Cyril Seguenot

2016

Résumé

Description des exercices donnés aux stagiaires d’une formation sur C# 3.0 faite en présentiel. La formation s’adresse à des personnes ayant déjà des notions de base de programmation, mais pas nécessairement de connaissances sur C#.

Formation à C#

Exercices

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

© Cyril Seguenot - 2016

Exercices

[Quiz capitales 3](#_Toc26877241)

# Quiz capitales

**Objectifs** : apprendre la syntaxe de base du C#. Afficher des valeurs à l’écran, faire saisir des valeurs à l’utilisateur, utiliser les instructions if, else, for, while et les opérateurs de base.

Pour cet exercice, on utilisera une partie de la liste des pays et capitales européens :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pays | Capitale | Pays | Capitale | Pays | Capitale |
| Albanie | Tirana | Grèce | Athènes | Monténégro | Podgorica |
| Allemagne | Berlin | Hongrie | Budapest | Norvège | Oslo |
| Andorre | Andorre-la-Vieille | Irlande | Dublin | Pays-Bas | Amsterdam |
| Autriche | Vienne | Islande | Reykjavik | Pologne | Varsovie |
| Belgique | Bruxelles | Italie | Rome | Portugal | Lisbonne |
| Biélorussie | Minsk | Kazakhstan | Astana | Roumanie | Bucarest |
| Bosnie-Herzégovine | Sarajevo | Kosovo | Pristina | Royaume-Uni | Londres |
| Bulgarie | Sofia | Lettonie | Riga | Russie | Moscou |
| Chypre | Nicosie | Liechtenstein | Vaduz | Serbie | Belgrade |
| Croatie | Zagreb | Lituanie | Vilnius | Slovaquie | Bratislava |
| Danemark | Copenhague | Luxembourg | Luxembourg | Slovénie | Ljubljana |
| Espagne | Madrid | Macédoine | Skopje | Suisse | Berne |
| Estonie | Tallinn | Malte | La Valette | Suède | Stockholm |
| Finlande | Helsinki | Moldavie | Chisinau | République tchèque | Prague |
| France | Paris | Monaco | Monaco | Ukraine | Kiev |

**Etape 1 : Création du projet**

Dans la solution Exercices, créer un nouveau projet nommé Capitales

Dans la méthode Main, faire en sorte que le programme ne se termine qu’après avoir appuyé sur Entrée.

Lancer le programme pour tester.

**Etape 2** : première question

Dans la méthode Main, demander à l'utilisateur quelle est la capitale de l'Espagne. S'il répond bien, afficher "Bravo !", sinon, afficher "Mauvaise réponse".

Lancer le programme pour tester

**Etape 3** : ajouter 2 autres questions pour le Portugal et l'Italie.

Dans les étapes suivantes, on va automatiser le processus, pour ne pas avoir à écrire plusieurs fois le même code.

**Etape 4** :

* Mettre le code précédent en commentaire (il pourra être supprimé à la fin de l’étape suivante)
* Créer un premier tableau contenant les 10 premiers pays européens
* Créer un second tableau contenant leurs capitales

**Etape 5** : Parcourir le 1er tableau, et pour chaque pays :

* Demander à l'utilisateur la capitale
* « S'il répond bien, afficher "Bravo !", sinon, afficher "Mauvaise réponse. La réponse était…"

**Etape 6** : Modifier le code précédent pour ne poser qu’une question sur deux (soit 5 questions en tout). Tester et remettre le code comme il était (pour poser toutes les questions).

**Etape 7** : Modifier le code précédent pour poser les questions en partant de la fin

**Etape 8** : Créer un compteur de bonnes réponses, et afficher sa valeur à la fin du jeu

**Etape 9** : A la fin du jeu, demander à l’utilisateur s’il veut rejouer. S’il tape « o » ou « O », vider l’écran et relancer le jeu, sinon, afficher un message « Merci d’avoir joué »

**Etape 10 : création d’une fonction**

Déplacer le code de la fonction Main dans une fonction statique Jouer, et appeler cette dernière dans Main.

Sortir les tableaux pays et capitales de la méthode Jouer, pour les mettre directement dans la classe Program. Pour qu’ils restent utilisable dans la méthode Jouer, les faire précéder du mot clé static.

Exécuter le programme pour vérifier que tout fonctionne toujours bien.

**Etape 11** : Créer une méthode PoserQuestion qui prend en paramètre le numéro de la question à poser (qui correspond à l’indice du pays dans le tableau des pays).

Tester cette méthode en l’appelant dans la méthode Main

**Etape 12** : Questions aléatoires

Créer une méthode Jouer2, avec la logique suivante :

* Vider la console
* Afficher le message "Appuyer sur Echap pour arrêter le jeu"
* Générer un nombre aléatoire entre 0 et le nombre de pays – 1
* Appeler la méthode PoserQuestion en lui passant ce nombre en paramètre
* Répéter ceci tant que l’utilisateur n’a pas appuyé sur la touche Echap

Tester cette méthode en l’appelant dans Main à la place de la méthode Jouer.

Indication : pour générer un nombre aléatoire compris entre 0 et X, utiliser le code suivant :

Random rd = new Random();  
Rd.Next(X)

# Fonction avec plusieurs paramètres

Dans le projet Demo, écrire une fonction qui prend en paramètre le nom, l’année de naissance et le sexe d’une personne et qui renvoie un message du type :

« Bonjour Madame xxx, vous avez 30 ans » s’il s’agit d’une femme

ou bien « Bonjour Monsieur yyy, vous avez 30ans » s’il s’agir d’un homme

Tester cette fonction en l’appelant dans main pour une femme puis pour un homme.

# Analyse d’un mot

**Objectifs** : mettre en œuvre tableaux, fonctions, passage de paramètres en ref et out

**Etape 1** :

* Dans la fonction Main(), faire saisir un mot à l’utilisateur. On ne fera pas de vérification du mot saisi ; on s’attend à ce qu’il ne comporte que des lettres.
* Créer une fonction CompterLettres (vide pour l’instant) qui prend en entrée un mot, et renvoie les nombres de lettres, de voyelles et de consonnes
* Afficher le résultat de l’appel de cette fonction sous la forme : « ”livre” comporte 5 lettres, dont 3 consonnes et 2 voyelles »

**Etape 2** : implémenter le corps de la fonction vide créée précédemment et tester.

# Jeu du pendu

**Objectifs** : mettre en œuvre les notions suivantes : fonctions, énumérations, exceptions

**Etape 1** : dans la solution Exercices, ajouter le projet suivant :

****

**Etape 2** : compléter le code pour que le jeu fonctionne conformément au scénario suivant :

Jouez au pendu!

Saisissez un mot de 3 à 25 lettres sans accent :

en

Le mot doit contenir entre 3 et 25 lettres

Saisissez un mot de 3 à 25 lettres sans accent :

te8st

Le mot ne doit contenir que des lettres, sans accent

Saisissez un mot de 3 à 25 lettres sans accent :

Developpeur

Une fois le mot à deviner correctement saisi, on efface l’écran pour que l’autre joueur ne voit pas le mot, puis on affiche ceci :

Mot en cours de déchiffrage : -----------

Proposez une lettre :

Dès que le joueur saisi une lettre, on réaffiche la même chose :

* En mettant à jour le dessin de l’échafaud (pour l’instant vide) au-dessus du texte si la lettre proposée n’est pas dans le mot à deviner
* Ou bien en affichant la lettre à la place du ou des tirets correspondants dans la première ligne

Exemple, après avoir trouvé plusieurs lettres et fait plusieurs erreurs :

\_\_\_\_

|/ |

| o

| /|\

| /

|\_\_\_\_

Mot en cours de déchiffrage : -evel---e-r

Proposez une lettre :

A la fin, on affiche l’une des deux phrases suivantes :

Bravo, vous avez gagné :

Perdu ! Le mot à deviner était : developpeur

**Etape 3** : modifier le code pour gérer les erreurs du joueur au moyen d’exceptions de type FormatException.

**Etape 4** : remplacer les méthodes GetXXX de la classe Jeu par des propriétés XXX

**Etape 5** : pour la propriété MotADeviner, utiliser une implémentation automatique et faire les modifications nécessaires dans le reste du code.

# Mois et saisons

**Objectifs** : apprendre à manipuler les énumérations à bits indicateurs.

**Etape 1** : créer 2 énumérations de type flags pour les mois et les saisons.

**Etape 2** : écrire une fonction nommée SaisonsDuMois, qui permet de déterminer la ou les saisons d’un mois passé en paramètre de type énuméré. La fonction doit retourner également un énuméré, et doit utiliser exclusivement des opérations binaires.

Le tableau suivant montre la correspondance des mois et saisons :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Janv. | Fév. | Mars | | Avr. | Mai | Juin | | Juil. | Août | Sept. | | Oct. | Nov. | Déc. | |
| Hiver | | | Printemps | | | | Eté | | | | Automne | | | |  |

On voit par exemple que le mois de juin est à cheval sur le printemps et l’été.

**Etape 3**: tester cette fonction en l’appelant dans la méthode Main et en affichant son résultat pour différents mois

**Etape 3** : enlever l’attribut Flags sur les énumérations pour voir le résultat.

**Etape 4** : appeler la fonction pour tous les mois de l’année dans une boucle for

# Calcul du PGCD

**Objectifs** : écrire un programme simple à partir d’un algorithme décrit en UML

Voici un diagramme d’activité qui décrit l’algorithme de calcul du plus grand commun diviseur de deux entiers non nuls selon la méthode dite « égyptienne » :



Ecrivez un programme C# qui implémente cet algorithme. Exemple de résultat attendu :

Entrez le premier nombre : 21

Entrez le deuxième nombre : 45

Le PGCD de 21 et 45 est : 3

# Tri d’un tableau

**Objectifs** :

* Modéliser un algorithme avec un diagramme UML d’activité
* Implémenter cet algorithme en C#

Un algorithme possible pour trier un tableau est de comparer ses éléments deux à deux, et les permuter s’ils ne sont pas dans le bon ordre. Le tableau est trié lorsqu’on parcourt tous ses éléments sans faire aucune permutation.

**Etape 1** : modéliser l’algorithme de tri en UML

**Etape 2** : Avant d’implémenter cet algorithme, créer une fonction AfficherTableau permettant d’afficher le contenu d’un tableau d’entiers sur une seule ligne.

**Etape 3** : Créer une fonction TrierTableau permettant de trier par ordre croissant les éléments d’un tableau d’entiers passé en paramètre, selon l’algorithme modélisé précédemment.

**Etape 4** : initialiser un tableau de 10 entiers non triés, et le passer en paramètre à la fonction TrierTableau.

Utiliser la fonction AfficherTableau pour afficher le contenu du tableau avant et après l’appel à la fonction TrierTableau.

# Création d’un compte

**Objectifs** : apprendre à émettre et intercepter des exceptions

**Etape 1** : dans la classe Program, créer une fonction CréerCompte, qui :

* Demande à l’utilisateur de saisir successivement un login puis un mot de passe
* Affiche un message « Votre compte a bien été créé. Un message vient de vous être envoyé ».

Appeler cette fonction dans Main.

**Etape 2** : créer 2 fonctions pour vérifier les formats des informations du compte :

* Le login doit faire au moins 5 caractères.
* Le mot de passe doit comporter au moins 6 caractères, dont au moins une lettre et un chiffre.

La vérification du mot de passe peut se faire en parcourant les caractères de la chaîne, et en comparant leur valeur ASCII aux plages suivantes :

* Chiffres : codes 48 à 57
* Lettres majuscules : codes 65 à 90
* Lettres minuscules : codes 97 à 122

Si les formats ne sont pas bons, lever des exceptions du type FormatException, avec des descriptions explicites.

# Ecriture d’une note

**Objectifs** : illustrer le fonctionnement des instructions try…finally et using, et des espaces de noms.

**Etape 1** : dans la classe Program, créer une fonction SaisirNote, qui demande à l’utilisateur de saisir successivement un texte puis un chemin de fichier.

Appeler cette fonction dans Main.

**Etape 2** : dans une classe Note, créer une fonction EnregistrerNote qui prend en paramètre le chemin et le texte saisis précédemment. Cette fonction doit créer le fichier et écrire le texte dedans. Si le fichier existe déjà, il doit être complété.

Faire en sorte que le fichier soit fermé à la fin de l’opération même si une erreur se produit.

**Etape 3** : dans la fonction SaisirNote, appeler la méthode EnregistrerNote, en interceptant l’exception qui se produit dans le cas où le répertoire spécifié dans le chemin n’existe pas.

**Etape 4** : placer un point d’arrêt sur la ligne de code qui ferme le fichier s’il est ouvert

Vérifier en exécution que ce point d’arrêt est bien atteint, que le répertoire spécifié existe ou qu’il n’existe pas.

**Etape 5**: refaire le même test en mettant au préalable le fichier créé précédemment en lecture seule, en modifiant ses options dans l’explorateur de fichier de Windows.

**Etape 6** : faire en sorte que le point d’arrêt soit bien atteint quelle que soit l’erreur qui se produit au moment de l’enregistrement.

**Etape 7** : gérer la fermeture du fichier au moyen d’une instruction using.

**Etape 8** : dans le fichier Program.cs, utiliser le nom complet de l’exception DirectoryNotFoundException au lieu d’utiliser une directive using.

# Boîtes

**Objectifs** : mettre en œuvre les notions suivantes : constantes, propriétés, méthodes, constructeurs, surcharges et agrégation. Et de façon annexe : énumérations, exceptions et structuration du code.

**Etape 1** : Dans un nouveau projet nommé Boites, créer une classe nommée Boite avec des propriétés en lecture seule pour sa hauteur, sa largeur, sa longueur, toutes 3 initialisées par défaut à 30.0.

**Etape 2** : Créer un type énuméré pour les couleurs (Blanc, Bleu, Vert, Jaune, Orange, Rouge, Marron). Créer une propriété en lecture/écriture de ce type sur la classe Boite.

**Etape 3** : Créer une propriété énumérée en lecture seule pour la matière (Carton, Plastique, Bois, Métal), initialisée par défaut à Carton.

**Etape 4** : Créer une propriété Volume en lecture seule qui retourne le volume calculé d’après les dimensions.

**Etape 5** : Ajouter une méthode publique Etiqueter, avec un paramètre « destinataire » de type string. Créer une surcharge avec un paramètre booléen supplémentaire nommé « fragile ». La surcharge doit appeler la première méthode Etiqueter.

**Etape 6** : Ajouter une méthode Compare qui renvoie vrai si les dimensions et la matière des 2 boîtes sont identiques. La tester.

**Etape 7** : Ajouter 2 constructeurs à la classe Boîte. Le 1er permet d’initialiser ses dimensions. Le 2d permet d’initialiser en plus sa matière. Tester ces constructeurs en créant des boîtes dans la fonction Main

**Etape 8** : Faire en sorte que le second constructeur appelle le premier

**Etape 9** : ajouter un compteur d’instances sur la classe Boite, en utilisant une propriété statique pour retourner sa valeur. Tester son fonctionnement.

**Etape 10** : Ajouter une classe Etiquette avec 3 propriétés en lecture/écriture : Texte (string), Couleur (énuméré déjà créé), Format (énuméré avec valeurs XS, S, M, L, XL)

**Etape 11** : Dans la classe Boite, ajouter 2 champs privés de type Etiquette. Nommez-les \_etiquetteDest, \_etiquetteFragile

Dans la première méthode Etiqueter créée à l’étape 5, créer une instance d’Etiquette de couleur blanche, de format L, et dont le texte est le destinataire. Cette instance doit être accessible via le champ \_etiquetteDest

Dans la seconde méthode Etiqueter créée à l’étape 5, si le paramètre fragile vaut Vrai, créer une instance d’étiquette de couleur rouge, de format S, et dont le texte vaut « FRAGILE ». Cette instance doit être accessible via le champ \_etiquetteFragile.

Dans la fonction Main, créer une boîte, et l’étiqueter avec son destinataire et une étiquette « FRAGILE ».

On vient d’illustrer la composition.

**Etape 12 :** Créer une 3ème surcharge de la méthode Etiqueter, qui prend deux paramètres de type Etiquette (une pour le destinataire et une pour indiquer le caractère fragile)

Les étiquettes passées en paramètre doivent être affectées aux champs \_etiquetteDest et \_etiquetteFragile

Dans la méthode Main de la classe Program

* En utilisant le constructeur adéquat, créer une boîte en plastique de dimensions 30 x 40 x 50
* En utilisant un initialiseur, créer une étiquette de couleur blanche, de format L, avec comme texte un destinataire de votre choix
* Créer de la même façon une étiquette rouge de format S, avec le texte « FRAGILE »
* Affecter ces étiquettes à la boîte à l’aide de la méthode Etiqueter créée précédemment

On vient d’illustrer l’agrégation.

# Véhicules

**Objectifs** : mettre en pratique les notions suivantes : héritage, appels des constructeurs, classes abstraites, implémentation d’interface

**Etape 1** : Créer un type énuméré nommé Energies avec les valeurs : Aucune, Essence, Gazole, GPL, Electrique

Créer une classe Véhicule avec 3 propriétés en lecture : Nom, NbRoues et Energie

Ajouter un constructeur pour initialiser ces propriétés.

**Etape 2** : Créer une classe dérivée Voiture

Générer le constructeur proposé par VS, qui appelle celui de Véhicule

Mettre la valeur 4 pour le paramètre du nombre de roues.

Tester en instanciant une voiture et en exécutant le code pas à pas en debug.

Noter qu’on n’a pas eu besoin de propriétés en écriture.

**Etape 3** : Dans la classe Véhicule, ajouter une propriété virtuelle Description qui renvoie une chaîne « Véhicule X roule sur X roues et à l’énergie X » (remplacer X par les valeurs de propriétés).

Dans Voiture, redéfinir cette propriété. Par défaut VS génère automatiquement le corps en ajoutant un appel à la propriété de la classe ancêtre. Ajouter « Je suis une voiture \r\n » devant l’appel.

Dans Main, ajouter l’affichage de la description de la voiture et exécuter.

**Etape 4** : Créer une classe Moto dérivée de Véhicule et redéfinir la propriété Description

Dans la fonction Main, créer une instance de Voiture et une instance de Moto en les référençant par des variables de type Véhicule. Afficher la valeur de la propriété Description à partir de ces 2 variables.

On obtient un comportement polymorphique

# Distributeur de boissons

**Objectifs** : mettre en œuvre les notions suivantes :

* Propriétés et méthodes
* Séparation du visuel et de la logique applicative
* Accessoirement : énumérations et exceptions

On souhait modéliser un distributeur de boissons, dont le fonctionnement est décrit par le diagramme d’activité et les règles de gestion suivants :

|  |  |
| --- | --- |
|  | Règles de gestion :  Les boissons possibles sont :   * Café à 0.80 € * Chocolat à 0.75 € * Soupe à 1.00 €   Par défaut, la quantité de sucre est réglée sur 3g. Elle ne peut pas dépasser 10g. |

Pour cela, créer une classe Distributeur comportant :

* Les propriétés nécessaires
* Une méthode ServirBoisson qui débite la carte si le solde est suffisant et renvoie une chaîne indiquant la boisson et la quantité de sucre choisies. Exemples : « Café avec 5g de sucre », « Soupe », « Chocolat avec 0g de sucre »

Attention : pour que cette classe soit utilisable dans différents contextes, elle ne doit pas faire référence à la console.

Pour tester cette classe, dans la méthode Main de Program, simuler **en dur** le scénario suivant :

* Mettre une carte avec un solde de 2€
* Commander une soupe avec 5g de sucre 🡪 vérifier que la soupe est préparée, mais sans sucre
* Commander un café avec 12g de sucre 🡪 vérifier que le café est préparé, mais avec 10g de sucre
* Commander un chocolat 🡪 vérifier qu’une exception est levée à cause du solde insuffisant

# Relevés météo

**Objectifs** : mettre en œuvre les notions suivantes :

* Méthodes de la classe string et formats (étape 1)
* Types nullables (étape 2)
* Listes génériques (étapes 3 et 4)
* Requêtes Linq (étape 5)



Le fichier ci-joint donne les relevés météo mensuels de la ville de Paris depuis 2007 (données issues du [site de Météo France](http://www.meteofrance.com/climat/france/paris/75114001/releves)). Il contient une ligne d’en-tête et des lignes de données. Les valeurs sont séparées par des **tabulations**

**Etape 1 : Affichage du contenu du fichier à l’écran**

On souhaite afficher les données à l’écran sous la forme suivante :

Mois | T° min | T° max | Précip (mm) | Ensol (H)

----------------------------------------------------

01/2007 | 6,2 | 10,2 | 28,0 | 46,0

02/2007 | 6,2 | 11,6 | 79,2 | 76,4

03/2007 | 5,4 | 12,3 | 41,4 | 145,6

04/2007 | 10,5 | 21,7 | 5,2 | 312,4

05/2007 | 12,0 | 20,4 | 102,4 | 153,5

06/2007 | 14,7 | 23,0 | 52,6 | 146,0  
...

Pour cela :

* Copier le fichier de données dans le répertoire du projet. Son chemin pourra ainsi être spécifié dans le code de la façon suivante : «..\..\DonnéesMétéoParis.txt ». Il s’agit d’un chemin relatif par rapport à l’exécutable de l’application qui se trouve dans le répertoire bin\debug
* Lire son contenu en utilisant la méthode File.ReadAllLines
* Utiliser une classe RelevéMensuel pour modéliser les infos d’un relevé mensuel et les extraire d’une ligne de fichier.
* Utiliser la méthode ToString de RelevéMensuel pour générer chaque ligne à afficher

Le programme devra bien séparer l’interface visuelle de la logique métier.

NB/ Pour changer le jeu de caractères de la console, utiliser la ligne suivante :

Console.OutputEncoding = Encoding.UTF8;

# Listes de véhicules

En repartant de l’exercice sur les véhicules :

**Etape 1 : ajout d’une propriété Prix et d’une méthode de comparaison**

Sur la classe Véhicule :

* Ajouter une propriété Prix
* Ajouter un constructeur permettant d’initialiser le nom et le prix du véhicule
* Implémenter l’interface IComparable<T> et faire en sorte que la méthode CompareTo compare les véhicules selon leur prix

Sur les classes Voiture et Moto, ajouter un constructeur semblable à celui ajouté sur Véhicule.

**Etape 2 : création de véhicules**

Dans la fonction Main, créer les 4 véhicules suivants :

* Voiture Mégane à 19 000 €
* Moto Intruder à 13 000€
* Voiture Enzo à 380 000€
* Moto Yamaha XJR1300 à 11 000€

**Etape 3 : création de la première liste triée**

Instancier une liste triée générique nommée liste1 et ajouter dedans les 4 véhicules, en prenant comme clé le nom du véhicule et comme valeur l’instance de véhicule.

Afficher le contenu de la liste, chaque élément étant affiché sous la forme suivante : « Nom : prix ».

Selon quel ordre la liste est-elle triée ?

**Etape 4** : **création de la seconde liste triée**

Instancier une seconde liste triée générique nommée liste2 et **a**jouter dedans les 4 véhicules, en inversant la clé et la valeur par rapport à l’étape 2 (c’est-à-dire que la clé est désormais l’instance de véhicule, et la valeur son nom)

Afficher le contenu de la liste, chaque élément étant affiché sous la forme suivante : « Nom : prix ».

Selon quel ordre la liste est-elle triée ? Comment s’est fait ce tri ?

**Etape 5 : unicité**

Affecter un prix de de 13 000 € à la seconde moto et relancer l’appli. Que se passet-il ? Pourquoi ?

Remettre le prix initial

**Etape 6 : recherche**

Créer un tableau de chaînes initialisé avec les valeurs suivantes : Clio, Mégane, Golf, Enzo, Polo

Pour chaque élément de ce tableau, chercher le véhicule correspondant dans la liste1, et afficher son nom et son prix s’il est trouvé.