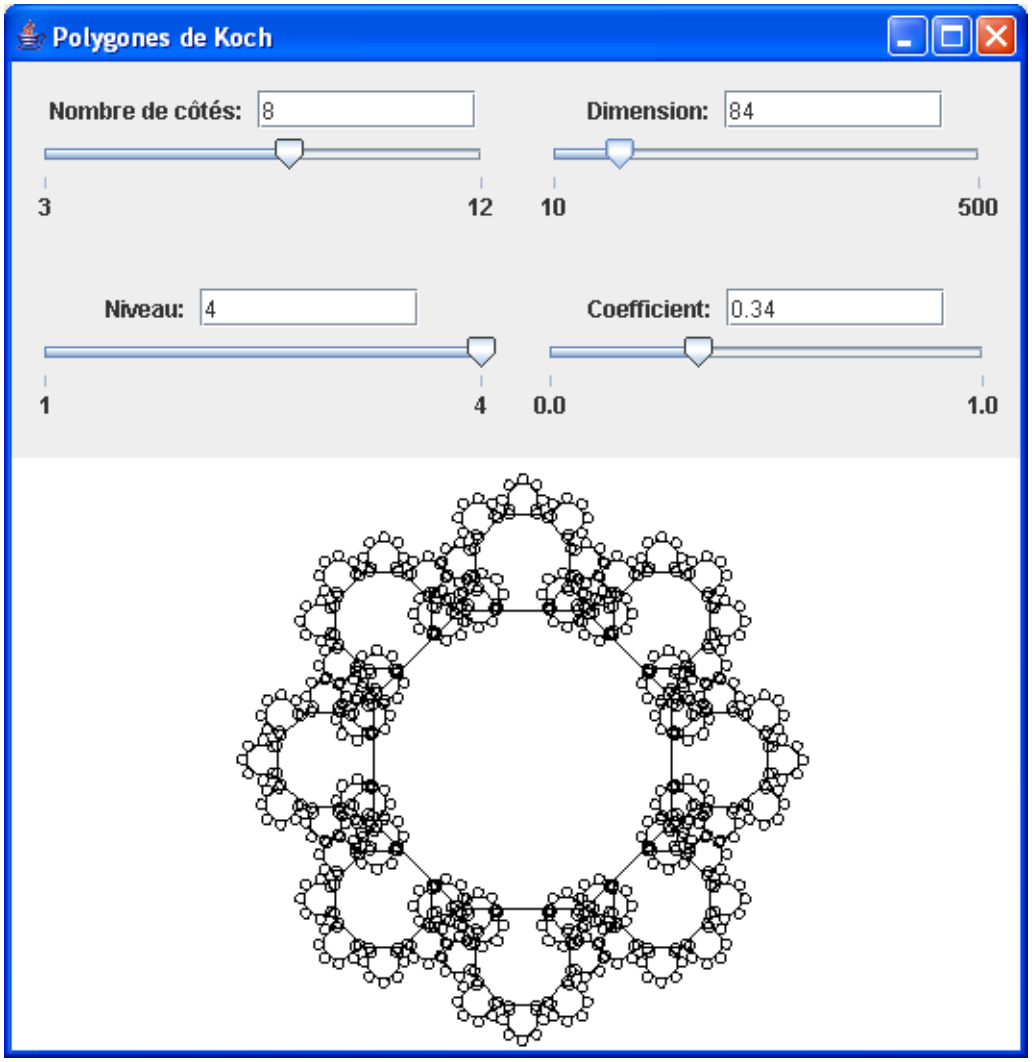
Expresso
Manuel de l'utilisateur



André Boileau
boileau.andre@uqam.ca

Maurice Garançon
garamau@gmail.com

Table des matières

1. Introduction

Expresso a vu le jour un peu par hasard. L'un des auteurs cherchait un langage de programmation pour initier ses étudiants, futurs enseignants en mathématiques, à la programmation. Ses exigences étaient simples, il fallait un langage puissant, multi-plateforme, facile à apprendre et gratuit. Pendant ce temps l'autre auteur découvrait Java et l'environnement de programmation BlueJ.
D'un côté Java fournissait le langage puissant, multi-plateforme, gratuit, en plus compatible avec le Web mais malheureusement trop complexe pour les besoins envisagés.
D'un autre côté BlueJ fournit un environnement de programmation Java multi-plateforme, simple à utiliser et doté de capacités largement suffisante pour nos besoins.
Si proches du but nous ne pouvions pas abandonner et c'est ainsi que nous avons décidé de créer Expresso qui est un langage basé sur Java qui permet cependant de programmer sans objets.
Expresso met à la disposition de l'utilisateur une interface configurable composé minimalement d'une page graphique dans laquelle on peut dessiner à l'aide de commandes de style « géométrie de la tortue » et aussi de commandes de style « géométrie analytique ». On peut aussi compléter cette interface, à l'aide de commandes élémentaires, en y ajoutant des menus, des boutons, des glissières, une zone de texte, en utilisant des fenêtres de dialogue ou en réagissant aux clics ou aux « glisser » de souris.
Ça donne une interface limitée mais facile à construire et qui, pour l'instant, nous semble assez complète pour nos besoins.
Nous avons dû faire des choix et le temps dira si ces choix étaient bons. De toute façon nous sommes ouverts à toute suggestion et aux critiques .. constructives.

2. L'environnement de programmation

La programmation avec Expresso se fait dans l'environnement gratuit BlueJ. Dans les paragraphes qui suivent nous expliquons comment se procurer BlueJ et comment le configurer pour rendre la programmation Expresso possible.

2.1. Comment se procurer « BlueJ »

BlueJ peut être téléchargé gratuitement à partir du site suivant:
http://www.bluej.org/download/download.html
On conseille d'utiliser la version 2.0.2 ou une version plus récente.
Pour Windows :
Il est nécessaire que Java soit installé. Il faut la version JDK5 ou une version plus récente (il faut une version appelée JDK pas JRE ou la version pour Netbeans).
Pour Macintosh :
Une version récente de Java est installée par défaut.

2.2. Comment installer « BlueJ »

Pour Windows il suffit de double cliquer l'installateur « bluejsetup.exe » et de suivre les instructions.
Pour Mac ..

2.3. Comment franciser « BlueJ »

Une fois BlueJ installé on peut le franciser, pour cela il faut trouver le fichier « **bluej.defs** ».
Pour Windows, si lors de l'installation de BlueJ vous avez accepté les paramètres proposés par défaut il doit se trouver dans « **C:/Program Files/BlueJ/lib/** ».
Pour Mac, il faut d'abord trouver l'icône de **BlueJ** qui normalement se trouve dans le dossier « **Applications** ». Une fois l'icône de **BlueJ** localisée, il faut la cliquer avec le bouton droit (Ou contrôle-clic du bouton gauche) et dans le menu contextuel qui apparaît choisir « **Afficher le contenu du paquet** ». On obtient alors un dossier « **Contents** » qui contient un dossier « **Ressources** » qui contient lui-même un dossier « **Java** ». Le fichier « **bluej.defs** » doit se trouver dans ce dernier dossier.
Le fichier « **bluej.defs** » doit maintenant être modifié. Pour cela il faut l'ouvrir dans un éditeur de texte et trouver la ligne

```
#bluej.langage=french
```

sur cette ligne enlever le symbole # pour quelle devienne

```
bluej.langage=french
```

et d'autre part ajouter le symbole # au début de toute autre ligne « bluej.langage=... » qui n'en possède pas.

2.4. Comment installer la bibliothèque « Expresso »

En téléchargeant « **Expresso** » on obtient plusieurs fichiers dont « **Expresso1.1.ja**r » (**1.1** indique la version courante au moment de l,écriture de ce manuel. Ça pourra avoir changé au moment ou vous procéderez à l'installation, ça n'a aucune importance!). La localisation de ce fichier sur le disque dur n'a pas d'importance mais nous conseillons de le placer dans le même dossier que **BlueJ** de façon à pouvoir le retrouver facilement.
Pour rendre « Expresso » disponible dans BlueJ il faut:
- ouvrir BlueJ
- sélectionner le menu Outil/Préférences

- dans la fenêtre qui apparaît cliquer sur le tab « Bibliothèques »
- dans le nouvel affichage cliquer sur le bouton « Ajouter »
- dans la fenêtre de sélection qui s'ouvre naviguer pour localiser « **Expresso1.1.jar** » et le sélectionner. Cliquer sur « Ouvrir » (ou « Choisir » selon le système utilisé).

C'est fait. Au prochain démarrage de **BlueJ**, **Expresso** sera disponible.

2.5. Comment installer les « templates »: « Expresso » et « AppletExpresso »

En téléchargeant « **Expresso** » on obtient plusieurs fichiers dont « **Expresso.tmpl** » et « **AppletExpresso.tmpl** », il reste à placer ces fichiers.
Localiser d'abord le fichier « **bluej.defs** » comme en 2.3 dépendant de la plateforme utilisée. Dans le même dossier que « **bluej.defs** » se trouve le dossier « french » (si vous utilisez une autre langue que le français choisissez le dossier correspondant). Dans ce dossier « **french** » il y a un dossier « **templates** » et dans ce dernier un dossier « **newclass** ». Il faut copier les fichier « **Expresso.tmpl** » et « **AppletExpresso.tmpl** » dans ce dossier « newclass ».
Une fois toutes ces étapes parcourues vous êtres prêt à programmer en « **Expresso** » à l'aide de **BlueJ.**

3. La programmation issue de Java

3.1. Généralités

Un programme « Expresso » est en fait un programme « **Java** », il s'en suit que « **Expresso** » utilise les même structures de base et suit la même syntaxe que Java. Notre intention, ici, n'est pas d'écrire un cours de programmation « Java », mais simplement de donner les éléments indispensables pour écrire des programmes « Expresso ».

3.1.1. Commentaires

Il est utile d'ajouter des commentaires dans un programme pour en faciliter la lecture par un être humain. Les commentaires permettent d'expliquer le code. Un commentaire ne fait rien, il n'est pas exécuté. Il y a deux façons d'écrire un commentaire :
- Un commentaire contenu sur une seule ligne est précédé par « // ».

```
// Ceci est un commentaire sur une ligne
```

- Un commentaire sur plusieurs lignes est précédé par « /* » et suivi par « */ »

```
/* Voici
un
commentaire
sur plusieurs
lignes */
```

3.1.2. Les noms de variables et de procédures

Pour nommer les variables et les procédures sans problèmes, commencez toujours par une lettre et n'utilisez que des lettres non accentuées et des chiffres. Évidemment les mots réservés comme **if**, **for**, **cos**, **sin**, etc. ne peuvent pas être utilisés.

Par convention (non obligatoire), en Java on commence toujours les noms de variables, et de procédures par une minuscule, et si le nom consiste en la juxtaposition de plusieurs mots, les autres

mots commencent avec une majuscule. Exemple : **maVariablePreferee**. Par contre on écrit les noms de constantes en majuscules et si le nom est constitué de plusieurs mots on les sépare par une barre de soulignement : **MA_CONSTANTE**.
Comme les commentaires, ces dernières conventions ne sont destinées qu'à faciliter la lecture des programmes par les humains.

3.2. Les types de données, les constantes et les variables

3.2.1. Les données

Pour écrire des programmes « **Expresso** » nous pouvons nous limiter aux 4 types de données suivantes :
Nombres entiers : exemple 5. Pour des raisons techniques ne sont permis que les entiers compris entre -2147483648 et 2147483647. En Java les entiers sont dits de type « **int** ».
Nombres décimaux : exemple 3.45. Notez l'utilisation obligatoire du point au lieu de la virgule décimale. Pour des raisons techniques ne sont permis que les décimaux compris entre -1.79E308 et 1.79E308. En Java les décimaux sont dits de type « **double** ».
Chaînes de caractères : doivent être écrites entre guillemets "Ceci est une chaîne de caractères". En Java les chaines de caractère sont dites de type « **String** » (oui, oui **S majuscule** pour des raisons qu'on n'expliquera pas ici…).
Booléens : ne peuvent prendre que deux valeurs possibles « true » et « false » (vrai et faux). En Java les données booléennes sont dites de type « **boolean** »

3.2.2. Les variables

Les variables sont destinées à contenir des données d'un certain type. Lorsqu'on crée une variable il faut lui donner un nom et indiquer le type de données qu'elle va contenir, ce type ne pourra plus changer.

Exemples :

```
int maVariable; /* déclare qu'on va utiliser une variable de nom « maVariable »
                   qui ne pourra contenir que des nombres entiers.*/
```

De même les instructions :

```
double maVariable2;
String maVariable3;
boolean maVariable4;
```

déclarent des variables qui ne pourront contenir que des décimaux dans le premier cas, des chaînes de caractères dans le second et des valeurs booléennes dans le troisième.

On place des valeurs dans ces variables à l'aide de l'instruction d'affectation « = ».

```
maVariable = 5;
maVariable2 = 56.45;
maVariable3 = "Ma chaîne de caractères préférée";
maVariable4 = false;
```

On peut déclarer une variable et lui donner une valeur avec une seule instruction :

```
int maVariable = 67;
```

Une fois une variable créée on accède à sa valeur en écrivant son nom. Par exemple

double maNouvelleVariable = maVariable2 + 3.5;

placera 59.95 dans **maNouvelleVariable** (56.45 + 3.5) puisque **maVariable2** contient 56.45 .

Les mêmes règles s'appliquent pour les constantes mais on indique qu'il s'agit d'une constante (c'est-à-dire que la valeur ne pourra plus changer) en précédant l'instruction de déclaration par le mot clé « **final** ». Exemple :

final int MA_CONSTANTE = 100;

3.3. Opérateurs arithmétiques, relationnels et logiques

3.3.1. Les opérateurs arithmétiques

Ce sont les opérateurs : -, +, *, /. Le symbole « - » a deux rôles : soustraction et – unaire (signe).
Il faut noter que lors de l'évaluation d'une expression * et / ont la même priorité et sont prioritaires sur – (soustraction) et + qui eux aussi sont de même priorité. Le moins unaire est prioritaire sur tous les autres.
L'évaluation se fait en effectuant les opérations en parcourant l'expression de gauche à droite mais en tenant compte des priorités. Comme en mathématique, l'utilisation de parenthèses permet de modifier l'ordre d'exécution des opérations.
Il convient aussi de noter que lorsque « / » opère sur deux entiers le résultat sera un entier, c'est-à-dire que la partie décimale du résultat sera négligée. Par exemple 4/3 donnera 1 comme résultat. Dans ce cas on peut résoudre le problème en transformant l'un des arguments en décimal en le faisant simplement suivre d'un point décimal : 4./3 donnera 1.3333… comme résultat.
Dans le cas où les arguments entiers à diviser sont dans des variables on prend la valeur décimale de l'un en lui appliquant la fonction **valDec**. Donc si x contient 4 et y contient 3, x/y donnera 1 mais valDec(x)/y donnera 1.3333…

3.3.2. Les opérateurs de comparaison

Ce sont les opérateurs suivants :

Opérateur	Signification	Exemple	Résultat
= =	est égal à	2 = = 3	false
!=	n'est pas égal à	2 = = 3	true
>	est plus grand que	2 > 3	false
>=	est plus grand ou égal à	2 >= 3	false
<	est plus petit que	2 < 3	true
<=	est plus petit ou égal à	2 <= 3	true

Notez le double signe d'égalité pour tester l'égalité alors que le signe d'égalité simple est réservé pour l'affectation. L'utilisation de l'affectation (=) en place de l'égalité (= =) est une source d'erreur très commune.

3.3.3. Les opérateurs logiques

Ce sont les opérateurs suivants:

Opérateur	Signification	Exemple	Résultat

&&	ET	(3 <= 5) && (5 < 8)	true ET true qui donne true
||	OU	(5 > 3) || (3 > 5)	true OU false qui donne true
!	NON	! (5 > 3)	NON true qui donne false

3.4. Les tableaux

Quelquefois on aimerait regrouper des données de même type qui ont un lien entre elles. C'est le rôle des tableaux, ils permettent de regrouper des données de même type sous un seul nom.
On déclare un tableau en écrivant le type des données suivi de crochets vides « [] », un espace, puis le nom qu'on veut donner au tableau. Par exemple : **int[] monTableau;**
On peut ensuite créer le tableau de deux façons :

- En lui assignant immédiatement des valeurs : **monTableau = {2, 3, 4, 5, 6};**
  Cette dernière instruction va créer un tableau contenant les 5 entiers 2, 3, 4, 5 et 6.
- Sans lui assigner de valeurs, mais on doit alors dire combien de valeurs il contiendra :
  int[] monTableau2;
  monTableau2 = new int[50];
  Cette dernière instruction crée un tableau avec de la place pour 50 entiers. Notez l'utilisation du mot clé **new**, obligatoire avec cette méthode de création.

On peut déclarer et créer nos tableaux en une seule instruction :

double[] monVecteur3D = {2.5, 0.4, 10.2 }; ou
String[] mesMessages = new String[5];

On accède aux différents éléments contenus dans un tableau en utilisant le nom du tableau suivi, entre crochets, du rang de l'élément. Il faut noter que les éléments d'un tableau sont numérotés en commençant à zéro. Par exemple avec monVecteur3D défini ci-dessus, on aura :
monVecteur3D[0] qui vaut **2.5**, **monVecteur3D[1]** qui vaut **0.4** et **monVecteur3D[2]** qui vaut **10.2**.
Donc en général, si un tableau contient n valeurs, celles-ci sont numérotées de 0 à n-1. C'est une source fréquente de problèmes chez les débutants.
Les références aux différents éléments d'un tableau se comportent comme des variables normales, on peut les utiliser dans des expressions ou leur affecter des valeurs :

monVecteur3D[1] = 6.9; // monVecteur3D est devenu {2.5, 6.9, 10.2 }

double longueurAuCarre = monVecteur3D[0]*monVecteur3D[0] +
monVecteur3D[1]*monVecteur3D[1] + monVecteur3D[2]*monVecteur3D[2];
double longueur = racine(longueurAuCarre);

où **racine** désigne la fonction racine carré.

3.4.1. Les tableaux à plusieurs dimensions

Nous venons de voir des tableaux à une seule dimension. Il est possible de travailler avec des tableaux à plusieurs dimensions, en fait ce sont des tableaux de tableaux. Par exemple supposons que nous voulions regrouper dans un tableau 5 points du plan qui ont donc chacun 2 coordonnées. Il nous faudra un tableau à 5 éléments qui seront chacun des tableaux à 2 éléments. On pourra écrire :

double[][] listeDePoints = new double[5][2];

ou si on connaît déjà les coordonnées des points

```
double[][] listeDePoints = {{3.1, 0.4},{2.0, 6.3},{0, 0},{4.5, 3.2},{1.5, 3.3}};
```

3.5. Les instructions

Une instruction peut être considérée comme un ordre pour « **Expresso** ». Quand l'ordre est exécuté, une certaine action est déclenchée. Nous pouvons donc considérer un programme comme une suite d'ordres pour accomplir une tâche donnée.

```
String nom = "Expresso";
ecris(nom);
```

Le point virgule « ; » indique la fin d'une instruction. On peut écrire une instruction par ligne ou, au contraire nous pouvons en mettre plusieurs sur la même ligne. Pour une bonne pratique, il est recommandé de se contenter d'une instruction par ligne.
Un programme « **Expresso** » est exécuté du début à la fin, en partant de la première instruction jusqu'à la dernière. Cependant avec une telle exécution séquentielle, on ne peut écrire que des programmes rudimentaires. Pour nous donner plus de possibilités « **Expresso** » comporte des structures pour permettre les sauts et les répétitions dans l'ordre des instructions.

3.5.1. Instructions conditionnelles

L'instruction « if »
L'instruction conditionnelle « **if** » permet l'exécution d'ordres suivant la réalisation d'une condition.

```
if (condition)
{
   instructions;
}
```

Les instructions entre accolades « {…} » (ce qu'on appelle un *bloc d'instructions* ou en raccourci un *bloc*) suivant **if**(..){le bloc **if**} ne seront exécutées que si et seulement si la condition — une expression booléenne — mise entre parenthèses renvoie la valeur « true ». Quand le bloc « **if** » est composé de plusieurs instructions, elles sont placés entre { et }. Si nous n'avons qu'une instruction, nous pouvons ne pas écrire ces accolades, mais nous conseillons de toujours les écrire. Remarquez aussi que nous n'avons pas fait suivre l'accolade fermante d'un point virgule. En général, il n'est pas nécessaire de placer un point virgule après une accolade terminant un bloc d'instructions, mais en placer un ne nuit pas. Lorsqu'il ne s'agit pas d'un bloc, par exemple après la définition d'un tableau (il s'agit alors d'une instruction), le point virgule est nécessaire.
Exécuter différentes instructions avec **if…else**
L'instruction « **if..else** » ajoute la possibilité de spécifier des instructions alternatives, exécutées si la condition n'est pas vérifiée. Par exemple, dans le code qui suit, nous cherchons si un point (x, y) est à l'intérieur d'un carré avec l'angle supérieur gauche en (100, 200) et un côté de 50.

```
boolean intérieur ;
if (x < 100 || y > 200 || x > 150 || y< 150){
   interieur = false;
}else {
   interieur = true;
```

```
}
```

ce qu'on pourrait aussi écrire

```
boolean intérieur ;
if (x >= 100&& x <= 150 && y <= 200  && y >= 150){
   interieur = true;
}else {
   interieur = false;
}
```

Fréquemment, les instructions « **if..else** » sont emboîtées pour tester plusieurs conditions:

```
if (couleur1 == "noir"){
   couleur2 = "blanc";
}else if (couleur1 == "blanc"){
   couleur2 = "noir";
}else {
   couleur2 = "vert";
}
```

L'instruction « switch »

Quelque fois, lorsque les instructions à exécuter dépendent de la valeur entière d'une variable, plutôt que d'imbriquer des séries de if, il est plus pratique d'utiliser l'instruction « switch ». Par exemple :

```
switch  (numeroDuMois){
        case 1: case 3: case 5: case 7: case 8: case 10: case 12 :
                nombreDeJours = 31;
                break;
        case 4: case 6: case 9: case 11:
                nombreDeJours = 30;
                break;
        case 2:
                nombreDeJours = 28;
        default:
                nombreDeJours = 0;
                message = "numéro de mois inacceptable" ;
}
```

On imagine que la variable « **numeroDuMois** » peut prendre les valeurs 1, 2, 3, ..,12. Alors, en clair, l'exemple signifie :

- dans les cas où **numeroDuMois** vaut 1 ou 3 ou 5 ou 7 ou 8 ou 10 ou 12 alors le nombre de jours est égal à 31. L'instruction **break** qui suit est très importante, elle termine l'instruction **switch** et évite que les instructions suivantes soient aussi exécutées.
- dans les cas où **numeroDuMois** vaut 4 ou 6 ou 9 ou 11 alors le nombre de jours est égal à 30.
- dans le cas où **numeroDuMois** vaut 2 alors le nombre de jours est égal à 28.

- dans tous les autres cas le nombre de jour est mis à zéro, mais un message indique qu'il y a un problème.

default n'est pas obligatoire, il est là pour traiter tous les cas non prévus ou non pertinents. C'est une bonne pratique de toujours en placer un pour obtenir de l'information sur les circonstances qui font qu'il est exécuté.

3.5.2. Les répétitions

L'instruction « while »

L'instruction « **while** » fonctionne comme « **if** », mais les instructions sont exécutées **tant que** la condition est réalisée.

```
while (condition)
{
   instructions ;
}
```

Exemple :
(à propos de l'origine de cet exemple on pourra consulter http://fr.wikipedia.org/wiki/Conjecture_de_Syracuse).

```
int n = 50 ;
while (n != 1)
{
   if ( mod(n, 2) == 0){ // Si n modulo 2 est nul (i.e. si n est pair).
                  n = n/2 ;
   }else{
                  n = 3 * n + 1 ;
   }
}
```

L'instruction « for »

Nous avons souvent besoin de traiter une série de données. Pour cela, nous utilisons un compteur incrémenté à chaque répétition

```
int somme = 0;
for (int i=1; i<=100; i=i+1) {
   somme = somme + i;
}
```

Cet exemple nous donnera la somme des 100 premiers entiers soit 5050.
Nous avons utilisé l'instruction for. Voyons sa syntaxe :

```
for (initialisation; condition; mise à jour)
{
   instructions;
}
```

La « *condition* » doit être une expression booléenne, l' « *initialisation* » donne la valeur de départ du compteur et la « *mise à jour* » indique comment le compteur varie à chaque répétition.
Ceci fonctionne ainsi:

- Définir un compteur dans « *initialisation* ».
- Evaluer « *condition* ».
- Si « *condition* » renvoie true, exécuter les « *instructions* », effectuer « *mise à jour* » et revenir à l'étape précédente, évaluer « *condition* » . Si « *condition* » renvoie false, sortir de l'instruction « **for** ».

Cette instruction « **for** » est utilisée très fréquemment, elle permet en une seule instruction de définir, initialiser et mettre à jour un compteur. Les sections « *initialisation* » , « *condition* » ou « *mise à jour* » peuvent être vides mais vous devez mettre les « ; ».
L'instruction « **break** » permet de sortir de « **for** » (et aussi de **while**) en fonction d'une condition.
Par exemple:

```
int somme = 0 ;
for (int i = 0; i<100; i=i+1)
{
   somme = somme + 2 * i ;
   if (somme >= 1000){
      break;
   }
}
```

La boucle for est très utile pour placer des valeurs dans un tableau, surtout s'il est de grande dimension. Voyons des exemples :

```
int[] les100PremierCarres = new int[100] ;

for(int i=1 ; i<=100 ;i=i+1){
        les100PremierCarres[i] = i*i ;
}
```

ou pour un tableau à deux dimensions

```
double[][] listeDePoints = new double[50][2] ;

for(int i = 0 ; i <50 ; i=i+1){
        for(int j = 0; j <2 ; j = j+1){
                listeDePoints[i][j] = i * j; // par exemple
   }
}
```

3.6. Les procédures : commandes et fonctions

Quelques fois, lorsqu'on écrit un programme, on se rend compte qu'on est amené à écrire plusieurs fois les mêmes portions de code. Le rôle des procédures est justement de nous éviter de réécrire plusieurs fois ce même code.
Pour créer une procédure on :

- isole une partie de code qui remplit une tâche précise

- lui donne un nom
- exécute ce code n'importe où dans notre programme en y référent par son nom et en donnant des valeurs particulières aux variables qu'il utilise.

Nous utiliserons le nom « **procédure** » pour désigner du code nommé qui exécute des instructions et nous distinguerons deux types de procédures : les **fonctions** et les **commandes**.
Nous utiliserons le nom « **fonction** » pour désigner une procédure qui retourne une valeur et nous réserverons le nom « **commande** » pour une procédure qui exécute ses instructions sans retourner de valeur. Notons que dans certains langages on ne fait pas de distinction et que tout est nommé procédure ou fonction selon le cas.

Exemple de fonction: *Une fonction qui calcule la longueur d'un segment*
Supposons que nous ayons fréquemment, dans notre programme, à calculer la longueur de segments déterminés par leurs extrémités (x1, y1) et(x2, y2). Nous décidons de créer le morceau de code qui le fera.

```
public double longueurSegment(double x1, double y1, double x2, double y2) {
    double dx = x2 - x1 ;
    double dy = y2 - y1 ;
    double longueurAuCarre = dx*dx + dy*dy ;
    double longueur = racine(longueurAuCarre);
    return longueur;
}
```

Voyons ce code de près.

1. **public** : bien que ce mot au début de la définition d'une procédure ou d'une fonction ne soit pas toujours nécessaire nous conseillons de toujours l'inclure, dans le cadre des programmes **Expresso**. Expliquer ici dans quelles circonstances il est nécessaire ou non nous entrainerait trop loin.
2. Après **public** nous indiquons d'abord le type de la valeur retournée : **double** ;
3. Nous donnons ensuite le nom de la fonction — ici **longueurSegment** (rappelons que les noms de procédures obéissent aux mêmes règles que les noms de variables) ;
4. Nous accompagnons le nom de notre fonction de la donnée des arguments de la fonction, noms de variables précédés de leur type et séparés par des virgules. Ces variables seront utilisables par la fonction ;
5. Le corps de la fonction est entre **{** et **}**. Nous y définissons quelques variables (**dx**, **dy**, **longueurAuCarre** et **longueur**) et les initialisons avec des nombres calculés à l'aide des arguments de la fonction ;
6. Pour avoir la longueur elle-même, nous faisons appel à une fonction prédéfinie dans « **Expresso** », la fonction « **racine** », qui calcule la racine carrée d'un nombre positif. Et nous terminons en indiquant la valeur que renvoie notre fonction avec l'instruction « return » ;

```
double longueur = racine(longueurAuCarre);
return longueur;
```

Dans le cas d'une **commande** les choses sont presque identiques. D'une part on indique que la commande ne retourne pas de valeur avec le mot clé « **void** » et d'autre part on ne trouvera pas, dans

le corps de la commande, d'instruction « **return *quelqueChose*** ». Notons qu'on peut cependant rencontrer « **return ;** » tout seul, ce qui est une façon de terminer la commande un peu comme **break** permettait de quitter une boucle **for**.

Exemple :
Voici deux commandes qui font exactement la même chose :

```
void message1(String unTexte){
   if(unTexte = = ""){    // Si unTexte est une chaine vide (i.e. sans aucun caractère)
                return ;          // Terminer sans rien faire
   }
                ecris(unTexte) ;
}

void message2(String unTexte){
        if(unTexte != ""){
                ecris(unTexte) ;
        }
}
```

3.7. Utiliser des procédures : fonctions et commandes.

Après avoir défini une procédure, nous pouvons l'appeler, c'est à dire exécuter son code.
Voyons exactement comment appeler une fonction ou une commande en utilisant celles définie dans la section précédente :

```
double xDebut = 100.5 ;
double yDebut = 50 ;
double xFin = 200.3 ;
double yFin = 100.5 ;
// Utilisation de la fonction « longueurSegment »
double longueur = longueurSegment(xDebut, yDebut, xFin, yFin) ;
String testLongueur ;
if (longueur > 500){
   testLongueur = "Segment trop long" ;
}else{ if (longueur < 100){
        testLongueur = "Segment trop court" ;
}else{
        testLongueur = "Segment idéal" ;
}
// Utilisation de la commande « message1 »
message1(testLongueur) ;
```

Nous constatons que l'appel se fait avec le nom suivi de la valeur des arguments entre parenthèses.

Nous pouvons utiliser les procédures n'importe où dans notre programme et dans le cas des fonctions utiliser les valeurs renvoyées dans n'importe quelle expression.
Voici des choses importantes à retenir à propos des procédures :

- Elles peuvent avoir plusieurs définitions pourvu que les différentes définitions diffèrent par le nombre ou le type des arguments ;
- Elles peuvent être appelées n'importe où dans le code ;
- Elles peuvent s'appeler elles-mêmes (récursivité) ou en appeler d'autres ;
- Elles peuvent avoir n'importe quel nombre d'arguments et même aucun ;
- Lorsqu'elles n'ont pas d'argument, le nom est suivi de parenthèses vides « ( ) », dans la définition et dans les appels ;
- Lors d'un appel le nombre et le type des arguments doit correspondre à une définition ;
- Les variables définies dans le corps d'une procédure ne peuvent pas être utilisées en dehors de la fonction ou de la procédure ;

3.8. Remarques importantes à propos des variables

- Une variable déclarée dans un bloc (voir 3.5.1) n'a pas d'existence à l'extérieur du bloc et donc ne peut être utilisée qu'à l'intérieur de ce même bloc. On dit qu'une telle variable est locale au bloc où elle est déclarée.
- Une variable déclarée dans une procédure n'a pas d'existence à l'extérieur de la procédure et donc ne peut être utilisée qu'à l'intérieur de la procédure. On dit qu'une telle variable est locale à la procédure où elle est déclarée.
- Les arguments d'une procédure peuvent être considérés comme des variables locales à cette même procédure
- En général lorsqu'on passe une variable en argument à une procédure cette variable ne peut pas être modifiée par la procédure. Par exemple :
  définissons la procédure suivante :

```
void test(double x){
    x = 50;
}
```

et exécutons les instructions suivantes :

```
double y = 0;
test(y);
écris(y); // Va écrire 0 et non 50
```

Cependant les tableau font exception à cette règle. Par exemple :
définissons la procédure suivante :

```
void testTableau(double[] unTableau){
    unTableau[3] = 0;
}
```

et exécutons les instructions suivantes :

```
double[] monTableau = {5, 5, 5, 5, 5};
écris(monTableau[3]); // Va écrire 5
```

testTableau(monTableau);
écris(monTableau[3]); // Va maintenant écrire 0 et non 5

Le tableau **monTableau** à donc été modifié par la procédure.

En utilisant des tableaux il est important de se souvenir de cette particularité pour ne pas avoir de mauvaises surprises.

4. Le langage Expresso

Le langage « Expresso » est construit au dessus de Java dans le but de fournir un langage simplifié. Il permet tout d'abord de construire facilement une interface graphique comprenant une zone de dessin, une zone de texte, des menus, des boutons et des glissières. Cette interface est configurable et seule la zone de dessin est toujours présente, la présence et le comportement des autres composants sont laissés au gré de l'utilisateur à travers des instructions d'utilisation élémentaires.
Le langage lui même comprend, en particulier, des instructions de dessin et de gestion de liste inspirées de la géométrie de la tortue du langage Logo. Nous verrons également qu'il fournit des fonctions de création de couleurs mais aussi des couleurs prédéfinies dont voici les noms :
noir, **bleu**, **cyan**, **grisFonce**, **gris**, **vert**, **grisClair**, **magenta**, **orange**, **rose**, **rouge**, **blanc**, **jaune**.
Noter que ces noms sont des variables de type **Color** et doivent être utilisés sans guillemets.

Passons maintenant en revue les différentes instructions en précisant leurs paramètres, leur syntaxe et leur effet.

4.1. Le langage de la tortue

On peut dessiner à l'écran en imaginant qu'on commande à un animal électronique, la tortue, qui est caractérisé par:

- sa position donnée par des coordonnées x et y,
- sa direction (ou son cap) donnée par un angle avec l'axe vertical et mesuré positivement dans le sens horaire.

La tortue transporte avec elle un crayon qui peut être levé ou baissé et qui, s'il est baissé, laissera une trace de couleur sur le parcours de la tortue.
Voici donc les commandes qui permettent de commander la tortue et par le fait même de dessiner dans la zone de dessin.

Déplacements .

avance(d) *abréviation* **av(d)**

Le paramètre d est un nombre entier ou décimal et cette instruction fait avancer la tortue de d pixels dans sa direction courante. Si le crayon est baissé un segment sera tracé entre les positions initiale et finale de la tortue.

recule(d) *abréviation* **re(d)**

Le paramètre d est un nombre entier ou décimal et cette instruction fait reculer la tortue de d pixels par rapport à sa direction courante. Si le crayon est baissé un segment sera tracé entre les positions initiale et finale de la tortue.

Position

fixePos(x, y) *abréviation* **fPos(x, y)**

Les paramètres x et y sont des nombres entiers ou décimaux. L'instruction modifie la position de la tortue (mais pas son **cap**) pour qu'elle prenne les coordonnées x et y. La tortue trace un segment entre les positions initiale et finale si le crayon est baissé.

posX()

Retourne un nombre décimal représentant l'abscisse de la position de la tortue.

posY()

Retourne un nombre décimal représentant l'ordonnée de la position de la tortue.

Orientation

droite(angle) *abréviation* **dr(angle)**

Le paramètre angle est un nombre entier ou décimal et cette instruction change la direction de la tortue en lui ajoutant la valeur angle. Donc si angle est positif la tortue tourne vers sa droite.

cap()

Retourne un nombre décimal (type double) qui représente la direction courante de la tortue, c'est à dire l'angle qu'elle fait avec la verticale mesuré positivement dans le sens horaire.

fixe Cap(d) *abréviation* **fCap(d)**

Le paramètre d est un nombre entier ou décimal et cette instruction modifie la direction de la tortue pour qu'elle fasse un angle de d avec la verticale.

gauche(angle) *abréviation* **ga(angle)**

Le paramètre angle est un nombre entier ou décimal et cette instruction change la direction de la tortue en lui retranchant la valeur angle. Donc si angle est positif la tortue tourne vers sa gauche.

vers(x, y)

Les paramètres x et y sont des nombres entiers ou décimaux et cette instruction retourne un nombre décimal représentant l'angle que doit faire la tortue avec la verticale pour pointer en direction du point de coordonnées (x, y).

Le crayon

baisseCrayon() *abréviation* **bc()**

Met la tortue dans un état ou ses déplacements laisseront une trace.

couleurCrayon(c) *abréviation* **cc(c)**

couleurCrayon(r, v, b) *abréviation* **cc(r, g, b)**

Cette instruction a deux syntaxes, elle accepte pour paramètres une couleur c prédéfinie (noir, blanc, jaune, vert, rouge, bleu, etc..) ou trois entiers compris entre 0 et 255 définissant une couleur RGB. Les paramètres r, v, b donnant les quantités de rouge, vert et bleu respectivement. Comme son nom l'indique, son effet est de changer la couleur avec laquelle la tortue trace.

leveCrayon() *abréviation* **lc()**

Met la tortue dans un état ou ses déplacements ne laisseront aucune trace.

retourneCouleurCrayon()

Retourne une couleur (type Color) représentant la couleur courante du crayon.

retourneCouleurCrayonR() *abréviation* **retCCr()**

Retourne un entier représentant la quantité de rouge dans la couleur courante du crayon.

retourneCouleurCrayonV() *abréviation* **retCCv()**

Retourne un entier représentant la quantité de vert dans la couleur courante du crayon.

retourneCouleurCrayonB() *abréviation* **retCCb()**

Retourne un entier représentant la quantité de bleu dans la couleur courante du crayon.

retourneTailleCrayon() *abréviation* **retTC()**

Retourne un nombre décimal (type double) représentant la taille actuelle du crayon.

tailleCrayon(ep) *abréviation* **tc(ep)**

Le crayon est de forme carré, cette instruction change la taille de ce carré pour pour que la longueur de son côté devienne ep, où ep est un nombre entier ou décimal.

traceInverse()

Met le crayon dans un mode où lorsqu'il trace deux fois un pixel avec la même couleur, ce pixel reprend la couleur qu'il avait avant les deux tracés. Autrement dit le second tracé annule l'effet du premier. (Noter que ce mode affecte tous les graphiques et pas seulement la tortue.)

traceNormal()

Annule le mode de tracé inverse.

4.2. La zone graphique

centreTortue()

Cette instruction place l'origine des coordonnées de la tortue au centre de la zone graphique.

ecrisGraphique(x, y, texte)

Les paramètres x et y sont des nombres entiers ou décimaux et le paramètre texte est une chaîne de caractères. Cette instruction écris le contenu du paramètre « texte » dans la zone graphique à partir du point (x, y) et en utilisant la couleur du crayon. Noter que le point (x, y) sera le bas, gauche du début du texte.

effaceGraphique()

Efface le contenu de la fenêtre graphique sans aucun effet sur l'état de la tortue..

fixeOrigineTortue(x, y)

Les paramètres x et y sont des nombres entiers ou décimaux qui représentent un point de la zone graphique en coordonnées de cette zone (c'est à dire origine au coin en haut à gauche, axe horizontal pointant à droite et axe vertical pointant vers le bas). Cette instruction place l'origine des coordonnées de la tortue au point (x, y) de la zone graphique.

hauteurZoneGraphique()

Retourne un nombre entier qui représente la hauteur actuelle de la zone graphique (en pixels).

hauteurTexteGraphique() *abréviation:* **hTG()**

Retourne un nombre entier qui représente la hauteur, en pixel, de la police de caractères utilisée pour écrire dans la zone graphique.

largeurZoneGraphique()

Retourne un nombre entier qui représente la largeur actuelle de la zone graphique (en pixels).

longueurTexteGraphique(ch) *abréviation:* **lTG(ch)**

Retourne un nombre entier donnant la longueur, en pixel, de la chaîne de caractères passée en argument. Cette longueur est évaluée avec la police de caractère utilisée pour écrire dans la zone graphique et selon la taille actuelle de cette police..

miseAJourGraphique()

Force un rafraîchissement de la zone graphique.

origineTortueX()

Retourne un nombre décimal (type double) qui représente l'abcisse de l'origine des coordonnées de la tortue considérée comme un point de la zone graphique.

origineTortueY()

Retourne un nombre décimal (type double) qui représente l'ordonnée de l'origine des coordonnées de la tortue considérée comme un point de la zone graphique.

tailleTexteGraphique(n)

Fixe la taille de la police de caractère utilisée pour écrire dans la zone graphique à **n** points.

videGraphique()

Efface le contenu de la zone graphique et ré-initialise la tortue.

Pour tracer des formes pleines avec la tortue

couleurRemplissage(c)

couleurRemplissage(r, v, b)

Cette instruction a deux syntaxes, elle accepte pour paramètres une couleur **c** prédéfinie (voir la liste dans l'introduction à la section 4) ou trois entiers compris entre 0 et 255 définissant une couleur RGB. Les paramètres **r**, **v**, **b** donnant les quantités de rouge, vert et bleu respectivement. Comme son nom l'indique, son effet est de changer la couleur avec laquelle les figures seront remplies.

debutRemplir()

finRemplir()

Toutes les instructions de déplacement de la tortue rencontrées entre debutRemplir() et finRemplir() contribuent à créer un polygone qui sera rempli avec la couleur de remplissage courante.

retourneCouleurRemplissage()

Retourne la couleur (type Color) actuelle de remplissage.

retourneCouleurRemplissageR() *abréviation* **retCRr()**

Retourne un entier représentant la quantité de rouge dans la couleur de remplissage actuelle.

retourneCouleurRemplissageV() *abréviation* **retCRv()**

Retourne un entier représentant la quantité de vert dans la couleur de remplissage actuelle.

retourneCouleurRemplissageB() *abréviation* **retCRb()**

Retourne un entier représentant la quantité de bleu dans la couleur de remplissage actuelle.

Dessiner sans la tortue

Toutes les instructions ci-dessous tracent avec la couleur du crayon de la tortue, à l'exception de celles qui tracent des formes pleines qui utilisent la couleur du crayon pour la frontière et celle de remplissage pour l'intérieur. De plus tous les paramètres représentant des coordonnées sont tous relatifs aux coordonnées de la tortue.

arc(cx, cy, r, angleDeb, angleFin)

Les paramètres sont tous des nombres entiers ou décimaux. Cette instruction trace un arc de cercle centré en (cx, cy) de rayon r débutant à l'angle angleDeb et finissant à l'angle angleFin. Le centre et les angles sont tous relatifs aux coordonnées de la tortue.

cercle(cx, cy, r)

Les paramètres sont tous les trois des nombres entiers ou décimaux. Cette instruction trace un cercle de rayon r centré en (cx, cy).

disque(cx, cy, r)

Les paramètres sont tous les trois des nombres entiers ou décimaux. Cette instruction trace un disque de rayon r centré en (cx, cy) en utilisant la couleur de remplissage.

ellipse(x1, y1, x2, y2)

Les paramètres sont tous les quatre des nombres entiers ou décimaux. Cette instruction trace une ellipse inscrite dans le rectangle dont une diagonale va du point (x1, y1) au point (x2, y2).

rectangle(x1, y1, x2, y2)

Les paramètres sont tous les quatre des nombres entiers ou décimaux. Cette instruction trace un rectangle dont une diagonale va du point (x1, y1) au point (x2, y2).

rectanglePlein(x1, y1, x2, y2)

Les paramètres sont tous des nombres entiers ou décimaux. Cette instruction trace un rectangle plein dont une diagonale va du point (x1, y1) au point (x2, y2) et en utilisant la couleur de remplissage.

segment(x1, y1, x2, y2)

Les paramètres sont tous les quatre des nombres entiers ou décimaux. Cette instruction trace un segment allant du point (x1, y1) au point (x2, y2). Il faut noter que lorsque cette instruction se retrouve entre debutRemplir() et finRemplir() elle contribue à définir les polygones à remplir.

4.3. La zone de texte

ecris(texte)

Le paramètre texte peut être une chaîne de caractères, un entier, un décimal, un booléen ou une liste. Cette instruction écrit le contenu du paramètre texte dans la zone de texte.

ecrisRC(texte)

Le paramètre texte peut être une chaîne de caractères, un entier, un décimal, un booléen ou une liste. Cette instruction écrit le contenu du paramètre texte dans la zone de texte et le fait suivre d'un retour à la ligne.

fixeHauteurZoneTexte(n)

Règle la hauteur de la zone de texte à n pixels ou à la totalité de l'espace disponible dans la fenêtre s'il est inférieur à n pixels. La zone de texte reste modifiable à la souris.

fixeProportionZoneTexte(d)

Le paramètre d est un nombre entier ou décimal entre 0 et 1 (0 <= d <= 1). Cette instruction fixe la proportion de la fenêtre occupée par la zone de texte à (d * 100)%.

geleHauteurZoneTexte()

Après invocation de cette procédure, les modifications à la taille de la fenêtre ne changent pas la hauteur (en pixels) de la zone de texte. Celle-ci reste cependant modifiable à la souris.

tailleTexte(t)

Le paramètre t est un nombre entier et cette instruction fixe la taille du texte de la zone texte à t points.

videTexte()

Efface tout le texte présent dans la zone de texte.

4.4. Le langage des Listes

Création

listeVide()

Retourne une liste vide.

vecteurVersListe(tableau)

Le paramètre *tableau* est un tableau d'entiers, de décimaux, de booléens ou de chaînes de caractères. Cette instruction retourne une liste contenant les éléments du tableau.

Ajout d'éléments

insereItem(elem, index, liste)

Le paramètre *elem* peut être un entier, un décimal, une chaîne de caractères, un booléen ou une liste, le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* avec l'élément *elem* ajouté a la position *index*. *(version destructive: itemPut(elem, index, liste))*

metsPremier(elem, liste) *abréviation* **mp(elem, liste)**

Le paramètre *elem* peut être un entier, un décimal, une chaîne de caractères, un booléen ou une liste, le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* avec l'élément *elem* ajouté au début. *(version destructive: fPut(elem, liste))*

metsDernier(elem, liste) *abréviation* **md(elem, liste)**

Le paramètre *elem* peut être un entier, un décimal, une chaîne de caractères, un booléen ou une liste, le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* avec l'élément *elem* ajouté a la fin. *(version destructive: lPut(elem, liste))*

Remplacement d'éléments

remplaceItem(elem, index, liste)

Le paramètre *elem* peut être un entier, un décimal, une chaîne de caractères, un booléen ou une liste, le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* avec l'élément *elem* a la position *index* à la place de l'élément qui s'y trouvait précédemment. *(version destructive: replaceItem(elem, index, liste))*

Lecture d'éléments

Noter que toutes les instructions de lecture qui suivent retournent une erreur si on les applique à une liste vide.

dernier(liste) *abréviation* **de(liste)**

Le paramètre *liste* doit être une liste. Cette instruction retourne le dernier élément de la liste *liste*. La liste *liste* n'est pas modifiée.

item(index, liste) *abréviation* **pr(liste)**

Le paramètre *liste* doit être une liste. Cette instruction retourne l'élément de la liste *liste* qui se trouve à la position *index*. La liste *liste* n'est pas modifiée.

premier(liste) *abréviation* **pr(liste)**

Le paramètre *liste* doit être une liste. Cette instruction retourne le premier élément de la liste *liste*. La liste *liste* n'est pas modifiée.

Suppression d'éléments

saufDernier(liste) *abréviation* **sd(liste)**

Le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* sans son dernier élément. *(version destructive: butFirst(liste))*

saufPremier(liste) *abréviation* **sp(liste)**

Le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* sans son premier élément. *(version destructive: butLast(liste))*

supprimerItem(index, liste)

Le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste* sans l'élément qui se trouvait à la position index. Retourne une erreur si la liste n'a pas d'élément à la position **index**.*(version destructive: butItem(index, liste))*

Les prédicats

booleenP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est un booléen et *false* autrement. Si *elem* est un booléen on récupère sa valeur avec l'instruction valBool(elem).

chaineP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est une chaîne de caractères et *false* autrement. Si *elem* est une chaîne de caractères on récupère sa valeur avec l'instruction valChaine(elem).

decimalP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est un décimal et *false* autrement. Si *elem* est un décimal on récupère sa valeur avec l'instruction valDec(elem).

elementP(elem, liste)

Le paramètre *elem* peut être un entier, un décimal, une chaîne de caractères ou une liste. Le paramètre *liste* doit être une liste. Cette instruction retourne un booléen : *true* si *elem* est un élément de la liste *liste* et *false* autrement.

entierP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est un entier et *false* autrement. Si *elem* est un entier on récupère sa valeur avec l'instruction valEnt(elem).

listeP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est une liste et *false* autrement.

nombreP(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un booléen : *true* si *elem* est un nombre et *false* autrement. Si *elem* est un nombre on peut récupérer sa valeur avec l'instruction valDec(elem).

videP(elem)

Le paramètre *elem* doit être une liste. Cette instruction retourne un booléen : *true* si *elem* est une liste vide et *false* autrement.

Divers

clone(liste)

Le paramètre *liste* doit être une liste. Cette instruction retourne une copie de la liste *liste*.

compte(liste)

Le paramètre *liste* doit être une liste. Cette instruction retourne un le nombre d'élément dans *liste*.

phrase(liste1, liste2)

Les paramètres *liste1 et liste2* doivent être des listes. Cette instruction retourne une copie de la concaténation des deux listes. *(version destructive sentence(liste1, liste2))*

type(elem)

Le paramètre *elem* doit être de type Liste. Cette instruction retourne un entier représentant le type de *elem* (1 pour entier, 2 pour décimal, 3 pour chaîne, 4 pour booléen, 5 pour liste).

valBool(elem)

Le paramètre *elem* doit être de type Liste et représenter un booléen. Cette instruction retourne le booléen représenté par *elem*.

valChaine(elem)

Le paramètre *elem* doit être de type Liste et représenter une chaîne. Cette instruction retourne la chaîne représentée par *elem*.

valDec(elem)

Le paramètre *elem* doit être de type Liste et représenter un décimal. Cette instruction retourne la valeur décimale de *elem*.

valEnt(elem)

Le paramètre *elem* doit être de type Liste et représenter un entier. Cette instruction retourne la valeur entière de *elem*.

4.5. Divers

annulation()

Procédure qui retourne « **true** » si la dernière boîte de dialogue utilisée a été fermée avec un bouton « Annuler » ou avec la case de fermeture. Elle retourne « **false** » dans tous les autres cas.

appletP()

Cette instruction retourne *« true »* si l'exécution à lieu dans un applet et *« false »* autrement.

executionLocale()

Cette instruction est la négation de la précédente elle retourne *« false »* si l'exécution à lieu dans un applet et *« true »* autrement.

attendre(n)

n est un entier et cette instruction laisse passer n millisecondes.

bip()

Instruction qui produit un son « bip ».

chaine(arg)

Retourne son argument sous forme de chaîne de caractères. L'argument peut être de type **int**, **double** ou **Liste**.

chaine(arg, n)

Retourne son argument, arrondi à la **n**^{e} décimale, sous forme de chaîne de caractères avec exactement n décimales (donc avec des « 0 » à la fin si nécessaire). L'argument peut être de type **int**, **double** ou **Liste**. Dans le cas d'un type **Liste** qui ne représente pas un nombre, la forme « chaîne » est retournée en ignorant **n**.

couleurRVB(r, v, b)

Les paramètres r, v, b sont des entiers entre 0 et 255. Cette instruction retourne une couleur (type Color) dont r, v, b représentent les quantités de rouge, vert et bleu.

valDec(n)

Le paramètre *n* doit être de type int (entier). Cette instruction retourne un double (décimal) de même valeur que n.

egalesP(ch1, ch2)
Retourne vrai si les chaînes ch1 et ch2 sont égales et faux autrement.

hasard()

Cette instruction retourne un nombre décimal d (type double) aléatoire entre 0 et 1 (0 <= d <1).

hasard(min, max)

Les paramètres min et max sont des entiers (min <= max). Cette instruction retourne un entier n aléatoire entre min et max(min <= n <= max).

interruption()

Retourne « true » si une interruption a été demandée (via un bouton d'interruption) et « false » autrement.

interruptionPermise()

Retourne « true » si le programme est configuré de façon que les interruptions soient permises.

permettreInterruption(b)

Le paramètre **b** doit être un booléen. Cette instruction configure l'environnement pour que le bouton d'interruption soit actif (*b = true*) ou inactif (*b=false*) s'il existe.

quitter()

Ferme l'application.

unicode(ch)

Le paramètre ch est une chaîne de caractères dans laquelle les caractères accentués ont été représentés par un code particulier. Cette fonction retourne une chaîne dans laquelle les caractères accentués ont été remplacés par leur représentation unicode.

Le code:

suivi de la lettre sans accent, suivi des 3 premiers caractères du nom de l'accent ou du symbole (gra pour grave, aig pour aigu, cir pour circonflexe, ced pour cédille, tre pour trema).

Par exemple: à ---> #agra (# a gra).

valEnt(d)

Le paramètre d est un nombre décimal (type double). Cette instruction retourne l'entier le plus proche de d.

remplacer(ch1, ch2, ch3)

Les paramètres **ch1**, **ch2**, **ch3** sont trois chaînes de caractères et cette fonction retourne la chaîne **ch1** dans laquelle la sous chaîne **ch2** a été remplacée par la chaîne **ch3**.

Une utilisation possible de cette fonction c'est de transformer la virgule décimale d'un nombre en point pour une utilisation informatique. Par exemple **remplacer("1234,56" , "," , ".")** va retourner **1234.56 .**

4.6. Les boutons

activerBouton(int i)

Rend actif le bouton numéro *i*.

changerNomBouton(i, nom)

i est un entier et nom une chaîne de caractères. L'effet est de changer le nom du bouton numéro i pour qu'il prenne la valeur du paramètre nom.*(Voir la façon de numéroter les boutons chapitre ...)*

desactiverBouton(int i)

Désactive le bouton numéro *i*.

4.7. Les champs

On peut ajouter jusqu'à 2 lignes de champs de textes au dessous des lignes de boutons. Ces champs présentent une légende suivie du champ proprement dit. Ces champs sont numérotés à partir de 1 dans l'ordre ou ils apparaissent de gauche à droite en commençant par la première ligne et en poursuivant par la deuxième. On lit et on écrit dans ces champs à l'aide de deux fonctions :

valeurChamp(int i)

Retourne la chaîne de caractères contenue dans le champ numéro *i.* S'il n'y a pas de champ numéro *i* la valeur retournée est la chaîne vide.

fixeValeurChamp(int i, String val)

Écrit la chaîne *val* dans le champ numéro *i*. S'il n'y a pas de champ numéro *i* la fonction ne fait rien.

4.8. Les glissières

On peut ajouter jusqu'à 12 glissières sur deux lignes à raison d'un maximum de six par ligne. Ces glissières présentent une légende, un champ de texte indiquant la valeur courante de la glissière et, au dessous, la glissière proprement dite avec ses valeur minimale et maximale affichées.

ajouterGlissiereLigne1(légende, min, max, posDebut, nbDec)

ajouterGlissiereLigne1(légende, min, max)

Le paramètre « légende » est une chaîne de caractères qui représente le texte affiché au dessus de la glissière. Les paramètres min, max et posDebut sont des nombres décimaux représentant respectivement les valeurs maximale, minimale et initiale de la glissière. Les paramètre nbDec est un

entier indiquant le nombre de décimales désirées. Cette instruction ajoute sur la ligne 1 une glissière configurée a l'aide des paramètres fournis. Lorsque les paramètres posDebut et nbDec ne sont pas présents ils prennent les valeurs par défaut posDebut = min et nbDec = 0.

ajouterGlissiereLigne2(légende, min, max, posDebut, nbDec)

ajouterGlissiereLigne2(légende, min, max)

Identique à ci-dessus mais la glissière est ajoutée sur la deuxième ligne.

fixeValeurGlissiere(*i*, *v*)

i est un entier et *v* un décimal. L'instruction donne à la glissière numéro *i* la valeur *v.*

valeurGlissiere(i)

Le paramètre i est un entier. L'instruction retourne un nombre décimal qui est la valeur actuelle de la glissière numéro i.

4.9. Dialogues

choixMultiple(titre, message, bouton1)

choixMultiple(titre, message, bouton1, bouton2)

choixMultiple(titre, message, bouton1, bouton2, bouton3)

choixMultiple(titre, message, bouton1, bouton2, bouton3, bouton4)

Cette instruction ouvre une fenêtre de dialogue avec un titre, un message et de 1 à 4 boutons. Elle retourne le numéro du bouton cliqué.

Les paramètres titre et message sont des chaînes de caractères donnant le titre de la fenêtre et le message à afficher. Les paramètres bouton1, bouton2, bouton3, bouton4 sont des chaînes de caractères donnant les noms des boutons à afficher.

Lorsqu'une telle fenêtre est fermée via la case de fermeture d'une part elle retourne 0, et d'autre part la procédure annulation() va retourner « true » si elle est consultée avant l'utilisation d'une autre boîte de dialogue.

demanderChaine(titre, message)

demanderChaine(titre, message, valInit)

Cette instruction ouvre une fenêtre de dialogue avec un titre, un message, un champ de texte et un bouton « OK ». Si le paramètre valInit est présent, ce doit être une chaîne et il apparaît dans le champ de texte lors de l'ouverture de la fenêtre.

Lorsqu'on clique le bouton « OK » l'instruction retourne la chaîne écrite dans le champ de texte. Lorsqu'une telle fenêtre est fermée via la case de fermeture ou le bouton « Annuler », d'une part elle retourne la chaîne vide, et d'autre part la procédure **annulation()** va retourner « true » si elle est consultée avant l'utilisation d'une autre boîte de dialogue.

demanderEntier(titre, message)

demanderEntier(titre, message, valInit)

Cette instruction ouvre une fenêtre de dialogue avec un titre, un message, un champ de texte et un bouton « OK ». Si le paramètre valInit est présent, ce doit être un entier et il apparaît dans le champ de texte lors de l'ouverture de la fenêtre.

Lorsqu'on clique le bouton « OK » l'instruction retourne l'entier écrit dans le champ de texte. Lorsqu'une telle fenêtre est fermée via la case de fermeture ou le bouton « Annuler », d'une part elle retourne le plus petit entier disponible dans le langage, et d'autre part la procédure **annulation()** va retourner « true » si elle est consultée avant l'utilisation d'une autre boîte de dialogue.

demanderNombre(titre, message)

demanderNombre(titre, message, valInit)

Cette instruction ouvre une fenêtre de dialogue avec un titre, un message, un champ de texte et un bouton « OK ». Si le paramètre valInit est présent, ce doit être un entier ou un décimal et il apparaît dans le champ de texte lors de l'ouverture de la fenêtre.

Lorsqu'on clique le bouton « OK » l'instruction retourne le décimal écrit dans le champ de texte. Lorsqu'une telle fenêtre est fermée via la case de fermeture ou le bouton « Annuler », d'une part elle retourne le plus petit décimal disponible dans le langage, et d'autre part la procédure **annulation()** va retourner « true » si elle est consultée avant l'utilisation d'une autre boîte de dialogue.

message(mess)

Cette instruction ouvre une fenêtre de dialogue avec un titre, un message et un bouton « OK ».

Lorsqu'on clique le bouton « OK » la fenêtre disparaît sans rien retourner.

4.10. Fichiers

ajouterDonnee(donnee)

Le paramètre donnee peut être un nombre entier ou décimal, un booléen ou une chaîne de caractères. Cette instruction ajoute la valeur du paramètre donnee à l'ensemble des données destinées à être sauvegardées dans un fichier.

existeFichier(ch)

Le paramètre **ch** est une chaîne de caractères représentant un nom de fichier. Cette instruction retourne « **true** » si le fichier correspondant existe et « **false** » autrement.

lireDonnee(i)

i est un entier qui désigne le numéro de ligne de la donnée à lire, parmi les données qui ont été lues dans un fichier. Cette méthode retourne, sous forme de chaîne de caractères, la donnée qui était à la ligne i dans le fichier.

lireDonneeSuivante()
Cette méthode retourne, sous forme de chaîne de caractères, la donnée qui suit la dernière donnée lue. Si une telle donnée n'existe pas il y aura une erreur.

lireFichier()

Cette méthode permet de donner le nom, ou de sélectionner, un fichier à ramener en mémoire. Une fois le fichier ramené en mémoire on accède aux données ligne par ligne avec la méthode lireDonnee(*numéro de ligne*).

rembobiner()
Replace la prochaine donnée à lire au début du fichier lu.

sauverDonnees()

Cette méthode permet de donner un nom de fichier et de sélectionner un emplacement où sera sauvegardé un fichier texte portant le nom choisi. Dans ce fichier seront sauvegardées les données mise en réserve à l'aide de la méthode ajouterDonnee(*donnee*).

tailleFichier()

retourne le nombre de données présentes dans le fichier lu.

valEnt(donnee)

Le paramètre donnee est une chaîne de caractère. Si donnee est une chaîne représentant un entier, cette méthode transforme la chaîne en entier et retourne cet entier.

valDec(donnee)

Le paramètre donnee est une chaîne de caractère. Si donnee est une chaîne représentant un nombre décimal, cette méthode transforme la chaîne en décimal et retourne ce décimal.

valBool(donnee)

Le paramètre donnee est une chaîne de caractère. Si donnee est une chaîne représentant un booléen (*true* ou *false*), cette méthode transforme la chaîne en booléen et retourne ce booléen.

viderDonneesASauver()

Cette méthode réinitialise à vide l'ensemble des données à sauvegarder.

viderDonneesRamenees()

Cette méthode réinitialise à vide l'ensemble des données lues à partir d'un fichier.

4.11. Les mathématiques

Les constantes

E la constante d'Euler, base du logarithme népérien.

Pi le rapport de la circonférence du cercle à son diamètre

Les fonctions

abs(x)

Le paramètre x peut être un entier ou un décimal. Cette fonction retourne la valeur absolue de **x**. Le type de la valeur de retour est le même que celui de x.

acos(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'arc (*en radians*) dont le cosinus est **x.**

acosD(x)

Le paramètre x peut être un entier ou un décimal. Cette fonction retourne l'arc (*en degrés*) dont le cosinus est **x.**

arrondis(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'entier le plus proche de **x.**

arrondis(x, n)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le décimal, à n décimales, le plus proche de **x.**

asin(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'arc (*en radians*) dont le sinus est **x.**

asinD(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'arc (*en degrés*) dont le sinus est **x.**

atan(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'arc (*en radians entre -Pi et Pi*) dont la tangente est **x.**

atanD(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne l'arc (*en degrés entre -90 et 90*) dont la tangente est **x.**

atan2(y, x)

Les paramètres **x** et **y** peuvent être entiers ou décimaux. Cette fonction retourne l'arc (*en radians entre 0 et 2Pi*) dont la tangente est y/x **.**

atan2D(y, x)

Les paramètres **x** et y peuvent être entiers ou un décimaux. Cette fonction retourne l'arc (*en degrés entre 0 et 360*) dont la tangente est y/x **.**

cos(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le cosinus de x radians**.**

cosD(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le cosinus de x degrés**.**

divDec(a, b)

Quel que soit le type des nombres **a** et **b**, cette fonction retourne un **décimal** (double) résultat de la division de a par b.

divEnt(a, b)

a et b sont des entiers et l'instruction retourne un entier qui est le quotient entier de a par b.

exp(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne $\mathbf{E^x}$.

log(x)

Le paramètre **x** peut être un entier ou un décimal positif. Cette fonction retourne le logarithme de x.

max(x, y)

Les paramètres **x** et **y** peuvent être entiers ou un décimaux. Cette fonction retourne le plus grand parmi **x** et **y**. Si **x** et **y** sont entiers la valeur retournée est de type **int** sinon elle est de type **double**.

min(x, y)

Les paramètres **x** et **y** peuvent être entiers ou un décimaux. Cette fonction retourne le plus petit parmi **x** et **y**. Si **x** et **y** sont entiers la valeur retournée est de type **int** sinon elle est de type **double**.

mod(a, b)

a et b sont des entiers et l'instruction retourne un entier qui est le reste de la division entière de a par b (a modulo b)

plafond(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le plus petit entier plus grand ou egal à **x**.

plafond(x, n)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le plus petit décimal, à n décimales, plus grand ou égal à **x**.

plancher(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le plus grand entier plus petit ou egal à **x**.

plancher(x, n)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le plus grand décimal, à n décimales, plus petit ou égal à **x**.

pow(x, y)

Les paramètres **x** et **y** peuvent être entiers ou décimaux. Cette fonction retourne **x^y**.

puissance(x, y)
Les paramètres **x** et **y** peuvent être entiers ou décimaux. Cette fonction retourne **x^y**.

racine(x)
Le paramètre **x** peut être un entier ou un décimal positif ou nul. Cette fonction retourne la racine carré positive de x.

sin(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le sinus de **x** radians.

sinD(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne le sinus de **x** degrés.

sqrt(x)

Le paramètre **x** peut être un entier ou un décimal positif ou nul. Cette fonction retourne la racine carré positive de x.

tan(x)

Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne la tangente de **x** radians.

tanD(x)
Le paramètre **x** peut être un entier ou un décimal. Cette fonction retourne la tangente de **x** degrés.

4.12. Les images

Dans « Expresso » les images possèdent le type « ImageMemoire » et les seules images qui peuvent être utilisées doivent avoir un des trois formats suivants : gif, jpg ou png.
Une variable destinée à contenir une image doit donc être de type ImageMemoire, elle peut être créée de différentes façons :

ImageMemoire img = new ImageMemoire(); // Création d'une image vide dans la variable **img**

ImageMemoire img = chargerImage(nomFichier);
Dans ce dernier cas « **img** » contiendra l'image référée par « **nomFichier** ». La procédure à droite du signe « = » peut être remplacée par toute procédure qui retourne une « ImageMemoire » c'est à dire l'une des 4 premières procédure décrites ci-dessous.

chargerImage(nomFichier)

Le paramètre « nomFichier » est une chaine de caractère qui indique l'emplacement d'un fichier image sur l'ordinateur où fonctionne le programme . La procédure charge l'image en mémoire et retourne un objet de type « ImageMemoire » représentant l'image chargée. (nomFichier peut être un

chemin absolu ou relatif.)

chargerImageFD()
Comme ci-dessus mais l'adresse de l'image est donnée via une boite de dialogue.

chargerImageWeb(adresse)
Comme ci-dessus mais l'image réside sur un serveur et le paramètre « **adresse** » est l'adresse « **http** » de l'image.

chargerImageWeb()
Comme ci-dessus mais l'adresse est donnée via une boite de dialogue.

afficherImage(img, x, y, l, h)
Affiche à l'écran l'image « img » (de type ImageMemoire) dans le rectangle de coin supérieur gauche (x, y) et de dimensions largeur = l et hauteur = h. x et y sont des coordonnée Tortue.

enregistrerImageEcran(nomFichier, x, y, l, h)
Enregistre à l'emplacement « nomFichier » une partie de l'image à l'écran. Cette partie est déterminée par son coin supérieur gauche (x, y) sa largeur « l » et sa hauteur « h ». x et y sont des coordonnée Tortue. Si la chaine de caractères « nomFichier » se termine par une des extensions .gif, .jpg ou .png , cette extension déterminera le format sous lequel sera enregistrée l'image sinon le format utilisé sera automatiquement « png ».

enregistrerImage(nomFichier, imag)
Enregistre à l'emplacement « nomFichier » et sous le format « png », l'image « imag » contenue en mémoire.

enregistrerImageFD(imag)
Comme ci-dessus mais le nom et l'emplacement du fichier sont donnés via une boite de dialogue.

copierImageEcran(x, y, l, h)
Retourne une image de type « ImageMemoire » constituée par une partie de l'image à l'écran. Cette partie est déterminée par son coin supérieur gauche (x, y) sa largeur « l » et sa hauteur « h ». x et y sont des coordonnée Tortue.

copierImage(img, x, y, l, h)
Copie une partie d'une image en mémoire « img ». Cette partie est déterminée par son coin supérieur gauche (x, y) sa largeur « l » et sa hauteur « h ». **Attention** : on utilise ici les coordonnées internes de l'image, en particulier x, y, l, h doivent être entiers.

largeurImage(img)
Le paramètre « img » est de type « ImageMemoire » et la procédure retourne la largeur de l'image « img » en pixels.

hauteurImage(img)
Le paramètre « img » est de type « ImageMemoire » et la procédure retourne la hauteur de l'image « img » en pixels.

5. La programmation dans l'environnement Expresso

Pour programmer dans l'environnement Expresso il faut d'abord ouvrir l'environnement BlueJ en double cliquant son icône, et dans le menu « Projet » choisir l'item « Nouveau Projet ». Dans la fenêtre de dialogue qui s'ouvre il faut choisir un emplacement pour notre projet, lui donner un nom et cliquer sur le bouton « Enregistrer ».
Ceci étant fait, on programme dans un fichier « Expresso » obtenu en cliquant sur le bouton « Nouvelle classe », en sélectionnant « Expresso », dans la fenêtre de choix qui s'ouvre et en donnant le nom « Expresso » dans le champ de texte « Nom ».
Le fichier obtenu est divisé en plusieurs zones et au début de chacune de ces zones on trouve des indications de ce qu'on peut et ne peut pas y faire (*Pour une description en grands détails, consulter le tutoriel*).

5.1. Configuration de l'interface

5.1.1. Les variables de paramétrage de l'interface

Dans la zone « Paramètres de l'interface » on trouve plusieurs variables dont les valeurs peuvent êtres modifiées et qui permettent de paramétrer les différents éléments de l'interface.

La fenêtre

baseFenetre = ... ; cette variable entière (**int**) fixe la largeur de la fenêtre en pixels. On peut modifier sa valeur, en inscrivant un entier après le signe =.

hauteurFenetre = ... ; cette variable entière (**int**) fixe la hauteur de la fenêtre en pixels. On peut modifier sa valeur, en inscrivant un entier après le signe =.

titreFenetre = ... ; cette variable chaîne de caractères (**String**) fixe le titre de la fenêtre. On peut modifier sa valeur, en inscrivant une phrase entre guillemets ("...") après le signe =.

zoneTexte = true (ou **false**); cette variable booléenne (**boolean**) indique si on veut avoir une zone de texte (valeur **true**) ou si on ne veut pas en avoir (valeur **false**).

Les boutons

On peut également avoir, dans l'interface, des boutons (maximum 16) disposés sur une ou deux lignes:

La variable **nomsBoutonsLigne1** reçoit les noms des boutons de la première ligne et la variable **nomsBoutonsLigne2** reçoit les noms des boutons de la deuxième ligne. Les noms de boutons sont des chaînes de caractères entre guillemets et sont séparés par des virgules. Par exemple

nomsBoutonsLigne1 = {"Dessiner", "Quitter"};

fera apparaître deux boutons nommés **Dessiner** et **Quitter** sur une ligne. Si on ajoute

nomsBoutonsLigne2 = {"Effacer"};

un bouton nommé **Effacer** apparaîtra sur une deuxième ligne au dessous des deux autres boutons. Lorsque une de ces variables est laissée vide la ligne de boutons correspondante n'apparaît pas. Si les deux variables sont vides il n'y a pas de boutons. Cependant, même vides, ces variables doivent être présentes, sans quoi il y aura une erreur de compilation.

Le bouton d'interruption

On peut aussi créer un bouton d'interruption pour permettre à un utilisateur éventuel d'arrêter en cours de route une action qu'il trouve trop longue à s'exécuter.

Pour créer un tel bouton on précède son nom des caractères « Interruption_ » dans la variable nomBouton1 ou 2 où il est déclaré.

Pour l'utiliser, il faut d'abord déclarer que les interruptions seront permises avec la procédure « permettreIntrruption() » que l'on place dans la procédure initialisation. Ensuite, dans les actions

que l'on veut pouvoir interrompre, il faut vérifier périodiquement si une interruption est demandée à l'aide de la procédure « interruption() » (qui retourne « true » si une interruption a été demandée et « false » autrement) et insérer les instructions nécessaires pour arrêter l'action en cours si l'interruption a été demandée.

Les champs de textes

Il est possible d'insérer dans l'interface, au dessous des boutons, une ou deux lignes de champs de textes :

La variable **nomsChampsLigne1** reçoit les légendes (les textes qui apparaîtront devant les champs) et les longueurs (en nombre de caractères) des champs de la première ligne et la variable **nomsChampsLigne2** reçoit les noms et longueurs des champs de la deuxième ligne. Les légendes et longueurs des champs sont des chaînes de caractères et sont séparés par des virgules. Par exemple :

nomsChampsLigne1 = {"Longueur", "20", "Hauteur", "43"};

fera apparaître deux champs précédés des légendes **Longueur** et **Hauteur** sur une ligne le premier montrant 20 caractères et le second 43. Si on ajoute

nomsChampsLigne2 = {"Profondeur", "27"};

un champs nommé **Profondeur** apparaîtra sur une deuxième ligne au dessous des deux autres champs et pourra montrer jusqu'à 27 caractères.

Lorsque une de ces variables est laissée vide la ligne de champs correspondante n'apparaît pas. Si les deux variables sont vides il n'y a pas de champ. Cependant, même vides, ces variables doivent être présentes, sans quoi il y aura une erreur de compilation.

Les menus

Pour créer des menus (maximum 8) il suffit, comme pour les boutons, d'écrire les noms des menus et de leurs items dans les variables **nomsMenus**. Par exemples

nomsMenus1 = {"Graphique", "Dessiner", "Effacer", "Quitter"};

créera un premier menu nommé **Graphique** avec pour items de menu, dans l'ordre, **Dessiner**, **Effacer**, **Quitter**.

Il y a huit variables **nomsMenus1**, **nomsMenus2**, **...**, **nomsMenus8** permettant de créer jusqu'à huit menus. Ces variables doivent toujours être présentes, elles peuvent cependant être vides. Lorsqu'une de ces variables est vide elle est ignorée **ainsi que toutes celles qui suivent**. Par exemple dans l'exemple ci-dessus si la variable **nomsMenus2** est vide

nomsMenus2 = {};

il n'y aura qu'un seul menu (le menu Graphique) que les autres variables soient vides ou pas.

De plus chaque menu ne doit contenir qu'un maximum de 8 items, les items supplémentaires s'il y en a seront ignorés.

Une fois les menus créés on peut par programmation changer les noms des items qu'ils contiennent à l'aide de la procédure

- **changerNomItemMenu(numeroMenu, numeroItem, nouveauNom).**

Les menus spéciaux

On peut créer des menus dont les items sont des cases à cocher ou des boutons poussoirs. Pour cela il suffit de précéder le nom du menu par les caractères « s_ » pour un menu « cases à cocher » et par « c_ » pour un menu « boutons poussoirs ». Un menu « cases à cocher » permet de faire des sélections multiples c'est à dire que plusieurs cases peuvent être cochées simultanément. Par contre un menu « boutons poussoirs » ne permet qu'une sélection unique, lorsqu'on sélectionne un item son bouton poussoir devient sélectionné et les autres deviennent désélectionnés.

Par programmation on peut vérifier l'état des items de ces menus à l'aide des procédures

- **etatMenuItem(numeroMenu, numeroItem)** qui retourne « true » si l'item de menu

correspondant est sélectionné et « false » autrement.

- **selectionMenuItem(numeroMenu, numeroItem, b)** qui sélectionne l'item de menu correspondant si « b » est « true » et le désélectionne si « b » est « false ».

À part ces particularités, ces menus se comportent exactement comme les autres.

La procédure initialisation

Toujours dans la zone « Paramètres de l'interface » on trouve une procédure « initialisation() ». C'est dans cette procédure qu'on doit placer les instructions qu'on veut voir exécutées dès l'ouverture du programme.

5.1.2. La création des glissières

On peut ajouter jusqu'à 12 glissières réparties sur deux lignes (avec un maximum de 6 par ligne) en invoquant les méthodes **ajouterGlissiereLigne1(...)** et **ajouterGlissiereLigne2(...)** dans la méthode **ajoutDeGlissieres()**. Les méthodes **ajouterGlissiereLigne..** servent à configurer les glissières mais c'est la méthode **ajoutDeGlissieres()** qui les ajoute réellement à l'interface. C'est pourquoi l'invocation des méthodes **ajouterGlissiereLigne..** doit toujours se faire dans la méthode **ajoutDeGlissieres()**.

Les méthodes **ajouterGlissiereLigne1 et ajouterGlissiereLigne2** possèdent 3, 4 ou 5 paramètres entiers, les trois premiers sont obligatoires et donnent la légende, les valeurs minimale et maximale de la glissière. Le quatrième paramètre indique la position initiale du curseur. S'il est absent le curseur sera sur la valeur minimale. Le cinquième paramètre indique le nombre de décimales voulues. S'il est absent il est considéré comme nul, c'est à dire 0 décimales. Il est a noter que la valeur retournée par la méthode **valeurGlissiere(numéro)** est toujours un nombre décimal (type double), ceci même lorsque la glissière est configurée avec 0 décimales.

La valeur d'une glissière apparaît dans un champ situé au-dessus. Un utilisateur peut modifier cette valeur en déplaçant le curseur de la glissière, en tapant une nouvelle valeur dans le champ ou en utilisant les flèches du clavier lorsque la glissière est sélectionnée.

5.2. Les actions associées aux éléments d'interface.

5.2.1. Les actions des boutons

Les boutons sont numérotés dans l'ordre ou on les rencontre dans les variables **nomsBoutons1** puis **nomsBoutons2**. À chaque bouton est associé une méthode dont le nom est composé du mot **actionBouton** suivi du numéro du bouton cette méthode sera automatiquement invoquée lorsqu'on cliquera sur le bouton et provoquera une action correspondant au programme qu'elle contient. Par exemple pour associer une action au bouton numéro 1 on écrira

```
public void actionBouton1(){ // sera déclenchée par un clic sur le bouton 1
instructions;
}
```

Les méthodes **actionBouton*i*** inutiles peuvent être détruites sans danger.

5.2.2. Les actions des menus

Les menus sont numérotés comme les variables qui les contiennent, donc le menu 1 est le menu contenu dans la variable **nomsMenus1**, le menu 2 est celui contenu dans la variable **nomsMenus2**, etc.

À l'intérieur d'un menu les items sont numérotés dans l'ordre ou ils sont rencontrés dans la variable **nomsMenus** correspondante. Par exemple si on a déclaré

nomsMenus3 = {"Graphique", "Dessiner", "Effacer", "Quitter"};

Graphique sera le menu 3, **Dessiner** sera l'item 1 du menu 3, **Effacer** sera l'item 2 du menu 3, **Quitter** sera l'item 3 du menu 3.
A chaque item de menu est associée une méthode qui sera invoquée lorsqu'on sélectionnera cet item. Cette méthode a un nom composé comme suit:
le mot **actionMenu** suivi du numéro du menu, suivi du mot **Item**, suivi du numéro de l'item de menu.
Par exemple pour associer une action à l'item de menu **Effacer** de l'exemple ci-dessus on programmera :

```
public void actionMenu3Item2(){ // puisque Effacer est l'item 2 du menu 3
instructions;
}
```

5.2.3. Les actions des glissières

Les glissières sont numérotées de gauche à droite dans l'ordre ou elles apparaissent d'abord sur la ligne 1 puis sur la ligne 2.
La valeur d'une glissière correspond à la position de son curseur et est toujours indiquée dans le champ de texte situé au dessus de la glissière. Par programmation on peut consulter la valeur de la glissière numéro i à l'aide de la méthode **valeurGlissiere(i).** Il est important de noter que cette valeur est toujours un nombre décimal. On peut modifier la valeur d'une glissière en déplaçant son curseur ou par programmation à l'aide de la méthode **fixeValeurGlissiere(i, v)**
qui donnera à la glissière numéro i la valeur v.
On peut associer une action à chaque glissière, action qui sera déclenchée lors du déplacement du curseur. Pour cela on place les instructions qu 'on désire voir exécutées lors de la manipulation de la glissière numéro *i* dans la méthode **actionGlissiere*i*(d)**:

```
public void actionGlissiere(d){
...instructions
}
```

Cette méthode sera invoquée lors du déplacement du curseur de la glissière #i.
Il faut noter que le paramètre d de cette méthode prendra, lors de l'exécution, la valeur de la glissière on peut donc l'utiliser dans *instructions* à la place de **valeurGlissiere(i)**.

5.2.4. Les actions associées à la souris

Il y a quatre actions associées à la souris:

- Le « clic », qui correspond à peser sur le bouton et le relâcher sans déplacer la souris.
- Le « glisser » qui se décompose en 3 actions: peser sur le bouton, déplacer la souris et relâcher le bouton.

On peut faire exécuter des instructions pour répondre à chacun de ces 4 événements.
Par exemple les instructions à exécuter pour répondre à un clic doivent être placées dans la méthode clicSouris(x, y) ou x et y représentent les coordonnées du point cliqué en coordonnées tortue. On écrira donc

```
public void clicSouris(double x, double y){
    instructions pouvant utiliser x et y
}
```

Les autres méthodes associées à la souris sont
debutGlisser(x, y), finGlisser(x, y), glisserEnCours(x, y)
elles sont aussi invoquées lors de l'événement correspondant, debutGlisser(x, y) lorsqu'on pèse sur le bouton au début d'un glissement, glisserEnCours(x, y) pendant le déplacement de la souris

avec le bouton appuyé et finGlisser(x, y) lorsqu'on relâche le bouton de la souris après un déplacement. Dans tous les cas les variables x et y sont des nombres décimaux (type double) qui représentent les coordonnées du point ou se trouve la souris lors de l'événement. Ces coordonnées sont toujours des coordonnées tortue.

6. Pour continuer

Dans ce manuel nous venons de faire le tour de l'environnement et du langage d'Expresso.
Pour continuer on peut également trouver sur le site

http://www.math.uqam.ca/expresso/

un tutoriel qui expose en détail la programmation de quelques exemples.
Sur ce même site on peut visionner quelques applets dont les sources sont disponibles.
Nous espérons que l'ensemble de cette documentation permettra à toutes les personnes intéressées d'utiliser Expresso avec autant de plaisir que nous en avons.

Annexe

7. L'analyse syntaxique dans Expresso

7.1. Introduction

Supposons qu'avec Expresso vous vouliez construire un programme qui permet de tracer des graphes de fonctions données par l'utilisateur du programme. Vous commencerez par ouvrir une boîte de dialogue demandant à l'utilisateur de donner l'expression définissant sa fonction, mais ce que cela fournira à votre programme ne sera, du point de vue informatique, qu'une chaîne de caractères. Le problème qui se posera alors sera : « Comment fournir au programme un moyen de reconnaître dans cette chaîne de caractères les variables, les fonctions, les opérateurs, l'ordre d'évaluation, etc ? ».
C'est pour répondre de façon opérationnelle à cette question que nous avons adjoint à Expresso la bibliothèque JEP (Java expression parser, http://www.singularsys.com/jep/) et un petit nombre de procédures permettant d'exploiter, dans Expresso, les capacités de JEP.
Ces procéduress qui permettrons d'analyser une chaîne de caractères pour répondre à la question ci-dessus et d'obtenir un objet informatique d'un nouveau type (Expression) qui contient toutes les informations désirées : les variables, les fonctions, les opérateurs, l'ordre d'évaluation et même la position des erreurs, au cas ou l'expression fournie n'est pas considérée comme valide.
De plus, à l'aide d'un objet de type Expression on pourra, en donnant des valeurs aux variables, évaluer la fonction pour la valeur de ces variables.
Nous donnerons un peu plus loin la liste des procédures disponibles et leurs définitions mais d'abord, pour rendre un peu plus concret le discours précédent, illustrons le par quelques exemples:
considérons la fonction $f(x)=a\cos(x)+b\ln(x)$
l'instruction `Expression f = analyse( "a cos(x) + b ln(x)" );`
fournit un objet informatique de type expression nommé « f ». Remarquez que la procédure « **analys**e » a reçu, pour construire « f » une chaîne de caractères.
Il est possible que l'analyse de l'expression n'ait pas réussi, on peut le savoir en exécutant l'instruction:

int i = **posErreur**(f);

si on obtient pour « i » la valeur 0 c'est que l'analyse a réussi, sinon « i » aura une valeur entière positive qui indiquera le rang du premier caractère fautif dans la chaîne initiale "a cos(x) + b ln(x)".
On pourra ensuite assigner des valeurs aux variables avec des instructions du type:

var(f, "a", 5);

qui signifie : donner à "a" la valeur 5 dans l'expression « f ».
Remarque : de notre point de vue dans $f(x)=a\cos(x)+b\ln(x)$ x est la variable et a et b sont des paramètres. Par contre dans l'objet « f » x , a et b ont exactement le même statut, ce sont des variables. Ce sera au programmeur de savoir faire la distinction dans son programme si nécessaire.
On peut aussi retrouver la valeur de "a" à l'aide d'une instruction de la forme:

double u = **valVar**(f, "a");

et « u » contiendra la valeur 5. Remarquons encore que dans les procédures **var** et **valVar** la variable «"a" est donnée sous forme de chaîne de caractères.
Assignons maintenant des valeurs à toutes les variables :

var(f, "b", 2);
var(f, "x", Pi);

maintenant que toutes les variables ont des valeurs l'expression peut être évaluée :

double z = **eval**(f);

et « z » contiendra la valeur de $5\cos(pi)+2\ln(pi)$ c'est à dire approximativement 2,71.

On peut savoir si une variable est présente dans l'expression :

boolean t = **existeVar**(f, "x"); // sera « true »

boolean s = **existeVar**(f, "q"); // sera « false »

On peut savoir le nombre de variables et leurs noms :

int j = **ndDeVariables**(f); // j contiendra le nombre 3

String[] table = **nomsDesVariables**(f); // table sera un tableau de chaînes de caractères contenant "a", "x" et "b".

Dans ce dernier cas les noms des variables n'est pas nécessairement dans cet ordre et de plus l'ordre peut changer lors de l'ajout de nouvelles variables.

Ce petit tour d'horizon devrait avoir donné une idée de certaines des procédures disponibles et de la façon de les utiliser.

7.2.Les procédures disponibles

Expression analyse(String chaine)

Analyse la chaîne de caractères « chaine » et retourne un objet de type Expression qui contient toutes les informations nécessaires à l'évaluation de « chaine » considéré comme une expression mathématique.

void var(Expression formule, String nom, double val)

Dans « formule », assigne à la variable « nom » la valeur « val ».

double valVar(Expression formule, String nom)

Retourne la valeur (de type double) de la variable de nom « nom » dans « formule »

double eval(Expression formule)

Retourne le résultat de l'évaluation de « formule ». Il faut évidemment que toutes les variables aient une valeur sinon on obtient une erreur.

double eval(Expression formule,double z)

Retourne la valeur de l'évaluation de « formule » avec pour la variable « x » la valeur « z ». Il faut évidemment que toutes les autres variables aient une valeur sinon on obtient une erreur.

int posErreur(Expression formule)

Retourne le rang du premier caractère fautif dans la chaîne de caractères qui a servi à construire l'objet « formule ». La valeur 0 indique qu'il n'y a pas d'erreur et que l'analyse a réussi.

String[] nomsDesVariables(Expression formule)

Retourne un tableau de chaines de caractères qui sont les noms des variables dans « formule ».

int nbDeVariables(Expression formule)

Retourne un entier qui est le nombre de variables dans « formule ». C'est aussi la dimension du tableau retourné par « **nomsDesVariables »**

boolean existeVar(Expression formule, String nom)

Retourne « true » si la variable de nom « nom » existe déjà dans « formule » et « false » sinon.

void ajouteVar(Expression formule, String nom, double val)

Si la variable de nom « nom » n'existe pas elle est ajoutée à la table avec la valeur « val ». Si elle existe déjà sa valeur est changée pour « val ».

void ajouteVar(Expression formule, String nom)

Si la variable de nom « nom » n'existe pas elle est ajoutée à la table avec la valeur 0. Si elle existe déjà sa valeur est changée pour 0.

String expressionDefinissante(Expression formule)

Retourne la chaîne de caractére initiale qui a servi à la construction de l'objet « formule »

7.3.L'écriture des expressions

7.3.1.Fonctions

Voici une liste des fonctions disponibles dans l'écriture des expressions :

Noms	**Syntaxe**
Sinus	sin(x)
Cosinus	cos(x)
Tangente	tan(x)
Arc Sinus	asin(x)
Arc Cosinus	acos(x)
Arc Tangente	atan(x)
Arc Tangente (avec 2 paramètres)	atan2(y, x)
Sinus Hyperbolique	sinh(x)
Cosinus Hyperbolique	cosh(x)
Tangente Hyperbolique	tanh(x)
Sinus Hyperbolique Inverse	asinh(x)
Cosinus Hyperbolique Inverse	acosh(x)
Tangente Hyperbolique Inverse	atanh(x)
Logarithme Naturel	ln(x)
Logarithme à base 10	log(x)
Exponentielle (e^x)	exp(x)
Valeur Absolue	abs(x)
Nombre aléatoire (>= 0 et < 1)	rand()
Modulo	mod(x,y) = x % y
Racine carré	sqrt(x)
Somme	sum(x,y,z)
Si (ou if)	si(condition,valeur si vrai,valeur si fauxl)
Transforme le nombre x en chaine	str(x)

7.3.2.Opérateurs

Voici une liste des opérateurs disponibles pour l'écriture des expressions

Noms	Symboles
Puissance	^

Négation	!
Plus unaire, Moins unaire	+x, -x
Modulo	%
Division	/
Multiplication	*
Addition, Soustraction	+, -
Plus petit ou égal, Plus grand ou égal	<=, >=
Plus petit, Plus grand	<, >
Différent, Égal	!=, = =
Et	&&
Ou	||

7.3.3.Remarques sur la syntaxe:

dans les expressions à analyser on peut omettre les signes de multiplication, il faut cependant les remplacer par des espaces. Par exemple $x\,y$ est équivalent à $x*y$, par contre xy sera interprété comme le nom d'une variable.
Il existe quelques cas ou cette règle ne s'applique pas, par exemple devant ou après une parenthèse $x(...)$ et $(...)x$ seront bien interprétés comme des produits de même que 2x , mais, d'un autre côté, il faut être prudent car $x2$ sera lui interprété comme le nom d'une variable.