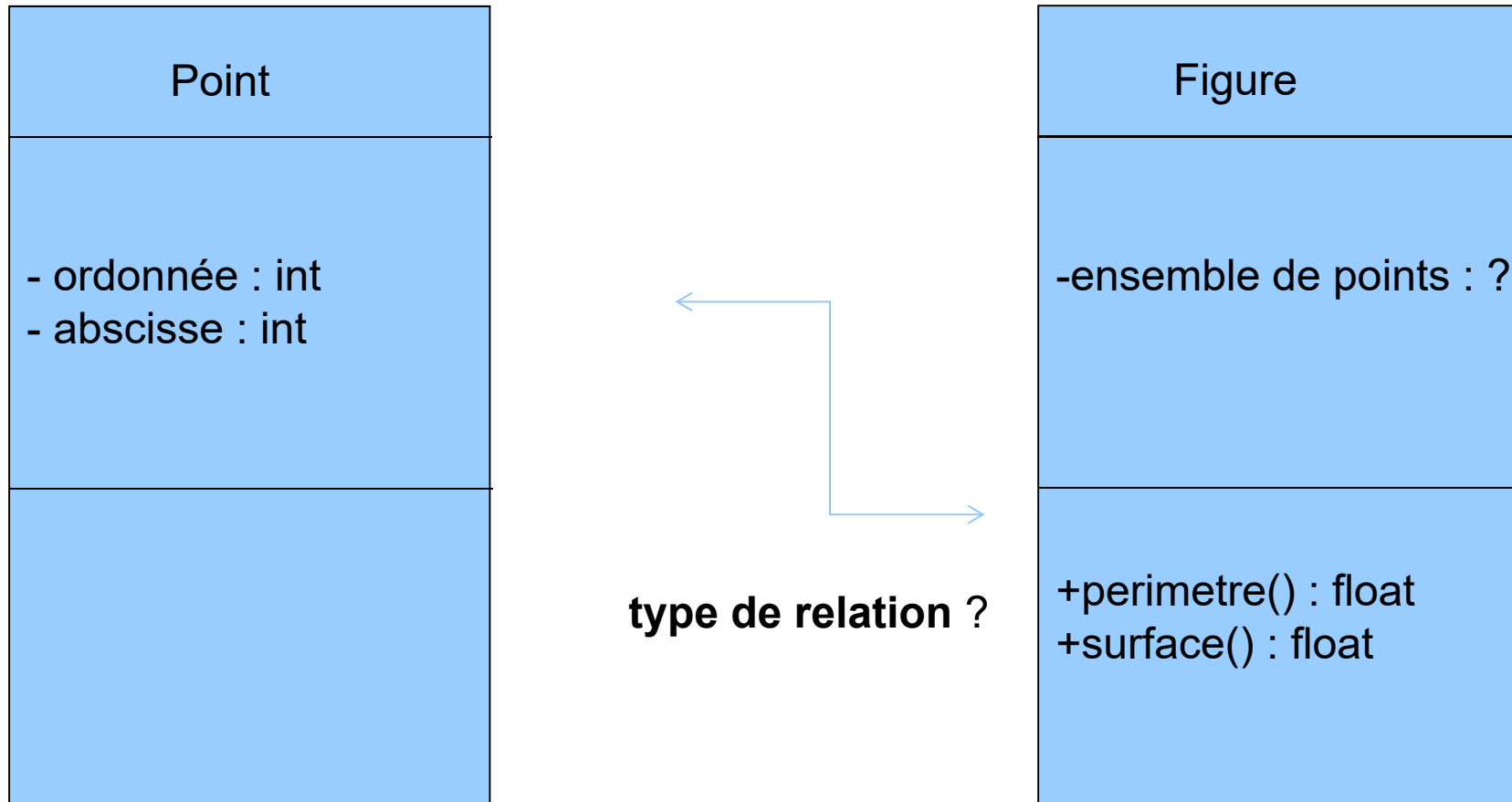
- « un point a une abscisse et une ordonnée »
- « une figure géométrique possède un certain nombre de points, une surface et un périmètre »

# Représentation des objets

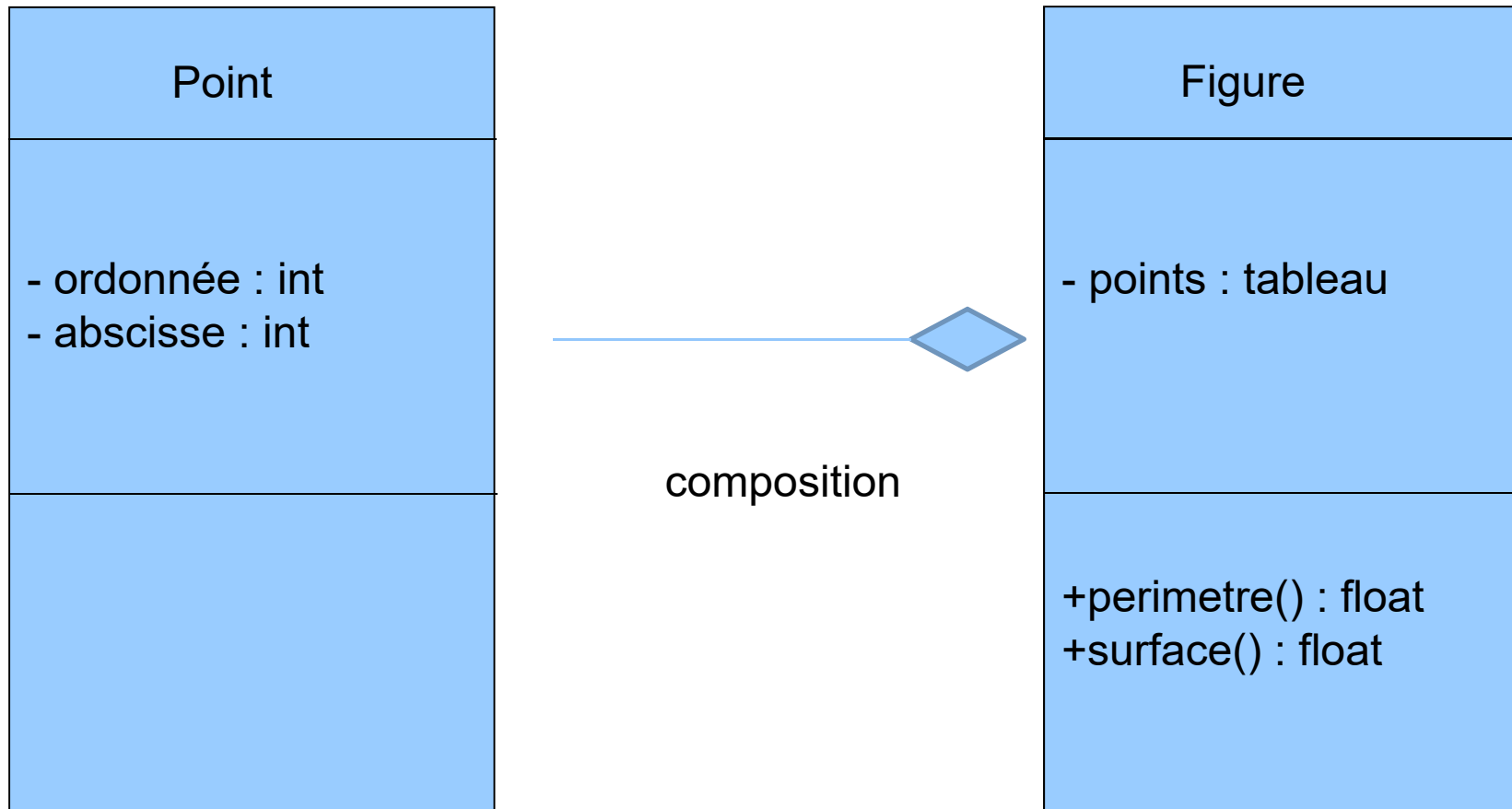
---



# Représentation des objets



# Représentation des objets



# *Représentation des objets*

---

- « Un **polygone** est une figure géométrique plane, formée d'une suite cyclique de segments consécutifs et délimitant une portion du plan. »



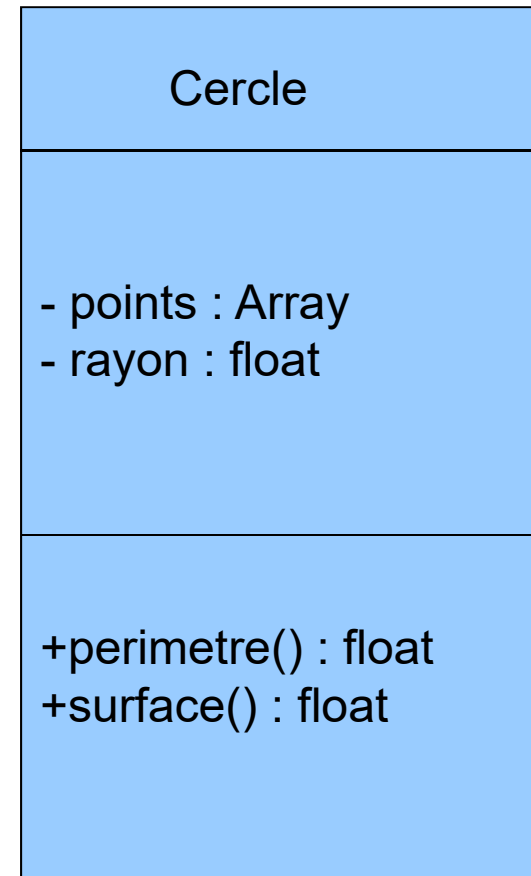
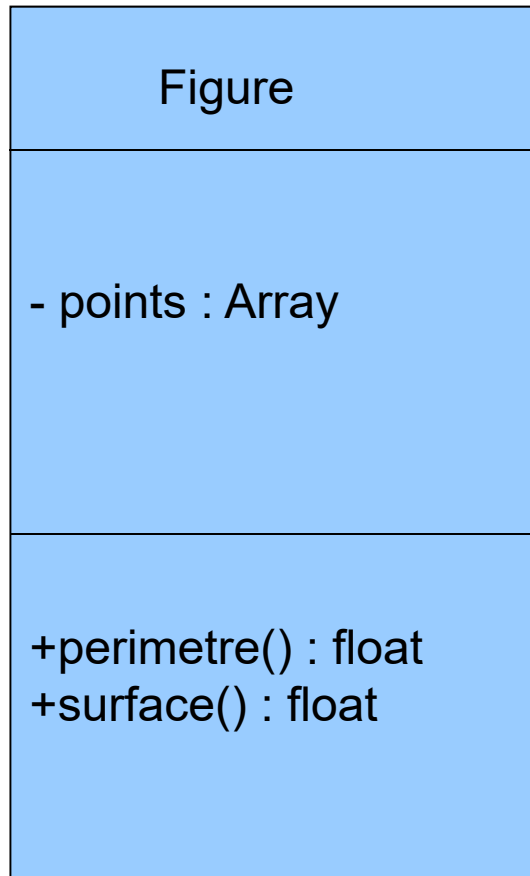
# *Représentation des objets*

---

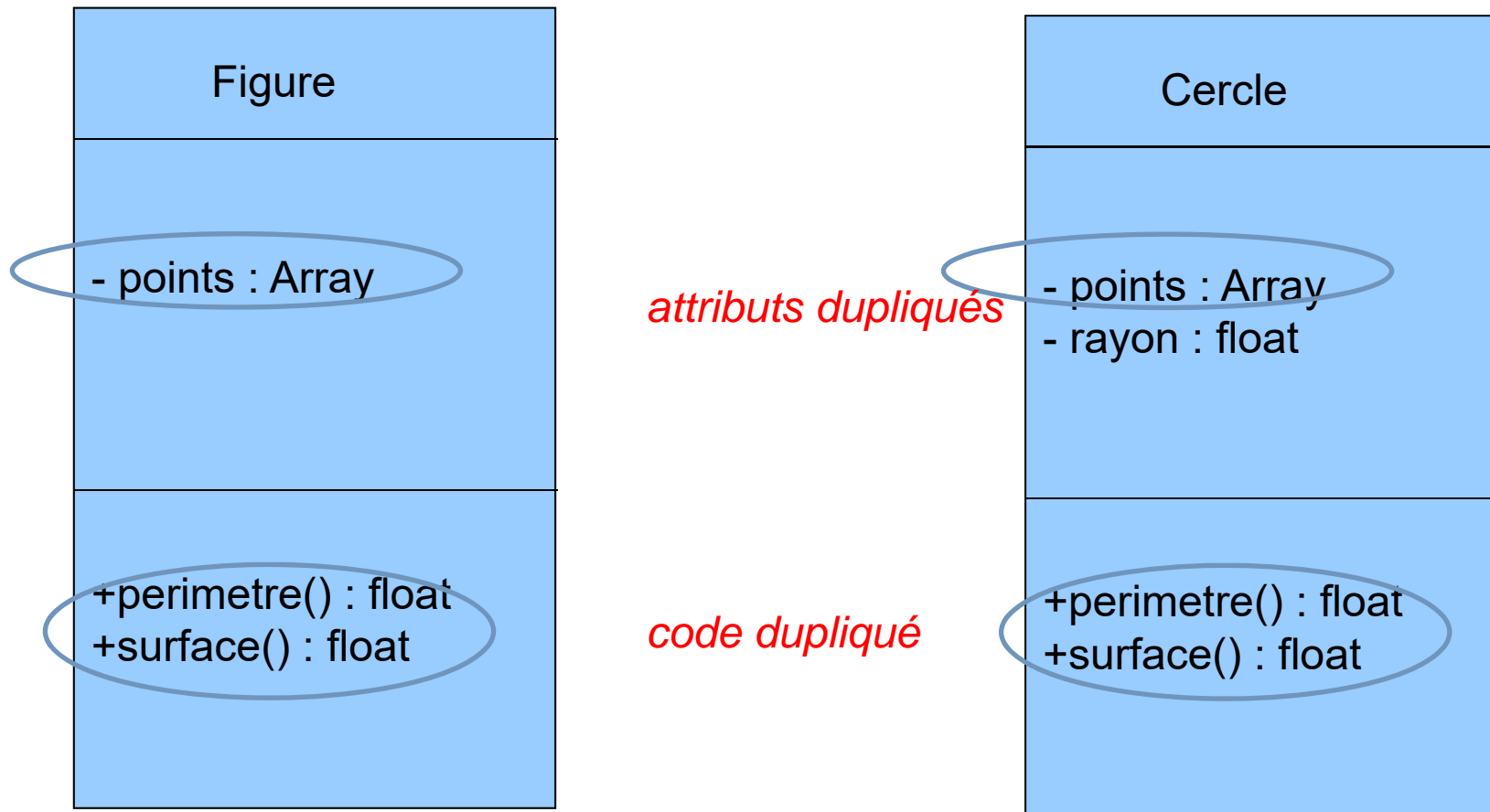
- « Un **cercle** est une courbe plane fermée constituée des points situés à égale distance d'un point nommé **centre**. La valeur de cette distance est appelée rayon du cercle. »
- Un cercle **est-un** objet de type Figure + spécialisation

# Représentation des objets

---



# Représentation des objets

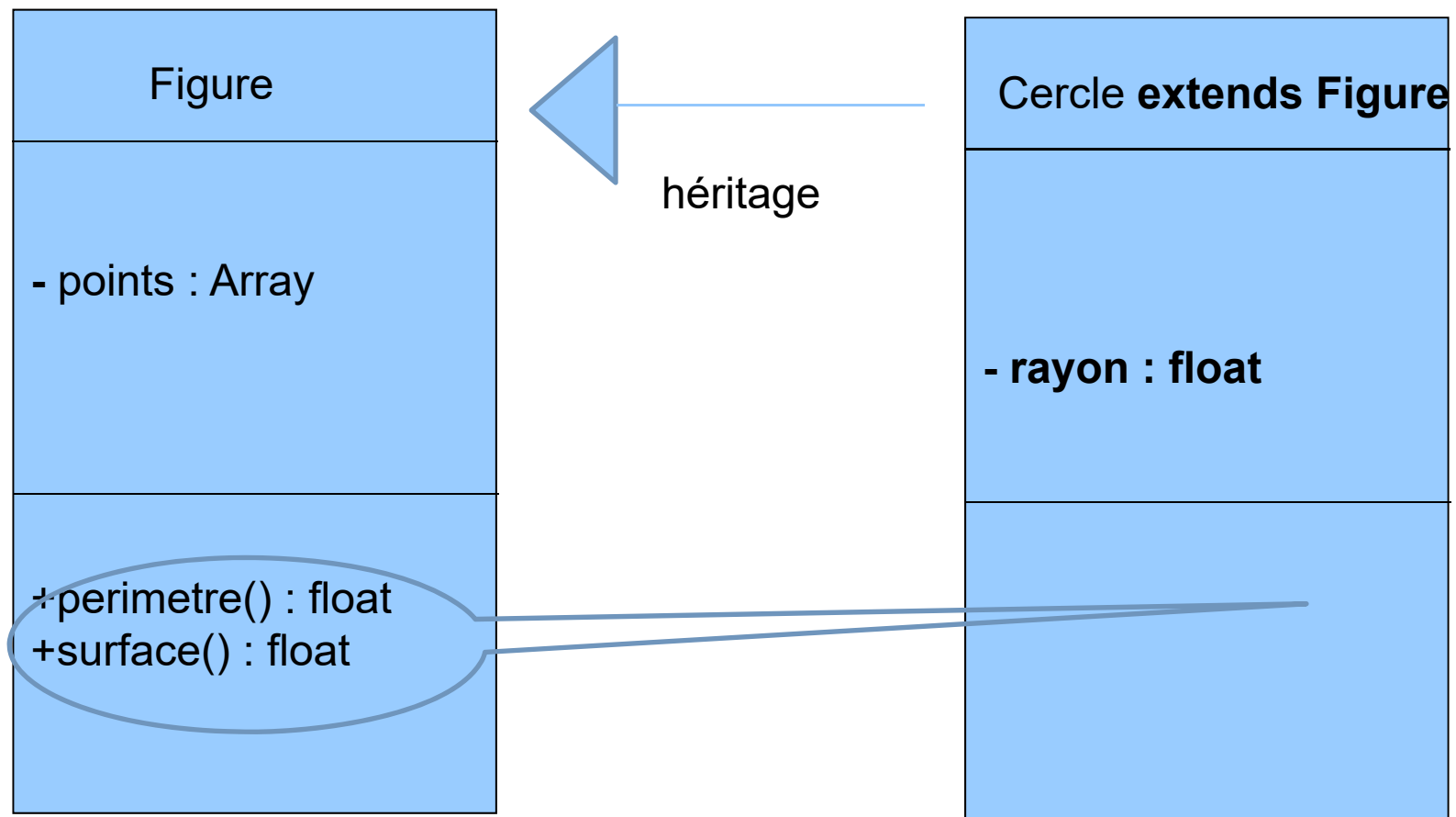


# *Mécanisme d'héritage*

---

- Quand une classe fille (sous-classe) hérite d'une classe mère (super-classe), elle possède tous les attributs et les méthodes non privés de la classe mère

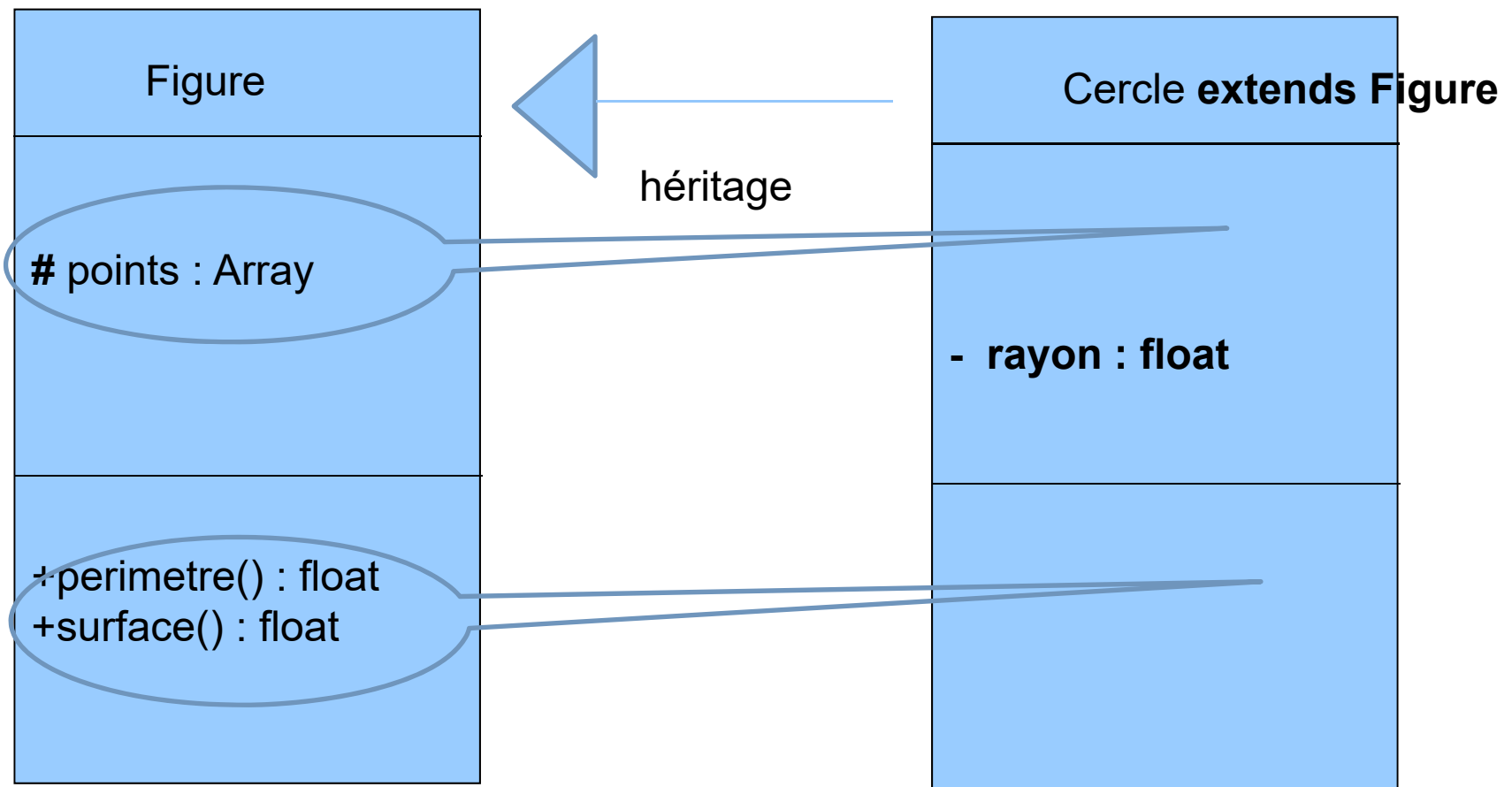
→ inutile de les écrire à nouveau !!!



# *Modificateur d'accès pour les attributs*

---

- Un attribut « protected » n'est visible que dans la classe qui le possède et dans les classes filles



# *Mécanisme d'héritage*

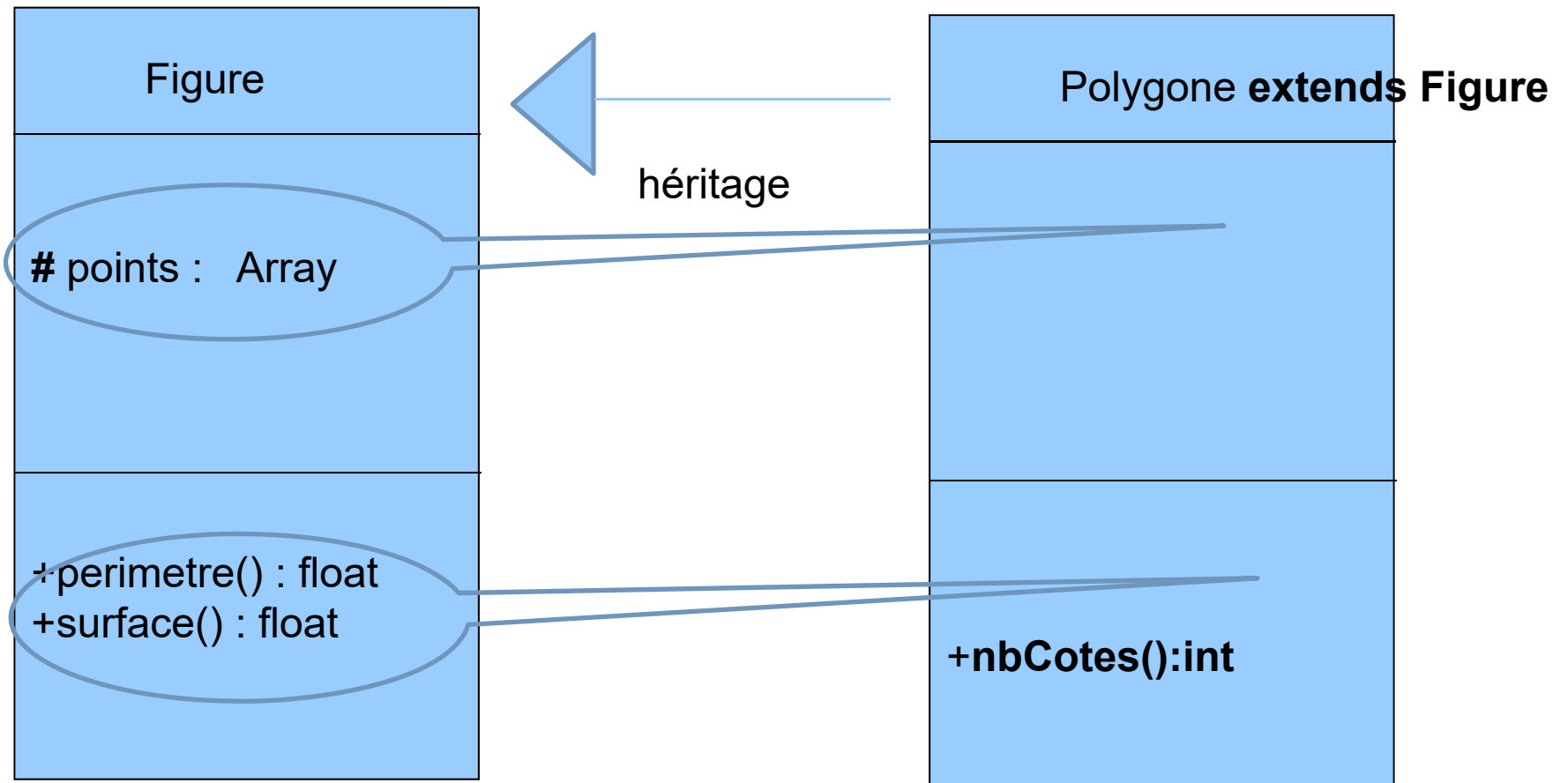
---

- Quand une classe fille (sous-classe) hérite d'une classe mère (super-classe), elle possède tous les attributs et les méthodes non privés de la classe mère

ET

- **On peut ajouter dans la classe fille les attributs et méthodes qui la spécialise**





# *Héritage en PHP*

---

- Mot-clé **extends** **ClasseMere**
- En java, une classe ne peut hériter que d'une autre classe.

→ impossible d'écrire

```
class A extends B , C
```

# Héritage en PHP

---

- Mot-clé **extends** **ClasseMere**
- En java, une classe ne peut hériter que d'une autre classe.

→ impossible d'écrire

~~class A extends B, C~~

→ class A extends B

## à retenir

---

- La composition est la traduction d'une relation « **a-un** »
- L'héritage est la traduction d'une relation « **est-un** »
- L'héritage permet la réutilisation de classes : une classe fille possède les attributs et méthodes non privés de sa classe mère
- La classe fille a des méthodes supplémentaires qui la spécialisent
- En PHP, le modificateur d'accès `protected` rend un attribut accessible par les classes filles

# Héritage en Javascript

---

- Attribut prototype :
  - tout **objet** a un attribut implicite `__proto__`
  - lorsqu'un objet est créé, il est lié au prototype de sa classe ( référencé par `__proto__`)
  - un prototype est un objet "patron" qu'un objet va pouvoir utiliser
  - le prototype contient les propriétés (attributs + méthodes) **partagées** par tous les objets d'une même classe

# *Exemple : la classe Mot*

---

Définition de la méthode constructeur :

```
Mot=function(valeur, type)
{
  this.valeur=valeur;
  this.lg=valeur.length;
  this.type=type;
}
```

- **Mot** est un objet de type **Function**
- Il récupère le prototype de **Function**
- **Mot.\_\_proto\_\_ = Function.prototype**

# Exemple : la classe Mot

L'attribut **prototype** de **Function** :

```
▼ Function: function Function()  
  arguments: null  
  caller: null  
  length: 1  
  name: "Function"  
▼ prototype: function ()  
  ► apply: function apply()  
    arguments: (...)  
  ► get arguments: function ThrowTypeError()  
  ► set arguments: function ThrowTypeError()  
  ► bind: function bind()  
  ► call: function call()  
    caller: (...)  
  ► get caller: function ThrowTypeError()  
  ► set caller: function ThrowTypeError()  
  ► constructor: function Function()
```

`Function.prototype.constructor()`

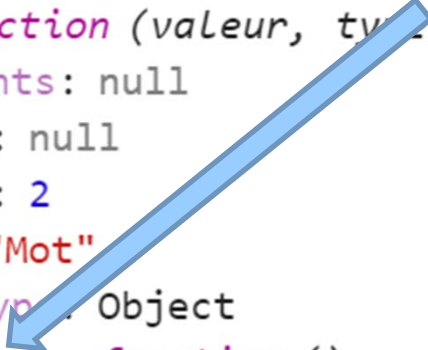
`Function.prototype.call()`

→ Appelle une fonction en précisant l'objet courant et les éventuels paramètres



# Exemple : la classe Mot

Le prototype de Mot :



```

▼ Function: function Function()
  arguments: null
  caller: null
  length: 1
  name: "Function"
  ▼ prototype: function ()
    ► apply: function apply(
      arguments: (...)
    ► get arguments: function
    ► set arguments: function
    ► bind: function bind()
    ► call: function call(
      caller: (...)
    ► get caller: function ThrowTypeError()
    ► set caller: function ThrowTypeError()
    ► constructor: function Function()
  ▼ Mot: function (valeur, type)
    arguments: null
    caller: null
    length: 2
    name: "Mot"
    ► prototype: Object
    ▼ __proto__: function ()
      ► apply: function apply(
        arguments: (...)
      ► get arguments: function ThrowTypeError()
      ► set arguments: function ThrowTypeError()
      ► bind: function bind()
      ► call: function call(
        caller: (...)
  
```

## *Exemple : la classe Mot*

---

Compléter le prototype de Mot:

```
Mot=function(valeur, type)
{
  this.valeur=valeur;
  this.lg=valeur.length;
  this.type=type;
}
```

```
Mot.prototype.convertirMin=function()
{...}
```


→ `convertirMin()` est ajouté à l'attribut `prototype`

# Exemple : la classe *Mot*

---

Le prototype de **Mot** :

```
▼ Mot: function (valeur, type)
  arguments: null
  caller: null
  length: 2
  name: "Mot"
  ▼ prototype: Object
    ► constructor: function (valeur, type)
    ► convertirMin: function ()
    ► __proto__: Object
    ► __proto__: function ()
    ► <function scope>
```



→ `convertirMin()` est commun à tous les objets de type **Mot**

## *Exemple : la classe Mot*

---

Passons maintenant à la création d'un mot :

```
let mot=new Mot("essai", "texte");
```

→ la zone mémoire pour l'objet est allouée

→ la méthode **Mot()** est exécutée

→ le prototype de **Mot** est récupéré

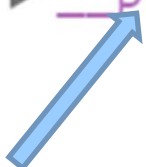
```
mot.__proto__ = Mot.prototype
```

→ l'adresse de l'objet créé est affecté à **mot**

# Exemple : la classe Mot

L'objet `mot` est lié au prototype de `Mot` :

```
▼ mot: Mot
  lg: 5
  type: "texte"
  valeur: "essai"
  ▼ __proto__: Object
    ► constructor: function (valeur, type)
    ► convertirMin: function ()
    ► __proto__: Object
```



→ Le prototype de `Mot` est aussi lié à un autre prototype = chaînage de prototypes

## C'est le prototype de Object

```
▼ mot: Mot
  lg: 5
  type: "texte"
  valeur: "essai"
  ▼ __proto__: Object
    ► constructor: function (valeur, type)
    ► convertirMin: function ()
    ▼ __proto__: Object
      ► __defineGetter__: function __defineGetter__
      ► __defineSetter__: function __defineSetter__
      ► __lookupGetter__: function __lookupGetter__
      ► __lookupSetter__: function __lookupSetter__
      ► constructor: function Object()
      ► hasOwnProperty: function hasOwnProperty()
      ► isPrototypeOf: function isPrototypeOf()
      ► propertyIsEnumerable: function propertyIsEn
      ► toLocaleString: function toLocaleString()
```

# *La classe Object*

---

`Object` est la classe de base

Le prototype d'`Object` est `null`

Constructeur : `function Object()`

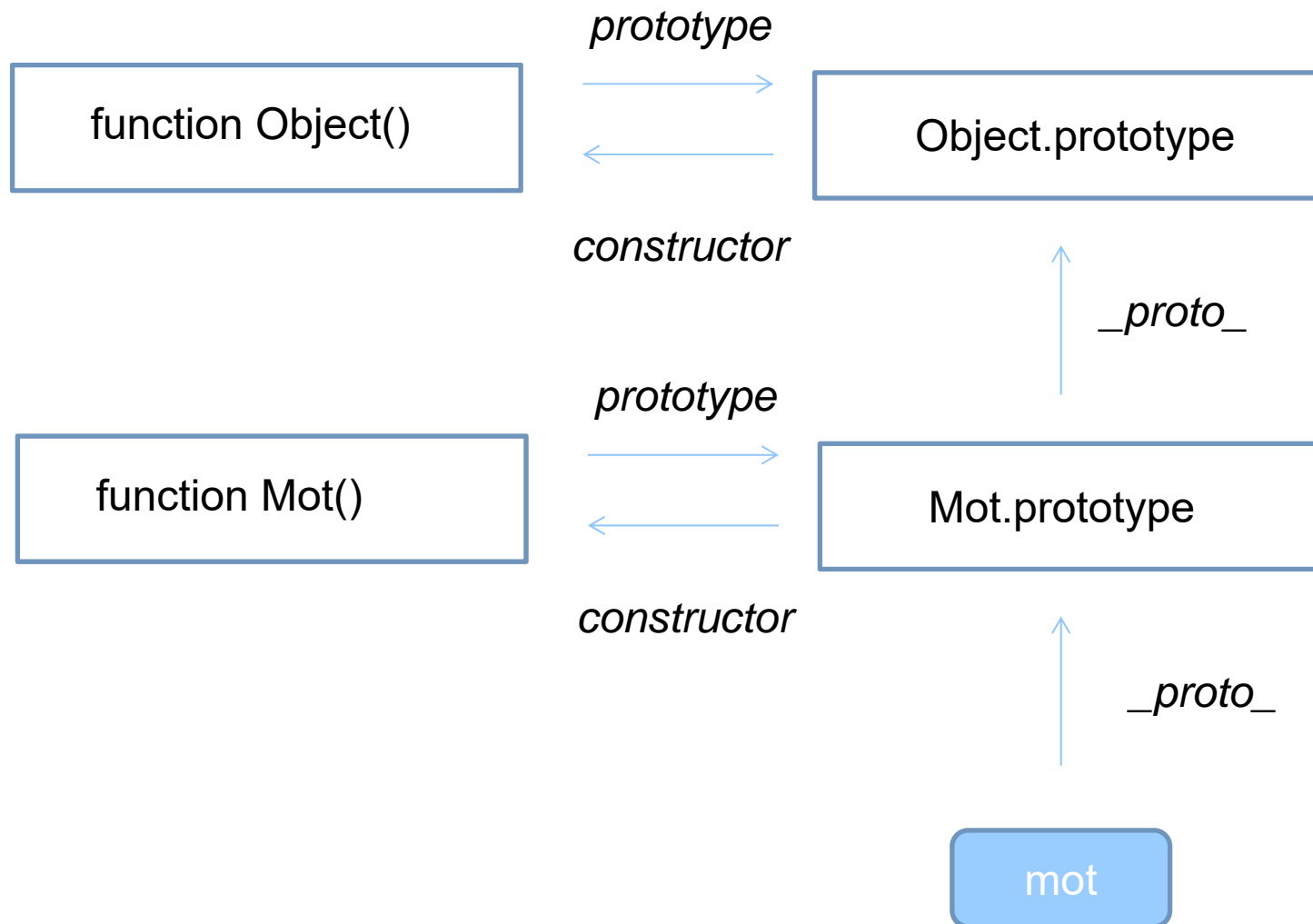
`Object.prototype.constructor`

Plusieurs méthodes :

`Object.prototype.isPrototypeOf()` : `true` si  
le paramètre est dans la chaîne des prototypes

`Object.create(proto)` : crée un nouvel objet avec  
le prototype en paramètre

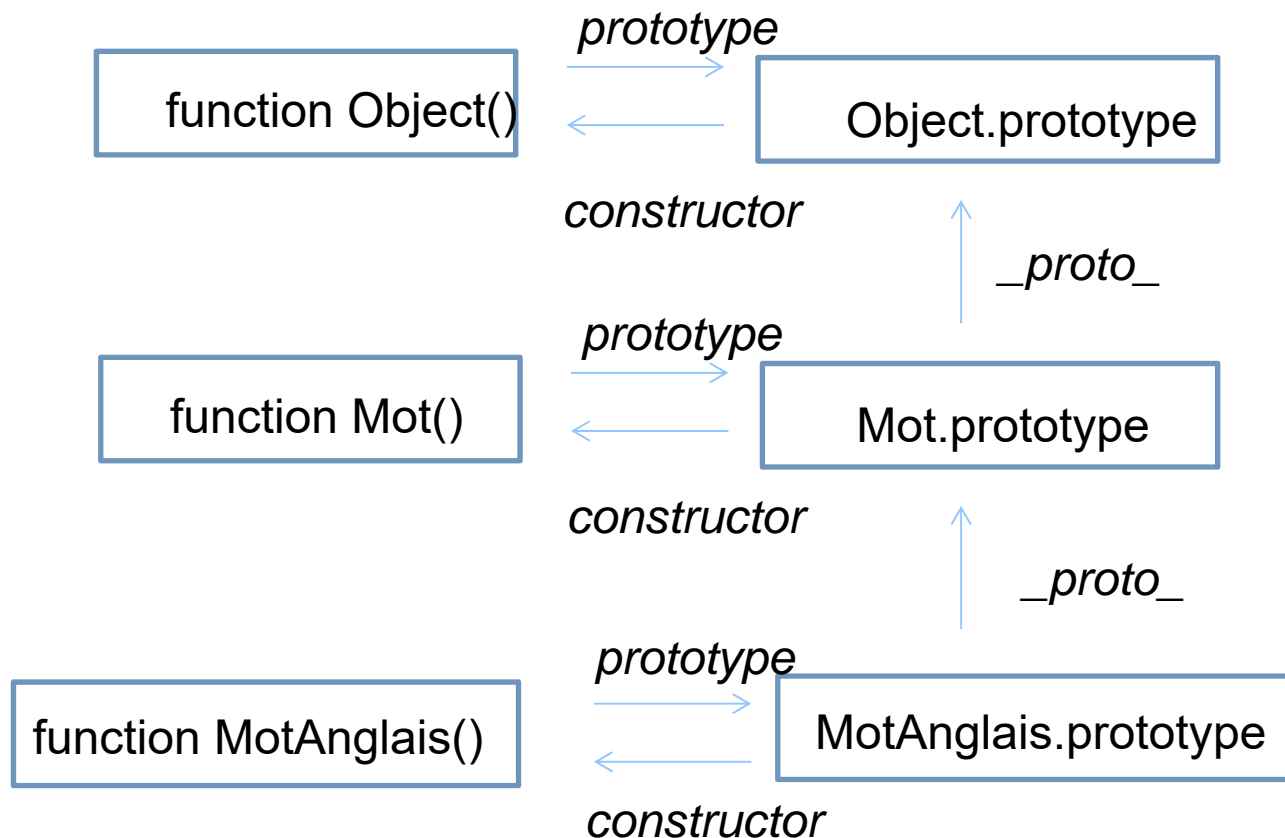
## ② Chaînage de prototypes





# Comment hériter d'une classe ?

- « un mot en anglais **est** un mot »



# Comment hériter d'une classe ?

---

- Écrire le constructeur de la classe fille  
« un mot en anglais est un mot »:

```
function MotAnglais(valeur, type, traduction)
{
  // appeler le constructeur de Mot
  ???
  // spécialiser l'objet
  this.traduction=traduction;
}
```

# Comment hériter d'une classe ?

---

- Écrire le constructeur de la classe fille  
« un mot en anglais est un mot »:

```
function MotAnglais(valeur, type, traduction)
{
  // appeler le constructeur de Mot
  this = new Mot(valeur, type);
  // spécialiser l'objet
  this.traduction=traduction;
}
```

# Comment hériter d'une classe ?

---

- Écrire le constructeur de la classe fille  
« un mot en anglais est un mot »:

```
function MotAnglais(valeur, type, traduction)
{
  // appeler le constructeur de Mot
  this = new Mot(valeur, type);
  // spécialiser l'objet
  this.traduction=traduction;
}
```

# Comment hériter d'une classe ?

---

- Écrire le constructeur de la classe fille  
« un mot en anglais est un mot »:

```
function MotAnglais(valeur, type, traduction)
{
  // appeler le constructeur de Mot
  Mot.call(this, valeur, type);
  // spécialiser l'objet
  this.traduction=traduction;
}
```

# Comment hériter d'une classe ?

---

```
▼ MotAnglais: function (valeur, type, traduction)
  arguments: null
  caller: null
  length: 3
  name: "MotAnglais"
▼ prototype: Object
  ► constructor: function (valeur, type, traduction)
  ► __proto__: Object
  ► __proto__: function ()
  ► <function scope>
```

# *Comment hériter d'une classe ?*


---

- Créer un nouveau prototype à partir du prototype « père »

```
MotAnglais.prototype =  
    Object.create(Mot.prototype);
```

# Quel est le problème ?

```
▼ MotAnglais: function (valeur, type, traduction)
  arguments: null
  caller: null
  length: 3
  name: "MotAnglais"
  ▼ prototype: Mot
    ▼ __proto__: Object
      ► constructor: function (valeur, type)
      ► convertirMin: function ()
      ► __proto__: Object
      ► __proto__: function ()
      ► <function scope>
```



→ Le constructeur est celui d'un objet Mot et non d'un objet MotAnglais



- Mettre le constructeur dans le prototype du nouvel objet

**MotAnglais.prototype.constructor=MotAnglais;**

```
▼ MotAnglais: function (valeur, type, traduction)
  arguments: null
  caller: null
  length: 3
  name: "MotAnglais"
▼ prototype: Mot
  ► constructor: function (valeur, type, traduction)
  ▼ __proto__: Object
    ► constructor: function (valeur, type)
    ► convertirMin: function ()
    ► __proto__: Object
  ► __proto__: function ()
  ► <function scope>
```

---

```
var n=new MotAnglais("no", "nom", "non");
```

```
n instanceof MotAnglais ; // true
```

```
n instanceof Mot ; // true
```

```
n instanceof Object ; // true
```

# *Vehicule – Voiture – Camion*

## *Route*

---

- Diagramme des classes ?
- Code du constructeur `vehicule` ?
- `Vehicule.__proto__` ?