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# 3.0

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

[Calcul du PGCD 3](#_Toc480738696)

[Calcul de nombres premiers 3](#_Toc480738697)

[Comptage des voyelles et consonnes 4](#_Toc480738698)

[Tri d’un tableau 4](#_Toc480738699)

[Login 5](#_Toc480738700)

[Relevés météo 6](#_Toc480738701)

[Texte formaté 7](#_Toc480738702)

[Mois et saisons 7](#_Toc480738703)

[Statuts 8](#_Toc480738704)

[Boîtes 9](#_Toc480738705)

[Boîtes (suite et fin) 10](#_Toc480738706)

[Véhicules 11](#_Toc480738707)

[Chauffage programmable 13](#_Toc480738708)

[Tableau params 15](#_Toc480738709)

[Liste triée non générique 15](#_Toc480738710)

[Conteneurs génériques 16](#_Toc480738711)

[Cryptage 16](#_Toc480738712)

[Jeu de roulette 17](#_Toc480738713)

[Création d’un délégué 20](#_Toc480738714)

[Explorateur de fichiers 21](#_Toc480738715)

[Analyseur LINQ 22](#_Toc480738716)

[Méthodes d’extension 22](#_Toc480738717)

[Gestion d’un stock 23](#_Toc480738718)

[Collection de bandes dessinées 24](#_Toc480738719)

# Calcul du PGCD

**Objectifs** : se familiariser avec la syntaxe C#, les opérateurs et structures de contrôle, la saisie et l’affichage sur la console

On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en C# à partir de l’algorithme de la méthode dite « égyptienne ». Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) p et q, selon cette méthode :

Lire (p, q);

Tantque p ≠ q faire

Si p > q alors

p ← p – q

sinon

q ← q – p

FinSi

FinTant;

Ecrire(" PGCD = ", p)

Ecrivez le programme C# complet qui produise le résultat suivant :

Entrez le premier nombre : 21

Entrez le deuxième nombre : 45

Le PGCD de 21 et 45 est : 3

# Calcul de nombres premiers

**Objectifs** : se familiariser avec la syntaxe C#, les opérateurs et structures de contrôle, la saisie et l’affichage sur la console.

En utilisant des boucles (do) while, écrire un programme de calcul et d'affichage des N premiers nombres premiers. Un nombre entier est premier s’il n’est divisible que par 1 et par lui-même (ex : 2, 3, 5, 7, 11, 13…)

Spécifications de l’algorithme : (on étudie la primalité des nombres uniquement impairs)

Algorithme Premier

Entrée: n ∈ N

Sortie: nbr ∈ N

Local: EstPremier ∈ {Vrai , Faux}

divis,compt ∈N²;

début

lire(n);

compt ← 1;

ecrire(2);

nbr ← 3;

Tantque(compt < n) Faire

divis ← 3;

EstPremier ← Vrai;

Répéter

Si reste(nbr par divis) = 0 Alors

EstPremier ← Faux

Sinon

divis ← divis+2

Fsi

jusquà (divis > nbr / 2)ou (EstPremier=Faux);

Si EstPremier = Vrai Alors

ecrire(nbr);

compt ← compt+1

Fsi;

nbr ← nbr+1

Ftant

FinPremier

Ecrivez le programme C# complet qui produise le résultat suivant :

Combien de nombres premiers : 5

2

3

5

7

11

# Comptage des voyelles et consonnes

**Objectifs** : se familiariser avec les bases du langage C# : 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 vide qui prend en entrée un mot, calcule ses nombres de voyelles et consonnes, et les renvoie en paramètres out.
* Afficher le résultat de l’appel de cette fonction sous la forme : « ”livre” comprend 3 consonnes et 2 voyelles »

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

# Tri d’un tableau

**Etape 1** : Créer une fonction permettant d’afficher le contenu d’un tableau (sur une seule ligne).

**Etape 2** : Créer une fonction permettant de trier par ordre alphabétique les éléments d’un tableau de mots passé en paramètre.

Indications :

* Pour trier un tableau, on peut comparer ses éléments deux à deux, et les permuter jusqu’à ce qu’on arrive à parcourir le tableau sans faire aucune permutation.
* Pour comparer une chaîne avec une autre, on peut utiliser la méthode CompareTo(). Exemple : mot1.CompareTo(mot2) ;

**Etape 3** : tester la tri grâce à la première fonction

**Etape 4** : modifier le type du paramètre d’entrée de la fonction en IComparable[]. Que constatez-vous ?

**Etape 5** : faire en sorte que la fonction ne modifie pas le tableau passé en paramètre, mais renvoie plutôt le tableau trié en résultat. Tester avec la fonction d’affichage.

# Login

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

**Etape 1** : Pour créer un compte, demander à l’utilisateur de saisir successivement un login puis un mot de passe. Afficher ensuite un message « Votre compte a bien été créé. Un message vient de vous être envoyé »

**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
* Lettre minuscules : codes 97 à 122

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

**Etape 3** : Dans le code de l’étape 1, appeler la méthode de vérification du login. Intercepter l’exception sur le format et afficher le message correspondant.

**Etape 4** : Faire en sorte que la demande de login soit répétée tant qu’un login correct n’a pas été saisi

**Etape 5** : appliquer les 2 dernières étapes à la gestion du mot de passe. Le mot de passe ne doit être demandé que si la saisie du login est correcte.

**Etape 6** : Imaginons que la saisie du mot de passe soit faite en mode silencieux (sans affichage des caractères à l’écran). Quoi qu’il arrive, on veut revenir en mode normal après saisie du mot de passe. Que faudrait-il faire ?

Pour vérifier que votre solution fonctionne :

* Simulez l’entrée et la sortie du mode silencieux en affichant de simples messages « Entrée en / Sortie du mode silencieux »
* Levez arbitrairement une exception InvalidOperationException dans la méthode de vérification du mot de passe, et gérez-là dans la méthode Main. Vérifiez que le message de sortie du mode silencieux s’affiche bien.

# Relevés météo

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

* Passer des paramètres en ref, out
* Gérer des exceptions
* Utiliser l’instruction using
* Lire un fichier texte (pas abordé dans le cours, mais simple à faire)

## Description du programme

Le fichier ci-joint contient les relevés mensuels de températures min et max de la ville de Paris sur les 10 dernières années :



NB/ Données issues du [site de Météo France](http://www.meteofrance.com/climat/france/paris/75114001/releves).

Le fichier contient une ligne d’en-tête et des lignes de données. Les valeurs sont séparées par des **tabulations**. On suppose que les seules erreurs que le fichier peut contenir sont un format de date ou de nombre incorrect (ex : lettres au lieu de chiffres).

Le programme doit :

* Demander à l’utilisateur de saisir le chemin complet du fichier
* Lire le fichier ligne à ligne s’il existe, ou bien afficher un message d’erreur dans le cas contraire
* Afficher à l’écran les informations de synthèse suivantes :
* Mois et valeur de la température min la plus basse
* Mois et valeur de la température max la plus élevée

Exemple :

Mois le plus froid : … avec … °C

Mois le plus chaud : … avec … °C

Si une des lignes de relevé présente une erreur de format de date ou nombre, le programme doit signaler l’erreur en affichant le N° de la ligne et son texte, et poursuivre le traitement des lignes suivantes.

Le programme doit comporter :

* Une fonction AnalyserLigne qui prend en entrée une ligne de texte, et en extrait les valeurs de date et températures.
* Une fonction AnalyserFichier qui gère le parcours du fichier et qui sera appelée par la fonction Main.

/!\ L’affichage d’informations à l’écran doit être fait uniquement dans la fonction Main, ce qui permettrait de réutiliser les 2 autres fonctions dans différents types d’applications, pas seulement console.

Le format d’affichage du mois n’a pas d’importance.

## Indications

* La lecture d’un fichier texte se fait avec un StreamReader (regarder l’aide en ligne pour voir comment l’utiliser)
* Pour isoler des morceaux de chaîne de caractère, utiliser les méthodes IndexOf et Substring du type string, à titre d’entrainement (car il y a en fait une méthode plus simple)
* Pour interpréter une chaîne en date, utiliser la méthode DateTime.Parse.
* Pour interpréter une chaîne en nombre réel, utiliser la méthode double.Parse
* Pour tester les exceptions, introduisez des erreurs sur les dates et nombres de quelques lignes du fichier.

Bonus si vous avez terminé en avance :

* Calculer et afficher les moyennes annuelles des températures max. Ex :
  + 2007 : 17.5°C
  + 2008 : 18.1°C
  + …

# Texte formaté

**Objectifs** : apprendre à utiliser les chaînes de formats

Afficher le texte suivant à l’écran, en respectant exactement les formats :

Hier, Bryan a gagné $148.00 à la roulette en 02 heures et 25 minutes.

Durant l'hiver, c'est à dire depuis le 21 Dec 2016, il a gagné $1,234.

… et en utilisant les variables suivantes :

enum Saisons { hiver, printemps, été, automne };

decimal gain = 148m;

TimeSpan durée = new TimeSpan(2, 25, 46);

DateTime dateDeb = new DateTime(2016, 12, 21);

Saisons saison = Saisons.hiver;

decimal total = 1234;

# Mois et saisons

**Objectifs** : apprendre à manipuler les énumérations de type flag.

**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.

Ex : le mois de décembre appartient à la fois à l’automne et à l’hiver.

**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

# Statuts

**Objectifs** : apprendre à manipuler les énumérations de type flag, et utiliser les listes génériques

Dans une entreprise, une personne peut avoir les statuts suivants :  
Salarié en CDI, salarié en CDD, délégué du personnel, membre du CHSCT, , représentant syndical.

Ces statuts ne sont pas exclusifs, c’est-à-dire qu'une personne peut avoir plusieurs statuts à la fois.

**Etape 1** : créer une classe Personne avec des propriétés Nom, Prénom et Statut, ainsi qu’un constructeur permettant d’initialiser ces 3 propriétés.

**Etape 2** : sur la classe Personne, redéfinir la méthode ToString pour qu’elle renvoie une chaîne contenant le nom, le prénom et les statuts de la personne.

**Etape 3**: dans Main, créer une liste contenant les personnes avec les statuts suivants :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Nom** | **Prénom** | **CDI** | **CDD** | **DP** | **CHSCT** | **SYND** |
| TURPIN | Abel | 1 | 0 | 0 | 0 | 0 |
| BONNEAU | Achille | 0 | 1 | 1 | 0 | 0 |
| BLONDEL | Adelphe | 1 | 0 | 1 | 1 | 1 |
| BLACK | Aimé | 1 | 0 | 0 | 0 | 0 |
| PERRIER | Aimée | 1 | 0 | 0 | 0 | 0 |
| JORDAN | Alain | 0 | 1 | 0 | 1 | 0 |
| BAUDRY | Alban | 0 | 1 | 0 | 0 | 0 |
| ORLEANS | Albert | 1 | 0 | 1 | 0 | 1 |
| VALOIS | Alexandra | 1 | 0 | 0 | 0 | 1 |
| WEST | Alexandre | 1 | 0 | 1 | 1 | 0 |

**Etape 4**: parcourir la liste des personnes, et en utilisant des masques, déterminer les personnes qui sont :

* A la fois en CDD **et** membre du CHSCT (+ éventuellement d’autres fonctions)
* A la fois en CDI **et** DP (+ éventuellement d’autres fonctions)

Copier ces personnes dans 2 autres listes distinctes

**Etape 5** : afficher le détail des personnes de la seconde liste en utilisant la méthode ToString redéfinie à l’étape 2

**Etape 6** : pour ces personnes, leur affecter en plus le statut de membre du CHSCT, et afficher de nouveau le détail pour vérifier.

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

**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.

# Boîtes (suite et fin)

Dans une boîte, on peut ajouter ou enlever des articles. Un article est décrit par son id (de type guid, généré automatiquement), son libellé, son volume et son poids. Seul le libellé est modifiable après création de l’article.

Une boîte peut contenir plusieurs articles si son volume le permet (on ne tient pas compte de la forme des articles). Toutefois, le poids total ne doit pas dépasser 10 kg (valeur qui doit être modifiable de façon centralisée). Lever une exception si c’est le cas.   
NB/ on peut utiliser la même exception pour le cas où la boîte est trop petite pour contenir les articles, et pour le cas où le poids total dépasse 10kg

Les boîtes sont numérotées automatiquement dans l’ordre de leur création

On doit pouvoir afficher de façon synthétique le contenu d’une boîte. Exemple :

La boîte 3 contient 2 articles :

Code : "45E510BD-B1D8-403A-8AE4-90C65D8C2395"  
La cape de Zorro  
300g, 1240cm3

Code : "A0B24B09-DB15-4DE9-A91B-73AE329F32B4"  
Transformers Megatron  
280g, 900cm3

L’utilisation de WriteLine doit se faire exclusivement dans la fonction Main.

Bonus pour les plus rapides : On doit pouvoir transférer les articles d’une boîte vers une autre, contenant éventuellement déjà des articles. Si le volume disponible dans cette dernière n’est pas suffisant ou si on dépasse le poids maxi, lever une exception. On transfère soit tous les articles, soit aucun.

**Indications :**

* Pour générer un Guid, utiliser la méthode statique Guid.NewGuid()
* Pour mémoriser les articles d’une boîte, utiliser une collection de type List<Article> instanciée en même temps que la boîte. Elle se manipule comme un tableau, et possède des méthodes Add et Remove pour ajouter ou enlever des éléments.
* Dans n’importe quelle classe, on peut surcharger la méthode ToString du type Object (utiliser le mot clé override).
* La fonction Main ne doit contenir que le code de test
* Utiliser des exceptions de type InvalidOperationException

**Scénario de test :**

* Créer 3 articles de volumes 64, 125 et 216
* Essayer de les mettre dans une boîte de volume 343 (7x7x7) 🡪 un message doit s’afficher. Vérifier que la boîte contient tout de même les 2 premiers articles
* Essayer d’ajouter un article lourd, de sorte que le poids total excède 10kg 🡪 un message doit s’afficher.
* Afficher le détail du contenu de la boîte. Retirer le premier article et modifier le libellé du second. Puis afficher de nouveau le contenu de la boîte pour vérifier que tout est bon
* Transférer les articles dans une nouvelle boîte et afficher le contenu de la boîte d’origine et de la boîte de destination

# 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

**Etape 5** : rendre la classe Véhicule abstraite et ajouter une méthode abstraite void CalculerConso

Noter que VS souligne Voiture et Moto en rouge pour signaler que la méthode CalculerConso n’est pas implémentée.

Cliquer sur le nom puis sur l’info-bulle pour implémenter cette méthode dans chaque classe. On se contentera de lever une exception car implémenter cette méthode avec du code plus réaliste nécessiterait un scénario d’exercice beaucoup plus poussé.

Dans Main, essayer d’instancier Véhicule. VS souligne la ligne en rouge car cette classe est abstraite.

**Etape 6** : Ajouter une propriété PRK (Prix de revient kilométrique) abstraite en lecture seule sur Véhicule

Fournir une implémentation dans les classes dérivées (renvoyer simplement des valeurs de PRK arbitraires).

**Etape 7** : Faire dériver Véhicule de IComparable. Dans l’info-bulle VS propose de l’implémenter de différentes façons. Choisir de l’implémenter explicitement.

Dans la méthode CompareTo ajouter le code suivant :

return PRK.CompareTo(((Vehicule)obj).PRK);

En effet, le type Double implémente IComparable, donc autant s’en servir.

Dans Main, afficher le résultat de la comparaison des 2 véhicules (voiture et moto) créés précédemment. On voit que pour appeler la méthode CompareTo, il faut utiliser des variables de type IComparable et non Vehicule.

Jouer avec les valeurs des PRK pour tester les 3 cas possibles.

**Etape 9** : en repartant des variables de type IComparable, utiliser is sur l’une d’elles avant de la transtyper en Véhicule, puis afficher sa propriété Description.

if (zoé is Véhicule) Console.WriteLine(((Véhicule)zoé).Description);

**Etape 10** : Dans la classe Program, créer une fonction Décrire, prenant en paramètre un objet, et affichant :

* Le nom de son type
* Le nom du type dont il hérite
* Le nom des interfaces qu’il implémente
* La liste de ses membres (type et nom)

Dans Main, appeler cette fonction en lui passant successivement 2 véhicules de types différents, et vérifier le résultat.

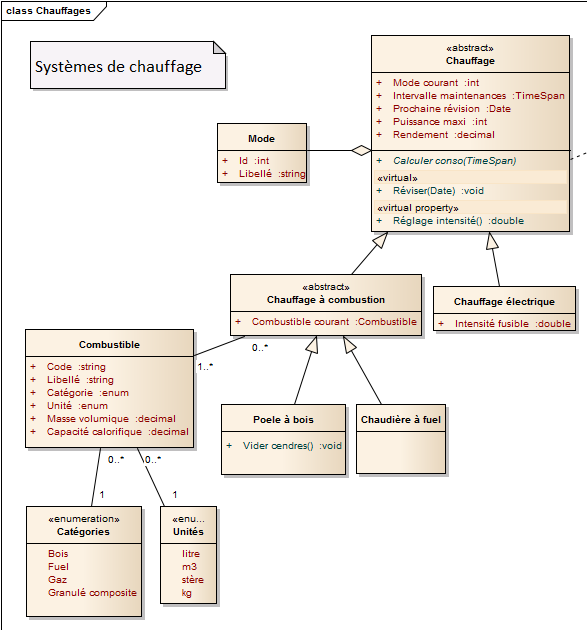
# Chauffage programmable

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

* Héritage, hiérarchie de classes
* Classes abstraites et membres abstraits
* Interfaces
* Intervalles de temps TimeSpan

## Description du programme

Le programme mettra en œuvre l’arborescence de systèmes de chauffage suivante :



Il doit également permettre de rendre n’importe quel chauffage programmable. Pour cela, il devra mettre en œuvre l’interface IProgrammable décrite ci-dessous :



Description des méthodes et propriétés de IProgrammable :

* PlagesProgramme : obtient la liste des plages de fonctionnement qui constituent le programme. Chaque plage est caractérisée par une heure de début, une heure de fin et un mode de fonctionnement durant cette période
* ProgrammeValide : Vérifie que les plages du programme couvrent bien 24h et ne se chevauchent pas
* GetModeProgrammé : renvoie le mode programmé pour une heure donnée
* PlageProgrammeCourante : obtient la plage de fonctionnement courante du système, c’est-à-dire celle dont la tranche horaire couvre l’heure courante
* HeureChangementMode : obtient l’heure du prochain changement de mode
* ProgrammeActif : obtient ou définit si le programme est actif ou non

NB/ IComparable est déjà fournie par le .Net framework. Elle permettra de comparer les heures de début des plages de fonctionnement

## A faire

Créer l’arborescence de classes et d’interfaces décrite par les diagrammes précédents, en utilisant des propriétés à la place des champs publics (principe d’encapsulation).

* On pourra utiliser des propriétés génériques de type List<T> pour stocker les plages et les modes.
* Inutile d’implémenter la méthode CalculerConso ; renvoyer une NotImplementedException.

Faire en sorte que la classe Chauffage hérite de l’interface IProgrammable, et propose une implémentation par défaut de cette interface, avec les règles de gestion suivantes :

* Lorsque ProgrammeActif passe à vrai, exécuter ProgrammeValide et renvoyer une exception si elle renvoie faux
* Lorsqu’on modifie ModeCourant, renvoyer des messages d’erreurs explicites dans les cas suivants :
  + ProgrammeActif est à vrai. En effet, lorsqu’un programme est actif, on ne doit pas pouvoir modifier manuellement le mode de fonctionnement du chauffage.
  + La valeur de mode qu’on essaie d’affecter n’est pas dans la liste des modes du chauffage
* La programmation se fait à la minute près. Dans ProgrammeValide, mettre à 0 la partie des secondes sur les heures de chaque plage.

Remarque : garder l’implémentation de ProgrammeValide et HeureChangementMode pour la fin (ce n’est pas le plus important dans cet exercice).

Utiliser la classe Program suivante pour tester le programme :



# Tableau params

Reprendre le projet sur les véhicules.

Sur la classe Véhicule :

* Ajouter une propriété Prix
* Ajouter un constructeur permettant d’initialiser le nom et le prix du véhicule.
* Modifier la méthode CompareTo pour que la comparaison des véhicules porte sur le prix (attention au type d’implémentation de cette méthode d’interface).
* Ajouter une méthode « LePlusCher » prenant un nombre quelconque de véhicules en paramètre et renvoyant le plus cher d’entre eux.

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

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€

Appeler la méthode LePluCher en lui passant ces 4 véhicules et afficher le nom du véhicule le plus cher.

# Liste triée non générique

En repartant de l’exercice précédent :

**Etape 1** : Instancier une liste triée non générique nommée liste1.

**Etape 2** : Ajouter dedans les 4 véhicules, en prenant comme clé le nom du véhicule et comme valeur l’instance de véhicule.

**Etape 3** : 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** : Instancier une seconde liste triée non générique nommée liste2.

**Etape 5** : Ajouter 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)

**Etape 6** : 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 ?

# Conteneurs génériques

En repartant de l’exercice précédent :

**Etape 1** : Instancier une liste triée générique nommée liste3.

**Etape 2** : Ajouter dedans les 4 véhicules, en prenant comme clé l’instance de véhicule, et comme valeur son nom

**Etape 3** : Afficher le contenu de la liste, chaque élément étant affiché sous la forme suivante : « Nom : prix ».

**Etape 4** : transférer le contenu de celle liste dans un dictionnaire générique de type Dictionary<string, Véhicule>, nommé dico. Liste3 doit être vidée.

**Etape 5** : créer un tableau de chaînes initialisé avec les valeurs suivantes : Clio, Mégane, Golf, Enzo, Polo

Pour chaque élément de ce tableau, chercher sa correspondance dans le dictionnaire dico, et s’il est trouvé, afficher son nom et son prix.

# Cryptage

**Objectifs** : mettre en œuvre des collections génériques

Faire une classe statique Cryptage avec 3 méthodes pour :

* Charger une clé de cryptage décrite par le fichier ci-dessous
* Crypter un texte avec une clé
* Décrypter un texte avec une clé

La clé consiste à associer chaque lettre de l’alphabet à une autre. Les caractères autres que des lettres ne seront pas cryptés.



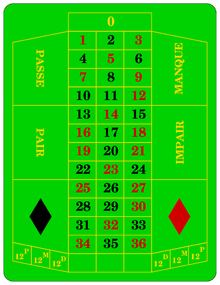
# Jeu de roulette

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

* Classes, champs, propriétés, méthodes
* Constructeurs
* Modificateurs d’accès
* Types énumérés
* Composition/agrégation
* Membres statiques

## Description du jeu

Une table de roulette ressemble à l’image suivante :



Pour simplifier, on suppose que le joueur ne peut faire que les types de mises suivants :

|  |  |
| --- | --- |
| **Type de mise** | **Gain** |
| Un des 24 premiers numéros (1 à 24) | 0.5 x la mise  (arrondi au nombre <) |
| Un des 24 derniers numéros (13 à 36) | 0.5 x la mise  (arrondi au nombre <) |
| Couleur rouge | 1 x la mise |
| Couleur noire | 1 x la mise |
| Numéro impair | 1 x la mise |
| Numéro pair | 1 x la mise |
| Un numéro précis | 35 x la mise |

Ex : le joueur mise 4 jetons sur le fait que la bille tombe sur un des 24 derniers numéros. Si la bille tombe sur un de ces numéros, le joueur récupère sa mise + 2 jetons, sinon, il perd sa mise.

On ne peut pas miser sur le 0. Le jeu s’arrête si le joueur n’a plus de jetons.

**Déroulement du jeu :**

Demander d’abord à l’utilisateur de saisir le nombre de jetons achetés au début de la partie.

Puis, pour chaque tour :

* Demander à l’utilisateur de saisir une mise (combinaison et nombre de jetons)
* Simuler le lancement de la bille sur la roulette en générant un nombre aléatoire compris entre 0 et 36 et afficher le résultat du lancé
* Calculer et afficher le gain ou la perte, ainsi que le nombre total de jetons restants
* Demander si on veut continuer à jouer
* Vider l’écran

A la fin, afficher un message « Merci d’avoir joué ».

Si l’utilisateur saisit une information non valide, elle est ignorée, et le programme attend une nouvelle saisie (sans poser de nouveau la question).

**Exemple de déroulement du programme :**

NB/ Les réponses de l’utilisateur sont en bleu

Combien de jetons avez-vous achetés ?

50

Quelle combinaison choisissez-vous ?

24p/24d : 24 premiers ou derniers numéros

r/n : Couleur rouge ou noire

i/p : Numéro impair ou pair

x : Un numéro précis

i

Combien de jetons misez-vous (max : 50) ?

8

La bille est tombée sur le N° 30, qui est rouge et pair.

Vous perdez 8 jetons. Vous possédez désormais 42 jetons.

Voulez-vous continuer (O/N) ?

O

[Vider l’écran]

Quelle combinaison choisissez-vous ?

24p/24d : 24 premiers ou derniers numéros

r/n : Couleur rouge ou noire

i/p : Numéro impair ou pair

x : Un numéro précis

24p

Combien de jetons misez-vous (max : 42) ?

20

La bille est tombée sur le N° 7, qui est rouge et impair.

Vous gagnez 10 jetons. Vous possédez désormais 52 jetons.

Voulez-vous continuer (O/N) ?

O

[Vider l’écran]

Quelle combinaison choisissez-vous ?

24p/24d : 24 premiers ou derniers numéros

r/n : Couleur rouge ou noire

i/p : Numéro impair ou pair

x : Un numéro précis

X

Choisissez un numéro compris entre 1 et 36 :

24

Combien de jetons misez-vous (max : 54) ?

52

La bille est tombée sur le N° 21, qui est rouge et impair.

Vous perdez 52 jetons. Il ne vous reste plus aucun jeton.

La partie est terminée.

Merci d’avoir joué

## A faire

En partant du diagramme de clases ci-dessous et du projet C# qui vous sera fourni, vous devez compléter le code pour implémenter le jeu décrit précédemment. Tout l’affichage doit être fait dans la classe Jeu.



## Indications

Pour le contrôle des saisies, s’inspirer du code de la méthode SaisirNbJetonsInit.

Pour générer un nombre aléatoire, il suffit de créer un objet de type Random et d’utiliser sa méthode Next().

## Bonus

A la fin du jeu, afficher des statistiques, comme ceci :

2 mises réalisées, dont 1 gagnante(s) et 1 perdante(s)

Nombre de jetons initial : 50

Nombre de jetons final : 51 (+1)

Merci d’avoir joué.

Afficher la valeur du compteur de mises au début de chaque nouvelle mise :

Mise 3 – Quelle combinaison choisissez-vous ?

# Création d’un délégué

**Objectifs** : savoir créer et utiliser un délégué

En repartant de l’exercice des véhicules (il suffit d’avoir la classe ancêtre Vehicule) :

**Etape 1** : A côté de la classe Vehicule, créer un nouveau type délégué nommé DelegueEntretien prenant en paramètre un véhicule.

**Etape 2** : dans la classe Véhicule ajouter une propriété CarnetEntretien en lecture seule. Ce carnet contiendra les dates et descriptions des opérations d’entretien réalisées au cours de la vie du véhicule. Plusieurs opérations peuvent avoir lieu à la même date.

**Etape 4** : dans la classe Véhicule, ajouter une méthode Entretenir, prenant en paramètre une date, et un délégué du type déclaré plus haut. Dans cette méthode :

* Ajouter une entrée vide dans le carnet d’entretien à la date reçue en paramètre
* Appliquer la méthode déléguée à l’instance de véhicule courante

**Etape 3** : Dans la classe Program, créer les méthodes suivantes qui devront être compatibles avec le délégué DelegueEntretien :

* ChangerPneus
* Vidanger
* RetoucherPeinture

Chacune d’elle ajoutera la description de l’opération effectuée dans la dernière entrée du carnet d’entretien du véhicule.

Conseil : ajouter un using sur System.Linq, si ce n’est pas déjà fait…

**Etape 5** : Dans la méthode Main :

* Créer un délégué du type défini plus haut
* Brancher dessus les 3 méthodes créées précédemment

**Etape 6** : Appeler la méthode Entretenir sur un véhicule de votre choix (moto ou voiture), et afficher le contenu de son carnet d’entretien. Exemple de résultat :

Entretien du véhicule Mégane du 23/04/2017 :

- Changement des pneus

- Vidange

- Retouche de peinture

# Explorateur de fichiers

**Objectifs** : savoir créer et utiliser un délégué

On souhaite analyser les fichiers d’un dossier dont le chemin est saisi par l’utilisateur. On veut pouvoir afficher :

* Le nombre total de fichiers, et le nombre de fichiers .cs
* Les noms des fichiers le plus court et le plus long
* La liste des noms des fichiers projet C# (sans l’extension .csproj)

Exemple de résultat :

Saisissez le chemin du dossier à explorer :

D:\Temp\Dev\Console

646 fichiers, dont 154 fichiers .cs

Nom de fichier le plus long : Microsoft.VisualStudio.TestPlatform.MSTestAdapter.PlatformServices.Interface.dll

Fichiers projets C# :

JobOverview

TestUnitaire

Boites

EssaisCS

POO

TestsUnitaires

Job Overview

UnitTest

**Etape 1** : créer un nouveau projet console nommé ExplorateurFichiers

**Etape 2** : création de l’explorateur :

* Créer une classe Explorateur dans un nouveau fichier
* A côté de cette classe, déclarer un type délégué DelegueExplorateur prenant un paramètre de type FileInfo (classe décrivant un fichier)

A la classe Explorateur, ajouter une méthode statique Explorer qui prend en paramètre :

* Le chemin d'un dossier
* Un délégué de type DelegueExplorateur

**Etape 3** : Dans la méthode Explorer, utiliser la classe DirectoryInfo et sa méthode EnumerateFiles pour parcourir les fichiers du dossier passé en paramètre, ainsi que ses sous-dossiers de façon récursive, et pour chaque fichier rencontré, exécuter le délégué

**Etape 4** : Créer une classe Analyseur qui sera chargée de l’analyse des fichiers parcourus par l’explorateur.

Dans cette classe, créer les propriétés nécessaires pour stocker les informations qu’on souhaite obtenir.

**Etape 5** : ajouter une méthode AnalyserDossier, prenant en paramètre le chemin du dossier à analyser. Cette méthode doit lancer l’exploration du dossier à l’aide de l’explorateur, en lui passant un délégué de type DelegueExplorateur.

**Etape 6** : Créer les méthodes nécessaires à l’analyse des fichiers, et les brancher sur le délégué :

* CompterFichiers : chargée de compter les fichiers, en distinguant les fichiers .cs
* AnalyserNom : chargée d’analyser les noms des fichiers et d’isoler le plus grand
* FiltrerProjet : chargée de filtrer les fichiers csproj et de mémoriser leur nom dans une collection

**Etape 7** : Dans la méthode Main :

* Faire saisir et récupérer le chemin du dossier à explorer
* Instancier un Analyseur et appeler sa méthode Analyse en lui passant le chemin du dossier
* Afficher les informations récupérées par l’analyseur

**Etape 8** : Faire en sorte de demander à nouveau la saisie du chemin du dossier, si le chemin saisi n’était pas valide.

# Analyseur LINQ

**Objectifs** : construire des requêtes LINQ mettant en œuvre les principales techniques

Le fichier suivant fournit le code d’une classe permettant de charger et d’analyser les données météo de la ville de Paris (données fournies dans le fichier texte d’un précédent exercice) :



Vous devez compléter la méthode AfficherStats en écrivant les requêtes Linq permettant d’obtenir les statistiques décrites en commentaires.

Dans la méthode Main, ajouter le code nécessaire pour instancier et utiliser l’analyseur.

# Méthodes d’extension

Sur la classe DateTime, créer les méthodes d’extension suivante :

* IsWeekEnd pour déterminer si une date correspond à un jour de week-end
* GetLastDayOfMonth (qui renvoie un DateTime) pour récupérer le dernier jour du mois d’une date passée en paramètre.
* IsInRange pour déterminer si une date est dans un intervalle donné (la fonction prend donc 3 paramètres)

Autres exemples de méthodes d’extensions sur la classe DateTime sur cette page : <http://www.extensionmethod.net/csharp/datetime>

# Gestion d’un stock

**Objectifs** : utiliser les collections génériques, créer un évènement, créer et utiliser des méthodes anonymes.

## Description du programme

Le diagramme de classes suivant propose une modélisation de gestion d’un stock :



Un mouvement de stock est caractérisé par :

* Son type : soit une entrée (on alimente le stock), soit une sortie (on puise dans le stock), soit une remise à zéro du stock
* Sa date
* Sa quantité

Méthodes et propriétés de la classe **Stock** :

* **Ajouter** : ajoute la quantité spécifiée au stock, à la date spécifiée. Cette méthode lève une System.ArgumentException s'il y a déjà un mouvement de stock à la même date
* **Retirer** : retire la quantité spécifiée du stock à la date spécifiée.  
  Lève une System.ArgumentException s'il y a déjà un mouvement de stock à la même date  
  Lève une InvalidOperationException si la quantité en stock est insuffisante
* **GetEtatStock** : Obtient l'état du stock, c’est-à-dire la quantité restante à une date donnée (pas forcément la date du jour
* **SeuilAlerte** : valeur de stock au-dessous de laquelle la méthode Retirer émet un évènement pour prévenir l’utilisateur.
* **AlerteStockBas** décrit l’événement qui est déclenché par la méthode Retirer, lorsque le stock restant est inférieur au seuil d’alerte.

**DateAndDecimalEventArgs** est la classe qui modélise le paramètre passé au gestionnaire de l’événement AlerteStockBas.

Les mouvements de stock sont agrégés par la classe Stock dans une collection privée, et doivent toujours rester **ordonnés** selon la date de mouvement, même si ces derniers ne sont pas insérés dans l’ordre.

## A faire

Créer le code correspondant au diagramme de classes.

Dans la méthode Main, et dans cet ordre :

* Créer un stock vide et définir son seuil d’alerte à 50kg
* S’abonner à l’évènement AlerteStockBas pour afficher le message suivant : « Attention, au …, il ne reste que … kg en stock ! »
* Créer les mouvements de stock à partir des enregistrements fournis dans le fichier ci-joint (l’application devra demander le chemin du fichier)
* S’il y a 2 mouvements à la même date, afficher un message « Erreur : il y a déjà un mouvement de stock au … », et poursuivre la création des mouvements suivants. Inutile de gérer d’éventuelles erreurs de format de données.
* Afficher le nombre de mouvements de stock créés.
* Afficher l’état du stock au 1er jour de chaque mois de l’année 2016 : « Etat du stock au … : … kg ».

# Collection de bandes dessinées

**Objectifs** : sérialiser un fichier xml en liste d’entités, utiliser la notion d’attribut

Le fichier ci-dessous décrit des collections de bandes dessinées :



Il s’agit de charger le contenu de ce fichier dans une liste mémoire, en utilisant les attributs définis par le .net framework pour la sérialisation xml.

Le chargement doit être fait dans une classe statique nommée BD\_DAL, comportant une seule méthode ChargerCollectionsBD, qui renvoie une List<CollectionBD>.

Créer dans le même fichier toutes les classes que vous jugerez nécessaires pour charger les données par sérialisation xml. Ces classes ne doivent comporter aucune méthode.