Exceptions : Rappel

```
try {
    res = 2 + f(x)
} catch (e) {
    res = e+5
}
function f (x) {
    if (x < 0) throw x;
    if (x == 0) return x;
    return 1+f(x-5);
}
```

Remarques

- Le `catch` rattrape le throw.
- L'exception est rattrapé à travers de multiples appels de fonctions.
- La pile est encombrée quand l'exception est lancée.
- Tous les calculs en cours sont interrompus.

Si `x` est divisible par 5 alors `res` est son diviseur plus 2, sinon son reste.

Exceptions : Trois solutions

Table d'exceptions

- aussi appelé 0-cost (optimal si pas d'exception),
- ajoute d'une table : l. de code → que faire si exception.
- Compilateur lourd à coder.
- Empêche des optimisations brutales

Utilisé par Java, mais de moins en moins choisi.

"mode" exception

- flag d'exception,
- double jeu d'instructions (exécute ou non selon le flag),
- lent si non optimisé,
- difficile à optimiser

Historique, peut s'implémenter au niveau softwear (bibliothèque/framework)

continuation d'erreur

- similaire au retour de fonction,
- si exception dépiler jusque trouver la continuation,
- difficile à optimiser si grosse pile,
- demande un "typage" de la pile d'appel.

De plus en plus utilisé (LLVM...)

Les continuations d'erreurs

En vrais une continuation contient

- un pointeur de code,
- un contexte à rétablir,
- un "type" (pour nous : retour ou erreur),

certains langages ont aussi des continuations de backtrack (p.ex.: ocaml avec lwt).

Instruction `Throw`

Sauvegarde la tête de pile,
Dépile tout jusque la prochaine continuation d'erreur,
Rétablit l'état (PC et CC) de la continuation d'erreur,
Remet la tête de pile.

Intermezzo : instruction `Return`

Dificile de garder sa pile “propre”

Les instructions font un “Pull” ou un “Pop” sur la pile, mais pas forcément celui qu'il faut.
Exemple :

```
42; y = 2 + (x=3);
```

Il faudrait utiliser `Copy` ou `Drop` quand ça arrive (ou utiliser des instructions alternatives)

L'instruction `Return` nettoie la pile

Pour éviter les bugs en début de projet, `Return` fait comme `Throw` : elle cherche la prochaine continuation de retours.

Encodage naïf

$$[\![\texttt{throw } e]\!] = \begin{array}{|l} [\![e]\!] \\ \texttt{Throw} \end{array}$$

$$[\![\texttt{try } c_1 \texttt{ catch } (n)\ c_2]\!] = \begin{array}{|l} \texttt{Catch} * \\ \quad [\![c_1]\!] \\ \texttt{Drop} \\ \texttt{Jump} * \\ \texttt{SetVar n} \\ \quad [\![c_2]\!] \end{array}$$

Catch off	$\texttt{Push}(\texttt{NewContinuation}\{\texttt{cont} = \texttt{CC},\ \texttt{code} = \texttt{PC} + \texttt{off} + 1,\ \texttt{err} = \texttt{true}\})$	pile	#t:pile

Exemple

```
try {
    res = 2 + f(x)
} catch (e) {
    res = e+5
}
function f (x) {
    if (x < 0) throw x;
    if (x == 0) return x;
    return 1+f(x-5);
}
```

```
NewClo 16
DecArg x
SetVar f
Catch 7
  CsteNb 2
  GetVar f
  StCall
  AddiNb
  SetVar res
Drop
Jump 5
SetVar e
  GetVar e
  CsteNb 5
  AddiNb
  SetVar res
Halt
```

```
#function f
GetVar x
CsteNb 0
LwStNb
ConJmp 2
  GetVar x
  Throw
GetVar x
CsteNb 0
Equals
ConJmp 2
  GetVar x
  Return
CsteNb 1
GetVar f
StCall
```

```
GetVar x
 CsteNb 5
 SubsNb
 SetArg
 Call
 Return
```

Problèmes à régler

Bug 1: Bien nettoyer sa pile

L'instruction `Drop` impose de faire attention à nétoyer sa pile correctement dans le bloc "try".

Bug 2: Variable d'erreur visible globalement

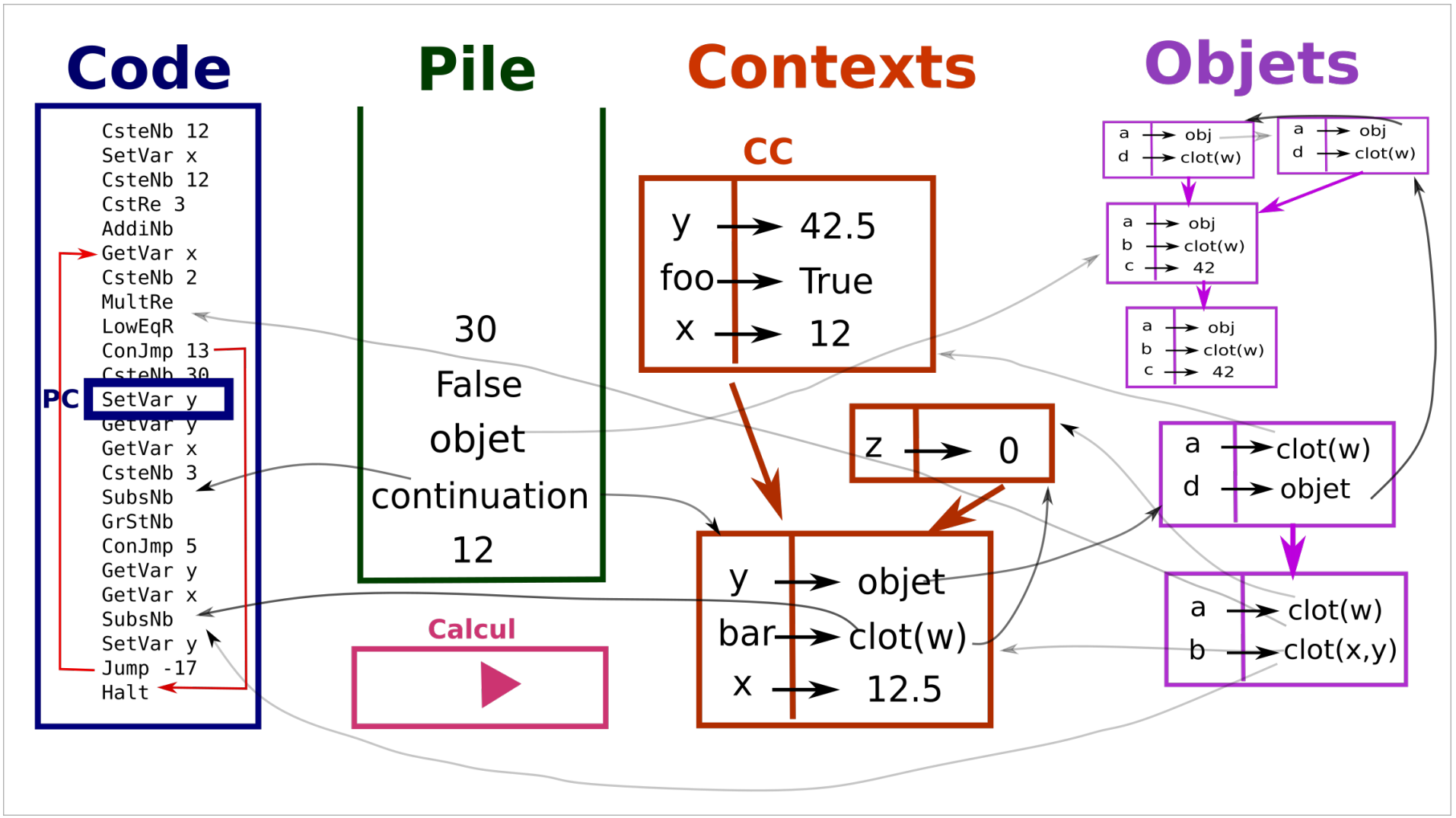
Il serait mieux de rendre x visible uniquement dans c_2 et non dans le contexte global.

$$[\![\texttt{try}\ c_1\ \texttt{catch}\ (x)\ c_2]\!] = \left|\begin{array}{l}\texttt{Catch}\ * \\ \quad [\![c_1]\!] \\ \texttt{Drop} \\ \texttt{Jump}\ * \\ \texttt{Local} \\ \texttt{SetVar x} \\ \quad [\![c_2]\!] \\ \texttt{Global}\end{array}\right.$$

Instructions utiles :

Local	$CC = NewContext(CC)$
Global	$CC = CC.oldContext$

Objets
(Attention: objets JS ≠ objets Java)



Les objets sont des valeurs

Un objet aussi est implémenté par un dictionnaire

Attributs et méthodes au même niveau et pointe sur la valeur associée.

```
{ attr1 : 42,
  attr2 : 256,
  meth1 : function (x)
            {return x+1;}
}
```

```
NewObj
CsteNb 42
SetObj attr1
CsteNb 256
SetObj attr2
NewClo 3
DecArg x
SetObj meth1
Halt
```

```
# function meth1
GetVar x
CsteNb 1
AddiNb
Return
```

NewObj	Push(NewObjet{})	pile	#o:pile
SetObj nom	Pull.cont.Set(nom, Pop)	#v:#o:pile	#o:pile

Assignation et récupération dans un objet

```
z = {
  a : 42
  b : 0
  }
z.b = 1 + z.a;
```

```
#créer z    #z.b = z.a+1;
NewObj      GetVar z
CsteNb 42   CsteNb 1
SetObj a    GetVar z
CsteNb 0    GetObj a
SetObj b    AddiNb
SetVar z    SetObj b
```

GetObj nom	Push(Pop.Get(nom))	#o:pile	#v:pile
SetObj nom	Pull.cont.Set(nom, Pop)	#v:#o:pile	#o:pile

GetObj et SetObj sont dans un objet

Ce sont les mêmes que GetVal et SetVal mais dans un objet sur la pile au lieu du CC.

Le problème du mot clé `this`

```
z = {
a : 42,
getA : function (x)
    {return this.a;}
};
z.getA();
```

```
#créer z
NewObj
CsteNb 42
SetObj a
NewClo 12
DecArg x
SetObj getA
SetVar z
```

```
#z.getA()
GetVar z
 Copy
GetObj getA
StCall
Swap
SetIn this
StCall
```

```
Call
Halt
 #getA
GetVar this
GetObj a
return
```

L'opérateur "." de `ob.var` dépent du type de retour

sur une clôture, il faut ajouter `this` dans le contexte,

sur une autre valeur on ne fait rien.

Le problème du mot clé `this` : en JS, le `this` fait référence à l'objet dont est tiré la clôture courrante...

```
o1 = { a:0,
       f:function(){return this.a}};
o2 = {a:42, f:o1.f}
o2.f();
```

Le problème du mot clé `this` : en JS, le `this` fait référence à l'objet dont est tiré la clôture courrante...

```
o1 = { a:0,
       f:function(){return this.a}};
o2 = {a:42, f:o1.f}
o2.f();
```

$\longrightarrow 42$

Le problème du mot clé `this` : en JS, le `this` fait référence à l'objet dont est tiré la clôture courrante...

```
o1 = { a:0,
       f:function(){return this.a}};
o2 = {a:42, f:o1.f}
o2.f();
```

$\longrightarrow 42$

```
#créer o1     #créer o2     #o2.f()
NewObj        NewObj        GetVar o2
CsteNb 0      CsteNb 42     Copy          #code f
SetObj a      SetObj a      GetObj f      GetVar this
NewClo 21     GetVat o1     StCall        GetObj a
SetObj f      Copy          Swap          return
SetVar o1     GetObj getA   SetIn this
              StCall        Call
              Swap          Halt
              SetIn this
              SetObj f
              SetVar o2
```

Les classes historiquement : Sucre syntaxique pour un constructeur

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
}
new Maclasse();
```

$\simeq$

```
function new_MaClasse (){
  rez = {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
    }       }
  return rez;
}
new_MaClasse();
```

Les classes historiquement : Sucre syntaxique pour un constructeur

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
  constructor(x){
    this.attr2 = x+a;
    }
}
new MaClasse(64);
```

$\simeq$

```
function new_MaClasse (x){
  rez = {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
    }         }
  rez.attr2 = x+a;
  return rez;
}
new_MaClasse(64);
```

Conséquence : environement lié à l'endroit où est créée la classe

```
function f(x) {
  y=42
  class MaClass(){
    x = y;
    f() {return y++;}
  }
}
maclasse = f(42);
ob1 = new maclasse();
ob1.f();
ob2 = new maclasse();
ob1.x;
ob2.x;
```

Conséquence : environement lié à l'endroit où est créée la classe

```
function f(x) {
  y=42
  class MaClass(){
    x = y;
    f() {return y++;}
  }
}
maclasse = f(42);
ob1 = new maclasse();
ob1.f();
ob2 = new maclasse();
ob1.x;                    ⟶ 42
ob2.x;                    ⟶ 43
```

Problème : Classes avec beaucoup de méthodes

Multiplication par le nombre de constructeur

Pour chaque constructeur, on réécrit toutes les méthodes...

Initialisation lentes

À chaque execution d'un constructeur, on réinitialise toutes les clôtures.

Espace inneficace

Une clôture stoquée pour chaque méthode dans chaque objet.

Exemple : Dans une liste chaînée, chaque noeud contiendrait plus de 70 clôtures à initialiser à chaque insertion !

Classes JS de nos jours : Prototype contenant un constructeur

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth1(){
    this.attr1++;
  }
}
new Maclasse();
```

$\simeq$

```
MaClasse = {
  meth1 : function (){return 128;}
  meth1 : function (){this.attr1++;}
  constructor : function {
    rez = {
      __proto__ : MaClasse,
      attr1 : 42,
      attr2 : undefined
    }
    return rez;
  }
}
MaClass.constructor();
```

Wraper objet autours des classes/fonctions

En JS : une classe est un objet

- un attribut `nom` (non spécifié, pas forcément le nom de a classe...)
- un attribut `prototype` (notion définie précédement)
- potentielement d'autres champs

une fonction aussi...

- un attribut `nom` (pas forcément le nom dans votre environement...)
- une clôture invisible
- potentielement d'autres champs

avec un lien vers le prototype des fonctions.

... de manière à ce qu'une classe ressemble une fonction

Le prototype d'une classe est le même que celui de la fonction.

Le but est uniquement la rétrocompatibilité (et la non-exposition des clôtres).

Wraper objet autours des classes/fonctions

Dans le projet

- Une classe doit être un objet avec un champ `prototype`.
- Les fonctions restent des clôtures.

L'attribut `nom` n'est pas obligatoire mais utile pour débugguer.
De même que l'attribut `length` qui contient l'arrité du constructeur.

Avec Wraper

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth1(){
    this.attr1++;
  }
}
```

$\simeq$

```
MaClasse = {
  name : ``MaClasse'';
  prototype : {
    meth1 : function (){return 128;}
    meth1 : function (){this.attr1++;}
    constructor : function {
      rez = {
        __proto__ : MaClasse.prototype,
        attr1 : 42,
        attr2 : undefined
      }
      return rez;
    }
  }
  __proto__ : prototype_des_fonctions;
}
```

JSMachine

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth1(){
    this.attr1++;
  }
}
new MaClasse();
```

```
NewObj
NewObj
NewClo 14
SetObj meth1
NewClo 14
SetObj meth2
NewClo 19
SetObj constructor
SetObj prototype
SetVar MaClasse
GetVar MaClasse
GetObj prototype
FrmPrt
GetObj constructor
StCall
Call
Halt
```

```
#meth1
CsteNb 128
Return
#meth2
GetVal this
Copy
GetObj attr1
CsteNb 1
AddiNb
SetObj attr1
Return
# constructeur
GetVar this
CsteNb 42
SetObj attr1
CsteUn
SetObj attr2
Return
```

Héritage : Imbrication des Prototypes

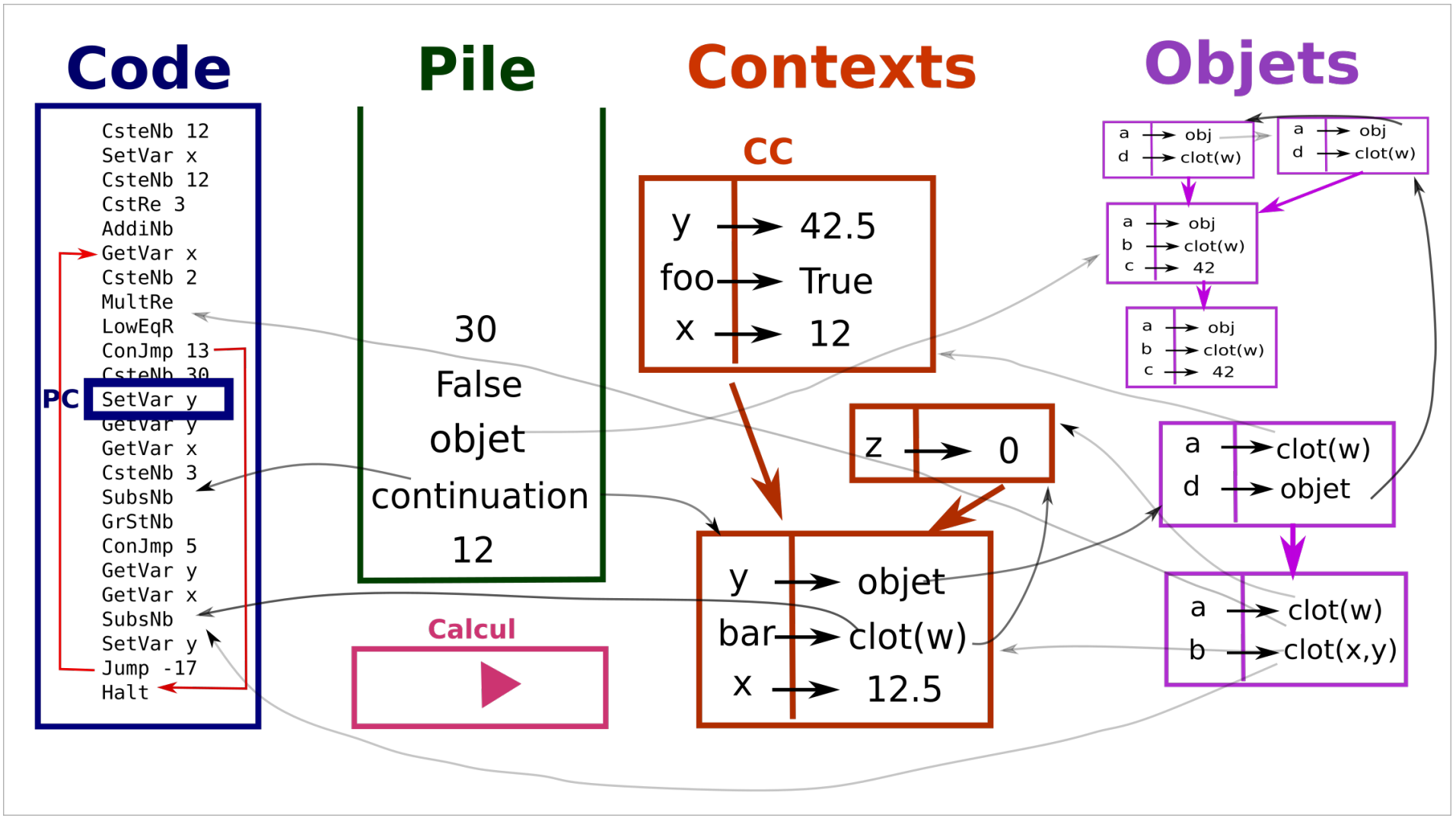
Rappel : le prototype est un objet

Il peut donc avoir un proto.

On peut donc avoir une liste chaînée de proto

Chaque descente correspond au prototype héritée d'un ancètre de la classe.

Objets


Code
CsteNb 12
SetVar x
CsteNb 12
CstRe 3
AddiNb
GetVar x
CsteNb 2
MultRe
LowEqR
ConJmp 13
CsteNb 30
PC
SetVar y
GetVar y
GetVar x
CsteNb 3
SubsNb
GrStNb
ConJmp 5
GetVar y
GetVar x
SubsNb
SetVar y
Jump -17
Halt
Pile
30
False
objet
continuation
12
Calcul
Contexts
cc
y 42.5
foo True
x 12
z 0
y objet
bar clot(w)
x 12.5
Objets
a obj
d clot(w)
a obj
d clot(w)
a obj
b clot(w)
c 42
a obj
b clot(w)
c 42
a clot(w)
d objet
a clot(w)
b clot(x,y)

Héritage : Imbrication des Prototypes

```
class Foo extends Bar {
  a;
  constructor(x){
    this.a=x;
  }
}
```

$\simeq$

```
MaClasse = {
  name : ``MaClasse'';
  prototype : {
    constructor : function {
      rez = {
        a : 42,
        __proto__ : MaClasse.prototype
      }
      return rez;
    },
    __proto__ : Bar.prototype
  }
  __proto__ : prototype_des_fonctions;
}
```

JSMachine

```
class Foo extends Bar {
  a;
  constructor(x){
    this.a=x;
  }
}
new Foo();
```

```
#class Foo
NewObj
GetVar Bar
GetObj prototype
FrmPrt
NewClo 9
SetObj constructor
SetObj prototype
SetVar Foot
```

```
GetVar Foo
GetPrt
FrmPrt
GetVar constructor
Call
Halt
# constructeur
GetVar this
GetVar x
SetObj a
Return
```

Petit mensonge sur le attributs hérités (manque de temps)

Comportement par défaut des "get" iddentiques

GetVar x

- cherche la variable `x` dans l'environement locale
- si non trouvée remonte au niveau + global
- si non trouvé à la racine rend undefined.

GetObj x

- cherche l'attribut/methode localement
- si non trouvé remonte le proto
- si non trouvé à la racine rend undefined.

Comportement par défaut des "set" différents

SetVar x

- cherche la variable `x` dans l'environement locale
- si non trouvée remonte au niveau + global,
- si non trouvé insert à la racine,
- si trouvé modifie la version trouvée.

SetObj x

- cherche l'attribut/methode localement,
- si trouvé modifie la version trouvée,
- si non trouvé insert à l'origine,
- ne remonte jamais les niveaux de prototypes

Implémentations possibles des dictionnaires et Choix en pratique

Discution selon le temps

Disclaimer :

Ces slides (et ce cours) n'ont pas pour but de présenter la sémentique officiel de JS, mais une présentation pédagogique de consceptes transversaux en s'appuiant sur JS.

Pour connaitre la sémentique officielle de JS, voir la norme ECMA : https://262.ecma-international.org/