Devoir #2 — INF600A (gr. 20)

Automne 2018

Date de remise: jeudi 22 novembre, 12h00. Un travail remis *après* l'heure indiquée sera considéré **en retard** : pénalité de 10 % par jour, **partiel** ou complet.

Aucun travail ne sera accepté après lundi 26 novembre, 10h00.

Script `Ruby` pour gestion d'une cave à vins

1 Objectif du travail : Familiarisation avec Ruby et avec le style fonctionnel en Ruby

> *Learn to use* `Enumerable`*. You will not be a rubyist until you do.*
>
> *«Ruby QuickRef», R. Davis (*`http://www.zenspider.com/Languages/Ruby/QuickRef.html`*)*

Le but de ce travail est de vous familiariser avec l'utilisation du langage Ruby, notamment avec l'utilisation des méthodes du module `Enumerable` — donc avec le sous-ensemble de Ruby qui favorise l'utilisation du style **fonctionnel** de programmation.

2 Ce que vous devez faire

Pour ce travail, vous devez développer un script `gv.rb` qui permet, comme dans le devoir 1, de gérer une cave à vins. Un squelette pour ce script et des fichiers auxiliaires (classe et module) vous sont fournis **et vous devez les compléter** :

```
git clone http://www.labunix.uqam.ca/~tremblay/git/GestionVinsRuby.git
```

NOTE : La première chose à faire une fois ce dépôt cloné est d'exécuter (une seule fois) la commande `make perms` pour **que `gv.rb` soit exécutable** :

```
$ cd GestionVinsRuby
$ make perms
```

Contraintes à respecter ⇒ Utilisation du style fonctionnel!

- Vous ne devez pas modifier le programme principal — le répartiteur (*dispatcher*) qui identifie la commande et effectue l'appel à la méthode qui traite la commande. Vous devez compléter les méthodes partiellement définies... ou ajouter d'autres méthodes.

- Les diverses méthodes qui mettent en oeuvre les commandes — méthodes qui reçoivent en argument le tableau `les_vins` — **doivent être mises en oeuvre dans un style fonctionnel**.

  Vous devez donc utiliser des méthodes telles que `map`, `reduce`, `select`, `reject`, `sort`, etc. — c'est-à-dire les méthodes exportées par le module `Enumerable` **qui ne modifient pas la collection**. *Vous ne devez donc pas utiliser les versions* **mutables** *des méthodes* — telles que `map!`, `select!`, etc. — ni les méthodes telles que `delete`, `delete_if`, `delete_at`, etc.

- Dans l'utilisation du style fonctionnel, les méthodes `map` (`collect`), `select` (`find_all`), `reject`, etc., doivent être utilisées **pour produire une nouvelle collection**, et non pour générer des effets de bord.

  Voici un exemple **de code à ne pas écrire** lorsque vous utilisez le style fonctionnel, où la méthode `map` est utilisée comme un `each`, donc pour son effet de bord, sans que le résultat retourné par `map` ne soit utilisé :

  ```
  res = []
  a.map { |x| res << foo(x) } # UTILISATION INAPPROPRIEE!
  ```

- Sauf quelques cas[1] — notamment, lorsqu'aucun résultat n'a à être retourné — vous ne devriez pas avoir besoin d'utiliser très les méthodes `each` et `while`, qui favorisent plutôt le style impératif. (Voir aussi remarques plus bas.)

[1] Par exemples : la méthode `Vin#to_s` (avec un `format` en argument optionnel), pour laquelle le traitement associé avec `while` classique peut être plus simple ; le traitement dans un ordre **arbitraire** des options de certaines commandes.

Les commandes acceptées par le script

La liste des commandes acceptées par le script est la même que dans le devoir 1. Par contre, certaines commandes comportent **des fonctionnalités additionnelles**. Dans ce qui suit, ces fonctionnalités seront illustrées en supposant que la base de données des vins (textuelle, comme dans le devoir 1) contient les vins suivants :

```
$ cat .vins.txt
1:10/06/18:rouge:Chianti Classico:2015:Volpaia:26.65:4:Fonce, dense, opaque. Aro
2:10/06/18:rouge:Chianti Classico:2014:Volpaia:26.65::
4:03/07/18:blanc:Alsace:2016:Pfaff:16.50::
5:03/07/18:rose:Cotes de Provence:2017:Roseline:18.50:3:Frais, leger.

$ ./gv.rb lister
1 [rouge - 26.65$]: Chianti Classico 2015, Volpaia (10/06/18) => 4 {Fonce, dense
2 [rouge - 26.65$]: Chianti Classico 2014, Volpaia (10/06/18) =>  {}
4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) =>  {}
5 [rose - 18.50$]: Cotes de Provence 2017, Roseline (03/07/18) => 3 {Frais, lege
```

Les nouvelles fonctionnalités

`lister`

Comme dans le devoir 1, il est possible de spécifier un `format` d'affichage des vins, tel que présenté dans la Figure 1. Toutefois, on peut aussi spécifier une **largeur de champ**, positive (justification à droite) ou négative (justification à gauche). Et, dans le cas du `prix` (`%P`), on peut aussi spécifier le nombre de chiffres qui doivent apparaitre après le point décimal.

```
$ ./gv.rb lister --court
1 [26.65$]: Chianti Classico 2015, Volpaia
2 [26.65$]: Chianti Classico 2014, Volpaia
4 [16.50$]: Alsace 2016, Pfaff
5 [18.50$]: Cotes de Provence 2017, Roseline

$ ./gv.rb lister --format="%3I: [%-6T - %4.0P] = '%-10N'"
  1: [rouge  -   27] = 'Volpaia   '
  2: [rouge  -   27] = 'Volpaia   '
  4: [blanc  -   16] = 'Pfaff     '
  5: [rose   -   18] = 'Roseline  '
```

Figure 1: Exemples d'exécution de la commande «`lister --format="..."`».

L'affichage approprié se fait par l'intermédiaire de la méthode `Vin#to_s`, dont la Figure 2 présente la description, telle qu'indiquée dans l'en-tête de la méthode. Si aucun format n'est spécifié, le même format que dans le devoir 1 est utilisé — donc équivalent à l'option «`--long`».

```
#
# Formate un vin selon les indications specifiees par le_format.
#
# Les items de specification de format sont les memes que dans le
# devoir 1:
#   I => numero
#   D => date_achat
#   T => type
#   A => appellation
#   M => millesime
#   N => nom
#   P => prix
#   n => note
#   c => commentaire
#
# Des indications de largeur, justification, etc. peuvent aussi etre
# specifiees, par exemple, %-10T, %-.10T, etc.
#
def to_s( le_format = nil )
```

Figure 2: Description de la méthode `Vin#to_s` qui peut recevoir un format (optionnel).

`ajouter`

Il est possible d'ajouter une **série** de vins en ne spécifiant aucun argument sur la ligne de commande après `ajouter`. Dans ce cas, le/les vins est/sont lu/s à partir de `stdin`, avec (au plus) **un vin par ligne** — une ligne vide ou contenant uniquement des blancs est ignorée.

Si une erreur est rencontrée en traitant un des vins — par ex., millésime ou prix invalide —, alors la commande n'a aucun effet, c'est-à-dire, **aucun** des vins indiqués n'est ajouté. La Figure 3 donne des exemples d'exécution — voir les tests pour plus de détails.

```
$ ./gv.rb lister --court
1 [26.65$]: Chianti Classico 2015, Volpaia
2 [26.65$]: Chianti Classico 2014, Volpaia
4 [16.50$]: Alsace 2016, Pfaff
5 [18.50$]: Cotes de Provence 2017, Roseline

# Aucun vin n'est specifie, donc aucun effet.
$ ./gv.rb ajouter

^D

$ ./gv.rb lister --court
1 [26.65$]: Chianti Classico 2015, Volpaia
2 [26.65$]: Chianti Classico 2014, Volpaia
4 [16.50$]: Alsace 2016, Pfaff
5 [18.50$]: Cotes de Provence 2017, Roseline

# Deux vins sont specifies, sur deux lignes.
$ cat data.txt
'Beaujolais Villages'    2016    "G. Duboeuf"    22.00

    "Pays d'Oc" 2016 Listel  20.00

$ cat data.txt | ./gv.rb ajouter

$ ./gv.rb lister --court
1 [26.65$]: Chianti Classico 2015, Volpaia
2 [26.65$]: Chianti Classico 2014, Volpaia
4 [16.50$]: Alsace 2016, Pfaff
5 [18.50$]: Cotes de Provence 2017, Roseline
6 [22.00$]: Beaujolais Villages 2016, G. Duboeuf
7 [20.00$]: Pays d'Oc 2016, Listel
```

Figure 3: Exemples d'exécution de la commande `ajouter` sans argument sur la ligne de commande, donc avec les vins spécifiés via `stdin`. Note : `^D` = Ctrl-D = Fin du flux d'entrée.

`supprimer`

Comme pour `ajouter`, il est possible de supprimer une série de vins en ne spécifiant aucun argument sur la ligne de commande après `supprimer`, les numéros de vins étant spécifiés via `stdin`.

Si une erreur est rencontrée en traitant une des suppressions — par ex., numéro inexistant ou vin déjà noté —, alors la commande n'a aucun effet, c'est-à-dire **aucun** des vins n'est supprimé. La Figure 4 donne des exemples d'exécution — voir les tests pour plus de détails.

```
$ ./gv.rb lister --long
1 [rouge - 26.65$]: Chianti Classico 2015, Volpaia (10/06/18) => 4 {Fonce, dense, opaqu
2 [rouge - 26.65$]: Chianti Classico 2014, Volpaia (10/06/18) => {}
4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) =>  {}
5 [rose - 18.50$]: Cotes de Provence 2017, Roseline (03/07/18) => 3 {Frais, leger.}
6 [rouge - 22.00$]: Beaujolais Villages 2016, G. Duboeuf (22/10/18) => {}
7 [rouge - 20.00$]: Pays d'Oc 2016, Listel (22/10/18) =>  {}


$ ./gv.rb supprimer
   2   4

6
^D

$ ./gv.rb lister --long
1 [rouge - 26.65$]: Chianti Classico 2015, Volpaia (10/06/18) => 4 {Fonce, dense, opaqu
5 [rose - 18.50$]: Cotes de Provence 2017, Roseline (03/07/18) => 3 {Frais, leger.}
7 [rouge - 20.00$]: Pays d'Oc 2016, Listel (22/10/18) =>  {}
```

Figure 4: Exemples d'exécution de la commande `supprimer` sans argument sur la ligne de commande, donc avec les vins à supprimer spécifiés via `stdin`. Note : `^D` = Ctrl-D = Fin du flux d'entrée.

3 Ce qui vous est fourni

Les fichiers fournis sont les suivants, ceux soulignés étant des fichiers que vous devrez modifier/compléter :

1. gv.rb : Le script `Ruby` — programme principal et méthodes auxiliaires — à compléter.
2. vin.rb : Un fichier définissant une classe pour des objets `Vin`, avec diverses méthodes publiques d'instance à compléter, notamment les deux méthodes suivantes :
   - `to_s` : Retourne une chaine représentant un vin selon le format spécifié. Par défaut, le format devrait être **équivalent** au suivant :
     ```
     '%I [%T - %.2P$]: %A %M, %N (%D) => %n {%c}'
     ```
   - `<=>` : Compare deux vins de façon à les ordonner, et ce en fonction de divers comparateurs associés aux champs des vins, par exemple :
     ```
     date = Time.now.to_date
     v1 = Vin.new(10, date, :rouge, "Chianti", 2011, "Fontodi", 20.99)
     v2 = Vin.new(11, date, :rouge, "Chianti Classico", 2012, "Fontodi", 19.99)

     Vin.comparateurs = [:appellation, :millesime]
     assert v1 < v2

     Vin.comparateurs = [:prix, :nom]
     assert v1 > v2

     Vin.comparateurs = [:nom, :prix]
     assert v1 > v2

     Vin.comparateurs = [:type, :nom, :date_achat]
     assert v1 == v2
     ```
     Voir les tests pour des exemples additionnels.
3. `vin-texte.rb` : Un fichier pour un module `VinTexte` qui spécifie les détails du format, textuel, pour la base de données des vins.
4. motifs.rb : Un fichier pour un module `Motifs` définissant divers motifs (expressions régulières) à compléter pour identifier les divers attributs d'un vin, utilisés lors d'un ajout (direct, i.e., sur la ligne de commande) ou indirect (via `stdin`) : voir plus bas.
5. Un répertoire `Tests` contenant divers fichiers définissant des **tests** — tests **unitaires** pour `Vin` et `Motifs`, tests **d'acceptation** pour le script `gv.rb`.

   Les tests d'acceptation sont un *sur-ensemble* de ceux du devoir 1 = tous les tests (inchangés) du devoir 1 + tests additionnels pour les nouvelles fonctionnalités.

6. Un fichier `makefile` définissant diverses cibles :

   - `make` : Exécute la cible par défaut, spécifiée par la variable `WIP` — *W*ork-*I*n-*P*rogress.

     Initialement, cette cible par défaut est `wip_test`, avec le test pour la classe `Vin` — `WIP=wip_test` et `ITEM_A_TESTER=vin`.
   - `make test_vin` et `make test_motifs` : Tests **unitaires** pour les méthodes de la classe `Vin` et pour les motifs (expressions régulières) du module `Motifs`.
   - `make wip_ex` : Exécute `gv.rb` avec l'exemple pour la commande indiquée par la variable `ITEM_A_TESTER` — dont la valeur initiale est `lister`.

     Les cibles suivantes sont aussi définies : `ex_ajouter`, `ex_lister`, `ex_noter`, `ex_supprimer`, `ex_selectionner`, `ex_trier`. D'autres cibles avec ces mêmes noms mais suffixés par un «`+`» pour des exemples plus avancés sont aussi définies : `ex_{ajouter,lister,selectionner,trier}+`.

     Vous pouvez (**devriez!**) modifier la variable `ITEM_A_TESTER` au fur et à mesure où vous développez une nouvelle commande. De cette façon, il vous suffira d'exécuter «`make`» pour lancer l'exécution de l'exemple.

     Rappel : Ces cibles représentent des **exemples** d'exécution, qui ont pour but de vous aider lors de l'écriture initiale du script et *ne constituent pas des tests* — **il n'y a pas de vérification des résultats obtenus par rapport aux résultats attendus**.
   - `make wip_test` : Exécute les tests pour l'item indiqué par `ITEM_A_TESTER` — classe `Vin`, module `Motifs` ou commande du script `gv.rb`.

     Trois **niveaux** de tests sont définis :
     - `base`
     - `intermediaire`
     - `avance`

     C'est la variable `NIVEAU` qui spécifie le niveau désiré. Tous les niveaux sont testés si la valeur de cette variable est «`tous`» ou si cette variable n'est pas définie.
   - Vous pouvez désactiver — **temporairement!** — un test spécifique, dans un des fichiers `Tests/*_test.rb`, en changeant l'identificateur «`it_`» par «`_it_`».

     Vous pouvez aussi faire de même pour un groupe de tests en changeant l'identificateur «`describe`» par «`_describe`».
   - `make test` : Exécute l'ensemble des tests, donc pour toutes les commandes et pour le niveau par défaut spécifié dans le `makefile`.

     On peut aussi exécuter tous les tests pour un niveau spécifique, par exemple, `intermediaire`, comme suit :

     ```
     make test NIVEAU=intermediaire
     ```

     Ou encore exécuter un test pour une commande et un niveau spécifiques :

     ```
     make test_lister NIVEAU=base
     ```
   - `make remise` : Remet la copie électronique du travail — **voir plus bas**.

4 Ce que vous devez remettre

La remise — electronique & papier — doit se faire au plus tard jeudi 22 novembre, 12h00.

Remise électronique

Vous devez exécuter la commande suivante sur `java.labunix.uqam.ca`, **à partir du répertoire** `GestionVinsRuby` (qui contient le script `gv.rb`, les fichiers `vin.rb` et `motifs.rb`, le répertoire `.git`, etc.) :

```
make remise
```

Important : Auparavant, **vous devez modifier la variable** `CODES_PERMANENTS` dans le `makefile` pour indiquer votre/vos code/s permanent/s (seul/équipe).

Remise papier

Vous devez remettre, **sous forme papier**, le listing de votre fichier `gv.rb` et des fichiers `vin.rb` et `motifs.rb`. Votre document papier doit être remis dans la boite de remise des travaux du secrétariat du département d'informatique.

Lors de la remise de votre listing, utilisez **la page avec grille de correction** disponible à la fin du présent document.

5 Remarques sur la correction

- Si le fichier remis contient des erreurs de syntaxe, alors vous perdrez une très grande (la majeure!) partie des points — voir la dernière page pour la grille de correction.
- J'exécuterai votre script sur la machine `java.labunix.uqam.ca`. Si vous développez votre script sur votre machine personnelle, assurez-vous qu'il fonctionne correctement sur `java` avant de le remettre. **S'il ne fonctionne pas sur `java`, tant pis!**
- J'effectuerai la vérification du bon fonctionnement *avec (possiblement) d'autres tests* que ceux qui vous sont fournis — donc avec des **tests privés additionnels.**
- Une partie des points sera accordée pour la *qualité* et le style de votre code : présentation, clarté et simplicité, structure, choix des identificateurs, mise en page, ainsi que **respect des règles de style de Ruby**. Deux principes de base à considérer : **KISS** (= simplicité) et **DRY** (= pas de code dupliqué).[2]
- Une partie des points sera aussi accordée pour l'utilisation de `git`![3]

  Donc, utilisez `git` pour gérer l'évolution de votre projet et éviter sa «régression». Notamment, lorsque vous avez réussi à faire fonctionner une partie du code — par ex., lorsque l'exemple ou les tests d'une commande s'exécutent avec succès — faites un `commit`! Et j'examinerai l'historique des *commits* pour m'assurer que *vous avez utilisé* `git` *de façon appropriée* — donc il n'y a pas un seul «gros» `commit` fait juste avant la remise finale!
- Comme dans le devoir 1, plusieurs des tests **intermédiaires** ou **avancés** vérifient des cas d'erreurs — et ils sont nombreux!

  Pour plus de détails sur les messages d'erreur qui doivent être générés — sur `stderr` et non sur `stdout` — voir les tests indiqués avec «`:intermediaire`» et contenant des appels à la méthode `genere_erreur`.[4]

  Note importante : **N'utilisez pas d'accents dans vos messages,** car cela peut créer des problèmes de portabilité.
- Pour respecter les principes KISS et DRY, **n'hésitez pas à introduire des fonctions auxiliaires** — un exemple : une méthode qui retourne l'enregistrement (le vin) associé à un numéro de vin!
- Profitez du fait que de nombreux tests sont disponibles pour effectuer du **refactoring**, i.e., pour modifier votre code — lorsqu'il fonctionne — de façon à améliorer sa qualité.

---

[2] De façon informelle = Approche «WTF/min» = plus je mets de rouge parce que je ne comprends pas le code ou parce que le code est mal écrit, plus la note est faible ☹

[3] Pour des informations sur `git` : `http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Liens` `http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Materiel/git.pdf`

[4] Comme dans le devoir 1, la partie entre les barres obliques spécifie une expression régulière qui doit être matchée par le message d'erreur généré sur `stderr`.

6 Indices et suggestions

- Pour lire des lignes sur le flux d'entrée standard, il suffit d'utiliser `readlines` — avec ou sans le préfixe «`STDIN.`». **N'utilisez pas** `ARGF` (inapproprié ici!) et **n'utilisez pas** `IO#tty?` (le programme devrait fonctionner même si une redirection du flux d'entrée a été indiquée).

- Les lignes retournées par `readlines` contiennent le saut de ligne (`\n`). Pour supprimer ce caractère, on utilise `chomp` ou `chomp!` :

```
>> a = "abc\n"
=> "abc\n"
>> a.chomp
=> "abc"
>> a = "abc\n"
=> "abc\n"
```

```
>> a = "abc\n\n"
=> "abc\n\n"
>> a.chomp!.chomp!
=> "abc"
>> a
=> "abc"
```

- Pour supprimer tous les blancs en début ou fin de chaine — y compris le saut de ligne — on utilise `strip` ou `strip!` :

```
>> b = "  def   ghi   "
=> "  def   ghi   "
>> b.strip
=> "def   ghi"
>> b
=> "  def   ghi   "
```

```
>> b.strip!
=> "def   ghi"
>> b
=> "def   ghi"

>> "  xyz 000   \n".strip
=> "xyz 000"
```

- Alors que «`+`» concatène deux tableaux pour en produire un nouveau — i.e., crée un nouveau tableau avec les éléments des deux tableaux — «`-`» retourne un nouveau tableau qui contient les éléments du premier tableau **qui ne sont pas dans le deuxième** :

```
>> ["abc", "def"] + ["abc", "xyz"]
=> ["abc", "def", "abc", "xyz"]

>> ["abc", "def", "def", "abc", "xyz", "ghi" ] - ["def", "xyz"]
=> ["abc", "abc", "ghi" ]
```

- La méthode `Enumerable#flat_map` est semblable à la méthode `map`, à la différence que si les résultats produits par le bloc sont des tableaux, ils sont «aplatis» :

```
>> "abc def".split
=> ["abc", "def"]

>> ["abc def", "ghi jkl"].map(&:split)
=> [["abc", "def"], ["ghi" , "jkl"]]

>> ["abc def", "ghi jkl"].flat_map(&:split)
=> ["abc", "def", "ghi" , "jkl"]
```

- Lors d'une opération de *pattern-matching*, on peut utiliser une variable comme motif (complet ou partiel) — donc les barres obliques permettent l'interpolation de chaines. Et on peut ignorer la casse en ajoutant le suffixe «`i`» après la barre oblique finale :

```
motif = "..."

/...#{motif}.../i =~ chaine_a_matcher
```

- Il faut éviter les effets de bord dans les gardes (i.e., une garde ne doit rien modifier) :

```
puts x if x = ARGV.shift    # NON!
```

  À la limite, on peut accepter une affectation simple **au début** d'une instruction `if` — parce que l'effet de bord est alors bien visible **dès le début de l'instruction** :

```
if x = ARGV.shift
  puts x
end
```

- On utilise une instruction avec garde seulement **si le cas complémentaire n'a pas besoin d'être traité** ou si on peut retourner un résultat (ou signaler une erreur) immédiatement.

  Donc, le segment de code qui suit n'est pas approprié (NON!) :

```
instruction1 if condition
instruction2 unless condition  # NON!
```

  Dans un tel cas, on utilise plutot une instruction `if` :

```
if condition
  instruction1
else
  instruction2
end
```

- On peut chainer plusieurs méthodes si l'instruction est relativement courte :

```
res = a.select { |x| ... }.join  # OK: Instruction courte!
```

  Toutefois, si l'instruction est longue, alors on met les appels sur plusieurs lignes avec **un appel** de méthode par ligne — plus facile de lire le code, de le modifier, d'ajouter des appels à des méthodes additionnelles, etc. :

```
# Forme preferable si plusieurs appels.
res = a.select { |x| ... }
       .map { |x| ... }
       .sort
       .join
```

- Pour obtenir plusieurs éléments d'un tableau, dont «`ARGV`», on peut utiliser `shift` avec un argument numérique :

```
>> a = [10, 20, 30, 40]
=> [10, 20, 30, 40]
>> x, y = a.shift(2)
=> [10, 20]
>> x
=> 10
>> y
=> 20
>> a
=> [30, 40]
```

- **N'hésitez pas à introduire des méthodes auxiliaires** — donc d'autres méthodes en plus de celles déjà partiellement définies!

Travail remis à Guy Tremblay

INF600A-20 : Devoir #2

Remise électronique (`oto`) : jeudi 22 novembre, 12h00

Remise listing papier : jeudi 22 novembre, 12h00

Ne pas mettre dans une enveloppe

Nom	
Prénom	
Code permanent	
Nom	
Prénom	
Code permanent	

	`base`	`intermediaire`	`avance`
Classe `Vin`	/10	/ 13	
init	/4	/ 0	
lister	/2	/ 11	
ajouter	/3	/ 10	/ 5
supprimer	/1	/ 7	/ 3
noter	/1	/ 6	
selectionner	/4	/ 6	
trier	/4	/ 7	/ 2
Module `Motifs`	/0	/ 18	

Tests pour `Vin` (`vin_test.rb: :base/:intermediaire`)		/10
Tests des commandes `:base`		/30
Tests des commandes `:intermediaire`		/15
Tests pour `Motifs` (`motifs_test.rb`)		/5
Utilisation du **style fonctionnel**		/5
Qualité générale du code (KISS, DRY, **style Ruby**)		/5
Utilisation appropriée de `git`		/5
Tests `:avance` (**Bonus**)		/5
Total		/75
Note globale		**/ 10**