Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr

Objectifs: Créer et utiliser des fonctions. Valider des fonctions de test. Créer des fonctions de test. Utiliser des constantes et variables globales. Imbriquer des structures de contrôle.

Rappel : Lorsqu'un programme contient des fonctions de test, dont le nom commence par `test`. Ces fonctions sont exécutées à condition qu'il n'y ait pas de fonction `void algorithm` présente. Généralement, on positionne une fonction de tests au-dessus de la fonction testée.

```
class MonProgramme extends Program {
    //test (exécuté automatiquement) utilisant des assertions
    //pour vérifier certains cas d'utilisation de maFonction
     void testMaFonction() {
        assertEquals(... , ...); //vérifie que les deux paramètres ont la même valeur
    }

    // Déclaration de la fonction nommée maFonction
     <type du résultat> maFonction(<type et nom des paramètres>) {
       // déclarations de variables si utile
       // calcul dépendant des paramètres
        return <valeur de retour>;
    }
}
```

Exercice 1 : Masque d'une chaîne

Écrivez un programme `MasqueDeChaine` contenant la fonction `masque` qui prend en paramètre une chaîne `phrase` et un caractère `car`, et retourne une nouvelle chaine qui ne contient que les occurrence de `car` à leur emplacement. Les autres caractères sont remplacés par des points. **Attention, soyez bien attentif au fait qu'ici c'est une fonction qui retourne la chaîne créée, pas d'affichage donc**. Réalisez la fonction devant passer le test qui suit. Vous pouvez vous inspirez de la fonction `copieSans` de votre cours.

```
void testMasque () {
  assertEquals("a...a...a....",     masque("au bal masqué",     'a'));
  assertEquals(".............",     masque("Tonari no Totoro",  'u'));
  assertEquals(".o.....o..o.o.o",   masque("Tonari no Totoro",  'o'));
  assertEquals("",                  masque("",                  'z'));
}
```

Une fois votre fonction réalisée, pensez bien entendu à la tester en l'incluant avec la fonction de test dans un programme. À l'exécution, `testMasque` s'affichera en vert si votre fonction a bien été réalisée, en rouge sinon, avec un message expliquant le problème. Dans ce cas, retravaillez votre fonction jusqu'à ce que le test passe au vert.

Exercice 2 : L'odyssée de Pi

Voici un programme proposant quelques fonctions

```
1  class Cercle extends Program{
2
3    double circonference(double rayon) {
4      return 3.14 * diametre(rayon);
5    }
6
7    double diametre(double rayon) {
8      return 2 * rayon;
9    }
10
11   double aire(double rayon) {
```

```
12        return 3.14 * rayon * rayon;
13      }
14
15      double volume(double rayon){
16        return 4.0/3.0 * 3.14 * rayon * rayon *rayon;
17      }
18    }
```

Q1. Améliorez ce code en y intégrant deux constantes globale, l'une pour éviter la redondance d'une même valeur, l'autre pour éviter un calcul à chaque appel de la fonction volume.

Q2. Pour des calculs plus précis, on souhaite une valeur de pi avec 5 décimales après la virgule, modifiez votre programme dans ce sens.

Q3. Écrivez un `void algorithm()` qui pour des rayons allant de 1 à 15 affiche la circonference, l'aire et le volume, puis indique le nombre de multiplications effectuées au total. Pour ce faire, vous utiliserez une variable globale que vous mettrez à jour dans chaque fonction réalisant des multiplications.

Exercice 3 : Années bissextiles et tests

Dans le TP précédent, vous deviez créer une fonction `bissextile` capable de dire si une année est bissextile ou non. On souhaite automatiser la vérification de la validité de cette fonction. Reprenez le programme contenant cette fonction (ou réalisez-la si ça n'est pas déjà fait) et ajoutez lui des tests.

Créez un programme `TestBissextile` qui contient la fonction `bissextile` et la fonction de test `testBissextile` exprimant les assertions données ci-dessous:

— les années 2013, 2006, 1999, 1000, qui ne sont pas bissextiles
— les années 2000, 2012, 2024, 1600, qui sont bissextiles

Une fois votre fonction de test réalisée, compilez votre programme et exécutez-le pour vérifier que le test passe bien au vert.

Exercice 4 : Dessins de figures

On souhaite disposer de trois programmes offrant la possibilité de réaliser trois types de figures : un triangle plein, un triangle creux, une croix. On pourra dans chaque cas spécifier la taille et le caractère souhaité.

```
ijava Dessins
Figure : 1
Taille : 5
Caractere : +
+
++
+++
++++
+++++
```

```
ijava Dessins
Figure : 2
Taille : 4
Caractere : O
O
OO
O O
OOOO
```

```
ijava Dessins
Figure : 3
Taille : 5
Caractere : X
X...X
.X.X.
..X..
.X.X.
X...X
```

Une fois vos trois programmes fonctionnels, créez un programme `Dessins` qui définit les fonctions `trianglePlein`, `triangleCreux`, `triangleCroix` et une fonction `algorithm` qui effectue la saisie de la taille et du caractère et affiche successivement les trois figures avec les paramètres saisis.

Voici la fonction de test vérifiant la validité pour la fonction `trianglePlein` (optionnel : ajoutez les fonctions de tests pour les autres figures) :

```
1  void testTrianglePlein() {
2    assertEquals("*\n", trianglePlein(1, '*'));
3    assertEquals("o\noo\n", trianglePlein(2, 'o'));
4    assertEquals("+\n++\n+++\n", trianglePlein(3, '+'));
5  }
```

Exercice 5 : Chaîne de nombres décroissants

On désire concevoir une fonction générant une chaîne de caractères contenant les nombres pairs décroissants entre n (dont on supposera qu'il est pair et supérieur ou égal à 2) et 1, sans espace entre les différents nombres.

Créez le programme `Decompte` dans lequel vous veillerez à ne pas utiliser d'alternative (oui, c'est possible !).

1. Écrire la fonction `genereNombresPairs` qui passe ce test.

```
void testGenereNombresPairs1() {
  assertEquals("8642", genereNombresPairs1(8));
  assertEquals("12108642", genereNombresPairs1(12));
  assertEquals("2", genereNombresPairs1(2));
}
```

2. Écrire une fonction similaire mais qui produit une étoile avant et après la suite de nombres:

```
void testGenereNombresPairs2() {
  assertEquals("*8642*", genereNombresPairs2(8));
  assertEquals("*12108642*", genereNombresPairs2(12));
  assertEquals("*2*", genereNombresPairs2(2));
}
```

3. Écrire une troisième version de la fonction, qui produit une étoile entre chaque paire de nombres consécutifs.

```
void testGenereNombresPairs3() {
  assertEquals("8*6*4*2", genereNombresPairs3(8));
  assertEquals("12*10*8*6*4*2", genereNombresPairs3(12));
  assertEquals("2", genereNombresPairs3(2));
}
```

Exercice 6 : Toutes les sous-chaînes

On souhaite réaliser un algorithme qui énumère toutes les sous-chaînes possibles d'un chaine donnée. Pour vous aider à trouver l'algorithme, répondez à la série de questions ci-dessous qui devraient vous permettre d'identifier les éléments clés de ce problème:

1. Avec la chaîne initiale, `"ABC"` écrivez toutes les sous-chaînes possibles. Combien en trouvez vous ?
2. Avec la chaîne initiale `"abcdefghijk"`, pouvez vous caractériser la chaîne `"defg"` au moyen de deux entiers ? Quels sont-t-ils ?
3. Quels sont les valeurs possibles de ces deux entiers sur cet exemple ?
4. Existe-t-il une ou plusieurs contraintes entre ces deux entiers ?
5. Ecrivez la boucle qui parcourt toutes les valeurs possibles pour le premier entier.
6. Ecrivez la boucle qui, étant donné la valeur du premier entier, parcourt toutes les valeurs possibles pour le second entier.
7. Ecrivez l'expression qui retourne la sous-chaîne à partir des deux entiers et de la chaîne.

Muni de tous ces éléments, vous devriez pouvoir maintenant écrire le programme `SousChaines` qui saisit un mot auprès de l'utilisateur et affiche toutes les sous-chaînes de ce mot.

Exercice 7 : Remplacer dans une chaine

On souhaite créer la fonctionnalité *Rechercher-remplacer* dans un éditeur de texte. On vous demande pour cela d'écrire une fonction qui prend en paramètre trois chaines (`phrase`, `avant` et `apres`) et retourne en résultat une nouvelle chaine dans laquelle toutes les occurrences de `avant` ont été remplacées par `apres`.

Créez le programme `Remplacer` validant la fonctin de tests suivante :

```
void testCopieEnRemplacant () {
  assertEquals("15x35"  , copieEnRemplacant("15*35"     , "*"   , "x"));
  assertEquals("15 + 35", copieEnRemplacant("15 plus 35", "plus", "+"));
  assertEquals("abcd"   , copieEnRemplacant("abcd"      , "cb"  , "xy"));
  assertEquals("abcd"   , copieEnRemplacant("abcd"      , ""    , "x"));
  assertEquals("abcd"   , copieEnRemplacant("-ab-cd-"   , "-"   , ""));
  assertEquals("xx"     , copieEnRemplacant("aaaa"      , "aa"  , "x"));
  assertEquals("xxa"    , copieEnRemplacant("aaaaa"     , "aa"  , "x"));
  assertEquals("9 plus 3 plus 3", copieEnRemplacant("9 moins 3 moins 3", "moins", "plus"));
}
```