Devoir #2 — INF5171 (gr. 20)

Automne 2017

Date de remise: Mardi, 21 novembre, 13h30. Un travail remis *après* l'heure indiquée sera **en retard** ⇒ pénalité de 10 % par jour, partiel ou complet.
Aucun travail ne sera accepté après **Jeudi, 23 novembre, 16h00**.

Mise en oeuvre et utilisation de variables synchronisées en Ruby

Le but de ce travail est de vous familiariser avec l'utilisation et la mise en oeuvre de moniteurs utilisant des verrous et des variables de conditions. Le but est aussi de vous familiariser avec d'autres mécanismes de synchronisation que ceux vus en cours.

Plus spécifiquement, dans ce devoir, vous devrez mettre en oeuvre en Ruby un type `SVar` modélisant des «**variables synchonisées**» — qu'on peut aussi appeler «variables de synchronisation».

Le comportement de ces variables est inspiré de celui des futures [CS89] (langage Future Lisp), des I-Structures [ANP87], M-Structures [BNA91] et L-Structures [Hel89] (langage Id), ainsi que des `CompletableFuture` (Java 8 [UFM14, Ora17]) et des `promise` et `atom` (Clojure [But14]),

Dans le cadre du devoir, il s'agira de mettre en oeuvre une version simple/simplifiée de ces constructions, dans la mesure où le traitement des erreurs et exceptions ne sera pas abordé, non plus que les alarmes et délais d'expiration — les *time out*.

Remarque : Je vous suggère fortement de ne pas vous y prendre à la dernière minute pour faire ce devoir. Bien que la quantité de code à écrire ne soit pas tellement grande, ce n'est jamais «facile» ou «évident» d'écrire du code avec des constructions concurrentes de bas niveau — verrous, variables de condition — et, surtout, de déboguer ce type de code — notamment, en cas de présence de *deadlocks*, fréquents dans ce genre de programmes ☹
(Et si vous rencontrez un tel cas de *deadlock*, voir la cible `debug` dans le fichier `Rakefile`!)

Remarque : La séance de laboratoire du jeudi 16 novembre sera consacrée au Devoir #2 — donc c'est moi qui serai présent au labo pour répondre à vos questions, vous aider à deboguer, etc.

Qu'est-ce qu'une variable synchronisée?

La notion de variable synchronisée a été introduite initialement — sous la forme de *I-structures* — par Arvind, Nikhil et Pingali dans le cadre d'un langage appelé Id [ANP87]. Une telle variable permet la synchronisation implicite entre le producteur de la valeur de la variable — en Id, ce producteur est unique car il s'agit d'un langage **purement fonctionnel** — et les consommateurs du contenu de cette variable. Initialement, la variable est vide, et ce jusqu'à ce que le producteur, qui s'exécute en parallèle, affecte une valeur à la variable. Si un consommateur tente d'accéder au contenu de la variable alors qu'elle est vide, il est bloqué, et ce jusqu'à ce que la valeur soit définie par le producteur, ce qui a alors pour effet de débloquer le consommateur, qui obtient la valeur et qui poursuit son exécution. Une variable synchronisée ressemble donc à un future, à la différence qu'avec une variable synchronisée l'allocation de la variable et sa définition sont des étapes séparées — d'où le terme utilisé, en Java 8, de `CompletableFuture` : on crée un future définie de façon partielle, puis on le *complète*.

Toujours dans le langage Id, une forme de variable synchronisée permettant les écritures multiples a aussi été introduite, sous le nom de *M-structures* [BNA91]. Avec une telle variable, l'atomicité de la mise à jour est assurée par le fait qu'une écriture ne peut être effectuée **que si la variable est «vide»**. Une opération — appelée `take` — permet de «prendre» le contenu de la variable, i.e., d'obtenir sa valeur tout en la **vidant** de son contenu.

Une autre forme de variable synchronisée — appelée `L-structures` [Hel89] — a aussi été introduite en Id, variable dont le contenu est évaluée **de façon paresseuse**, i.e., uniquement si/lorsque son contenu devient nécessaire en cours d'exécution.

Dans ce qui suit, nous présentons une version Ruby des variables synchronisées qui combine tous ces aspects — et même plus, avec des méthodes de la classe `CompletableFuture` de Java 8 (`then`, `all/any`) et une méthode du type `atom` de Clojure (`mutate!`). Cette version Ruby est définie dans un module `SVar` et ses classes et méthodes, que vous devrez mettre en oeuvre.

Les méthodes que vous devez mettre en oeuvre

Les paragraphes qui suivent décrivent, de façon informelle, le module `SVar` et leurs principales classes et méthodes associées.

Pour plus de détails, voir le fichier `lib/svar/svar.rb`, la documentation `yard`,[1] ainsi que les divers fichiers de test — `spec/svar_*_spec.rb`.

Création d'une `SVar` avec `SVar.new`

Il existe trois (3) types de variables synchronisées, qu'on peut créer avec la méthode de fabrication `SVar.new`, où le premier argument sert à spécifier le type de variable désirée :

- `:read_only` : Variable en lecture seulement — lisible avec la méthode «`value`».
- `:write_once` : Variable pouvant être écrite *une seule fois* — écrite avec la méthode «`value=`».
- `:mutable` : Variable pouvant être lue et écrite plusieurs fois — lue (vidée)/modifiée avec les méthodes «`take`»/«`mutate!`».

Quant au deuxième argument de `SVar.new`, il permet d'indiquer la façon dont la valeur initiale de la variable est évaluée, et ce à l'aide du bloc qui suit le `new` :

- `:immediate` : Le bloc fourni sera évalué immédiatement, donc la valeur de la variable sera disponible tout de suite après l'appel à `SVar.new`.
- `:async` : Le bloc fourni sera évalué en parallèle, dans un *thread* indépendant. La valeur de la variable pourrait donc ne pas être disponible tout de suite après l'appel à `SVar.new`... ou pourrait l'être si le *thread* a complété rapidement.
- `:frozen` : Le bloc fourni sera évalué de façon **paresseuse**, c'est-à-dire, uniquement lorsque la valeur de la variable sera obtenue avec `value` — ou demandée de façon **anticipée** avec `eval`.

Lorsqu'aucun bloc n'est spécifié, alors le type doit nécessairement être `:write_once` ou `:mutable` et la variable est alors initialement vide.

Quelques autres contraintes sur les combinaisons des deux arguments à `Svar.new` sont les suivantes :

- Si le deuxième argument est spécifié, alors **un bloc doit nécessairement être présent** pour produire cette valeur initiale.
- Si le deuxième argument est absent et qu'un bloc est spécifié, alors le mode d'évaluation par défaut de la valeur initiale est `:async`.
- Lorsque ni le type de la variable, ni le mode d'initialisation ne sont spécifiés, alors il s'agit d'une variable `:write_once`, initialisée en mode `:async` si un bloc est présent, vide sinon.

[1] `http://www.labunix.uqam.ca/~tremblay_gu/INF5171/Devoirs/SVarDoc/`

Méthodes de lecture et d'écriture

Les méthodes de lecture/écriture sont toujours exécutées de façon **atomique** et indivisible, même en présence de *threads* multiples utilisant et manipulant une même variable.

Les méthodes permises dépendent du type de variable, tel qu'indiqué dans le tableau suivant, lesquelles méthodes sont décrites plus bas :

`SVar.new(...)`	`value`	`value=`	`take`	`mutate!`
`:read_only`	X			
`:write_once`	X	X		
`:mutable`	X	X	X	X

- `value`

  Obtient la valeur de la variable. Bloque si la valeur n'est pas disponible.

- `value=( v )`

  Écrit une valeur dans la variable, **sous réserve qu'elle soit vide**.

- `take`

  Prend le contenu de la variable — donc la vide (devient `empty?`) — et le retourne comme résultat. Tout comme `value`, cette méthode bloque si la valeur n'est pas disponible.

- `mutate! { |v| ... }`

  Modifie le contenu de la variable de façon **atomique**, en utilisant le bloc passé à `mutate!`. Ce bloc va recevoir un argument — le contenu courant de la variable — et son résultat sera affecté à la variable. C'est aussi ce résultat qui est produit comme résultat de l'appel à `mutate!`.

- `eval`

  Dans le cas d'une variable initialisée en mode `:frozen`, si la variable n'a pas encore été évaluée, alors son évaluation est lancée, de façon asynchrone — donc `eval` ne bloque pas, mais un appel subséquent à `value` pourrait bloquer, bien que ce nouvel appel ne lancerait pas l'évaluation puisque cela a déjà été fait.

La figure 1 présente un diagramme de transition pour une variable synchronisée de type `:mutable`.

Les diagrammes pour les variables `:read_only` et `write_once` seraient semblables, avec les différences suivantes :

- `:read_only` : Pas de transitions `value=`, `take` et `mutate!`, et pas d'état `:empty`.
- `:write_once` : Pas de transitions `take` et `mutate!` — une fois `full?`, la variable demeure toujours ainsi.

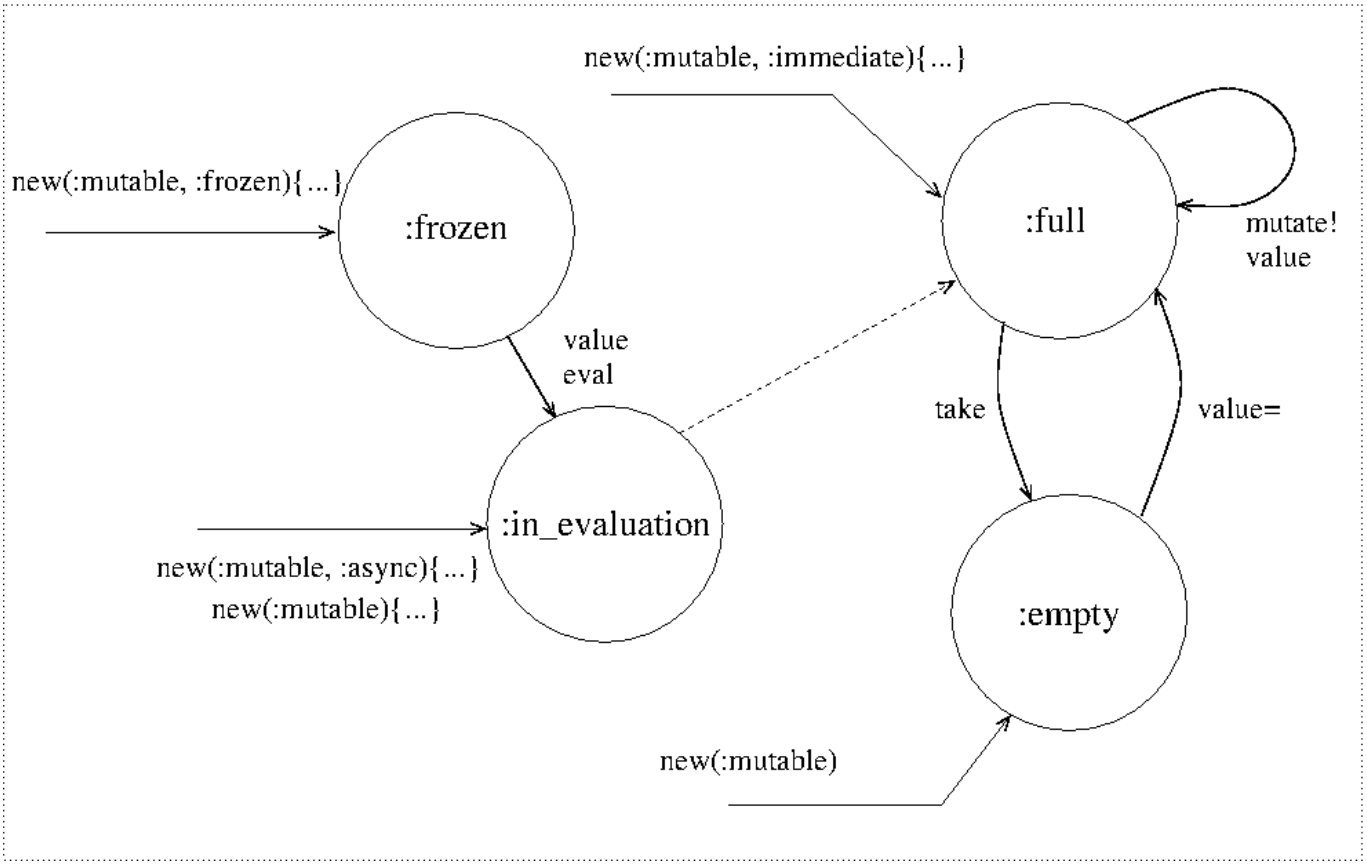

Figure 1: Diagramme de transitions pour une `SVar` de type `:mutable`, illustrant l'effet des diverses méthodes.

Méthodes pour le chainage d'opérations

Les méthodes qui suivent permettent de chainer des opérations de manipulation de variables synchronisées. Voir le fichier `bin/services.rb` pour un problème (simplifié) où l'utilisation de ces méthodes simplifie le programme résultant — programme que **vous devez compléter/écrire**.

- `SVar::Svar#then`

  Produit une nouvelle variable synchronisée suite à la réception de la valeur de la variable synchronisée.

- `SVar.all( *svars )`

  Produit une nouvelle variable synchronisée qui pourra être disponible uniquement lorsque toutes les valeurs des variables synchronisées indiquées seront disponibles. Retourne un `Array` des valeurs reçues.

- `SVar.any( *svars )`

  Produit une nouvelle variable synchronisée qui pourra être disponible dès que l'une des valeurs des variables synchronisées indiquées sera disponible. Retourne la valeur reçue.

Autres méthodes

De nombreuses autres méthodes — notamment plusieurs prédicats — sont aussi disponibles (et déjà définies) :

- `state`
- `read_only?`
- `writable?`
- `mutable?`
- `empty?`
- `full?`

Un exemple : Un programme et une méthode `fibo` pour calculer le N^e nombre de Fibonacci

Le Programme Ruby 1 présente le contenu du fichier `bin/fibo.rb`, un programme qui permet de calculer le N^e nombre de Fibonacci avec du parallélisme à granularité très fine — donc pas nécessairement efficace ☹ — et ce en utilisant des `SVars`.

Voici des exemples d'exécution de ce programme :

```
$ bin/fibo.rb 0
0
$ bin/fibo.rb 2
1
$ bin/fibo.rb 13
233
```

Ce programme illustre l'utilisation — naïve — de variables synchronisées pour calculer de façon extrêmement parallèle le N^e nombre de Fibonacci :

- L'approche utilisée est une forme de «programmation dynamique», i.e., fondée sur l'utilisation d'une structure de données intermédiaire pour éviter d'exécuter plusieurs fois un même appel récursif — donc avec mémorisation des appels déjà effectués.
- Sauf pour les cas de base, un *thread* est lancé pour chaque position de la matrice — une forme de parallélisme de résultat à granularité (très!) fine.
- Chaque case du tableau est une `SVar`, qui ne change pas une fois définie.
- Lors d'une utilisation d'un élément du tableau `sv`, l'accès est bloqué jusqu'à ce que la valeur devienne définie si elle ne l'est pas encore.

Programme Ruby 1 Un programme «*fibo.rb!*» qui utilise des variables synchronisées (en lecture seulement).

```ruby
#!/usr/bin/env ruby

#
# Programme pour calculer le N-ieme nombre de Fibonacci
#

require_relative "../lib/svar"

module Fibonacci
  Thread.abort_on_exception = true

  def self.fibo( n )
    return n if n == 0 || n == 1 # Cas de base trivial

    sv = Array.new(n+1) { SVar.new }

    # Cas de base
    sv[0].value = 0
    sv[1].value = 1

    # Cas recursif
    (2..n).each do |i|
      Thread.new { sv[i].value = sv[i-1].value + sv[i-2].value }
    end

    sv[n].value
  end
end

if __FILE__ == $0
  ##############################################################
  # Programme execute lorsque lance au niveau du shell.
  ##############################################################
  n = ARGV.empty? ? 0 : ARGV.shift.to_i

  puts Fibonacci.fibo( n )
end
```

Ce que vous devez faire

Divers fichiers vous sont fournis, sous forme d'un dépôt `git` :

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

1. Fichier `lib/svar/svar.rb` : Vous devez compléter les méthodes du module `SVar` et des classes `SVar`, `SVarWritable` et `SVarMutable` de façon à satisfaire les tests unitaires qui vous sont fournis.

2. Fichier `bin/wavefront.rb` : Vous devez compléter la méthode `Wavefront.run_par`[2] de façon à satisfaire les tests unitaires qui vous sont fournis.

3. Fichier `bin/graphe.rb` : Vous devez compléter les méthodes `NoeudPar.marquer!` et `NoeudPar.reset_marque`[3] de façon à satisfaire les tests unitaires qui vous sont fournis.

4. Fichier `bin/services.rb` : Vous devez compléter les méthodes `TraiterRequeteThread.run` et `TraiterRequeteSVar.run`[4] de façon à satisfaire les tests unitaires qui vous sont fournis.

Votre mise en oeuvre du module `SVar` ne doit utiliser que les classes et méthodes de base de Ruby, donc les classes `Thread`, `Mutex`, `ConditionVariable`, etc. — c'est-à-dire les éléments présentés dans le chapitre 6 des notes de cours. **Vous ne devez donc pas utiliser les méthodes de la bibliothèque `pruby` pour les méthodes de `SVar`!**

Par contre, dans les programmes `bin/*`, vous pouvez (et parfois devez!) utiliser les méthodes de `SVar` ainsi que celles de `pruby` (notamment pour générer des appels parallèles lors du parcours d'un graphe).

Quelle machine vous devez utiliser

Les tests pour la correction seront exécutés sur la machine `japet`, avec `jruby`, qui fournit des «vrais» *threads* parallèles.

Attention si vous testez votre programme sur une autre machine avant de le faire sur `japet` : Certains des tests sont relativement «fragiles», car ils reposent sur l'utilisation de pauses d'exécution (appels à `sleep`), donc sur le «*timing*» des *threads*. Or, le comportement de l'endormissement des *thread*s peut parfois varier d'une machine à une autre — durée plus ou moins précise d'endormissement.

[2] Méthode indiquée avec «`self.run_par`», définie dans la classe `Wavefront`.

[3] Méthodes indiquées avec «`self.marquer!`» et «`self.reset_marque`», définies dans la classe `NoeudPar`.

[4] Méthodes indiquées avec «`self.run`» dans `TraiterRequeteThread` et «`self.run`» dans `TraiterRequeteSVar`.

Ce qui vous est fourni

Plusieurs fichiers vous sont fournis, dont certains que vous devez modifier :

1. `lib/debug.rb` : Méthodes pour aider au débogage, notamment une méthode `jiggle`.
2. `lib/dbc.rb` : Méthodes pour l'approche *Design By Contract* (DBC), donc pour exprimer des pré-conditions, post-conditions, invariants et assertions.
3. `spec/*_spec.rb` : Diverses suites de tests unitaires définies à l'aide de `MiniUnit` — voir plus bas pour les cibles de test, telles qu'identifiées par «`rake -T`».
4. Quelques exemples programmes[5] qui utilisent des variables synchronisées :
   - `bin/fibo.rb`
   - `bin/wavefront.rb` — À compléter!
   - `bin/graphe.rb` — À compléter!
   - `bin/lazy_list.rb`
   - `bin/services.rb` — À compléter!
5. `Rakefile`, où les principales cibles sont les suivantes (résultats produits par «`rake -T`», mais présentés en ordre de complexité) :

```
rake svar_new        # Tests pour new et state
rake svar_value      # Tests pour value
rake svar_state      # Tests plus detailles pour state
rake svar_writable   # Tests pour les SVarWritable
rake svar_mutable    # Tests pour les SVarMutable
rake svar_frozen     # Tests pour evaluation paresseuse
rake svar_eval       # Tests pour eval
rake svar_then       # Tests pour le then
rake svar_all_any    # Tests pout all et any

rake tests           # Tous les tests unitaires

rake fibo            # Tests pour calcul du N-ieme nombre de Fibonacci
rake wavefront       # Tests pour wavefront simplifie
rake graphe          # Tests pour marquage de graphes
rake lazy_list       # Tests pour listes paresseuses
rake services        # Tests pour services Web asynchrones

rake programmes      # Tous les tests avec des programmes

rake remise          # Remise du travail avec oto
```

   C'est la variable `wip` (*Work In Progress*) qui indique la suite de tests à exécuter par défaut — donc lorsqu'on exécute simplement la commande «`rake`». Initialement, c'est la suite de tests `svar_new` qui est active.

[5] Certains ne définissent que des classes, donc à strictement parler ne sont pas des «programmes». Mais il s'agit de classes qui **utilisent** les `SVar`, donc elles n'ont pas été mises dans le répertoire `lib`.

Ce que vous devez remettre

Remise papier

Vous devez remettre un document **papier** (dans la boîte du secrétariat du département) contenant le listage du vos fichiers `lib/svar/svar.rb`, `bin/wavefront.rb`, `bin/graphe.rb` et `bin/services.rb` — **et uniquement ces fichiers!**

En fait, ne remettez que les parties de code que vous avez modifiées ou complétées — inutile de me remettre les parties que j'ai moi-même écrites et qui sont inchangées.

Pour la remise de ce document papier, vous devez utiliser **la page couverture (avec grille de correction)** disponible à la fin du présent du document.

Remise électronique

Vous devez remettre une version électronique des fichiers `lib/svar/svar.rb`, `bin/wavefront.rb`, `bin/graphe.rb` et `bin/services.rb` en exécutant la commande suivante sur `japet` :

```
$ rake remise
```

Note : Vous devez *auparavant* modifier la variable `CODES_PERMANENTS` dans le `Rakefile` pour indiquer vos codes permanents (si vous travaillez en équipe, deux personnes maximum) ou votre code permanent (si vous travaillez seul).

Remarques sur la correction

- Si les fichiers remis contiennnent des erreurs de syntaxe, alors vous perdrez une très (très!) grande partie des points — voir plus bas.
- Une partie des points sera accordée pour la *qualité* et le style de votre code : présentation, clarté et simplicité, structure, choix des identificateurs, respect des conventions Ruby,[6] etc. Par contre, même si la présence de commentaires est suggérée, je n'évaluerai pas la présence, ou l'absence, de tels commentaires.

  Deux principes de base pour la clarté du code : KISS et DRY.[7]

  Autre principe pour la correction du style : 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 ☹
- J'effectuerai possiblement la vérification du bon fonctionnement *avec d'autres tests* que ceux qui vous sont fournis — donc avec des *tests privés* additionnels.

  En gros, la répartition des points sera la suivante : compilation correcte ≈ 20 %, bons résultats sur les tests publics ≈ 60 %, bons résultats sur les tests privés ≈ 20 %.

Suggestions et indices

- La correction des travaux se fera sur la machine `japet`, avec `jruby`. Donc, vous devez absolument vous assurer que votre programme fonctionne correctement sur cette machine avant de le remettre.
- N'hésitez pas à introduire des méthodes auxiliaires — gardez votre code **DRY**, c'est-à-dire, éviter le plus possible le même code qui se répète.
- Utilisez `git` pour gérer l'évolution de votre travail et éviter sa régression, puisque le code qui vous est fourni est déjà sous forme d'un dépôt local `git`. Donc, lorsque vous avez réussi à faire fonctionner une partie de votre code, faites un `commit`!

  Plus de détails sur `git` sont disponibles via la page suivante :

  `http://www.labunix.uqam.ca/~tremblay/INF5171/Liens/`

[6] `https://github.com/styleguide/ruby`

[7] KISS = *Keep It Simple, Stupid*; DRY = *Don't Repeat Yourself.*

Références

[ANP87] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: Data structures for parallel computing. In *Graph Reduction*, pages 336–369. Springer-Verlag, LNCS-279, 1987.

[BNA91] P.S. Barth, R.S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional languages with state. CSG Memo 327, MIT, March 1991.

[But14] P. Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel.* Pragmatic Bookshelf, 2014.

[CS89] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Papers from the Second Workshop on Languages and Compilers for Parallel Computing*. University of Illinois at Urbana-Champaign, 1989.

[Hel89] S.K. Heller. Efficient lazy data-structures on a dataflow machine. Technical Report MIT/LCS/TR-438, Laboratory for Computer Science, MIT, Feb. 1989.

[Ora17] Oracle. `Class CompletableFuture<T>`. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html`, 2017.

[UFM14] R.-G. Urma, M. Fusco, and A. Mycroft. *Java 8 In Action.* Manning Publications, 2014.

Travail remis à Guy Tremblay

INF5171-20 : Devoir 2

À remettre au plus tard **Mardi, 21 novembre, 13h30**

NE PAS METTRE DANS UNE ENVELOPPE

Nom	
Prénom	
Code permanent	
Courriel	
Nom	
Prénom	
Code permanent	
Courriel	

`svar_{new,state}`	/ 5
`svar_value`	/ 5
`svar_{writable,mutable}`	/ 5
`svar_{frozen,eval}`, `lazy_list`	/ 5
`svar_{then,all_any}`	/ 5
`bin/{wavefront,graphe}.rb` `spec/{wavefront,graphe}_spec.rb`	/ 5
`bin/services.rb` `spec/services_spec.rb`	/ 5
Qualité générale du code (KISS, DRY)	/ 5
Respect du style Ruby	/ 5
Total	/ **45**
Note globale	/ **10**