Tutoriel Expresso

André Boileau
boileau.andre@uqam.ca

Maurice Garançon
garanson@math.uqam.ca

Table des matières

Introduction

Dans ce tutoriel nous allons essayer de montrer comment exploiter **Expresso** en réalisant de petits programmes qui utilisent ses différentes possibilités.

Pour profiter pleinement de ce tutoriel il faut le lire tout en programmant les exemples détaillés. Pour cela il faut au préalable avoir installé **Java**, **BlueJ** et les différents fichiers d'**Expresso**. Si ce n'est pas déjà fait, c'est le moment de le faire en vous référant au chapitre 2 du manuel de l'utilisateur.

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 « NouveauProjet ». 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 ».

La fenêtre de **BlueJ** change alors d'apparence, tous les boutons qui étaient grisés deviennent actifs. Il faut commencer par créer un fichier qui contiendra notre programme, pour ça cliquons sur le bouton 'Nouvelle classe » (voir Figure 1).

Figure 1

Dans la fenêtre qui s'ouvre nous avons de nombreux choix mais un seul nous intéresse: sélectionnons le « bouton poussoir » « Expresso » et dans le champ de texte « Nom » inscrivons **Expresso** (malheureusement nous n'avons pas le choix, tous les fichiers contenant nos programmes devrons s'appeler **Expresso)** (voir Figure 2).

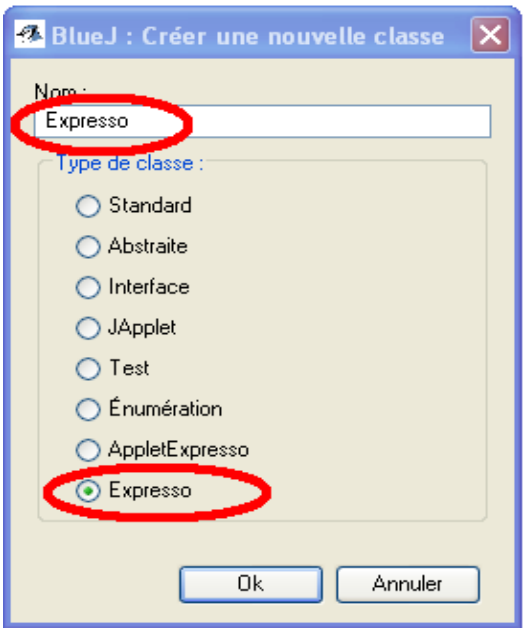


Figure 2

On obtient alors un fichier nommé « Expresso ». Notre programme devra être écrit dans ce fichier.

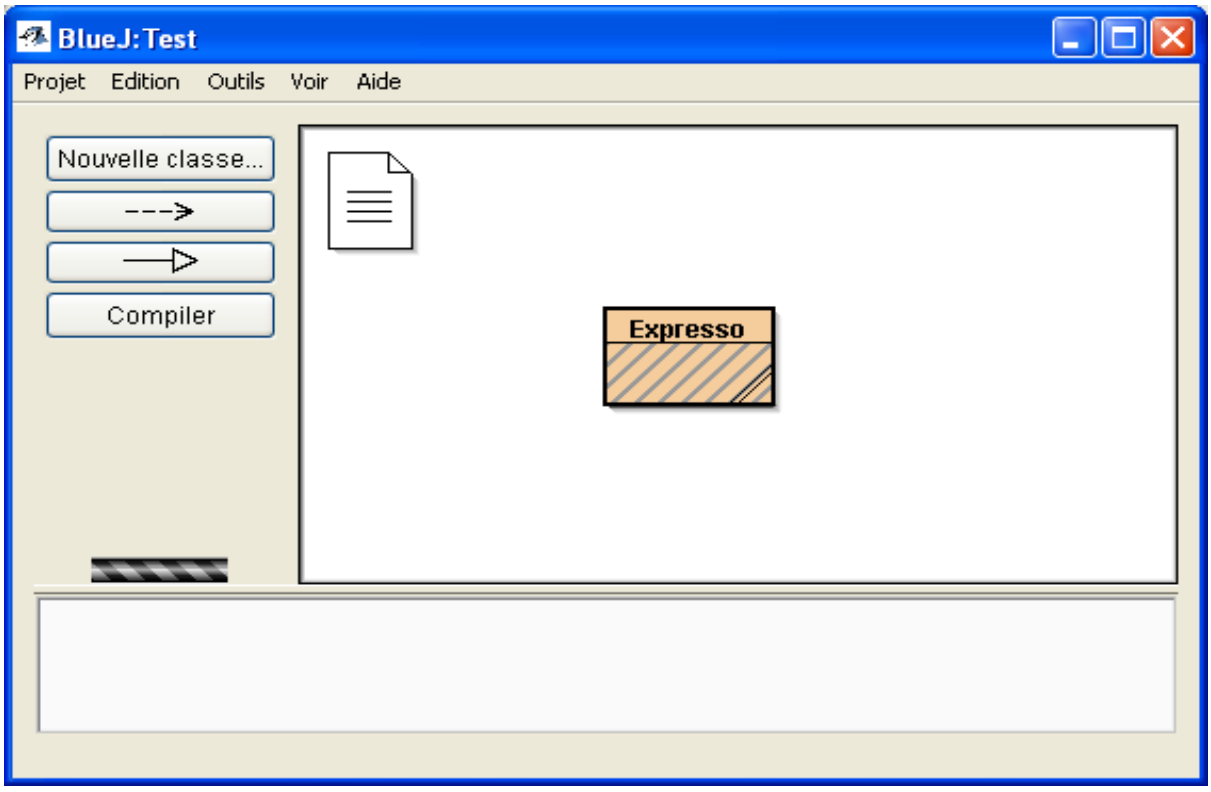


Figure 3

On ouvre le fichier "Expresso" en double cliquant son icône ou en la cliquant avec le bouton droit (Ctrl-Clic pour les souris à un seul bouton) et en choisissant « Éditer » dans le menu contextuel.

L'éditeur de BlueJ s'ouvre alors avec le fichier suivant :

```
/*************************************************************************
*
*               NE PAS MODIFIER CETTE SECTION
*
*************************************************************************/
import java.awt.*;
import Listes.*;

public class Expresso extends FenetreTortue{

/*************************************************************************
*
*                  SECTION MODIFIABLE
*
*************************************************************************/

/********************** Parametres de l'interface ************************/

public static int baseFenetre = 600; // Largeur de la fenêtre
public static int hauteurFenetre = 500; // Hauteur de la fenêtre
public static String titreFenetre = ""; // Titre de la fenêtre

// Le texte
public static boolean zoneTexte = false; // Indique si on désire ou non une zone de texte

// Les boutons
public static String[] nomsBoutonsLigne1 = {}; // Noms des boutons de la ligne 1
public static String[] nomsBoutonsLigne2 = {}; // Noms des boutons de la ligne 2

// Les menus
public static String[] nomsMenus1 = {};
public static String[] nomsMenus2 = {}; // Laissez vide si vous ne désirez pas de menu
public static String[] nomsMenus3 = {};
public static String[] nomsMenus4 = {};
public static String[] nomsMenus5 = {};
public static String[] nomsMenus6 = {};
public static String[] nomsMenus7 = {};
public static String[] nomsMenus8 = {};

public static void ajoutDeGlissieres(){
}


/****************** Initialisation ********************************/

public void initialisation(){
        // Placer ici les actions a realiser a l'ouverture
```

Cette section doit absolument être présente. À ne modifier sous aucun prétexte

```
}
/****************** Placer vos procedures ici ***************************/


/*************  Les actions des boutons  ********************************/
public void actionBouton1(){
}
public void actionBouton2(){
}
// etc...
/*************  Les actions des menus  ***************************/
public void actionMenu1Item1() {
}
public void actionMenu1Item2() {
}
// etc..
/*************** Les actions des glissieres ********************************/
public void actionGlissiere1(double d) {
}
public void actionGlissiere2(double d) {
}
public void actionGlissiere3(double d) {
}
/**************** Les actions de la souris ******************************/
public void clicSouris(double x, double y){}
public void debutGlisser(double x, double y){}
public void finGlisser(double x, double y){}
public void glisserEnCours(double x, double y){}
/*************************************************************************
*
*               NE PAS MODIFIER CETTE SECTION
*
*************************************************************************/
public static String[][] Menus = {nomsMenus1, nomsMenus2, nomsMenus3, nomsMenus4,
nomsMenus5, nomsMenus6, nomsMenus7, nomsMenus8};
public Expresso(int l, int h, String titre, String[] nomsBoutons1, String[] nomsBoutons2, String[][]
Menus, boolean avecTexte){
        super(l, h, titre, nomsBoutons1, nomsBoutons2, Menus, avecTexte);
```

```
}
public Expresso() {
        this(baseFenetre, hauteurFenetre, titreFenetre, nomsBoutonsLigne1, nomsBoutonsLigne2,
Menus, zoneTexte);
}
public static void executer(boolean applet){
        initGlissieres();
        ajoutDeGlissieres();
        Expresso maFenetre = new Expresso();
        faireApplet(applet);
        maFenetre.toFront();
        maFenetre.initialisation();
}
public static void main(String[] args){
        executer(false);}
}
```

Encore une section à ne pas modifier pour assurer le bon fonctionnement.

On trouve dans ce fichier différentes sections. Celles de début et de fin ne doivent pas être modifiées sous peine de perturber le bon fonctionnement des programmes. S'il arrivait qu'une de ces sections soit modifiée par inadvertance, il faudrait rouvrir un fichier "Expresso" intact et recopier les deux sections impliquées. Par la suite nous identifierons les autres sections par leur nom mais aussi par leur position dans le fichier (première, deuxième, etc) pour faciliter la navigation dans le fichier.
Voici donc les différentes sections et leur rôle:

- Paramètres de l'interface (première section)
  Cette section contient principalement des variables dont la présence est nécessaire au bon fonctionnement des programmes. Seules leurs valeurs peuvent être modifiées pour adapter l'interface à nos besoins.
- Initialisation (deuxième section)
  Cette section contient une procédure « initialisation() » qui est appelée par le système aussitôt que la fenêtre principale est créée. C'est dans cette procédure qu'on doit placer toutes les actions qu'on veut voir exécutées dès l'ouverture de la fenêtre.
- Placer vos procédures ici (troisième section)
  Cette section va contenir l'ensemble de nos procédures à l'exception de celles, prédéfinies, qui servent à gérer les actions déclenchées par des événements.
- Les actions des boutons (quatrième section)
  Cette section contient des exemples de procédures prédéfinies, associées aux boutons, et qui sont appelées automatiquement lorsqu'on clique sur un bouton correspondant. C'est dans ces procédures qu'on place les actions que l'on veut voir exécutées lors d'un clic sur un bouton.
- Les actions des menus (cinquième section)
  Cette section contient des exemples de procédures prédéfinies, associées aux items de menus, et qui sont appelées automatiquement lorsqu'on sélectionne un item de menu. C'est dans ces procédures qu'on place les actions que l'on veut voir exécutées lors de la sélection d'un item de menu.
- Les actions des glissières (sixième section)
  Cette section contient des exemples de procédures prédéfinies, associées aux glissières, et qui sont appelées automatiquement lorsqu'on modifie l'état de la glissière associée. C'est dans ces

procédures qu'on place les actions que l'on veut voir exécutées lors de la modification de l'état d'une glissière.

- La gestion des événements souris (septième section)
  Cette section contient quatre procédures prédéfinies « clicSouris() », « debutGlisser() », « glisserEnCours() » et « finGlisser » qui sont associées aux événements « clic de souris », « début de déplacement de la souris avec le bouton enfoncé », « déplacement de la souris avec le bouton enfoncé » et « relâchement du bouton de la souris après un déplacement » respectivement. Lorsque l'un de ces événements se produit dans la fenêtre graphique la procédure correspondante est appelée automatiquement. C'est donc dans ces procédures qu'on doit placer les actions qu'on veut voir exécutées lors de l'événement correspondant.

Cette division en sections n'est là que pour nous aider à structurer nos programmes. La présence et l'ordre de ces sections n'ont rien d'essentiel, nous trouvons cependant pratique de les conserver et de les utiliser.

Comment exécuter vos programmes dans BlueJ

Une fois votre programme écrit dans le fichier « Expresso » il faut le compiler soit en choisissant «compiler» dans la partie gauche de la fenêtre de BlueJ, soit en cliquant avec le bouton droit sur le fichier «Expresso» et en choisissant Compiler dans le menu contextuel qui s'ouvre.

Si tout se passe bien (les rayures présentes sur l'image du fichier Expresso disparaissent), il faut cliquer sur le fichier Expresso avec le bouton droit et dans la liste qui s'affiche sélectionner «void main(String[] args)» et cliquer «OK» dans la fenêtre qui s'ouvre. Votre programme doit alors s'exécuter.

Après avoir fait connaissance avec l'environnement passons aux choses sérieuses .. programmons!

Exemple 1: Calcul du PGCD de deux entiers positifs.

Nous commençons avec un exemple purement textuel, le calcul du PGCD de deux nombres entiers avec différents affichages du résultat et différentes façons d'entrer les données. Une version de ce programme terminé est disponible sous forme d'applet sur le site

http://www.math.uqam.ca/_boileau/AMQ2005.html .

Étape 1: préparation

Tout d'abord intéressons nous à la méthode de calcul d'un pgcd. Nous allons utiliser un algorithme connu sous le nom «**algorithme d'Euclide**»

Commençons avec un exemple numérique. Par exemple calculer le pgcd de 127 et 19. L'algorithme consiste à exécuter une suite de divisions avec reste:

$$dividende = diviseur \cdot quotient + reste$$

en passant d'une division à la suivante en prenant respectivement le diviseur et le reste de la première pour dividende et diviseur de la suivante. Le dernier reste non nul est le pgcd.

Illustrons le processus, on débute avec nos deux nombres 127 et 19 pour dividende et diviseur respectivement:

$127 = 19 \cdot 6 + 13$	*On passe d'une division à la suivante en prenant respectivement le diviseur et le reste de la première pour dividende et diviseur de la suivante.*
$19 = 13 \cdot 1 + 6$	
$13 = 6 \cdot 2 + 1$	
$6 = 1 \cdot 6 + 0$	*On termine lorsqu'on obtient un reste nul. Le pgcd est alors le dernier reste non nul, qui est aussi le diviseur de la dernière division.*

L'application de l'algorithme à nos deux nombres 127 et 19 nous permet de conclure que leur pgcd est 1.

Écrivons maintenant ce processus sous une forme générale applicable à tout couple d'entiers positifs. Si **u** et **v** sont deux entiers positifs on commence par effectuer la division entière (ou division avec reste) de u par v

$u = v \,.\, q_1 + r_1$

ou q_1 est le quotient entier et r_1 est le reste. On a $0 \leq r_1 < v$.

On recommence avec v comme dividende et r_1 comme diviseur:

$v = r_1 \,.\, q_2 + r_2$ avec $0 \leq r_2 < r_1$.

Et on continue comme ça jusqu'à ce qu'on trouve un reste nul.

$r_1 = r_2 \,.\, q_3 + r_3$ avec $0 \leq r_3 < r_2$.

...

...

$r_n = r_{n+1} . q_{n+2} + 0$

Puisque les restes forment une suite strictement décroissante d'entiers positifs on est sûr que le processus va se terminer le dernier reste non nul (ici r_{n+1}) étant le pgcd de u et v.

Avant de programmer le processus nous allons le formaliser un peu en utilisant toujours les mêmes variables: **a** pour représenter les dividendes, **b** pour les diviseurs, **d** pour les quotients et **c** pour les restes. Nous obtenons la forme suivante

a = u

b = v

Tant que b > 0

effectuer la division c'est à dire:

d = quotient entier de a par b

c = reste de la division de a par b

préparer la division suivante

a = b

b = c

Lorsque la division donne un reste nul (**c** = 0) le dernier reste non nul est **b** qui est donc le pgcd, Lorsqu'on prépare la division suivante **a** devient le pgcd (**a** = **b**) et **b** devient nul (**b** = **c** = 0) et le processus s'arrête puisqu'on ne l'effectue que ***Tant que b > 0***.

Nous avons donc formalisé le processus de façon à ce que lorsqu'il prend fin la valeur de la variable **a** soit le pgcd cherché.

Étape 2: programmation de l'algorithme

Représentons le processus que nous venons de décrire ci-dessus dans le langage **Expresso**. Pour cela nous écrivons le texte ci-dessous dans le fichier Expresso dans la troisième section «**Placer vos procédures ici**»

```
//****************** Placer vos procédures ici ********************
  public int pgcd(int u, int v){
        int a = u;
        int b = v;
        int c;
        int d;
        while (b>0) {
           c= mod(a, b);
           d = divEnt (a, b);
           a=b;
           b=c;
        }
        return a;
     }
```

Un certain nombre d'éléments purement informatiques sont apparus dans la traduction, passons les en revue :

public *:* pour simplifier les choses disons qu'il faut toujours l'écrire au début de la définition d'une procédure.

public ***int*** : ici ***int*** indique que la procédure que nous écrivons est en fait une fonction qui va retourner un entier (le pgcd).

pgcd(int u, int v) : ***pgcd*** est le nom de la procédure et ***(int u, int v)*** indique que cette procédure a besoin de deux entiers pour faire son travail (i.e. retourner le pgcd de **u** et **v**).

{ : marque le début de la définition de la procédure, définition qui doit toujours être incluse entre **{** et **}**.

int a = u; : est une instruction et toute instruction doit se terminer par **;**. Cette instruction dit que **a** est une variable de type entier (int) et qu'elle prend la valeur de **u**. (Dans Expresso, comme en Java, toutes les variables doivent avoir un type). On remarque aussi le point virgule a la fin, il est essentiel: toute instruction doit se terminer par un point virgule.

int b = v; int c; int d; : de même **b** est une variable de type entier qui prend la valeur de **v** et **c** et **d** sont des variables de type entier qui, pour le moment n'ont pas de valeur..

while (b > 0) : est la traduction en langage Expresso de ***Tant que b > 0*** et tout ce qui devra être fait pendant que **b>0** doit suivre entre **{** et **}**.

mod (a, b) est une fonction d'Expresso qui fournit le reste de la division de **a** par **b**.

divEnt(a, b) est une fonction d'Expresso qui fournit le quotient entier de la division de **a** par **b**.

return a est l'instruction qui retourne la valeur de **a**.

On peut, si on veut, compiler notre première procédure en cliquant sur le bouton «Compiler» en haut à gauche dans la fenêtre de BlueJ ou en cliquant avec le bouton droit (Ctrl-clic pour les souris à 1 seul bouton) sur l'icône du fichier Expresso et en choisissant «compiler».

Si tout se passe bien, la représentation graphique du fichier Expresso qui était hachurée (indication que le fichier n'est pas compilé) devient non hachurée (indication que la compilation a réussi) et donc que notre programme a du sens.

Cependant même compilé notre programme n'est pas très utile il nous faut une façon de le déclencher en lui donnant les deux nombres dont on veut le PGCD. Pour cela écrivons une autre procédure qui nous permettra de fournir deux entiers à la procédure « pgcd ». Nous plaçons cette nouvelle procédure à la suite de la procédure « pgcd » dans la troisième section «Placer vos procédures ici».

```
public void faireCalcul(){
  int a = demanderEntier("Calcul du pgcd (#1/2)", "Donnez un premier nombre   :", 34);
  int b = demanderEntier("Calcul du pgcd (#2/2)", "Donnez un second nombre    :", 71);
  int c = pgcd(a,b);
  message("Le pgcd de "+a+" et de "+b+" est "+c);
}
```

Expliquons les éléments nouveaux:

public **void** faireCalcul() : **void** indique que la procédure nommée «**faireCalcul**»ne retourne aucun résultat.

demanderEntier est une procédure d'Expresso qui ouvre une fenêtre de dialogue dans laquelle l'utilisateur peut taper un entier. C'est cet entier qui sera retourné par la procédure et, ici, placé dans la variable **a**. La procédure **demanderEntier** nécessite 2 ou trois arguments:

- le premier est le titre de la fenêtre à ouvrir, ici **"Calcul du pgcd (#1/2)",** les guillemets indiquent qu'il s'agit d'une chaîne de caractères à écrire telle quelle.
- le second est le texte à écrire dans la fenêtre pour informer l'utilisateur de ce qu'on attend de lui, ici la chaîne de caractères**"Donnez un premier nombre :"**. Encore une fois ce texte sera écrit tel quel.
- Le troisième est facultatif, s'il est présent c'est le nombre qui sera proposé par défaut à l'utilisateur.

message est une procédure d'Expresso qui ouvre une fenêtre et affiche le message qui lui est passé en argument. Ici le message est **"Le pgcd de " + a + " et de " + b + " est " + c**. Une petite explication est de rigueur. Les parties entre guillemets seront affichées telles quelles, le + indique ici la concaténation des chaînes de caractères et les variables seront remplacées par leurs valeurs. Donc en supposant que a vaut 34, b vaut 71 et c vaut 1 on obtiendra l'affichage de «**Le pgcd de 34 et de 71 est 1»**

Donc, pour résumer, notre procédure demande un entier, en proposant 34, puis place le résultat obtenu dans la variable **a**. Ensuite elle redemande un entier en proposant 71 et place le résultat dans la variable b. Elle utilise ensuite la procédure pgcd écrite plus haut pour calculer de pgcd de **a** et **b** et place le résultat dans la variable c. Finalement elle affiche un message donnant le résultat en utilisant **a**, **b** et **c**.

Étape 3: création d'un menu

Nous allons créer un menu nommé Programme pgcd et dans ce menu nous allons inclure deux items: un premier nommé « **Faire un calcul »** et un second nommé « **Quitter »**. L'item de menu « **Faire un calcul »** devra activer la procédure « **faireCalcul »** écrite plus haut et l'item « **Quitter »** nous permettra, comme son nom l'indique, de quitter le programme, ce qui ests d'ailleurs déjà possible en cliquant sur la case de fermeture de la fenêtre de tous les programmes « **Expresso »**.

3.1 Faire apparaître le menu

Rien n'est plus simple que faire apparaître un menu dans « **Expresso** ». Il faut aller dans la section « **Paramétres de l'interface** » du fichier « **Expresso** », localiser les variables dont le nom commence par **public static String[] nomsMenus** suivi d'un chiffre, du signe = et d'une paire d'accolades. Le chiffre indique le numéro du menu et on peut avoir jusqu'a huit menus comprenant chacun un maximum de huit items. On crée le menu en plaçant des chaînes de caractères entre les accolades, la première chaîne sera le nom du menu et les autres les noms des items de menu. Dans notre cas on écrira donc :

```
public static String[] nomsMenus1 = {"Programme pgcd", "Faire un calcul", "Quitter"};
```

Si on compile le fichier « Expresso » puis qu'on l'exécute on obtient une fenêtre avec le menu « **Programme pgcd »** et les deux items de menu « **Faire un calcul »** et **« Quitter »**. évidemment ces menus n'ont aucun effet, nous n'avons pas encore programmé leurs actions. Allons y.

3.2 Associer des actions aux items de menu

Pour associer des actions aux différents items de menu allons dans la cinquième section « **Les actions des menus** » du fichier « **Expresso** ». Rappelons nous ce que nous voulons: la sélection de l'item de

menu « **Faire un calcul** » doit activer la procédure « **faireCalcul()** ». L'item « **Faire un calcul** » étant l'item 1 du menu 1 il lui est associé une procédure nommée « **actionMenu1Item1** » qui sera activée automatiquement lorsque cet item de menu sera sélectionné. Il nous suffit de placer dans cette procédure les actions que nous désirons voir exécutées lors de la sélection de cet item. Donc ici

```
public void actionMenu1Item1() {
    faireCalcul();
}
```

Pour l'item de menu « **Quitter** » nous faisons la même chose avec la procédure « **actionMenu1Item2 »** puisque nous sommes toujours dans le menu 1 mais à l'item 2. Nous utilisons ici la procédure « **quitter()** », qui est une procédure d'**Expresso** qui sert à quitter un programme.

```
public void actionMenu1Item2() {
    quitter();
}
```

Si nous compilons et exécutons, dans la fenêtre obtenue le choix de l'item de menu « **Faire un calcul** » devrait faire tout ce que nous avions prévu, demander deux entiers et donner le résultat.

Étape 4: création de boutons

La création de boutons et leur association avec des actions fonctionne de façon analogue à la création des menus. Nous allons créer deux boutons « **Faire un calcul** » et « **Quitter** » qui vont faire la même chose que les items de menu de même noms.

4.1Faire apparaître les boutons

Pour créer les boutons nous allons dans la première section « **Paramètres de l'interface** » et localisons les variables dont le nom commence par

public static String[] nomsBoutonsLigne suivi d'un chiffre, du signe = et d'une paire d'accolade.

On peut répartir nos boutons sur deux lignes et le chiffre indique le numéro de ligne. Les noms des boutons vont entre les accolades sous forme de chaînes de caractères. Ici on écrira donc

public static String[] nomsBoutonsLigne1 = {“Faire un calcul”, “Quitter”}

La séquence compilation, exécution montre qu'on a bien créé les boutons, mais les cliquer montre aussi qu'ils sont inactifs.

4.2 Associer des actions aux boutons

Pour associer des actions aux boutons il faut d'abord comprendre comment ils sont numérotés. Les boutons de la ligne 1 sont numérotés de gauche à droite à partir de 1 (1, 2, .., n) puis les boutons de la ligne 2 sont numérotés à la suite de gauche à droite (n+1, n+2,...). Pour leur associer des actions nous y référons par leur numéro, à travers des méthodes **actionBouton1**, **actionBouton2**, etc, qui sont activées automatiquement lors d'un clic sur le bouton concerné.

Ici nous avons deux boutons, « **Faire un calcul** » est le numéro 1 et « **Quitter** » est le numéro 2 nous écrirons donc dans la quatrième section « **Les actions des boutons** »:

```
public void actionBouton1(){
   faireCalcul();
}
```

pour le premier et

```
public void actionBouton2(){
```

```
        quitter();
    }
```
pour le second.

Compilation, exécution montre que nos boutons sont maintenant opérationnels.

Étape 5: écriture dans la zone de texte

Nous allons maintenant voir comment on peut écrire dans la zone de texte. Dans le cadre de notre projet nous allons faire écrire les différentes divisions impliquées dans le calcul d'un pgcd. L'espace disponible dans la fenêtre ouverte par « **Expresso** » est partagé entre la zone graphique et la zone de texte, nous allons commencer par régler l'espace occupé par la zone de texte. Il faut d'abord s'assurer que la zone texte est présente pour ça nous devons aller dans la première section « **Paramètres de l'interface** », trouver la variable **zoneTexte** et, si nécessaire modifier sa valeur pour qu'elle devienne **true**. Ce qui nous donne la ligne suivante :

```
public static boolean zoneTexte = true;
```

Ici nous n'utilisons pas la zone graphique donc, en principe la zone de texte peut occuper 100% de l'espace disponible. En pratique il faut laisser de l'espace pour les boutons. Nous allons donc donner à la zone de texte 85% de l'espace disponible. Ce travail doit être fait dès l'ouverture du programme, nous allons donc placer les instructions nécessaire dans la procédure « **initialisation()** » qui se trouve dans la deuxième section « **Initialisation »** du fichier Expresso. Profitons en pour fixer également la taille du texte à 18 points.

```
public void initialisation(){
    fixerProportionZoneTexte(0.85);
    tailleTexte(18);
}
```

La signification de chaque instruction est assez claire pour que nous ne nous étendions pas plus sur ce point.

Pour faire écrire les différentes divisions dans cette zone de texte nous allons maintenant modifier la procédure **pgcd(int u, int v)** en y ajoutant des instructions d'écriture. Voici la procédure modifiée avec les nouvelles instructions en caractères gras:

```
public int pgcd(int u, int v){
    int a = u;
    int b = v;
    int c;
    int d;
    videTexte();
    ecrisRC("Algorithme d'Euclide");
    while (b>0) {
        c= mod(a,b);
        d = divEnt (a,b);
        ecrisRC (a + " = " + b + "*" + d + " + " + c);
        a=b;
        b=c;
```

```
    }
    ecris("pgcd(" + u + "," + v + ") = " + a);
    return a;
}
```

La procédure comprend donc quatre nouvelles instructions. Expliquons leur rôle:
videTexte() : efface tout ce qui se trouve actuellement dans la zone de texte.

ecrisRC("Algorithme d'Euclide") : écris la chaîne de caractères « Algorithme d'Euclide » et place le curseur d'écriture sur la ligne suivante (RC pour « retour de chariot »).

Notons que ces deux dernières instructions, se trouvant à l'extérieur de la boucle « while » sont exécutées une seule fois. Nous rencontrons maintenant une instruction qui se trouve dans la boucle et sera donc exécutée autant de fois qu'il y aura de divisions:

ecrisRC (a + " = " + b + "*" + d + " + " + c) : cette instruction écris sur une seule ligne
a : valeur courante de la variable **a** (le dividende) suivi de
" = " : espace, signe =, espace suivis de
b : valeur courante de la variable **b** (le diviseur) suivi de
"*" : signe de multiplication suivi de
d : valeur courante de la variable **d** (le quotient) suivi de
" + " : espace, signe +, espace suivis de
c : valeur courante de la variable **c** (le reste).
Par exemple si **a** vaut 50 et **b** vaut 15 on aura que **d** vaut 3 et **c** vaut 5 et on écrira
50 = 15*3 + 5
Ce travail sera fait pour chaque division avec un retour à la ligne entre chacune.

La dernière instruction n'est pas dans la boucle et ne sera donc exécutée qu'une fois.
ecris("pgcd(" + u + "," + v + ") = " + a) : écrit que le pgcd des deux nombres donnés initialement (u et v) est égal à la dernière valeur de a (puisque nous avons déjà vu que cette valeur est le pgcd). Noter que cette instruction écrit son argument mais ne va pas à la ligne ensuite (différence entre **ecris()** et **ecrisRC()**)

Étape 6: création de glissières

6.1Faire apparaître deux glissières qui fournissent les nombres dont on veut le pgcd

Expresso nous donne la possibilité d'insérer jusqu'à 12 glissières dans notre interface utilisateur. Ces glissières peuvent être réparties sur deux lignes à raison d'un maximum de 6 par ligne. Pour créer ces glissières nous disposons de deux procédures **ajouterGlissiereLigne1** et **ajouterGlissiereLigne2**.

Dans notre cas nous voulons créer deux glissières qui donneront les valeur des deux entiers dont on veut le pgcd et disons que nous allons laisser varier les valeurs de ces entiers entre 1 et 200. Pour le réaliser nous écrirons

```
ajouterGlissiereLigne1("a",1 ,200 ,1 ,0);
ajouterGlissiereLigne1("b",1 ,200 ,1 ,0);
```

Ces deux instructions vont créer deux glissières sur la ligne 1. la signification des paramètres est la suivante:

- le premier est la légende qui sera affichées avec la glissière, c'est une chaîne de caractère. Ici « **a** » avec la première glissière et « **b** » pour la seconde.

- le second et le troisième sont les valeurs minimale et maximale de la glissière, ce sont des nombres décimaux. Ici 1 et 200.
- le quatrième est la valeur initiale de la glissière, c'est un nombre décimal. Ici 1.
- le cinquième est le nombre de décimales désirées, c'est un nombre entier. Ici, puisque nous ne voulons que des valeurs entières nous mettons 0 décimales.

Les deux instructions ci-dessus créent les glissières mais ne les intègrent pas à l'interface. Cette intégration se fait en les invoquant dans la procédure **ajoutDeGlissieres()** que l'on trouve dans la première section « **Paramètres de l'interface** ». Nous complétons donc la création/insertion des glissières en écrivant dans cette section:

```
public static void ajoutDeGlissieres(){
   ajouterGlissiereLigne1("a",1,200,1,0);
   ajouterGlissiereLigne1("b",1,200,1,0);
}
```

6.2 Associer des actions à ces deux glissières

À l'intérieur des programmes chaque glissière est identifiée par un numéro. Les glissières de la ligne 1 sont numérotées à partir de 1 dans l'ordre ou elles ont étés insérées (l'ordre ou elles apparaissent dans la procédure **ajoutDeGlissieres()**, qui correspond aussi à l'ordre ou elles apparaissent de gauche à droite dans l'interface), la numérotation continue ensuite avec les glissières de la ligne 2, également dans l'ordre d'insertion.

Pour associer des actions aux glissières nous disposons pour chacune d'une procédure prédéfinie que nous aurons à compléter et qui sera exécutée automatiquement chaque fois que la glissière changera de valeur. Ces procédures se nomment

- **actionGlissiere1** pour la glissière numéro 1
- **actionGlissiere2** pour la glissière numéro 2 et ainsi de suite.

Puisque les valeurs de nos glissières représentent les nombres dont nous voulons le pgcd, nous voulons que lors d'un changement de valeur de l'une des deux glissières le nouveau pgcd soit calculé et affiché. C'est justement le travail fait par la procédure **pgcd()**.

Programmons donc, dans la sixième section « **actions des glissieres** », l'action de la première glissière, nous expliquerons les quelques détails restant après:

```
public void actionGlissiere1(double d) {
   pgcd(valEnt(d),valEnt(valeurGlissiere(2)));
}
```

La procédure **actionGlissiere1()** reçoit un argument **d** qui représente la valeur de la glissière. Cet argument est un nombre décimal, nous devons donc le transformer en entier avant de le passer à la procédure **pgcd()**, c'est ce qui explique que le premier argument de **pgcd()** soit **valEnt(d)** plutôt que **d**. Il faut ensuite passer à pgcd() un second argument représentant la valeur de la glissière 2 (valeurGlissiere(2)) qui est aussi un nombre décimal et qu'on transforme en entier avec **valEnt()**. Finalement le second argument est **valEnt(valeurGlissiere(2))**. Ce qui nous donne bien

pgcd(valEnt(d),valEnt(valeurGlissiere(2)));

pour calculer et afficher le pgcd des valeurs des deux glissières.

Il faut faire le même travail avec la glissière numéro 2, donc avec la procédure **actionGlissiere2()** en inversant évidemment les rôles des deux glissières

```
public void actionGlissiere2(double d) {
   pgcd(valEnt(valeurGlissiere(1)),valEnt(d));
}
```

6.3Faire apparaître une glissière qui ajuste la taille du texte dans la zone de texte

Juste pour le plaisir de pratiquer nos nouvelle connaissances ajoutons une glissière qui contrôle la taille du texte dans la zone de texte.

Nous allons ajouter cette nouvelle glissière sur la ligne 1, avec la légende « Taille du texte », avec 9 points comme taille minimale du texte et 36 points comme taille maximale, avec 12 points comme valeur initiale et 0 décimales. L'instruction pour créer la glissière sera

```
ajouterGlissiereLigne1("Taille du texte" ,9 ,36 ,12 ,0);
```

Pour insérer la glissière dans l'interface utilisateur nous devons ajouter cette instruction dans la procédure ajoutDeGlissieres() ce qui donne

```
public static void ajoutDeGlissieres(){
   ajouterGlissiereLigne1("a",1,200,1,0);
   ajouterGlissiereLigne1("b",1,200,1,0);
   ajouterGlissiereLigne1("Taille du texte" ,9 ,36 ,12 ,0);
}
```

De plus cette nouvelle glissière portera le numéro 3.

6.4 Associer une action à la glissière qui contrôle la taille du texte

La procédure qui contrôle la taille du texte est **tailleTexte(int t)** ou t est un entier représentant la taille désirée du texte en nombre de points.

En fonction de ce que nous avons déjà vu pour associer des actions aux glissière, il est très simple d'associer l'action désirée à la glissière numéro 3 :

```
public void actionGlissiere3(double d) {
   tailleTexte(valEnt(d));
}
```

Étape 7: un peu de finition

Pour terminer nous pouvons ajuster quelques autres paramètres de l'interface, par exemple ajoutons le titre « Algorithme d'Euclide » à la fenêtre principale. Pour cela allons dans la première section « **Paramètres de l'interface** », nous y trouvons la variable **titreFenêtre** que nous modifions de la façon suivante:

```
public static String titreFenetre = "Algorithme d'Euclide";
```

De la même façon nous pouvons régler la taille de la fenêtre principale à l'ouverture du programme en modifiant les variables **baseFenetre** et **hauteurFenetre**. Par exemple pour avoir une fenêtre de 800 pixels de large et 600 de haut on modifiera ces variables de la façon suivante :

```
public static int baseFenetre = 800;
public static int hauteurFenetre = 600;
```

Étape 8: un regard admiratif sur l'ensemble de notre travail

Notre programme est terminé, c'est le moment de jeter un regard d'ensemble sur notre travail et de revenir sur les points importants. Voici donc le programme terminé avec quelques commentaires:

```
/**************************************************************************
*
*              NE PAS MODIFIER CETTE SECTION
*
**************************************************************************/
import java.awt.*;
import Listes.*;

public class Expresso extends FenetreTortue{

/**************************************************************************
*
*                  SECTION MODIFIABLE
*
**************************************************************************/

/********************** Parametres de l'interface ************************/

public static int baseFenetre = 800; // Largeur de la fenêtre
public static int hauteurFenetre = 600; // Hauteur de la fenêtre
public static String titreFenetre = "Algorithme d'Euclide";; // Titre de la fenêtre

// Le texte
public static boolean zoneTexte = true; // Indique si on désire ou non une zone de texte

// Les boutons
public static String[] nomsBoutonsLigne1 = {"Faire un calcul", "Quitter"}; // Noms des boutons de la
ligne 1
public static String[] nomsBoutonsLigne2 = {}; // Noms des boutons de la ligne 2

// Les menus
public static String[] nomsMenus1 = {"Programme pgcd", "Faire un calcul", "Quitter"};
public static String[] nomsMenus2 = {}; // Laissez vide si vous ne désirez pas de menu
public static String[] nomsMenus3 = {};
public static String[] nomsMenus4 = {};
public static String[] nomsMenus5 = {};
public static String[] nomsMenus6 = {};
public static String[] nomsMenus7 = {};
public static String[] nomsMenus8 = {};

public static void ajoutDeGlissieres(){
        ajouterGlissiereLigne1("a",1,200,1,0);
        ajouterGlissiereLigne1("b",1,200,1,0);
        ajouterGlissiereLigne1("Taille du texte",9,36,12,0);
}


/****************** Initialisation ********************************/

public void initialisation(){
        // Placer ici les actions a realiser a l'ouverture
        fixerProportionZoneTexte(0.85);
```

Voir Étape 7

Voir Étape 5

Voir Étape 4 section 4.1

Voir Étape 3 section 3.1

Voir Étape 6 sections 6.1 et 6.3

Voir Étape 5

```
        tailleTexte(18);
}
/****************** Placer vos procedures ici ***************************/


public int pgcd(int u, int v){
        int a = u;
        int b = v;
        int c;
        int d;
        videTexte(); // 2
        ecrisRC("Algorithme d'Euclide"); // 2
        while(b > 0){
        c = mod(a, b);
        d = divEnt(a, b);
        ecrisRC (a + " = " + b + "*" + d + " + " + c); // 2
        a = b;
        b = c;
        }
        ecris("pgcd(" + u + "," + v + ") = " + a); // 2
        return a;
}
public void faireCalcul(){
        int a = demanderEntier("Calcul du pgcd (#1/2)", "Donnez un premier nombre :", 34);
        int b = demanderEntier("Calcul du pgcd (#2/2)", "Donnez un second nombre :", 71);
        int c = pgcd(a, b);
        message("Le pgcd de " + a + " et de " + b + " est " + c);
}
/**************  Les actions des boutons  ***********************************/
public void actionBouton1(){
        faireCalcul();
}
public void actionBouton2(){
        quitter();
}


/**************  Les actions des menus  ****************************/
public void actionMenu1Item1() {
        faireCalcul();
}
public void actionMenu1Item2() {
        quitter();
```

Voir Étapes 1 et 2 et étape 5 pour ce qui est marqué // 2

Voir Étape 3 section 3.2

Voir Étape 4 section 4.2

```
}
/*************** Les actions des glissieres ********************************/
public void actionGlissiere1(double d) {
        pgcd(valEnt(d),valEnt(valeurGlissiere(2)));
}
public void actionGlissiere2(double d) {
        pgcd(valEnt(valeurGlissiere(1)),valEnt(d));
}
public void actionGlissiere3(double d) {
        tailleTexte(valEnt(d));
}
/*************** Les actions de la souris ******************************/
public void clicSouris(double x, double y){}
public void debutGlisser(double x, double y){}
public void finGlisser(double x, double y){}
public void glisserEnCours(double x, double y){}
/*********************************************************************
*
*               NE PAS MODIFIER CETTE SECTION
*
*********************************************************************/
public static String[][] Menus = {nomsMenus1, nomsMenus2, nomsMenus3, nomsMenus4,
nomsMenus5, nomsMenus6, nomsMenus7, nomsMenus8};
public Expresso(int l, int h, String titre, String[] nomsBoutons1, String[] nomsBoutons2, String[][]
Menus, boolean avecTexte){
        super(l, h, titre, nomsBoutons1, nomsBoutons2, Menus, avecTexte);
}
public Expresso() {
        this(baseFenetre, hauteurFenetre, titreFenetre, nomsBoutonsLigne1, nomsBoutonsLigne2,
Menus, zoneTexte);
}
public static void executer(boolean applet){
        initGlissieres();
        ajoutDeGlissieres();
        Expresso maFenetre = new Expresso();
        faireApplet(applet);
        maFenetre.toFront();
        maFenetre.initialisation();
}
public static void main(String[] args){
```

Voir Étape 6 section 6.2

Voir Étape 6 section 6.4

```
        executer(false);}
}
```

Étape 8: comment fabriquer une application autonome *(c'est à dire indépendante de BlueJ)*

Lorsque votre programme fonctionne à votre goût vous pouvez en faire une application autonome. Pour cela, dans **BlueJ**, il faut sélectionner dans le menu « **Projet** » l'item « **Exporter (jar)** ». Une première fenêtre s'ouvre dans laquelle, dans la section « **Inclure les bibliothèques utilisateur** » il faut cocher la case « **Expresso.jar** », dans le menu déroulant il faut choisir « **Expresso** » puis cliquer « **continuer** ». Dans la fenêtre de sauvegarde qui s'ouvre ensuite choisir un emplacement et donner un nom (par exemple *pgcd*) puis cliquer sur « **Créer** ».

Comme résultat on obtient un dossier portant le nom choisi (dans notre exemple *pgcd*) et dans ce dossier deux fichier « **jar** »: **Expresso.jar** et ***pgcd*.jar**.

C'est fait, un double clic sur ***pgcd*.jar** devrait démarrer votre application. Et cela fonctionnera sur tout ordinateur ou Java est installé.

Étape 9: comment fabriquer un applet à partir de notre programme

À partir de votre programme dans BlueJ choisir « **Nouvelle classe** », dans la fenêtre de choix qui s'ouvre sélectionner « **AppletExpresso** », donner le nom « **AppletExpresso** » et cliquer « **OK** ». Un nouveau fichier nommé « **AppletExpresso** » apparaît dans la fenêtre de **BlueJ**.

Compiler le tout, « **Expresso** » et « **AppletExpresso** ».
Dans le menu « **Projet** » choisir « **Exporter (jar)** » et procéder comme pour une application autonome.

On obtient encore un dossier portant le nom choisi (dans notre exemple *pgcd*) et dans ce dossier deux fichier « **jar** »: **Expresso.jar** et ***pgcd*.jar**. Ajouter au dossier la page html « **AppletExpresso.html** » téléchargée avec **Expresso** et changer le nom de ***pgcd*.jar** en **AppletExpresso.jar**. C'est fini. Lorsque les fichiers « **Expresso.jar** », « **AppletExpresso.jar** » et « **AppletExpresso.html** » sont tous les trois au même niveau dans un site Web tout lien pointant vers « **AppletExpresso.html** » ouvrira cette page qui elle même ouvrira automatiquement l'application contenue dans « **AppletExpresso.jar** »

Exemple 2 : Tracé de polygones emboîtés

Dans ce deuxième exemple nous allons nous donner pour objectif d'écrire un programme qui permet de tracer une série de polygones emboîtés. Le second polygone sera obtenu en joignant les milieux des côtés du premier, le troisième en joignant les milieux des côtés du second et ainsi de suite.

Dans une première version nous mettrons en pratique ce que nous avons déjà vu dans le premier exemple en contrôlant avec des glissières le nombre de côtés des polygones, le rayon du cercle circonscrit du polygone initial et le nombre de polygones.

Dans une seconde version nous verrons comment gérer les événements liés à la souris en déplaçant les polygones en fonction des clics et des glisser (« drag ») de la souris.

Finalement dans une troisième version nous introduirons de la couleur et apprendrons comment remplir nos polygones. Nous introduirons un bouton (ce que nous savons déjà faire: Exemple 1 étape 4) qui permettra à l'utilisateur de décider si les polygones doivent être remplis ou non et nous verrons comment changer le nom de ce bouton pour qu'il reflète toujours l'état du programme.

Étape 1: calcul des coordonnées des polygones.

Même si dans la suite, pour alléger le texte, nous omettrons le qualificatif régulier il doit être clair que nous nous intéressons seulement à des polygones réguliers. Pour commencer calculons les coordonnées des sommets d' un polygone à **n** côtés, centré à l'origine et dont le cercle circonscrit est de rayon **r**.
Supposons de plus qu'un sommet se trouve sur l'axe des abscisses avec coordonnées (r, 0)(Voir Figure 1). Appelons ce sommet P_0 et appelons les autres P_1, P_2, .., P_{n-1} en tournant dans le sens anti-horaire (remarquons que si on continue jusqu'à P_n on a $P_n = P_0$). Si nous appelons C le centre il est clair qu'un angle au centre orienté CP_0P_i mesure $2i\frac{\pi}{n}$.

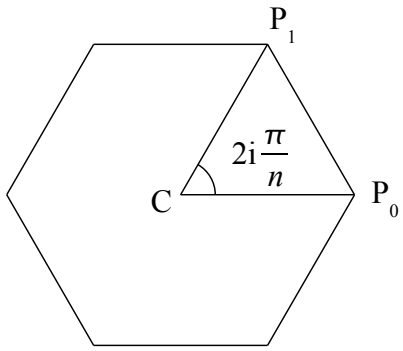


Figure 1

Les coordonnées du sommet Pi seront donc

$$(rcos(2i\frac{\pi}{n}), rsin(2i\frac{\pi}{n}))$$

Coordonnées sont valables en particulier pour i = 0 (c'est à dire pour P_0) et aussi pour i = n (c'est à dire encore pour $P_n = P_0$).

Maintenant si le sommet P_0 n'est pas sur l'axe des X mais est tel que l'angle orienté CXP_0 mesure **a** radians (voir Figure 2), les angles orientés CXP_i mesurerons $2i\frac{\pi}{n}+a$ et les coordonnées du sommet P_i seront

$$(r\cos(2i\frac{\pi}{n}+a), r\sin(2i\frac{\pi}{n}+a))$$.

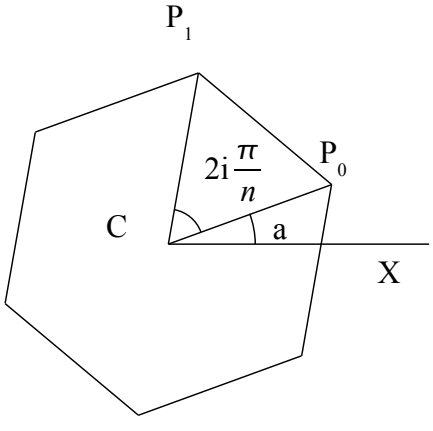


Figure 2

Finalement si le polygone n'est pas centré à l'origine mais en

un point (x, y), on obtient les coordonnées en translatant le polygone centré à l'origine par un vecteur (x, y) et les coordonnées deviennent

$$(x+r\cos(2\mathrm{i}\frac{\pi}{n}+a),\ y+r\sin(2\mathrm{i}\frac{\pi}{n}+a))$$

Maintenant un polygone G étant donné, regardons comment trouver les sommets du polygone G' formé en joignant les milieux des côtés. D'abord il est clair que le rayon r' du cercle circonscrit à G' est égal à l'apothème de G donc $r'=r\cos(\frac{\pi}{n})$. Pour passer de G à G' il suffit donc d'appliquer à G une homothétie centrée au centre de G et de rapport $\cos(\frac{\pi}{n})$ et ensuite de faire tourner de $\frac{\pi}{n}$ l'image obtenue autour du centre de G.

Les sommets P'_i de G' seront donc donnés par les formules

$$r'=r\cos(\frac{\pi}{n})$$

$$(x+r'\cos(a+2\mathrm{i}\frac{\pi}{n}+\frac{pi}{n}),\ y+r'\sin(a+2\mathrm{i}\frac{\pi}{n}+\frac{pi}{n}))$$

Étape 2: les procédures de traçage des polygones.

Comme dans l'exemple précédent, créons, dans **BlueJ**, un projet nommé par exemple « Polygones » et dans ce projet créons une nouvelle classe de type « Expresso » que nous nommons aussi « Expresso » (Voir Introduction). Ceci étant fait ouvrons le fichier « Expresso » et dans la troisième section « **Placer vos procédures ici** » écrivons notre programme:

D'abord une ligne pour créer trois variables de type double (nombres décimaux) qui vont représenter les coordonnées du centre commun de tous les polygones (centreX et centreY) et l'angle de rotation des polygones par rapport à leur position initiale (angleGlisser). Ces trois variables sont initialisées à zéro.

double centreX=0, centreY=0, angleGlisser = 0;

Ensuite la méthode de traçage d'un polygone qui reçoit 5 paramètres:
n le nombre de côtés du polygone, **r** le rayon du cercle circonscrit, **x** et **y** les coordonnées du centre et **angle** l'angle de rotation par rapport à l'axe des x.

Cette méthode pourrait s'écrire en pseudo code:

pour j allant de 0 à n-1
tracer un segment
du point Pj de coordonnées

$$(x+r\cos(2\mathrm{j}\frac{\pi}{n}+angle),\ y+r\sin(2\mathrm{j}\frac{\pi}{n}+angle))$$

au point Pj+1 de coordonnées

$$(x+r\cos(2(j+1)\frac{\pi}{n}+angle),\ y+r\sin(2(j+1)\frac{\pi}{n}+angle))$$

Ce qui, en langage « Expresso » se traduit simplement par :

```
public void polygone(int n, double r, double x, double y, double angle){
        for(int j = 0; j < n; j++){
                // On joint Pj à Pj+1 par un segment.
                segment(x + r * cos(j*2*Pi/n + angle), y + r * sin(j*2*Pi/n + angle),
                x + r * cos((j+1)*2*Pi/n + angle), y + r * sin((j+1)*2*Pi/n + angle));
        }
}
```

Remarquons qu'on trace un segment entre les sommets P_j et P_{j+1} pour j allant de 0 à n-1. On trace donc de P_0 à P_1, puis de P_1 à P_2, etc et on termine en traçant de P_{n-1} à $P_n = P_0$, ce qui clos le polygone.

Finalement la méthode de traçage des polygones emboîtés, qui reçoit les même paramètres que la méthode « polygone » ci-dessus avec en plus le paramètre **iter** qui représente le nombre de polygones emboîtés.

```
public void polygonesEmboites(int n, int iter, double r, double x, double y, double
angle) {
        for (int i = 1; i<=iter; i++) {
                polygone(n, r, x, y, angle);   // On trace un polygone
                 r = r*cos(Pi/(n));            // Rayon du prochain polygone
                angle = angle + Pi/(n);        // Angle du prochain polygone
        }
}
```

Étape 3: ajout de glissières pour contrôler les paramètres.

Nous allons ajouter trois glissières qui vont contrôler respectivement le nombre de côtés des polygones, le rayon du cercle circonscrit du polygone initial et le nombre de polygones emboîtés.

Puisque nous avons déjà vu comment faire ce travail dans l'exemple 1, nous passons rapidement sur cette partie.

3.1 Intégrer les glissières à l'interface

Nous allons d'abord dans la première section « **Paramètres de l'interface** » et nous complétons la procédure **ajoutDeGlissières** de la façon ci-dessous:

```
public static void ajoutDeGlissieres(){
        ajouterGlissiereLigne1("Nombre de côtés", 3, 24, 6, 0);
        ajouterGlissiereLigne1("Rayon du cercle circonscrit", 10, 500, 100, 0);
        ajouterGlissiereLigne2("Nombre d'itérations", 1, 50, 1, 0);
}
```

Nous avons donc placé nos deux premières glissières sur une même ligne et la troisième sur une deuxième ligne. Si on se reporte à l'exemple 1 section 6.1 pour l'interprétation des paramètres on voit que :

- le nombre de côtés peut varier de 3 à 24 avec une valeur initiale de 6 et possède 0 décimales.
- le rayon du cercle circonscrit peut varier de 10 à 500 avec valeur initiale de 100 et possède 0 décimales
- le nombre de polygones emboîtés varie de 1 à 50 avec valeur initiale de 1 et 0 décimales

3.2 Associer des actions aux glissières

Pour associer des actions aux glissières nous allons maintenant ajouter un peu de code dans la sixième section « **actions des glissières** ».
Comme l'action des trois glissières sera la même, nous écrivons une seule procédure qui servira pour les trois. Cette procédure efface le graphique et le retrace en tenant compte des valeurs actuelles des trois glissières et du centre des polygones défini par les variables **centreX** et **centreY**.

```
public void actionGlissiere(double a){
        videGraphique();
        polygonesEmboites(valEnt(valeurGlissiere(1)), valEnt(valeurGlissiere(3)),
                                valEnt(valeurGlissiere(2)), centreX, centreY, a);
}
```

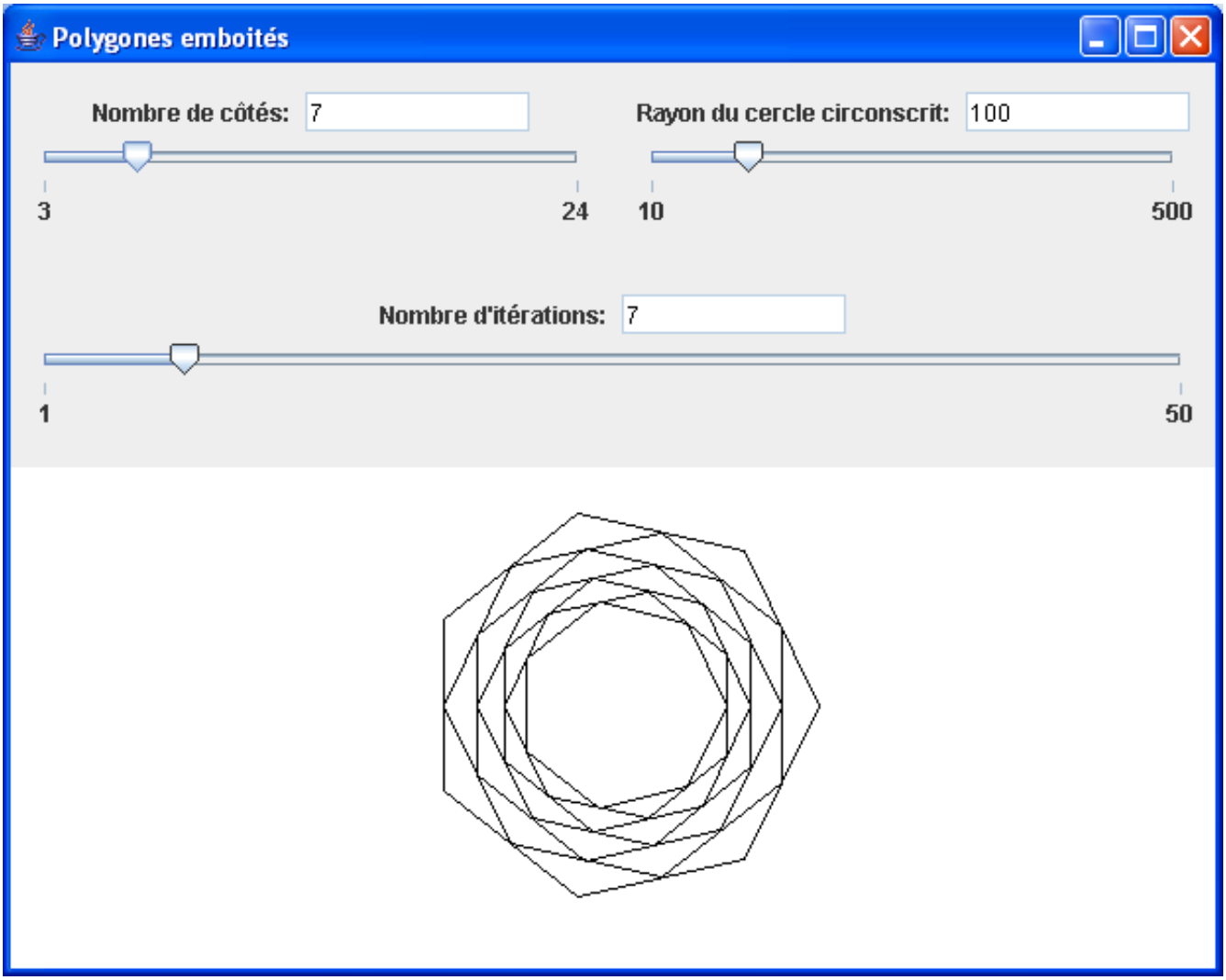
Nous utilisons maintenant cette procédure pour réagir à la modification de chaque glissière.

```
public void actionGlissiere1(double d){
        actionGlissiere(0);
}

public void actionGlissiere2(double d){
        actionGlissiere(0);
}


public void actionGlissiere3(double d){
        actionGlissiere(0);
}
```

La première version de notre programme est terminée et la séquence compilation/exécution (voir « Comment exécuter vos programme dans BlueJ » page 9) nous permet de voir le résultat de notre travail:



Étape 4: gestion des actions de la souris.

Pour réagir aux événements générés par la souris nous allons maintenant ajouter du code dans la septième section « **gestion des événements souris** » en utilisant les quatre procédures prédéfinies **clicSouris**, **debutGlisser**, **glisserEnCours** et **finGlisser**.

Cas d'un clic

Dans le cas d'un clic nous retraçons les polygones en les centrant au point cliqué. Pour cela nous utilisons la procédure **clicSouris** qui est invoquée par le système lorsqu'on clique dans la fenêtre graphique. Notons que cette procédure possède deux arguments x et y qui lors d'un clic prennent pour valeurs les coordonnées du point cliqué.

```
public void clicSouris(double x, double y){
        centreX=x;
        centreY=y;              // On place le centre au point cliqué
        actionGlissiere(0);     // On retrace
}
```

Cas d'un glisser

On va retracer les polygones en suivant la souris et en les faisant tourner pendant la durée du glisser.

Au début

Lors du clic initial on recentre les polygones sur le point cliqué avec un angle nul. Pour cela on utilise la procédure **debutGlisser** qui est invoquée par le système lorsque, après avoir enfoncé le bouton de la souris, on commence à la déplacer. Notons que cette procédure possède deux arguments x et y qui prennent pour valeurs les coordonnées du point ou se trouve la souris lorsqu'on enfonce le bouton.

```
public void debutGlisser(double x, double y){
        centreX=x;
        centreY=y;                      // On place le centre au point cliqué
        angleGlisser = 0;               // On met l'angle à 0
        actionGlissiere(angleGlisser);  // On retrace
}
```

Pendant

Pendant le glisser on retrace constamment les polygones en les centrant sur la souris (coordonnées x et y) et pour chaque traçage on augmente l'angle, ce qui a pour effet de faire tourner les polygones. Pour ce faire on utilise la procédure **glisserEnCours** qui est invoquée par le système lorsqu'on déplace la souris en tenant le bouton enfoncé. Cette procédure possède deux arguments x et y qui, à tout moment, prennent pour valeurs les coordonnées du point où se trouve la souris.

```
public void glisserEnCours(double x, double y){
        centreX=x;
        centreY=y;                      // On place le centre au point courant
        angleGlisser = angleGlisser+ 5; // On incrémente l'angle
        actionGlissiere(angleGlisser);  // On retrace avec le nouvel angle
}
```

À la fin

À la fin du glisser on retrace les polygones avec un angle nul. Pour cela on utilise la procédure **finGlisser** qui est invoquée par le système lorsque, après un glisser, on relache le bouton de la souris. Cette procédure possède deux arguments x et y qui prennent pour valeurs les coordonnées du point où se trouve la souris lorsqu'on relâche le bouton.

```
public void finGlisser(double x, double y){
        centreX=x;
        centreY=y;                      // On place le centre au point courant
        actionGlissiere(0);             // On retrace
}
```

Nous avons complété la seconde version du travail. Nous disposons maintenant d'un programme qui trace les polygones mais réagit aussi aux clics et aux déplacements de la souris en recentrant et en déplaçant nos polygones. On peut tester le programme en compilant puis en exécutant (voir page 9).

Étape 5: ajout de couleurs de remplissage et de boutons de contrôle.

Nous désirons maintenant donner à l'utilisateur de notre programme la possibilité de choisir entre tracer des polygones remplis avec des couleurs ou bien, comme le programme le fait actuellement, tracer seulement les contours des polygones.

Nous commençons en ajoutant une variable de type tableau (Array) qui va contenir l'ensemble des couleurs que nous désirons utiliser. Nous plaçons cette variable avec les autres (**centreX**, **centreY**, etc) au début de la troisième section « **Placer vos procédures ici** ».

Color[] couleurs = {rouge, jaune, vert, bleu, gris, noir, blanc, cyan, magenta, orange, rose};

Color[] couleurs : déclare une variable nommée **couleurs.** Dans **Color[]** les crochets **[]** indiquent qu'il s'agit d'un tableau et le mot **Color** indique que ce tableau va contenir des objets de type **Color**, c'est à dire des couleurs. Nous plaçons dans ce tableau 11 couleurs prédéfinies (on peut aussi créer des couleurs avec l'instruction **couleurRVB(n1, n2, n3)** où n1, n2, n3 indiquent les quantités de rouge, vert et bleu dans un codage RGB de la couleur).

Notons que dans un tableau les éléments sont numérotés à partir de 0. Donc, dans notre tableau, nos couleurs sont numérotées de 0 à 10 et rouge est la couleur 0, jaune la couleur 1, etc.

Créons à la suite une variable booléenne qui servira à indiquer si les polygones doivent êtres remplis ou non. Nous lui donnons lui une valeur initiale de « **false** » pour que dans l'état initial de programme les polygones ne soient pas remplis:

boolean remplir = false;

Nous allons maintenant créer deux boutons, un bouton « **Quitter** » qui permettra de quitter le programme, ce que nous avons déjà fait dans l'exemple 1 et un bouton « **Remplir/Contours** » qui permettra de choisir entre tracer des polygones pleins ou seulement leurs contours. La particularité de ce second bouton c'est qu'il devra changer de nom suivant l'état du programme, son nom indiquera l'état à venir si on le clique. Donc si les polygones sont actuellement remplis le nom du bouton sera « **Contours** » pour qu'en le cliquant on passe en mode Contours et son nom devra alors devenir « **Remplir** ». Puisque dans l'état initial la variable remplir vaut « **false** » (on est donc en mode **Contours**) le nom initial de notre bouton sera « **Remplir** » (Puisque en le cliquant on passera en mode Remplir!).

5.1 Ajout des boutons

Procédons donc à la création des boutons dans la première section « **Paramètres de l'interface** ». Il suffit d'écrire les noms de nos boutons dans la variable **nomsBoutonsLigne1** (Voir exemple 1):

public static String[] nomsBoutonsLigne1 = {"Remplir", "Quitter"};

Donc le bouton « Remplir » sera le bouton 1 et le bouton « Quitter » sera le bouton 2.

5.2 Les actions des boutons

Pour le bouton « Quitter » il nous suffit de refaire ce que nous avons déjà fait dans l'exemple 1:

```
public void actionBouton2(){
        quitter();
}
```

Pour le bouton « **Remplir/Contours** », un clic doit faire trois choses :

- changer la valeur de la variable « **remplir** », si elle est à vrai (polygones pleins) elle doit passer à faux (mode contours) et vice-versa. Pour cela nous utilisons la négation logique: remplir devient NON remplir, ce qui s'écrit **remplir = ! remplir** dans « **Expresso** ».
- Retracer les polygones en tenant compte de la valeur de la variable « **remplir** »
- Changer le nom du bouton, nom qui doit refléter l'état futur de la variable remplir si le bouton est

cliqué. Donc si « **remplir** » est vrai le nom doit devenir « **Contour** » et inversement si « **remplir** » est faux le nom doit devenir « **Remplir** ».

Pour changer le nom d'un bouton on doit utiliser la méthode **changerNomBouton** avec deux paramètres, d'abord le numéro du bouton et ensuite le nouveau nom sous forme d'une chaîne de caractères. Sachant cela nous sommes prêts à écrire la procédure qui définit l'action associée au bouton « **Remplir/Contours** »:

```
public void actionBouton1(){
        remplir = !remplir;                     // On change le mode de remplissage
        actionGlissiere(0);                     // On retrace
        if (remplir){                           // Si remplir est vrai
        changerNomBouton(1, "Contours");        // Le nom devient « Contours »
}else{                                          // Sinon
        changerNomBouton(1, "Remplir");         // Le nom devient « Remplir »
        }
}
```

Il y a encore un problème à régler, notre méthode de traçage des polygones ne tient pas compte de la variable « **remplir** ». En fait si nous testons notre programme, nous avons bien nos boutons, le bouton « **Remplir/Contours** » change bien de nom si on le clique mais nous n'observons aucun effet sur nos polygones.

Nous devons donc modifier la procédure qui trace les polygones emboîtés pour remplir les polygones lorsque c'est nécessaire. Les modifications vont consister à

- Choisir une couleur de remplissage (méthode **couleurRemplissage**).
- Indiquer, si nécessaire, qu'il faut remplir avant de tracer (méthode **debutRemplir**).
- Indiquer qu'il faut compléter le remplissage après avoir tracé (méthode **finRemplir**).

Voici la procédure modifiée avec les ajouts en caractères gras:

```
public void polygonesEmboites(int n, int iter, double r, double x, double y, double angle) {
        for (int i = 1; i<=iter;i++) {
                couleurRemplissage(couleurs[mod(i,11)]);
                if (remplir){debutRemplir();}
                polygone(n, r, x, y, angle);
                if (remplir){finRemplir();}
                 r = r*cos(Pi/(n));             // Rayon du prochain polygone
                angle = angle + Pi/(n);         // Angle du prochain polygone
        }
}
```

Expliquons brièvement les éléments nouveaux:

mod(i, 11) : cette instruction retourne la valeur de i modulo 11. Puisque nous n'avons que 11 couleurs numérotées de 0 à 10 nous nous servons de cette instruction pour choisir un numéro de couleur sans sortir du domaine des numéros permis.

couleurRemplissage(couleurs[mod(i, 11)]): fixe la couleur de remplissage comme étant la couleur de numéro **mod(i, 11)** dans le tableau **couleurs**.

if (remplir){debutRemplir();}: vérifie la variable remplir et si elle est vrai démarre le remplissage.

if (remplir){finRemplir();}: vérifie la variable remplir et si elle est vrai termine le remplissage.

Étape 6: un peu de finition

Retournons dans la première section « **Paramètres de l'interface** » pour donner un titre à la fenêtre principale et supprimer la zone de texte puisque nous ne l'utilisons pas. Nous ne commenterons pas ce travail puisque nous avons déjà vu comment faire dans l'exemple 1.

public static String titreFenetre = "Polygones emboîtés"; // Le titre

public static boolean zoneTexte = false; // Suppression de la zone de texte

Et voilà notre troisième version est terminée.

Étape 7: un regard attendri sur l'ensemble de notre programme.

```
/**************************************************************************
*
*              NE PAS MODIFIER CETTE SECTION
*
**************************************************************************/
import java.awt.*;
public class Expresso extends FenetreTortue
{
/**************************************************************************
*
*                SECTION MODIFIABLE
*
**************************************************************************/
/********************** Parametres de l'interface ***********************/
public static int baseFenetre = 600;  // Largeur de la fenetre
public static int hauteurFenetre = 500;  // Hauteur de la fenetre
public static String titreFenetre = "Polygones emboîtés");  // Titre de la fenetre
// La zone texte
public static boolean zoneTexte = false; // Indique si on desire ou non une zone de texte
// Les boutons
public static String[] nomsBoutonsLigne1 = {"Remplir", "Quitter"}; // Noms des boutons de la ligne 1
public static String[] nomsBoutonsLigne2 = {}; // Noms des boutons de la ligne 2
/ Les menus
public static String[] nomsMenus1 = {}; // Exemple: nomsMenus1 = {"nomMenu", "nomItem_1", ...};
public static String[] nomsMenus2 = {}; // Laissez vide si vous ne desirez pas de menu
public static String[] nomsMenus3 = {};
```

Voir étapes 5.1 et 6

```
public static String[] nomsMenus4 = {};
public static String[] nomsMenus5 = {};
public static String[] nomsMenus6 = {};
public static String[] nomsMenus7 = {};
public static String[] nomsMenus8 = {};


   public static void ajoutDeGlissieres(){
      ajouterGlissiereLigne1("Nombre de côtés"), 3,24,6,0);
      ajouterGlissiereLigne1("Rayon du cercle circonscrit", 10,500,100,0);
      ajouterGlissiereLigne2("Nombre d'itérations"), 1,50,1,0);
   }
/****************** Initialisation ********************************/
public void initialisation(){
}
/****************** Placer vos procedures ici ***************************/
Color[] couleurs = {rouge, jaune, vert, bleu, gris, noir, blanc, cyan, magenta, orange, rose};
double centreX=0, centreY=0, angleGlisser = 0;
boolean remplir = false;


public void polygone(int n, double r, double x, double y, double angle){
        for(int j = 0; j < n; j++){
                segment(x + r * cos(j*2*Pi/n + angle), y + r * sin(j*2*Pi/n + angle),
                x + r * cos((j+1)*2*Pi/n + angle), y + r * sin((j+1)*2*Pi/n + angle));
         }
}

public void polygonesEmboites(int n, int iter, double r, double x, double y, double angle) {
   for (int i = 1; i<=iter;i++) {
      couleurRemplissage(couleurs[mod(i, 11)]);
      if (remplir){debutRemplir();}
      polygone(n, r, x, y, angle);
      if (remplir){finRemplir();}
      r = r*cos(Pi/(n));
      angle = angle + Pi/(n);
   }
```

Voir étape 3.1

Voir étapes 2 et 5

```
}
/**************  Les actions des boutons  ***********************************/
public void actionBouton1(){
   remplir = !remplir;
   actionGlissiere(0);
   if (remplir){
      changerNomBouton(1, "Contours");
   }else{
      changerNomBouton(1, "Remplir");
   }
}
public void actionBouton2(){
   quitter();
}
/**************  Les actions des menus  ***************************/
public void actionMenu1Item1(){
}
public void actionMenu1Item2(){
}
public void actionMenu1Item3(){
}
/*************** Les actions des glissieres ********************************/
  public void actionGlissiere(double a){
      videGraphique();
      polygonesEmboites(valEnt( valeurGlissiere(1)), valEnt(valeurGlissiere(3)),
                        valEnt(valeurGlissiere(2)), centreX, centreY, a);
   }
  public void actionGlissiere1(double d){
      actionGlissiere(0);
   }
  public void actionGlissiere2(double d){
      actionGlissiere(0);
   }
```

Voir étape 5.2

Voir étape 3.2

```
 public void actionGlissiere3(double d){
      actionGlissiere(0);
   }
/*************** Les actions de la souris ******************************/
   public void clicSouris(double x, double y){
      centreX=x;
      centreY=y;
      actionGlissiere(0);
   }
   public void debutGlisser(double x, double y){
      centreX=x;
      centreY=y;
      angleGlisser = 0;
      actionGlissiere(angleGlisser);
   }
   public void finGlisser(double x, double y){
      centreX=x;
      centreY=y;
     actionGlissiere(0);
   }
   public void glisserEnCours(double x, double y){
      centreX=x;
      centreY=y;
      angleGlisser = angleGlisser+ 5;
      actionGlissiere(angleGlisser);
   }
/*************************************************************************
 *
 *              NE PAS MODIFIER CETTE SECTION
 *
 *************************************************************************/
   public static String[][] Menus = {nomsMenus1, nomsMenus2, nomsMenus3, nomsMenus4,
nomsMenus5, nomsMenus6, nomsMenus7, nomsMenus8};
```

Voir étape 4

```
   public Expresso(int l, int h, String titre, String[] nomsBoutons1, String[] nomsBoutons2, String[][]
Menus, boolean avecTexte){
      super(l, h, titre, nomsBoutons1, nomsBoutons2, Menus, avecTexte);
   }
   public Expresso() {
      this(baseFenetre, hauteurFenetre, titreFenetre, nomsBoutonsLigne1, nomsBoutonsLigne2, Menus,
zoneTexte);
   }
   public static void executer(boolean applet){
      initGlissieres();
      ajoutDeGlissieres();
      Expresso maFenetre = new Expresso();
      faireApplet(applet);
      maFenetre.toFront();
      maFenetre.initialisation();
   }
   public static void main(String[] args){
      executer(false);
   }
}
```

Étape 8 fabriquer une application autonome et un applet

On procède exactement comme décrit aux étapes 8 et 9 de l'exemple 1.