EcmaScript et ISIS-Fish

L'ecmascript est comme son nom l'indique un langage dit de script, c'est à dire qu'il n'est pas compilé avant son exécution (Pareil que R, VisualBasic...). L'EcmaScript va être utile dans ISIS-Fish à principalement trois niveaux:

pour spécifier des équations (par exemple, équation de croissance, de mortalité naturelle, de migration, de reproduction)

pour modifier les valeurs de certains paramètres sans avoir à modifier la base de données à travers de scripts de présimulations. Ceci est particulièrement intéressant dans le cadre d'analyse de sensibilité

pour coder des règles de gestion.

Si la connaissance de l'EcmaScript n'est pas complètement indispensable à l'utilisation de ISIS-Fish, avoir quelques notions s'avère fort utile.

1 Les bases de l'EcmaScript

1.1 Les commentaires en EcmaScript

Comme pour tout langage de programmation, il est particulièrement intéressant de décrire (documenter) un code afin de le rendre plus compréhensible pour un autre utilisateur ou lors d'une réutilisation postérieure. Le commentaire doit donc être un texte apparaissant dans le code mais non exécuter lors de l'exécution. Un commentaire peut-être ajouté de deux manières en Ecmascript comme détaillé dans l'exemple.

Ex:

var a=2 //tout ce qui figure sur cette ligne après le signe est un commentaire

*/je peux mettre un commentaire

sur plusieurs lignes entre étoile antislash

et antislash étoile*/

var b=3;

1.2 Les types primitifs de données – les variables

En Ecmascript, les variables quelque soit leur type se déclarent d'une seule et même façon. L'instruction

var NouvelleVariable;

crée une nouvelle variable appelée NouvelleVariable (attention: les majuscules comptent).

Il existe cinq types de données primitives en EcmaScript: undefined, null, Boolean, Number et String, admettent en revanche plusieurs valeurs différentes.

- undefined: variable qui n'a pas encore de valeurs
- null: variable vide ou fonction non gérée
- string: chaîne de caractères
- booléen: true ou false
- numeric: un nombre

L'affectation d'une valeur à une variable se fait avec l'opérateur =.

1.3 Opérations sur les variables primitives

- Opérations mathématiques et concaténations

Les opérations mathématiques classiques telles que +, -, *, / sont définies sur les variables numériques, ainsi que l'opérateur % qui donne le reste de la division euclidienne
ex:
var a=2;
var b=5;
a+b; //renvoie 7
a-b; //renvoie -3
b%a //renvoie 1

++ et – permettent d'incrémenter ou de diminuer d'une unité une variable numérique.
Ex:
var a=2
a++; //a vaut 3
a--; //a vaut 2

+= -= /= et *= prennent la valeur de gauche y additionne (ou soustrait, ou divise ou multiplie) la valeur de droite et affecte le résultat à la variable de gauche.
Ex:
var a=2;
a+=3; //a vaut 5

Pour les chaînes de caractères, l'opérateur + (+= rajoute à la fin) permet de concaténer plusieurs chaînes.

- Opérateurs de comparaisons

Les opérateurs de comparaisons renvoient un booléen si la comparaison est vraie ou fausse.

Opérateurs	***Type de comparaison***
>	Supérieur strict
<	Inférieur strict
<=	Supérieur ou égal
>=	Inférieur ou égal
==	Strictement égal
!=	Différent de

Opérateurs logiques
Un opérateur logique est un opérateur qui combine deux booléens et renvoie un booléen. Ils sont particulièrement intéressants dans les conditions de contrôle de flux (

	syntaxe	***Opérande 1***	***Opérande 2***	***Valeurs renvoyées***
Et logique	Opérande 1 && opérande 2	true true false false	true false true false	true false false false

	syntaxe	*Opérande 1*	*Opérande 2*	*Valeurs renvoyées*
Ou logique	Opérande 1 || opérande 2	true true false false	true false true false	true true true false

1.4 Le contrôle de flux

Dans les programmes, il est en général intéressant de pouvoir exécuter certaines instructions (on parle de blocs d'instructions) uniquement si une ou plusieurs conditions sont satisfaites. En ecmascript deux types de méthodes vont permettre cela

- Les blocs

Un bloc est une suite d'instructions comprises entre deux accolades.
Ex:

```
{//début du bloc
var a=3;
a++;
}// fin du bloc
```

- if, else et else if

Les méthodes, if, else if et else suivent la logique si, sinon si, sinon. Plusieurs conditions peuvent éventuellement être imbriquées.

Ex:

```
if (a==1){
        //bloc si a est égal à 1
        if (b<2){
         b++; //instruction réalisée si en plus b est inférieur à 2
        }
        else{
         b--; //instruction réalisée si en plus b est supérieur ou égal à 2
        }
} //fin du bloc if
else if (a>0 && b>0){
        b++;    //bloc réalisé si a est différent de 1 mais supérieur à 0 et que b est inférieur à 0
        a++;
}
else {
        b--; //bloc réalisé si a est différent de 1, et si a et b ne sont pas strictement positifs
}
```

On note dans cet exemple l'utilisation des opérateurs logiques. D'autre part, pour la clarté du code, il est conseillé de décaler les blocs (deux espaces ou une tabulation) selon leur niveau d'imbrication.

- switch

La méthode switch va permettre de réaliser un bloc d'instructions selon la valeur que va prendre une certaine variable.

Ex:

```
switch (Option){ //
        case 1:
         a++;       //ce bloc est réalisé si Option vaut 1
         break;
        case 2:
         b++;
         break;       //ce bloc est réalisé si Option vaut 2
}
```

Chaque bloc doit se terminer par l'instruction break;.

1.5 Structures de boucles

Dans un programme, il est souvent utile de répéter un certain nombre de fois un bloc d'instructions. On réalise alors des boucles.

- Boucles for

Les boucles for font appel à un compteur, le bloc étant réalisé tant que le compteur n'a pas atteint une certaine limite. On spécifié également comment évolue le compteur à chaque itération

Ex:

```
var a=1;
for (var i=1; i<3; i++){/*le compteur i vaut 1 en début de simulation, on renouvelle l'opération tant
                    que i<3 et i augmente d'une unité à chaque opération*/
        a++;
} //en sortie a vaut 3
```

- boucles while

Le fonctionnement de cette méthode est analogique à la boucle for, mis à part le fait que l'on ne définit pas directement de compteurs. Le bloc est répété tant que la condition est respectée

Ex:

```
var a=1;
var i=1;
while (i<3){
        a++;
        i++;
} // le résultat est ici exactement le même que dans l'exemple précédent
```

- boucles do...while

Le principe est le même mais la condition n'est vérifiée qu'à la fin de l'itération, le bloc est donc exécuter au moins une fois.

Ex:

```
var a=1;
var i=1;
do{
        a++;
        i++;
} while(i<3); // attention au ;
```

2 La structure de la base de données et les classes d'objet

2.1 Les classes d'objets

Nous avons vue dans le chapitre précédent que l'EcmaScript contenait 5 types de variables primitifs. Cependant vous vous rendrez vite compte que dans ISIS, on utilise beaucoup d'autres types de variables. Pour cela il n'est pas inutile d'introduire un peu le langage orienté objet.

En programmation « classique », un programme se compose de fonctions et de variables. Toute l'architecture du programme repose donc sur une succession d'appels à différentes fonctions. On peut en plus créer ce qu'on appelle des structures qui sont de nouveaux types de variables composés de différents « champs ». Par exemple dans un programme, on pourrait avoir besoin de créer un type de variable Personne contenant deux champs: un numérique Taille et un numérique Poids.

En langage orienté objet, ce qui va guider la structures d'un programme n'est plus les fonctions à utiliser mais les type de variables sur lesquelles on va travailler, on parle alors de classes. Pour reprendre l'exemple précédent Personne pourrait-être une classe avec deux attributs (les « champs ») taille et poids et des méthodes (fonctions) qui s'appliquent à cette classe (par exemple SaisirPoids(), SaisirTaille()...

2.2 ISIS et la base de données

Pour voir la structure d'une région sous ISIS, il est intéressant de jeter un oeil à l'UML:

http://isis-fish.labs.libre-entreprise.org/devel/IsisFishModel.png

Chacune des boîtes du schéma correspond à une classe d'objets. ISIS reposant sur une base de données, à chaque classe d'objets correspond une classe Factory (qui correspondrait dans une base de données à une table) dans lequel chaque objet de la classe est stockée.

Par exemple pour la classe Population, on trouve une PopulationFactory dans lequel sont stockés toutes les populations de la région. Ces Factory vont permettre de récupérer des objets de la base de données.

Ex:

var Thon=PopulationFactory.findByName('Thon'); //On récupère la population dont le nom est //Thon

Un descriptif des différentes classes disponibles dans ISIS est consultable en ligne:

http://isis-fish.labs.libre-entreprise.org/api/isis-fish/version2/index.html (api du simulateur)

http://isis-fish.labs.libre-entreprise.org/api/codelutin/index.html (api des librairies annexes, essentiellement les dbCollections qui sont des containers)

http://lutinmatrix.labs.libre-entreprise.org/apidocs/index.html (api des matrices)

Quelques remarques:

- Pour créer un objet d'un certain type (par exemple Population ou MatriceND), on doit le créer à partir de la Factory correspondant (PopulationFactory ou MatrixFactory) via la méthode create.
- Quand on récupère un objet, on travaille sur l'objet lui-même, donc si après l'avoir récupéré je le modifie par un script, l'objet sera modifié dans la base de données pour la simulation dans laquelle s'applique le script. (voir les scripts de simulation)
- Pour les matrices, il est parfois intéressant de travailler sur la matrice (par exemple, multiplier une sous-dimension classe de la matrice capture en nombre par le poids de la classe pour avoir une matrice en poids) sans pour autant la modifier dans la base de données (on veut quand même garder la matrice ne nombre). On doit alors travailler sur une copie de la matrice qui s'obtient en faisant

var copie=MatrixFactory.create(original);

3 Concrètement à quoi ça sert?

3.1 Dans l'interface de saisie

Comme vous avez pu le remarquer, certains paramètres de ISIS sont renseignés non pas par une simple valeur mais au travers d'une équation. Celle ci doit permettre de renvoyer un résultat à partir des arguments que Benjamin nous a gracieusement fourni. Rentrons un peu plus dans le détail

- Schéma général

Pour le moment on doit écrire ça de cette façon:

result = ECMAScript(#

//corps de la fonction: DOIT renvoyer un résultat

#);

result

Ne vous embêtez pas trop à comprendre ce que veut dire ce qu'il y a autour du corps, dans la version 3 on devrait pu avoir à mettre tout ça.

- Equation de croissance
  - arguments disponibles: Benjamin qui est fort gentil nous donne le droit d'utiliser l'âge qui est comme son nom l'indique un age. Attention, c'est un age en mois, ne pas oublier de diviser par 12 si on veut un mois en année...
  - ce qu'on doit retourner: la longueur correspondant à l'âge transmis en argument
  - un exemple: une belle Equation de Von Bertalanffy

```
result=ECMAScript(#
Linf = 100.0;
K = 0.0010;
T0 = -1.0;
(Linf*(1.0-Exp((-K*((age-T0)*12.0)))));
#);
result
```

- Inverse Croissance: la même sauf que c'est l'inverse
  - arguments disponibles: Ca concerne les populations structurées en longueur. Ici Benjamin nous envoie la longueur
  - ce qu'on doit retourner: l'âge en mois à la longueur transmise en argument
  - un exemple: une belle Equation de Von Bertalanffy inversée

```
result=ECMAScript(#
Linf = 48.0;
K = 0.14;
t0 = 1.0;
((-t0-((1.0/K)*Ln((1.0-(longueur/Linf)))))*12.0);
#);
result
```

- Equation de Reproduction
  - un bon paquet, c'est Byzance...

  - (pop) la population sur lequel on travaille
  - (N) la matrice N qui donne le nombre d'individu par Classe et par zone
  - (mois) le mois auquel on travaille
  - (prepro) la proportion de reproduction pour le mois courant
  - (zoneRepro) la liste des zones de reproduction
  - (classes) la liste des classe de la pop
  - (zones) la liste de zone de la pop
  - ce qu'on doit retourner: en fait on s'en fout, on doit juste remplir une matrice result structurée en zone qui contient le nombre d'oeufs produits dans la zone à la date courante
  - un exemple: une équation fécondité*effectif de la zone

```
r = ECMAScript(#
```

```
for(var izone=0; izone<zoneRepro.size(); izone++){
        var zone=zoneRepro.get(izone); // on prend le izone-ième élément de la liste
        var tot = 0.0; //on veut compter le nombre d'oeufs total produit dans la zone
        for (var iclasse=0; iclasse=pop.getClasses();iclasse++){
//on va sommer tous les oeufs produits par toutes les classes
         var classe=pop.getClasses().get(iclasse);
         tot=tot+classe.getCoefficientFecondite()*N.getValue(classe,zone);
        //on multiplie l'effectifs de la classe dans la zone par la fécondité et on rajoute à tot
        }
        //on a finit la boucle sur les classes, on a donc tous les oeufs produits
        result.setValue(zr, tot*prepro);
        //on remplit la matrice result, on multiplie par prepro pour prendre en compte la proportion
        //de reproduction pour le mois courant
}
// on retourne une valeur qui ne sert a rien
// puisque result sera utilisé
0;
#);
r
```

- Equation de Mortalite Naturelle: la dernière en date, elle marche toujours pas...
  - arguments disponibles: classe la classe courante ou -1 pour la classe pré-recrutée; zone, la zone courante, pop, la pop courante
  - ce qu'on doit renvoyer: tout simplement la mortalité naturelle de la classe dans la zone
  - un exemple: quand ça marchera...

- Equation de migration:
  - arguments disponibles: 'classe', 'zoneDepart', 'zoneArrive' et 'N'
  - ce qu'on doit renvoyer: une proportion
  - un exemple: quand j'aurai le temps

- La sélectivité
  - arguments disponibles: longueur ou age (toujours en mois)
  - ce qu'on doit renvoyer: une proportion
  - un exemple: au la belle sigmoïde

```
result=ECMAScript(#
        var SF=10;
```

```
        var L50=27;
        var beta=2*Ln(3)/SR;
        var alpha=-beta*L50;
        1/(1+Exp(-alpha-beta*longueur));
#);
result
```

3.2 Des scripts de présimulation

- Des scripts tout bête

Et ben à tout plein de choses! C'est un des grands atouts d'ISIS. Dans l'interface de lancement de simulation on peut rajouter un script de présimulations. Ce script va permettre de changer certaines valeurs de paramètres pour la simulation et uniquement pour la simulation. Ca évite quand on veut tester différentes valeurs d'aller dans l'interface de saisie, de changer la valeur, de sauver, de lancer la simu puis de retourner dans la saisie pour remettre la valeur normale.

Ex:

```
var Thon=PopulationFactory.findByName('Thon'); //décidément j'aime bien les thons
var classe0=Thon.getClasses().get(0); //j'ai récupéré la première classe du thon
classe0.setPoidsMoyen(13); //dans ma simu, le poids moyen vaudra 13
```

Comme ça ça paraît déjà pas mal intéressant, mais on peut faire encore plus fort!

- Vers les plans de simulation

Imaginez maintenant sur le même exemple que vous vouliez faire 3 simulations pour trois valeurs de poids moyen. On peut créer un tableau dans le script de présimulation, dans chaque case, on met un le code correspondant à un script, sous forme de chaîne de caractères et sans passer à la ligne

Ex:

```
new Array("var Thon=PopulationFactory.findByName('Thon');var classe0=Thon.getClasses().get(0);classe0..setPoidsMoyen(13);",
"var Thon=PopulationFactory.findByName('Thon');var classe0=Thon.getClasses().get(0);classe0..setPoidsMoyen(12);",
"var Thon=PopulationFactory.findByName('Thon');var classe0=Thon.getClasses().get(0);classe0..setPoidsMoyen(11);")
```

à chaque case du tableau correspondra une simulation avec les valeurs 13, puis 12 puis 11 de poids pour la classe 0.

Remarque, on peut aussi faire comme ça:

```
var result = new Array();
```

```
for (var i=1; i<=3; i++){
        var temp="var pop = PopulationFactory.findByName('Thon');var
classe0=Thon.getClasses().get(0);classe0..setPoidsMoyen(";
        switch (i){
         case 1:
                  temp+=13;
                  break;
         case 2:
                  temp+=12;
                  break;
         case 3:
                  temp+=11;
                  break;
        }
        temp+=");"
        result.push(temp);   //on met en fin de tableau la chaîne temp
}
result;
```

Remarque: temp est une caractères donc entourée de guillemets, si à l'intérieur on a aussi besoin de caractères (par exemple ici findByName('Thon')), il faut utiliser ' pour ne pas qu'il puisse y avoir de confusion.

bon Ok comme ça ça paraît un peu compliqué mais vous verrez que si un jour vous avez besoin de faire un vrai plan de simulations, c'est l'outil qu'il vous faut.

3.3 Les règles de gestion

Alors là, j'ai aucune envie de trop rentrer dans le détail car c'est un poil plus compliqué. Je pense que le plus simple est de regarder les règles déjà existantes et d'essayer de comprendre ce qui se fait. C'est sûr que si l'abruti qui avait codé ça avait un peu documenté son code ça vous faciliterait le boulot (on va dire que j'ai fait ça pour montrer que c'est important de documenter. En gros comment ça marche, la règle à des paramètres (Constructeur), à chaque pas de temps (les infos courantes sont récupérées dans paramètres), une boucle est réalisée sur les métiers pour voir si la règle s'applique au métier (Condition). Si oui, avant tout calcul du pas de temps, on applique des changements via ActionAvant (par exemple, si un TAC est atteint, arrêter de cibler l'espèce. A la fin du pas de temps, on réalise ActionAprès qui sont des modifs à faire une fois que les calculs du pas de temps sont réalisés (par exemple affecter des captures au rejet quand le tac est atteint)

En gros, y a 5 onglets:

- constructeur: permet de récupérer les paramètres que l'utilisateur devra saisir pour paramétrer la règle. Ce bout de code est exécutez avant le début de la simulation, on peut donc y placer des scripts de présimulations (voir CantonnementPresimu)
- Paramètres: récupère les infos courantes date, effectifs, métier courant...
- Condition: code pour juger si le métier courant (p.metier) est affecté. Retourne un booléen
- ActionAvant: Correspond aux modifications liées à l'application de la règle avant le calcul

de F. Doit retourner p.gestionMetier (cherchez pas à comprendre)

- ActionAprès: Correspond aux modifications liées à l'application de la règle après le calcul de F. Doit retourner p.gestionMetier (cherchez toujours pas à comprendre)

4 Quelques astuces

- writeln

on peut n'importe où dans le code utiliser la fonction writeln. Celle ci va noter dans les logs (mais si vous savez le fichier erreur.txt qui apparaît) la chaîne de caractères qui est entre parenthèses. C'est assez utile en particulier lors du débuggage. Par exemple quand ça plante, on peut mettre des writeln("x"); un peu partout. En voyant lesquels sont écrits dans les logs, on a une bonne idée d'où le code a planté. En vous baladant dans les règles de gestion, vous verrez que y en a un peu partout, et qu'on s'en sert pour vérifier que le code fait bien ce qu'on lui demande.

- Obtenir un élément dans une matrice

Il existe deux façon de récupérer un élément dans une matrice ou une liste, soit en fournissant ses coordonnées en entier, soit en fournissant les objets correspondant à la case. Pas clair? un exemple

imaginez qu'on est une matrice N(classe, zone) d'effectifs. Pour avoir une valeur on peut faire soit N.getValue(0,0); soit si on dispose de l'objet classe correspondant à la classe qui nous intéresse et l'objet zone correspondant à la zone qui nous intéresse N.getValue(classe,zone);

Vous verrez assez vite que cette seconde méthode est souvent bien utile...

Sur les matrices vous verrez qu'il y pour chaque dimension une liste dite Semantics que vous pouvez récupérer. En fait dans notre cas les Semantics de la dimension 0 seraient une liste contenant tous les objets classe sur lesquels j'ai de l'info dans ma matrice, les Semantics de la dimension 1 étant une liste contenant tous les objets zone sur lesquels j'ai de l'info dans ma matrice.

Ca aussi vous trouverez ça rapidement très pratique.

- les itérateurs

Je sais pas si vous savez mais imbriquer pleins de boucles c'est assez long. Benjamin a implémenté un truc vachement plus efficaces pour se balader sur les matrices où les dbcollections: les itérateurs. Là encore vous en trouverez un peu partout dans les codes écrits par votre humble serviteur, et vous trouverez de la doc sur l'api. C'est pas indispensable, mais quand vos codes sont longs, c'est bien pratique.