# Les objets

IFT1015 — Philippe Langlais

# Plan

- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre première classe: Piece
- Rational
- Listes chaînées
- Les arbres

# I - Objets littéraux

```
var cours = {
    noDeCours: "IFT 1015",
    campus: "Montréal",
    crédits: 3,
    cycle: 1,
    département: "Informatique et rech. opér.",
    faculté: "Faculté des arts et des sciences"
    période: "sun.png",
    trimestre: "Hiver 2019",
    description: "Éléments de base d'un langage de
programmation : types, expressions, énoncés
conditionnels et itératifs, procédures, fonctions,
paramètres, récursivité, tableaux, enregistrements,
pointeurs et fichiers."
};
```

IFT 1015 - Programmation 1

| | |
|---|---|
| **No DE COURS** | IFT 1015 |
| **CAMPUS** | Montréal |
| **CRÉDITS** | 3.0 |
| **CYCLE** | 1er cycle |
| **DÉPARTEMENT** | Informatique et rech. opér. |
| **FACULTÉ** | Faculté des arts et des sciences |
| **PÉRIODE** | |
| **TRIMESTRE** | Hiver 2019, |

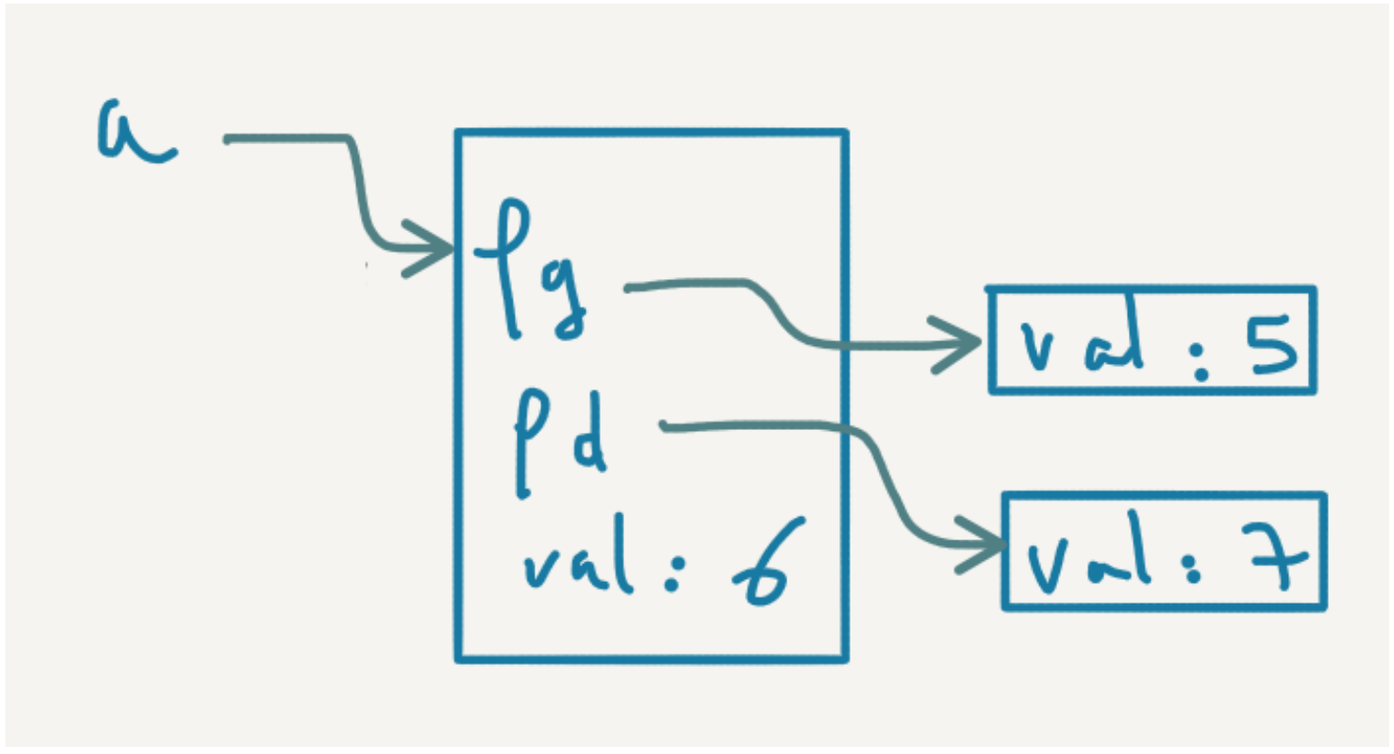
**DESCRIPTION**

Éléments de base d'un langage de programmation : types, expressions, énoncés conditionnels et itératifs, procédures, fonctions, paramètres, récursivité, tableaux, enregistrements, pointeurs et fichiers.

# Objet littéral

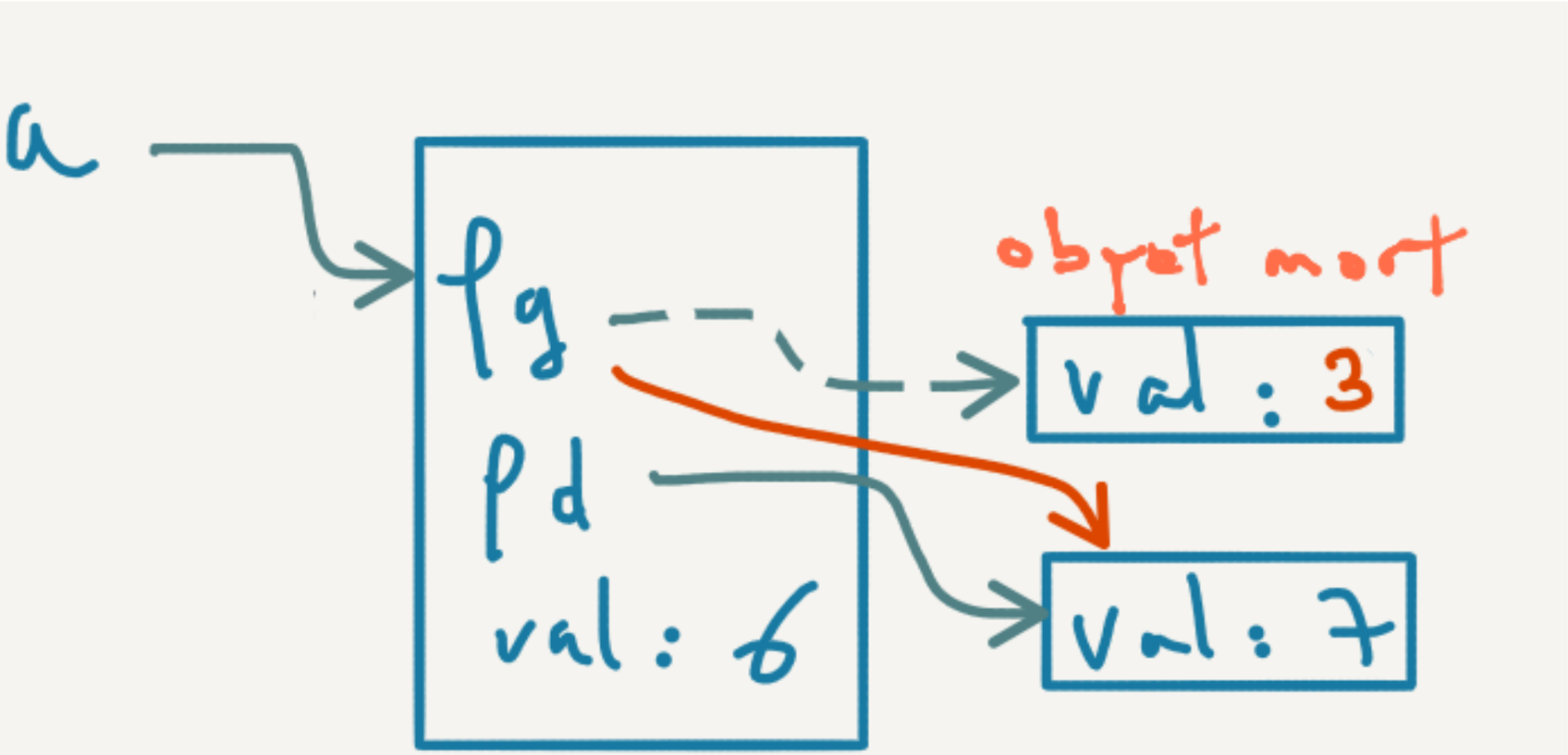
- collection de paires attributs (ou propriétés) / valeurs
- **lecture** d'une propriété
  - cours.crédits, cours["crédits"] sont des expressions qui valent 3
  - undefined si non définie (ex: cours.professeur)
- **écriture** d'une propriété
  - cours.cycle = "1er cycle";
  - cours.professeur = "Philippe Langlais"
- le **nom** d'un objet est son **adresse**

```
var a = {
        fg: {val:5},
        fd: {val:7},
        val: 6
};
```



# Objet littéral

```
var a = {
        fg: {val:5},
        fd: {val:7},
        val: 6
};

print(a.val); // 6
a.fg.val = 3;
print(a.fg.val); // 3
a.fg = a.fd;
print(a.fg.val); // ?
```




a.fg et a.fd sont des alias

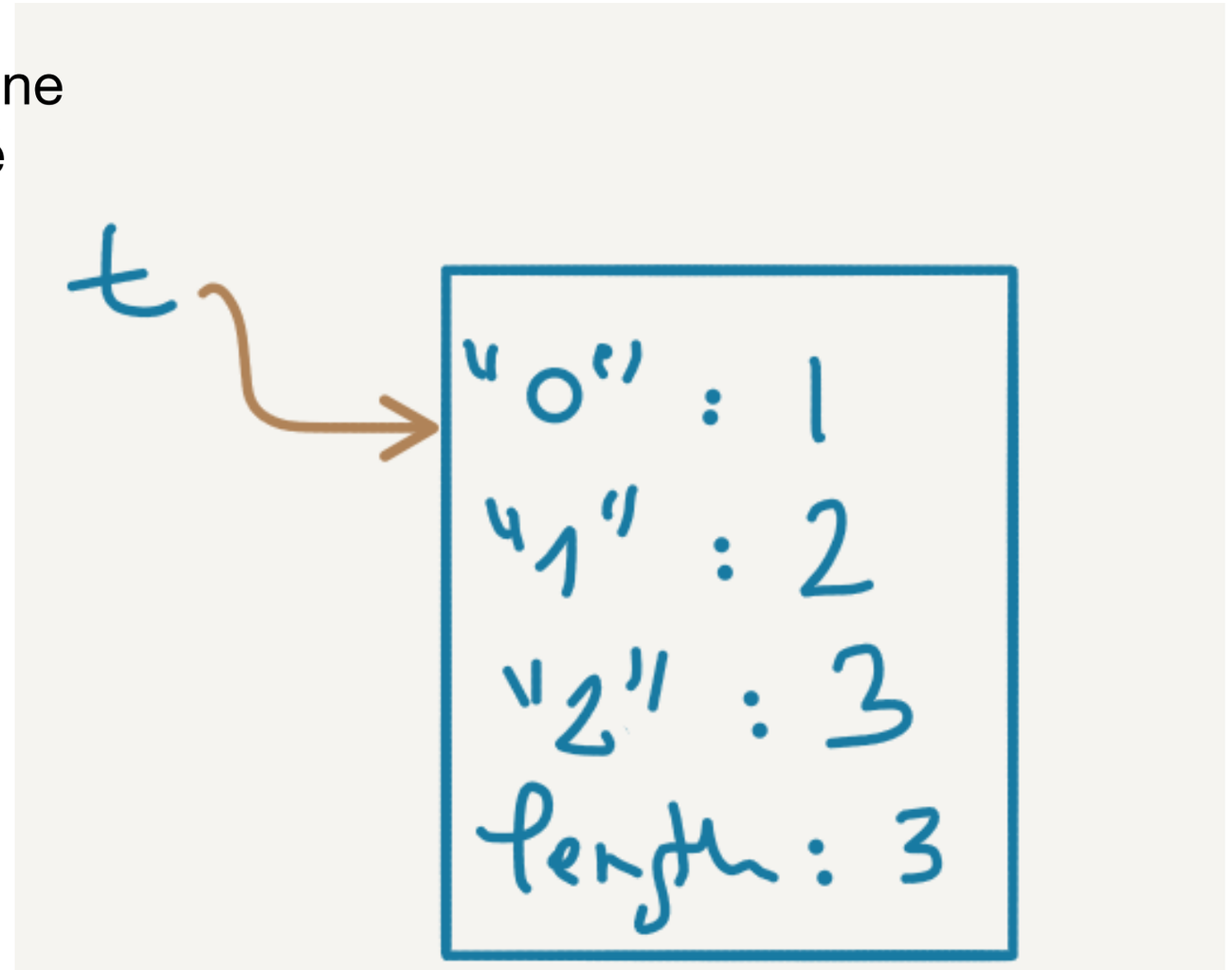
**On peut accéder aux propriétés d'un objet via:**

```
for (prop in a) {
    print(prop);
}
```

fg
fd
val

# Les tableaux en javascript

Un tableau est stocké comme une structure avec comme propriété « 0 », « 1 », « 2 », etc. et length




```
var t = [1,2,3];
for (p in t)
    print(p, t[p]);
```

la notation *t.1* n'est pas valide car *1* n'est pas un identificateur légitime

# À quoi servent les objets?

- Un niveau de structuration supplémentaire
  - Jusqu'à maintenant tout était organisé autours du processus (algorithme).
  - En **programmation orientée objet** (POO), les algorithmes sont les propriétés des objets
  - Objet = façon de regrouper des fonctionnalités reliées

```
var Tools = {
    alea: function(from,to) {
        return from + Math.floor(Math.random() * (to-from));
    },
    swap: function(t,i,j) {
        var temp = t[from];
        t[from] = t[to];
        t[to] = temp;
    },
    timing: function(func,mess) {
    },
    ...
};
```

```
print(Tools.alea());
```

- **Héritage**
  - Abordé en IFT1025

# Plan

- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre premier objet: Piece
- Listes chaînées
- Les arbres

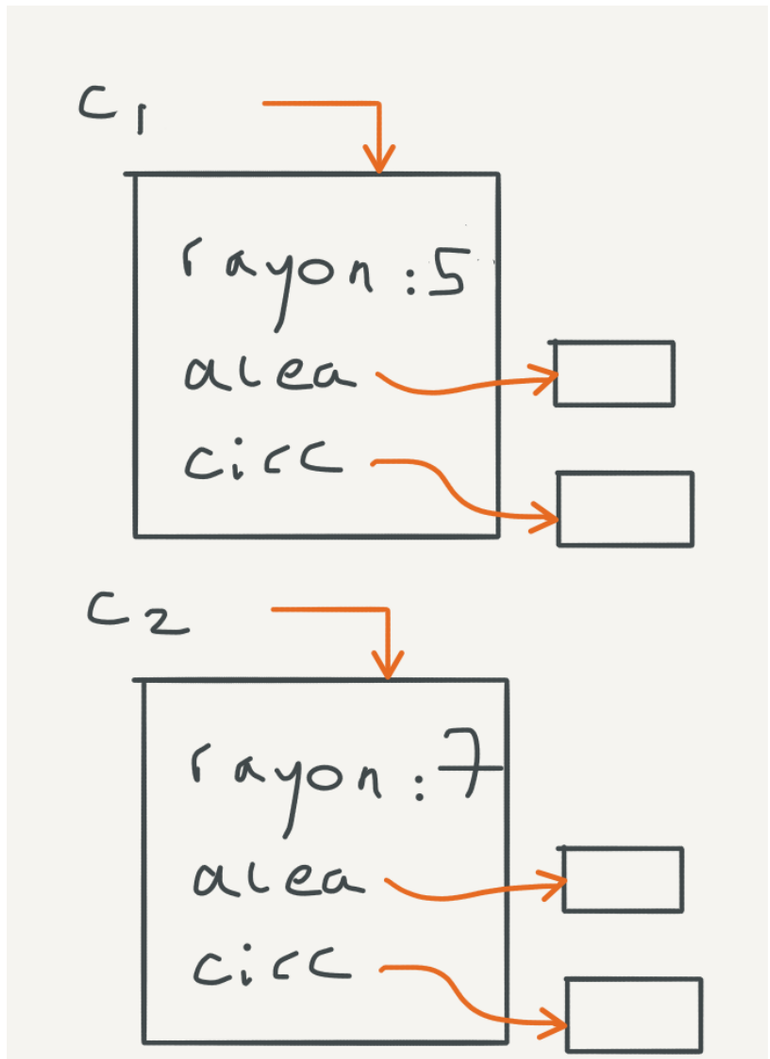
# Créer un objet

## Prise 1

```
var c1 = { rayon: 5 }, c2 = { rayon: 7 };
```

---

```
var c1 = {
    rayon: 5,
    area: function() {
      return Math.PI *  this.rayon *this.rayon;
    },
    circ: function() {
      return 2 * Math.PI * this.rayon; }   };

var c2 = {
    rayon: 7,
    area: function() {
      return Math.PI *  this.rayon *this.rayon;
    },
    circ: function() {
      return 2 * Math.PI  * this.rayon; }   };
```



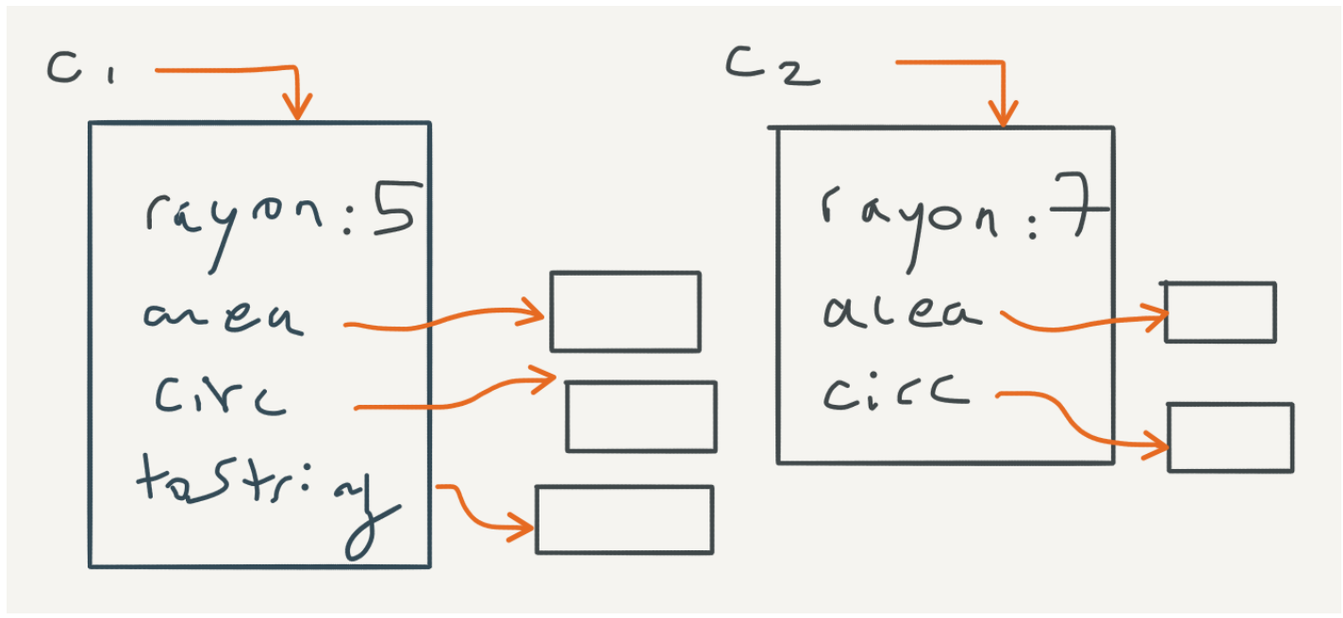
# Fastidieux ?

## Constructeur

```
var Circle = function (r) {
    return {
        rayon: r,
        area : function() { return Math.PI * this.rayon * this.rayon; },
        circ: function() { return 2 * Math.PI * this.rayon; }
    };
};
```

---

```
var c1 = Circle(5);
var c2 = Circle(7);

print( c1.area(), c2.area() );
print( c1 ); // [Object Object]

c1.toString = function() {
    return "je suis un cercle";
};
print(c1, c2);
```



# Le rôle de *this*

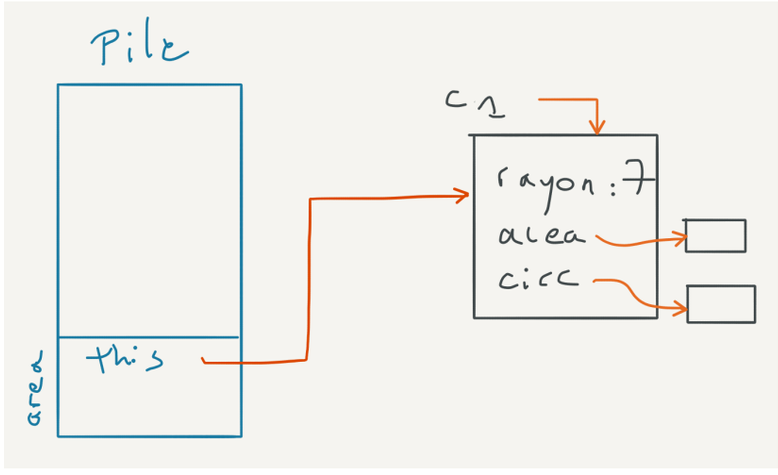
```
var c1 = Circle(7);
c1.area();
```

Lorsqu'on appelle une fonction depuis un objet (o*n parle dans ce cas de méthode*), l'adresse de l'**objet appelant** est passée à la fonction via ***this*** (comme un argument de plus passé à notre insu).

```
var c1 = Circle(7);
c1.area = function () {
   return Math.PI *
          this.rayon *
          rayon;
};
```

rayon n'est pas dans l'environnement de **area** et est donc recherché dans l'environnement global (variable globale).

**=> ne pas oublier *this*** dans une **méthode**



# Création d'un objet

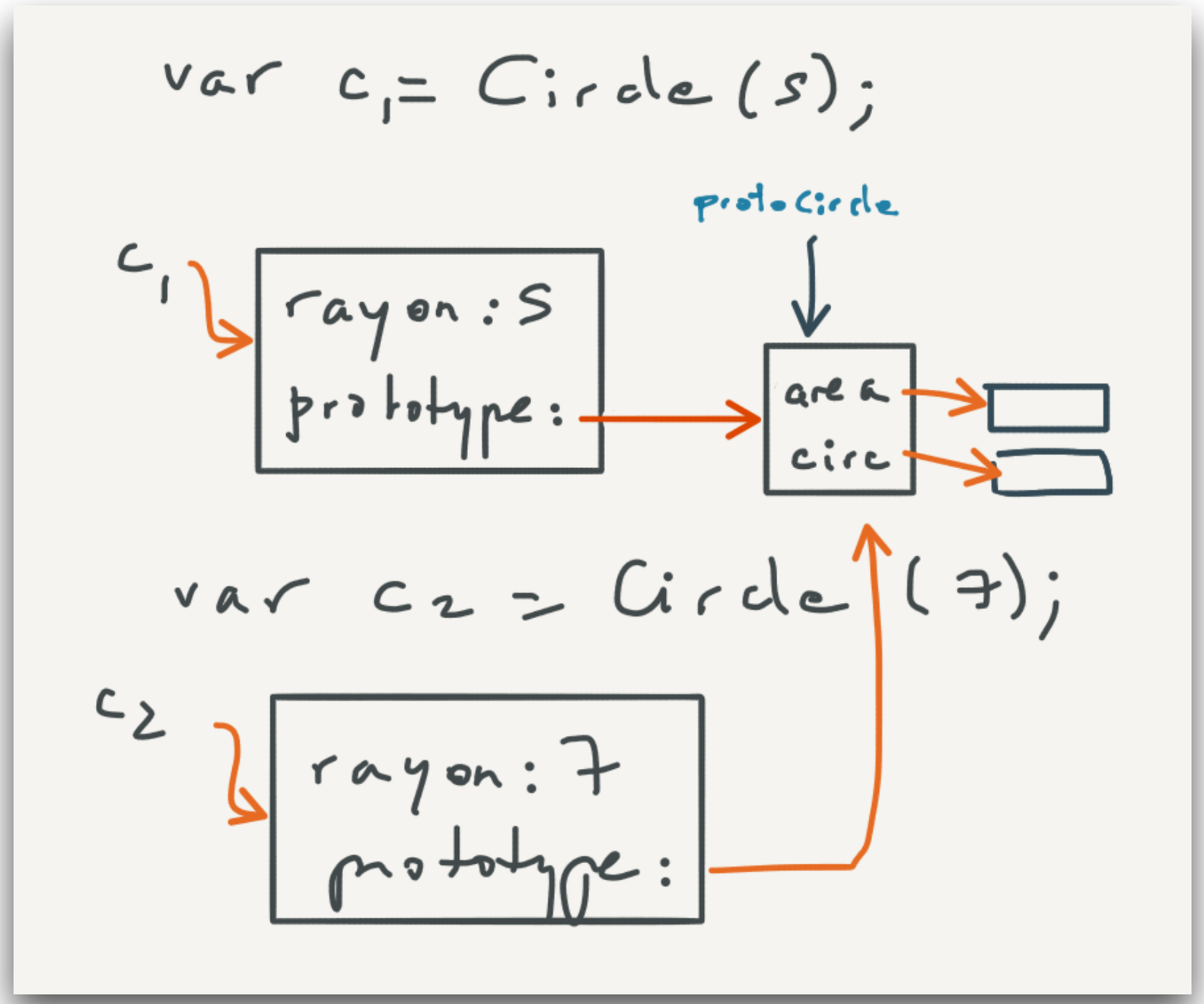
## Prise 2

**La solution précédente duplique le méthodes dans chaque objet, ce qui gaspille la mémoire. D'où l'idée de ranger ces méthodes dans un seul endroit.**

```
var protoCircle = {
      area : function() {
          return Math.PI * this.rayon
                              * this.rayon; },
      circ: function() {
          return 2 * Math.PI
                      * this.rayon; }
 };

var Circle = function (r) {
    var c = Object.create(protoCircle);
    c.rayon = r;
    return c;
};
```

**Object.create est une méthode qui:**

1. **Crée un objet (vide)**
2. **Met la propriété prototype de cet objet à la valeur passée en argument**
3. **Retourne l'adresse de l'objet**




Un seul endroit où sont stockées les méthodes

# Création d'un objet

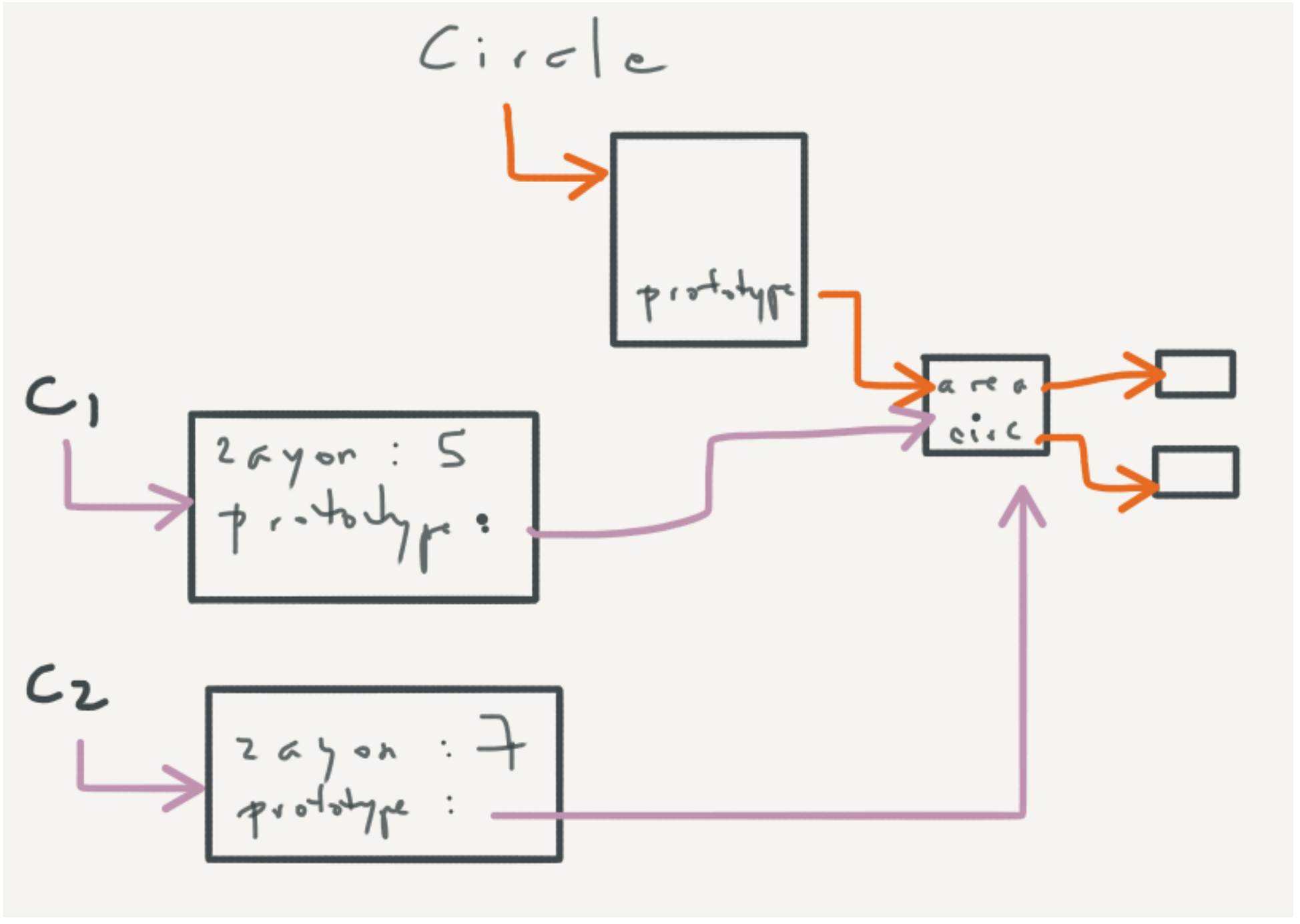
## Prise 3

bien … mais deux variables dans l'espace global.

```
var Circle = function (r) {
    var c = Object.create(Circle.prototype);
    c.rayon = r;
    return c;
};

Circle.prototype = {
      area : function() {
          return Math.PI * this.rayon
                         * this.rayon; },
      circ: function() {
          return 2 * Math.PI
                   * this.rayon; }
 };
```

pas une variable

possible car une fonction (ici Circle) est un objet et tout objet possède une propriété **prototype**


Circle
prototype
C1
radyon : 5
prototype :
area
circ
C2
radyon : 7
prototype :

# Création d'un objet

## Prise 4

**Compliqué ? Une version plus simple que nous utiliserons dans la suite et qui peut être déployée sans comprendre tous les « détails ».**

```
var Circle = function (r) {
    this.rayon = r;
};

Circle.prototype.area = function() {
        return Math.PI * this.rayon
                       * this.rayon;
};

Circle.prototype.circ = function() {
        return 2 * Math.PI
                 * this.rayon;
};
```

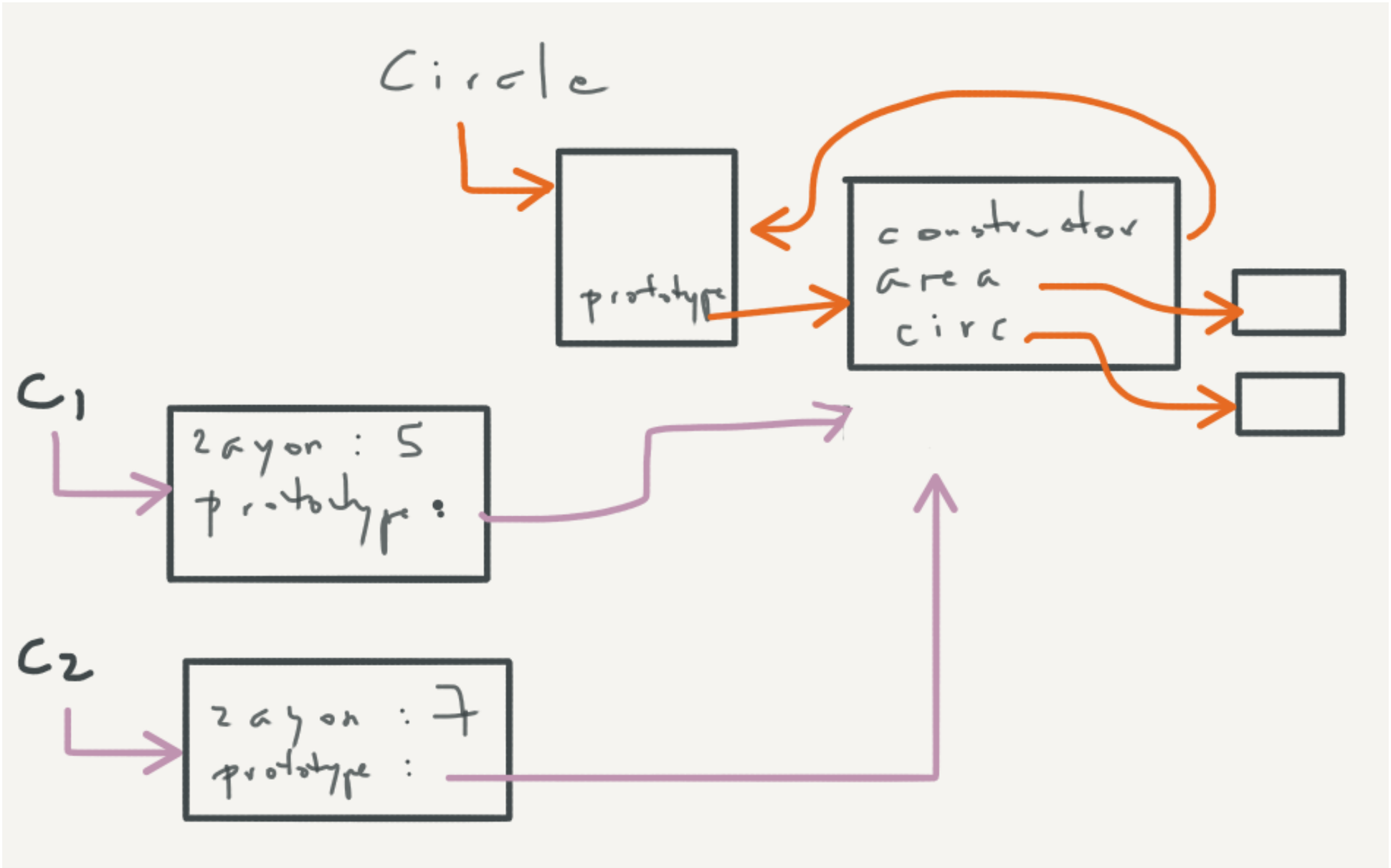
Définition

```
var c1 = new Circle(5);
var c2 = new Circle(7);
```

**new**

1. Crée un objet
2. Appelle le **constructeur** avec cet objet comme valeur de *this*
3. Met à jour le **prototype** de l'objet crée à la fonction (ou son prototype, je ne sais plus) appelée
4. Retourne l'adresse de l'objet crée **ssi** le constructeur ne retourne rien

Utilisation


Circle
prototype
constructor
area
circ
c1
zayon : 5
prototype :
c2
zayon : 7
prototype :

# Création d'un objet

Respirez !

constructeur

```
var Circle = function (r) {
    this.rayon = r;
};

Circle.prototype.area = function {
        return Math.PI * this.rayon
                       * this.rayon;
};

Circle.prototype.circ = function {
        return 2 * Math.PI
                 * this.rayon;
};
```

Définition

```
var c1 = new Circle(5);
var c2 = new Circle(7);
```

ne pas oublier !

syntaxe un peu nouvelle

convention: le constructeur commence par une majuscule

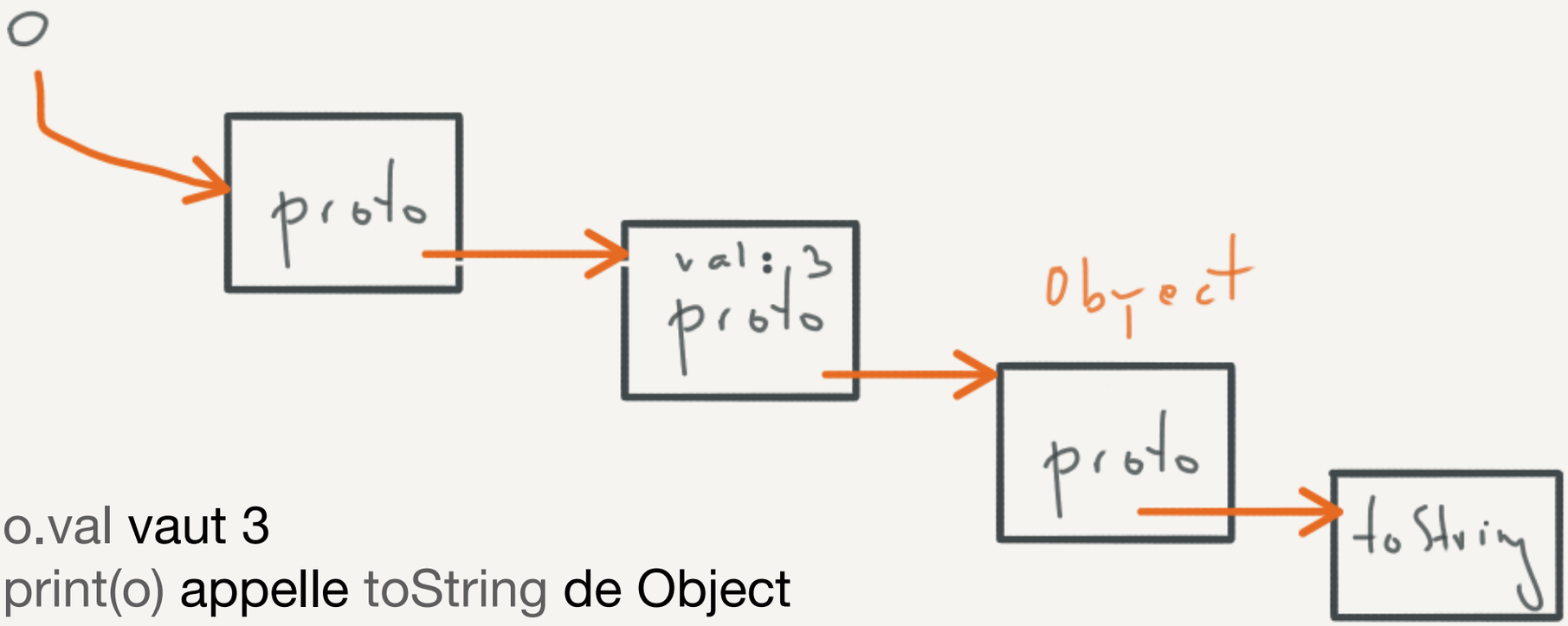
Utilisation

# Si loin si bon…

- Création d'un objet en passant le constructeur à **new**.
- Le rôle du **constructeur**:
  - préparer l'objet  crée par **new** avec **toute l'information utile** (les propriétés définitoires d'un objet particulier)
  - toujours en majuscule
    - par convention et afin d'éviter de l'oublier
- Les **propriétés communes** à tous les objets (notamment les **méthodes**) sont dans le « **prototype** » de l'objet
- On invoque une méthode depuis l'adresse (le nom) d'un objet et ***this*** est lié lors de l'appel à cette adresse
  - c1.area() invoque la méthode (fonction) area,
  - this vaut alors c1, l'adresse de l'objet appelant

# À propos des prototypes

- Tous les objets ont un prototype.
- Par défaut, le prototype d'un objet pointe sur celui de la classe **Object** qui contient des services génériques comme **toString**
  - **q**ui par défaut retourne [object Object]
- Chainage des prototypes
  - lecture: en suivant le chaînage
  - écriture: dans l'objet pointé




o.val vaut 3
print(o) appelle toString de Object

# Plan

- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre premier objet: Piece
- Rational
- Listes chaînées
- Les arbres

# Pièce

On veut créer des pièces avec lesquelles on peut faire des tirages pile ou face.

Choses à faire lorsqu'on crée un objet:

1. penser à l'**état** de l'objet
   - ensemble des variables
2. penser à l'**interface** de l'objet
   - ensemble des fonctions
3. penser au **constructeur**
   - quelles valeurs doivent être spécifiées?

Nous allons faire cela de façon itérative sur notre exemple Piece

# Pièce

## Prise 1

- **état**:
  - un entier (val): 1 = pile, 2 = face
- **interface**:
  - flip()
  - getValue()
- **constructeur**
  - pas d'argument

1. couplage
2. construction déficiente (état toujours à 1)

```
var Piece = function() {
  this.val = 1;
};
Piece.prototype.flip = function () {
  this.val = alea(1,3);
}
Piece.prototype.getValue = function() {
  return this.val;
};
```

alea dans *tools.js*

Classe **(modèle)**

```
var p1 = new Piece();
var p2 = new Piece();
p1.flip();
p2.flip();
print(p1.getValue(), p2.getValue());
if (p1.getValue() === 1) {
   print(" pile ");
   p1.flip();
}
```

couplage

Instances **(utilisation)**

# Pièce

## Prise 2

On pourrait très bien écrire p1.PILE, mais l'utilisation du nom de la classe est plus clair (ne dépend pas d'une instance)

```
var Piece = function() {
  this.val = 1;
};

Piece.prototype.PILE = 1;
Piece.prototype.FACE = 2;

Piece.prototype.flip = function () {
  this.val = alea(1,3);
}

Piece.prototype.getValue = function() {
  return this.val;
};
```

```
var p1 = new Piece();
var p2 = new Piece();

p1.flip();
p2.flip();

print(p1.getValue(), p2.getValue());
if (p1.getValue() === Piece.PILE) {
   print(« pile »);
   p1.flip();
}
```

Mieux mais, la convention pile=1, face=2 est toujours exposée

des "constantes" de la classe `Piece`que partagent toutes les instances

# Pièce

## Prise 3

tirage à la création

```
var Piece = function() {
  this.val = alea(1,3);
};
```

```
Piece.prototype.flip = function () {
  this.val = alea(1,3);
}
```

prédicat

```
Piece.prototype.isPile = function() {
  return this.val === 1;
};
```

Changement de l'interface

```
var p1 = new Piece();
var p2 = new Piece();

if (p1.isPile()) {
   print(" pile ");
   p1.flip();
}
```

Intuitif à l'usage

# Pièce

## Prise 4

booléen

```
var Piece = function() {
  this.pile = alea(1,3) === 1;
};


Piece.prototype.flip = function () {
  this.pile = alea(1,3) === 1;
}

Piece.prototype.isPile = function() {
  return this.pile;
};
```

Changement d'état

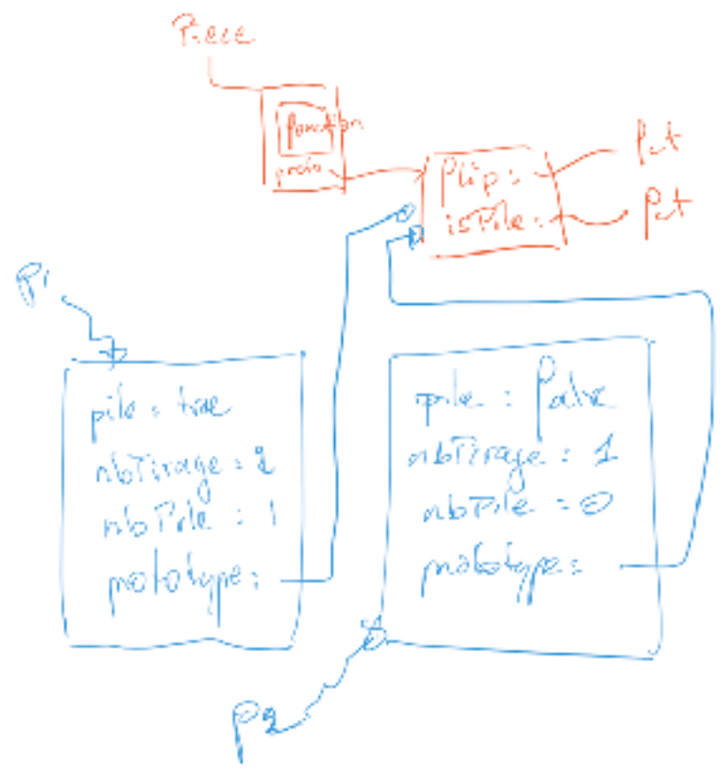
```
var p1 = new Piece();
var p2 = new Piece();

if (p1.isPile()) {
   print(" pile ");
   p1.flip();
}
```

Aucun changement !

Le changement de variable d'état n'engendre pas de modification à l'usage (pas de couplage)

# Pièce

## Prise 5




```
var Piece = function() {
  this.nbTirage = 0;
  this.nbPile = 0;
  this.flip();
 };


Piece.prototype.flip = function () {
  this.pile = alea(1,3) === 1;
  this.nbTirage++;
  if (this.pile) this.nbPile++;
}
Piece.prototype.isPile = function() {
  return this.pile;
};
```

Changement d'état

```
var p1 = new Piece();
var p2 = new Piece();


p1.flip();

print(p1.nbTirage); // 2
print(p2.nbTirage); // 1
print(p1.nbPile);

print(p1); // [object Object]
```

Les propriétés de l'objet sont accessibles (lecture et écriture)

# Pièce

## Prise 6

```
var Piece = function() {
  this.nbTirage = 0;
  this.nbPile = 0;
  this.flip();
};

Piece.prototype.flip = function () {
  this.pile = alea(1,3) === 1;
  this.nbTirage++;
  if (this.pile) this.nbPile++;
}
Piece.prototype.isPile = function() {
  return this.pile;
};

Piece.prototype.toString = function() {
   return this.pile? "PILE":"FACE";
};
```

```
var p1 = new Piece();
var p2 = new Piece();

p1.flip();

print(p1.nbTirage); // 2
print(p2.nbTirage); // 1
print(p1.nbPile);

print(p1); // PILE ou FACE
```

Toujours définir toString. C'est une fonction qui retourne une string et qui ne prend à priori pas d'argument.

# Pièce

## Prise 7

naturel de l'offrir

```
var Piece = function(biais){
  if (undefined === biais) biais = 0.5;
  this.biais = bias;

  this.nbTirage = 0;
  this.nbPile = 0;
  this.flip();
};

Piece.prototype.flip = function () {
  this.pile = (Math.random() < this.biais);
  this.nbTirage++;
  if (this.pile) this.nbPile++;
}
Piece.prototype.isPile = function() {
  return this.pile;
};

Piece.prototype.toString = function() {
   return this.pile? "PILE":"FACE";
};
```

```
var p1 = new Piece(0.3);
var p2 = new Piece();

p1.flip();

print(p1.nbTirage); // 2
print(p2.nbTirage); // 1
print(p1.nbPile);

print(p1); // PILE ou FACE
```

Raffiner l'interface (constructeur)

# Pièce

## Prise 8

```
var Piece = function(biais) {
  if (undefined === biais) biais = 0.5;
  this.biais = bias;

  this.nbTirage = 0;
  this.nbPile = 0;

  this.flip();
  ++(Piece.prototype.nb);
};
Piece.prototype.nb = 0;

Piece.prototype.flip = function () {
  this.pile = (Math.random() < this.biais);
  this.nbTirage++;
  if (this.pile) this.nbPile++;
}

/**
 * @return true si la piece est pile
 */
Piece.prototype.isPile = function() {
  return this.pile;
};
```

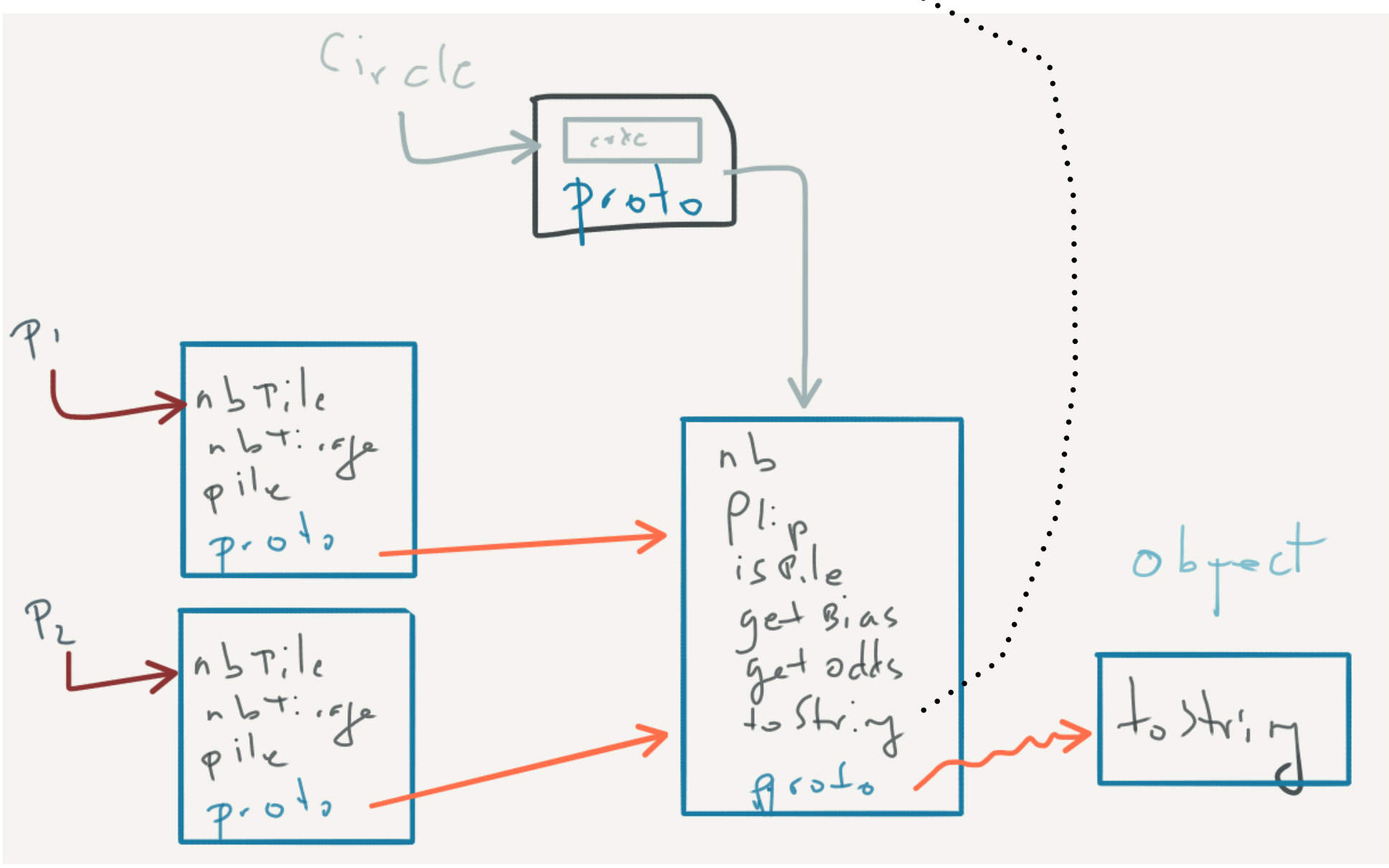
```
Piece.prototype.toString = function() {
   return this.pile? "PILE":"FACE";
};

/**
 * @return le biais  spécifié à la constructio
 */
Piece.prototype.getBias = function () {
    return this.biais;
};

/**
 * @return le biais empirique  p(PILE)
 */
Piece.prototype.getOdds = function () {
    return this.nbPile/ this.nbTirage;
};
```

Raffiner l'interface (méthodes)

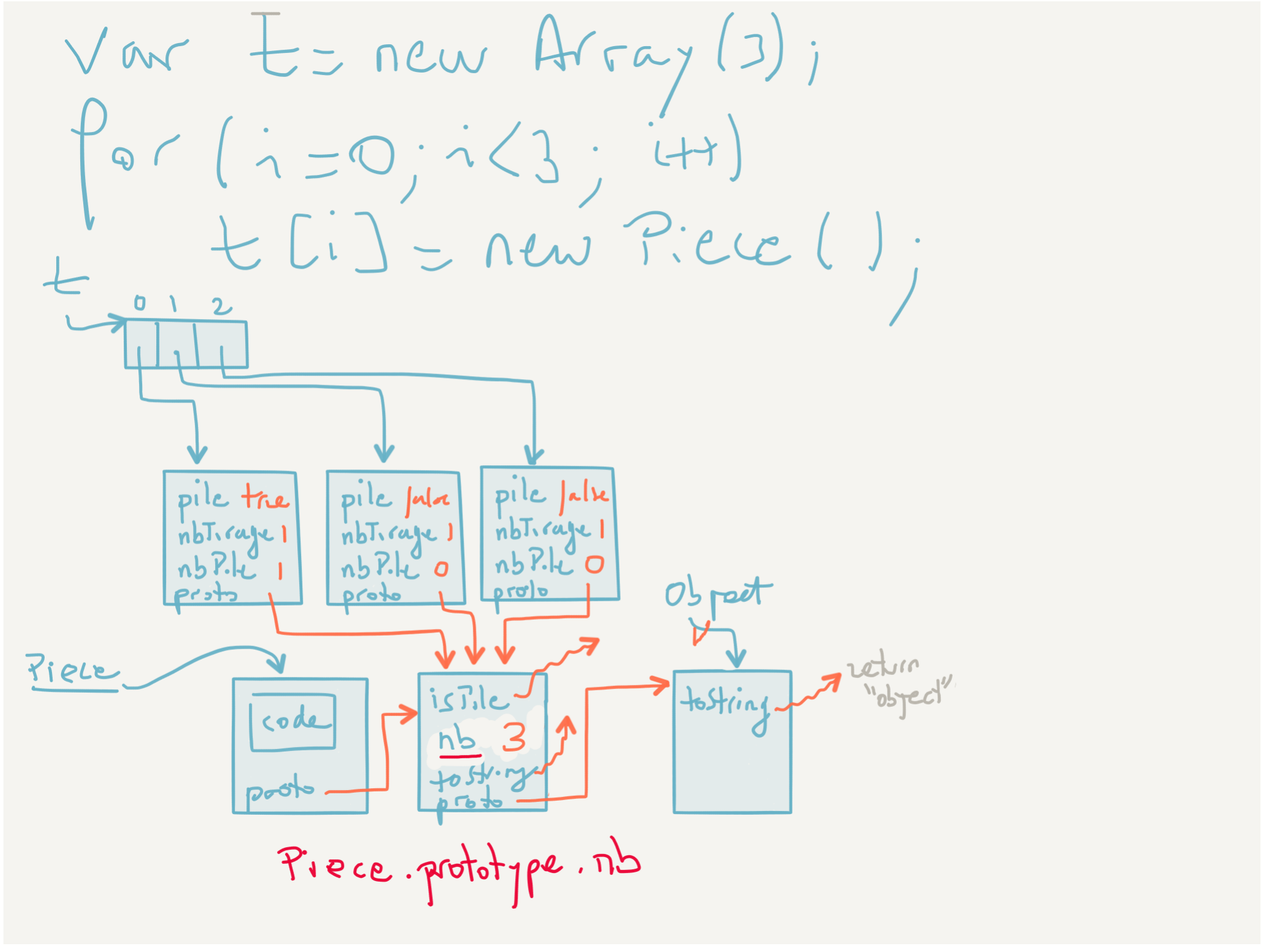
# Utilisation

```
var p1 = new Piece();
var p2 = new Piece();
```

*p1.toString()*
exécute ce *toString* avec *this* alias de *p1*



# Utilisation



# Utilisation

Code dans *piece.js*

```
var test2 = function(nbTirage) {

    var i, j, t = new Array(11);

    for (i=0; i<t.length; ++i)
        t[i] = new Piece(0.1*i); // biais

    print("nb pieces",Piece.prototype.nb);

    for (i=0; i<t.length; ++i)
        for (j=0; j<nbTirage; ++j)
          t[i].flip();

    print("distribution des piles apres",nbTirage,"tirages:");
    for (i=0; i<t.length; ++i)
        print("piece",(i+1)," bias="+t[i].getBias().toFixed(1),
                "p="+t[i].getOdds().toFixed(2));
};
```

# Utilisation

```
var test2 = function(nbTirage) {

    var biais = new Array(11);
    var piece = new Array(11);
    var nbPile = new Array(11);
    var tirage = new Array(11);

    for (var i=0; i<piece.length; ++i) {
        biais[i] = 0.1*i; // biais
        nbPile[i] = 0;
        tirage[i] = 0;
    }

    for (var i=0; i<piece.length; ++i) {
        for (var j=0; j<nbTirage; ++j) {
            piece[i] = Math.random() < biais[i];
            if (piece[i]) nbPile[i]++;
        }
        tirage[i] += nbTirage;
    }

    for (var i=0; i<piece.length; ++i)
        print("piece",(i+1)," bias="+biais[i].toFixed(1),
                « p="+percent(nbPile[i],tirage[i]).toFixed(2));
};
```

Code dans *piece_sans_objet.js*

- Moins lisible
  - Il faut comprendre ici que les tableaux sont synchronisés
  - Mélange algorithmique / usage
- Moins maintenable
  - On risque par erreur de désynchroniser un tableau
  - Une horreur  si on avait à swapper des pièces (par exemple pour les trier)

# Plan

- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre premier objet: Piece
- Rational
- Listes chaînées
- Les arbres

# Rational

**État**:
- numérateur
- dénominateur
- signe?

**Services:**
- getNumerateur
- getDenominateur
- add, sub, div, mul
- equals
- getApproximateValue
- toString

**Construction:**

```
var r1 = new Rational(1,2); // 1/2
var r2 = new Rational(3);    // 3/1
var r3 = new Rational();      // random
var r4 = new Rational(-2,4); //  -1/2
var r5 = new Rational(1,-2); // -1/2
var r6 = new Rational({num:1, den:2, pos:false}); // -1/2
```

**Convention: le numérateur porte le signe (choix discutable)**

# Rational

```
var Rational = function (num,den) {

    if ((undefined === num) && (undefined === den)) {
       num = alea(1,11); // [1,10]
       den = alea(1,11);
    }
    else if (undefined === den) {
       den = 1;
    }
    else if (den === 0) { // lever une exception
          print ("!!!! denominateur null -> je le mets a 1")
          den = 1;
       }

    // le cas general
    this.num = num;
    this.den = den;
    this._reduce();
};
```

- Un constructeur doit vérifier les entrées de l'utilisateur
- le traitement en cas de problème est ici très arbitraire (déclencher une erreur  serait une alternative pertinente)

# Rational

```
Rational.prototype._reduce = function() {

    var pgcd = function(a,b) {
        while (a !== b) {
            if (a < b) b -= a;
            else a -= b;
        }
        return a;
    };

    if (this.den<0) {
        this.num = -this.num; // le numerateur porte le signe (convention interne)
        this.den = -this.den;   // qui devient donc positif
    }

    if (this.num === 0) this.den = 1; // tous les 0/n seront encodes en 0/1 (why? pour equals)
    else {
        var common = pgcd(Math.abs(this.num),this.den);
        if (common !== 1) {
            this.num /= common;
            this.den /= common;
        }
    }
```

# Rational

```
/*
 * @return denominateur du rationnel
 */
Rational.prototype.getDenominateur = function() {
    return this.den;
};

/*
 * @return numerateur du rationnel
 */
Rational.prototype.getNumerateur = function() {
    return this.num;
};
```

On pourrait très bien utiliser *den* et *num* directement

# Rational

```
/*
 * @return valeur approchee (en faisant le calcul)
 */
Rational.prototype.getApproximateValue = function() {
    return this.num / this.den;
};
```

```
/*
 * @return une string representant un rationnel
 */
Rational.prototype.toString = function() {
    if (this.num === 0) return "0";
    if (this.den === 1) return this.num + " ";
    return this.num + "/" + this.den;        // le cas general
};
```

# Rational

Comment offrir les opérations arithmétiques ?

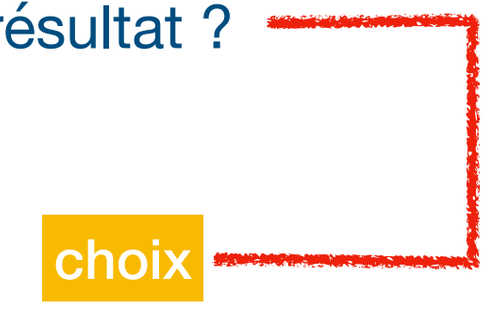
```
var r1 = new Rational(2,3);
var r2 = new Rational(4,5);
```

**add**(r1,r2);

- en faisant porter le résultat sur r1 ? sur r2 ?
- en retournant un nouvel objet portant le résultat ?

r1.**add**(r2);

- en modifiant r1 ?
- en modifiant r2 ?
- en retournant un nouvel objet portant le résultat ?

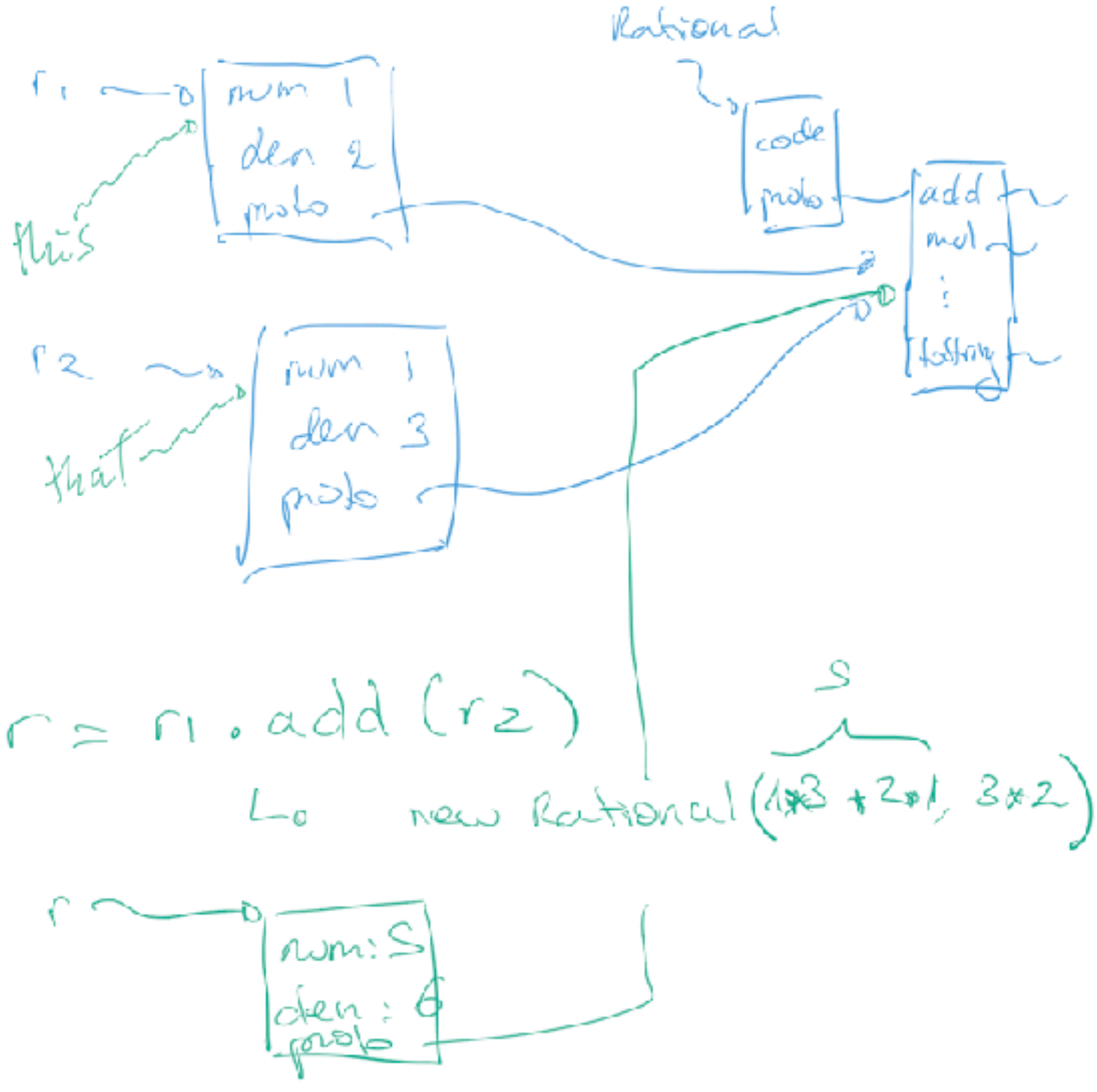


# Rational

```
/*
 * @param that un rationnel
 * @return nouveau rationnel contenant la somme de that et this
 */
Rational.prototype.add = function (that) {
    return new Rational(this.getNumerateur() * that.getDenominateur() +
                        this.den * that.getNumerateur(),
                        this.den * that.getDenominateur());
};


/*
 * @param that un rationnel
 * @return nouveau rationnel contenant le produit de that et this
 */
Rational.prototype.mul = function(that) {
    return new Rational(this.num * that.getNumerateur(),
                        this.den * that.getDenominateur());
};
```

# Rational

```
var r1 = new Rational(1,2);
var r2 = new Rational(1,3);

var r = r1.add(r2);
```



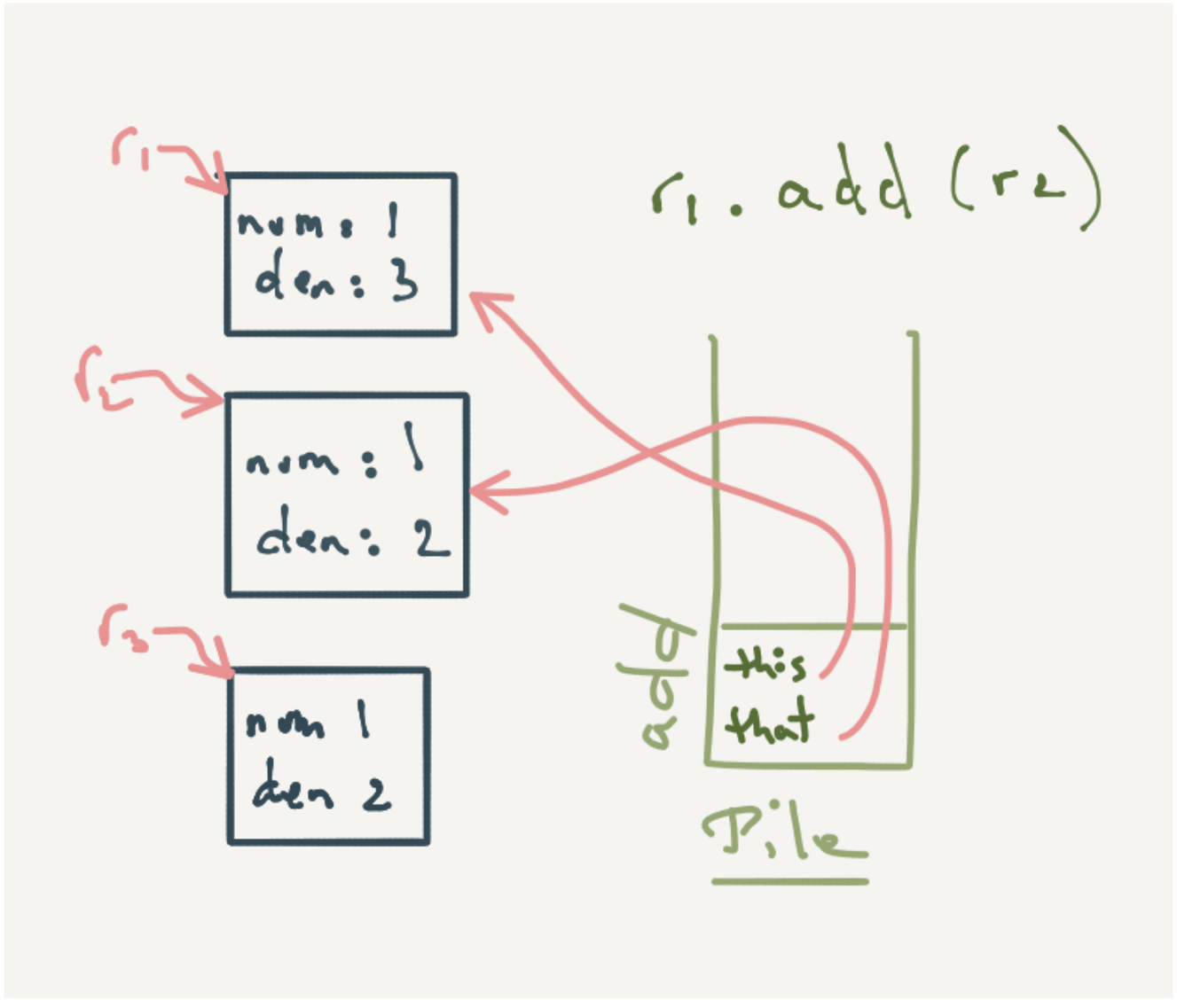
# Rational

```
/*
 * @return true si l'objet appelant est égal à l'objet spécifié
 */
Rational.prototype.equals = function(that) {
    return  (this.den === that.getDenominateur()) &&
              (this.num === that.getNumerateur());
};
```

```
var r1 = new Rational(2,3);
var r2 = new Rational(-4,-6);

assert(r1.equals(r2));
```

# Rational

```
var test1 = function() {

    var r1 = new Rational(1,3);
    var r2 = new Rational(7,14);
    var r3 = new Rational(-1,-2);

    var r_add = r1.add(r2);
    print (r1 + " + " + r2 + " = " + r_add);

    var r_sub = r1.sub(r2);
    print (r1 + " - " + r2 + " = " + r_sub);

    var r_mul = r1.mul(r2);
    print (r1 + " * " + r2 + " = " + r_mul);

    var r_div = r1.div(r2);
    print (r1 + " / " + r2 + " = " + r_div);

    var r = r1.sub(r1);
    print(r);
    print(r2,r2.equals(r3)? "=":"!=",r3);
};
```



# Rational

```
var test2 = function(nbRat) {

    var t = new Array(nbRat);
    var i;
    for (i=0; i<t.length; ++i)
        t[i] = new Rational(); // aleatoire

    var r = new Rational(0);
    var a = 0;

    for (i=0; i<t.length; i++) {
        r = r.add(t[i]);
        a += t[i].getApproximateValue();
    }
    print(t.join("+"),"=",r); // cool eh?
    print("approx (each time)=",a,"approx (once)=",r.getApproximateValue());
};
```

# Rational

```
var test3 = function() {

   var r = new Rational(0);
   for (var i=1; i<20; ++i)
       r = r.add(new Rational(i));

   assert(r.equals(new Rational(190)),
          "erreur de calcul dans test3 (rational.js)");

   assert( new Rational(1,2).add(new Rational(1,4))
                            .add(new Rational(1,4))
                            .equals(new Rational(1)),
          "erreur de calcul dans test3 (rational.js)");
};
```

outch !

# Plan

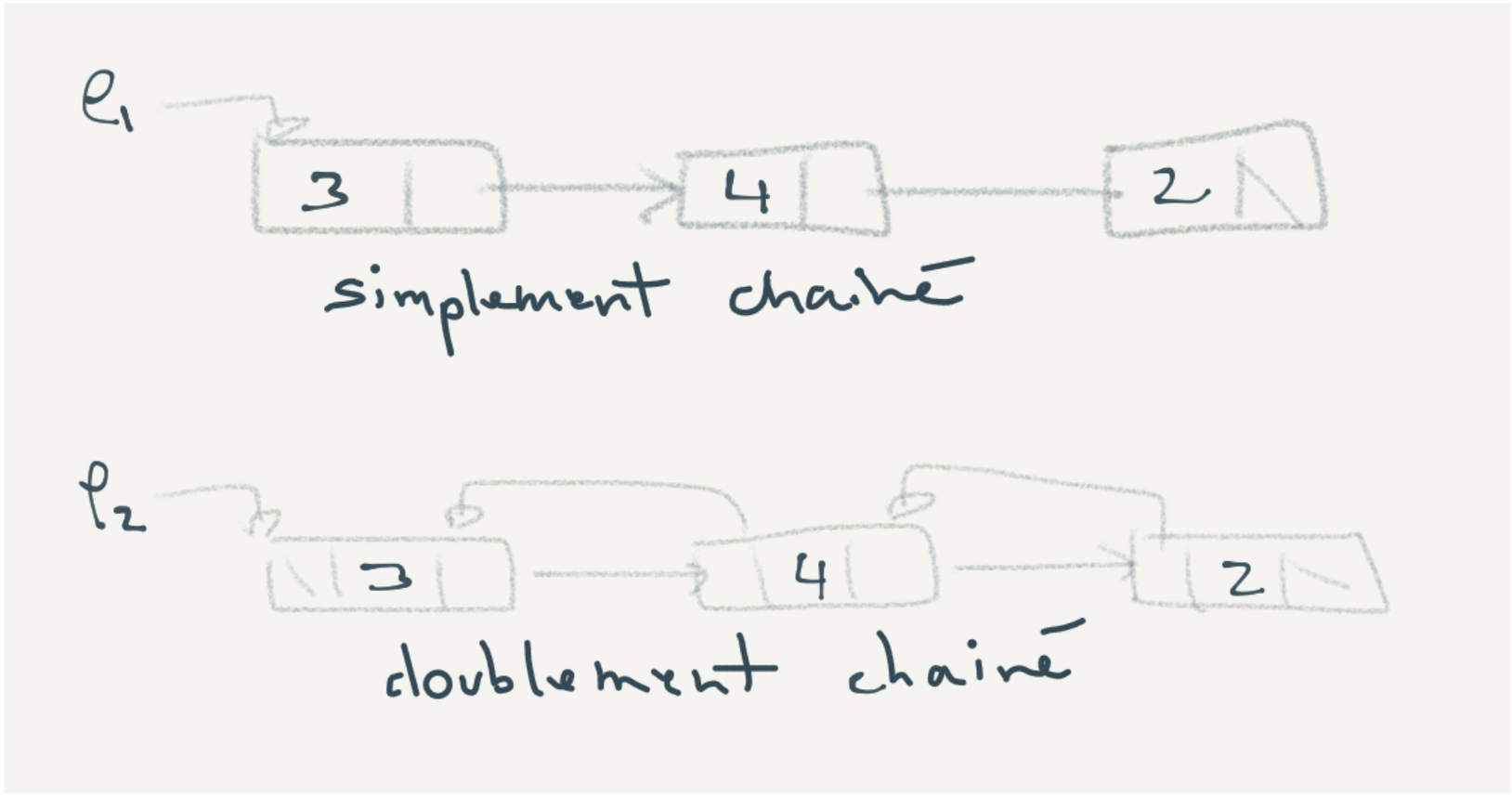
- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre premier objet: Piece
- Rational
- Listes chaînées
- Les arbres

# Listes chaînées

## Principes

- **Pro**: insertion fréquente « au milieu »
- **Cons**:  pas d'indexation

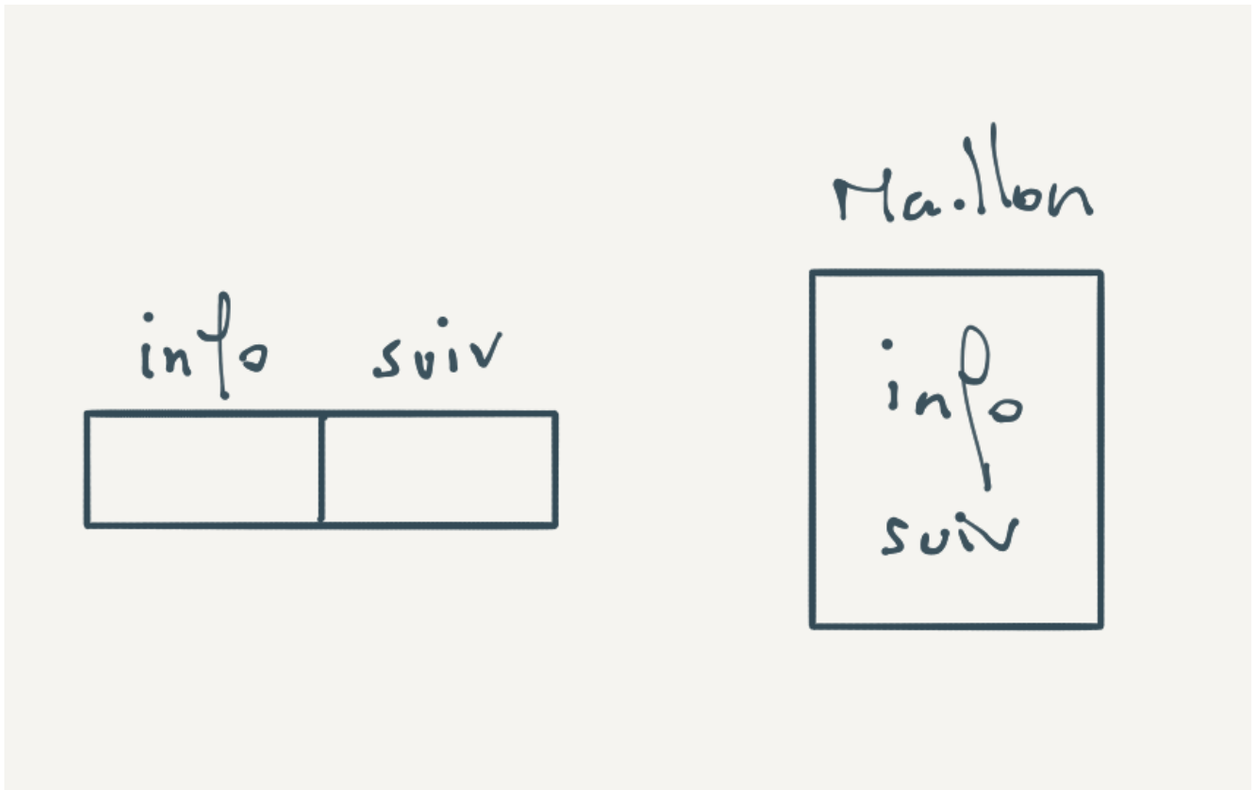
La structure de choix en IA pendant longtemps



# Maillon

```
var Maillon = function(info,suiv) {
    this.info = info;
    this.suiv = (undefined === suiv)? null : suiv;
};
```

pointe sur rien

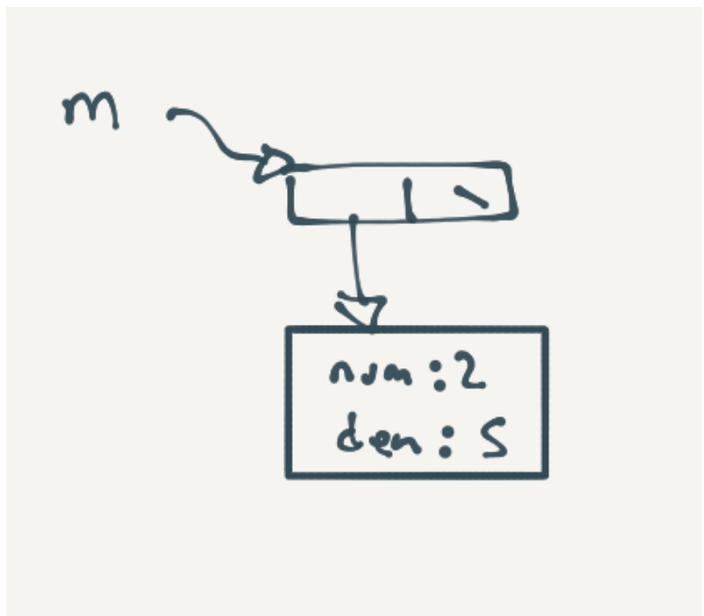


# Maillon

```
var Maillon = function(info,suiv) {
    this.info = info;
    this.suiv = (undefined === suiv)? null : suiv;
};
```

pointe sur rien

```
var m = new Maillon( new Rational(2,5) );
```

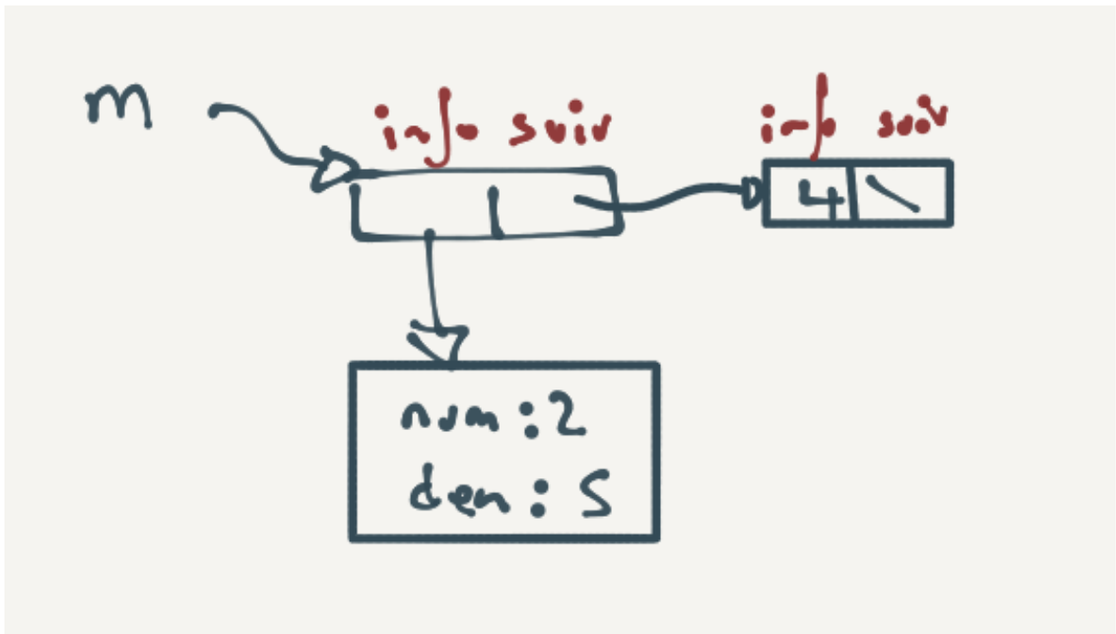


# Maillon

```
var Maillon = function(info,suiv) {
    this.info = info;
    this.suiv = (undefined === suiv)? null : suiv;
};
```

pointe sur rien

```
var m = new Maillon( new Rational(2,5) );
m.suiv = new Maillon(4);
```

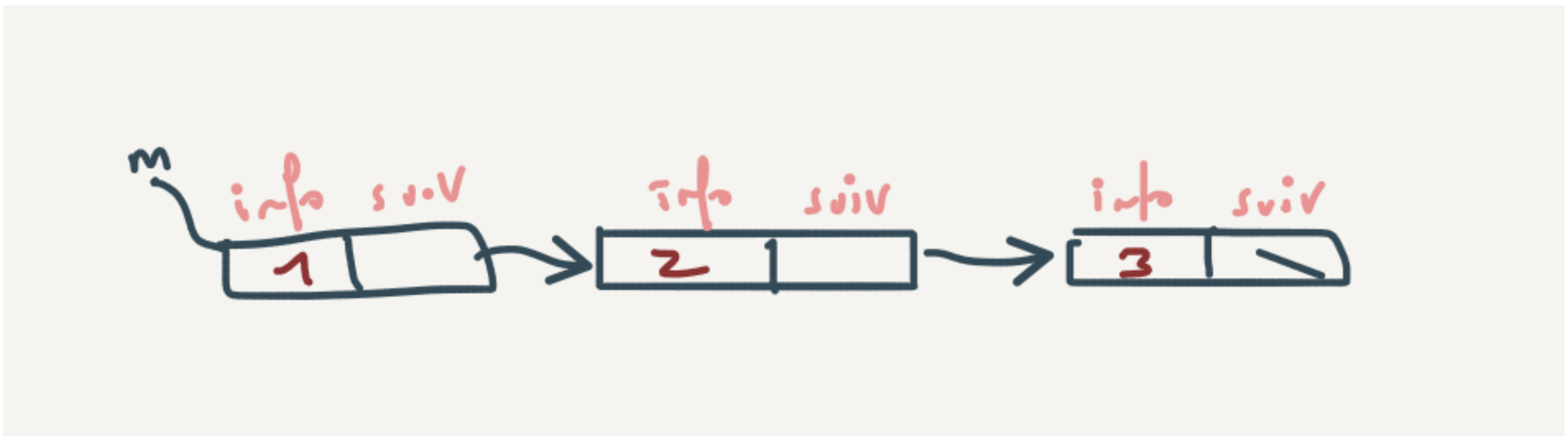


# Maillon

```
var Maillon = function(info,suiv) {
    this.info = info;
    this.suiv = (undefined === suiv)? null : suiv;
};
```

pointe sur rien

```
var m = new Maillon( 1, new Maillon( 2, new Maillon(3) ) );
```



# Liste

**État:**

- tête
- queue (pourquoi)?
- size (pourquoi)?

**Services:**

- addFirst(info)
- addLast(info)
- exists(info)
- isEmpty()
- getFirst()
- getLast()
- remove(info)
- add(p)
- toString()
- clear()

**Construction / usage:**

```
var list1 = new Liste();
var list2 = new Liste([1,2,3,4]);
```

Il existe des implémentations bien meilleures que celle que nous allons regarder !

**https://www.collectionsjs.com/list**

# Constructeur

```
/**
 * Constructeur
 * @param t un tableau de valeurs à
 *              ajouter (ou rien si aucune valeur)
 */
var Liste = function (t) {
    this.size = 0;
    this.tete = this.queue = null;

    if ((t !== undefined) && (t.length !== undefined)) {
       for (var i=0; i<t.length; i++)
           this.addLast(t[i]);
    } // si t est un tableau
};
```

```
var l1 = new Liste(); // liste vide
var l2 = new Liste([1,2,3,4]);

print(l2.size); // 4
print(l2);      // [object Object]
```

*length* plutôt que *size* aurait été plus pertinent (pour faire comme les tableaux)

# toString

```
Liste.prototype.toString = function() {
    var s="";
    for (var p=this.tete; p != null; p = p.suiv)
       s += (((p === this.tete)? "":" ")+p.info);
    return s;
};
```

---

```
l = new Liste([1,2,3,new Rational(1,2),4]);
print(l);
```

1 2 3 1/2 4

# Assesseurs (get)

```
/**
 * @return le premier element de la liste
 *         ou null (si la liste est vide)
 */
Liste.prototype.getFirst = function() {
    return (this.size === 0)? undefined : this.tete.info;
};

/**
 * @return le dernier element de la liste
 *         ou null (si la liste est vide)
 */
Liste.prototype.getLast = function() {
    return (this.size === 0)? undefined: this.queue.info;
};

/**
 * @return true si la liste est vide
 */
Liste.prototype.isEmpty = function() {
    return this.size === 0;
};
```

# Ajout en tête

```
/**
 * @param info l'information a ajouter en tete dans la liste
 */
Liste.prototype.addFirst = function(info) {
    this.tete = new Maillon(info,this.tete);
    if (this.size === 0)
        this.queue = this.tete;
    this.size++;

    return this; // pour le chainage des operations
};
```

Rend ceci possible

```
var l = new Liste();

l.addFirst(1);
l.addFirst(2);
l.addFirst(3);
```

```
var l = new Liste();

l.addFirst(1).addFirst(2).addFirst(3);
```

# Ajout en queue

```
/**
 * @param info l'information a ajouter en fin dans la liste
 */
Liste.prototype.addLast = function(info) {
    if (this.size === 0) this.addFirst(info);
    else {
      this.queue.suiv = new Maillon(info);
      this.queue = this.queue.suiv;
      this.size++;
    }

    return this; // pour le chainage des operations
};
```

```
var l = new Liste();

print(l.addFirst(1).addLast(2).addFirst(3)); // 3 2 1
```

# Recherche d'une info

```
/**
 * @param info à rechercher dans la liste (avec ===)
 * @return la reference r du maillon qui contient l'info si trouvee (note: r.info contient l'info)
 *         ou null si l'info n'existe pas
 * @warning complexité linéaire
 */
Liste.prototype.exists = function(info) {
    var p;
    for (p = this.tete; (p != null) && (p.info !== info); p = p.suiv);
    return (p === null)? null : p;
};
```

---

```
print( (new Liste([1,2,3])).exists(2).info ); // should print 2
```

# Indexation

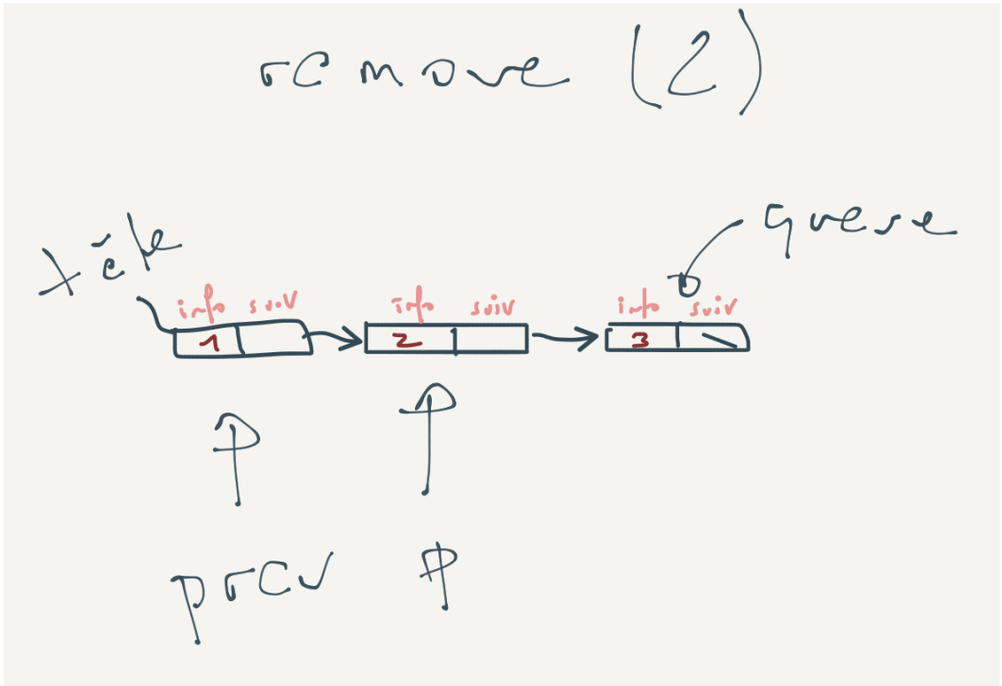
```
/**
 * @param i indice (0-indice) de l'element que l'on souhaite
 * @return le ieme element dans la liste (undefined si i >= size)
 * @warning lineaire (pas efficace)
 * @comment le choix d'offrir une telle methode est discutable
 */
Liste.prototype.getInfo = function(i) {
    var p;
    for (p = this.tete; (p != null) && (i > 0); p = p.suiv, i--) ;

    return (p === null)? undefined: p.info;
}
```

---

```
print( (new Liste([1,2,3])).getInfo(1)); // should print 2
```

# Détruire un élément

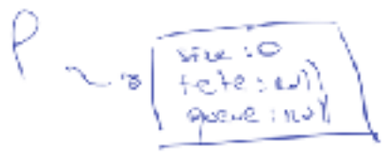
```
/**
 * @return une reference sur la liste
 *              une autre possibilité serait de retourner l'info.
 */
Liste.prototype.remove = function (info) {
    var prev,p;

    for (prev=null,p = this.tete;  (p !== null) && (p.info !== info); prev=p,p = p.suiv) ;

    ...

    prev.suiv = p.suiv;
    if (p.suiv === null) // p pointait sur le dernier maillon
        this.queue = prev;

    --this.size;

    ...
};
```



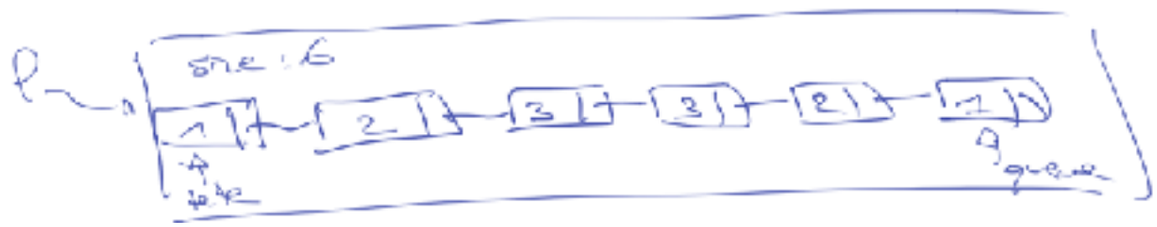
# Utilisation

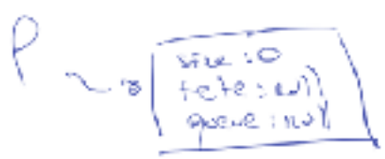
liste.js

```
var l = new Liste();

assert(l.size === 0);
l.addFirst(1).addFirst(2).addFirst(3);

assert(l.getFirst() === 3);
assert(l.getLast() === 1);
assert(l.size === 3);

l.addFirst(3).addFirst(2).addFirst(1);
assert(l.size === 6);

assert(l.exists(4) === null);
assert(l.exists(3) !== null);

assert (l.toString() === "1 2 3 3 2 1");
l.remove(3);
assert(l.toString() === "1 2 3 2 1");
l.remove(3);
assert( l.toString() === "1 2 2 1");
l.remove(1).remove(1).remove(2).remove(2);
assert(l.isEmpty());
```





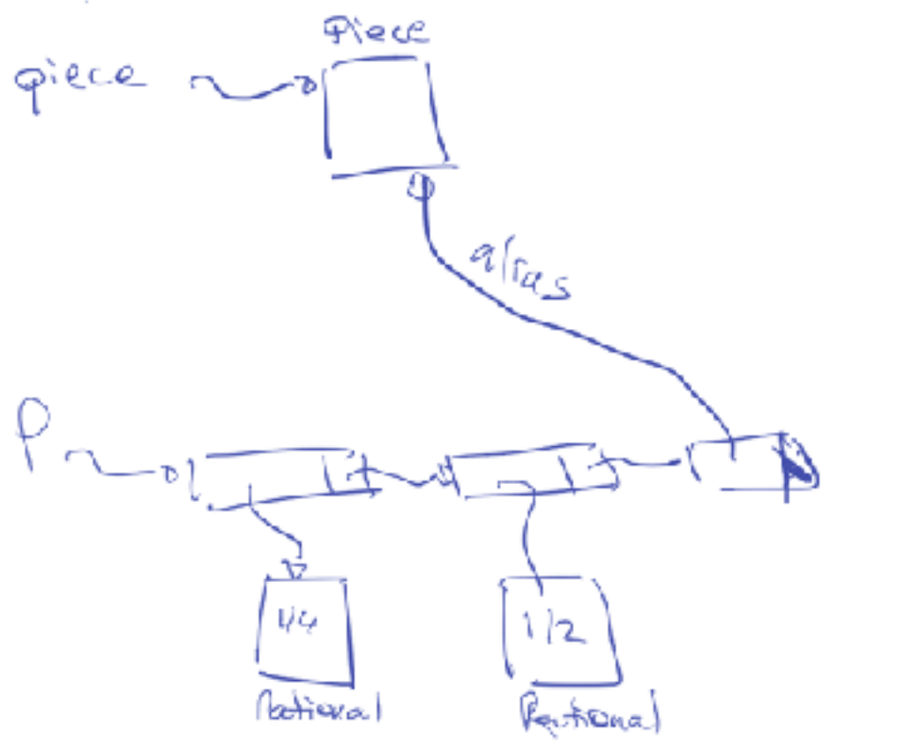


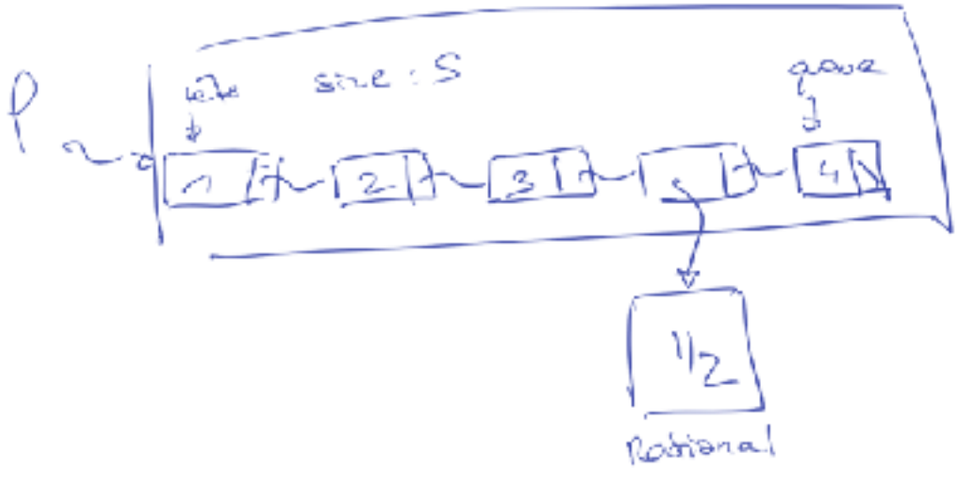



# Utilisation

liste.js

```
var piece = new Piece();
l.addFirst(piece);
l.addFirst(new Rational(1,2)).addFirst(new Rational(1,4));
print("list:", l);                                          //1/4 1/2 FACE

assert(l.exists(piece));                                    // quiz: why does it "work"?

assert(l.getInfo(3) === undefined);                         // car 3 elements seulement
assert(l.getInfo(2).constructor === Piece);     // facon de tester un type
assert(l.getInfo(1).constructor === Rational);
assert(l.getInfo(0).constructor === Rational);

l.clear();
assert(l.isEmpty());
assert(l.constructor === Liste);

l = new Liste([1,2,3,new Rational(1,2),4]);
print(l);                                                   // 1 2 3 1/2 4
assert(l.size === 5);
l.clear();
assert(l.isEmpty());
```



# Plan

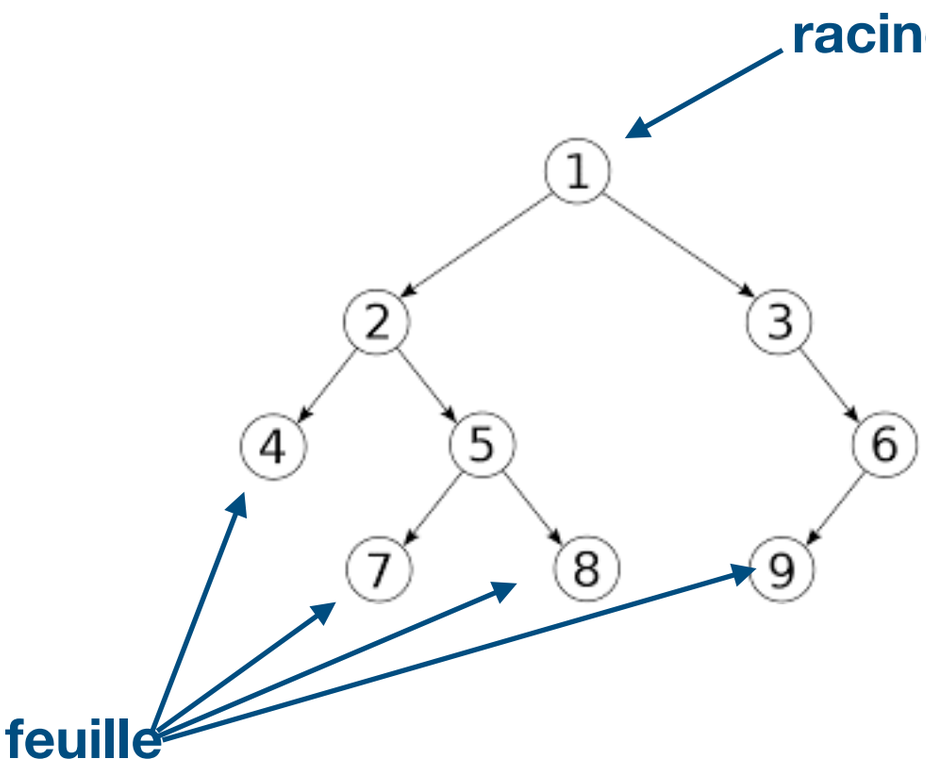
- Objets littéraux
- 4 façons de créer des objets (**technique**)
- Notre premier objet: Piece
- Rational
- Listes chaînées
- Les arbres

# Arbres binaires

Structure omniprésente en informatique

**Déf**:

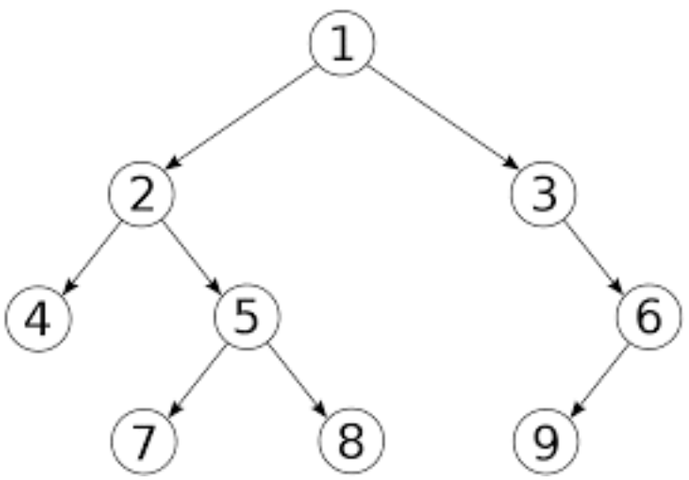
- un noeud **racine**
- chaque noeud contient au plus deux fils (cas binaire)
- les noeuds sans fils sont appelés les **feuilles**



# Node

Pour encoder chaque noeud d'un arbre

```
var Node = function(info,fg,fd) {
    this.fg = fg || null;
    this.fd = fd || null;
    this.info = info;
};
```




```
var wiki = new Node(1,
                new Node(2, new Node(4),
                            new Node(5, new Node(7),
                                        new Node(8))
                ),
                new Node(3, null,
                            new Node(6, new Node(9))
                )
);
```

feuille

pas de fils droit

pas de fils droit

# Node.prototype.toString

print(wiki); // produit [object Object]

**note:** sur codeboot, écrire print( wiki.toString());

```
Node.prototype.toString = function() {

    if (this.isFeuille())
       return "node("+this.info+")";

    return "node(" +
            this.info + ","
            + ((this.fg)? this.fg : "-")
            + ","
            + ((this.fd)? this.fd : "-")
            + ")";
};
```

appelle toString … il s'agit donc d'une version récursive

**Quiz: pourquoi?**
**réponse: à cause du + de concaténation**

si le fils droit n'est pas `null` (`null` est un *falsy*)

print(wiki.toString()); //produit maintenant:
node(1,node(2,node(4),node(5,node(7),node(8))),node(3,-,node(6,node(9),-)))

# Node.prototype.isFeuille

```
/**
 * @return true si le noeud n'a pas de fils
 */
Node.prototype.isFeuille = function() {
    return (this.fg === null) && (this.fd === null);
};
```

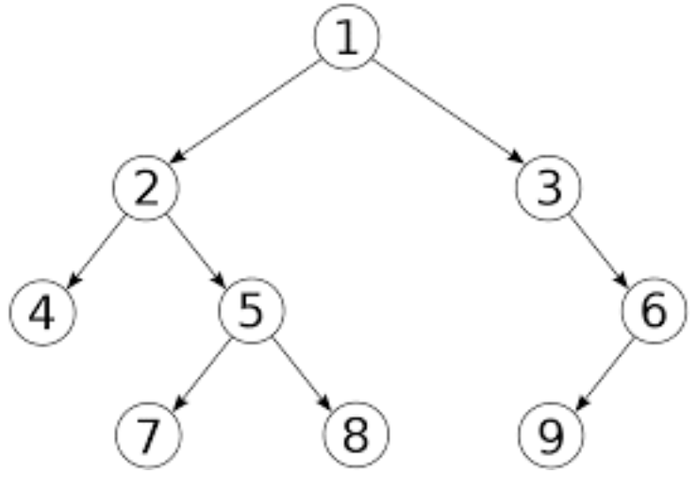
Tant qu'à y être …

```
/**
 * @return true si les deux fils existent
 */
Node.prototype.hasTwoChildren = function() {
    return (this.fg !== null) && (this.fd !== null);
};
```

# Parcours d'arbres (binaires)

*Dans un parcours, tous les nœuds de l'arbre sont visités. Le traitement associé à un noeud consiste ici à consommer (ex: afficher) l'information du noeud (propriété `info`). On visitera les fils en commençant par le fils gauche, puis le fils droit. Voici 3 parcours classiques et l'ordre des traitements associés:*

- **préfixe** (*pre-order*): chaque noeud est visité avant que ses enfants ne le soient
  - 1 2 4 5 7 8 3 6 9
- **postfixe** (*post-order*): chaque noeud est visité après ses enfants
  - 4 7 8 5 2 9 6 3 1
- **infixe** (*in-order*): chaque nœud est visité après son enfant gauche et avant son enfant droit.
  - 4 2 7 5 8 1 3 9 6



# Parcours

```
/**
 * @return tableau des informations rencontrees lors d'un parcours en profondeur prefixe
 */
Node.prototype.prefixe = function() {

    var res = [];
    var helper = function(p) {
                if (p !== null) {
                    res.push(p.info);
                    helper(p.fg);
                    helper(p.fd);
                }
    };

    helper(this);
    return res;
};
```

this pointe sur le noeud à parcourir (la racine)

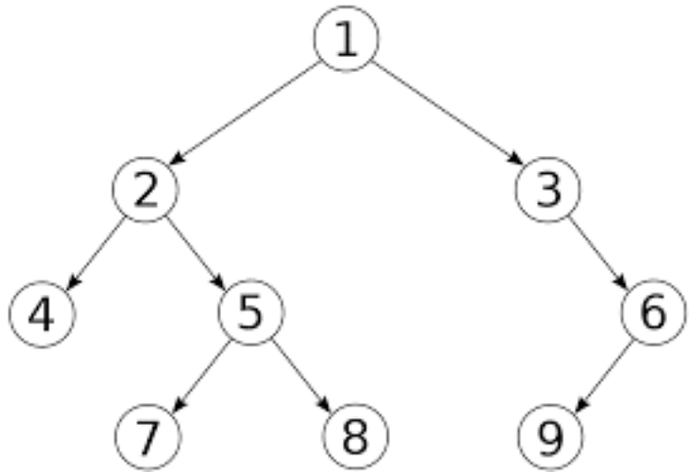
le traitement (ici) consiste à ajouter l'info d'un noeud  dans l'ordre où il est parcouru.

```
/**
 * @return tableau des informations rencontrees lors d'un parcours en profondeur infixe
 */
Node.prototype.infixe = function() {

    var res = [];
    var helper = function(p) {
                if (p !== null) {
                    helper(p.fg);
                    res.push(p.info);
                    helper(p.fd);
                }
    };

    helper(this);
    return res;
};
```

```
/**
 * @return un tableau des informations rencontrees lors d'un parcours en profondeur postfixe
 */
Node.prototype.postfixe = function() {

    var res = [];
    var helper = function(p) {
                if (p !== null) {
                    helper(p.fg);
                    helper(p.fd);
                    res.push(p.info);
                }
    };

    helper(this);
    return res;
};
```

# Parcours

```
var wiki = new Node(1,
                new Node(2, new Node(4),
                            new Node(5, new Node(7),
                                        new Node(8))
                        ),
                new Node(3, null,
                            new Node(6, new Node(9))
                        )
                );
    print("prefixe(wiki):",wiki.prefixe().join(" "));
    print("postfixe(wiki):", wiki.postfixe().join(" "));
    print("infixe(wiki):", wiki.infixe().join(" "));
```




**Produit:**

```
prefixe(wiki): 1 2 4 5 7 8 3 6 9
postfixe(wiki): 4 7 8 5 2 9 6 3 1
infixe(wiki): 4 2 7 5 8 1 3 9 6
```

# C’est terminé

- Il y a beaucoup de choses à dire et faire sur les arbres,
- mais cela fera l’objet d’autres cours.

- Pour les plus enthousiastes (il faut bien sûr accepter la récursivité…), tentez d’écrire:
  - une fonction qui retourne la profondeur d’un arbre
    - ici 3
  - une fonction qui cherche une valeur dans un arbre
    - wiki.exists(4) vaut `true`
  - une fonction qui retourne le nombre de noeuds dans un arbre
    - wiki.nbNodes() vaut 9