



# CNED

## B1.2-PARTIE-I-SQ1-S2



MINISTÈRE  
DE L'ÉDUCATION  
NATIONALE, DE  
L'ENSEIGNEMENT  
SUPÉRIEUR ET DE  
LA RECHERCHE

[www.cned.fr](http://www.cned.fr)

Les cours du CNED sont strictement réservés à l'usage privé de leurs destinataires et ne sont pas destinés à une utilisation collective. Les personnes qui s'en serviraient pour d'autres usages, qui en feraient une reproduction intégrale ou partielle, une traduction sans le consentement du CNED, s'exposeraient à des poursuites judiciaires et aux sanctions pénales prévues par le Code de la propriété intellectuelle. Les reproductions par reprographie de livres et de périodiques protégés contenues dans cet ouvrage sont effectuées par le CNED avec l'autorisation du Centre français d'exploitation du droit de copie (20, rue des Grands Augustins, 75006 Paris)

CNED, BP 60200, 86980 Futuroscope Chasseneuil Cedex, France - © CNED

# Table des matières

<b>1</b>	<b>ACCUEIL .....</b>	<b>3</b>
1.1	OBJECTIFS.....	3
<b>2</b>	<b>SITUATION PROFESSIONNELLE .....</b>	<b>4</b>
2.1	VOTRE TÂCHE .....	4
<b>3</b>	<b>PARTIE 1 - APPORTS MÉTHODOLOGIQUES.....</b>	<b>5</b>
3.1	CE QU'IL FAUT SAVOIR .....	5
3.2	CONFIGURATION DE L'OUTIL DE REVUE DE CODE .....	5
3.3	EXEMPLE DE MODULES .....	6
<b>4</b>	<b>PARTIE 2 - APPLICATION .....</b>	<b>11</b>
4.1	APPLICATION .....	11
4.2	VOTRE TÂCHE .....	12
4.3	ÉTAPES À SUIVRE .....	13
4.4	CORRECTION .....	14
<b>5</b>	<b>EVALUER SES ACQUIS .....</b>	<b>14</b>
5.1	EVALUER SES ACQUIS : MODULES .....	14
5.2	EVALUER SES ACQUIS : GITHUB .....	15
5.3	EVALUER SES ACQUIS : REVUE DE CODE .....	16
<b>6</b>	<b>COMPLÉTER LES SAVOIRS .....</b>	<b>16</b>
6.1	COMPLÉTER LES SAVOIRS .....	16
<b>7</b>	<b>SYNTHÈSE .....</b>	<b>17</b>
7.1	SYNTHÈSE DE LA SÉANCE .....	17
<b>8</b>	<b>GLOSSAIRE.....</b>	<b>17</b>
<b>9</b>	<b>DOCUMENTATION .....</b>	<b>22</b>
9.1	SITUATION PROFESSIONNELLE .....	22
9.2	EXEMPLE .....	22
9.3	SECTION.....	22

## 1 Accueil

---

### 1.1 Objectifs



Image illustrative de développeurs testant un programme

#### 1.1.1 Rappel de la mission

Votre travail consiste à rechercher l'origine et les causes de différents [bugs](#) dans les [applications](#) informatiques de votre client. Ces applications sont développées dans un langage procédural.

#### 1.1.2 Ce que nous allons aborder

- Comprendre le fonctionnement et l'intérêt des [module](#) afin d'optimiser une application.
- Découvrir les différentes possibilités pour corriger, optimiser et normaliser une application en suivant les règles de la [revue de code](#).
- Configurer un outil d'aide à la revue de code dans l'IDE (Visual Studio).

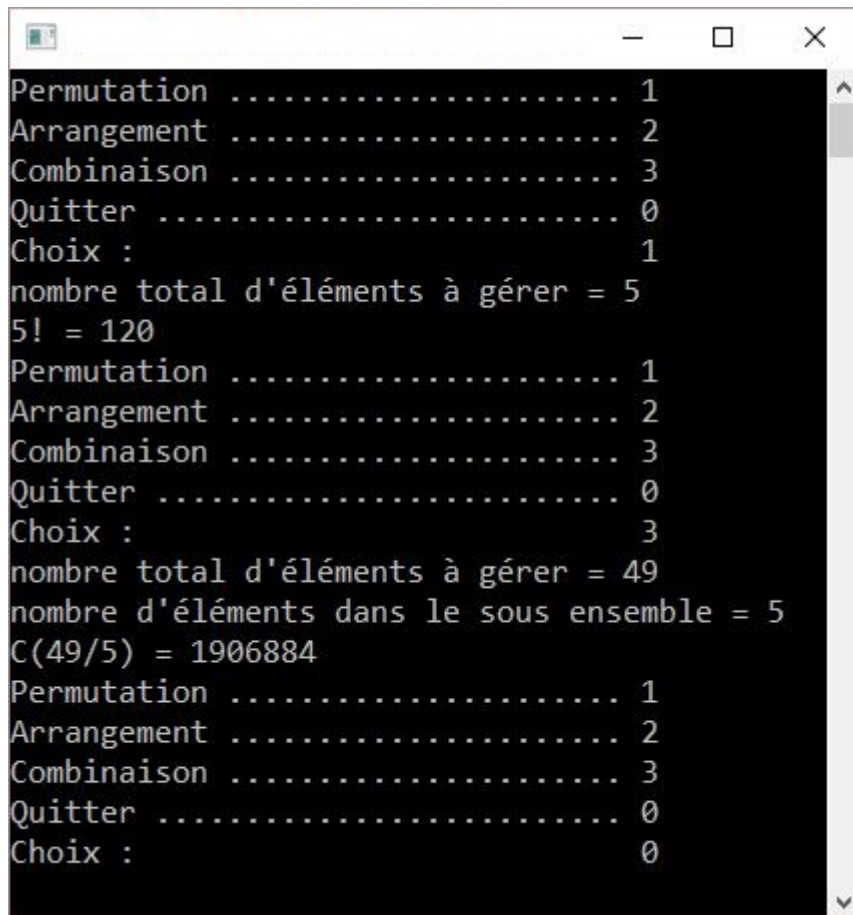
- Dans un contexte professionnel, apprendre à gérer la revue de code d'une application existante.

### 1.1.3 Temps de travail indicatif

3 heures

## 2 Situation professionnelle

### 2.1 Votre tâche



```

Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 1
nombre total d'éléments à gérer = 5
5! = 120
Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 3
nombre total d'éléments à gérer = 49
nombre d'éléments dans le sous ensemble = 5
C(49/5) = 1906884
Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 0
  
```

Capture montrant un exemple d'exécution du programme dans une fenêtre console.

Affichage du menu avec 4 propositions :

choix 1 : permutation

choix 2 : arrangement

choix 3 : combinaison

choix 0 : quitter

Choix saisi : 1

Le programme demande le nombre total d'éléments à gérer.

5 a été saisi.

Affichage final :  $5! = 120$

Nouvel affichage du menu.

Choix saisi : 3

Le programme demande le nombre total d'éléments à gérer

49 est saisi.

Le programme demande le nombre d'éléments dans le sous ensemble.  
5 est saisi.  
Affichage final :  $C(49/5) = 1906884$   
Nouvel affichage du menu.  
Choix saisi : 0 (pour arrêter le programme)

### 2.1.1 Situation professionnelle

#### 2.1.1.1 Présentation du contexte :

L'entreprise cliente a demandé la création d'une nouvelle fonctionnalité qui, pour le moment, doit prendre la forme d'une petite application console. Celle-ci sera ensuite adaptée et intégrée dans un projet global de tests de connaissances en mathématiques, à différents niveaux de difficultés. Un des développeurs de votre entreprise a pris en charge ce développement.

#### 2.1.1.2 Présentation de l'application :

L'application permet de réaliser 3 calculs mathématiques : permutation, arrangement et combinaison (vous pouvez trouver des explications et exemples simples <http://villemin.gerard.free.fr/Wwwgvm/Compter/Factcomb.htm>).

En rappel :

**Permutation** : tous les mélanges possibles dans un ensemble d'objets

**Arrangement** : tirage d'une quantité d'objets dans l'ordre

**Combinaison** : tirage d'une quantité d'objets sans ordre

#### 2.1.1.3 Présentation de la tâche qui vous a été attribuée :

Votre chef de projet vous demande de réaliser la [revue de code](#) de cette application. Il vous rassure sur un point : les formules mathématiques sont correctes dans l'application. Pour vous aider dans cette tâche, vous avez à votre disposition la **fiche sur la revue de code** ([téléchargeable ici](#)), et les [règles de codage](#) établies au sein de l'équipe de développeurs ([téléchargeable ici](#)). Après avoir récupéré le projet ([téléchargeable Denombrements.zip](#)), votre travail consiste à remplir le rapport d'analyse de la revue de code ([téléchargeable Rapport revue de code](#)) qui contient la checklist des points à contrôler. Vous travaillerez aussi avec [GitHub](#), pour soumettre les propositions de corrections.

## 3 Partie 1 - Apports méthodologiques

---

### 3.1 Ce qu'il faut savoir



Pour réaliser cette situation professionnelle, vous devez avoir acquis les savoirs suivants : Configuration de l'outil de revue de code dans l'IDE Visual Studio. Pour cela, suivez le mode opératoire présenté dans la vidéo page suivante (vous pourrez aussi télécharger la version pdf sous la vidéo). Découverte des modules à travers un exemple : outil indispensable pour optimiser le code. Cet exemple est présenté à la suite de de la configuration de l'outil de revue de code.

### 3.2 Configuration de l'outil de revue de code

### 3.2.1 Mode opératoire en vidéo

Le mode opératoire au format PDF est [téléchargeable ici](#)

Le mode opératoire au format PDF est téléchargeable ici et vous pouvez aussi retrouver la vidéo sur <https://youtu.be/p4Y0WvpEGgU>.

## 3.3 Exemple de modules

### 3.3.1 But de l'exemple

### 3.3.2 Découverte des modules pour optimiser le code

Un module est un bloc de code isolé, qui peut être appelé à tout moment. Il existe deux variantes qui n'ont pas tout à fait le même objectif et qui ne s'utilisent pas de la même façon : la procédure et la fonction. Sans vous en apercevoir, vous avez déjà utilisé des modules dans les applications précédentes (par exemple, Parse pour changer le type, Write pour écrire à l'écran...). Ces modules sont un peu spéciaux car ils sont précédés de ce que l'on appelle une Classe, notion que vous verrez plus tard. Dans cette séance, vous allez apprendre à créer vos propres modules, directement utilisables dans votre programme.



Après la réalisation de la situation professionnelle qui suit cet exemple, vous serez invité à approfondir vos connaissances en étudiant la fiche de savoirs : cette étape est indispensable pour aborder les séances suivantes.

#### 3.3.2.1

#### But de l'exemple :

Le but est de reprendre le précédent exemple du nombre caché, afin de l'optimiser, étape par étape. A droite, vous avez un rapide aperçu du code, avec 3 blocs de code similaires. L'utilisation de module va permettre d'éviter ces répétitions de code.

```

static void Main(string[] args)
{
    // déclaration
    int valeur = 0, essai = 0, nbre = 1;
    bool correct = false;
    // saisie du nombre à chercher
    while (!correct)
    {
        try
        {
            Console.Write("Entrez le nombre à chercher = ");
            valeur = int.Parse(Console.ReadLine());
            correct = true;
        }
        catch
        {
            Console.WriteLine("Erreur de saisie : saisissez une nombre entier");
        }
    }
    Console.Clear();
    // saisie du premier essai
    correct = false;
    while (!correct)
    {
        try
        {
            Console.Write("Entrez un essai = ");
            essai = int.Parse(Console.ReadLine());
            correct = true;
        }
        catch
        {
            Console.WriteLine("Erreur de saisie : saisissez une nombre entier");
        }
    }
    // boucle sur les essais
    while (essai != valeur)
    {
        // test de l'essai par rapport à la valeur à chercher
        if (essai > valeur)
        {
            Console.WriteLine(" --> trop grand !");
        }
        else
        {
            Console.WriteLine(" --> trop petit !");
        }
        // saisie d'un nouvel essai
        correct = false;
        while (!correct)
        {
            try
            {
                Console.Write("Entrez un essai = ");
                essai = int.Parse(Console.ReadLine());
                correct = true;
            }
            catch
            {
                Console.WriteLine("Erreur de saisie : saisissez une nombre entier");
            }
        }
        // compteur d'essais
        nbre++;
    }
    // valeur trouvée
    Console.WriteLine("Vous avez trouvé en "+nbre+" fois !");
    Console.ReadLine();
}

```

Capture montrant un aperçu du code de l'application NombreCache avec, sur fond gris, 3 blocs de code qui vont subir une optimisation dans les pages suivantes.

### 3.3.3 Eviter la répétition de code

#### 3.3.3.1 La procédure (module sans retour) :

Ouvrez le projet NombreCache (vu dans la séance 1, que vous pouvez à nouveau télécharger [NombreCache.zip](#)) et remarquez la partie de code qui permet de gérer la saisie de l'essai. Il y a une saisie avant la boucle (la première saisie) et une autre dans la boucle. Les 2 blocs de codes qui permettent aussi de contrôler que la saisie est correcte, sont strictement identiques :

Le but est d'éviter la répétition de ce code identique. Pour cela, on va créer un module en dehors du [Main](#), pour isoler le bloc de code. Juste au-dessus du Main, ajoutez le code suivant :

et entre les accolades, collez le code précédent.

Dans le nouveau module, remarquez que les [variable](#) 'correct' et 'essai' sont soulignées en rouge. Cela vient du fait que les variables ne sont pas [déclaration](#) dans le module. Pour corriger l'erreur sur 'correct', faites juste en sorte de déclarer la variable en mettant 'bool' devant la première ligne :

Il ne reste plus que 'essai' souligné. Mais on ne va pas pouvoir procéder de la même façon, car 'essai' doit pouvoir être récupéré dans le Main. La solution serait que 'essai' ne soit pas déclaré que dans Main ou dans le module 'saisie', mais à un niveau plus haut, visible par tout le monde. C'est possible et c'est ce qu'on appelle une "variable globale". Au-dessus du module 'essai', déclarez la variable de cette façon :

Remarquez qu'il n'y a plus d'erreur sur 'essai' dans le module. La variable 'essai' devant être unique et la même entre le module et le Main, il faut enlever sa déclaration dans le Main, tout en laissant son initialisation à 0 :

Remarquez que la variable 'essai' n'est pas de la même couleur, pour la distinguer des variables locales au Main.

Il ne reste plus qu'à supprimer les 2 blocs de code identique dans le Main (concernant la saisie de l'essai) et de les remplacer juste par l'appel du module :

### 3.3.4 test

Faites un test d'exécution. Normalement tout doit fonctionner comme avant.

*En cas de problème, le code de cette étape est téléchargeable [NombreCacheEtape1.zip](#).*

### 3.3.5 Se créer un outil

#### 3.3.5.1 La fonction (module avec retour) :

Il est possible d'éliminer la variable globale et de transformer le module pour qu'il "retourne" vers le programme appelant, la valeur saisie. Commencez par supprimer la déclaration de la variable globale



'essai' et remettez sa déclaration dans le Main, comme c'était avant :

Au niveau du module, la variable 'essai' est à nouveau soulignée. On va changer le nom de la variable. Mettez 'nombre' à la place de 'essai' et, en début de module, ajoutez la déclaration de 'nombre', de type 'int' et initialisé à 0. Sur la 1ère ligne du module, modifiez 'void' en 'int' car le module va retourner une valeur de type 'int'. Enfin, à la fin du module, ajoutez la ligne 'return nombre;':

Maintenant que la procédure a été transformée en fonction (un module est une procédure lorsqu'il ne retourne rien, une fonction lorsqu'il retourne quelque chose), il faut récupérer dans le Main, l'information retournée. Pour vous en persuader, vous allez faire un test :

### 3.3.6 test

Lancez l'application : quel que soit l'essai, vous obtenez "trop petit".

Pour mieux comprendre, placez un [point d'arrêt](#) sur la ligne "while (essai != valeur)", lancez le [débogage](#), saisissez la 1ère valeur et le 1er essai. Le programme s'arrête sur le point d'arrêt. Regardez la valeur dans 'essai' : 0.

Le problème vient du fait que dans le Main, on appelle la fonction 'saisie' mais on ne récupère par ce qu'elle retourne, et il n'y a plus de variable globale. Pour récupérer le retour, il suffit d'affecter l'appel de la fonction, dans une variable. Changez les 2 lignes de l'appel de 'saisie' par, à chaque fois, cette ligne :

### 3.3.7 test

Cette fois, l'application marche correctement.



Une fonction se manipule comme une valeur, puisqu'elle retourne une valeur. Voilà pourquoi on peut l'affecter dans une variable.

En cas de problème, le code de cette étape est téléchargeable [NombreCacheEtape2.zip](#).

### 3.3.8 Rendre l'outil adaptable

### 3.3.8.1 La fonction paramétrée :

Vous avez forcément remarqué qu'il y a encore un bloc de code, très similaire à celui copié dans le module 'saisie' : c'est celui qui concerne la saisie de la valeur à chercher. La différence est au niveau du message qui doit s'afficher.

Il est possible d'ajouter des [paramètres](#) à un module, pour lui envoyer une information. Modifiez l'entête de la fonction comme ceci :

On va pouvoir utiliser le paramètre 'message' au niveau de l'affichage. Dans le module, modifiez la ligne d'affichage comme ceci :

Remarquez qu'on a gardé " = " en dur, partant du principe que, quel que soit le message, l'affichage se finira ainsi. Dans le Main, les 2 appels à la fonction 'saisie' sont maintenant soulignés en rouge. C'est normal : la fonction attend un paramètre. Modifiez les 2 appels de la façon suivante :

L'information "Entrez un essai" va être transférée dans le paramètre 'message'.

### 3.3.9 test

Avant de poursuivre, vérifiez que tout fonctionne correctement. Essayez aussi de mettre un point d'arrêt sur la ligne du 1er appel de 'saisie'. Une fois arrivé sur cette ligne, vous remarquerez dans le menu "Déboguer" que vous pouvez soit faire F10 (pas à pas principal) soit F11 (pas à pas détaillé). Si vous faites F10, le débogueur ne va pas rentrer dans la fonction. Si vous faites F11, vous pourrez avancer pas à pas dans l'exécution de la fonction. Cela vous permet de voir la différence entre ces 2 modes.

Maintenant que la fonction est paramétrée, on va pouvoir l'utiliser aussi pour la saisie de la valeur à chercher. Remplacez tout le bloc de saisie de la valeur à chercher (de "bool correct = false;" à la fin de la boucle) par :

Remarquez que la même fonction peut être appelée avec une valeur de paramètre différente et aussi qu'elle peut être affectée dans une variable différente.

### 3.3.10 test

Vérifiez que tout fonctionne correctement.

Observez le code du Main : il est maintenant nettement plus court et compréhensible, et il n'y a plus de répétition de code.

*En cas de problème, le code de cette étape est téléchargeable [NombreCacheEtape3.zip](#).*

### 3.3.11 Perfectionner l'outil

#### 3.3.11.1 Plusieurs paramètres :

Il est possible de mettre plusieurs paramètres à un module (procédure ou fonction). Par exemple, on aimerait limiter la saisie pour des valeurs entre 1 et 100. L'idée est d'éviter les valeurs "en dur" dans la fonction, donc on va ajouter les bornes en paramètre, de la façon suivante :

Pour prendre en compte ces bornes, on va d'abord en informer l'utilisateur en les affichant dans le message :

Il y a déjà une [itération](#) pour vérifier que la saisie est correcte (numérique). Il suffit de modifier le test de la boucle pour prendre aussi en compte les bornes :

À nouveau, dans le Main, les appels de la fonction 'saisie' sont soulignés, car il manque des paramètres. Plutôt que de mettre à chaque fois 1 et 100, et toujours dans l'esprit d'éviter les valeurs "en dur" et les répétitions de code, commencez par déclarer et initialiser 2 variables pour mémoriser les bornes (dans le Main) :

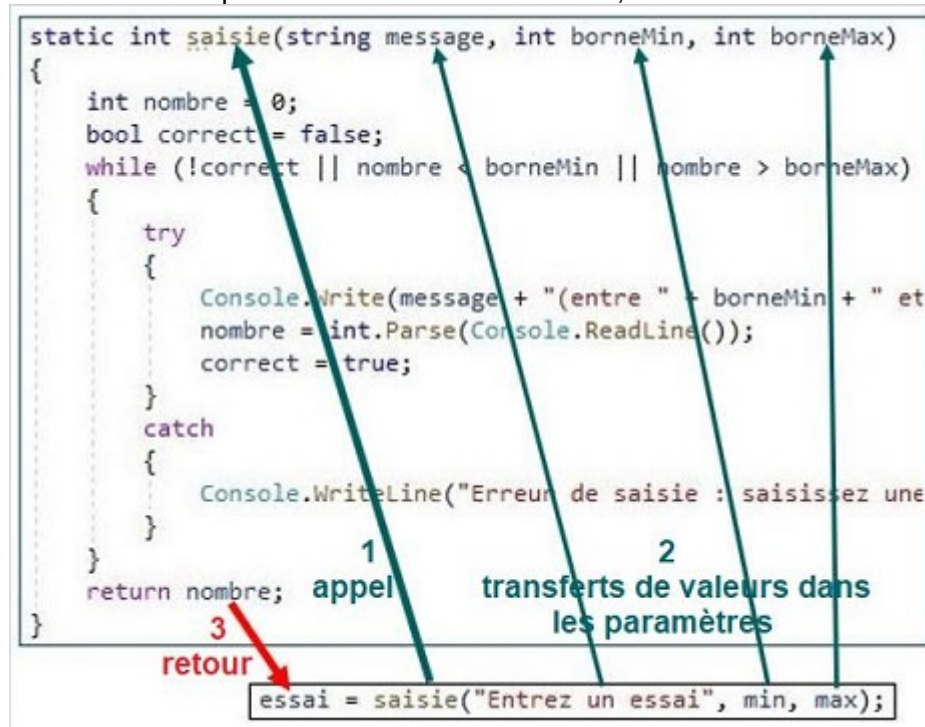
Les noms des variables peuvent être identiques ou différents des paramètres de la fonction : cela n'a aucune importance : les 2 variables du Main ne sont visibles que dans le Main. Les 2 paramètres de la fonction ne sont visibles que dans la fonction. En revanche, en appelant la fonction, on envoie en paramètres, les variables ou valeurs que l'on veut. Donc, pour les 3 appels de la fonction, ajoutez les 2 paramètres manquants. Par exemple :

### 3.3.12 test

Vérifiez que tout fonctionne correctement. Contrôlez aussi que si vous tapez un nombre inférieur à 1 ou supérieur à 100, la saisie est redemandée.

En cas de problème, le code de cette étape est téléchargeable [NombreCacheEtape4.zip](#).

La capture ci-dessous récapitule le fonctionnement : la fonction 'saisie' est appelée avec transfert des valeurs vers les paramètres. En fin d'exécution, une valeur est retournée.



## 4 Partie 2 - Application

### 4.1 Application





A partir de la situation professionnelle déjà présentée en début de séance, à votre tour désormais de réaliser en autonomie la mission qui vous a été confiée. Vous trouverez à la page suivante le rappel de la situation professionnelle, puis les étapes à suivre. Pour la réalisation de la situation professionnelle, vous devez avoir suivi les étapes présentées dans la Partie 1 - Apports méthodologiques.

## 4.2 Votre tâche

```
Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 1
nombre total d'éléments à gérer = 5
5! = 120
Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 3
nombre total d'éléments à gérer = 49
nombre d'éléments dans le sous ensemble = 5
C(49/5) = 1906884
Permutation ..... 1
Arrangement ..... 2
Combinaison ..... 3
Quitter ..... 0
Choix : 0
```

Capture montrant un exemple d'exécution du programme dans une fenêtre console.

Affichage du menu avec 4 propositions :

choix 1 : permutation

choix 2 : arrangement

choix 3 : combinaison

choix 0 : quitter

Choix saisi : 1

Le programme demande le nombre total d'éléments à gérer.

5 a été saisi.

Affichage final :  $5! = 120$

Nouvel affichage du menu.

Choix saisi : 3

Le programme demande le nombre total d'éléments à gérer

49 est saisi.

Le programme demande le nombre d'éléments dans le sous ensemble.

5 est saisi.

Affichage final :  $C(49/5) = 1906884$

Nouvel affichage du menu.

Choix saisi : 0 (pour arrêter le programme)

## 4.2.1 Rappel de la situation professionnelle

### 4.2.1.1 Présentation du contexte :

L'entreprise cliente a demandé la création d'une nouvelle fonctionnalité qui, pour le moment, doit prendre la forme d'une petite application console. Celle-ci sera ensuite adaptée et intégrée dans un projet global de tests de connaissances en mathématiques, à différents niveaux de difficultés. Un des développeurs de votre entreprise a pris en charge ce développement.

### 4.2.1.2 Présentation de l'application :

L'application permet de réaliser 3 calculs mathématiques : permutation, arrangement et combinaison (vous pouvez trouver des explications et exemples simples

<http://villemin.gerard.free.fr/Wwwgvm/Compter/Factcomb.htm>).

En rappel :

**Permutation** : tous les mélanges possibles dans un ensemble d'objets

**Arrangement** : tirage d'une quantité d'objets dans l'ordre

**Combinaison** : tirage d'une quantité d'objets sans ordre

### 4.2.1.3 Présentation de la tâche qui vous a été attribuée :

Votre chef de projet vous demande de réaliser la [revue de code](#) de cette application. Il vous rassure sur un point : les formules mathématiques sont correctes dans l'application. Pour vous aider dans cette tâche, vous avez à votre disposition **la fiche sur la revue de code** (téléchargée depuis la page 2 de cette séance), et les [règles de codage](#) établies au sein de l'équipe de développeurs (téléchargeable [ici](#)). Après avoir récupéré le projet (téléchargeable [Denombrements.zip](#)), votre travail consiste à remplir le rapport d'analyse de la revue de code (téléchargeable [Rapport revue de code](#)) qui contient la checklist des points à contrôler. Vous travaillerez aussi avec [GitHub](#), pour soumettre les propositions de corrections.

## 4.3 Etapes à suivre

1. Si ce n'est pas déjà fait, récupérez le code du projet [Denombrements.zip](#).
2. Ouvrez le sous Visual Studio et publiez-le sur [GitHub](#).
3. Fermez le projet et, dans un nouveau dossier, ouvrez un nouveau projet à partir du clone du premier projet (avec l'adresse de clone récupérée sur GitHub)
4. Ouvrez la solution et visualisez le code.
5. Créez une [branche](#).
6. Analysez le code et, pour chaque sous domaine\* présenté dans le rapport d'analyse de la [revue de code](#) (si vous ne l'avez pas encore récupéré, vous pouvez le télécharger [rapport revue de code.docx](#)) :
  - a. Apportez les corrections qui vous paraissent pertinentes.
  - b. Testez à nouveau le programme pour voir s'il marche correctement.

- c. Remplissez le rapport d'analyse de la revue de code.
7. Répétez l'étape 6 jusqu'à ce que vous ayez traité tous les points.
8. Envoyez les modifications vers GitHub ([valider tout et pousser](#)).
9. Demandez la prise en compte des modifications ([Pull Request](#)).

\* sous domaine : dans le rapport, cela correspond à chaque case (par exemple, la case "Commentaires" dans le domaine "Règles de codage")



Après avoir jeté un oeil au code, votre responsable vous dit qu'il est évident qu'une fonction doit être créée et utilisée dans tout le programme. Elle doit permettre de multiplier une suite d'entiers, d'une valeur à une autre (par exemple la multiplication de 4 à 6, donc  $4*5*6$ ). Il précise aussi qu'avec de tels calculs, il peut rapidement y avoir un dépassement de capacité, même avec le type numérique le plus grand, mais il vous demande de ne pas vous soucier de ce problème un peu complexe qui sera confié à un autre développeur.

#### 4.4 Correction



Vous pouvez désormais télécharger : le code de la version corrigée et optimisée de l'application le rapport d'analyse de la revue de code complété

### 5 Evaluer ses acquis

#### 5.1 Evaluer ses acquis : modules

Exercice 1 - Corrigé à la page 23

Répondez par vrai ou faux aux affirmations suivantes :

	Vrai	Faux
a) Une fonction se manipule comme une valeur dans le programme appelant.	<input type="radio"/>	<input type="radio"/>
b) Un paramètre est une variable locale au module.	<input type="radio"/>	<input type="radio"/>
c) On ne peut pas donner le même nom à un paramètre d'un module et une variable du programme appelant.	<input type="radio"/>	<input type="radio"/>
d) 2 variables locales de 2 modules différents peuvent avoir le même nom.	<input type="radio"/>	<input type="radio"/>
e) Un module peut appeler un autre module.	<input type="radio"/>	<input type="radio"/>
f) Un module peut avoir une variable locale du même nom que l'un de ses paramètres.	<input type="radio"/>	<input type="radio"/>

g) Un module doit toujours retourner une valeur.	<input type="radio"/>	<input type="radio"/>
h) Une fonction peut retourner plusieurs variables.	<input type="radio"/>	<input type="radio"/>
i) Une fonction peut être appelée plusieurs fois par le même module.	<input type="radio"/>	<input type="radio"/>

## 5.2 Evaluer ses acquis : GitHub

Exercice 2 - Corrigé à la page 24

Complétez les phrases en choisissant les termes corrects :

### Intérêt de GitHub :

GitHub permet de mémoriser en ligne, l'historique des ..... (1) d'une application et de la partager avec d'autres développeurs pour un travail collaboratif. Il est possible de configurer Visual Studio pour qu'il gère le lien avec GitHub.

### Mise en commun de l'application, par l'auteur, via GitHub :

L'auteur peut demander à VS de ..... (2) son application sur GitHub.

### Travail d'un collaborateur :

Le développeur qui veut collaborer au projet, doit récupérer ..... (3) sous VS. Il faut ensuite ouvrir ..... (4) . Avant toute modification sur le code, il doit créer ..... (5) pour travailler de façon isolée, sans risque de modification directe sur le projet. Une fois des corrections apportées au code, il peut demander d'enregistrer ces modifications sur GitHub en prenant l'option de modification ..... (6) . Enfin, il peut signaler à l'auteur qu'il a fait des propositions de modifications en envoyant une requête de ..... (7) vers GitHub.

### Prise en compte par l'auteur, des demandes du collaborateur :

L'auteur voit la requête et le message qui l'accompagne. S'il veut des informations complémentaires, il peut envoyer un message au développeur. S'il est d'accord avec la demande, il peut cliquer sur ..... (8) pour que les modifications soient enregistrées dans le projet qui est sur la branche master. Il peut aussi refuser et annuler la demande.

### Solutions proposées:

1:	sélectionner, fichiers, solutions, versions, packages
2:	sélectionner, publier, coller, modifier, sélectionner
3:	sélectionner, le zip, la solution, le clone, la configuration
4:	sélectionner, le zip, la solution, le clone, la configuration
5:	sélectionner, une branche, un clone, un pull request, un merge,
6:	sélectionner, "Valider tout", "Valider tout et enregistrer", "Valider tout et synchroniser", "Valider tout et pousser"
7:	sélectionner, sauvegarde, commit, tirage, push
8:	sélectionner, "Merge pull request", "Merge push response", "Pull merge response", "Push Merge request"

### 5.3 Evaluer ses acquis : revue de code

#### Exercice 3 - Corrigé à la page 24

*La revue de code consiste à chercher les erreurs et dysfonctionnements, contrôler le respect des règles de codage, mais aussi corriger les malfaçons. Pour chaque type de malfaçon, précisez le type de correction possible :*

1	variable non utilisée	.... .	créer une constante
2	valeur "en dur"	.... .	surveiller les imbrications, les répétitions
3	code mort	.... .	créer un module
4	code obscure	.... .	supprimer la déclaration
5	code non optimisé	.... .	supprimer le code
6	code dupliqué	.... .	réécrire
7	trop d'arguments	.... .	supprimer l'importation
8	commentaire obsolète	.... .	supprimer des paramètres
9	bibliothèque non utilisée	.... .	mettre à jour le commentaire

## 6 Compléter les savoirs

### 6.1 Compléter les savoirs







Pour compléter les savoirs abordés dans cette séance, vous pouvez maintenant consulter la fiche de savoirs sur les modules. L'étude de cette fiche est indispensable pour la suite de votre formation.. Consultez la Fiche de savoirs - Les modules. Téléchargez les codes de corrections des exercices de la fiche de savoirs. Vous pouvez aussi retrouver la correction en vidéo de la plupart des exercices, sur Youtube. Allez dans la playlist "Bases de la programmation (C#)" et suivez les exercices 43 à 55.

## 7 Synthèse

---

### 7.1 Synthèse de la séance

#### 7.1.1 Ce qu'il faut retenir

Pour gérer correctement la revue de code d'une application, voici la démarche à suivre :

- configurer un outil de gestion de versions dans l'IDE (par exemple l'extension GitHub sous Visual Studio) ;
- récupérer le clone de l'application déposée sur le serveur de gestion de versions (par exemple le site GitHub) ;
- créer une branche pour apporter des modifications sans toucher à la branche master ;
- faire un "commit and push" pour que les modifications soient envoyées vers le serveur ;
- faire un "pull request" pour informer le propriétaire de l'application, des propositions de modifications apportées (il décidera de les valider ou de les refuser).

La revue de code consiste au contrôle d'une application au niveau fonctionnement, respect des règles de codage et aussi recherche et corrections de malfaçons.

La création de modules (fonctions ou procédures) représente un outil indispensable pour optimiser le code.

#### 7.1.2 Notions clés

- **Revue de code** pour corriger, optimiser une application mais aussi harmoniser les codes, partager et former
- **GitHub** pour gérer le versionning et le travail collaboratif
- **Règles de codage** pour harmoniser le code dans une entreprise
- **Modules** (fonctions ou procédures) pour éviter les répétitions de code et se créer des outils réutilisables

#### 7.1.3 Pour approfondir

- [Revue de code \(Wikipedia\)](#)
- [Revue de code \(atomrace\)](#)
- [Règles de codage \(Wikipedia\)](#)

## 8 Glossaire

---

## Affectation

Transfert d'une valeur, variable ou calcul dans une variable (avec compatibilité de type).

En C#, le symbole de l'affectation est '='.

Exemples :

```
correct = true;
```

```
nbre = nbre + 1; // raccourci d'écriture : nbre++;
```

```
variable1 = variable2;
```

## Alternative

Ensemble d'instructions qui s'exécutent sous condition (synonymes : condition, test)

Exemple :

```
if (essai > valeur)
```

```
{  
    Console.WriteLine("--> trop grand !");  
}
```

```
else  
{
```

```
    Console.WriteLine("--> trop petit !");  
}
```

## Application

Ensemble d'instructions formant un tout et pouvant s'exécuter pour répondre à une demande (synonyme : programme).

## Autocompletion

Aide en cours de frappe qui affiche les différentes possibilités (mot du langage, variable...) par rapport à ce qui a commencé à être saisi.

## Bloc de code

Ensemble d'instructions entourées par délimiteurs (en C#, des accolades).

Les différentes structures (programme, alternative, itération, ...) contiennent un bloc de code.

## Bug

## Erreur d'exécution

## Commentaire

Information normalement ignorée par l'ordinateur, mais qui apporte des explications en clair sur les fonctionnalités du programme.

Les commentaires peuvent être mis à n'importe quel endroit du code.

Il existe 3 types de commentaires. Les voici avec les syntaxes en C# :

### **Commentaire d'une ligne :**

// ce qui suit le double slash est un commentaire, jusqu'à la fin de la ligne

### **Commentaire de plusieurs lignes :**

/\* tout ce qui est entre le slash-étoile

et le étoile-slash, même sur plusieurs lignes

est un commentaire \*/

### **Commentaire de type cartouche :**

C'est le seul interprété par l'ordinateur et il ne se positionne pas n'importe où : il est en début de programme (ou avant des modules : notion qui sera vue plus tard).

/\*\*

\* Jeu du nombre caché

\* author : Emads

\* date : 23/05/2020

\*/

## Compilation

Analyse du code pour repérer les erreurs syntaxiques (par exemple une variable utilisée mais non déclarée) puis, en l'absence d'erreur, traduction du code en langage "machine" directement compréhensible par l'ordinateur.

## Concaténation

Construction d'une chaîne en ajoutant, les unes à la suite des autres, plusieurs chaînes (valeurs entre guillemets et/ou variables).

## Condition

Prédicat qui est soit vrai, soit faux.

Une alternative (if, switch) contient une condition. Une itération (while...) contient une condition.

## Débogage

Recherche de dysfonctionnements dans le programme.

La recherche doit se faire à 3 niveaux :

- tests des possibilités classiques (fonctionnement de base de l'application)
- tests des cas particuliers
- tests des comportements inattendus

## Déclaration

Déclarer une variable consiste à lui donner un nom et un type. Cette étape est obligatoire en C# (et dans plusieurs langages) avant de pouvoir utiliser la variable.

## Fonction

Bloc de code isolé, indépendant, qui peut être sollicité par un programme ou un autre module.

Il existe 2 catégories de modules :

- la procédure : elle permet juste d'isoler un bloc de code et peut être appelée au moment où ce bloc de code doit être exécuté
- la fonction : elle se manipule comme une valeur car elle retourne une valeur (on peut donc l'affecter à une variable, l'utiliser dans un calcul, dans un test, l'afficher...)

Les modules peuvent recevoir des paramètres, donc des informations du programme appelant.

## GitHub

Site proposant l'hébergement d'applications basé sur le logiciel de versionning Git qui offre de nombreux outils dont la mémorisation des différentes versions d'une application et le travail collaboratif.

Quelques termes en relation avec GitHub :

- **dépôt (repository)** : zone de stockage des fichiers du projet
- **branche** : zone de travail indépendante, dans laquelle il est possible d'enregistrer différentes modifications, sans affecter les autres branches du même projet
- **branche principale (master)** : contient le projet d'origine et final, cette branche reçoit les modifications des autres branches lorsque le responsable de la branche le décide.
- **valider (commit)** : enregistrement des modifications dans le dépôt actuel local
- **pousser (push)** : envoyer les modifications validées, vers le serveur distant, dans la branche concernée
- **requête de tirage (pull request)** : requête pour demander la prise en compte des modifications proposées
- **fusionner (merge)** : réunion de 2 branches (par exemple, récupération d'une branche secondaire pour mettre à jour la branche principale)

## IDE

Environnement de développement intégré (Integrated Development Environment).

Logiciel permettant de coder une application, avec un ensemble d'outils d'aide au codage (colorisation du code, aide en cours de frappe, débogage...).

Visual studio est un IDE.

## Indentation

Décalage dans le code pour mettre en évidence les différents blocs.

Exemple :

```
if (essai > valeur)
{
    Console.WriteLine("Trop grand !")
}
```

## Initialisation

Première affectation d'une valeur dans une variable

## Instruction

Ordre que l'ordinateur va exécuter : affichage, saisie, affectation, test...

## Itération

Ensemble d'instructions qui peuvent s'exécuter plusieurs fois, tant qu'une condition est respectée (synonyme : boucle).

Exemple :

```
while (essai != valeur)
{
    // instructions qui se répètent tant que la condition est vraie.
}
```

## Main

Module principal qui s'exécute automatiquement dès l'exécution de l'application.

Vous verrez par la suite qu'il existe d'autres types de modules (un module est un regroupement d'instructions qui représente une unité indépendante).

## Module

Bloc de code isolé, indépendant, qui peut être sollicité par un programme ou un autre module.

Il existe 2 catégories de modules :

- la procédure : elle permet juste d'isoler un bloc de code et peut être appelée au moment où ce bloc de code doit être exécuté
- la fonction : elle se manipule comme une valeur car elle retourne une valeur (on peut donc l'affecter à une variable, l'utiliser dans un calcul, dans un test, l'afficher...)

Les modules peuvent recevoir des paramètres, donc des informations du programme appelant.

## Paramètre

Variable qui est déclarée dans l'entête d'un module, qui est locale au module mais qui a la particularité de pouvoir recevoir une information provenant du programme qui appelle le module.

## Pas à pas

Le débogueur permet une exécution "pas à pas", c'est à dire en s'arrêtant après chaque ligne (instruction) de code exécutée, pour montrer l'ordre d'exécution des instructions et l'évolution du contenu des variables.

## Point d'arrêt

Marque placée sur une ligne du programme, repérée par le débogueur qui arrêtera l'exécution au niveau de cette ligne, permettant ensuite d'avancer pas à pas dans l'exécution du code.

## Procédure

Bloc de code isolé, indépendant, qui peut être sollicité par un programme ou un autre module.

Il existe 2 catégories de modules :

- la procédure : elle permet juste d'isoler un bloc de code et peut être appelée au moment où ce bloc de code doit être exécuté
- la fonction : elle se manipule comme une valeur car elle retourne une valeur (on peut donc l'affecter à une variable, l'utiliser dans un calcul, dans un test, l'afficher...)

Les modules peuvent recevoir des paramètres, donc des informations du programme appelant.

## Programme

Ensemble d'instructions formant un tout et pouvant s'exécuter pour répondre à une demande (synonyme : application).

## Règles de codage

Ensemble de règles à respecter dans un code, afin d'armoniser les applications dans une même entreprise. Ces règles portent essentiellement sur la présentation du code, les commentaires et les pratiques de nommages.

## Revue de code

Elle consiste à analyser un code afin de repérer les erreurs, de proposer des améliorations pour respecter les normes de codages et les malfaçons.

Elle est généralement réalisée par un ou plusieurs développeurs qui ne sont pas à l'origine de la création du code.

Le créateur du code est informé des propositions de corrections et les valide ou non : l'utilisation d'un outil de versionning est généralement utilisé pour gérer la revue de code.

## Sensibilité à la casse

Distinction entre majuscule et minuscule.

Exemple : les variables 'total' et 'Total' sont différentes.

## Test

Ensemble d'instructions qui s'exécutent sous condition (synonymes : alternative, condition)

Exemple :

if (essai > valeur)

```
{  
    Console.WriteLine(" --> trop grand !");  
}  
else  
{  
    Console.WriteLine(" --> trop petit !");  
}
```

## Ticket d'incident

(synonyme : rapport d'incident)

Document qui recense des informations sur un dysfonctionnement repéré (application concernée, demandeur, date, catégorie, niveau de priorité, description du dysfonctionnement...).

Le ticket d'incident est alors confié à un technicien qui a la responsabilité de trouver l'origine du dysfonctionnement et de le corriger. Il complète alors le ticket d'incident et le clôture après avoir rendu une version opérationnelle de l'application.

## Transtypage

Changement de type d'une variable ou valeur (synonymes : parse, caste)

Exemples :

```
int essai = int.Parse(Console.ReadLine))  
int essai = int.Parse("23")
```

Type

Format d'une case mémoire pouvant mémoriser une variable.

Types simples (avec équivalence en C#) :

- entier (int)
- réel (float)
- chaîne (string)
- booléen (bool)

Variable

Zone mémoire nommée et typée, permettant de stocker une information utilisable et modifiable dans le programmes.

Visual Studio

Suite de logiciels de développement pour Windows et mac OS conçue par Microsoft.

## 9 Documentation

---

### 9.1 situation professionnelle

Denombrements

Règles de codage

Rapport de revue de code

Rapport de revue de code complété

Denombrements correction

### 9.2 Exemple

NombreCache

NombreCacheEtape1

NombreCacheEtape2

NombreCacheEtape3

NombreCacheEtape4

### 9.3 Section

Correction des exercices

mode opératoire au format PDF

Fiche savoirs

## Solutions

### Exercice 1 - Page 14

Répondez par vrai ou faux aux affirmations suivantes :

	Vrai	Faux
a) Une fonction se manipule comme une valeur dans le programme appelant.	●	○
b) Un paramètre est une variable locale au module.	●	○
c) On ne peut pas donner le même nom à un paramètre d'un module et une variable du programme appelant.	○	●
d) 2 variables locales de 2 modules différents peuvent avoir le même nom.	●	○
e) Un module peut appeler un autre module.	●	○
f) Un module peut avoir une variable locale du même nom que l'un de ses paramètres.	○	●
g) Un module doit toujours retourner une valeur.	○	●
h) Une fonction peut retourner plusieurs variables.	○	●
i) Une fonction peut être appelée plusieurs fois par le même module.	●	○

a) Une fonction se manipule comme une valeur dans le programme appelant : VRAI

C'est effectivement un outil qui se manipule comme une valeur. La fonction retourne une valeur. Voilà pourquoi on peut par exemple directement afficher l'appel d'une fonction.

b) Un paramètre est une variable locale au module : VRAI

Même si le paramètre permet de récupérer une valeur ou une adresse envoyée par le programme appelant, cela reste une variable locale au module. Il n'y a que le module qui le connaît et qui peut le manipuler.

c) On ne peut pas donner le même nom à un paramètre et une variable du programme appelant : FAUX

Même si le programme appelant peut envoyer une valeur ou une adresse dans un paramètre, il ne connaît même pas le nom du paramètre et ne peut pas y accéder (excepté lui transférer une information : valeur ou adresse). Du coup, ce n'est pas interdit d'avoir le même nom pour un paramètre et une variable du programme appelant.

d) 2 variables locales de 2 modules différents peuvent avoir le même nom : VRAI

Pour les mêmes raisons, chaque module voit ses propres variables locales mais ne voit pas les variables locales des autres modules. Donc les noms peuvent être identiques, ce sont tout de même des variables différentes.

e) Un module peut appeler un autre module : VRAI

C'est un cas qu'on a déjà vu. Le Main peut appeler des modules et chaque module peut en appeler d'autres. D'ailleurs, rappelons que le Main est un module, même s'il est un peu particulier.

f) Un module peut avoir une variable locale du même nom que l'un de ses paramètres : FAUX

Rappelons qu'un paramètre est une variable locale (même s'il permet en plus de recevoir une information du programme appelant). Du coup, au même titre que les variables locales, un paramètre ne peut pas avoir le même nom qu'une variable locale (dans le même module).

g) Un module doit toujours retourner une valeur : FAUX

Il y a 2 sortes de modules : ceux qui retournent une valeur (les fonctions) et ceux qui n'en retournent pas (les procédures).

h) Une fonction peut retourner plusieurs variables : FAUX

Le principe même d'une fonction est d'être manipulé comme une valeur, et pas plusieurs. La fonction ne peut donc retourner qu'une seule valeur. Ceci dit, cette valeur peut être de type complexe (par exemple un tableau).

i) Une fonction peut être appelée plusieurs fois par le même module : VRAI

C'est le cas aussi d'une procédure. On a déjà vu ce cas, par exemple pour les tests de saisie.

Exercice 2 - Page 15

*Complétez les phrases en choisissant les termes corrects :*

### **Intérêt de GitHub :**

GitHub permet de mémoriser en ligne, l'historique des **versions (1)** d'une application et de la partager avec d'autres développeurs pour un travail collaboratif. Il est possible de configurer Visual Studio pour qu'il gère le lien avec GitHub.

### **Mise en commun de l'application, par l'auteur, via GitHub :**

L'auteur peut demander à VS de **publier (2)** son application sur GitHub.

### **Travail d'un collaborateur :**

Le développeur qui veut collaborer au projet, doit récupérer le **clone (3)** sous VS. Il faut ensuite ouvrir la **solution (4)**. Avant toute modification sur le code, il doit créer une **branche (5)** pour travailler de façon isolée, sans risque de modification directe sur le projet. Une fois des corrections apportées au code, il peut demander d'enregistrer ces modifications sur GitHub en prenant l'option de modification **"Valider tout et pousser" (6)**. Enfin, il peut signaler à l'auteur qu'il a fait des propositions de modifications en envoyant une requête de **tirage (7)** vers GitHub.

### **Prise en compte par l'auteur, des demandes du collaborateur :**

L'auteur voit la requête et le message qui l'accompagne. S'il veut des informations complémentaires, il peut envoyer un message au développeur. S'il est d'accord avec la demande, il peut cliquer sur **"Merge pull request" (8)** pour que les modifications soient enregistrées dans le projet qui est sur la branche master. Il peut aussi refuser et annuler la demande.

Les termes présentés sont couramment utilisés : vous devez facilement les reconnaître et les comprendre. Ne cherchez pas forcément à les apprendre par coeur : il faut plutôt pratiquer pour les mémoriser naturellement.

Vous devriez revoir la configuration et l'utilisation de GitHub sous Visual Studio (partie 1 "apports méthodologiques").

Exercice 3 - Page 16

*La revue de code consiste à chercher les erreurs et dysfonctionnements, contrôler le respect des règles de codage, mais aussi corriger les malfaçons. Pour chaque type de malfaçon, précisez le type de correction possible :*



1	variable non utilisée	2	créer une constante
2	valeur "en dur"	5	surveiller les imbrications, les répétitions
3	code mort	6	créer un module
4	code obscure	1	supprimer la déclaration
5	code non optimisé	3	supprimer le code
6	code dupliqué	4	réécrire
7	trop d'arguments	9	supprimer l'importation
8	commentaire obsolète	7	supprimer des paramètres
9	bibliothèque non utilisée	8	mettre à jour le commentaire

Revoir la fiche sur la revue de code téléchargeable en page 2 de cette séance.